Programming Languages and Compilers Qualifying Examination

Monday, February 4, 2013

This exam asked students to answer 4 out of 6 questions. This document contains the 5 questions that were answered.

Question 1: (Security Policies)

Part (a): 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 $L = 0$. Draw a finite-state automaton with alphabet $\{\texttt{lock(L)}, \texttt{unlock(L)}\}$ that corresponds to the following policy (assume that the initial value of $L = 0$):

> Call `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 in detail.

Part (c): If a security-policy is expressed as a push-down automaton, what are some examples of useful policies that you can enforce, and that could not be enforced using a finite-state automaton? Provide an example to explain your answer.

How does your answer to Part (b) change? (Include something about the relative efficiencies of the two kinds of reference monitors.)

Part (d): Let the security policy be given as a finite-state automaton A. 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.

How can you use your static-analysis to optimize the reference monitor of part (b)?

Question 2: (Register Allocation)

Most compilers start by assuming an unlimited supply of virtual registers, then (in the register-allocation phase) mapping virtual registers to physical registers.

Part (a): Describe three approaches to register allocation. For each, describe the steps that are performed, and illustrate the approach with a small example. Also say what happens to virtual registers that are not mapped to any physical register.

Part (b): Compare the approaches that you described in Part (a). What are the relative advantages and disadvantages of each?

Question 3: (Scanning)

Part (a): Regular expressions are routinely used to define lexical (token-level) syntax. Explain how regular expressions are translated into an executable form suitable for scanning. Then describe an algorithm for performing "maximal-munch tokenization". In maximal-munch tokenization, each time the scanner is called, its task is to find the longest prefix of the (remaining) input that is matched by one of the regular expressions. The scanner is to return the token name for the regular expression that was matched, along with the text that matched.

Part (b): Suppose that the tokens in our language are defined as follows:

Regular expression	Token name
abc	Token1
$(abc)^*d$	Token2

Consider an input sting of the form "$(abc)^m$" (i.e., m repetitions of the character sequence "abc"), and the repeated application of the algorithm that you gave in Part (a) until it "tokenizes" the input string (i.e., until all of the tokens in the input string have been recognized). Note that for such a string, the tokenization of the input string is Token1^m. In particular, each time the scanner is called, it recognizes the next occurrence of "abc" as the token Token1. At no stage does the scanner return Token2.

In terms of m, what is the running time of the algorithm that you gave in Part (a)? Explain your answer.

Part (c): Unless you did something particularly clever in Part (a), the running time of your algorithm is non-linear (in m). Some kinds of algorithms with non-linear running times can be turned into linear-time algorithms by means of *tabulation*. Explain how to modify your algorithm from Part (a) to use tabulation so that its running time for tokenizing an input string is always linear in the length of the input string. Note that the goal is to have a linear-time algorithm to tokenize the overall input string; however, the tokenizer could take more than $O(p)$ steps to recognize some token of size p.

(If for some reason tabulation would not work for your algorithm, give a naive algorithm for maximal-munch tokenization with non-linear running time on the example from Part (b), and explain how tabulation can be used to make its running time for tokenizing an input string always linear in the length of the input string.)

Part (d): Regular expressions are also used in interactive applications. Examples include the Unix shell and context searching in text editors. In these applications the goal is simply to see whether a particular string is in the language of a regular expression (not to do "maximal-munch tokenization"). It is easy to translate a regular expression to a non-deterministic finite automaton, but translating from there to a deterministic finite automaton may be too slow. What can be done instead to allow almost immediate execution and yet still process characters at reasonable speeds. What is the time complexity of your approach?

Question 4: (Delta Debugging)

To refresh your memory, the minimizing Delta Debugging algorithm, called *ddmin*, produces a minimal test case by simplifying a failing test case $r_{✘}$. We assume that $r_{✘}$ can be thought of as a succeeding test case $r_{✔}$ to which we have applied a set of changes $c_{✘} = \{\delta_1, \ldots, \delta_k\}$, where $\{\delta_1, \ldots, \delta_k\}$ is a set of elementary changes. The *ddmin* algorithm searches for a non-empty set of changes $c''_{✘}$ that, when applied to $r_{✔}$, produces a minimal failing test case. Thus, the answer is a set of changes $c''_{✘}$, such that $\emptyset \subset c''_{✘} \subseteq c_{✘}$.

There is a function $test_{r_{✔}}$ that can be called to test $r_{✔}$ with respect to a set of changes c:

$$test_{r_{✔}}(c) = \begin{cases} ✔ & \text{if } r_{✔} \text{ with changes } c \text{ applied passes} \\ ✘ & \text{if } r_{✔} \text{ with changes } c \text{ applied fails} \\ ? & \text{if } r_{✔} \text{ with changes } c \text{ applied is inconclusive} \end{cases}$$

In particular, $test_{r_{✔}}(\emptyset) = ✔$ and $test_{r_{✔}}(c_{✘}) = ✘$. Henceforth, we assume that $r_{✔}$ is fixed and abbreviate $test_{r_{✔}}$ as *test*.

The *ddmin* algorithm performs a recursive search on its current estimate ($c'_{✘}$) of the answer, using the helper function $ddmin_2$ given below. At each stage, $ddmin_2(c'_{✘}, n)$ partitions the current set $c'_{✘}$ into n pairwise-disjoint subsets of changes of approximately the same size. That is, $c'_{✘} = \Delta_1 \uplus \Delta_2 \uplus \cdots \uplus \Delta_n$, where for all Δ_i, $|\Delta_i| \approx |c'_{✘}|/n$.

A failure-inducing set of changes c is *1-minimal* if removing any single change δ_j from c causes the failure to disappear, i.e., for all $\delta_j \in c$, $test(c - \{\delta_j\}) \neq ✘$. The inputs to *ddmin* are $c_{✘}$ and (implicitly) *test*, such that $test(\emptyset) = ✔$ and $test(c_{✘}) = ✘$. The goal is to find $c''_{✘} = ddmin(c_{✘})$ such that $c''_{✘} \subseteq c_{✘}$, $test(c''_{✘}) = ✘$, and $c''_{✘}$ is 1-minimal. The minimizing Delta Debugging algorithm $ddmin(c_{✘})$ is defined as follows:

$$\begin{aligned} ddmin(c_{✘}) &= ddmin_2(c_{✘}, 2) \\ ddmin_2(c'_{✘}, n) &= \\ &\quad \text{let } \Delta_1 \uplus \Delta_2 \uplus \cdots \uplus \Delta_n \text{ be a partitioning of } c'_{✘} \text{ such that} \\ &\qquad \text{all } \Delta_i \text{ are pairwise disjoint, and } \forall \Delta_i \,.\, |\Delta_i| \approx |c'_{✘}|/n \text{ holds} \\ &\quad \text{in} \\ &\quad \text{let } \nabla_i = c'_{✘} - \Delta_i \text{ in} \\ &\quad \text{if } \exists i \in \{1, \ldots, n\} \text{ such that } test(\Delta_i) = ✘ \text{ then } ddmin_2(\Delta_i, 2) \\ &\quad \text{else if } \exists i \in \{1, \ldots, n\} \text{ such that } test(\nabla_i) = ✘ \text{ then } ddmin_2(\nabla_i, \max(n-1, 2)) \\ &\quad \text{else if } n < |c'_{✘}| \text{ then } ddmin_2(c'_{✘}, \min(|c'_{✘}|, 2n)) \\ &\quad \text{else } c'_{✘} \end{aligned}$$

The recursion invariant (and thus precondition) for $ddmin_2$ is $(test(c'_{✘}) = ✘) \wedge (n \leq |c'_{✘}|)$.

Part (a): 1-Minimality

Sketch a proof that the result computed by *ddmin* is always 1-minimal.

Part (b): Best-Case Complexity

What is the best-case asymptotic complexity of *ddmin*, stated in terms of the size of $c_{✘}$? Under what circumstances does this best case occur?

Part (c): Granularity

Suppose Delta Debugging is operating with a granularity of eight. We have a failure-inducing configuration $c'_{\text{✘}} = \Delta_1 \uplus \Delta_2 \uplus \cdots \uplus \Delta_8$. No single Δ_i fails, but $test(\nabla_3)$ does fail. Then ∇_3 is our new, smaller, failure-inducing configuration. This falls under case two of $ddmin_2$ as given above.

In this situation, $ddmin_2$ recursively continues minimizing ∇_3 with a granularity of seven: one smaller than before. An alternative would have been to reset the granularity to two, ensuring that we try to split ∇_3 roughly in half. Splitting in half tends to eliminate large groups of irrelevant changes quickly. So why is that *not* done here? Why does the second case of $ddmin_2$ use a granularity of $n-1$ (but at least two) instead of always using a granularity of two?

Part (d): Practical Considerations

To use Delta Debugging, one must devise a suitable representation for changes, configurations (sets of changes), and the testing function *test*. Describe the most significant practical challenges in creating these items for large, complex software systems. What properties must these items have, and what properties are desirable but perhaps not required? How does Delta Debugging fail (or degrade) if these properties are not upheld?

Question 5: (Def-Use and Use-Def Chains)

Part (a): Def-Use and Use-Def chains are essential data structures for many optimizations. Explain what information Def-Use and Use-Def chains encode.

Part (b): Outline how Def-Use and/or Use-Def chains can be used to solve two different problems arising in either a compiler's optimization phase or its back-end phase.

Part (c): Suppose the programming language has no procedure calls. Given the Use and Def sets for each node in a program's control-flow graph, how can Use-Def chains be computed for the program?

Part (d): This part refers to a language with the following characteristics:

- The language has parameterless procedure calls but no function calls.
- Procedures cannot be nested; any procedure other than the main procedure can be called by any other procedure.
- Variables declared in the main procedure are global to all procedures; there are no other global variables.
- A variable can be declared to be of type *procedure*. The *only* way procedure variables are assigned values is using assignment statements of the form: "*var* := *name*", where *name* is the name of some procedure; for example, "*x* := fib".
- Procedure calls are of the form: "call *name*", where *name* is the name of some procedure, or "call *var*", where *var* is a variable of type *procedure* (you may assume that no variable can have the same name as a procedure).

Suppose that we want to compute *inter*procedural Use-Def chains for the language described above; i.e., we want to match a definition of a global variable with its uses (where the definition and the uses may be in different procedures). Describe an algorithm for computing a safe approximation to these Use-Def chains (i.e., your algorithm may err on the side of computing too many chains, but never too few).