Computer Science 15–212–ML

Midterm Examination

October 1, 1998

Sample Solution

Instructions

- This is a closed-notes, closed-book, closed-notebook (computer) examination.
- There are 14 pages in this examination, including three worksheets.
- The examination consists of three (3) problems worth a total of 100 points, plus two extra-credit problems worth a total of 30 points. The extra credit will be recorded separately; so you should finish all of the regular questions before attempting the extra credit.
- Read each problem completely before attempting to solve any part.
- You do not need to re-state the given invariants, but you should annotate auxiliary functions you define. Your answers should be correct, simple and clean, and they should take advantage of the given invariants. In other respects, your functions do not need to be particularly efficient, and they do not need to be tail-recursive.
- Write your answers legibly *in the space provided* on the examination sheet. If you use the back of a sheet, 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 for your own use; they will not be used in grading.

Grading

Problem	1	2	3	Total	Extra Credit
Score					
Maximum	35	35	30	100	30

1 Induction and Recursion [35 points]

We define binary trees having data only at the leaves by

```
datatype 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
```

We can uniquely identify a subtree of a binary tree by specifying a *path* from the root to the subtree: at each step during the top-down traversal we are instructed that we have arrived (`H`), that we have to go left (`L`) or that we have to go right (`R`). We represent paths by

```
datatype path = H | L of path | R of path
```

We call a path p *valid in a tree* t if the path designates some subtree of t. We also refer to a valid path as the *address* of a subtree or node. For example, the root of any tree has address `H`, the left subtree has address `L(H)` and the right subtree has address `R(H)`. The addresses `L(H)` and `R(H)` are not valid in a tree `Leaf(`x`)`.

1.1 [10 points] Write a function

```
val subtree : 'a tree * path -> 'a tree
```

where `subtree (`t`, `p`)` returns the subtree in t with address p, assuming the invariant that p is valid in t.

```
fun subtree (t, H) = t
  | subtree (Node(l, r), L(p)) = subtree (l, p)
  | subtree (Node(l, r), R(p)) = subtree (r, p)
```

1.2 [10 points] Write a function

```
val replace : 'a tree * path * 'a tree -> 'a tree
```

where `replace (`t`, `p`, `s`)` returns the tree which results from replacing that subtree in t with address p by s. You may assume that p is valid in t.

```
fun replace (t, H, s) = s
  | replace (Node (l, r), L (p), s) = Node (replace (l, p, s), r)
  | replace (Node (l, r), R (p), s) = Node (l, replace (r, p, s))
```

1.3 [15 points] Let the concatenation of two paths be defined by the following function.

```
fun concat (H, q) = q
  | concat (L(p), q) = L(concat (p, q))
  | concat (R(p), q) = R(concat (p, q))
```

Prove that

$$\texttt{subtree (replace (}t\texttt{, }p\texttt{, }s\texttt{), concat (}p\texttt{, }q\texttt{))} \cong \texttt{subtree (}s\texttt{, }q\texttt{)}$$

assuming that p is valid in t and q is valid in s. You may assume, without proof, that `concat` always terminates. Make sure to clearly state the induction principle you use and mark the places where you use the induction hypotheses and the validity assumptions about p and q (in case you need them).

We proceed by induction on the structure of p.

Base Case: $p = \texttt{H}$. Then

$$\begin{array}{ll} & \texttt{subtree (replace (}t\texttt{, H, }s\texttt{), concat (H, }q\texttt{))} \\ \Longrightarrow & \texttt{subtree (}s\texttt{, concat (H, }q\texttt{))} \\ \Longrightarrow & \texttt{subtree (}s\texttt{, }q\texttt{)} \end{array}$$

Induction Step: $p = \texttt{L(}p'\texttt{)}$. (The case for $p = \texttt{R(}p'\texttt{)}$ is symmetric).

Since p is valid in t, t must have the form $\texttt{Node(}l\texttt{,}r\texttt{)}$ where p' is valid in l. We compute

$$\begin{array}{ll} & \texttt{subtree (replace (Node (}l\texttt{, }r\texttt{), L(}p'\texttt{), }s\texttt{), concat (L(}p'\texttt{), }q\texttt{))} \\ \Longrightarrow & \texttt{subtree (Node (replace (}l\texttt{, }p'\texttt{, }s\texttt{), }r\texttt{), concat (L(}p'\texttt{), }q\texttt{))} \\ \Longrightarrow & \texttt{subtree (Node (}l'\texttt{, }r\texttt{), concat (L(}p'\texttt{), }q\texttt{))} \\ & \quad \text{where } \texttt{replace (}l\texttt{, }p'\texttt{, }s\texttt{)} \Longrightarrow l' \text{ since } p' \text{ valid in } l \\ \Longrightarrow & \texttt{subtree (Node (}l'\texttt{, }r\texttt{), L (concat (}p'\texttt{, }q\texttt{)))} \\ \Longrightarrow & \texttt{subtree (Node (}l'\texttt{, }r\texttt{), L (}p''\texttt{))} \\ & \quad \text{where } \texttt{concat(}p'\texttt{, }q\texttt{)} \Longrightarrow p'' \\ \Longrightarrow & \texttt{subtree (}l'\texttt{, }p''\texttt{)} \\ \cong & \texttt{subtree (replace (}l\texttt{, }p'\texttt{, }s\texttt{), concat (}p'\texttt{, }q\texttt{))} \\ & \quad \text{by definition of } l' \text{ and } p'' \\ \cong & \texttt{subtree (}s\texttt{, }q\texttt{)} \\ & \quad \text{by induction hypothesis on } l \end{array}$$

2 Continuations and Higher-Order Functions [35 points + 20 points extra credit]

Given a tree as introduced in the previous problem, with the data at the leaves, suppose that we are interested in a function that determines the address of the leftmost leaf satisfying a given predicate *pred*. That is, the function should search through the tree to find the leftmost leaf whose datum d satisfies $pred(d) \implies$ `true`, and return the path from the root of the tree to that leaf. If there is no leaf satisfying the predicate, then this should be indicated. So, we'd like to define a function

```
val find : ('a -> bool) -> 'a tree -> path option
```

This problem asks you to implement such a function using continuations.

2.1 [15 points] Write a function

```
val find' :  ('a -> bool) -> 'a tree -> (path -> 'b) -> (unit -> 'b) -> 'b
```

satisfying the following specification:

1. `find'` $pred\ t\ sc\ fc \Longrightarrow sc\ (p)$
   if p is a path to the leftmost datum in t satisfying predicate *pred*,
2. `find'` $pred\ t\ sc\ fc \Longrightarrow fc$ `()`
   if there is no datum in t satisfying *pred*.

You may assume *pred* always terminates. In the language of functional programming, *sc* is called a *success continuation* and *fc* a *failure continuation*. (Recall that `unit` is the type of the empty tuple `()`; for example, the expression `(fn () => 1)` has type `unit -> int`.)

```
fun find' pred (Leaf(d)) sc fc =
      if pred(d) then sc H else fc ()
  | find' pred (Node(l,r)) sc fc =
      find pred l (fn p => sc (L (p)))
                  (fn () => find pred r (fn p => sc (R (p))) fc)
```

2.2 [10 points] Using the function `find'`, write a function

```
val find : ('a -> bool) -> 'a tree -> path option
```

satisfying the following specification:

1. `find` *pred* $t \Longrightarrow$ `SOME(`p`)`
   if p is the path to the leftmost datum in t satisfying predicate *pred*,
2. `find` *pred* $t \Longrightarrow$ `NONE`
   if there is no datum in t satisfying *pred*.

```
fun find pred t = find' pred t (fn p => SOME(p)) (fn () => NONE)
```

2.3 [10 points] Prove the correctness of your function `find` assuming the correctness of `find'`.

The proof is by cases.

Case: Assume p is the path to the leftmost datum in t satisfying *pred*. By correctness of `find'`,

$$\begin{array}{ll} & \texttt{find'}\ \mathit{pred}\ t\ \texttt{(fn p => SOME(p)) (fn () => NONE)} \\ \Longrightarrow & \texttt{(fn p => SOME p)}\ p \\ \Longrightarrow & \texttt{SOME(}p\texttt{)} \end{array}$$

Case: Assume there is no datum in t satisfying *pred*. Again, by correctness of `find'`,

$$\begin{array}{ll} & \texttt{find'}\ \mathit{pred}\ t\ \texttt{(fn p => SOME(p)) (fn () => NONE)} \\ \Longrightarrow & \texttt{(fn () => NONE) ()} \\ \Longrightarrow & \texttt{NONE} \end{array}$$

2.4 [20 points extra credit] Prove the correctness of the function `find'`.

The proof proceeds by induction on the structure of t.

Case: t = `Leaf(`d`)`. If *pred* `(`d`)` $\Longrightarrow$ `true` then `find'` *pred* `(Leaf(`d`))` *sc fc* $\Longrightarrow$ *sc* `(H)`, which is what we needed to show.

If *pred* `(`d`)` $\Longrightarrow$ `false` then `find'` *pred* `(Leaf(`d`))` *sc fc* $\Longrightarrow$ *fc* `()`, which was required for this case.

Case: t = `Node(`l`,`r`)`. There are three subcases to consider: l contains a datum satisfying *pred*, l does not contain such a datum, but r does, and neither l nor r contain a datum satisfying *pred*.

Subcase: l contains a datum d satisfying *pred* with address p'. Then

```
    find' pred (Node(l,r)) sc fc
==> find' pred l (fn p => sc (L(p)))
        (fn () => find' pred r (fn p => sc (R(p))) fc)
==> (fn p => sc (L(p))) p'
      by induction hypothesis on l
==> sc (L(p'))
```

and `L(`p'`)` is indeed the path to d in t.

Subcase: l contains no datum satisfying *pred*, but r contains such a datum d with address p'. Then

```
    find' pred (Node(l,r)) sc fc
==> find' pred l (fn p => sc (L(p)))
        (fn () => find' pred r (fn p => sc (R(p))) fc)
==> (fn () => find' pred r (fn p => sc (R(p))) fc) ()
      by induction hypothesis on l
==> find' pred r (fn p => sc (R(p))) fc
==> sc (R(p'))
      by induction hypothesis on r
```

and `R(`p'`)` is indeed the path to d in t.

Subcase: Neither l nor r contain a datum satisfying *pred*. Then

```
    find' pred (Node(l,r)) sc fc
==> find' pred l (fn p => sc (L(p)))
        (fn () => find' pred r (fn p => sc (R(p))) fc)
==> (fn () => find' pred r (fn p => sc (R(p))) fc) ()
      by induction hypothesis on l
==> find' pred r (fn p => sc (R(p))) fc
==> fc ()
      by induction hypothesis on r
```

which is the desired result in this case.

3 Data Abstraction and Representation Invariants [30 points + 10 points extra credit]

Dyadic numbers are rational numbers such as 0, 1, $5\frac{3}{4}$, and $-\frac{17}{2^{10}}$ that can be written in the form $\frac{a}{b}$ where b is a power of two. This problem will work with the following signature for representing dyadic numbers:

```
signature DYADIC =
sig
   type number
   val zero : number
   val one  : number
   val half : number
   val neg  : number -> number           (* negation of a number   *)
   val add  : number * number -> number  (* sum of two numbers     *)
   val mul  : number * number -> number  (* product of two numbers *)
   val eq   : number * number -> bool    (* equal to  *)
end
```

Each of the operations in the signature has the obvious specification. Dyadic numbers are useful in symbolic computation, since even very small numbers have compact representations.

3.1 [5 points] Choose a representation for dyadic numbers by defining the type `number`, and stating any representation invariants. Your implementation should be able to represent dyadic numbers as small as 2^{-N}, where N is the largest integer that can be represented by the type `int`.

```
type number = int * int
```

We represent $\frac{a}{b}$ as a pair of integers (k,n) such that

1. $\frac{a}{b} = \frac{k}{2^n}$,
2. k is odd if $n > 1$,
3. $n \geq 0$.

3.2 [15 points] Write the code for a structure named `Dyadic` that matches the signature `DYADIC`, by defining the appropriate values, using the type you defined in the previous question. Invariants for auxiliary functions should be clearly stated.

```
structure Dyadic :> DYADIC =
struct
  (* cancel (k,n) = (k',n') where n >= 0,
     (k',n') satisfies representation invariants *)
  fun cancel (0, n) = (0, 0)
    | cancel (k, 0) = (k, 0)
    | cancel (k, n) =
      if k mod 2 = 0 then cancel (k div 2, n-1) else (k, n)
  (* pow2 (n) = 2^n for n >= 0 *)
  fun pow2 (0) = 1
    | pow2 (n) = 2 * pow2 (n-1)
  type number = int * int
  val zero = (0, 0)
  val one = (1, 0)
  val half = (1, 1)
  fun neg (k, n):number = (~k, n)
  fun add ((k1, n1), (k2, n2)) = cancel (k1*pow2(n2)+k2*pow2(n1), n1+n2)
  fun mul ((0, _), _) = zero
    | mul (_, (0, _)) = zero
    | mul ((k1, n1), (k2, n2)) = (k1*k2, n1+n2)
  fun eq (d1, d2) = (d1 = d2)
end
```

3.3 [10 points] Given a correct implementation `Dyadic :> DYADIC`, which of the following expressions and declarations are well-typed? If the expression or declaration is *not* well-typed, mark its box with an `X`; if it *is* well-typed, leave the box empty.

- [X] `val a : Dyadic.number = 0`
- [ ] `val b : Dyadic.number = Dyadic.add(Dyadic.zero, Dyadic.one)`
- [X] `val c : bool = (Dyadic.one = Dyadic.add(Dyadic.half, Dyadic.half))`
- [ ] `Dyadic.eq(Dyadic.add(Dyadic.one, Dyadic.neg(Dyadic.one)), Dyadic.zero)`
- [ ] `datatype interval = Empty | Interval of Dyadic.number * Dyadic.number`
- [ ] `fun f 1 = Dyadic.half | f n = Dyadic.mul(f 1, f (n-1))`
- [ ] `(fn x => (Dyadic.eq x = true))`
- [ ] `structure D : DYADIC = Dyadic`
- [ ] `structure D :> DYADIC = Dyadic`
- [X] `false orelse (Dyadic.zero = Dyadic.one)`

3.4 [10 points extra credit] Given a correct implementation `Dyadic :> DYADIC`, is it possible to write a function

```
val dyadic : int * int -> Dyadic.number option
```

that takes a rational number $\frac{a}{b}$ represented as a pair (a, b) in cancelled form and returns an equivalent dyadic number `SOME` d, if one exists, and returns `NONE` otherwise?

Write such a function, if possible, or explain why such a function cannot be defined.

Yes, such a function can be defined (see next page).

```
local
  exception Fail
  (* dyadic' (a,b), a/b cancelled, a >= 0, b >= 0 *)
  fun dyadic' (0, _) = Dyadic.zero
    | dyadic' (a, 1) = Dyadic.add (Dyadic.one, dyadic' (a-1, 1))
    | dyadic' (a, b) =
      if b mod 2 = 0
         then Dyadic.mul (Dyadic.half, dyadic' (a, b div 2))
      else raise Fail
in
  fun dyadic (a,b) =
      SOME (case (a >= 0, b >= 0)
              of (true, true) => dyadic' (a, b)
               | (false, true) => Dyadic.neg (dyadic' (~a, b))
               | (true, false) => Dyadic.neg (dyadic' (a, ~b))
               | (false, false) => dyadic' (~a, ~b))
      handle Fail => NONE
end
```