CS 701

Final Exam

Tuesday, December 19, 2006

1:00 — 4:00 p.m.

1263 Computer Science

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)
   What are the **top**-most and **bottom**-most values in a lattice called?

2. This question involves points-to analysis in a language like C or C++. Assume the following pointer manipulation statements appear in a subprogram (points-to analyses are flow-insensitive so non-pointer statements are irrelevant):

```
p1 = &a;
p2 = p1;
p1 = &b;
p3 = &c;
p2 = &d;
r  = &p3;
s  = &p2;
*r = *s;
```

   (a) (11 points)
   Show the points-to graph that Andersen's Algorithm would compute for these statements.

   (b) (11 points)
   Show the points-to graph that Steengaard's Algorithm would compute for these statements. How does it compare to that of Andersen's Algorithm?

   (c) (11 points)
   Show the points-to graph that Horwitz's Algorithm would compute for these statements. Assume one run and two categories are used. The category assignments are:
   Category 1: `p1, a, b, r`
   Category 2: `p2, p3, c, d, s`
   How does this points-to graph compare with those produced in parts (a) and (b)?

3. Assume we have a Java-style try statement of the form

```
try
    body
catch
   handler;
```

In this try statement `body` is executed. If any statement in `body` executes a **throw** statement, execution transfers immediately to the statements in `handler`. If no throw is executed within `body`, then the statements in `handler` are ignored. Thus either all of `body` is executed or a prefix of `body` is executed, followed by the statements in `handler`.

   (a) (11 points)
   Assume we limit our try statement to the case in which `body` contains only one top-level throw controlled by a predicate. That is, the try is now of the form

```
try
   body₁
   if (pred)
            throw;
   body₂;
catch
   handler;
```

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

   (b) (11 points)
   Extend your solution to (a) to the case in which body contains more than one top-level throw, each controlled by its own predicate. That is, the try is now of the form

```
try
   body₁
   if (pred₁)
            throw;
   body₂;
   if (pred₂)
            throw;
   ...
   if (predₙ)
            throw;
   bodyₙ₊₁;
catch
   handler;
```

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

(c) (11 points)
Extend your solution to (a) to the case in which a try statement contains one top-level throw, controlled by a predicate, nested within a top-level do-while loop. That is, the try is now of the form

```
try
   body_1
   do
         body_2
         if (pred_1)
              throw;
         body_3;
   while (pred_2)
   body_4;
catch
   handler;
```

Explain how to compute the transfer function of this form of try statement given the transfer functions of $\texttt{body}_1$,..., $\texttt{body}_4$, $\texttt{pred}_1$, $\texttt{pred}_2$ and `handler`.

4. In a language like Java or C# in which all non-scalar objects are accessed through references, it is very useful to know, at a particular point, whether a reference is definitely null or definitely non-null, or possibly either (null or non-null).

   This motivates an "is null" data flow problem. For a given reference, `r`, $\mathsf{IsNull_{in}}$(b) tells us whether `r` is null at the start of block b. Possible values of $\mathsf{IsNull_{in}}$(b) are T (must be null), F (certainly is non-null), $\top$ (don't know yet), $\perp$ (may possibly be either null or non-null). Similarly, $\mathsf{IsNull_{out}}$(b) tells us whether `r` is null at the end of block b.

   We will focus on the following kinds of assignments to references: `ref1 = null`, `ref1 = new()`, `ref1 = method()`, and `ref1 = ref2`, where `ref1` and `ref2` are local reference variables and `method` is a call to a known method. We will assume all methods are preanalyzed and that we know the status of the reference each method returns (null, non-null, maybe either). Moreover, we assume only one such assignment to a reference variable occurs in each basic block (blocks can be split, as needed, to guarantee this).

   (a) (11 points)
   Give a data flow framework (solution lattice, direction, transfer function and meet operation) that can be used to solve the "is null" problem described above.

   (b) (11 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) (11 points)
   Is the data flow problem you formulated rapid? If it is, explain carefully why. If it is not, give a *simple* counter example.

5. (a) (8 points)
   In implementing loop-invariant code motion, **very busy** expression analysis would seem to be useful and appropriate since a very busy loop-invariant expression is an ideal candidate to move outside the loop. However, experience shows that identifying and moving very busy loop-invariants is quite ineffective. Explain why.

   (b) (17 points)
   Assume we profile the execution of a subprogram and mark certain transitions between basic blocks as **highly probable**. A transition is so marked if it is certain to be taken or exceeds a high threshold probability (e.g., 95% or better). A basic block may have at most one transition from it marked as highly probable. For some blocks **none** of its out transitions will be marked.

   Let us generalize very busy expression analysis to a new analysis called **highly probably expression analysis**. At the top of a basic block an expression is highly probable if it is computed in that block before any of the expression's operands are assigned to **or** if the block is transparent and the expression is highly probable at the end of the block.

   An expression is highly probable at the end of a basic block if it is highly probable at the start of all successor blocks **or** if it is highly probable at the start of a single successor connected by a highly probable transition.

   Give a data flow framework for determining if an expression is highly probable. That is, give the solution lattice, direction, transfer functions and meet operation necessary to compute $\mathsf{HP_{in}}$ and $\mathsf{HP_{out}}$ for each basic block.

   (c) (8 points)
   Assuming that we've done highly probable expression analysis, can we now effectively identify choice candidates for loop-invariant code motion? Why is this new analysis more effective than very busy expression analysis?

6. This question involves partial redundancy elimination. The data flow equations that define partial redundancy are listed at the end of this question.
   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 from b_0 to an exit block). 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 is transparent with respect to an expression *e* (none of the *e*'s operands are changed in the block) and has only one predecessor. 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*.

$PPOut_b = 0$ 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)

$$= Const_b \text{ AND } (AntLoc_b \text{ or } (Transp_b \text{ AND } PPOut_b) \underset{p \in pred(b)}{\text{AND}} (PPOut_p \text{ OR } AvOut_p)$$

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

$Remove_b = AntLoc_b$ AND $PPIn_b$

Partial Redundancy Equations