CS 434 Meeting 24 — 5/9/06

Global Common Sub-expression Elimination

1. Now, we want to consider how to do an even better job of eliminating common sub-expressions. In particular, we want to handle control structures. So, in a piece of (meaningless) code like:

```
x = y*z;
m := z/n;
while ( y*z > 0 ) {
    if ( z/n > 1 ) {
       z = y*z;
    } else {
       z = y*z - 1;
   }
    m = z/n;
}
```

   we would like to be smart enough to realize that the boolean of the if is not a redundant common sub-expression (it will have been precomputed on the first iteration but not on later iterations), but that the boolean of the if statement is redundant and that y*z is redundant in the assignments within the if statement.

   Note, as I mentioned earlier, "global" in compiler-optmization really means one-procedure-at-a-time.

2. The first step in the process of recognizing common sub-expressions globally is (somewhat surprisingly) a simplification of the technique for basic blocks.

   We begin by scanning the code of the procedure being processed to identify *textually* equivalent expressions. That is, we ignore whether the actual values of the expressions we identify might be different because they reference variables whose values have changed.

   This is not sufficient to identify CSE's, but it serves as an important first step.

3. Next, we need to determine whether the value of each distinct expression that appears in the program will be *available* whenever program execution reaches each point in the program where the expression appears. This will be the case if we can be sure that another copy of the expression will be evaluated on every path from the start of execution to the evaluation of the expression in question and none of the variables used in the expression are changed after the last evaluation of a copy of the expression.

4. To determine which expressions are available at each program point, we will associate a variable, AVAIL(p), with each program point. The value of AVAIL(p) can be any subset of the distinct expressions found in the procedure being processed. Our goal is to specify the equations relating the values of the AVAIL(p) variables in such a way that a solution to the equations will assign to each AVAIL(p) variable a conservative approximation to the set of expressions actually available at that program point.

   Representing such a set at compile-time can be fairly easy. Assuming we have made a prepass over the procedure identifying textually equivalent expressions, we can just use a counter to assign small integer "name" to the expressions that appear in the procedure. Then, our set of expressions can be represented as a set of small integers (using a bit-vector).

5. The notion of a "point in the program" will depend on how we represent our code internally.

   - Most "real" compilers translate the source program into assembly-language-like statements. Segments of these statements that involve no branches or branch targets are grouped into "basic blocks". The branches possible in the program are then represented by treating the basic blocks as the nodes of a graph called the *control flow graph* in which each edge represents a possible branch.

     In this world, the program points that are interesting are usually the beginnings and ends of basic blocks.

- To keep our discussion closer to the internal form you have been using in your Woolite compilers, we can work with a notion of program points defined in terms of the structure of our abstract syntax trees.

  Basically, we want to imagine a program point between any two statements or boolean expressions used as conditions in the program. The program we are considering as an example is shown below with the program points identified with names like $< p_3 >$.

$< p_{init} >$ x := y*z;
$< p_0 >$ m := z/n;
$< p_1 >$ **while** $< p_2 >$ (y*z > 0) {
  $< p_3 >$ **if** (z/n > 1) {
    $< p_4 >$ z := y*z $< p_6 >$
  } **else** {
    $< p_5 >$ z := y*z - 1 $< p_7 >$
  }
  $< p_8 >$ m := z/n $< p_9 >$
} $< p_{10} >$

6. Given an expression α that appears at several places in a program, to deterimine which (if any) of the evaluations of α are redundant we need to examine how the flow of control through the program relates each occurence of α to:

   - other statements in the program where α is evaluated.
   - other statements in the program where the values of variables used in α may be changed.

   We say that α is "generated" wherever it is evaluated and "killed" by statements that may change variables used in the expression.

7. If you are reading carefully, you will notice that I am trying to be very careful about the use of the word "may". In particular, above I said "statements that may change" rather than "statements that change".

   When we see a statement like

   x = β

   when doing program analysis, we know that a value will be assigned to x, but we can't be sure that it will be different from x's old value. So, assuming this statement changes x would be wrong. We can only say it may change x. If it turns out it doesn't, we may assume two equivalent expressions are not CSE's when they really are.

   This is another example of a "conservative" approximation.

8. One advantage of my approach (i.e. working with the abstract syntax tree rather than with basic blocks in a control flow graph) is that the specification of the equations that determine the values of the AVAIL(p) variables is tied to the syntax of the language. For each statement type, we give a rule for generating equations involving the program points in and around the statement.

9. To simplify the equations a bit, we will assume that for each variable, x, in the procedure we pre-compute the set KILL(x) of expressions that appear in the procedure and reference the value of x. This is the set of expressions that would be killed by an assignment to x.

assignment statements Given an assignment of the form

$$< p_1 > \text{x} = \text{exp} < p_2 >$$

where p_1 is the program point just before the assignment and p_2 is the point just after the assignment it is clear that

$$AVAIL(p_2) = (AVAIL(p_1) + \{\text{sub-expression of exp}\} - KILL(x))$$

if statement Given an if statement of the form:

$< p_0 >$ **if** (exp) {
  $< p_1 > stmt_1 < p_3 >$
} **else** {
  $< p_2 > stmt_2 < p_4 >$
} $< p_5 >$

$$AVAIL(p_5) = AVAIL(p_3) \cap AVAIL(p_4)$$

$$AVAIL(p_1) = AVAIL(p_2) = AVAIL(p_0) + \{\text{expressions appearing in exp}\}$$

while loop Given a while loop of the form:

$$< p_0 > \textbf{while} < p_1 > (\text{ exp }) \{ \\ < p_2 > stmt < p_3 > \\ \} < p_4 >$$

$$AVAIL(p_1) = (AVAIL(p_0) \cap AVAIL(p_3))$$

$$AVAIL(p_2) = AVAIL(p_4) = AVAIL(p_1) + \{\text{expressions appearing in exp}\}$$

10. We can solve the equations for AVAIL (and for many other similar problems that arise in global optimization) by an iterative technique.

    (a) Start by setting all the $AVAIL(p)$ sets to the empty set.

    (b) execute all the "equations" as assignment statements.

    (c) If any of the AVAIL sets changed when all the equations were executed, do it again.

11. I'd like to quickly show an example of how these techniques can be applied to a simple sample program. I will use the program point names included in the annotated version of our sample program shown above.

12. There are only two expressions that appear more than once in this example, y*z and z/n. So, we need only consider these expressions (it would make sense to ignore expressions that only appear once in a real compiler too).

13. The KILL sets associated with the variables that may be changed by assignments in the fragment are KILL(x) = $\emptyset$, KILL(z) = {y*z,z/n} and KILL(m) = $\emptyset$.

14. The equations generated are then:

$$AVAIL(p_0) = \emptyset + \{y * z\} - \emptyset$$
$$AVAIL(p_1) = AVAIL(p_0) + \{z/n\} - \emptyset$$
$$AVAIL(p_2) = AVAIL(p_1) \cap AVAIL(p_9)$$
$$AVAIL(p_3) = AVAIL(p_{10}) = AVAIL(p_2) + \{y * z\}$$
$$AVAIL(p_4) = AVAIL(p_5) = AVAIL(p_3) + \{z/n\}$$
$$AVAIL(p_6) = AVAIL(p_4) + \{y * z\} - \{y * z, z/n\}$$
$$AVAIL(p_7) = AVAIL(p_5) + \{y * z\} - \{y * z, z/n\}$$
$$AVAIL(p_8) = AVAIL(p_6) \cap AVAIL(p_7)$$
$$AVAIL(p_9) = AVAIL(p_8) + \{z/n\} - \emptyset$$

15. Repeatedly applying these equations as assignments we obtain:

p_0	p_1	p_2	p_3, p_{10}	p_4 ,p_5	p_6 ,p_7 ,p_8	p_9
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
{ y*z }	{ y*z, z/n }	$\emptyset$	{ y*z }	{ y*z }	$\emptyset$	{z/n}
{ y*z }	{ y*z, z/n }	{z/n}	{ y*z, z/n }	{ y*z, z/n }	$\emptyset$	{z/n}
{ y*z }	{ y*z, z/n }	{z/n}	{ y*z, z/n }	{ y*z, z/n }	$\emptyset$	{z/n}

(to keep things readable, I have merged variables which clearly must have equal values)

16. From these results, we can see that the evaluation of z/n in the boolean of the if statement and the instances of y*z in the branches of the if statement are redundant.