CS 434 Meeting 25 — 5/11/06

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

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

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

   if statement Given an if statement of the form:

   $$\begin{array}{l} < p_0 > \textbf{if } \text{exp } \textbf{then} < p_1 > stmt_1 < p_3 > \\ \qquad\qquad \textbf{else} \;\; < p_2 > stmt_2 < p_4 > \\ \quad \textbf{end} < p_5 > \end{array}$$

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

   $$REACHING(p_1) = REACHING(p_2) = REACHING(p_0) -$$
   $$\{\text{other instances of subexpressions of exp}\} + \{\text{GEN(exp) }\}$$

while loop Given a while loop of the form:

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

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

$$REACHING(p_2) = REACHING(p_4) = REACHING(p_1) -$$
$$\{\text{other instances of subexpressions of exp}\} + \{\text{GEN(exp) }\}$$

7. There is one interesting difference between the equations for the reaching definitions problem and those we wrote for the available expressions problem. Reaching definitions uses set union while available expressions uses intersection. This reflects the fact that determining available expressions requires "MUST" information. An expression must be evaluated on all program paths to be available. Reaching definitions, on the other hand, involves "MAY" information. A definition reaches a program point if it may be the last instance evaluated on a path to that program point.

Live "variable" analysis

1. The last step in this process is to actually assign temporaries to the groups of related instances found using the results of the "reaching definitions" analysis.

2. The standard approach to this problem is based on graph-coloring:
   - We build a graph with one node for each set of instances that must be assigned a temporary locations. This graph is called the interference graph.
   - We connect two nodes with an edge if at some point in the program the values of at least one instance occurring in each of the sets of CSE instances associated with the nodes may be needed in the future to avoid computing the value of a redundant CSE.
   - We then assign colors (actually temporary location names) to the nodes in such a way that no two connected nodes have the same color.

3. Coloring a graph is an NP-complete problem, but this doesn't seem to bother anyone. Using simple, greedy heuristics apparently works well in progress.

4. The issue I want you to think about, is how to we determine which nodes in the graph to connect. That is, how to we decide if the value produced by an instance of an expression may be used in the future.

5. Once again, we can answer the problem using the data-flow analysis approach.

6. Recall from the discussion of Reaching definitions that once we have solved the Available expressions problem we can identify instances of CSE's as uses and definitions.

7. To figure out what values must be kept in temporaries at a particular point we will solve a problem that might be called "reachable uses" (but is actually called "live variables"). Basically, for each program point we will determine the set of CSE instances that are a) uses and b) reachable from the program point through an execution path that does not include any other definition of the expression.

8. Once again, it helps to have some auxiliary functions:

   use(exp) will be the set of redundant CSE instances occurring as sub-expressions of exp

   kill(exp) will be the set of all redundant CSE instances that are textually equivalent to any non-redundant CSE instance appearing as a sub-expression of exp

9. Then, the equations for determining which instances are "LIVE" at a particular point are:

   assignment statements Given an assignment of the form

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

   $$LIVE(p_1) = LIVE(p_2) + use(exp) - kill(exp)$$

   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** < $p_5$ >
```

It must be the case that:

$$LIVE(p_0) = LIVE(p_1) + LIVE(p_2) + use(exp)$$

$$LIVE(p_3) = LIVE(p_4) = LIVE(p_5)$$

while loop Given a while loop of the form:

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

it must be the case that:

$$LIVE(p_0) = LIVE(p_1) + LIVE(p_3) + use(exp)$$

$$LIVE(p_2) = LIVE(p_0)$$

10. The values needed by two expression instances that are live at a given program point must not be assigned to the same temporaries (unless the instances are instances of the same CSE).

    So, the "liveness" information gives you what is needed to build an interference graph and do temporary (i.e. register) allocation.

11. All the data-flow problems used to do redundant CSE elimination obviously have a lot in common. There are some important differences:

    - Two of the problems (available expressions and reaching definitions) involved the flow of information in the same direction as program execution would flow. These are called "forward" analysis problems. The Live variable problem, on the other hand, is a "backward" analysis problem.
    - Two of the problems collected information about what "may" happen during execution (live variables and reaching definitions) while one (available expressions) involved things that "must" happen.

    These categories are used to partition data-flow analysis problems into four groups (of which you have seen everything but a backwards-must example).