COS 326 Functional Programming Midterm Exam, Fall 2017

Name (print clearly): Sample Solution with Commentary

Login:

You are allowed to bring a hand-written sheet (8.5x11in page, both sides) into the exam containing any information you want. Other than that, the work on this midterm must be your own without any outside help. ***A few students are taking this exam on Thursday. Do not talk about this exam outside the exam room or with your friends until Friday.***

Honor Pledge and signature:

General Instructions:

- **Write your name and login on this page. Write and sign the Honor Code pledge**.
- You have **1 hour and 20 minutes** to complete this exam.
- There are **5 parts** to the exam (each part may have several questions). Write answers in the spaces provided. We will give partial credit on questions. So part of a solution, if it is correct, or at least sensible, is better than no solution at all.
- Minor errors in OCaml syntax will not be penalized significantly, if at all. However, we have no choice but to penalize errors that render a solution incomprehensible.
- If you cannot complete the details of a proof but can show that you know how to structure the proof, you will receive some credit. Show that you know how to break down a proof into appropriate cases. Show that you know what the induction hypothesis is by writing it down clearly. Show that you know what must be proven
- Do not give two answers to a question hoping that one of them is correct. If you give two different answers to a question, you will receive no credit if one of the answers is correct and one is incorrect. If ambiguous, circle the answer you intend.
- Grades on the 5 questions:

  1 _____ / 3 4 _____ / 10

  2_____ / 2 5 _____ / 6 **Total _____ / 25**

  3_____ / 4

Problem 1 [3 points]

(a) Circle (clearly!) any free variables of the following expression.

```
(fun x ->
  let x = x in
  match (fun y -> x) y with
    (x::y)::z -> z
  | _ -> [])
```

The underlined red y is the only free variable. The other variables are bound at the sites in their corresponding colors. The anonymous function of y never uses its parameter within its body.

(b) Assume we are working with the substitution-based model of evaluation. What value does this expression evaluate to?

```
(fun x ->
  match x with
    (x::y)::z -> y
  | _ -> []) [[1;2;3]; [4;5;6]; [7]]
```

value: [2;3]

(x::y)::z must have x be of type int, y of type int list, and z of type int list list.
Matching using these types yields x as 1, y as [2;3], and z as [[4;5;6];[7]].

(c) Assume we are working with the substitution-based model of evaluation. Does this expression evaluate to a value? If so, provide the value it evaluates to. If not, explain:

```
(fun x -> let (x,y) = x in x) ((fun y z -> y) 3, 2)
```

value or explanation: fun z -> 3

The anonymous function of x is just the function fst.
It gets applied to the pair ((fun y z -> y) 3, 2). The result is the first component, the function call (fun y z -> y) 3. When we evaluate that function call, we must recall the reality of a function of multiple parameters: (fun y z -> y) is really (fun y -> (fun z -> y)); so applying the outer function such that y=3 results in the inner function with 3 substituted for y.

Problem 2 [2 points]:

(a) The following function `foo` contains the free variables `w` and `v`. Either, give types for `w` and `v` that allow `foo` to type check or explain why there are no possible types you can give to `w` and `v` that make `foo` type check.

```
let rec foo x y z =
  if (x w) then
    3 + z
  else
    x (foo x y v)
```

answer :
We can, in fact, find the types that w and v must be in this function:

* v must be of type int, because it takes the place of the parameter z in the function call in the alternative, and z must be an int because it is added to 3 in the consequent.

* w must be of type int, because it is the argument to x in the conditional, and x's argument must be an int because (foo x y v) is its argument in the alternative, and (foo x y z) returns an int in the consequent.

BUT: the function still does not type check, even with those type assignments. Consider the return type for x. It must be boolean, because it is the conditional: *if (x w) then*.
And it must also be int, because we already established that the consequent returns an int, so the alternative must too, and the alternative is just a call to x: *x (foo x y v)*

Thus, there are no types for w and v that will make foo type check.

(b) What is the type of the following expression?

```
let id x = x in
let id x = x id in
id
```

type of expression: ((‘a -> ‘a) -> ‘b) -> ‘b

The first id is the identity function, and has type (‘a -> ‘a).

The parameter of the second id is a function that takes the identity function as an argument and returns some unknown type. Thus this x's type is (‘a -> ‘a) -> ‘b.

The second id takes x as an argument and returns whatever x returns, for the x discussed immediately above. Thus if this x is of type (‘a -> ‘a) -> b, the second id must be of type: ((‘a -> ‘a) -> ‘b) -> ‘b.

Problem 3 [4 points] Consider the following code. Prove the inductive case of the theorem below – we have proven the base case for you (you'll note that we evaluate a function and its immediate match in one proof step – you can do the same in your case of the proof).

```
let rec dup1 xs =
  match xs with
  | [] -> []
  | hd::tl -> hd::hd::(dup1 tl)

let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> f hd :: (map f tl)

let rec flatten xs =
  match xs with
  | [] -> []
  | [] :: tl -> flatten tl
  | (hh::ht) :: tl -> hh :: flatten (ht::tl)

let d x = x::x::[]

let dup2 xs =
  flatten (map d xs)
```

Theorem:

```
for all xs:int list.
  dup1 xs = dup2 xs
```

Proof: By induction on the structure of xs.

case xs = []:

```
   dup1 []                 (LHS)
== []                      (eval dup1)
== flatten []              (reverse eval flatten)
== flatten (map d [])      (reverse eval map)
== dup2 []                 (reverse eval dup2 -- RHS)
```

(see following page for inductive case – you fill that in)

case xs = hd::tl:

```
IMS: dup1 hd::tl = dup2 hd::tl
IH:  dup1 tl     = dup2 tl
(1)    dup2 hd::tl                          RHS
(2) == flatten (map d hd::tl)               eval dup2
(3) == flatten ((d hd)::map d tl)           eval map
(4) == flatten ((hd::hd::[])::map d tl)     eval d
(5) == hd::flatten ((hd::[])::map d tl)     eval flatten
(6) == hd::hd::flatten ([]::map d tl)       eval flatten
(7) == hd::hd::flatten (map d tl)           eval flatten
(8) == hd::hd::dup2 tl                      inv. eval dup2
(9) == hd::hd::dup1 tl                      IH
(A) == dup1 hd::tl                          inv. eval dup1
```

Note that we started on the RHS because it makes the step that produces ([]::map d tl) less mysterious — it is simply the result of calling flatten, and we can simply call flatten again. Whereas from the LHS, it would take a bit of creative insight to realize that from flatten (map d tl) we can inverse evaluate the 2nd branch of flatten to get flatten ([]::map d tl).

Problem 4 [10 points] Consider the following code.

```
let rec rev1 l =
  match l with
      [] -> []
    | hd::tl -> (rev1 tl) @ [hd]

let rec rev_aux l r =
  match l with
      [] -> r
    | hd::tl -> rev_aux tl (hd::r)

let rev2 l = rev_aux l []
```

Your goal is to prove the following theorem:

```
For all l:int list.
rev1 l == rev2 l
```

Above @ is OCaml's list concatenation operator. You will need to assume some properties of @ to complete the proof. For example, you may need to assume that concatenation is associative:

Property 1: for all xs, ys, zs:int list, (xs @ ys) @ zs == xs @ (ys @ zs)

You can assume any such (true) facts about @ that you want. Simply write down the property of @ that you are assuming (like I wrote down "Property 1" above – write down "Property 2", "Property 3", etc). Your assumptions/properties concerning @ should not involve "rev1" "rev2" or "rev_aux" directly – they should involve arbitrary lists or integers.

Note: If you can't figure out how to prove this theorem, at least explain clearly where you get stuck. A partial proof and a clear explanation of where you get stuck, and why, is worth more partial credit than a proof with erroneous justifications/incorrect proof steps.

(Do your proof starting on the next page.)

(You may have as few or as many properties of @ as you need. I recommend filling them in after completing the proof so you know what you need.)

Property 2: `for all r:int list. [] @ r == r`
Property 3: `for all r:int list. r @ [] == r`
Property 4: `for all hd:int. for all r:int list. [hd]@r == hd::r`

Observation: What we will really need to prove is that rev1 l == rev_aux l []. But in the inductive case, we quickly get rev_aux tl [hd], which will not match the IH, leaving us stuck.

Insight: Our invariant is that rev_aux l' r at any step has l' with what is left from the original list l (still unreversed), and r with what we have already reversed from the original list l.

Lemma:
```
for all l:int list, for all r:int list.
rev_aux l r == (rev1 l) @ r.
```

Proof: By induction on the structure of l.

case: l == []

```
pick any r.

   rev_aux [] r        (LHS)
== r                   (eval of rev_aux)
== [] @ r              (by property 2)
== (rev1 l) @ r        (by eval reverse of rev1)
```

case: hd::tl

```
   (rev1 (hd::tl)) @ r      (RHS)
== ((rev1 tl) @ [hd]) @ r   (eval rev1)
== rev1 tl @ ([hd] @ r)     (by property 1)
== rev1 tl @ (hd::r)        (by property 4)
== rev_aux tl (hd::r)       (IH)
== rev_aux (hd::tl) r       (by reverse eval rev_aux)
```
Proof of Lemma complete.

Proof of theorem:
```
For all l:int list.
rev1 l == rev2 l
```

Proof: By simple equational reasoning.

```
Pick any l.

   rev2 l              (RHS)
== rev_aux l []        (eval of rev2)
== rev1 l @ []         (by Lemma)
== rev1 l              (by property 3)
```
Proof of Theorem complete!

Problem 5 [8 points]: Consider the following code

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

let rec tmap (f:'a -> 'b) (t:'a tree) : 'b tree =
  match t with
      Leaf -> Leaf
    | Node (x,l,r) -> Node (f x, tmap f l, tmap f r)

let rec treduce (f:'a -> 'b -> 'b -> 'b) (b:'b) (t:'a tree):'b =
  match t with
      Leaf -> b
    | Node (x,l,r) -> f x (treduce f b l) (treduce f b r)

let rec tfold (f:'a -> 'b -> 'b) (b:'b) (t:'a tree) : 'b =
  match t with
      Leaf -> b
    | Node (x,l,r) -> f x (tfold f (tfold f b l) r)
```

(a) Using the functions provided above, but without using additional occurrences of the "rec" keyword, write a function `tmerge` that takes two trees and returns a single resulting tree containing all (and only) the elements of the first two. The elements in the result tree can be in any order, but you should not produce extra copies on an element. For example:

`tmerge Leaf Leaf` can only return `Leaf`

On the other hand:

```
tmerge (Node(2,Leaf,Leaf)) (Node(2,Leaf,Leaf))
```

can return:

```
Node(2, (Node(2,Leaf,Leaf)), Leaf)
```

or

```
Node(2, Leaf, Node(2,Leaf,Leaf))
```

but can not return:

```
Node(2, (Node(2,Leaf,Leaf)), (Node(2,Leaf,Leaf)))
```

and can not return

```
Node(2,Leaf,Leaf)
```

```
let tmerge (t1:'a tree) (t2:'a tree) : 'a tree =
```

```
One answer: tfold (fun x b -> Node (x, Leaf, b)) t1 t2
```

… it may be helpful to write out the explicit recursive solution to this problem to gain insight into developing the combinator library version:

```
match t1 with
| Leaf -> t2
| Node (x,l,r) -> Node (x, merge l t2, r)
```

Though the straightforward explicit recursive version will not yield the same exact tree as the combinator version, it could help develop ideas about the tree invariant (the number of Leafs in the tree is always one more than the number of Nodes in the tree); the general idea that we pull one item out of one tree at a time, looking for a chance to replace a Leaf with the other tree; etc.

(b) Once again, without using explicit recursion, write a function `tflatten` that converts an `'a tree tree` into an `'a tree`. For any `tt : 'a tree tree, tflatten tt` should have the same elements as `tt`. The order that the elements appear in the result and the structure of the tree is unimportant (make any choice you want that is convenient). For example,

```
let t2 = Node(2,Leaf,Leaf)
let t3 = Node(3,Leaf,Leaf)
let tt = Node(t2, Node (t3, Leaf, Leaf), Leaf)
```

Now, `tflatten tt` could be:

```
Node(2,Node(3,Leaf,Leaf),Leaf)
```

Note: You may assume that you have a working copy of `tmerge` from the previous part, even if you couldn't get part (a) to work.

```
let tflatten (t:int tree tree) : int tree =
  one answer: treduce (fun x l r -> tmerge x (tmerge l r)) Leaf t

  another answer: tfold merge Leaf t
```

Again, it may be helpful to write the explicit recursive version. Conveniently, a version of flatten for lists that is easily adapted to trees is given on another problem …

```
match t with
| Leaf -> Leaf
| Node(Leaf,tl,tr) -> tmerge (tflatten tl) (tflatten tr)
| Node(n,tl,tr) -> tmerge n (tmerge  (tflatten tl) (tflatten tr))
```

If you do, you'll notice that the general case is a merge of the Node's item with the result of merging the flattened left and right subtrees. This yields exactly the result in the treduce answer above. Alternately, you might notice that we're really just doing a merge on the item and the result of merging all the other Nodes' items, which translates easily into the tfold answer above.