CS 434 Meeting 27— 4/19/02

Announcements

1. Written assignment due Today

Constant Folding and Algebraic Transformations (cont.)

- Performing computations involving constants can improve code quality.
- To identify many of the important situations in which we can improve code by doing constant calculations at compile time we must be able to apply algebraic laws to program expressions.
- The tough question is how to organize the process of applying arithmetic identities to find constants and otherwise simplify expressions. Our solution will be to attempt to use arithmetic identities to transform all expressions into a canonical form that exposes constant sub-terms.
- Given the value of additive constants, a good first approximation of the desired canonical form would be:

  $$K + \alpha$$

  where K represents some constant and α represents the non-constant part.

  - To see the value of this, assume we start with an expression of the form

    $$\alpha' + \beta'$$

  - Assume we have some transformation

    canonize(α') -> $K + \alpha$.

  - By applying this transformation to both α' and β', we can rewrite out original expression as:

    $$K_1 + \alpha + K_2 + \beta$$

    and then as:

    $$(K_1 + K_2) + (\alpha + \beta)$$

    (which is now in our potential canonical form!).
- While the preceding hopefully gets the point across, $K + \alpha$ is not an ideal canonical form:
  - While additive constants are most important, multiplicative constants can also lead to opportunities for partial constant folding:
    * If we reduce α' and β' to $K_1\alpha$ and $K_2\beta$, then we can reorder the terms in the multiplication $\alpha'\beta'$ to obtain

      $$(K_1K_2)\alpha\beta$$

      and perform the constant multiplication at compile time.
- As a result, a useful canonical form is:

  $$K + M\alpha$$

  where K and M are constants and α is some expression (out of which you have hopefully sucked all the constants already).
- Expressions in canonical form could be represented as syntax trees, but the use of a specialized structure type containing components corresponding to K , M and α will make the code much, much simpler (no long chains of ->internal.child[i]).
- Given the use of such a specialized data structure, your constant folder will consist of two main functions:

- canonize(expr) — will take an expression syntax tree and return its canonical form, and
- treeify(canon) — will take an expression in canonical form and map it back to tree form (in as efficient a way as possible).

- To "fold" the constants in an expression you will say:

  treeify(canonize(expr))

- Canonize will work a lot like other expression processing functions:
  - It will consist of a large switch statement with a case for each expression node type.
  - It will typically apply itself recursively to the sub-expressions representing the operands of the root operator.
  - If an operand can be reduced to a constant, it will come back in the form "K + 0 * α".
  - If all the operands of any operator are constants, canonize can apply the operator and return a constant.
  - For most operators, if the operands are not both constant, canonize must basically just return "0 + 1 * α". The α included, however may not quite be the original expression.
    * While canonize may not have reduced all sub-expressions to constants, it may have simplified them. SO, given the input $\beta_1 \text{op} \beta_2$, canonize will return

      $0 + 1 *$ (treeify(canonize(β_1)) op treeify(canonize(β_2)))

      when it can't do more.

- Canonize may do more interesting things when the operator at the root of an expression is +, - or *. (Integer division is too messy to think about).
  - As an example, lets consider multiplication.
  - Given the input $\alpha' * \beta'$, canonize will apply itself recursively to rewrite the input in the form

    $$(K_1 + M_1 * \alpha) * (K_2 + M_2 * \beta)$$

  - If all the K's and M's are non-zero, canonize can't do much (other than resort to the result suggested above).
  - If all the K's happen to be 0, canonize can improve things by multiplying the M's together at compile time and returning:

    $$0 + (M_1 * M_2) * (\alpha * \beta)$$

  - If either M is zero, canonize can distribute the associated K over the other term to obtain something like:

    $$(K_1 * K_2) + (K_1 * M_2) * \beta$$

- To make this all work, treeify has to be smart enough to do a few things:
  - recognize that 0+x = x,
  - recognize that 1*x = x,
  - recognize that k+(-x) = k-x,
  - etc.

- To maximize the effectiveness of these transformations on address arithmetic, canonize should treat refvar nodes specially.
  - First, it should canonize the base address expression.
  - It should then add any additive constant in the canonized form to the refvar's displacement.
  - It should only re-treeify the base address after setting the canonized form's additive constant to zero.

Value Numbering

1. Another important optimization technique is common subexpression elimination. The goal of this optimization is to identify expressions that are guaranteed to produce identical values at runtime and arrange to only performed the associated computation once (when the first instance of the expression is encountered).

2. This is a more interesting problem that constant folding because it 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.

3. As a result, in a general implementation of CSE (common subexpression elimination), we have to take into account assignments to variables and control flow. 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.

4. Eliminating common sub-expressions involves two non-trivial subproblems.

   - Expressions that look identical only produce the same results if none of the values of variables referenced by the two expressions may be changed between their evaluations.
   - Even if we only looked for examples of identical expressions, we would need to find an efficient way to look for matching subtrees.
   - Expressions that don't look identical may produce identical results:

     ```
     x = 1 + y;

     a[y + 1] = a[x] + 1
     ```

5. The scheme we will use is called *Value Numbering*.

6. The idea is to traverse the expression sub-trees of a basic block assigning an identifying number to each sub-expression. The goal is to assign the same number to all expressions we can be sure will produce the same value.

7. The algorithm will be based on a form of table lookup. The table will associate expressions with their "value numbers". For each expression we encounter as we traverse the subtrees of a basic block we will.

   - Look the expression up in our table.
   - If the expression is not in out table, we will assign it the next unused value number (i.e. add one to the last one) and enter the expression/number pair in the table.
   - If an expression is already in the table, we will associate it with its already assigned value number.

8. The trick that makes this interesting is that we will use the numbers we assign to make our table lookups efficient.

   - We will actually partition our table into two parts.

   - For simple variables, since the refvar nodes in the syntax tree point directly to their declaration descriptors, we will simply store their value numbers in a new field within the descriptor.
   - The main table will be a hash table. It will be used for all expressions except references to simple variables.

9. We will account for assignments by changing value numbers appropriately.

   - In this scheme, variables used in expressions will be assigned value numbers when they are first encountered and (at least logically) be included in our table.
   - For each assignment encountered as we process a basic block, we will assign a new value number to the variable (and update our table) so that we will recognize that subsequent expressions that reference the variable may produce distinct values from earlier, syntactically identical expressions.
   - We don't 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.

10. The hash table will map a key that uniquely identifies an expression's values to the expression's value number. The key used will depend on the form of the expression.

    - For binary operators, the key will be a triple composed of the operator type (i.e. the value of the type field of the root of the expression subtree) and the value numbers of the sub-expressions.
    - For unary operators, the key will be a pair consisting of an operator number and the sub-expression's value number.
    - For constant nodes, the key pair will consist of the node type (Nconst) and the constant's value.
    - Refvar subexpressions are interesting enough to deserve separate consideration below.

11. So, to process an expression, we:

    - process its sub-expressions (receiving their value numbers as return values),
    - build the appropriate key based on the expression type and the value numbers of its sub-expressions.
    - look the key up in the hash table, and
    - make a new table entry associating the key with the next value number if no match is found, or
    - recognize that we have found a CSE if a match is found (ard then return the associated value number).