CS 434 Meeting 35— 5/8/02

Announcements

1. Is there a good time I could offer a review session on aspects of the optimization components of the project?

2. I may be a bit late for office hours today.

Global Common Sub-expression Elimination

1. 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.

2. We can do this by providing rules used to generate equations relating the values of avail(p) at different program points based on how these program points relate to the statement structure of the program.

   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 $< p_5 >$

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

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

   while loop Given a while loop of the form:

   $< p_0 >$ **while** $< p_1 >$ exp **do**
   $< p_2 > stmt < p_3 >$
   end $< p_4 >$

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

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

3. 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.

4. I'd like to quickly show an example of how these techniques can be applied to a simple sample program. To make the example work, I have annotated the program with program point names below:

$\emptyset$ x := y*z;
$< p_0 >$ m := z/n;
$< p_1 >$ **while** $< p_2 >$ y*z > 0 **do**
if z/n > l **then**
$< p_3 >$ $< p_4 >$ z := y*z $< p_6 >$
else
$< p_5 >$ z := y*z - 1 $< p_7 >$
end;
$< p_8 >$ m := z/n $< p_9 >$
end $< p_{10} >$

5. 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).

6. 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) = $\emptyset$.

7. The equations generated are then:

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

8. 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)

9. 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.

The Reaching Definitions Problem

1. To give you a sense that data flow analysis is a technique that can be applied to many problems (not just identifying available expressions), I'd like to consider one more problem related to common sub-expression elimination.

2. Recognizing which instances of a common sub-expression are redundant isn't enough to enable us to eliminate the redundant expressions. We also have to make sure that the code generated for certain instances of common sub-expressions leaves their values in places (temporaries) from which they can be retrieved when redundant instances are encountered.

   This can get tricky. In the example program, for instance, we somehow have to arrange to use the same temporary to hold z/n when evaluated before the loop and at the end of the loop.

3. One simple solution to this difficulty would be to ensure that the code generator used the same temporary for all instances of any expression that appears repeatedly in a program.

   This puts more constraints on the code generator than necessary. In the example, there is no reason to put the result of the first instance of y*z in the same location as the result of the instance used for the while loop boolean. Only the value produced for the boolean is used to eliminate evaluation of a redundant CSE.

4. So, we would like to figure out which instances of a CSE are interconnected in the sense that their results have to be left in a common location so that this location can be known when redundant instances are encountered.

5. Somewhat surprisingly, this turns into another data flow problem:

   - In this problem, we will again be computing sets of expressions, but this time instances of CSE's will be considered distinct.
   - We will only consider CSE's with at least one redundant instance.
   - Given a set of identical expressions appearing in the program, we will say that an instance which is not redundant *defines* or *generates* the value of the expression while a redundant instance *uses* the value.
   - Each "use" of a CSE depends on some subset of the instances that "define" the value. For each use, we want to determine the set of definitions on which it depends.
   - Given a program point p, an expression α and some "definition", d, for α we say that the d "reaches" p if there is some path from the start of the program to p that passes through d such that d is the last definition of α on the path.
   - The *reaching definitions* problem involves associating with each program point p the set of definitions that reach p.
   - We will again associate a KILL set with each variable and a GEN set with each expression. This time KILL will contain the set of all defining instances of all expressions referencing a given variable.

6. With all this said, the equations needed are almost identical to those for AVAIL.

   assignment statements Given an assignment of the form

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

$$REACHING(p_2) = (REACHING(p_1) + \{\text{GEN(exp)}\} - \{\text{other instances of GEN}$$

if statement Given an if statement of the form:

$$< p_0 > \textbf{if exp then} < p_1 > stmt_1 < p_3 > \textbf{else} < p_2 > stmt_2 < p_4 > \textbf{end} < p_5 >$$

$$REACHING(p_5) = REACHING(p_3) \cup REACHING(p_4)$$

$$REACHING(p_1) = REACHING(p_2) = REACHING(p_0) - \{\text{other instances of su}$$

while loop Given a while loop of the form:

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

$$REACHING(p_1) = (REACHING(p_0) \cup REACHING(p_3))$$

$$REACHING(p_2) = REACHING(p_4) = REACHING(p_1) - \{\text{other instances of su}$$