15-312 Foundations of Programming Languages

Midterm Examination

October 16, 2003

Name: _____

Andrew User ID: _____

- This is a closed-book exam; only one double-sided sheet of notes is permitted.
- Write your answer legibly in the space provided.
- There are 11 pages in this exam, including 3 worksheets.
- It consists of 3 questions worth a total of 100 points.
- You have 85 minutes for this exam.

Problem 1	Problem 2	Problem 3	Total
45	**30**	**25**	**100**

1. Static and Dynamic Semantics (45 pts)

So far in this course, expressions have been decorated with types that have been maintained in the operational semantics. This may tempt us to add a typecase construct to the language. In this question we consider a particularly simple version of typecase that allows us to recognize base types, but not compound types. For example, in a version of MinML with primitive types bool and string (omitting integers for simplicity) we might write a generic function

```
toString : All t => t -> string
= Fn t => fn x:t =>
  typecase t
        of bool => (if x then "true" else "false")
         | string => x
         | _ => "<unprintable>"
```

The general form of typecase is

typecase τ of bool => e_1 | string => e_2 | _ => e_3

with abstract syntax typecase(τ, e_1, e_2, e_3).

1. (15 pts) Show the typing rule(s) for typecase. Make sure that the above example is type-correct with your rule(s). **[Hint: you may need more than one rule.]**

2. (10 pts) Show the new transition rule(s) for typecase in a structured operational semantics.

3. (5 pts) State the type preservation theorem in its form appropriate for the judgments considered here. It is proved by rule induction over which derivation?

4. (10 pts) Show the case(s) for typecase in the proof of the type preservation theorem.

5. (5 pts) In a language with existential types, typecase breaks data abstraction, that is, client code can sometimes tell the implementation of a supposedly abstract type. Demonstrate this with a simple example.

2. Environments and Closures (30 pts)

We have focused on a small-step operational semantics as a way of specifying the meaning of program constructs. However, early in the course, we discussed an alternative called big-step semantics. In a big-step semantics, a single judgment describes how an expression evaluates all the way down to a value.

Consider the following big-step environment-based semantics for a language with functions and integers. The judgments are of the form

$$\eta \vdash e \Downarrow v$$

which can be read, "In the environment η, expression e evaluates to value v." The rules below are actually unsound; your job will be to find out why and fix the problem.

$$\frac{}{\eta \vdash \mathtt{num}(k) \Downarrow \mathtt{num}(k)}$$

$$\frac{}{\eta, x{=}v, \eta' \vdash x \Downarrow v}$$

$$\frac{}{\eta \vdash \mathtt{fn}(\tau, x.e) \Downarrow \mathtt{fn}(\tau, x.e)}$$

$$\frac{\eta \vdash e_1 \Downarrow \mathtt{fn}(\tau_2, x.e_1') \quad \eta \vdash e_2 \Downarrow v_2 \quad \eta,\ x{=}v_2 \vdash e_1' \Downarrow v}{\eta \vdash \mathtt{apply}(e_1, e_2) \Downarrow v}$$

1. (5 pts) State the preservation theorem in the form appropriate for the rules given above. That is, state the preservation property that these rules *should* have if they were correct. You may assume that expression typing is the same as in MinML.

2. (10 pts) Give a program that, when executed using the rules above, would violate type preservation.

3. (15 pts) Show how to fix the rules to ensure soundness. Your solution should obey the preservation theorem you stated in part 1, although there is no need to prove this. Your solution should retain the environment-based, big-step character of the semantics, i.e. do not use a substitution $\{v/x\}e$ to express what happens at a function call.

3. Recursive Types (25 pts)

Consider the ML data type

```
datatype U = Int of int | Fun of U -> U
```

1. (5 pts) Give a definition of `U` as a recursive type using the $\mu t.\tau$ construct.

2. (5 pts) Give the definition of the constructors `Int` and `Fun`.

3. (5 pts) On your representation, write a function `apply` : `U` $\rightarrow$ `U` $\rightarrow$ `U` that applies a function to its argument or raises an exception `Error` if the first argument does not represent a function (in which case it must represent an integer).

4. (5 pts) Give a definition of `U` as an abstract type $\exists t.\tau$. Your operations on the abstract type should just be the two constructors and a destructor in the form of a case expression for `U`. You do not need to give an implementation of the abstract type.

5. (5 pts) Given a new implementation of `apply` in the scope of an `open` of `U` naming the abstract type t and its implementation x, or explain why such an implementation is impossible. Note that in this context, `apply` would have type `t` $\rightarrow$ `t` $\rightarrow$ `t` rather than `U` $\rightarrow$ `U` $\rightarrow$ `U`.

Worksheet

Worksheet

Worksheet