Lecture Notes on Common Subexpression Elimination

15-411: Compiler Design
Frank Pfenning

Lecture 18
October 29, 2015

1 Introduction

Copy propagation allows us to have optimizations with this form:

$$\left.\begin{array}{l} l : x \leftarrow y \\ \ldots \\ l' : instr(x) \end{array}\right\} \longrightarrow \left\{\begin{array}{l} l : x \leftarrow y \\ \ldots \\ l' : instr(y) \end{array}\right.$$

It is natural to ask about transforming a similar computation on compound expressions:

$$\left.\begin{array}{l} l : x \leftarrow s_1 \oplus s_2 \\ \ldots \\ l' : instr(x) \end{array}\right\} \longrightarrow \left\{\begin{array}{l} l : x \leftarrow s_1 \oplus s_2 \\ \ldots \\ l' : instr(s_1 \oplus s_2) \end{array}\right.$$

However, this will not work most of the time. The result may not even be a valid instruction (for example, if $instr(x) = (y \leftarrow x \oplus 1)$. Even if it is, we have made our program bigger, and possibly more expensive to run. However, we can consider the *opposite*: In a situation

$$\begin{array}{lcl} l & : & x \leftarrow s_1 \oplus s_2 \\ & & \ldots \\ k & : & y \leftarrow s_1 \oplus s_2 \end{array}$$

we can replace the second computation of $s_1 \oplus s_2$ by a reference to x (under some conditions), saving a reduction computation. This is called *common subexpression elimination* (CSE).

2 Common Subexpression Elimination

The thorny issue for common subexpression elimination is determining when the optimization above is performed. Consider the following program in SSA form:

$$
\begin{array}{llll}
\mathsf{Lab1}: & \mathsf{Lab2}: & \mathsf{Lab3}: & \mathsf{Lab4}(w): \\
\quad x \leftarrow a \oplus b & \quad y \leftarrow a \oplus b & \quad z \leftarrow x \oplus b & \quad u \leftarrow a \oplus b \\
\quad \mathsf{if}\ a < b & \quad \mathsf{goto\ Lab4}(y) & \quad \mathsf{goto\ Lab2}(x) & \quad \ldots \\
\quad\quad \mathsf{then\ goto\ Lab2} & & & \\
\quad\quad \mathsf{else\ goto\ Lab3} & & &
\end{array}
$$

If we want to use CSE to replace the calculation of $a \oplus b$ in Lab4, then there appear to be two candidates: we can rewrite $u \leftarrow a \oplus b$ as $u \leftarrow x$ or $u \leftarrow y$. However, only the first of these is correct! If control flow passes through Lab3 instead of Lab2, then it will be an error to access y in Lab4.

In order to rewrite $u \leftarrow a \oplus b$ as $u \leftarrow x$, in general we need to know that x will have the right value when execution reaches line k. Being in SSA form helps us, because it lets us know that the right-hand sides will always have the same meaning if they are syntactically identical. But we also need to know x even be defined along every control flow path that takes us to Lab4.

What we would like to know is that every control flow path from the beginning of the code (that is, the beginning of the function we are compiling) to line k goes through line l. Then we can be sure that x has the right value when we reach k. This is the definition of the *dominance* relation between lines of code. We write $l \geq k$ if l dominates k and $l > k$ if it l strictly dominates k. We see how to define it in the next section; once it is defined we use it as follows:

$$
\left.\begin{array}{l}
l : x \leftarrow s_1 \oplus s_2 \\
\cdots \\
k : y \leftarrow s_1 \oplus s_2
\end{array}\right\}
\longrightarrow
\left\{\begin{array}{l}
l : x \leftarrow s_1 \oplus s_2 \\
\cdots \\
k : y \leftarrow x
\end{array}\right.
\quad \text{(provided } l > k\text{)}
$$

It was suggested in lecture that this optimization would be correct even if the binary operator is effectful. The reason is that if l dominates k then we always execute l first. If the operation does *not* raise an exception, then the use of x in k is correct. If it does raise an exception, we never reach k. So, yes, this optimization works even for binary operations that may potentially raise an exception.

3 Dominance

On general control flow graphs, dominance is an interesting relation and there are several algorithms for computing this relationship. We can cast it as a form of *forward data-flow analysis*.

Most algorithms directly operate on the control flow graph. A simple and fast algorithm that works particularly well in our simple language is described

by Cooper et al. [CHK06] which is empirically faster than the traditional Lengauer-Tarjan algorithm [LT79] (which is asymptotically faster). We will not discuss these algorithms in detail.

The approaches we are taking exploits the simplicity of our language and directly generates the dominance relationship as part of code generation. The drawback is that if your code generation is slightly different or more efficient, or if your transformation change the essential structure of the control flow graph, then you need to update the relationship. In this lecture, we consider just the basic cases.

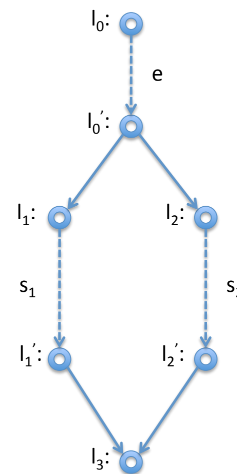
For *straight-line code* the predecessor of each line is its immediate dominator, and any preceding line is a dominator.

For conditionals, consider

$$\mathsf{if}(e, s_1, s_2)$$

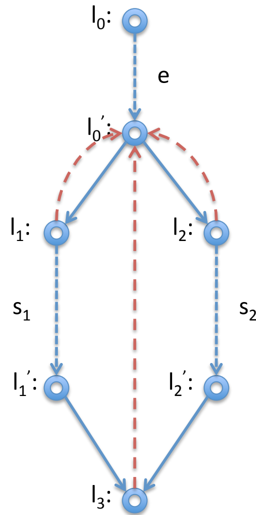
We translate this to the following code, $\check{e}$ or $\check{s}$ is the code for e and s, respectively and $\hat{e}$ is the temp through which we can refer to the result of evaluating e.

$$\begin{array}{lcl} l_0 & : & \check{e} \\ l_0' & : & \mathsf{if}\ (\hat{e}\ \mathsf{!=}\ 0)\ \mathsf{goto}\ l_1\ \mathsf{else\ goto}\ l_2 \\ l_1 & : & \check{s}_1\ ;\ l_1' : \mathsf{goto}\ l_3 \\ l_2 & : & \check{s}_2\ ;\ l_2' : \mathsf{goto}\ l_3 \\ l_3 & : & \end{array}$$

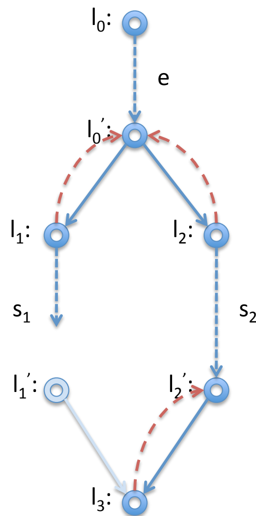


On the right is the corresponding control-flow graph. Now the immediate dominator of l_1 should be l_0' and the immediate dominator of l_2 should also be l_0'. Now for l_3 we don't know if we arrive from l_1' or from l_2'. Therefore, neither of these nodes will dominate l_3. Instead, the immediate dominator is l_0', the last node we can be sure to be traversed before we arrive at l_3'. Indicating immediate dominators with

dashed read lines, we show the result below.



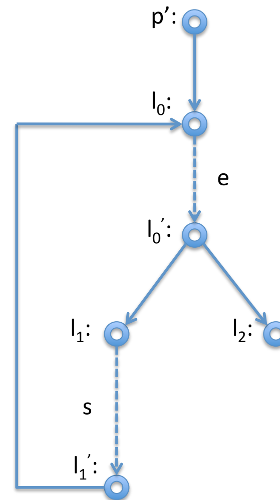
However, if it turns out, say, l_1' is not reachable, then the dominator relationship looks different. This is the case, for example, if s_1 in this example is a return statement or is known to raise an error. Then we have instead:



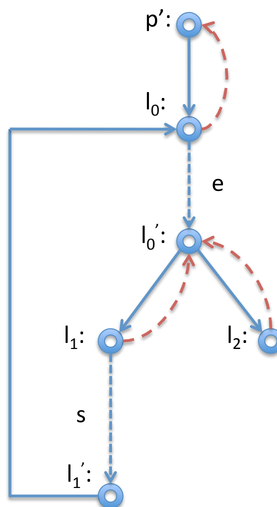
In this case, l_1' : goto l_3 is *unreachable* code and can be optimized away. Of course, the case where l_2' is unreachable is symmetric.

For loops, it is pretty easy to see that the beginning of the loop dominates all the statements in the loop. Again, considering the straightforward compilation of a while loop with the control flow graph on the right.

$$\begin{array}{lcl} l_0 & : & \check{e} \\ l_0' & : & \text{if } (\hat{e} == 0) \text{ goto } l_2 \text{ else goto } l_1 \\ l_1 & : & \check{s} \\ l_1' & : & \text{goto } l_0 \\ l_2 & : & \end{array}$$



Interesting here is mainly that the node p' just before the loop header l_0 is indeed the immediate dominator of l_0, even l_0 has l_1' as another predecessor. The definition makes this obvious: when we enter the loop we have to come through p' node, on subsequent iterations we come from l_1'. So we cannot be guaranteed to come through l_1', but we are guaranteed to come through p' on our way to l_0.



4 Implementing Common Subexpression Elimination

To implement common subexpression elimination we traverse the program, looking for definitions $l : x \leftarrow s_1 \odot s_2$. If $s_1 \odot s_2$ is already in the table, defining variable y at k, we replace l with $l : x \leftarrow y$ if k dominates l. Otherwise, we add the expression, line, and variable to the hash table.

Dominance can usually be checked quite quickly if we maintain a dominator tree, where each line has a pointer to its immediate dominator. We just follow these pointers until we either reach k (and so $k > l$) or the root of the control-flow graph (in which case k does not dominate l).

References

[CHK06] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. Technical Report TR-06-33870, Department of Computer Science, Rice University, 2006.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):115–120, July 1979.