Computer Science 15–212–X: Final Examination
Sample Solution

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` ($\ulcorner i \urcorner$) returns the infinite stream $i, i+1, i+2, \ldots$.

   ```
   fun from (i) = Delay (fn () => Cons (i, from (i+1)));
   ```

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 $n \geq 0$.

   ```
   fun expose (Delay(f)) = f ();
   fun take (s, 0) = nil
     | take (s, n) = take' (expose s, n)
   and take' (Cons(x, s), n) = x::take(s, n-1);
   ```

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

   take (from ($\ulcorner i \urcorner$), $\ulcorner n \urcorner$) $\hookrightarrow$ [$\ulcorner i \urcorner$, $\ulcorner i+1 \urcorner$, $\ldots$, $\ulcorner i+n-1 \urcorner$]

   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.

We prove this property by mathematical induction on n.

Base Case: $n = 0$. Then

take (from ($\ulcorner i \urcorner$), 0)
$\Longrightarrow$ take (Delay (fn () => Cons ($\ulcorner i \urcorner$, from ($\ulcorner i \urcorner$+1))), 0)
$\Longrightarrow$ nil

Induction Step: Assume the property for n. We have to show the property for $n+1$. We compute

take (from ($\ulcorner i \urcorner$), $\ulcorner n+1 \urcorner$)
$\Longrightarrow$ take' (Cons ($\ulcorner i \urcorner$, from ($\ulcorner i \urcorner$+1)), $\ulcorner n+1 \urcorner$)
$\Longrightarrow$ take' (Cons ($\ulcorner i \urcorner$, Delay (fn () => Cons($\ulcorner i+1 \urcorner$, from ($\ulcorner i+1 \urcorner$+1)))), $\ulcorner n+1 \urcorner$)
$\Longrightarrow$ $\ulcorner i \urcorner$::take (Delay (fn () => Cons($\ulcorner i+1 \urcorner$, from ($\ulcorner i+1 \urcorner$+1))), $\ulcorner n \urcorner$)
$\Longrightarrow$ $\ulcorner i \urcorner$::[$\ulcorner i+1 \urcorner$, $\ldots$, $\ulcorner (i+1)+n-1 \urcorner$]
$\Longrightarrow$ [$\ulcorner i \urcorner$, $\ulcorner i+1 \urcorner$, $\ldots$, $\ulcorner i+(n+1)-1 \urcorner$]

The appeal to the induction hypothesis in the second-to-last above is valid since

from ($\ulcorner i \urcorner$) $\Longrightarrow$ Delay (fn () => Cons($\ulcorner i+1 \urcorner$, from ($\ulcorner i+1 \urcorner$+1)))

and therefore, by induction hypothesis,

take (from ($\ulcorner i \urcorner$), $\ulcorner n \urcorner$)
$\Longrightarrow$ take (Delay (fn () => Cons($\ulcorner i+1 \urcorner$, from ($\ulcorner i+1 \urcorner$+1))), n)
$\Longrightarrow$ [$\ulcorner i+1 \urcorner$, $\ldots$, $\ulcorner (i+1)+n-1 \urcorner$]

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!

```
fun merge1 (Cons (x, s1'), s2) =
      Cons (x, Delay (fn () => merge2 (s1', expose s2)))
and merge2 (s1', Cons (y, s2')) =
      Cons (y, Delay (fn () => merge1 (expose s1', s2')));
fun merge (s1, s2) = Delay (fn () => merge1 (expose s1, s2));
```

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.

   We use

   ```
   type 'a ring = 'a list * 'a list
   ```

   A pair (l,r) represents the ring with the elements of r strung clockwise around the ring starting from the top, and the elements of l strung counterclockwise just left from the top.

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

```
structure Ring :> RING =
struct
  type 'a ring = 'a list * 'a list
  exception Empty
  val empty = (nil, nil)
  fun insert (x, (left, right)) = (left, x::right)
  fun delete (nil, nil) = raise Empty
    | delete (left, x::right) = (x, (left, right))
    | delete (left, nil) = delete (nil, rev(left))
  fun top (nil, nil) = raise Empty
    | top (left, x::right) = x
    | top (left, nil) = top (nil, rev(left))
  fun rotateLeft (left, x::right) = (x::left, right)
    | rotateLeft (nil, nil) = (nil, nil)
    | rotateLeft (left, nil) = rotateLeft (nil, rev(left))
  fun rotateRight (x::left, right) = (left, x::right)
    | rotateRight (nil, nil) = (nil, nil)
    | rotateRight (nil, right) = rotateRight (rev(right), nil)
end;  (* structure Ring *)
```

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.

```
fun del (r) = SOME(Ring.delete r) handle Ring.Empty => NONE;
fun mirr (r, acc) = mirr' (del r, acc)
and mirr' (SOME(x, r), acc) = mirr (r, Ring.insert (x, acc))
  | mirr' (NONE, acc) = acc;
fun mirror r = Ring.rotateRight (mirr (r, Ring.empty));
```

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.

   Using `rotateRight`, `top`, and `insert` it is possible to construct the mirror image, but we don't know when to stop, since we cannot determine the size of a ring with the functions in the given interface without `delete`.

   Since the type `'a ring` is abstract (and the signature is ascribed with `:>`), we cannot apply any other operations than those given in the interface.

   A proof of this would be quite involved, requiring the notion of observational equivalence for abstract types in ML.

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.

   Assume `D_P` and `D_Q` are decision procedures for P and Q. then

   ```
   fun D_notP (x) = not (D_P (x))
   fun D_PorQ (x) = P(x) orelse Q(x)
   ```

   are decision procedres for the negation of P and the disjunction of P and Q.

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.

   For part 2a assume we had a decision procedure `D_notP` for the negation of P. Then we could construct a decision procedure for P:

   ```
   fun D_P (x) = not (D_notP (x))
   ```

   Since P is undecidable we have a contradiction, and there cannot be a decision procedure for the negation of P. Therefore the negation of P must also be undecidable

   For part 2b consider an undecidable property P and call its negation Q, which is also undecidable. The disjunction of P and Q is true for all elements v of the domain and therefore trivially decidable. Therefore, the disjunction of two undecidable properties may be decidable.

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.

```
fun plus (Integer(n), Integer(k)) = Integer(n+k)
  | plus _ = raise TypeError;
fun theBool (Bool(b)) = b
  | theBool _ = raise TypeError;
fun equal (Integer(i), Integer(j)) = Bool(i=j)
  | equal (Bool(b), Bool(c)) = Bool(b=c)
  | equal _ = raise TypeError;
fun eval (Int (n)) = Integer(n)
  | eval (Plus (e1, e2)) = plus(eval (e1), eval(e2))
  | eval (Equal (e1, e2)) = equal(eval(e1), eval(e2))
  | eval (And (e1, e2)) =
       Bool (theBool(eval(e1)) andalso theBool(eval(e2)))
  | eval (Reg (ref(SOME(v)))) = v
  | eval (Reg (ref(NONE))) = raise Uninitialized;
```

Next we define a simple language of commands.

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

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

$$\begin{array}{l} \{ \\ \quad r_2 := 0; \\ \quad \textbf{loop } l_1 \ \{\textbf{if } r_1 = 0 \textbf{ then exit } l_1 \textbf{ else } \{r_1 := r_1 + (-1); r_2 := r_2 + 2\}; \\ \quad r_1 := r_2 + 1 \\ \} \end{array}$$

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

$$\textbf{loop } l \ \{\textbf{exit } l\}$$

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

$$\textbf{loop } l \ \{\,\}$$

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

$$\textbf{loop } l_1 \ \{\textbf{exit } l_1\}$$

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:

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

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
```

```
datatype Cmd =
    Assign of Register * Exp
  | Seq of Cmd list
  | If of Exp * Cmd * Cmd
  | Loop of Label -> Cmd
  | Exit of Label;
```

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

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

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

```
fun execute (Assign(r, e)) k = (r := SOME(eval(e)); k ())
  | execute (Seq(C::Cs)) k = execute C (fn () => execute (Seq(Cs)) k)
  | execute (Seq(nil)) k = k ()
  | execute (If(e, C1, C2)) k =
    if theBool(eval(e)) then execute C1 k
    else execute C2 k
```

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

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

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

loop l_1 $\{$**if** e **then** C **else exit** $l_1\}$
where l_1 is a new label not used in C.

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!

```
| execute (Loop(C)) k = execute (C k) (fn () => execute (Loop(C)) k)
| execute (Exit(k')) k = k' ();
```

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

We simply compute:

```
    execute (Loop (fn l1 => Exit l1)) k
==> execute ((fn l1 => Exit l1) k)
            (fn () => execute (Loop (fn l1 => Exit l1)) k)
==> execute (Exit k) (fn () => execute (Loop (fn l1 => Exit l1)) k)
==> k ()
```

and

```
    execute (Seq []) k
==> k ()
```

Since none of the operations has an effect and both reduce to the same expression (namely k `()`), the two are equivalent.