CS 434 Meeting 23 — 5/4/06

Value Numbering (cont.)

1. Value number identifies common subexpressions, by using a table lookup mechanism to associate numbers with instances of expressions in a program in such a way that:
   - If two expressions are certain to produce the same value at run-time, then they may be assigned the same value number.
   - If two expressions may produce different values at run-time, then they are certain to be assigned different value numbers.
2. The algorithm proceeds assigning value numbers while traversing a sequence of statements that must be executed sequentially.
   - Each variable used in the statements being processed is assigned a distinct value number before processing begins (at least logically).
   - Constants and expressions using unary and binary operators are assigned value numbers using a hash table lookup. When a constant is encountered we just hash it and look it up. When an expression is encountered, first determine the value numbers of its sub-expressions and then hash the tuple composed of the operator and sub-expression value numbers.
     - If the lookup fails, assign a new value number.
3. 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 that look 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.
   - We can handle cases like this if we simply change the value number associate with "a" after any assignment that changes an element of "a". That is, we won't really make any effort to keep individual value numbers for array elements. The array variable itself will have a value number. We will treat array subscripting as a binary operation. We will combine an array's value number with the value number of the index expression and the subscripting operator and look the combination up in our hash table to determine what value number to associate with the combination.

     This value number associated with an array should clearly be changed when we process an assignment such as

     ```
     a = new int[ size ];
     ```

     or

     ```
     a = b;
     ```

     but to play it safe, we will also change it when we encounter

     ```
     a[i] = n;
     ```
   - The example

     ```
     a = b;
     ```

     points up other horrible issues. Suppose we have to deal with code that looks like:

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

     Because "a" and "b" now refer to the same array, the assignment to b[k] may change a[i]. Once again, we cannot safely treat a[i] + z as a CSE. Somehow, when we process an assignment that changes b, we have to know enough to change the value number associated with a!

We can either do much more sophisticated analysis to estimate the set of variable that may refer to the same array as a name like "b" and change all the value numbers in this set when we assign to b, or we can just update the value numbers of all array variables that refer to arrays of the same type as b.

4. The handling of array elements is a fine example of the *conservative* nature of the analysis algorithms one uses when performing optimization. Such algorithms do not give exact information, but they only make "safe" mistakes. In this case, the algorithm will often fail to identify pairs of expressions that actually are CSE's. The result will be that the code generated will be less efficient (but correct!). On the other hand, the alternative of sometimes accidentally concluding that two expressions are CSE's when they are not is unacceptable. It would result in the generation of incorrect (though efficient) code.

5. BUT WAIT! It get's worse. We still need to account for control structures and method invocations.

6. Let's put off control structures for a few more moments.

7. Finally, if we encounter an invocation, we must:

   - change any value numbers assigned to instance variable in all the classes enclosing the method whose body we are processing.
   - change any value numbers associated with array variables (unless we can determine that the arrays associated with local variable names were allocated locally rather than obtained as parameters or through instance variables, and not assigned to any instance variable name.

Flow graphs and Basic Blocks

1. The value numbering scheme does not work in situations where there are loops or conditionals:

```
x[i+1] = y;
while ( i < max ) {
   x [ i+1 ] = x[ i+1 ] + z;
   i = i+1;
}
```

   In this example, we would assign the same value number to all four instances of i+1, but the assignment statement at the end of the loop means that the instance of i+1 outside the loop will not have the same value as those inside in all cases.

```
w = 2*x + 1;
if ( x > 0 )
   { z = 2*x }
y = z + 1;
```

   In this example, 2*x + 1 and z + 1 might be common subexpressions, but we can't be sure unless we know whether or not x will be positive.

2. Examples like this make this notion of "straight line code" important enough to deserve a name. We will call sequences of staight line code "basic blocks".

3. In the real world, before optimization, a compiler usually would rewrite the program into a form in which basic blocks and the flow graph were represented explicitly.

   - Most optimizers work on an intermediate form that is quite a bit closer to assembly language than our syntax trees. In such code a basic block block is simply a sequence of statements beginning with a label that contains no branches (other than subroutine calls) or other labels.
   - To preserve control flow information, such compilers build a directed graph whose nodes are basic blocks and whose edges represent possible branches between blocks. This is called a `control flow graph`.

4. Describing a basic block in our intermediate form is a bit trickier since trees aren't quite as linear as pseudo-assembly language.

   Luckly, since we are only interested in using basic blocks to identify sequences of straight line code we can take a simpler approach.

5. The approach I want you to imagine depends on two facts:

   - Local optimization algorithms have a fairly simple structure:

     ```
     for all basic blocks do
         initialize various data structures
         for each instruction in the block do
               scan the instruction and
               optimize (if possible).
     ```

   - Most of the algorithms you have already implemented (semantic processing, code generation), process syntax tree nodes in such a way that all the nodes that belong to one basic block are processed consecutively.

6. All we need to do to apply a local optimization algorithm to basic blocks is take the code used to do a "standard" traversal of the syntax tree and figure out where to put the "initialize various data structures" steps.

7. To make this precise, here are sketches of some pieces of the optimization traversal code:

```
void optimizeStmtList( node * slist ) {
   visitlist( slist, optimizeStmt, 0);
}

void optimizeStmt( node * stmt ) {
   switch (stmt->internal.type ) {
   case Nif:
      optimizeIf( stmt );
      break;
   case Nwhile:
   ...
}

void optimizeIf( node * stmt ) {
    optimizeExpr( stmt->internal.child[0] );
    startNewBlock();
    optimizeStmtList( stmt->internal.child[1] );
    startNewBlock();
    if ( there is an else part ) {
       optimizeStmtList( stmt->internal.child[2]);
       startNewBlock();
    }
}
```

8. We can also take advantage of the fact that our goal is local optimization by working with a slightly looser definition of basic blocks.

   - At a point where control branches we can continue to propagate CSE information down one of the branches (or both if we are willing to save the state of the algorithm when we head down the first branch). We just have to start over again whenever two control paths joing.
   - This leads to the notion of an "extended basic block".
     - In low-level (assembly language like) intermediate forms, a extended basic block is a sequence of statements starting with a label (or the entry point of a procedure) that includes no other labels (but may contain branches unlike simple basic blocks).
     - Note that an extended basic block will be the union of a sequence of basic blocks.
     - In our trees, extended basic blocks can be formed by leaving out the "startNewBlock" except at points where we know labels may be placed.

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 ) {
```

```
        if ( y*z > l ) {
           z = m/n
        } else {
           z = m/n - 1;
       }
        m = m/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 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.