CS 701

Final Exam

Thursday, December 14, 2000

11:00 a.m. — 1:00 p.m.

1263 CSST

Instructions

Answer question #1 and any three others. (If you answer more, only the first four will count.) Point values are as indicated. Please try to make your answers neat and coherent. Remember, if we can't read it, it's wrong. Partial credit will be given, so try to put something down for each question (a blank answer always gets 0 points!).

1. (1 point)
   In Static ***Single*** Assignment (SSA) form, a variable is assigned to:
   (a) Zero times.
   (b) One time.
   (c) 3.1415926535 times.
   (d) As many times as the Florida Supreme Court decides it wants to allow.

2. (a) (10 points)
   Recall that the solution to a data flow problem need not be unique. Give a *simple* example of an instance of a data flow problem that has two or more solutions.
   (b) (10 points)
   Assume that a Control Flow Graph is a DAG or Tree (it has no cycles). Is it possible for a forward data flow analysis performed on this CFG to have a non-unique solution? If so, give a *simple* example. If not, explain carefully why.
   (c) (13 points)
   Let us represent a solution to a forward data flow problem as a vector of the Out values for each block in the CFG being analyzed. That is, solution = (Out_0,Out_1,...,Out_n) where the CFG has blocks numbered 0 to n. We will say data flow solution A is ≥ Data Flow solution B (for the same CFG and the same analysis) if $A_0 \geq B_0$, $A_1 \geq B_1$, ..., $A_n \geq B_n$. That is, for each basic block, A's solution must be ≥ B's solution. Let S be any valid solution to a particular forward data flow problem for a given CFG, and let T be the solution computed by the iterative DFO solution algorithm presented in class. Show that T ≥ S. (That is, the iterative algorithm's solution is always as good as, or better than, any other valid solution.) You may assume, as usual, that all transfer functions are monotone, and that the standard lattice axioms (on ≥ and $\wedge$) hold.

3. It is often the case that an optimization requires accurate "points to" information to be effective. That is, it is necessary to determine the set of variables that a pointer might possibly reference.
   Assume that we are interested only in pointers to local or global variables of a program (dynamically allocated heap objects are ignored). We will focus on two kinds of assignments: `Var1 = const` and `var1 = var2`, where `var1` and `var2` are pointer variables (e.g., of type `int*`, or `char*`, etc.) and `const` is a pointer const (e.g., `null` or `&var`). Moreover, we will assume only one such assignment to a pointer variable occurs in each basic block (blocks can be split, as needed, to guarantee this).
   We wish to formulate a data flow problem that will determine, for a given pointer variable, the set of pointers it might contain at each basic block.
   (a) (13 points)
   Give a data flow framework (solution lattice, direction, transfer function and meet operation) that can be used to solve the "points to" problem described above.
   (b) (10 points)
   Is the data flow problem you formulated distributive? If it is, explain carefully why. If it is not, give a *simple* counter example.
   (c) (10 points)
   Is the data flow problem you formulated rapid? If it is, explain carefully why. If it is not, give a *simple* counter example.

4. In project 3 we investigated removing loop-invariant expressions from loops. Removing loop-invariant assignments is trickier, because of the side-effects of assignments. Consider the following two program fragments.

```
a=0;                            a=0;
b=0;                            b=0;
do                              do
   a=1;                            if (f(b) == 0)
   b++;                               a=1;
while (b<10);                      b++;
                                while (b<10);
```

   (a) (5 points)
   Explain carefully why moving `a=1` to the loop's preheader is correct in the left program, but is incorrect in the right program.
   (b) (14 points)
   Explain how to put these two programs into SSA (static single assignment) form. Show each of the two programs after they have been put into SSA form. (Show the programs in source form, with variables rewritten and phi functions added as needed.)
   (c) (14 points)
   Assuming a program is in SSA form, what rules are sufficient to determine if an assignment can be moved from within a loop to the loop's preheader?

5. (a) (17 points)

   Some language designers have suggested the following form of loop, which unifies while and do-while loops into a single more general structure:

```
loop
   body1
   exit when (pred);
   body2;
end;
```

   In this loop, the termination condition is in the "middle" of the loop. $\texttt{body}_1$ is executed. Then `pred` is tested. If it is true, the loop is terminated. Otherwise, $\texttt{body}_2$, followed by $\texttt{body}_1$, are executed, and `pred` is tested again. Iteration continues until `pred` becomes true.

   Explain how to compute the transfer function of this form of loop given the transfer functions of $\texttt{body}_1$, $\texttt{body}_2$, and `pred`.

   (b) (16 points)

   A sequence of if-then-else statements can be combined into a single if-then-elsif statement:

```
if (pred1)
   body1;
 elsif (pred2)
   body2;
 ...
 elsif (predn)
    bodyn;
 else
   bodyn+1;
```

   Explain how to compute the transfer function of this form of if statement given the transfer functions of $\texttt{body}_1$, $\texttt{body}_2$,..., $\texttt{body}_{n+1}$, $\texttt{pred}_1$,..., $\texttt{pred}_n$.

6. This question involves partial redundancy elimination. The data flow equations that define partial redundancy are listed at the end of the question.

   (a) (11 points)

   Assume an expression, e, has been placed at the end of block b using the Insert_b rule defined below. If all of b's immediate successors neither use nor kill e, under what circumstances can the computation of e be sunk to each of b's successors?

   (b) (11 points)

   Assume an expression, e, is computed in block b, and that no prior computation of e occurs on any path from b_0 to b. Show that the partial redundancy equations will never add a computation of e to any predecessor of b.

(c) (11 points)

Is it ever possible for the Insert_b and Remove_b rules of the partial redundancy equations to specify that the same expression is to be both inserted into and removed from the same block b? If so, give a *simple* example. If not, explain why this can't happen.

$$\text{PPOut}_b = 0 \text{ for all exit blocks}$$
$$= \underset{k \in \text{succ}(b)}{\text{AND}} \text{PPIn}_k$$

$$\text{Const}_b = \text{AntIn}_b \text{ AND } [\text{PavIn}_b \text{ OR } (\text{Transp}_b \text{ and } \neg \text{AntLoc}_b)]$$

$$\text{PPIn}_b = 0 \text{ for } b_0 \text{ (the start block)}$$
$$= \text{Const}_b \text{ AND } (\text{AntLoc}_b \text{ or } (\text{Transp}_b \text{ AND } \text{PPOut}_b) \underset{p \in \text{pred}(b)}{\text{AND}} (\text{PPOut}_p \text{ OR } \text{AvOut}_p)$$

$$\text{Insert}_b = \text{PPOut}_b \text{ AND } (\neg \text{AvOut}_b) \text{ AND } (\neg \text{PPIn}_b \text{ OR } \neg \text{Transp}_b)$$
$$\text{Remove}_b = \text{AntLoc}_b \text{ AND } \text{PPIn}_b$$

Partial Redundancy Equations