CS 434 Meeting 34— 5/6/02

Announcements

1. Make sure you submit a complete code generation phase before moving on.

2. You should probably move on to the parsing phase (although the other alternative would probably be more fun).

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;
while z > 0 do
  begin
    if y*z > l then
       z := m/n
    else
       z := m/n - 1;
    m := m/n
  end;
```

   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 pre-computed on the first iteration but not on later iterations), but that the m/n after the if is redundant.

   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. Given an expression α that appears at several points 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 points in the program where α is evaluated.
   - points 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.

4. If you are reading carefully, you will notice that I am beginning 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

   $$\text{x} := \beta$$

   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.

5. Given the idea of points that generate and kill a variable, we can now explain when an instance of an expression α is redundant. For us to safely assume we can avoid the evaluation of α, it must be the case that taking any path from the first step in the program to the point where α occurs, we must encounter at least one point that generates α and we may not encounter any points that kill α after the last point that generates it.

   If all these conditions are true for an expression at some point in a program, we will say that the expression is *available*.

6. Checking all paths through the program may not sound too feasible. We can effectively determine which expressions are available at each point in a program using a fairly simple example of a technique known as *data flow analysis*.

7. The trick (i.e. technique) is to associate with each program point a variable (ususally a boolean or set-valued variable). Then, we specify equations over these variables that somehow capture the property we are attempting to determine by analyzing the program. Finally, we solve the equations by iterating through successively improving approximations.

8. The phrase "program point" used in this vague description can have many interpretations.

9. In an typical optimizing compiler, before optimization occurs, a graph is constructed with one node for each basic block and an edge between 2 basic blocks if control can pass from the first to the second.

   - The edges of this graph then correspond to the program points that need to be considered (since information about which variables are used or assigned to in the basic block can be gathered by a simple scan).
   - This speeds things up since it reduces the number of equations we have to solve (i.e. it reduces the number of variables we have to solve for).
   - It is a very general approach, in that it makes no assumptions about the control structures provided by the language being compiled.

10. To keep my presentation a bit simpler, I will assume a program point exists between every pair of statements (and a few other places as you will see).

11. 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).

12. One advantage of my approach (i.e. having more program points) 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.

13. We are almost there! To simplify the equations a bit, I 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

x := exp

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

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

In the remaining cases, I will indicate where the program points I wish to talk about are by putting their names in angle brackets at the appropriate points. Thus, the assignment would become:

$< p_1 >$ x := exp $< p_2 >$

if statement Given an if statement of the form:

$< p_0 >$ **if** exp **then** $< p_1 > stmt_1 < p_3 >$
else $< p_2 > stmt_2 < p_4 >$
end $< p_5 >$

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

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