CS 434 Meeting 26— 4/17/02

Announcements

1. Written assignment due Friday

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.

Constant Folding and Algebraic Transformations

- As discussed above, performing computations involving constants can improve code quality.
- One could easily change your code generators to check to see if all the operands to an operator node were constants and in that case perform the operation at compile time.
- This simple change, however, would miss opportunities for constant folding like:

  ```
  (3+x)-1 = 2+x
  ```

  and

  ```
  4*(2*x - 1) = 8*x - 4
  ```

  To find examples like this, one has to be willing to perform transformations using rules of arithmetic like commutativity and associativity while looking for constant sub-expressions.
- Note: these transformations may have surprising effects on program results:
  - Consider the evaluation of:

    really-big-constant + variable + pretty-big-constant

    If the sum of the two constants is too big for the arithmetic type being used, combining the constants might result in an arithmetic overflow while, at run-time, the value of "variable" might have been a negative number big enough to avoid the overflow.
  - While these issues exist with integers, they are much more significant with floating point numbers (where the "laws" of arithmetic rarely hold at all!)

    A "real compiler" has to be careful about these issues. We will ignore them (from here on).
    - Even in a real compiler, these issues can usually be ignored in address arithmetic sub-expressions, and this is where we will get the most benefit from these optimizations.
- There are other identities that can be used to reduce computation time when constants are involved:
  - 1*x = x
  - 0+x = x
  - 0*x = 0

  (Not evaluating x, however, may change program behavior if the evaluation of x produced side-effects).
- Constant folding that is willing to perform algebraic transformations can be particularly productive in expressions generated to perform address arithmetic and, in this context, "additive constants" are particularly interesting.
  - A subscripted variable of the form x[i+1] might produce an address arithmetic expression of the form:

    $$\text{display-pointer} + \text{disp}(X) + 4 * (i + 1)$$

    (assuming the element of the array x occupy 4 units of memory).
  - Since many machines including an address mode that allows one to add a constant displacement to a computed address, rewriting this expression as:

    $$(\text{disp}(X) + 4) + (4 * i + \text{display-pointer})$$

    makes it possible to use the "disp(x)+4" as such a displacement.

- The tough question is how to organize the process of applying arithmetic identities to find constants and otherwise simplify expressions.