Programming Languages and Compilers Qualifying Examination

Fall 2012

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.
This question concerns interprocedural dataflow-analysis algorithms that use the "call-strings" approach to obtain a degree of context sensitivity. In such algorithms, each dataflow fact is tagged with a call-stack suffix (also known as a "call-string") to form (call-string, dataflow fact) pairs; the call-string is used at the exit node of each procedure to determine which call-site to propagate a (call-string, dataflow fact) pair to. For instance, if Main calls A at main-to-a, and A calls C twice, at a-to-c1 and a-to-c2, a tagged dataflow fact of the form ([main-to-a, a-to-c1, C], d) at the exit node of C would be propagated back to call-site a-to-c1 in A (where it would have the form ([main-to-a, A], d)), and not to call-site a-to-c2 in A.

Part (a)
In many program analyses, all instances of a given local scalar variable `x` of a procedure `P` are represented by a single abstract variable $x^{\#}$. Such a strategy can cause a loss of precision when `P` is recursive, either directly or through a chain of calls: when P is recursive, in general $x^{\#}$ represents multiple instances of `x` that occur in different activation records. However, a given assignment "`x := exp`" only modifies *one* occurrence of `x`. To be sound, program-analysis algorithms perform *weak updates* on abstract variables like $x^{\#}$ that represent more than one instance of a local variable. For example, in a forward analysis, `exp` is abstractly evaluated with respect to the current abstract state $S^{\#}_{\text{in}}$ to obtain some abstract value $v^{\#}$, and the post-state value of $x^{\#}$ is set to the pre-state value of $x^{\#}$ joined with $v^{\#}$:

$$S^{\#}_{\text{out}} := S^{\#}_{\text{in}}[x^{\#} \leftarrow S^{\#}_{\text{in}}(x^{\#}) \sqcup v^{\#}].$$

In other words, the abstract values obtained for `x` are *accumulated*.

In contrast, a *strong update* corresponds to a "kill" of a scalar variable: it represents a definite change in value to *all* concrete variables that the abstract variable represents. Strong updates cannot generally be performed on summary variables like $x^{\#}$ because a (concrete) update only affects *one* of the summarized concrete variables.

Describe how *unbounded-length* call-strings can be used to determine situations when $x^{\#}$ is definitely *not* a summary variable, and how this information can be used to improve precision by having the analysis perform a strong update instead of a weak update in such situations.

Illustrate your answer with an example, and indicate the calling contexts in your example in which $x^{\#}$ is a summary variable and those in which it is not.

Part (b)
To prevent call-strings from growing to unbounded lengths, a finite bound k on their length is generally imposed: a call-string that would grow longer than k (if represented fully precisely) is truncated to length k; it is said to have *saturated*. Each call-string has information saying whether or not it is saturated. For instance, if the call-string length k is 2, then [main-to-a, a-to-c1, C] would be represented by the saturated call-string *[a-to-c1, C], where * indicates that the call-string is a saturated call-string.

Describe how the call-string component of a tagged dataflow fact should be propagated at

(i) a call-site

(ii) the exit of a procedure

Your answer should explain what manipulations are performed both for saturated and unsaturated call-strings.

Part (c)
A *call multi-graph* is a graph in which the nodes represent procedures and each edge is of the form (P $\rightarrow^c$ Q), which represents the fact that P calls Q at call-site c. Given a call-string CS, call multi-graph CG, and procedure P, give an algorithm to determine whether, in a calling context described by CS, there could be 0, 1, or more-than-1 pending invocations of P. (If P is the currently active procedure, P is considered to be pending.) Stated another way, the algorithm should indicate whether an activation record for P can appear 0, 1, or more-than-1 times in some stack in the concretization $\gamma(\mathrm{CS})$ of call-string CS. (The answer reported by the algorithm should be the greatest value for some stack in the concretization of CS. For example, if different stacks in the concretization of CS could have 0, 1, and more-than-1 pending calls on P, the answer reported by the algorithm should be "more-than-1".)

Part (d)
Revisit the issue considered in Part (a) in light of your answer to Part (c). That is, describe how when working with bounded-length (and hence possibly saturated) call-strings, the information obtained via your answer to Part (c) can be used to determine situations when a local scalar variable like `x` is definitely not a summary variable.

Consider again the example you used in Part (a) and explain what would be known—via the strategy from Part (c)—about whether $x^\#$ is or is not a summary variable.

Question 2.
This question concerns fixed-point combinators and methods for finding fixed points in λ-calculus. Parts (a) and (b) concern the following theorem for characterizing fixed-point combinators themselves as fixed points:

$$\text{Let } G = \lambda y.\lambda f.f(yf)\text{. Then } M \text{ is a fixed point combinator if and only if } M = GM. \tag{1}$$

(Note: Recall that the following λ-calculus transformation is called the η-reduction rule:

$$(\lambda x.Mx) \rightarrow_\eta M,$$

where x does not occur as one of the free variables of M. You are allowed to use η-reduction in this question.)

Part (a)
Use (1) to show that $Y =_{\text{df}} \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))$ is a fixed-point combinator.

Part (b)
Prove (1). (Note that (1) involves an "if and only if"; consequently, your proof should have two parts.)

Part (c)
The fixed-point combinator discussed in Part (a) allows us to find a λ-term g that satisfies a single recursive equation over λ-terms of the form $g = \ldots g \ldots g \ldots$

Suppose that we are presented with a collection of k mutually recursive equations:

$$\begin{array}{rcl} g_1 & = & \ldots g_1 \ldots g_k \ldots \\ & \vdots & \\ g_k & = & \ldots g_1 \ldots g_k \ldots \end{array}$$

Explain how to solve for $g_1, \ldots, g_k$.

Question 3.
In Java, recall that any `try` statement may optionally include a `finally` clause. Code in the `finally` clause always executes after the main `try` code, regardless of whether any exception was thrown and caught, thrown and not caught, or not thrown at all.

The Java virtual machine includes an unusual pair of instructions, `jsr` and `ret`. 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.

Question 4.

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. Consider the program shown in Figure 1 (see next page). How could a malicious user cause a buffer overrun?

Part (b): Explain how a malicious user can exploit buffer overrun vulnerability in a program.

Part (c): We will sketch a static analysis technique to detect buffer overruns. Each buffer *buf* (variable of type `char *`) is associated with two range-valued variables $R_{len}(buf)$ and $R_{alloc}(buf)$ (one for the length and other for the allocated space). A program variable i (of type `int`) is associated with a single range-valued variable $R(i)$ (representing the possible values of i). Intuitively, if $R_{len}(buf) = (n, m)$, then the minimum and maximum length of the buffer *buf* are n and m respectively. Similarly, $R_{buf}(buf) = (n, m)$ indicates that the minimum and maximum allocated space for the buffer *buf* are n and m respectively. Each program statement generates a subset constraint on range-valued variables. For example, consider the following statement:

```
strcpy(a,b)
```

Since b is copied into a, the following constraint is generated:

$$R_{len}(b) \subseteq R_{len}(a)$$

Show the range constraints generated by the program given in Figure 1.
Note: You will have to use sets of constraints that model the library functions `fgets` and `strcpy`. I am assuming you know the semantics of `strcpy`. Description of `strlen` and `fgets` is given below:

```
strlen() returns the number of characters upto, but not including
till the nearest '\0'

char *fgets(char *s, int size, FILE *stream);
fgets() reads in at most size-1 characters
from stream and stores them into the buffer pointed to by s.
Reading stops after an EOF or a newline.  If a newline is read,
it is stored into the buffer.  A '\0' is stored
after the last character in the buffer.
```

Part (d): *Solving* a system S of range constraints means finding the "tightest" possible ranges that respect the constraints in S. For example, consider the following system S_1 of range constraints:

$$\begin{array}{rcl} (4,4) & \subseteq & R(a) \\ (8,8) & \subseteq & R(b) \\ R(b) & \subseteq & R(a) \end{array}$$

The following assignment of ranges is the "tightest" that respects constraints in S_1:

$$\begin{array}{rcl} R(a) & = & (4,8) \\ R(b) & = & (8,8) \end{array}$$

```
main(int argc, char* argv[]){
   char header[2048], buf[1024],
        *cc1, *cc2, *ptr;
   int counter;
   FILE *fp;
   ...
   ptr = fgets(header, 2048, fp);
   cc1 = copy_buffer(header);
   ptr = fgets (buf, 1024, fp);
   cc2 = copy_buffer(buf);
}

char *copy_buffer(char *buffer){
   char *copy;
   copy = (char *) malloc(strlen(buffer));
   strcpy(copy, buffer);
   return copy;
}
```

Figure 1: Example Program

Notice that the following assignment of ranges also respects the constraints in S_1, but does not assign the “tightest” possible ranges.

$$\begin{aligned} R(a) &= (1,9) \\ R(b) &= (5,8) \end{aligned}$$

Suppose there is procedure P for solving a system of range constraints, i.e., procedure P returns the “tightest” possible ranges that respect the constraints in a system S. How will you use procedure P to discover buffer overruns?

Part (e): Consider the range constraints from part(c). Give a graph algorithm to solve the range constrains generated in part(c). You need only explain your algorithm with respect to the specific set of constraints generated in part (c).

Question 5.

Part (a):
What is SSA form and what properties does it have? What (source-level) language constructs complicate the use of SSA form, and how are those constructs usually handled? Give examples to illustrate your answers.

Part (b):
Assume that a program has been translated to 3-address form and that a control-flow graph (in which nodes are basic blocks) has been built.

How can this representation be transformed to SSA form? (You don't have to give the best algorithm, but if you know that your algorithm is not the best, explain what it might do that is sub-optimal.)

Give a small example to illustrate your approach.

Part (c):
Why would anyone want to use SSA form?

```
assert (x > 0)
y = f(x);  // assertion checked before assign

assert (x == y && g(y))
for (k=0; k<y; k++) { ... }  // assertion checked once before loop

assert (! f(x)) {
  a = ...;
  b = ...;
  c = ...;
}  // assertion checked before EACH assignment

while (q)
  assert (x != 0)
  A[j++] = f(x);  // assertion checked before assign each time around loop
```

Figure 2: Four examples of uses of assertion blocks.

Question 6.
Consider adding assertion blocks to the C language. An assertion block is a Boolean expression P and a statement S. S can be a single statement (e.g., an assignment, an if-then-else, a loop) or a block (a sequence of statements inside curly braces). If S is a single statement, expression P is tested before S. If S is a block, P is tested before each statement in the block. If P evaluates to false, execution terminates with an error message.

Four examples of uses of assertion blocks are shown in Figure 2.

Part(a):
Describe how a compiler could implement assertions in a straightforward manner.

Part(b):
A straightforward implementation can sometimes be improved by avoiding evaluation of P (or parts of P). What could a compiler do to implement this kind of improvement?

Part(c):
A compiler can sometimes make use of assertions to do a better job of optimization. Give at least 3 (different) examples to illustrate this idea.