CS 434 Meeting 22 — 5/2/06

Optimization techniques

1. Although "optimization" is the popular term for our next topic, it is traditional to start by admitting that it is an inappropriate term.

   - It is theoretically hopeless to seek the optimal translation for a given program.
   - Optimization is really about code improvement.

2. Common forms of code improvement include:

   Constant Folding Recognizing expressions whose values are compile-time computable (even when program variables are involved).

   Common Sub-expression Elimination Avoiding the re-evaluation of expressions whose values have not changed. Can be done locally and globally.

   Code motion Moving loop-invariant computations to the header of a loop.

   Reduction in Operator Strength Replacing expensive operations (typically multiplications and divisions) with cheaper ones. Locally, this refers to using shifts instead of multiplies. Globally, it involves recognizing *induction variables* in loops.

   - For example, in the loop:

```
for i := 1 to 1000 do
  begin
      . . .
      a[2*i] := ...

  end
```

     the multiplication "2*i" can be avoided by keeping a counter that is incremented by 2 each time around the loop.

   Copy Propagation If an assignment of the form x := y is found, replacing instances of x with y after the assignment make make it possible to eventually eliminate the assignment.

   Dead Code Elimination Optimizations like copy propagation may result in useless instructions (the assignments) that can be eliminated.

   Procedure inlining Replacing calls to procedures with copies of the procedure body itself.

   Register Allocation Try to avoid loads and stores of values to and from memory by keeping them in registers.

   Instruction Scheduling Ordering the instructions in the generated code to deal with hardware timing issues (memory access delays, branch delays, pipeline features).

3. Optimizations can be classified according to the extend of code considered when they are applied.

   Peephole Optimization Looks at just a short segment of output machine code.

   Local Optimization Looks at just one statement of high-level code.

   Straight line code (or basic block) optimization Looks at sequences of instructions involving no branches (in or out).

   Global Optimization Looks at an entire procedure (i.e. not really global).

   Interprocedural Optimization Really global.

Value Numbering

1. Common subexpression elimination is an important optimization technique that we will use to explore several aspects of optimization methods.

2. The goal of this optimization is to identify expressions that are guaranteed to produce identical values at runtime and arrange to only perform the associated computation once (when the first instance of the expression is encountered).

3. This involves tracking the flow of information through variables.

   - In the code:

```
x = a + b;
y = c + d;
a = e;
z = a + b;
w = b + y;
v = b + c + d;
```

The two occurences of "a+b" are not common subexpressions because the value of a may change between the evaluation of the first and second copies of the expression. On the other hand, the last two expressions, "b + y" and "b + c + d" can be identified as common subexpressions even though they are not textually identical because they are guaranteed to produce identical values.

4. We will begin by restricting our attention to CSE in straight line code. Then, once the details are understood, we can see how to deal with programs (like most real ones) that include control constructs.

5. Eliminating common sub-expressions involves two non-trivial sub-problems.

   - It isn't enough to find expressions that look identical.
     - Expressions that look identical may produce distinct results if the values of variables referenced by the two expressions change between their evaluations.
     - Expressions that don't look identical may produce identical results:

       ```
       x = 1 + y;

       a[y + 1] = a[x] + 1
       ```
   - Even if we could proceed by simply looking for textually identical expressions, it isn't immediately clear how we could find them all efficiently.

6. One scheme that can be used to identify CSEs is called *Value Numbering.*

   The goal of this scheme will be to assign a number to each expression subtree in our program in such a way that two sub-trees will be assigned the same number only when they are guaranteed to produce the same value.

7. We can get a sense of how the value numbering scheme works by considering a simpler scheme designed to solve only the second problem described above: the problem of identifying textually identical expression efficiently.

   The goal of this scheme will be to assign a number to each expression subtree in our program in such a way that two sub-trees will be assigned the same number exactly when they are textually identical.

8. We will define a recursive procedure to assign these number.

   - The base cases are expression subtrees that are references to variables (let's ignore records and arrays for a minute and assume that all variables are simple Nident nodes), and constants.
   - To handle variables, we will just keep a counter of how many variables are declared in the program. Each time a variable is declared, we will store this current value of this counter in its declaration descriptor. When numbering expressions, we will use this value if we encounter a variable. This ensures that two references to the same variable will be assigned the same "expression number" as desired.
   - Constants are a bit harder. If "7" appears in two different places in our program, we want to assign it the same "expression number" (which probably won't be 7). We can do this by keeping a hash table. Each time we encounter a constant, we can look it up. If we don't find it, we will pick a previously unused expression number, add an entry for the constant's value and expression number to the hash table, and assign this number to the constant subtree.
   - More complex expression (i.e. those using binary and unary opeators) can be handled recursively with the same kind of help from a hash table.
     - First, recursively determine the expression numbers for the operand(s) to the operator.

  - Look up a tuple composed of the operator used and the value numbers for the operands in the hash table.
  - If no match is found in the hash table, assign a previously unused expression number and add an appropriate entry to the hash table.
  - if a match is found, associate the expression number found with the expression sub-tree.

- This scheme implements a mapping from expression subtrees to "expression numbers". The implementation of the mapping is partitioned into two components:
  - For variables we use information stored in declaration descriptors.
  - For all other expression we depend on hash table lookups.

9. Unfortunately, the fact that two expressions are identical is neither necessary or sufficient evidence that they will have the same value at runtime. The problem is that the values associated with variables may change. The solution is to change the "sequence numbers" we assign to variables used as sub-expressions.

10. The goal is to ensure that two expression are assigned the same number *only if* our compiler is certain that they will have the same value at runtime.

11. We can accomplish this by changing the way we assign numbers to expressions that reference variables.

    - We will still keep each variable's (current) value number in its declaration descriptor. Unlike the scheme proposed above, however, this value may change as our algorithm progresses.
    - The first time we encounter a reference to a variable within a block, we must assign the variable a previously unused value number (and store it in its declaration descriptor).
    - When we encounter an assignment statement, we must change the number associated with the variable that is the target of the assignment. We don't, however, assign brand new value numbers to the targets of assignments. Instead, when processing the assignment

      x := E

      we will assign E's value number to x since future references to x will produce the same value as E.
    - The hash table will still be used as before.
    - We will recognize that we have found an interesting CSE when a match is found in the hash table.

12. This scheme will work fairly well for straight line code containing nothing but references to simple, local variables. In the real world, unfortunately, there are a few more complications to deal with.

    - First, consider references to array elements (assuming simple arrays of ints is bad enough!).
      - In simple code like:

        ```
        x = a[i] + z
        y = k;
        z = a[i] + z
        ```

        we certainly would expect to be able to identify "a[i] + z" as a common sub-expression and eliminate the evaluation of the second copy of the expression (just using the value of x insead).
      - Suppose instead that we have code that looks like:

        ```
        x = a[i] + z
        a[k] = k;
        z = a[i] + z
        ```

        Even though the value of k is likely to be different from the value of i, our value numbering scheme doesn't provide a way to know this for sure. It only tells us when two variables definitely have the same value, not when they definitely are different. So, we would have to be careful and assume that the second copy of ”a[i] + z” might produce a different value from the first copy.