Computer Science 15–212–X: Final Examination

December 12, 1997

Name

Name: _____

Andrew ID: _____

Instructions

- This is an open-book, open-notes examination.
- There are 18 pages in this examination, including 5 worksheets.
- This examination consists of 4 questions worth a total of 200 points plus one question worth 20 points of extra credit. The point value of each question is given with the question.
- Read each question completely before attempting to solve any part.
- Write your answers legibly *in the space provided* on the examination sheet. If you use the back of a sheet or a worksheet, indicate *clearly* that you have done so on the front.
- Write your name and Andrew id in the space provided at the top of this page, and write your name at the top of each page in the space provided.
- Two worksheets are attached to the end of this examination for your personal use. *Nothing written on the worksheets will count for or against your grade on this exam.*

Grading

Question	1	2	3	4	Total	EC
Score						
Maximum	50	50	40	60	200	20

1 Streams [50 pts]

We define a datatype of *infinite stream* by

```
datatype 'a infStream = Delay of unit -> 'a infFront
and 'a infFront = Cons of 'a * 'a infStream;
```

1. [5 pts] Define a function

   ```
   val from : int -> int infStream
   ```

   such that `from` (⌜i⌝) returns the infinite stream $i, i+1, i+2, \ldots$.

2. [10 pts] Write an ML function

   ```
   val take : 'a infStream * int -> 'a list
   ```

   where `take` (s, n) returns a list containing the first n elements of s in order. You may assume without checking that $n \geq 0$.

3. [20 pts] Prove that for any i and for any $n \geq 0$,

   `take (from (`$\ulcorner i \urcorner$`), `$\ulcorner n \urcorner$`) ↪ [`$\ulcorner i \urcorner$`, `$\ulcorner i+1 \urcorner$`, ..., `$\ulcorner i+n-1 \urcorner$`]`

   where the result list should be empty for $n = 0$. Clearly state the induction principle you use and be careful to respect the details of the operational semantics of ML.

4. [15 pts] Write a function

   ```
   val merge : 'a infStream * 'a infStream -> 'a infStream
   ```

   such that `merge (`s_1`,` s_2`)` returns a stream in which the elements of s_1 and s_2 alternate, starting with the first element of s_1, then the first element of s_2, then the second element of s_1, etc. Make sure that your function is as lazy as possible!

2 Abstract Types [50 pts]

A *ring buffer* or simply *ring* is a useful data structure, for example, in task scheduling. It consists of elements arranged on a circle, where the top element can be accessed. When a new element is inserted, the present elements are pushed down clockwise. When the top element is deleted, elements move up counterclockwise. Rotating a ring to the left shifts all elements counterclockwise; rotating it to the right shifts elements clockwise. The following signature describes an interface to this data structure.

```
signature RING =
sig
  type 'a ring
  exception Empty
  val empty : 'a ring
  val insert : 'a * 'a ring -> 'a ring
  val delete : 'a ring -> 'a * 'a ring (* may raise Empty *)
  val top : 'a ring -> 'a              (* may raise Empty *)
  val rotateLeft : 'a ring -> 'a ring
  val rotateRight : 'a ring -> 'a ring
end;
```

Note that `delete` returns the deleted element and the remaining ring.

1. [5 pts] Give the implementation type for `'a ring` and state your representation invariant. Your data structure should be *persistent* and all operations should be constant time in the typical case, as for the implementation of queues presented in class. However, they do **not** need to be constant amortized time for all sequences of operations.

2. [30 pts] Implement a structure `Ring :> RING` satisfying the specification above.

3. [10 pts] Now assume you have a correct implementation `structure Ring :> RING` in your environment. Write a function

   ```
   val mirror : 'a Ring.ring -> 'a Ring.ring
   ```

   which computes the mirror image of a given ring.

4. [5 pts] Is it possible to write `mirror` if the function `delete` were omitted from the `RING` signature? Explain how it could be done or why it cannot be done. You don't need to actually give an implementation or proof.

3 Computability [40 pts]

In this problem we explore the combination of decision problems. Assume we have a type t and a property P for elements of type t. We call $D_P : t \to \texttt{bool}$ a *decision procedure* for P if

(i) D_P always has a value,

(ii) $D_P(v) \hookrightarrow \texttt{true}$ if property P holds for v, and

(iii) $D_P(v) \hookrightarrow \texttt{false}$ if property P does not hold for v.

We call a property *undecidable* if there is no decision procedure for it. We further define:

(a) The *negation of* P is the property which holds of an element v in t if P does not hold of v.

(b) The *disjunction of* P *and* Q is the property which holds for v if either P or Q (or both) hold for v.

1. [15 pts] Assume properties P and Q are decidable.

   (a) Is the negation of P always decidable?

   (b) Is the disjunction of P and Q always decidable?

   Prove or find a counterexample in each case.

2. [25 pts] Assume properties P and Q are undecidable.

   (a) Is the negation of P always undecidable?

   (b) Is the disjunction of P and Q always undecidable?

   Prove or find a counterexample in each case.

4 Interpreters [60 pts + 20 pts extra credit]

In this question we develop an interpreter for a tiny dynamically typed imperative language. We start with a language of expressions and then add commands (which only have effects but no values). In the language of expressions we have integers, addition, equality test, conjunction for booleans (written as $e_1 \& e_2$), and registers r (which later can be assigned values by commands).

$$\begin{array}{rrcl} \textit{Expressions} & e & ::= & n \mid e_1 + e_2 \mid e_1 = e_2 \mid e_1 \& e_2 \mid r \\ \textit{Values} & v & ::= & n \mid b \end{array}$$

Expressions evaluate to integers n or booleans b, or their evaluation might raise an exception. Note that comparing an integer and a boolean is a type error (instead of returning *false*), and that the right-hand side of a conjunction should not be evaluated if the left-hand side evaluates to *false*.

A register is a cell containing an optional value. If a register contains no value, it is considered uninitialized. Accessing an uninitialized register is considered a run-time error. We do not explicitly define, but assume representation functions $\ulcorner v \urcorner$: `Value`, $\ulcorner r \urcorner$: `Register`, and $\ulcorner e \urcorner$: `Exp` for values, registers, and expressions, respectively, using the types defined below.

```
exception TypeError;     (* raised for run-time type errors *)
exception Uninitialized; (* raise for uninitialized registers *)

datatype Value =
    Integer of int
  | Bool of bool;
type Register =
    Value option ref;

datatype Exp =
    Int of int
  | Plus of Exp * Exp
  | Equal of Exp * Exp
  | And of Exp * Exp
  | Reg of Register;
```

1. [20 pts] Define a function

   ```
   val eval : Exp -> Value
   ```

   to evaluate expressions, raising exceptions as appropriate.

Next we define a simple language of commands.

Commands	C	::=	$r := e$	assign value of e to r
		|	$\{C_1; \ldots; C_n\}$	execute commands in sequence, $n \geq 0$
		|	**if** e **then** C_1 **else** C_2	conditional
		|	**loop** l C	loop labelled l
		|	**exit** l	exit loop labelled l

The command **loop** l C executes C forever, unless the command **exit** l is encountered, in which case the loop is exited. Consider, for example, the program

$\{$
$\quad r_2 := 0;$
$\quad$ **loop** l_1 $\{$**if** $r_1 = 0$ **then exit** l_1 **else** $\{r_1 := r_1 + (-1); r_2 := r_2 + 2\}$;
$\quad r_1 := r_2 + 1$
$\}$

If this program is started with $n \geq 0$ in register r_1 then upon completion register r_1 will contain $2n + 1$ and r_2 will contain $2n$. As two other examples, consider

loop l $\{$**exit** $l\}$

which has the same effect as the empty sequence of commands $\{\ \}$, and

loop l $\{\ \}$

which is an infinite loop with no other effect.

Note that loops can be nested, and one can exit directly from an inner to an outer loop. However, labels are scoped so that l is only available in C for a loop of the form **loop** l C. In the representation, we model the scoping of labels by binding appropriate variables in ML. For example, the representation of

loop l_1 $\{$**exit** $l_1\}$

is

```
Loop (fn l1:Label => Exit(l1))
```

where `Loop` and `Exit` are constructors and `Label` is a given type of labels. We translate labels l to variables `l` with the same name, and commands can be translated as follows:

$\ulcorner r := e \urcorner$	=	`Assign (`$\ulcorner r \urcorner$`, `$\ulcorner e \urcorner$`)`
$\ulcorner \{C_1; \ldots; C_n\} \urcorner$	=	`Seq [`$\ulcorner C_1 \urcorner$`, ..., `$\ulcorner C_n \urcorner$`]`
$\ulcorner$**if** e **then** C_1 **else** $C_2 \urcorner$	=	`If (`$\ulcorner e \urcorner$`, `$\ulcorner C_1 \urcorner$`, `$\ulcorner C_2 \urcorner$`)`
$\ulcorner$**loop** l $C \urcorner$	=	`Loop (fn l:Label => `$\ulcorner C \urcorner$`)`
$\ulcorner$**exit** $l \urcorner$	=	`Exit(l)`

2. [5 pts] Complete the following `datatype` declaration for the representation of commands, assuming an appropriate type of `Label` has been defined.

```
datatype Cmd =
    Assign of
  | Seq of
  | If of
  | Loop of
  | Exit of
```

3. [20 pts] Write a continuation-passing interpreter

   ```
   val execute : Cmd -> (unit -> unit) -> unit
   ```

   for commands not containing `Loop` or `Exit`.

4. [5 pts] Show how to write a **while**-loop

   while e **do** C **end**

   using **loop** and **exit**.

5. [10 pts] Now extend the interpreter from above with two new cases to handle `Loop` and `Exit`. For this, we assume the definition

   ```
   type Label = unit -> unit
   ```

   before the declaration of `Cmd`, that is, we model an exit label as a continuation!

6. [20 pts Extra Credit] Prove that `Loop (fn l1 => Exit l1)` and `Seq []` are equivalent under your interpreter for any possible continuation k.

Worksheet

Worksheet

Worksheet

Worksheet

Worksheet