CIS 1200 Midterm I   September 27, 2024

Benjamin C. Pierce and Swapneel Sheth, instructors

SOLUTIONS

1. **Types** (21 points total)

   For each OCaml value below, fill in the missing type annotations or else write "ill typed" if there is no way to fill in the annotation that does not cause a type error.

   Your answer should be the *most generic* type that OCaml would infer for the value——*i.e.*, if `int list` and `bool list` are both possible types of an expression, you should write `'a list`.

   Some of these expressions refer to the types and functions defined in Appendix A, B, and C.

   We've done the first one for you.

```
let example: ________________int list________________ =
    [4; 5]
```

(a)
```
let ans: string list * string list =
   (["you"; "are"], ["a"; "rockstar"])
```

(b)
```
let ans: ill-typed =
   begin match [1200; 1600; 1210] with
   | [] -> false
   | hd::tl -> hd || tl
   end
```

(c)
```
let ans: int list -> int =
   fold (fun hd acc -> hd + acc) 0
```

(d)
```
let ans: int -> int -> int list =
   fun x y -> [x + 1; y]
```

(e)
```
let rec check_value
         (t: (bool * 'a) list) : 'a list =
   begin match t with
   | [] -> []
   | (f, s) :: tl -> if f then s :: check_value tl else check_value tl
   end
```

(f)
```
let ans: 'a -> 'a =
   fun x -> x
```

(g)
```
let ans: int -> (int -> int) list =
   fun a -> [ (fun x -> a + x); (fun y -> a + y); (fun z -> a + z) ]
```

2. **List Processing and Higher Order Functions** (31 points total)

   The definitions of the higher-order list processing functions `transform` and `fold` can be found in Appendix B.

   For these problems *do not* use any list library functions. Constructors, such as `::` and `[]`, are fine.

   (a) (8 points) Use `transform` and/or `fold`, along with suitable anonymous function(s), to implement a function `exists` such that the tests below pass.

```
let exists (pred: 'a -> bool) (l: 'a list) : bool =
  fold (fun x acc -> acc || pred x) false l


let test () : bool =
  not (exists (fun x -> x > 0) [])
;; run_test "exists: empty list" test

let test () : bool =
  exists (fun x -> x > 0) [1; 2; -5]
;; run_test "exists: multiple elements; returns true" test

let test () : bool =
  not (exists (fun x -> x > 0) [-1; -2; -5])
;; run_test "exists: multiple elements; returns false" test
```

   (b) (9 points) Use `transform` and/or `fold`, along with suitable anonymous function(s), to implement a function `filter` such that the tests below pass.

```
let filter (pred: 'a -> bool) (l: 'a list) : 'a list =
  fold (fun x acc -> if pred x then x :: acc else acc) [] l


let test () : bool =
  filter (fun x -> x > 0) [1; 2; -5; 3] = [1; 2; 3]
;; run_test "filter: multiple elements; some are filtered" test

let test () : bool =
  filter (fun _ -> false) ["a"; "b"; "c"] = []
;; run_test "filter: multiple elements; all are filtered" test
```

(c) (9 points) Complete the blanks in the pattern match to implement a function `assoc` such that the tests below pass.

```
let rec assoc (key: 'k) (l: ('k * 'v) list) : 'v =
  begin match l with
  | [] -> failwith "assoc: key not found"
  | (k, v) :: kvs -> if key = k then v else assoc key kvs
  end


let assoc_list: (int * string) list =
  [(1100, "Java"); (1200, "OCaml"); (1210, "Java"); (2400, "C")]

let test () : bool =
  assoc 1200 assoc_list = "OCaml"
;; run_test "assoc: key found" test

let test () : bool =
  assoc 42 assoc_list = "should fail"
;; run_failing_test "assoc: key not found" test
```

(d) (5 points) Use `transform` and/or `fold`, along with suitable anonymous function(s), to implement a function `double_all` such that the tests below pass.

(Hint: Remember that `transform` and `fold` can be used inside other expressions.)

Some possible solutions...

```
let double_all (l: int list list) : int list list =
  transform (fun x -> transform (fun y -> y*2) x) l

let double_all (l : int list list) : int list list =
  fold (fun inner_list acc_outer ->
          (fold (fun x acc_inner -> (2 * x) :: acc_inner)
                [] inner_list) :: acc_outer
   ) [] l

let double_all (l : int list list) : int list list =
  transform (fun inner_list ->
                fold (fun x acc -> (2 * x) :: acc)
                     [] inner_list) l

let double_all (l : int list list) : int list list =
  fold (fun inner_list acc_outer ->
          transform (fun x -> 2 * x) inner_list :: acc_outer)
       [] l

let double_all (l : int list list) : int list list =
  transform (transform (fun x -> 2 * x)) l
```

```
let test () : bool =
  double_all [[1; 2; 3; 4]; [5; 6; 7]]
           = [[2; 4; 6; 8]; [10; 12; 14]]
;; run_test "double_all 1: two sublists" test

let test () : bool =
  double_all [[]; [2]] = [[]; [4]]
;; run_test "double_all 1: empty sublist" test

let test () : bool =
  double_all [] = []
;; run_test "double_all 1: empty" test
```

3. **Binary Search Trees** (24 points total)

Consider generic binary search trees `'a tree`, as defined in the lectures, homework 3, and Appendix C.

(a) (12 points) Write a function `atleast` that takes a BST `t` and a number `n` and returns a new BST that contains all the values from `t` that are greater than or equal to `n`.

Your function **should not** use any of the BST functions we discussed in class (`insert`, `delete`, etc.) — instead, use the BST invariants to help you here!

It should pass these tests:

```
let t1 = Empty
let t2 = Node (Node (Empty, 1, Empty),
               2,
               Node (Empty, 3, Empty))

;; run_test "1"
  (fun () -> atleast t1 0 = Empty)
;; run_test "2"
  (fun () -> atleast t2 3 = Node (Empty, 3, Empty))
;; run_test "3"
  (fun () -> atleast t2 2 = Node (Empty, 2, Node (Empty, 3, Empty)))
;; run_test "4"
  (fun () -> atleast t2 1 = t2)
```

```
let rec atleast (t: int tree) (n: int) : int tree =
  begin match t with
  | Empty -> Empty
  | Node (l, x, r) ->
     if x < n then atleast r n
     else if x = n then Node (Empty, x, r)
     else Node (atleast l n, x, r)
  end
```

(b) (12 points) Fill in the blanks in the function `bst_delete_max`, which takes in a BST `s` and returns a tuple where the first entry is the maximum value in the tree `s` and the second entry is an updated BST with the max value deleted.

Again, your function **should not** use any of the BST functions we wrote in class — the BST invariants will help you here as well!

It should pass these tests:

```
let t1 = Empty
let t2 = Node (Node (Empty, 1, Empty),
               2,
               Node (Empty, 3, Empty))
let t3 = Node (Node (Empty, 1, Empty),
               2,
               Empty)

;; run_failing_test "1"
  (fun () -> bst_delete_max t1 = (0, t1))
;; run_test "2"
  (fun () -> bst_delete_max t2 = (3, t3))
;; run_test "3"
  (fun () -> bst_delete_max t3 = (2, Node (Empty, 1, Empty)))


let rec bst_delete_max (s: 'a tree) : 'a * 'a tree =
  begin match s with
  | Empty -> failwith "bst_delete_max called on Empty"
  | Node (lt, v, Empty) -> (v, lt)
  | Node (lt, v, rt) ->
      let max, new_rt = bst_delete_max rt in
      (max, Node (lt, v, new_rt))
  end
```

4. **Modules and Abstract Types (44 points total)**

Step 1: Understand the Problem The standard list operations like `length`, `append`, and `nth` take time proportional to the size of their list argument (their first list argument in the case of `append`). As a reminder, `nth l n` finds the n^{th} element of the list `l` by counting from the head (starting at 0) towards the tail one element at a time. For instance, `nth [0;1;2;3] 0` evaluates to `0` and `nth [0;1;2;3] 2` evaluates to `2`. If `nth` is given an index greater than (or equal to) the length of the list, it fails. For your reference, Appendix A gives the usual implementations of these operations, found in the `List` module.

Sometimes these functions are too slow for the task at hand. In this problem, we consider how to combine trees and lists to implement them more efficiently.

Nothing for you to do on this page

Step 2: Design the Interface The signature below defines an abstract type `'a rope` and operations on it. A `rope`, like a `list`, stores a sequence of data elements.

```
module type ROPE = sig
  type 'a rope
  val from_list : 'a list -> 'a rope
  val to_list   : 'a rope -> 'a list
  val append    : 'a rope -> 'a rope -> 'a rope
  val length    : 'a rope -> int
  val nth       : 'a rope -> int -> 'a
end
```

The *properties* of the `ROPE` functions are exactly the same as those for the corresponding list operations—in this regard, a rope is just a different implementation of the abstract type of lists. This means that a functionally correct implementation of this interface is:

```
module ListRope : ROPE = struct
  type 'a rope = 'a list

  let from_list (l : 'a list) : 'a rope = l
  let to_list   (r : 'a rope) : 'a list = r
  let length    (r : 'a rope) : int = List.length r
  let append    (lr : 'a rope) (rr : 'a rope) : 'a rope =
                   List.append lr rr
  let nth       (r : 'a rope) (n : int) : 'a =
                   List.nth r n
end
```

(a) (10 points) Which of the following statements are true of `ListRope`? Assume we have done

```
;; open ListRope
```

to import the definitions above, that `r`, `r1`, and `r2` refer to arbitrary values of type `'a rope`, and that `lst` is a `'a list`. Mark all that apply.

- ☒ `length r = List.length (to_list r)`
- ☒ If `to_list r = lst` then `nth r n = List.nth lst n`
- ☒ `length (append r1 r2) = (length r1) + (length r2)`
- ☒ If `(n < length r1)` then `nth (append r1 r2) n = nth r1 n`
- ☐ If `(n >= length r1)` then `nth (append r1 r2) n = nth r2 n`

Step 3: Define Test Cases (6 points) Our more efficient rope implementation, called `TreeRope`, should satisfy the same properties as `ListRope`. Complete each of the test cases below by filling in the blanks with identifiers `r0`, `r1`, `r2`, `r3`, or `r4` so that each test succeeds.

```
;; open TreeRope

let r0 = from_list [0;1;2]
let r1 = from_list [3;4]
let r2 = from_list [5;6;7;8]
let r3 = append r0 (append r1 r2)
let r4 = append (append r1 r1) r1
```

(b)
```
let test () =
  to_list r3 = [0;1;2;3;4;5;6;7;8]
;; run_test "test1" test
```

(c)
```
let test () =
  nth r2 2 = 7
;; run_test "test2" test
```

(d)
```
let test () =
  nth r1 2 = 0
;; run_failing_test "test3" test
```

Step 4: Implement the Code To implement these list operations more efficiently, we choose a different representation based on binary trees, encapsulated in a module named `TreeRope`. Intuitively, the idea is that a rope will be implemented by a binary tree whose leaves are lists, where **each leaf of the tree contains a subsequence of the complete sequence of data stored in the rope**.

```
                    /\
                   /  ["e"; "f"]
                  /\
["a"; "b"; "c"]  ["d"]
```

The benefit of this representation is that it makes the `append` operation faster. (If we want to append two ropes, we simply join them with a `Node` constructor.)

To accelerate the `length` and `nth` operations, we store extra information in the tree. Each leaf, in addition to a list, also stores the length of that list; the length is computed just once when the leaf is created, so repeatedly asking for length information about the leaf data doesn't require repeated traversals of the list at the leaf. Moreover, the total length of the lists in its left child is stored at the node.

```
                          4
                         / \
                        3   (["e"; "f"], 2)
                       / \
(["a"; "b"; "c"], 3)    (["d"], 1)
```

One last refinement will complete our definition of ropes. There is no point in storing lots of leaves that contain the empty list, so we require that the left subtree of every node have length strictly greater than 0, which means its leaves can't contain just empty lists.

The module declaration and tree type are shown below. It is basically yet another variant of generic binary trees, where leaves are labeled with `'a list`s together with their lengths and nodes are labeled just with lengths.

```
module TreeRope : ROPE = struct

  type 'a treerope =
    | Leaf of 'a list * int
    | Node of 'a treerope * int * 'a treerope
  type 'a rope = 'a treerope
```

For example, the rope pictured above is written in OCaml like this

```
;; open TreeRope

Node (Node (Leaf(["a";"b";"c"], 3),
            3,
            Leaf(["d"], 1)),
      4,
      Leaf(["e";"f"], 2))
```

and it can be built by evaluating this expression:

```
append (append (from_list ["a";"b";"c"])
               (from_list ["d"]))
       (from_list ["e";"f"])
```

In brief, the invariant of the tree representation of ropes is:

Rope Invariants

A value `r : 'a treerope` satisfies the rope invariants if:

- `r` is `Leaf(l, n)` and `List.length l = n`, or
- `r` is `Node(lt, n, rt)` and
  - `n > 0` and `n` is the total length of all the lists stored at the leaves in `lt`
  - `lt` and `rt` both recursively satisfy the rope invariants

(e) (2 points) Does this `string treerope` value satisfy the tree rope invariants?

```
([], 0)
```

☒ Yes ☐ No

(f) (2 points) Does this `string treerope` value satisfy the tree rope invariants?

```
          0
         / \
 ([], 0)    (["b"], 1)
```

☐ Yes ☒ No

(g) (2 points) Does this `string treerope` value satisfy the tree rope invariants?

```
                     2
                    / \
(["a"; "b"], 2)   ([], 0)
```

☒ Yes ☐ No

(h) (6 points) Given the invariants above, which of the following is a correct implementation for the `length` operation on ropes? (There may be zero, one, or more than one correct implementation.)

☐
```
let rec length (t : 'a treerope) : int =
  begin match t with
    | Leaf (l,_) -> 0
    | Node (_, _, rt) -> 1 + length rt
  end
```

☐
```
let rec length (t : 'a treerope) : int =
  begin match t with
    | Leaf (l,x) -> x
    | Node (lt, x, _) -> x + length lt
  end
```

☒
```
let rec length (t : 'a treerope) : int =
  begin match t with
    | Leaf (l,x) -> x
    | Node (_, x, rt) -> x + length rt
  end
```

Complete the code for each of the following operations that build rope trees. In each case, ensure that the resulting tree satisfies the rope invariants. You may use `List.length` to refer to the list version of length and just `length` to refer to the rope version defined above. Do *not* use `List.append` (or `@`) in this implementation. Note that neither operation below is recursive!

(i) (4 points)

```
let from_list (l : 'a list) : 'a treerope =
  Leaf (l, List.length l)
```

(j) (4 points)

```
let append (lt : 'a treerope) (rt : 'a treerope) : 'a treerope =
  let x = length lt in
  if x = 0 then rt else
    Node (lt, x, rt)
```

Complete the code for the rope version of the `nth` operation. Your implementation should exploit the rope invariants as much as possible. You may use `List.nth` to refer the list version of nth. Note that this function is recursive!

(k) (8 points)

```
let rec nth (t : 'a treerope) (n : int) : 'a =
  begin match t with
    | Leaf (l,_) -> List.nth l n
    | Node (lt, x, rt) ->
      if n < x then nth lt n else nth rt (n - x)
  end
```

A Basic List Processing Functions

Some standard list processing functions:

```
(* Relevant part of the list library *)
module List = struct
  (* ... other operations elided ... *)

  let rec length (l : 'a list) : int =
    begin match l with
      | [] -> 0
      | _::xs -> 1 + length xs
    end

  let rec append (l1 : 'a list) (l2 : 'a list) : 'a list =
    begin match l1 with
      | [] -> l2
      | x::xs -> x::(append xs l2)
    end

  let rec nth (l : 'a list) (n:int) : 'a =
    begin match l with
      | [] -> failwith "not found"
      | x::xs -> if n = 0 then x else nth xs (n-1)
    end
end
```

B Higher-Order List Processing Functions

The higher-order list processing functions `transform` and `fold`:

```
let rec transform (f : 'a -> 'b) (p : 'a list) : 'b list =
  begin match p with
    | (entry::rest) -> f entry :: transform f rest
    | [] -> []
  end

let rec fold
    (combine: 'b -> 'a -> 'a)
    (base:'a)
    (l : 'b list) :  'a =
  begin match l with
    | [] -> base
    | h :: tl -> combine h (fold combine base tl)
  end
```

C Generic Binary Search Trees

```
(* Generic binary trees, from HW 3 *)
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree

let rec lookup (t:'a tree) (n:'a) : bool =
  begin match t with
    | Empty -> false
    | Node(lt, x, rt) ->
      x = n || if n < x then lookup lt n else lookup rt n
  end

(* Inserts n into the binary search tree t *)
let rec insert (t:'a tree) (n:'a) : 'a tree =
  begin match t with
    | Empty -> Node(Empty, n, Empty)
    | Node(lt, x, rt) ->
      if x = n then t
      else if n < x then Node (insert lt n, x, rt)
      else Node(lt, x, insert rt n)
  end
```