

# Final Exam

CMSC 838Y  
Agile and Adaptive Programming Systems  
Spring 2003

May 21, 2003

## Guidelines

The exam time is from **1:30 pm to 3:30 pm**, totalling 2 hours.

This exam has 11 pages; count them to make sure have all of them. Put your name on each page before starting the exam. Each question may have multiple parts; write your answers on the exam sheets, using the back of the page as necessary. If you finish early, you may bring your exam to the front, but please be as quiet as possible. Use good test-taking skills—don't spend too much time on one question (particularly one worth a small number of points), and do the questions that seem easiest first.

If you have a question, raise your hand and I will come to you. However, please use discretion to minimize interruptions. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

Good luck, and enjoy your summer!

Question	Points	Score
1	25	
2	30	
3	25	
4	20	
Total	100	

## 1. Formal Semantics and Dynamic Updating (25 points)

The paper by Gupta et al. formalizes on-line software upgrades, and some conditions for their *validity*. A program is described as a pair  $(\Pi, s)$  in which  $\Pi$  is the program text, and  $s$  is its current state, including the stack, program counter, heap, etc. An on-line upgrade consists of a pair  $(\Pi', S)$  where  $\Pi'$  is the new code, and  $S$  is the *state transformer function* that maps the state  $s$  of the running program to a new state  $s'$  to be used by the new program.

a) (15 points) One form of valid update is a *functional enhancement*, and is defined as follows:

The function  $f'$  is a functional enhancement of function  $f$  with respect to a state transformer function  $S$  if the following conditions hold. Let  $s_1$  be a reachable (legal) state of  $\Pi$  for which  $S$  is defined and the statement to be executed next in  $s_1$  is a call to  $f$ . If in a process executing  $\Pi$ ,  $f$  is called in state  $s_1$  and the resultant state just after it returns is  $s_2$ , then if  $f'$  is called in a process executing  $\Pi'$  in state  $S(s_1)$ , the state just after its return *can* be  $S(s_2)$  for some inputs.

Say you have the program  $\Pi$ , shown in Figure 1(a), and you want to update it to the program  $\Pi'$ , shown in Figure 1(b), with the changed parts boxed. Write a state transformer function  $S$  such that the new versions of `init` and `adj` are functional enhancements of their old versions with respect to  $S$ . Note that  $S$  takes the *entire program state* as input. Also, updates are not allowed to functions that are currently running. The possible program counter values (e.g. `i1`, `i2`, `a1`, etc.) label the program statements.

b) (10 points) Gupta et al.'s formalism abstracts away from the details of any particular programming language. What would be the difficulty in mapping their formalism to the traditional operational formalization of the lambda calculus (see the appendix at the end of the exam)? That is, if you were to map Gupta et al.'s notion of process to a lambda-calculus program, what would be lost in the translation? (Think about how Gupta et al.'s notion of functional enhancement would appear in the translation, for example.) For extra credit (5 points), speculate on how you might fix the problems you describe.

---

<pre> struct f { int x; int y; }; struct f g; void init() { i1  g.x = 1; i2  g.y = 2; } void adj(int d) { a1  g.x += d; a2  g.y += d; } void main() { m1  init(); m2  adj(5); m3  ... } </pre>	<pre> struct f { int x; int y; <span style="border: 1px solid black; padding: 2px;">int z;</span> }; struct f g; void init() { i1  g.x = 1; i2  g.y = 2; i3  <span style="border: 1px solid black; padding: 2px;">g.z = 3;</span> } void adj(int d) { a1  g.x += d; a2  g.y += d; a3  <span style="border: 1px solid black; padding: 2px;">g.z -= d; // minus</span> } void main() { m1  init(); m2  adj(5); m3  ... } </pre>
--	---

(a) Program  $\Pi$

(b) Program  $\Pi'$

Figure 1: An old program and the program portion of its upgrade

Your answers: **Answer:**

- a) *Because the stack is always empty for legal updates, and there is no program heap, the state transformer  $S$  is indexed by the PC, and is defined as follows:*

```
void S(unsigned int PC) {
    switch (PC) {
        case i1:
        case i2:
        case a1:
        case a2: fail(); // update not allowed
        case m1: break; // nothing to do
        case m2:
            new(g) = old(g); new(g.z) = 3; break;
        case m3:
            new(g) = old(g); new(g.z) = -2; break;
    }
}
```

*Note that  $\mathit{new}(x)$  refers to the variable  $x$  in the new program, while  $\mathit{old}(x)$  refers to the variable  $x$  in the old program.*

- b) *The problem is that the notion of program and state are not separate in the lambda calculus—computation is described as a rewriting relation between expressions, which encapsulate both code and state. Therefore, a process  $(\Pi, s)$  would map to a single lambda-calculus expression, and an update  $(\Pi', S)$  would map to a function from expressions to expressions, transforming both code and state at the same time. Mapping Gupta et al.'s idea of a functional enhancement would be difficult because it's not even clear which function is running at the time of the update, and it's not clear how to characterize the program's state before and after the function is executed. You could fix this problem by writing a semantics that includes a program that binds names to function definitions, and modeling a stack with evaluation contexts in the style of Wallach and Rudys paper on language-based termination. This would allow you to identify which function is currently running, and which other functions are on the stack. However, this only models the control stack, not the value stack, which is still embedded as part of the expression. Therefore, state transformer functions would either have to be prevented from operating on values on the stack, or the stack would have to be extended to include bindings from a function's formal parameters to its arguments. Access to these arguments would have to be by name, rather than by substitution.*

2. **Modules** (30 points)

- a) (10 points) *Inheritance*, *subtyping*, and *delegation* are three important concepts in object-oriented programming languages. Define each one.

**Answer:**

*If type  $T$  is a subtype of  $T2$ , then any object of type  $T$  can be used in a context that expects an object of type  $T2$ . If object  $x$  inherits from another object  $y$ , it shares some or all of  $y$ 's implementation. When an object  $x$  delegates to an object  $y$ , some services (like methods) invoked on  $x$  are forwarded to  $y$ .*

- b) (10 points) In mainstream object-oriented languages like C++ and Java, these three concepts are often tied into a single mechanism, but two of our considered languages separate at least one from the other two. Explain through which mainstream languages merge together inheritance, subtyping, and delegation, and then name the aforementioned two languages, and how they separate these concepts in some way.

**Answer:**

*The Java class mechanism combines all three. That is, when class  $A$  extends class  $B$ , it inherits  $B$ 's implementation, and becomes a subtype of  $B$ . Moreover, when  $A$  is invoked with a method  $m$  that it does not implement, it delegates to its parent object of class  $B$ .*

*The language Lava provides an explicit construct for delegation. A object can define an instance variable as a delegate, and if it receives a method call that it doesn't understand, it will attempt to forward the call to its delegates.*

*The language Moby separates inheritance and subtyping. In particular, subtyping is performed structurally, rather than by declaration. This means that as long as object  $x$  implements the same methods (or strictly more) as object  $y$ , and these methods are themselves subtypes of  $y$ 's methods, then object  $x$  can be used where  $y$  is expected. This is analogous to Java's interfaces, but no explicit declaration is required.*

- c) (10 points) In Java, there is a way to separate inheritance from subtyping (that is, there is a way to define objects in which one is a subtype of the other, but does not inherit from it). Give an example that shows this.

**Answer:**

*Interfaces describe subtyping relationships that don't involve code sharing. For example, the `NamingEnumeration` interface is a subtype of `Enumeration`:*

```
interface NamingEnumeration extends Enumeration
    void close();
    boolean hasMore();
    Object next();
```

```
interface Enumeration
    boolean hasMoreElements();
    Object nextElement();
```

*A class `C` that implements `NamingEnumeration` can be used in any context expecting an `Enumeration`, whether or not `C` inherits from an `Enumeration`-implementing class.*

### 3. Extensible Systems (25 points)

Hardware-based virtual memory provides protection in most OS's. By creating separate address spaces, it is impossible for one application to directly access the memory of another, or for an application to directly access kernel memory. In addition, separating address spaces creates reasonably distinct boundaries between applications and the kernel, simplifying the tasks of resource accounting, isolation, and policy enforcement.

However, separating application address spaces has two major drawbacks:

1. There can be no direct sharing between processes (i.e. syscalls and IPC are used for communication, and this requires serialization of application datastructures in many cases).
2. Context-switching costs are tremendous (e.g. 2 orders of magnitude more than a procedure call is typical).

It has been proposed that if an operating system is written in a programming language having strong safety properties, with the cornerstone being type-safety, hardware-based techniques for providing protection can be avoided. Ideally, assuming such a programming language is used, application *extensions* can be loaded directly into the kernel without need of hardware protection.

- a) (5 points) Type-safety is useful in implementing (partial) isolation between application components and the kernel, or other applications. Explain why this is so.

**Answer:**

*When a language is type-safe, it is impossible to 'manufacture' an arbitrary pointer, say by casting it from an integer. As a result, an application can only access memory using pointers that it has been given through legal means, such as from the memory manager or the kernel (this approximates the idea of a capability in traditional operating systems). This means that an application cannot access the memory of another application unless that application provides it a pointer to do so.*

- b) (15 points) Explain why type-safety alone is unhelpful in implementing fair sharing of the CPU and memory. Describe how the language-based operating systems SPIN and KaffeOS attempt to fairly share CPU and memory resources via accounting or some other means.

**Answer:**

*The issue at the core of resource accounting is sharing. For example, two applications might have a pointer to the same memory location—who ‘owns’ that memory location? Type safety eliminates surreptitious means to sharing (e.g. by manufacturing pointers), but doesn’t eliminate legal means. Similarly, if a kernel consists of a pool of extensions to be run on behalf of a variety of users, to which user should the extension’s use be charged?*

*Hardware-enforced resource accounting is simple because (a) each application is given its own periodic CPU context, and (b) its own, unshared memory. System services (like context-switching, network access, etc.) are typically charged to the application that employs them. For each of the above systems:*

**KaffeOS** *Fair sharing is implemented via accounting. KaffeOS processes each have their own separate heap, simulating a hardware-enforced process heap. Sharing is enabled through explicit, shared heaps, in which accounting of shared memory is divided amongst the owners of (those who have pointers to) the resource. CPU accounting is done by carefully tracking application thread running times, and times the system runs on an application’s behalf. In particular, garbage collecting an application heap is done in isolation, and is charged to the owning application. Non-preemptible system functions that cannot be charged to a given application are kept to a minimum.*

**SPIN** *SPIN does not evidently perform general accounting of the resource usage of extensions, whether for their memory use or CPU use, although it could be coded up using the extension mechanism. For example, guards could be inserted that spawn threads to handle the events and track their resource use. User-space applications are accounted as usual. Fair sharing of resources is only at the level of preventing denial-of-service attacks. SPIN has the notion of an ephemeral handler that can be safely killed at any time. Untrusted, ephemeral extension handlers are run in their own thread, and can be killed if they exhaust a CPU quota. Ephemeral handlers cannot allocate memory. This prevents a handler from using too many resources without harming system integrity.*

- c) (5 points) If a process exceeds its resource quota (i.e. it begins misbehaving), it should be terminated, but without compromising the integrity of the kernel or other applications. Why is this difficult in a language-based system?

**Answer:**

*The difficulty arises due to sharing of datastructures between processes or between a process and the kernel, which is in general not possible (though not impossible) in hardware-based approaches. In particular, abruptly terminating a process might result in shared datastructures being left in an inconsistent state, locks not being released, or tasks only partially completed, thus violating invariants expected by the system. Past approaches have addressed this problem by preventing or limiting data sharing or lock acquiry, using transactions so as to roll back the terminated process, and/or by having a sounds means to implement revocation of shared pointers.*

- (extra credit) (5 points) Speculate on why language-based operating systems have not “taken off” in the mainstream. Indicate at least two technical (rather than social) problems you see.

**Answer:**

*Some possible technical reasons would be:*

- (a) Resource accounting; these are the problems mentioned in this question. In particular, asynchronous termination and “feature interaction” among in-kernel processes/handlers are still problematic.*
- (b) Performance; often type-safety comes at the cost of run-time checks, and/or garbage collection, which can add overhead.*
- (c) Legacy code; clearly, writing an operating system from scratch would be a daunting task, so some way of interacting safely with legacy code is needed. In general, there hasn’t been much research in multi-lingual programming.*

#### 4. Dynamic Code Generation (20 points)

Recall that run-time specialization is a process by which specialized versions of functions are created at run-time based on parameters that are expected to be constant. The following code implements the modular exponent function in C, and includes annotations developed for the *DyC* project: *dynamicRegion* indicates that its argument is expected to be constant at run-time, and *unrolled* indicates that if the loop bound is determined to be run-time constant, the loop should be unrolled:

```
int mexp(int base, int exp, int mod) {
    dynamicRegion (exp) {
        dynamicRegion (mod) {
            int s, t, u;

            s = 1; t = base; u = exp;

            unrolled while (u != 0) {
                if ((u&1) != 0)
                    s = (s+1) % mod;
                t = t*t % mod;
                u >>= 1;
            }

            return s;
        }
    }
}
```

- a) (5 points) Given that `exp` and `mod` will be constant at run-time, draw a box around each statement or expression in the code that will also be constant at run-time as a result.

**Answer:**

```
int mexp(int base, int exp, int mod) {
    dynamicRegion (exp) {
        dynamicRegion (mod) {
            int s, t, u;

            s = 1; t = base; u = exp;

            unrolled while (u != 0) {
                if ((u&1) != 0)
                    s = (s+1) % mod;
                t = t*t % mod;
                u >>= 1;
            }

            return s;
        }
    }
}
```

- b) (10 points) Show what the code for the inner dynamic region (that is, the block inside the **dynamicRegion** (mod) part) will look like after it is generated at run-time when `exp` is 2 and `mod` is 1234. Show the result in C, not in assembly.

**Answer:**

*You basically need to unroll the loop:*

```
int s, t;

s = 1; t = base;

t = (t*t) % 1234;
s = (s*t) % 1234;
t = (t*t) % 1234;

return s;
```

*However, this could be further optimized, yielding:*

```
int t;

t = base;

t = (t*t) % 1234;
s = t % 1234;

return s;
```

- c) (5 points) Compare run-time specialization, as implemented in the above examples, with more general dynamic compilation, as occurs in the Jalapeno compiler, for example.

**Answer:**

*Dynamic compilation performs arbitrary compilation from a low-level “source” language to machine code, as with JVM code to machine code. This typically follows a classic compiler approach of compiling to intermediate languages and ultimately to machine code. Run-time specialization takes a machine-code program and generates and links in fresh machine code by specializing routines based on values that are learned to be constant at run-time. Specializations are “prepared” in advance by compiling to a series of templates which are then “stitched” together at run-time when values become known. In both cases, because the task of compilation/code-generation occurs at run-time, it must be as fast as possible, since any time taken slows down the program.*

## Appendix: Call-by-value Lambda-calculus

### Syntax

*Expressions*  $e ::= i \mid \lambda x:T.e \mid e_1 e_2$   
*Values*  $v ::= i \mid \lambda x:T.e$   
*Types*  $T ::= \text{int} \mid T_1 \rightarrow T_2$

### Operational Semantics

$$(\lambda x:T.e) v \mapsto \{v/x\}e$$
$$\frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2}$$
$$\frac{e_2 \mapsto e_2'}{(\lambda x:T.e) e_2 \mapsto (\lambda x:T.e) e_2'}$$

Recall that  $\{v/x\}e$  denotes the capture-avoiding substitution of all occurrences of the variable  $x$  occurring in  $e$  with  $v$ . For example, given  $e$  equals  $\lambda x:\text{int}.\lambda y:\text{int}.z + y + x$ , the result of performing  $\{7/z\}e$  would be  $\lambda x:\text{int}.\lambda y:\text{int}.7 + y + x$ .