CS 701

Final Exam

Tuesday, December 18, 2001

10:00 a.m. — noon

1325 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 loop-invariant code motion, we move code out of what program construct:
   (a) If statements.
   (b) Loop bodies.
   (c) Declarations.
   (d) Null statements.

2. (a) (23 points)
   In constant propagation we formulate a data flow problem that determines, for each variable, whether it is known to hold a particular constant value at a given point in a control flow graph.
   We can perform a related analysis that tracks the value of control predicates used in loops and conditionals. For example, when we enter the body of a while loop, we know the loop's control predicate must be true, and when we exit the loop the predicate must be false. Similarly, in the then part of an if, the if's control predicate must be true, and in the else part it must be false.
   Formulate a data flow analysis that determines, for each control predicate used in a control flow graph, where the expression is known to be true, or false, or non-constant.

   (b) (10 points)
   In conditional constant propagation, we generalized constant propagation analysis to recognize that certain paths in a control flow graph were impossible, and hence could be ignored. Can we do a similar generalization of control predicate analysis, using the value of control predicates to determine that certain paths are impossible? If so, outline how. If not, explain why this generalization is infeasible.

3. Because statements frequently assign to variables (that is, "define" them), it is often useful to compute a Def set for a statement. The set Def(s) represents all the variables that might be defined by executing statement s. Statement s might be an assignment, a conditional, a loop, or a procedure body. For purposes of this question, we'll only consider assignments to scalar variables, ignoring arrays, structs, and pointers. We'll also ignore aliasing effects of pointers.
   (a) (12 points)
   Give rules that define how to compute Def(s) where s may be an assignment, a conditional, a loop or a sequence of statements. Conditionals and loops may nest, but the only atomic statement allowed is the assignment statement.
   (b) (10 points)
   How must your solution to part (a) be changed if break, continue, and return statements are also allowed?
   (c) (11 points)
   Now include in your def set computation the effects of calls to procedures and functions of no arguments. What must be done if scalar by-reference parameters are allowed?

4. (a) (20 points)
   A key step in putting a program in SSA (static single assignment) form is placement of the ϕ functions. Assume that all assignments to variables have been rewritten so that each variable is now assigned to exactly once. Give an algorithm to determine, for a set of variables x_0, ... ,x_n derived from a single variable x, where to place ϕ functions and the appropriate arguments for each ϕ. How are the arguments (the x_i values) of each ϕ determined?
   (b) (13 points)
   Show that once a program is in SSA form, copy propagation is greatly simplified. In particular no analyses beyond those needed to put a program in SSA form are needed.

5. (a) (11 points)
   We showed in class that if a value lattice has a finite height, then our iterative algorithm for solving data flow equations must terminate with a solution (since on each iteration one or more In_b or Out_b values must change, and each value can change only a bounded number of times because the lattice has finite height).

   Recall that some lattices that we used had finite height but infinite width (e.g., the lattice used for constant propagation). If we have such a lattice, explain why our iterative algorithm can't loop forever moving "sideways" across the lattice. For example, given the statement `a=a+1` in a loop body, why can't our algorithm, when doing constant propagation, start with value v for a, then try v+1, then v+2, etc., never terminating?
   (b) (11 points)
   Show that if we use a lattice that has infinite height, then our iterative data flow algorithm can fail to halt. Can we sidestep this problem by using a structured solution in which a transfer function for a whole program is created by composing transfer functions for each expression and statement in the program?

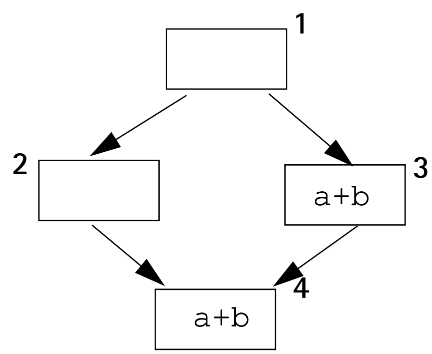
(c) (11 points)

Recall that DFO order (depth-first order) is recommended in solving data flow equations using an iterative algorithm. Why is DFO order better than an arbitrary visit order for nodes within the control flow graph? Is DFO order always best, or is a non-DFO order sometimes preferable (in reducing the number of iterations needed to solve a set of data flow equations)?

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

(a) (17 points)

Consider the following control flow graph. What are the PPIn and PPOut values for each block? Be sure to explain why your solutions are correct. How do these values determine where computations of `a+b` will be added or removed?



(b) (16 points)

Recall that one of the limitations of partial redundancy elimination is that computations of an expression are sometimes added earlier in a control flow graph than we'd like (extending the size of corresponding live ranges).

Assume we adopt the following heuristic in placing new computations of a expression:

If a block b is a candidate for removal of an expression (because $Remove_b$ is true), we will examine each of b's immediate predecessors. If the expression is fully available in predecessor p, we will add nothing in p. If the expression is partially available in p (but not fully available), we'll use the standard Morel & Renvoise placement rules. But, if the expression is unavailable in p, we'll add it there as long as the expression is anticipated in p and it reaches only expressions subject to removal (because their Remove values are true).

Does this heuristic work? That is, does it correctly transform partial redundancies into fully redundancies (subject to removal) or does this new placement rule sometimes fail to place expressions in the correct blocks? Justify your answer.

$PPOut_b = 0$ for all exit blocks

$$= \underset{k \in succ(b)}{\text{AND}} \; PPIn_k$$

$Const_b = AntIn_b$ AND

$[PavIn_b$ OR $(Transp_b$ and $\neg AntLoc_b)]$

$PPIn_b = 0$ for b_0 (the start block)

$= Const_b$ AND $(AntLoc_b$ or $(Transp_b$ AND $PPOut_b)$

$$\underset{p \in pred(b)}{\text{AND}} \; (PPOut_p \text{ OR } AvOut_p)$$

$Insert_b = PPOut_b$ AND $(\neg AvOut_b)$ AND $(\neg PPIn_b$ OR $\neg Transp_b)$

$Remove_b = AntLoc_b$ AND $PPIn_b$

Partial Redundancy Equations