SPRING 2015 PH.D. QUALIFYING EXAMINATION
Programming Languages, Compilers, and Security
Computer Sciences Department
University of Wisconsin–Madison

Monday, February 4, 2013

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.
3. Return all answer books in the folder provided. Additional answer books are available if needed.

SPECIFIC INSTRUCTIONS:

Answer 4 of the 9 questions. Questions 1–5 relate to programming languages, compilers, and software engineering. Questions 6–9 relate to security and cryptography. However, you are not restricted to answering questions from just one group or the other: you may answer any 4 out of the entire pool of 9 questions.

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 a problem has been stated incorrectly, mention this to the proctor. If necessary, the proctor can 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: (Post-Mortem Analysis)

To perform post-mortem analysis after a program has crashed, it can be useful to have one or more snapshots of some aspect(s) of the states that the program has passed through.

Part (a):

How could a post-mortem-analysis tool make use of one or more stack snapshots to support queries that a user might be interested in making?

Part (b):

How could a post-mortem-analysis tool make use of one or more heap snapshots to support queries that a user might be interested in making?

Part (c):

State the key property or properties that make a garbage collector "real-time."

Part (d):

One approach to creating a real-time garbage collector is to use two or more "semi-spaces." Give algorithms for the allocate and free operations of a semi-space-based real-time garbage collector. (For simplicity, you need only discuss the workings of a real-time garbage collector that would be used in the run-time system for a single-threaded application.)

Part (e):

Note that a frozen semi-space is a memory snapshot. One could just use an additional thread to write out the semi-space to disk; however, this approach could interfere with the real-time aspect of the garbage collector. Explain how your algorithms from the previous part can be modified to create a real-time garbage collector that also collects memory snapshots.

Question 2: (Structural Induction)

Consider the following definition of function `reduce`:

```
let rec reduce(f, L, b) =
cases (L) of
  nil:  b
  x::L1:  f(x, reduce(f, L1, b))
```

One way to think about `reduce` is to think of f as a binary (infix) operator and to think of `reduce` as:

- putting the value b at the end of the list L, then
- putting the operator between all of the items in L, then
- evaluating right-to-left.

I.e., when applied to function f, a list containing elements $x_1, x_2, \ldots, x_n$, and "base" value b, `reduce` returns

$$f(x_1, f(x_2, f(\ldots, f(x_n, b) \ldots))).$$

Part (a): Define a new function, `red` that is similar to `reduce`, but (conceptually)

- puts the value b at the *front* of the list L, then
- puts the operator between all of the items, then
- evaluates left-to-right.

I.e., when applied to function f, a list containing elements $x_1, x_2, \ldots, x_n$, and base value b, `red` returns

$$f(\ldots f(f(b, x_1), x_2), \ldots, x_n).$$

Part (b): Show, using structural induction, that if f is associative and commutative, then for all L and all b,

$$\texttt{red}(f, L, b) = \texttt{reduce}(f, L, b).$$

Question 3: Lazy Evaluation

Suppose we wish to add lazy evaluation to Java. If *e* is some expression, then **`lazy`** *e* is an unevaluated lazy expression that can later be evaluated to produce an actual value. If *e* is some unevaluated lazy expression, then **`force`** *e* evaluates *e* and returns an actual value, caching this result so that it can be returned again in the future without evaluating *e* for a second time. If *e* is a lazy expression that has already been forced once, **`force`** *e* does not evaluate *e* twice: it simply returns the value cached from the first forcing. Only lazy expressions can be forced. "**`force force`** *e*" is only sensible if *e* is a lazy expression whose computed result is itself another lazy expression.

We use the **`lazy`** keyword to qualify types as well: **`lazy`** τ is the type of a lazy expression (initially unevaluated) that can later be forced (evaluated at most once, cached thereafter) to produce a result of type τ. Fields, formal parameters, and local variables may all be declared with **`lazy`**-qualified types.

For example, consider the following code:

```
class Example {
    private static int counter;

    private static int next() {
        ++counter;
        System.out.println("next counter: ", counter);
        return counter;
    }

    public static void main(String [] args) {
        System.out.println("start of main");
        lazy int future1 = lazy next();
        lazy int future2 = lazy next();
        System.out.println("two lazy expressions stored");
        System.out.println("first lazy expression forced: ", force future1);
        System.out.println("direct call without lazy expression: ", next());
        System.out.println("second lazy expression forced: ", force future2);
        System.out.println("first lazy expression forced again: ", force future1);
        System.out.println("second lazy expression forced again: ", force future2);
    }
}
```

This code outputs:

```
start of main
two lazy expressions stored
next counter: 1
first lazy expression forced: 1
next counter: 2
direct call without lazy expression: 2
next counter: 3
second lazy expression forced: 3
first lazy expression forced again: 1
second lazy expression forced again: 3
```

Observe that each lazy expression is not evaluated until forced. Also notice that forcing the same expression a second time does not cause it to be reevaluated. Even if the first evaluation had side effects, the value computed by the first **`force`** is cached and reused thereafter.

A lazy expression may refer to certain named variables that are in scope at the point where the **`lazy`** *e* construct appears. In general, any name that is accessible to the code containing the **`lazy`** *e* expression is also accessible to the *e* expression itself. There is, however, one very important exception: local variables and formal parameters of the containing method are only accessible to *e* if declared **`final`**. (Note that the same restriction applies to inner classes, a standard Java feature.) For example:

```
void example(int position, final String message)
{
    final int offset = ... ;
    String line = ...;

    ... lazy message.charAt(position) ... ;  // error: position not final
    ... lazy line.charAt(offset) ... ;       // error: line not final
    ... lazy message.charAt(offset) ... ;    // OK
}
```

Part (a): Type Checking

Choose an appropriate general form for typing judgments on expressions and clearly state what that form and its components mean. Then use judgments of this form to write type-checking rules for the new **lazy** *e* and **force** *e* expressions.

Beyond giving suitable types to **lazy** *e* and **force** *e*, your rules should also enforce the named-variable access restrictions mentioned in the preceding paragraph. In particular, **lazy** *e* should not type check if *e* accesses non-**final** locals or formals.

Part (b): Subtyping

Let "$\leq$" represent the reflexive closure of the subtyping relation. Java already has a subtyping relation based on classes and interfaces: $\tau_1 \leq \tau_2$ if τ_1 extends or implements τ_2.

Consider the following possible options for extending subtyping to lazy expressions:

1. $\tau \leq \textbf{lazy}\ \tau$
2. $\textbf{lazy}\ \tau \leq \tau$
3. $\tau_1 \leq \tau_2 \implies \textbf{lazy}\ \tau_1 \leq \textbf{lazy}\ \tau_2$
4. $\tau_1 \leq \tau_2 \implies \textbf{lazy}\ \tau_2 \leq \textbf{lazy}\ \tau_1$

Which of these are correct and appropriate, and which would be incorrect or undesirable? Justify your response. You should consider both formal and practical implications of the decision you make.

"Correct" subtyping rules should at least preserve subject reduction and substitutability of subtypes for supertypes. If you have additional considerations in mind, state them clearly and explain why you find them necessary or appropriate.

Part (c): Implementation and Behavior

Can lazy evaluation be implemented using a standard, modern Java virtual machine and runtime environment, or would runtime changes be required? If no changes would be needed, describe in detail how to map lazy-evaluation features onto existing constructs in the Java language and/or virtual execution environment. If changes would be needed, describe the smallest set of changes that would allow lazy evaluation with reasonable semantics and efficiency.

Part (d): Erasure

Suppose we erase all lazy-evaluation constructs from a program, turning **lazy** *e* and **force** *e* into simply *e* and turning **lazy** τ into simply τ. Under what circumstances can be guarantee that the modified (non-lazy) program behaves indistinguishably from the original (lazy) version?

Ignore performance for purposes of this question. Consider only the observable output and/or values returned by the program.

Question 4: Lambda calculus

In this problem, we use the untyped lambda calculus with small-step call-by-value left-to-right evaluation. Recall this encoding of pairs:

$mkpair =_{\text{df}} \lambda x.\ \lambda y.\ \lambda z.\ z\,x\,y$
$fst =_{\text{df}} \lambda p.\ p(\lambda x.\ \lambda y.\ x)$
$snd =_{\text{df}} \lambda p.\ p(\lambda x.\ \lambda y.\ y)$

where *mkpair* models a pair of x and y, *fst* returns the first element of a pair, and *snd* returns the second element of a pair.

Part (a):

For any values v_1 and v_2, *fst* (*mkpair* v_1 v_2) produces a value in 6 steps. Writing only lambda terms (i.e., no abbreviations), show these steps. Show just the result of each step, not the derivation that produces it.

$(\lambda p.\ p\ (\lambda x.\ \lambda y.\ x))\ ((\lambda x.\ \lambda y.\ \lambda z.\ z\,x\,y)\ v_1\ v_2)$

$\rightarrow$ ____

$\rightarrow$ ____

$\rightarrow$ ____

$\rightarrow$ ____

$\rightarrow$ ____

$\rightarrow$ ____

Part (b):

Extend the encoding to include a *swap* function. Given an encoding of the pair (v_1, v_2), *swap* should return an encoding of the pair (v_2, v_1).

Part (c):

The most popular encoding of recursion in λ-calculus uses the Y combinator. Another approach uses the simple U combinator (which is not a fixed-point combinator):

$$U =_{\text{df}} \lambda x.\ x\,x$$

Using the U combinator, we can define the factorial function as follows:

$$fact =_{\text{df}} U(\lambda f.\ \lambda n.\ \ \textit{if}\ (n=0)\ \textit{then}\ 1\ \textit{else}\ n * ((f\,f)(n-1)))$$

(For clarity, we extend pure lambda calculus with conditionals and arithmetic.)
Prove that *fact* satisfies the following λ-calculus equation:

$$fact = (\lambda n.\ \textit{if}\ (n=0)\ \textit{then}\ 1\ \textit{else}\ n * (fact(n-1)))$$

Part (d):

Consider the following theorem for characterizing fixed-point combinators themselves as fixed points:

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

(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.)

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

(ii) Use Theorem 1 to show that the U combinator $(\lambda x.\ x\,x)$ is a not a fixed-point combinator.

(iii) Prove Theorem 1. (Note that the theorem involves an "if and only if"; consequently, your proof should have two parts.)

Question 5: Optimality of Delta Debugging

Let C be the set of all elementary changes. Let $T = C \rightarrow \{✓, ✗\}$ be the set of test functions that judge any subset of C to succeed or fail. For purposes of this question, assume that tests are always conclusive. Let $ddmin : T \times C \rightarrow C$ be the Delta Debugging minimization algorithm that minimizes failure-inducing changes, given some specific test function and an initial set of failing changes. Let $test \in T$ be a specific test function, and let $c \subseteq C$ be some initial failing set of changes such that $test(c) = ✗$. Then $ddmin(test, c)$ computes a smaller set of changes $c' \subseteq c$ such that $test(c') = ✗$ and c' is 1-minimal.

Part (a): 1-Minimality Defined

As noted, c' is guaranteed to be 1-minimal. What does that mean exactly? Formally define **1-minimality** in this context, using precise mathematical notation and the terms and symbols defined above.

Part (b): Local Optimality Defined

If c' were guaranteed to be **locally optimal**, what would that mean exactly? Formally define local optimality in this context, using precise mathematical notation and the terms and symbols defined above.

Part (c): Global Optimality Defined

If c' were guaranteed to be **globally optimal**, what would that mean exactly? Formally define global optimality in this context, using precise mathematical notation and the terms and symbols defined above.

Part (d): Local Optimality Assured

Although Delta Debugging only guarantees 1-minimality for arbitrary test functions, it may be able to do better under certain circumstances. Describe the subset of T for which *ddmin* will always provide a **locally-optimal** result. What is the weakest set of restrictions that ensure this? (Or equivalently, what is the largest subset of T for which this is assured?) Briefly justify your answer.

Part (e): Global Optimality Assured

Describe the subset of T for which *ddmin* will always provide a **globally-optimal** result. What is the weakest set of restrictions that ensure this? (Or equivalently, what is the largest subset of T for which this is assured?) Briefly justify your answer.

Question 6: Memory vulnerabilities [Security]

Refer to the C program in Figure 1.

Part (a):

Explain the memory-corruption vulnerability, how an attacker would exploit it assuming they can execute the program from the command line, and how to remove the vulnerability from the program.

Part (b):

Would a stack canary prevent exploitation of this vulnerability? Explain.

Part (c):

W^X (write xor executable) protections allow marking memory pages as writable or executable, but not both. Would marking the pages containing the stack as non-executable prevent exploitation of this vulnerability? If yes, explain. If not, explain how an exploit would work.

```c
#include <stdio.h>
#include <string.h>

#define BUF_SIZE 256

int main( int argc, char* argv[] )
{

    if( argc != 3 ) {
        printf( "Error: give a string and its length \n" );
        return -1;
    }

    read_string( argv[1], atoi(argv[2]) );
}

void read_string( char* str, int len )
{
    char buf[BUF_SIZE];

    if( len > BUF_SIZE ) {
        printf( "Error: input string too long" );
    }
    else {
        strcpy( buf, str );
    }
    return;
}
```

Figure 1: Vulnerable program.

Question 7: ECB Mode [Security]

This problem is about modes of operation of block ciphers to achieve secure symmetric encryption. Let $E: \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$ be a block cipher that maps k-bit keys and an n-bit message block to an n-bit output. Assume n is even. You can assume E is a secure block cipher such as AES.

Part (a):

Explain (i) how encryption of an nm bit message for some integer m is performed using the standard ECB mode of operation and (ii) why standard ECB is insecure.

Part (b):

Now consider the following variant of ECB mode. For simplicity we will only define this mode of operation to work on messages of length $|M| = i \cdot (n/2)$ for some positive integer i. We denote by $\|$ concatenation of two bit strings. Assume K is a uniformly chosen from $\{0,1\}^k$ and kept secret.

```
Encrypt($K$,$M$)
Parse $M$ into $n/2$-bit blocks $M_1,\ldots,M_m$
Let $R$ be a random $n/2$-bit string
For $i = 1$ to $m$ do
    $C_1 = E(K, R\|M_i)$
Ret $C_1,\ldots,C_m$
```

(i) What kinds of leakage about plaintexts is admitted by standard ECB but not the variant? (ii) What type of information about plaintexts is still leaked with this variant?

Part (c):

Recall that the semantic security notion of Goldwasser and Micali requires that ciphertexts not leak even a single bit of information about plaintexts. Describe how to change selection of R in order to achieve semantic security (for chosen-plaintext attacks only) assuming n is large enough.

Part (d):

Explain why the resulting algorithm is still not very good, both from a performance and a security perspective.

Question 8: Policy automatons [Security]

A *security policy* is a finite-state automaton with calls and their arguments as the alphabet. Assume that there is a global Boolean semaphore L and the call `lock(L)` sets $L = 1$ and `unlock(L)` sets $L = 0$.

Part (a):

Give a diagram of a finite-state automaton with alphabet {`lock(L)`, `unlock(L)`} that corresponds to the following policy (assume that the initial value of $L = 0$):

> A call on `lock()` should only be allowed if $L = 0$

Part (b):

A *reference monitor* interposes a security policy between the application and the operating system. A system call is allowed if it satisfies the security policy. Assuming that the security policy is expressed as a finite-state automaton with system calls and their arguments as the alphabet, explain the operation of a reference monitor.

Part (c):

Now suppose that a security policy is expressed as a push-down automaton. (i) What kind of policies can you enforce? Provide an example to explain your answer. (ii) How does your answer to part (b) change?

Part (d):

Let the security policy be given as a finite-state automaton A. (i) Describe a static-analysis technique that, given a program P, determines whether executing P can result in a sequence of system calls that violates the security policy. (ii) How can you use your static analysis to optimize the reference monitor of part (b)?

Question 9: Key exchange [Security]

In a three-party key exchange protocol, a trusted server S shares keys K_A with a party A and K_B with a party B. Consider using a shortened Needham-Schroeder-type protocol to derive a key K and send an encrypted message M from A to B:

$$
\begin{aligned}
&A \rightarrow S : A, B, N_A \\
&S \rightarrow A : \{N_A, K, B, \{K, A\}_{K_B}\}_{K_A} \\
&A \rightarrow B : \{K, A\}_{K_B}, \{M\}_K
\end{aligned}
$$

Here N_A is a large, random nonce, the keys are chosen uniformly, and the curly braces denote use of a secure authenticated encryption on the enclosed message (encoded in some canonical fashion) and using the key indicated in the subscript.

Part (a):

Explain how B obtains the message M.

Part (b):

Consider an adversary that observes traffic between A and B and that can inject traffic into the network (e.g., a compromised router on the network path). What kind of an attack can such a network adversary mount?

Part (c):

We turn to defending against your attack from Part (b). The modification should only use authenticated encryption and nonce generation and completely prevent an attack by any network adversary. (i) Create a protocol which optimizes the latency of B decrypting an authenticated message M. (ii) Create a protocol that minimizes storage costs, in particular achieving $\mathcal{O}(1)$ storage on both A and B across many sessions.