CS 434 Meeting 29— 4/24/02

Announcements

1. Colloquium speaker: David Detlefs on Garbage collection.

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. Giving this term a precise definition in our context is actually a bit difficult.

   - 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.
   - Describing a basic block in our intermediate form is quite a bit trickier since a) trees aren't quite as linear as pseudo-assembly language and b) we may even have control flow issues within expressions due to the short circuit evaluation of logical operators.

4. Our immediate interest in basic blocks will be that they provide a program unit larger than a single statement or expression to which we can apply optimization techniques without expending the effort to deal with control structures.

5. In the real world, however, basic blocks are also typically used by compilers that do perform global analysis.

   - A favorite intermediate form for global optimizers is the "flow graph" — a graph whose edges represent conditional and unconditional branches and whose nodes are the basic blocks of the original program.
   - The cost of global optimization techniques tends to be proportional to the number of nodes in a flow graphs. If basic blocks were replaced by nodes for each original statement, the cost of global optimization would grow.
   - To make this work, the optimizer must combine a global analyzer with a more local approach. For example, if we were doing constant folding/propagation:
     - The global analyzer would associate a set with the entry point of each basic block containing the variables whose values were known to be constant at that program point (and their values).

– The local process discussed above would then work through each blocks straight line codes starting with the globally computed set as a starting point.

6. 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. We don't have the time to do this. Luckly, since we are only interested in using basic blocks to identify sequences of straight line code we can take a simpler approach.

7. The approach I want you to use 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.

8. 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.

9. To make this precise, here are some examples:

```
void optimizeLogicalExpr( node * expr) {

optimizeExpr( expr->internal.child[0] );
startNewBlock();
optimizeExpr( expr->internal.child[1] );
startNewBlock();
}

void optimizeIf( node * stmt ) {
optimizeExpr( stmt->internal.child[0] );
startNewBlock();
optimizeStmtList( stmt->internal.child[1] );
startNewBlock();
optimizeStmtList( stmt->internal.child[2]);
startNewBlock();
}



void optimizeStmt( node * stmt ) {

    switch (stmt->internal.type ) {
    case Nif:
optimizeIf( stmt );
break;
    case Nwhile:
optimizeWhile( stmt );
break;
    ...
}


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

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

    - If you think about the value numbering example, you will realize that at a point where control branches we can continue to propagate information down 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.
    - To make this precise, here are some examples:

```
void optimizeLogicalExpr( node * expr) {

optimizeExpr( expr->internal.child[0] );
optimizeExpr( expr->internal.child[1] );
startNewBlock();
}

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

Some Implementation Details

1. When we first encounter a reference to a variable in a basic block, we need to know that it has not been previously assigned a value number.

    - Setting all variable value numbers to an unused value (like -1) initially will do the trick.

2. When we finish processing one block and want to begin another, we need to forget all the value numbers that had been assigned.

    - To avoid having to reset every variable's declaration descriptor, we can keep a list of variables that have been assigned values in the current basic block.

3. Such a list can also reduce the effort required to handle aliasing. When we process an assignment, we can just go through the list of variables to which value numbers have been assigned and unassign value number associated with possible aliases found in the list.

4. The whole point of value numbering is to avoid recalculating CSEs. To do this, we have to be able to:

    (a) know enough to leave a CSE's value in a temporary after we first compute it, and

    (b) find the temporary holding the value when we encounter the CSE again.

5. To make this easier, we will allocate operand descriptors for expressions while we are value numbering rather than while we are generating code.

   - Each time we assign a new value number, we will allocate a new operand descriptor.
   - We will store a pointer to the operand descriptor with the new value number in the hash table, and in the syntax tree node for the root of each copy of the CSE.
   - We will add "not-yet-calculated" to our existing operand descriptor types (i.e. areg, dreg, basedvar) and set this as the type of the descriptors we create.
   - When the code generator reaches an expression node, it will only generate code for the expression if the operand descriptor pointed to by the expression's root node is "not-yet-calculated". Otherwise, it will just assume the value is in the temporary described by the operand descriptor.

6. Finally, if the value of a CSE is put in a register, we have to know how long it needs to be kept so that we can reuse the register as soon as possible. We will take a very simple approach to this register/temporary allocation problem.

   - Add an extra field to the operand descriptor type to count how many copies of the associated expression (that are not subtrees of some larger CSE) were encountered.
   - The "not subtrees" part means that you don't increment a CSE's counter when you find a tree that matches it. Instead, when you find a node that represent a first instance of a new CSE, you increment the counters of all its children.
   - When generating code, each time you think about generating code for a tree that shares a given operand descriptor you will decrement the use counter. When its counter becomes 0 you can free the descriptor.

The Correctness of LR(0) parsing

1. The correctness of the LR(0) parsers we have discussed rests on the theorem:

   Theorem $[N \rightarrow \beta_1.\beta_2] \in \Delta(\pi_0, \gamma)\ iff\ [N \rightarrow \beta_1.\beta_2]$ is valid for γ.