FALL 2007
COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN–MADISON
PH.D. QUALIFYING EXAMINATION

Programming Languages and Compilers
Qualifying Examimation

Monday, September 17, 2007
3:00 PM - 7:00 PM
Room 1263, Computer Sciences Building

GENERAL INSTRUCTIONS:

1. Answer each question in a separate book.
2. On the cover of *each* book indicate 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* 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.

SPECIFIC INSTRUCTIONS:

Answer 4 of 6 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: Array and Function Subtyping

Let τ represent some type. For any type τ, let $\tau[\,]$ represent the type of arrays whose elements all have type τ. For any pair of types τ_1, τ_2, let $\tau_1 \to \tau_2$ represent the type of functions from τ_1 to τ_2. Let *int* be the type of integers.

Let $\sqsubset$ be a binary relation representing strict subtyping, with $\sqsubseteq$ as its reflexive closure.

Part (a): Array Subtyping Gone Bad

Java and C# extend subtyping across array elements. That is, $\tau \sqsubseteq \tau' \implies \tau[\,] \sqsubseteq \tau'[\,]$. Write a fragment of Java or C# code which type checks according to this rule, but which will fail at run time due to an incorrectly-typed element appearing in an array.

Part (b): Elimination of Run-Time Checks

Java and C# add run-time checks to every array-element assignment in order to trap this sort of error and throw an exception instead. Propose a static program analysis and optimization that could safely eliminate some of these run-time checks. Describe your analysis in detail, including placing it in context with respect to well-known families of analysis techniques.

Part (c): Function Subtyping

What are the most flexible subtyping relations that can safely be permitted among function types? That is, what are the weakest conditions on τ_1, τ_2, τ_1', and τ_2' for which we can safely treat $\tau_1 \to \tau_2$ as a subtype of $\tau_1' \to \tau_2'$?

Justify the correctness of your answer. It may be useful in your arguments to treat types as mathematical sets with subtyping as subsetting.

Part (d): Arrays as Functions

Suppose we model array operations as a pair of functions: one for getting the value of an array element and one for setting the value of an array element. These functions must work for arrays of all types, and therefore are actually a polymorphic family of functions parameterized by array element type:

$$\begin{aligned} \mathit{get} &: \forall\tau\,.\,\tau[\,] \to \mathit{int} \to \tau \\ \mathit{set} &: \forall\tau\,.\,\tau[\,] \to \mathit{int} \to \tau \to \tau[\,] \end{aligned}$$

Note that *set* is treated as purely functional: it returns the updated array as its result.

Using the function subtyping relation developed in (c), explain why Java and C# array subtyping is safe for immutable arrays but may be unsafe for mutable arrays.

Question 2: Evaluation Strategies

Part (a): Basic Terms

Several evaluation strategies have been used to pass arguments into function calls. Briefly describe the following strategies, clearly identifying how they differ from each other:

- Call by value
- Call by value-result
- Call by reference
- Call by name

Part (b): Behavioral Differences

Write a small program which produces different behavior depending on whether it is evaluated using call by value, value-result, reference, or name. You may use any reasonable syntax, including pseudo-code, provided that your intent is clear. Give both the code and the expected behavior for each of the four evaluation strategies.

The four possibilities must be clearly, easily, and unambiguously distinguished: do not rely on minuscule timing differences.

Part (c): Functional Programming

In a purely functional programming language without destructive assignment, is it possible to distinguish call by value from call by name? Briefly justify your answer.

Part (d): Call by Need Versus Call by Name

Call by need is similar to call by name, but uses memoization to avoid evaluating any argument more than once. Under what circumstances would call by need perform *faster* than call by name? Under what circumstances would call by need perform *slower* than call by name?

Part (e): Implementing Call by Need

Propose an efficient implementation strategy for call by need. Describe how function arguments will be represented at run time. Identify key events during function evaluation and describe what your implementation does at these events.

"Efficient" can mean several things, depending on the resource to be conserved. Discuss what resources you have considered in your interpretation of "efficient" and explain what makes your proposal efficient with respect to these resources.

Question 3: Static Single Assignment

Part (a): Basics

What is Static Single Assignment (SSA) form?

Explain how a program involving only scalar variables can be placed in SSA form. Illustrate your technique on the following simple program.

```
a = b + 1;
while (a > 0) {
  if (b > 0)
    a = a + b;
  else
    a = a - b;
}
print(a, b);
```

Part (b): Copy Propagation

One of the advantages of SSA form is that it greatly simplifies the analysis necessary to implement various optimizations.

Explain what copy propagation is. If a program is not in SSA form, what analyses are necessary to determine whether copy propagation may be applied?

If a program is in SSA form, how are the rules of validity for copy propagation simplified? Illustrate your answer using the following code fragment, copy propagating the initial assignment to `a`:

```
a = b;
if (a > 0)
  a = a + 1;
else
  b = a + 1;
c = a;
```

Question 4: Garbage Collection

This question concerns the design and use of a garbage collector as part of a program's run-time system.

Part (a): With Strong Typing

Assume you have a language like Java, in which all pointers (references) are strongly typed (known at compile-time to reference a single fixed type). Outline the design of a garbage collector that may be run while the application program (the mutator) is stopped. Your collector should (of course) have the property that all objects marked as garbage (and deleted) really are garbage. Does the collector you present also have the property that all inaccessible objects are recognized and deleted?

Part (b): Without Strong Typing

Languages like C and C++ complicate garbage collection in that pointers may be difficult to recognize because of casting and type unions. What changes are needed in the garbage collector you proposed in (a) to support languages like C and C++?

Part (c): Concurrency

Some languages, like Java, allow simultaneous execution of multiple threads. On processors that can execute more than one thread simultaneously, it may be important to make a garbage collector exhibit a degree of concurrency; i.e., if there is more than one thread, at least one should continue to run during garbage collection. What changes are needed for your garbage collector to make it concurrent?

Question 5: Security Policies and Monitors

Part (a): Policy Automata

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

> Call `lock(L)` should only be allowed if $L = 0$.

Part (b): Reference Monitors

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): Static Analysis

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.

Question 6: Function Fixed Points

Let S be a finite set of size n. Let $F : 2^S \to 2^S$ be a function.[1] A set $X \in 2^S$ is called a fixed point of a function F iff $F(X) = X$. Define $\mu X.F(X)$ as the least fixed point of F and $\nu X.F(X)$ as the greatest fixed point of F.

Part (a): Existence

What conditions should F satisfy so that $\mu X.F(X)$ exists? What conditions should F satisfy so that $\nu X.F(X)$ exists?

Part (b): Relationships

Assume that $\mu X.F(X)$. Define $G(X)$ as $\neg F(\neg X)$. Recall that given a set $X \subseteq S$, $\neg X$ is equal to $S - X$. What is the relationship between $\mu X.F(X)$ and $\nu X.G(X)$.

Part (c): Computation

Give an iterative algorithm to compute $\mu X.F(X)$. Analyze the time complexity of your algorithm.

[1] Recall that 2^S is the power-set of S.