Programming Languages and Compilers
Qualifying Examination

Monday, September 21, 2009

Answer 4 of 6 questions.

GENERAL INSTRUCTIONS

1. Answer each question in a separate book.
2. Indicate on the cover of *each* book the area of the exam, your code number, and the question answered in that book. On *one* of your books list the numbers of *all* the questions answered. *Do not write your name on any answer book.*
3. Return all answer books in the folder provided. Additional answer books are available if needed.

POLICY ON MISPRINTS AND AMBIGUITIES

The Exam Committee tries to proofread the exam as carefully as possible. Nevertheless, the exam sometimes contains misprints and ambiguities. If you are convinced that a problem has been stated incorrectly, mention this to the proctor. If necessary, the proctor will contact a representative of the area to resolve problems during the *first hour* of the exam. In any case, you should indicate your interpretation of the problem in your written answer. Your interpretation should be such that the problem is nontrivial.

Question 1.

Suppose that we have a collection of recursive equations

$$x_i = F_i(\vec{x}) \tag{1}$$

or, alternatively,

$$\vec{x} = F(\vec{x}) \tag{2}$$

where F_i defines the value of the i-th component of tuple $\vec{x}$. (You can assume that the domain and range of each of the F_i is a pointed chain-complete partial order $(D_\perp, \sqsubseteq)$, and that each of the functions F_i is continuous.)

Consider the related collection of recursive equations

$$x_i = x_i \sqcup F_i(\vec{x}) \tag{3}$$

or, alternatively,

$$\vec{x} = \vec{x} \sqcup F(\vec{x}) \tag{4}$$

This problem concerns what happens when a set of equations of the form Eqn. (2) is converted to one of the form Eqn. (4). To distinguish between the two forms, we will use the subscript "old" for the right-hand side of Eqn. (2); i.e., we rewrite Eqn. (2) as

$$\vec{x} = F_{old}(\vec{x})$$

and use the subscript "new" for the right-hand side of Eqn. (4); i.e., we rewrite Eqn. (4) as

$$\begin{array}{rcl} \vec{x} & = & F_{new}(\vec{x}) \\ & = & \vec{x} \sqcup F_{old}(\vec{x}) \end{array}$$

Part (a)
Show that every fixed point of Eqn. (2) is also a fixed point of Eqn. (4).

Part (b)
Explain why the converse of Part (a) does not necessarily hold—i.e., a fixed point of Eqn. (4) may not necessarily be a fixed point of Eqn. (2). (You can either give a specific counter-example, or you can explain where an attempt to show the converse of Part (a) breaks down.)

Part (c)
Show that the least fixed point of Eqn. (4) equals the least fixed point of Eqn. (2).

Part (d)
Show that, for every value $\vec{v}$, we have the following two properties:

1. The sequence of values $[F^i_{new}(\vec{v}) \mid i \in \mathit{Nat}]$ forms a chain (hence $\bigsqcup_{i=0}^{\infty} F^i_{new}(\vec{v})$ is a well-defined value).
2. $\bigsqcup_{i=0}^{\infty} F^i_{new}(\vec{v})$ is a fixed point of F_{new}.

(Note that $F^0_{new}(\vec{v})$ equals $\vec{v}$; it does *not* equal $\vec{\perp}$.)

Part (e)
Explain how the ideas from Parts (a)–(d) can be used in incremental dataflow analysis, where the goal is to use the results from a previously computed dataflow-analysis solution $\vec{v}_0$ to compute a conservative—but possibly not optimal—solution after a program has been changed.

To simplify the situation, assume that a program modification does not cause the dataflow lattice (or semi-lattice) to change.

Question 2.

A *regular tree grammar* is a formalism for specifying languages of trees. For instance, the following grammar G

$$\begin{array}{rcl} \textit{exp} & ::= & \textit{PlusExp}(\textit{exp}, \textit{exp}) \\ & | & \textit{TimesExp}(\textit{exp}, \textit{exp}) \\ & | & \textit{IntExp}(\textit{natNum}) \\ & | & \textit{Variable}(\textit{ident}) \end{array}$$

where *natNum* $= \{0, 1, 2, \ldots\}$ and *ident* is some finite or infinite set of allowable identifiers (e.g., $\{A, B, \ldots, X, Y, Z\}$), defines a language $L(G)$ of trees (or terms). $L(G)$ includes the trees

$$\begin{array}{l} \textit{Variable}(A), \textit{Variable}(B), \ldots, \textit{Variable}(Z), \\ \textit{IntExp}(0), \textit{IntExp}(1), \ldots, \\ \textit{PlusExp}(\textit{Variable}(A), \textit{IntExp}(0)), \textit{PlusExp}(\textit{Variable}(B), \textit{IntExp}(0)), \ldots, \\ \textit{TimesExp}(\textit{Variable}(A), \textit{IntExp}(0)), \textit{TimesExp}(\textit{Variable}(B), \textit{IntExp}(0)), \ldots, \\ \textit{PlusExp}(\textit{IntExp}(0), \textit{Variable}(A)), \textit{PlusExp}(\textit{IntExp}(0), \textit{Variable}(B)), \ldots, \\ \textit{TimesExp}(\textit{IntExp}(0), \textit{Variable}(A)), \textit{TimesExp}(\textit{IntExp}(0), \textit{Variable}(B)), \ldots, \end{array}$$

Let us now introduce some terminology: *exp, natNum,* and *ident* are called *nonterminals* (or *types*); *PlusExp, TimesExp, IntExp,* and *Variable* are called *operators*. It is useful to consider $0, 1, 2, \ldots$ as nullary operators of type *natNum* (in which case we might write them as $0(), 1(), 2(), \ldots$) and $A, B, \ldots, X, Y, Z$ as nullary operators of type *ident* (in which case we might write them as A(), B(), …, X(), Y(), Z()). Hence, with this notation one of the trees in $L(G)$ is *PlusExp*(*Variable*$(A())$, *IntExp*$(0())$).

Note that each operator has a fixed *arity* that specifies the number of children that it has. For instance, the arities of some of the operators of G are as follows:

Operator	*Arity*
PlusExp	2
TimesExp	2
IntExp	1
Variable	1
0	0
1	0
2	0
⋮	⋮
A	0
B	0
⋮	⋮
Z	0

Let the children of an arity-k operator be numbered $1, \ldots, k$.

A *path* in a tree can be described by a string over an alphabet of (compound) symbols of the form

$$(\textit{nonterminal} :: \textit{Operator.childNum})$$

(By convention, if *Operator* is a nullary operator, *childNum* is 0.) For instance, the set of root-to-leaf paths in the tree *PlusExp*(*Variable*($A()$), *IntExp*($0()$)) is

$$\left\{ \begin{array}{l} (\textit{exp} :: \textit{PlusExp}.1)(\textit{exp} :: \textit{Variable}.1)(\textit{ident} :: A.0), \\ (\textit{exp} :: \textit{PlusExp}.2)(\textit{exp} :: \textit{IntExp}.1)(\textit{natNum} :: 0.0) \end{array} \right\}.$$

Part (a)
Describe how to create an ordinary finite-state automaton that accepts the language of root-to-leaf paths in given a regular tree grammar H. That is, given a regular tree grammar H, your construction should produce the automaton A_H that accepts

$$\{p \mid p \text{ is a root-to-leaf path in some tree T } \in L(H)\}.$$

Part (b)
Give the automaton that would be produced by your construction for the regular tree grammar G

$$\begin{array}{rcl} \textit{exp} & ::= & \textit{PlusExp}(\textit{exp}, \textit{exp}) \\ & | & \textit{TimesExp}(\textit{exp}, \textit{exp}) \\ & | & \textit{IntExp}(\textit{natNum}) \\ & | & \textit{Variable}(\textit{ident}) \end{array}$$

Part (c)
Regular tree grammars are related to context-free grammars in the following way: Suppose that you normalize a context-free grammar F by introducing additional nonterminals so that terminal symbols only appear in leaf productions of the form *nonterminal* $\rightarrow$ *terminal*; then, by introducing an operator symbol for each production (and treating each terminal symbol as a nullary operator), one has a regular tree grammar whose language is the set of parse trees for the context-free grammar F.

A context-free grammar can have two kinds of useless nonterminals:

Useless 1 : nonterminal n is useless if there is no derivation $root \rightarrow^* \alpha\ n\ \beta$
Useless 2 : nonterminal n is useless if there is no finite parse tree derivable from n

Describe *two* algorithms, both working on finite automata of the kind described earlier for the language of root-to-leaf paths of a regular tree grammar:

Part (c.i) The algorithm for this part returns the set of nonterminals that are useless because of reason "Useless 1".

Part (c.ii) The algorithm for this part returns the set of nonterminals that are useless because of reason "Useless 2". (For this part, you may assume that all "Useless 1" nonterminals were removed from the context-free grammar before the automaton was constructed.)

Part (d)
Give an example of a *context-free grammar* that has both kinds of useless nonterminals, and illustrate the two algorithms on it.

Question 3.

Recall that a lambda expression that contains one or more redexes can be reduced using normal order reduction (always reduce the leftmost outermost redex) or using applicative order reduction (always reduce the leftmost innermost redex).

Part (a)
What is an advantage of normal order reduction over applicative order reduction? Give an example to illustrate this advantage.

Part (b)
What is a disadvantage of normal order reduction over applicative order reduction? Again, give an example.

Part (c)
Is the following strategy equivalent to normal order reduction?

> *Always reduce the rightmost outermost redex.*

(That is, will the two strategies lead to a normal form in exactly the same cases?) If yes, briefly justify your answer; if no, give a counter-example (a lambda term for which one strategy leads to a normal form, while the other strategy does not).

Part (d)
For each of the following statements, say whether it is true or false, and give a brief justification of your answer.

1. Every lambda term is equal to a lambda term that is in normal form.
2. Every lambda term is equal to a lambda term that is not in normal form.
3. Every lambda term has some lambda term as its fixed point.
4. Every lambda term is the fixed point of some lambda term.
5. There is a lambda term that is its own fixed point.

Question 4.

Assume that you have a simple C-like language with no pointers, but with 1-dimensional arrays that are declared with a statically known size. Furthermore, array-subscript expressions can only include integer literals and scalar variables (e.g., A[k] and A[k*5] are OK, but A[B[2]] is not allowed). As in Java programs, bounds checking is done so that an out-of-bounds array index (either less than zero or greater than or equal to the array size) causes a runtime error. For example, the statement "x = A[k+j];", where A has been declared to be of size 10, would be handled (in the low-level code that implements the statement) roughly as follows:

```
if (k+j < 0) ERROR;
if (k+j >= 10) ERROR;
x = A[k+j];
```

To simplify the question, assume that arithmetic operations cannot cause overflow (i.e., we are working with integers, not ints).

Part (a)

Your job is to define an intraprocedural dataflow analysis that can be used to remove bounds checks that are guaranteed to succeed because of some previous check having been done. For example, in the following code both upper and lower bounds checks can be removed for subscript expressions 3, 6, and 7.

```
A[k+j] = 12;          // subscript exp 1
if (A[k] == 0) {      // subscript exp 2
   A[k] = A[k+n];     // subscript exps 3 and 4
} else {
   A[k+n] = 0;        // subscript exp 5
}
A[k+n] = A[k+j];      // subscript exps 6 and 7
```

Because a subscript expression that includes only literals (e.g., A[3] or A[4+5]) can be checked statically, assume that every subscript expression includes at least one variable.

Define your dataflow analysis, and explain how to use the results to eliminate bounds checks. To simplify your presentation, assume that there is only 1 array in the program being analyzed.

Part (b)

What other optimizations and/or code transformations could be done to reduce the number of bounds checks? Consider both reducing the total number in the program as well as the total number that are executed when the program is run. If the techniques that you describe require more dataflow-analysis results, explain how to perform that analysis.

Part (c)

Now assume that the language does allow pointers. The goal is still to do an intraprocedural analysis and to check that array-subscript expressions yield in-bounds values, not to check that dereferences of pointers to arrays are in-bounds. Suppose that you have run a flow-insensitive points-to-analysis algorithm, and have the results in hand. How would you use the points-to-analysis results to make your answers to Parts (a) and (b) sound?

```
 (1) main(int argc, char* argv[]){
 (2)    char header[2048], buf[1024], *ptr;
 (3)    int counter;
 (4)    FILE *fp;
 (5)    ...
 (6)    ptr = fgets(header, 2048, fp);
 (7)    copy_buffer(header);
 (8)    ptr = fgets (buf, 1024, fp);
 (9)    copy_buffer(buf);
(10) }
(11)
(12) void copy_buffer(char *buffer){
(13)    char copy[20];
(14)    strcpy(copy, buffer);
(15) }
```

Figure 1: Example program.

Question 5.

Part (a)
Languages like C that do not guarantee array-bounds checking and that allow pointer arithmetic can lead to programs that are vulnerable to certain kinds of malicious attacks. Explain how a malicious user can exploit buffer overrun vulnerability in a program.

Part (b)
Consider the program shown in Figure 1. How could a malicious user exploit the buffer overrun in this program, to execute the system call `system("open /etc/passwd")`? Assume that the malicious user controls the contents of the file that `fp` points to. Based on your answer discuss why buffer- overrun attacks belong to a class of attacks called code-injection attacks.

Part (c)
How would you implement a source-to-source translator that transforms a C program to a safe C program. A safe C program reports an error before a buffer overrun. *Hint:* Think about keeping extra information with each pointer.

Part (d)
A safe C program generated by source-to-source translator can have prohibitive overheads. Describe how you can use static analysis to optimize your source-to-source translator.

Question 6.

The Java virtual machine includes an unusual pair of instructions, `jsr` and `ret`, which can be used to implement "lightweight subroutines". For purposes of this question, we give these instructions the following, simplified behaviors:

- `jsr` a pushes the address of the next instruction on a special execution-address stack which is distinct from the regular stack used to contain regular procedure activation records. It then continues execution with the instruction at address a.
- `ret` pops the most recent value from the special execution-address stack and continues executing with the instruction at that address.

Note that because `jsr` and `ret` manipulate a distinct stack, these instructions do not change the local variables visible to executing code.

In the question parts that follow, if you do not remember the details of the Java VM's unusual operand-stack architecture, feel free to assume a more standard execution environment such as that found on any modern, real processor. Answers using either machine model are equally acceptable.

Part (a): Code Generation for `Finally` Clauses
Describe how `jsr` and `ret` could be used to good effect when generating code for the `finally` clauses of `try` blocks.

Part (b): `Finally` Clauses Without Lightweight Subroutines
Suppose `jsr` and `ret` were not available. Describe an alternative strategy for compiling `finally` blocks which avoids using these but which could cause the compiled machine code (or bytecode) to be exponentially larger than the source code in some pathological cases.

Part (c): Pathological Expansion
Give an example of source code which exhibits the pathological expansion mentioned above.

Part (d): Trade-Offs
Even if `jsr` and `ret` are available, perhaps we do not want to use them. Describe a scenario in which we would prefer to use the second code-generation strategy (from Part (b)) even if the first (from Part (a)) is available.