15-312 Foundations of Programming Languages

Midterm Examination

March 4, 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 12 pages in this exam, including 3 worksheets.
- It consists of 3 questions worth a total of 100 points and one extra credit question worth 20 points.
- The extra credit is recorded separately, so only attempt it after you have completed all other questions.
- You have 85 minutes for this exam.

Problem 1	**Problem 2**	**Problem 3**	**Total**	**EC**
50	**30**	**20**	**100**	**20**

1. Operational Semantics and Type Safety (50 pts)

We can add support for lists to MinML by adding the following syntax and typing rules:

$$\begin{array}{lcl} \tau & ::= & \cdots \mid \tau\ \texttt{list} \\ e & ::= & \cdots \mid \texttt{nil} \mid e_1 :: e_2 \mid \texttt{listcase}(e, e_1, x\,y.e_2) \end{array}$$

(As in assignment 3, $\texttt{listcase}(e, e_1, x\,y.e_2)$ is concise notation for `case` e `of nil =>` e_1 `|` x`::`y `=>` e_2.)

$$\frac{}{\Gamma \vdash \texttt{nil} : \tau\ \texttt{list}} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau\ \texttt{list}}{\Gamma \vdash (e_1 :: e_2) : \tau\ \texttt{list}}$$

$$\frac{\Gamma \vdash e : \tau\ \texttt{list} \quad \Gamma \vdash e_1 : \tau' \quad \Gamma, x : \tau, y : \tau\ \texttt{list} \vdash e_2 : \tau'}{\Gamma \vdash \texttt{listcase}(e, e_1, x\,y.e_2) : \tau'}$$

Note: In what follows, your answers should be consistent with a *call-by-value* language. Lists should **not** be lazy.

1. (7 pts) Extend the syntactic definition of values below to include appropriate value forms for lists.

$$v \ ::= \ \cdots \mid$$

2. (15 pts) Give all the evaluation rules for these constructs for a small-step operational semantics.

3. (8 pts) State a canonical forms lemma appropriate for lists. You do not need to prove it.

4. (5 pts) State the progress lemma.

5. (15 pts) Show the cases for the proof of the progress lemma pertaining to the list constructs.

(Extra space.)

2. Polymorphism and Derived Forms (30 pts)

Often one programming construct can be implemented in terms of another. For example, we can define a `let` construct as follows:

$$\texttt{let}\ x : \tau = e_1\ \texttt{in}\ e_2 \stackrel{\text{def}}{=} (\lambda x{:}\tau.e_2)\ e_1$$

This implementation works because function application in MinML is call-by-value: First, the function is evaluated (which takes zero steps, since it is already a value), then the argument e_1 is evaluated, then e_1's value is substituted for x in e_2. In this problem you will implement some more sophisticated derived forms called *Church encodings* (named for Alonzo Church, the inventor of the lambda calculus).

The type `bool` need not be primitive in PolyMinML; it can be implemented using polymorphism. Consider the definitions:

$$\begin{array}{lcl}
\texttt{bool} & \stackrel{\text{def}}{=} & \forall\alpha.\alpha \to \alpha \to \alpha \\
\texttt{true} & \stackrel{\text{def}}{=} & \Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha.x \\
\texttt{false} & \stackrel{\text{def}}{=} & \Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha.y \\
\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \stackrel{\text{def}}{=} & e_1\ [\tau]\ e_2\ e_3 \\
 & & \text{(where } e_2 \text{ and } e_3 \text{ have type } \tau)
\end{array}$$

Observe that `true` and `false` have the type `bool`, as desired, and that if e_1 has the type `bool`, and e_2 and e_3 have the type τ, then `if` e_1 `then` e_2 `else` e_3 has the type τ, as desired.

1. (5 pts) Explain in words why these definitions are reasonable.

2. (5 pts) The implementation of `if` does not quite match the operational semantics of `if` in MinML. Why not?

3. (5 pts) Give a revised definition of `if` that corrects the problem. Do not change the definitions of `bool`, `true`, or `false`. Like the unrevised definitions, your solution should use only functions (polymorphic and/or regular) and function application.

4. (15 pts) The type `bool` is very much like a degenerate sum type in which neither arm of the sum carries any information (instead, all the information is given simply by which arm is in use). Using this intuition, we can extend the Church encoding of booleans to sums:

$$\tau_1 + \tau_2 \overset{\text{def}}{=} \forall\alpha.(\tau_1 \to \alpha) \to (\tau_2 \to \alpha) \to \alpha$$

Give definitions that implement:

(a) $\texttt{inl}_{\tau_1+\tau_2}\, e$

(b) $\texttt{inr}_{\tau_1+\tau_2}\, e$

(c) $\texttt{case}(e_1, x.e_2, x.e_3)$
(You may assume that e_1 has type $\tau_1 + \tau_2$, and that e_2 and e_3 have the type τ.)

5. (20 pts EXTRA CREDIT) Give a Church encoding for product types, and give definitions that implement the pairing and projection operations.

3. Abstract Machines (20 pts)

The abstract machine interpreter from assignment four was inefficient in its dealing with exceptions, because in the event of failure it had to peel frames off the stack one-by-one looking for a handler. A more efficient abstract machine would keep track of the nearest enclosing handler and jump to it immediately in the case of failure.

The following code implements this idea:

```
(* eval : exp -> (unit -> 'a) -> (exp -> 'a) -> 'a *)
fun eval (v as Fun _) fk k = k v
  | eval (Apply(e1, e2)) fk k =
       eval e1 fk
       (fn v1 => eval e2 fk
                 (fn v2 => eval (applyFun (v1, v2)) fk k))
  | eval (v as Fail _) fk k = fk ()
  | eval (Try(e1, e2)) fk k =
       eval e1 (fn () => eval e2 fk k) k
```

Each arm takes two continuations: the regular continuation `k`, and an *failure continuation* `fk`. The failure continuation is the prefix of the regular continuation that ends at the innermost failure handler. A `try` expression installs a new failure continuation before evaluating its first sub-expression. A `fail` expression invokes the failure continuation directly, bypassing any extra frames in the regular expression.

In this problem you will formulate a formal abstract machine that expresses what this code is doing. Your new abstract machine will have the following syntax for states and stacks:

$$
\begin{array}{lrcl}
\text{(states)} & s & ::= & k_{\mathit{fail}}; k > e \mid k_{\mathit{fail}}; k < v \\
\text{(stacks)} & k & ::= & \bullet \mid k \triangleright f
\end{array}
$$

In the state $k_{\mathit{fail}}; k > e$, the stack k_{fail} is the failure stack, and the stack k is the regular stack, and similarly in the state $k_{\mathit{fail}}; k < v$.

1. (5 pts) In the code above, what information does the `try` expression save in its stack frame?

2. (5 pts) Keeping in mind your answer to the previous question, give syntax for frames appropriate to the code above. You need only give the cases that deal with functions and/or exceptions.

   (When giving syntax for the stack frame for `try`, keep in mind that the format of the concrete syntax is not important, provided that the necessary information is present. There is no need to devise an elegant notation.)

3. (10 pts) Using the syntax you defined for the previous question, give transition rules for an abstract machine that formalizes the code above. Remember, your rules should preserve the invariant that k_{fail} is the prefix of k that ends at the innermost failure handler.

Worksheet

Worksheet

Worksheet