CS 701

Final Exam

Wednesday, December 18, 1996

12:25 - 2:25

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)

Loop invariant code motion is an optimization that moves invariant expressions out of the body of what programming language construct?

2. (a) (11 points)

Assume we have a program in static single assignment form (SSA). Explain how to perform constant propagation assuming all program paths are executable.

(b) (11 points)

Generalize your solution to (a) by removing paths determined to be unreachable by using the results of constant propagation to evaluate (where possible) predicates that control conditional branches.

(c) (11 points)

Recall that SSA form can be viewed as a way to give unique names to *values* that are computed at various program points. Show that by using the unique value associated with each SSA variable, it is possible to sometimes evaluate in advance predicates that control conditional branches even when they do not involve constant-valued operands.

3. (33 points)

Assume that we limit ourselves to programs that contain only integer-valued scalar variables (no pointers or arrays are allowed). Programs are represented, for purposes of analysis, by a conventional control-flow graph.

A variable `x` is (as usual) defined to be *live* at a particular point in a program if there exists a path from that point to the end of the program on which `x` is used before it is defined. A variable that is not live is *dead*.

A concept related to dead variables is the notion of a *faint* variable. At any point faint variables are a superset of dead variables. A variable `x` at a particular point is faint if it is dead at that point or if it a live only because it is used in an assignment to a faint variable. A variable used in a control predicate or a print statement is never faint at the point at which it is used.

The concepts of liveness and faintness are illustrated in the following example program; comments refer to the program point just after the corresponding statement.

```
main() {
     i = 0; // i is live and not faint
     x = i; // x is live and faint
     y = x; // y is dead and faint
     while ( i <= 10 ) { // i is live and not faint
          x = x + 1; // x is live and faint
          i = i + 1; // i is live and not faint
     }  // i is live and not faint
        // x is dead and faint
     print(i); // i is dead and faint
 }
```

(a) (10 points)

Recall that live/dead variable analysis can be used to find and remove useless assignments. What are the advantages of using faintness information (as well as live/dead information) in removing useless assignments?

(b) (23 points)

Call a variable that is not faint *truly live*. True-liveness analysis identifies the minimal set of truly-live variables at each basic block. Give a data flow framework for determining true-liveness, That is, give the solution lattice, direction, transfer functions and meet operation necessary to compute `trulyLiveIN(n)` and `trulyLiveOut(n)` for `n∈basicBlocks`.

4. (a) (11 points)
Assume we have a C-style “do while” loop of the form

```
do
    body
while (expr);
```

Assume we are doing data flow analysis. Explain how to compute the transfer function of the “do while” loop given the transfer functions of `body` and `expr`. Assume that `body` contains no `gotos`, `breaks`, `returns` or `continues`.

(b) (11 points)
Extend your solution to (a) to the case in which `body` contains one top-level `break` controlled by a predicate. That is, the loop is now of the form

```
do
    body1
    if (pred)
            break;
    body2;
while (expr);
```

Explain how to compute the transfer function of this form of loop given the transfer functions of `body1`, `body2`, `pred` and `expr`. Again, assume that `body1` and `body2` contain no `gotos`, `breaks`, `returns` or `continues`.

(c) (11 points)
Extend your solution to (a) to the case in which `body` contains one top-level `continue` controlled by a predicate. That is, the loop is now of the form

```
do
    body1
    if (pred)
            continue;
    body2;
while (expr);
```

Explain how to compute the transfer function of this form of loop given the transfer functions of `body1`, `body2`, `pred` and `expr`. Again, assume that `body1` and `body2` contain no `gotos`, `breaks`, `returns` or `continues`.

5. The following three statements all pertain to the effects of partial redundancy elimination. For each statement, say whether the statement is true or false. If it is true explain carefully why it is true. If it is false, give a counter-example that illustrates the statement's falsity.

(a) (11 points)

Partial redundancy elimination never increases program size; that is, it always removes at least as many occurrences of an expression as it adds.

(b) (11 points)

Consider any execution path in a program (viewed as a sequence of basic blocks). After partial redundancy elimination the *first* evaluation of an expression never occurs *earlier* than it did in the original (unoptimized) program.

(c) (11 points)

Consider any basic block *b* that has only one predecessor and that is transparent with respect to an expression *e* (none of the *e*'s operands are changed in the block). If partial redundancy elimination inserts an evaluation of *e* in the *b*'s sole predecessor, then an evaluation of *e* will never be inserted into *b*.