Computer Science 15–212–ML

Final Examination

December 8, 1998

Model Solution

Instructions

- This is an open-book, open-notes, closed computer examination.
- There are 19 pages in this examination, including 5 worksheets.
- This examination consists of 5 questions worth a total of 200 points 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.
- The worksheets attached to the end of this examination are for your own use; they will not be used in grading.

Grading

Question	1	2	3	4	5	Total
Score						
Maximum	50	30	15	50	55	200

1 Data Abstraction [40 points+10 bonus points]

An *approximation procedure* for a value of some type t can be described abstractly by an *initial approximation* $x_0 : t$ and an *iterator* $f : t \to t$ that computes the next approximation from a given approximation.

The following signature defines a simple interface for computing with *approximators*.

```
signature APPROX =
sig
  type 'a approx
  val create : 'a * ('a -> 'a) -> 'a approx
  val current : 'a approx -> 'a
  val next : 'a approx -> 'a approx
end;
```

We specify that for values x and f of appropriate type,

`current (create (`x`, `f`))` $\simeq x$
`current (next (create (`x`, `f`)))` $\simeq f\ x$
`current (next (next (create (`x`, `f`))))` $\simeq f\ (f\ x)$
etc.

where $e_1 \simeq e_2$ if e_1 and e_2 evaluate to the same value.

We call x, $f(x)$, $f(f(x))$, ... the *successive approximations* for an approximator a defined by `create (`x`, `f`)`.

1.1 [15 points] Implement a `structure A :> APPROX` satisfying the specification above.

```
structure A :> APPROX =
struct
 type 'a approx = 'a * ('a -> 'a)
 fun create (x, f) = (x, f)
 fun current (x, _) = x
 fun next (x, f) = (f x, f)
end;
```

For each of the remaining questions in this problem, give an implementation assuming a `structure A :> APPROX`, if possible. If this is not possible, **explain why** and give an implementation inside the structure `A` (replacing `A.approx` by `approx` in the types).

1.2 [5 points] An approximation of the golden ratio $\phi = \frac{1+\sqrt{5}}{2} = 1.61803398875\ldots$ is given by the continued fraction

$$\phi = 1 + \cfrac{1}{1 + \frac{1}{1+\frac{1}{1+\cdots}}}$$

Define

```
val phiA : real A.approx
```

to approximate ϕ via this continued fraction, using 1 as the initial approximation.

```
val phiA : real A.approx = A.create (1.0, fn x => 1.0+(1.0/x));
```

1.3 [5 points] Implement

```
val nth : 'a A.approx * int -> 'a
```

where `nth` $(a,\ n)$ returns the n-th approximation according to a, where $n \geq 0$. For $n = 0$, this should be the initial value of a.

```
fun nth (a, 0) = A.current a
  | nth (a, n) = nth (A.next a, n-1);
```

1.4 [5 points] We say that an approximator *converges finitely* if two successive approximations (and therefore all subsequent ones) are equal. We call this approximation the *limit*. Implement

```
val limit : ''a A.approx -> ''a
```

where `limit` a returns the limit of a for a finitely converging approximator a.

```
fun limit' (prev, a) =
    if prev = A.current a then a else limit a
and limit (a) = limit' (A.current a, A.next a);
```

1.5 [5 points] Implement

```
val skip : 'a A.approx -> 'a A.approx
```

where `skip` a returns an approximator a' with the same initial value as a whose iterator skips every other value of a. Note that if a converges finitely, then a' will also converge finitely.

This cannot be defined outside the structure `A`, since we cannot extract the iteration function from an approximator. Inside the structure we can define

```
fun skip (x, f) = (x, fn y => f (f y))
```

1.6 [5 points] Assuming an implementation `S :> STREAM` of streams as defined in class, implement

```
val toStream : 'a A.approx -> 'a S.stream
```

where `toStream (create (`x`, `f`))` returns the lazy stream $x, f(x), f(f(x)), \ldots$ of successive approximations. Make sure your implementation computes approximations only as needed.

```
fun toFront (a) = S.Cons (A.current a, nextToStream a)
and nextToStream (a) = S.delay (fn () => toFront (A.next a));
fun toStream (a) = S.delay (fn () => toFront a);
```

Wrong would be the attempt

```
fun toStream (a) =
    S.delay (fn () => S.Cons (A.current a, toStream (A.next a)))
```

because the second value will be computed when the first is accessed.

2 Ephemeral Data Structures [30 points]

In this problem we consider a memoizing version of the persistent approximators from the previous problem.

```
signature MAPPROX =
sig
  type 'a approx
  val create : 'a * ('a -> 'a) -> 'a approx
  val current : 'a approx -> 'a
  val next : 'a approx -> 'a
end;
```

Here, `create` is as in the previous problem. `current` *a* returns the current value of the approximator *a* without changing its state. `next` *a* calculates and returns the next approximation of *a* and changes the state of the approximator to memoize this value.

2.1 [15 points] Implement a `structure MA :> MAPPROX` according to the specification above.

```
structure MA :> MAPPROX =
struct
  type 'a approx = 'a ref * ('a -> 'a)
  fun create (x, f) =
      let val cell = ref x
      in
        (cell, f)
      end
  fun current (c, f) = !c
  fun next (a as (c, f)) =
      let val x' = f (!c)
      in (c := x'; x') end
end;
```

2.2 [10 points] Implement a functor M that maps an implementation of a structure satisfying signature MAPPROX to one satisfying APPROX. Make sure that your functor can be used as required in the next question.

```
functor M (structure MA : MAPPROX) :> APPROX =
struct
  type 'a approx = 'a MA.approx
  fun create (x, f) = MA.create (x, f)
  fun current (a) = MA.current a
  fun next (a) = (MA.next a; a)
end;
```

2.3 [5 points] We can use the functor M to create

```
structure A' = M (structure MA = MA);
```

Illustrate that A and A' are **not** equivalent by giving an expression involving A whose value is different when A is replaced by A'.

```
 let
  val fracs0 = A'.create (0, fn x => 1);
  val fracs1 = A'.next fracs0;
in
  A'.current fracs0
end;
(* returns 1 *)

let
  val fracs0 = A.create (0, fn x => 1);
  val fracs1 = A.next fracs0;
in
  A.current fracs0
end;
(* returns 0 *)
```

3 Concurrency [15 points]

Approximators as introduced in the preceding two questions can be implemented concurrently in CML. Below are a signature and implementation of a *concurrent approximator*.

```
signature CAPPROX =
sig
  type 'a approx
  val create : 'a * ('a -> 'a) -> 'a approx
  val current : 'a approx -> 'a
end;

structure CA :> CAPPROX =
struct
  datatype 'a approx = Server of unit CML.chan * 'a CML.chan
  fun create (x, next) =
      let
        val requestCh = CML.channel ()
        val replyCh = CML.channel ()
        fun loop y =
            (CML.recv requestCh; CML.send (replyCh, y); loop (next y))
      in
        CML.spawn (fn () => loop x);
        Server (requestCh, replyCh)
      end
  fun current (Server (requestCh, replyCh)) =
      (CML.send (requestCh, ()); CML.recv replyCh)
end;
```

For each of the following expressions, indicate if the expression is type-correct, and if so give its type and list the possible outcomes of its evaluation, including non-termination, deadlock, multiple possible values, or exceptions. We are only interested in the outcome of evaluating this main expression—you do not need to describe the behavior of other threads which may be spawned.

3.1 [5 points]

```
let val a = CA.create (1.0, fn x => 2.0/x)
 in CA.current a; CA.current a end;
```

type `real`, value `2.0`

3.2 [5 points]

```
let fun loop () = loop ()
    val a = CA.create (1.0, fn x => loop ())
 in CA.current a; CA.current a end;
```

type `real`, main thread blocks, aux thread loops

3.3 [5 points]

```
let val c = ref 0
    fun inc c = (c := !c+1; c)
    val a = CA.create (c, inc)
    val b = CA.create (c, inc)
 in !(CA.current a); !(CA.current b) end
```

type `int`, possible values `0`, `1`, or `2`

4 Computability and Continuations [50 points]

A language $\mathcal{L}$ is said to be *recursively enumerable*, abbreviated r.e., if there is a function `f` of type `string -> bool` for which $\texttt{f}(s) \hookrightarrow \texttt{true}$ if and only if $s \in \mathcal{L}$. Thus, $\texttt{f}(s)$ need not halt for $s \notin \mathcal{L}$.

There are several operations we would like to perform on languages: union, intersection, concatenation, complementation, and star. These operations can be specified using the following datatype:

```
datatype lang =
    Predicate  of (string -> bool)
  | Union      of lang * lang
  | Intersect  of lang * lang
  | Concat     of lang * lang
  | Complement of lang
  | Star       of lang
```

The language $\mathcal{L}(p)$ of an expression p of type `lang` is recursively defined using the usual set operations:

$$
\begin{aligned}
\mathcal{L}(\texttt{Predicate}(p)) &= \{s \mid p(s) \hookrightarrow \texttt{true}\} \\
\mathcal{L}(\texttt{Union}(p,q)) &= \{s \mid s \in \mathcal{L}(p) \text{ or } s \in \mathcal{L}(q)\} \\
\mathcal{L}(\texttt{Intersect}(p,q)) &= \{s \mid s \in \mathcal{L}(p) \text{ and } s \in \mathcal{L}(q)\} \\
\mathcal{L}(\texttt{Concat}(p,q)) &= \{s \mid s = s_1\, s_2 \text{ with } s_1 \in \mathcal{L}(p) \text{ and } s_2 \in \mathcal{L}(q)\} \\
\mathcal{L}(\texttt{Complement}(p)) &= \{s \mid s \notin \mathcal{L}(p)\} \\
\mathcal{L}(\texttt{Star}(p)) &= \left\{s \mid s \in \mathcal{L}(p)^k \text{ for some } k \geq 0\right\}
\end{aligned}
$$

In the following two questions we investigate which language operations preserve recursive enumerability.

4.1 [20 points] Under which of the union, intersection, concatenation, complementation, and star operations is the class of r.e. languages *not* closed? For each such operation, give a proof that it may not result in a recursively enumerable language.

> Complementation. If the class of r.e. languages were closed under complement, we could use the `accept` function (as specified in the next problem) to solve the halting problem for functions of type `'a -> bool`, using the reduction

```
fun H(f,x) =
   (f(x) orelse not (f(x))) porelse
   not (accept (Complement(Predicate(fn s => f(x) orelse not(f(x))))) "QED")
```

4.2 [30 points]

Implement a function

```
val acc : lang -> char list -> (char list -> bool) -> bool
```

that takes a `lang` expression, a character list, and a continuation, and yields a boolean value, so that the function

```
fun accept p s = acc p (String.explode s) List.null
```

satisfies the specification `accept p` $s \hookrightarrow$ `true` if and only if $s \in \mathcal{L}(\mathtt{p})$.

You may use a *parallel or* operator `porelse` having the property that an expression

exp_1 `porelse` exp_2

evaluates to `true` if either exp_1 or exp_2 does (even if the other does not terminate). Both exp_1 and exp_2 must have type `bool`. (This might be implemented using concurrency or a special interpreter, for example.) You may also wish to use the function `String.implode: char list -> string`, which is the inverse of `String.explode`. If the class of r.e. languages is *not* closed under a given operation, according to your answer in 4.1, then your code should simply raise exception `Error` when called on the corresponding constructor.

```
fun prefix pred s nil k = pred (String.implode s) andalso k nil
  | prefix pred s (t as (h::t')) k = pred (String.implode s) andalso k(t)
                                     porelse prefix pred (s @ [h]) t' k
and acc (Predicate(p)) s k = prefix p nil s k
  | acc (Union(p,q)) s k = acc p s k porelse acc q s k
  | acc (Intersect(p,q)) s k =
        acc p s (fn s2 => acc q s (fn s2' => s2' = s2 andalso k s2'))
  | acc (Concat(p,q)) s k = acc p s (fn s' => acc q s' k)
  | acc (Complement(p)) s k = raise Error
  | acc (r as Star(p)) s k =
        k(s) porelse acc p s (fn s' => if s = s' then false
                                       else acc r s' k)
fun accept p s = acc p (String.explode s) List.null
```

5 Type Checking and Inference [55 pts]

Type inference for Standard ML is quite complex, since the implementation must introduce placeholders for omitted types, calculate constraints on the values of the placeholders, and then find the most general solution of these constraints to obtain a most general type. For the type checking algorithms considered in class and assignments, we have always required enough types to be present so that a unique type can be calculated for every well-typed expression.

In this problem we consider an intermediate form, where we distinguish two forms of expressions: those for which a unique type can be inferred, and those which can be checked against a type. In this language, only **let** forms need to be annotated with a type, but not function parameters.

Expression are drawn from the following small language.

$$\begin{array}{rrcl} \text{Types} & t & ::= & b \mid t_1 \to t_2 \\ \text{Expressions} & e & ::= & x \mid \textbf{fn}\, x \Rightarrow e \mid e_1\, e_2 \mid \textbf{let}\, x{:}t = e_1\, \textbf{in}\, e_2\, \textbf{end} \\ \text{Type Environment} & \Gamma & ::= & \cdot \mid \Gamma, x{:}t \end{array}$$

Here, b stands for base types and x for identifiers.

We have two mutually recursive typing judgments for our language:

- $\Gamma \vdash e \uparrow t$, which means we can infer a type t for expression e, and
- $\Gamma \vdash e \downarrow t$, which means we can check that e has type t.

These are defined by the following rules.

$$\frac{\mathit{lookup}(\Gamma, x) = t}{\Gamma \vdash x \uparrow t} \qquad \frac{\Gamma, x{:}t_1 \vdash e \downarrow t_2}{\Gamma \vdash \textbf{fn}\, x \Rightarrow e \downarrow t_1 \to t_2}$$

$$\frac{\Gamma \vdash e_1 \uparrow t_2 \to t_1 \qquad \Gamma \vdash e_2 \downarrow t_2}{\Gamma \vdash e_1\, e_2 \uparrow t_1} \qquad \frac{\Gamma \vdash e_1 \downarrow t_1 \qquad \Gamma, x{:}t_1 \vdash e_2 \downarrow t_2}{\Gamma \vdash \textbf{let}\, x{:}t_1 = e_1\, \textbf{in}\, e_2\, \textbf{end} \downarrow t_2}$$

$$\frac{\Gamma \vdash e \uparrow t}{\Gamma \vdash e \downarrow t}$$

5.1 [5 points] Exhibit an expression e and a type t such that $\cdot \vdash e \downarrow t$ (in the empty type environment). You do not need to show the typing derivation.

$$\cdot \vdash \textbf{fn}\, x \Rightarrow x \downarrow b \to b$$

for any base type b.

5.2 [5 points] Exhibit an expression e in our language for which no type can be inferred in the empty environment, even though it would be well-typed in ML.

There is no type t such that

$$\cdot \vdash \textbf{fn}\, x \Rightarrow x \uparrow t.$$

In ML it has type `'a -> 'a`.

We assume a declaration

```
exception Error
```

which is to be raised for errors during type-checking (including undeclared identifiers).

5.3 [15 points] For this problem, we will work with a functional representation of environments, mapping identifiers to values of type `'a`. That is,

```
type 'a Env = string -> 'a;
```

Implement the following functions:

- Environment extension

  ```
  val add : 'a Env * (string * 'a) -> 'a Env
  ```

  where `add (`Γ`, (`x`, `t`))` extends the environment Γ by binding x to t. Earlier declarations of x should be shadowed.

  ```
  fun add (G, (x, t)) = fn y => if x = y  then t else G y;
  ```

- Environment lookup

  ```
  val lookup : 'a Env * string -> 'a
  ```

  where `lookup (`Γ`, `x`)` returns the value bound to x in environment Γ. If x is undeclared, it raises the exception `Error`.

  ```
  fun lookup (G, x) = G x;
  ```

- The empty environment

  ```
  val empty : 'a Env
  ```

  which represent the environment with no declarations.

  ```
  val empty = fn s => raise Error;
  ```

We represent types and expressions with

```
datatype Type =
    Base of string
  | Arrow of Type * Type;

datatype Exp =
    Id of string
  | Lam of string * Exp
  | App of Exp * Exp
  | Let of string * Type * Exp * Exp;
```

5.4 [30 points] Implement functions

```
val infer : Type Env -> Exp -> Type
val check : Type Env -> Exp -> Type -> unit
```

where

- `infer` Γ e returns t if $\Gamma \vdash e \uparrow t$ and raises exception `Error` otherwise, and
- `check` Γ e t returns `()` if $\Gamma \vdash e \downarrow t$ and raises exception `Error` otherwise.

```
fun check G (Lam (x, e)) t =
    (case t
       of (Arrow (t1, t2)) => check (add (G, (x, t1))) e t2
        | _ => raise Error)
  | check G (Let (x, t1, e1, e2)) t2 =
    (check G e1 t1;
     check (add (G, (x, t1))) e2 t2)
  | check G e t =
    if infer G e = t then () else raise Error
and infer G (Id x) = lookup (G, x)
  | infer G (App (e1, e2)) =
    (case infer G e1
       of Arrow (t2, t1) => (check G e2 t2; t1)
        | _ => raise Error)
  | infer G e = raise Error;
```