

# Final Exam

CMSC 838Z, Spring 2004

*Due Wednesday, May 19, 12:35PM AVW 4129*

This is a take-home exam. You have 24 hours to complete it. Unless otherwise specified, please write or print your answers on separate sheets of paper, with each problem clearly marked. The first problem involves writing some code, and explains how that should be turned in. Please turn in your answers to Jeff Foster at 12:35PM (he may be a few minutes late) in AVW 4129.

You may refer to any of the materials that we covered in class; i.e., those handouts, books, chapters, etc. linked from the course schedule page. Do not refer to any other materials. Do not talk to your fellow students or anyone else about the exam. If you think you have found an error, answer the question to the best of your ability given the error and explain in your answer what is wrong. Alternately, contact me to see if it really is an error. If you find an answer to a question in the allowed course materials (not likely), answer the question and cite your source.

Good luck!

Question	Points	Score
1	25	
2	15	
3	15	
4	25	
5	20	
Total	100	

1. (Universal and Existential Types, 25 points)

I have provided some Cyclone code<sup>1</sup> that partially implements sets of integers. There are two implementations; one implements a set as an existential type `IntSetE`, which allows the implementation type to be abstract, and the other as a parameterized type `IntSetA<a>`, where `a` is the datastructure used to implement the set. Using this code as a starting point, complete the following two tasks:

- (a) (10 points) Implement the missing `add` and `is_member` functions that are marked toward the bottom of the file. After doing this, you should be able to completely compile the file to an executable and run it, yielding `Test Succeeded!`.
- (b) (10 points) Change `IntSetA` type to be `SetA` instead, changing its type annotations as necessary (no new data fields need be added) so that it can store elements arbitrary type, rather than just integers. You can assume that elements of the set are comparable via physical equality (`==`), and that sets will still be homogeneous (that is, every element they contain will be of the same type). Finally, this change should not affect any of the code you wrote for `IntSetE`. That is, it should still be able to share the new polymorphic functions that you write with the `IntSetA` code, as before.

Please attach printouts of code, highlighting the parts that you changed or wrote from scratch. Also, e-mail me the entirety of the code, so I can easily run it.

- (c) (5 points) Finally, consider how you would change `IntSetE` to similarly work for arbitrary set elements. Why is this problem difficult? You might wish to try implementing this first, to see what problems you run into.

---

<sup>1</sup><http://www.cs.umd.edu/class/spring2004/cmsc838z/exams/intset.cyc>

2. (Lock and Alias Types, 15 points)

In Project 3<sup>2</sup>, problem 2, you explained how you would modify the rules in the Flanagan and Abadi type system<sup>3</sup> to be flow-sensitive. That is, to use separate primitives for acquiring and releasing a lock rather than a single `sync` statement. This was accomplished by changing judgments from the form  $E; p \vdash e : t$ , where  $p$  is the *permission* for the expression  $e$ , to have the form  $E; p \vdash e : t; p'$ . That is, they now have an *input permission*  $p$  prior to evaluating  $e$  and an *output permission*  $p'$  following its evaluation.

Alias types<sup>4</sup> also tracks a flow-sensitive property: the shape of the “store” after the execution of each instruction. However, rather than having an “input store” and an “output store” in each judgment, the authors use just the “input store” in combination with continuation-passing style (CPS). In this formulation, functions never return, but instead either stop (by invoking the `halt` instruction), or invoke a *continuation* function, which is passed as an argument to the current function. This is similar to the return address pushed on the stack to implement call/return semantics in stack-based architectures.

Change the *syntax* and *typing rules* of your modified Flanagan and Abadi monomorphic system<sup>5</sup> to use CPS instead. Be sure to include all the relevant rules for your modified system as well as those that must be added or changed to support CPS.

---

<sup>2</sup><http://www.cs.umd.edu/class/spring2004/cmsc838z/projects/p3.pdf>

<sup>3</sup>Figs. 1 and 2 in the paper at <http://research.compaq.com/SRC/personal/flanagan/papers/esop99.ps>

<sup>4</sup><http://www.cs.cornell.edu/talc/papers/alias.pdf>

<sup>5</sup>Figures 1 and 2 in the paper, with sequencing, `acquire`, and `release`, and no `sync` statement.

3. (Model Checking, 15 points) The grammar for Computation Tree Logic (CTL) is given below:

$$\begin{aligned} \phi ::= & \perp \mid \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \text{AX}\phi \mid \text{EX}\phi \\ & \mid \text{A}[\phi \text{ U } \phi] \mid \text{E}[\phi \text{ U } \phi] \mid \text{AG}\phi \mid \text{EG}\phi \mid \text{AF}\phi \mid \text{EF}\phi \end{aligned}$$

The meaning of the temporal operators AU, EU, AG, EG, AF and EF is typically defined to be such that “the present includes the future.” For example, EF  $p$  is true for a state  $s$  if  $p$  is true for state  $s$  already. Often one would like corresponding operators such that *the future excludes the present*. Define six modified temporal operators—AG', EG' etc—that refer only to future states and not the present one. Define them in terms of the existing connectives in the above grammar.

4. (Theorem Proving, 25 points)

- (a) (10 points) The syntax of Mini-ML, shown on page 10 of **Computation and Deduction**, contains pairs, and the primitives **fst** and **snd** for extracting the first and second elements of a pair, respectively. Say we remove these two primitives and replaced them with a pattern matching facility instead:

$$e ::= \dots \mid \mathbf{split} \ e_1 \ \mathbf{as} \ (x_1, x_2) \Rightarrow e_2$$

Here, **split** takes a pair  $e_1$  as its argument, and binds the first element of the pair to  $x_1$  and the second to  $x_2$  within the expression  $e_2$ .

Write the operational semantics rule for **split**. (The Mini-ML operational semantics rules for pairs as they stand are given on page 15 (Section 2.3) in **Computation and Deduction**). Then write the encoding for the syntax and semantics of **split** in LF.

- (b) (15 points) Consider the following program  $P$ :

```
y=0;
while (y != x) {
  y = y + 1;
}
```

Prove that, assuming  $x \geq 0$ , executing  $P$  will result in  $x = y$ . Do this in two ways:

- i. Use weakest preconditions. Show your proof tableau.
- ii. Use verification conditions, following what you did in the project. Show the verification condition that you generate, and explain how you would call a theorem prover using it.
- iii. What are the differences in the two techniques? What is the reason behind them?

5. (Software Quality: State-of-the-Art, 20 points)

- (a) (10 points) There were three main static techniques underlying the tools that we considered this semester: type-checking, theorem-proving, and model-checking. Compare and contrast them, in terms of their underlying mathematics and the contexts in which they are (or can be) used. Be crisp and precise. Use mathematics as much as possible to express your point. (There is not one right answer!)
- (b) (10 points) Name what you believe to be two outstanding problems in using automated techniques for improving code quality, based on your insight and experience gained from this class. Describe any ideas you have for solving the problems and/or the directions you would take in trying to solve them.