CIS 120 Midterm II March 25, 2015

SOLUTIONS

1. OCaml and Java: True/False (22 points)

Circle T or F.

a. [T] F Variables stored on the stack in the OCaml Abstract Stack Machine are not mutable, although they may refer to mutable records in the heap.

b. T [F] All records are mutable in OCaml.

c. T [F] A well-typed OCaml program can fail with an error because it tried to use a `null` pointer.

d. [T] F In both Java and OCaml, if `v1` and `v2` are references, then the expression `v1 == v2` evaluates to true exactly when they refer to the same location in the heap.

e. [T] F In OCaml, if `s` and `t` are variables of type `string` such that `s == t`, then `s = t` is guaranteed to return **`true`**.

f. T [F] In Java, if `s` and `t` are variables of type `String` such that `s.equals(t)`, then `s == t` is guaranteed to return **`true`**.

g. [T] F The following OCaml function will terminate by exhausting stack space if called as `loop 10`.

```
let rec loop (x:int) : int list =
  if x = 0 then [] else x :: loop x
```

h. [T] F There is a type that can be filled in for the blank in the following OCaml program to make it well typed.

```
type 'a ref = { mutable contents : 'a }

let x : ____________________ = { contents = [] }
;; x.contents <- ((fun () -> ()) :: x.contents)
```

i. [T] F In our GUI library, an `event_listener` is a first-class function stored in the hidden state of a `notifier` widget. When an event occurs in the widget, the `notifier` invokes all of the registered `event_listeners`.

j. T [F] Applications built using our OCaml GUI library should put all code for drawing in event listeners.

k. [T] F In the OCaml ASM, first-class functions are stored in the heap and may have local copies of variables that were on the stack when they were defined.

2. Higher-order functions: fold (15 points)

The following functions below are written without using `fold`. Circle the correct redefinition in terms of this higher-order function.

```
let rec fold (combine: 'a -> 'b -> 'b) (base : 'b) (x: 'a list) : 'b =
 begin match x with
 | [] -> base
 | h :: t -> combine h (fold combine base t)
 end
```

a.
```
let rec reverse (x : int list) : int list =
 begin match x with
 | [] -> []
 | h :: t -> reverse t @ [h]
 end
```

How would you rewrite this function using `fold`?

i.
```
let reverse (x : int list) : int list =
      fold (fun (h:int) (y:int list) -> h :: y) 0 x
```

ii.
```
let reverse (x : int list) : int list =
      fold (fun (h:int) (y:int list) -> y @ [h]) 0 x
```

iii.
```
let reverse (x : int list) : int list =
      fold (fun (h:int) (y:int list) -> y @ [h]) [] x
```

iv. `reverse` can't be written with fold.

Answer: iii

b.
```
let rec g (x: 'a) (m: 'a list) : bool =
 begin match m with
 | [] -> false
 | hd :: tl -> x == hd || g x tl
 end
```

How would you rewrite this function using fold?

i.
```
let g (x:'a) (m: 'a list) : bool =
     fold (fun (y:'a) (z:bool) -> (x == y) || z) false m
```

ii.
```
let g (x:'a) (m: 'a list) : bool =
     fold (fun (y:'a) (z:bool) -> (x == hd) || g x tl) false m
```

iii.
```
let g (x:'a) (m: 'a list) : bool =
     fold (fun (y:'a) (z:bool) -> (x == z)) false m
```

iv. `g` can't be written with fold.

Answer: i

c.
```
let rec f (x : 'a option list) : 'a list =
  begin match x with
  | [] -> []
  | None :: t -> f t
  | Some y :: t -> y :: f t
  end
```

How would you rewrite this function using fold?

i.
```
let f (x : 'a option list) : 'a list =
    fold (fun (h : 'a) (r : 'a list) -> h :: r) [] x
```

ii.
```
let f (x : 'a option list) : 'a list =
    fold (fun (h : 'a option) (r : 'a list) ->
      begin match h with
      | None -> r
      | Some y -> y :: r
      end) [] x
```

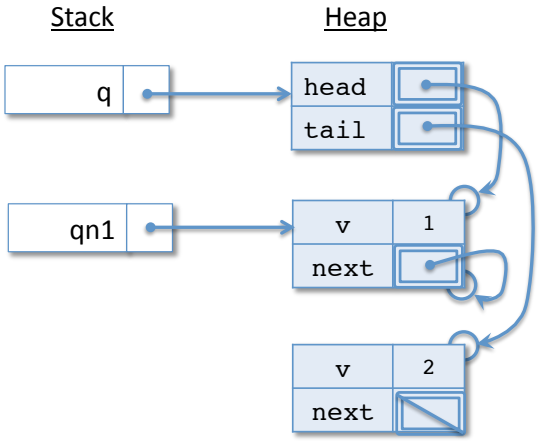
iii.
```
let f (x : 'a option list) : 'a list =
    fold (fun (h : 'a option) (r : 'a list) ->
      begin match r with
      | [] -> None
      | y :: tl -> Some y
      end) None x
```

iv. `f` can't be written with fold.

Answer: ii

3. OCaml Abstract Stack Machine (22 points)

Recall the mutable queue implementation from class and homework and shown in Appendix A. Suppose that we have the OCaml ASM shown below.



a. Does `q` satisfy the queue invariant? Circle **true** or **false** .

Grading Scheme: 3 points

b. Write a short piece of code that can be loaded onto the workspace to get the ASM to this configuration. Feel free to call any of the functions in Appendix A, if it helps. However, note that the stack and heap after your code executes must look *exactly* as in the drawing: it must put variables on the stack in the correct order and it cannot include any extra stack variables or qnodes in the heap.

```
let q = create () in
enq q 1;
enq q 2;
let qn1 = begin match q.head with
  | Some x -> x
  | None -> failwith "impossible"
 end in
qn1.next <- Some qn1
```

or

```
let q = { head = None; tail = None } in
let qn1 =
 let qn2 = { v = 2; next = None } in
 q.tail <- Some qn2;
 { v = 1 ; next = None} in
q.head <- Some qn1;
qn1.next <- Some qn1
```

Grading Scheme: 10 points total. Additions:

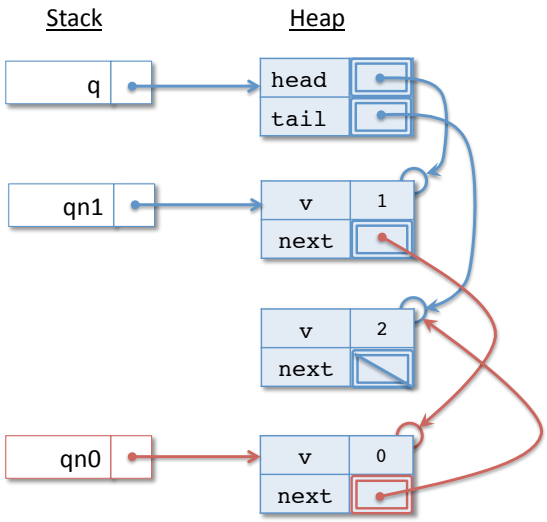
- *1 code to put q on stack ("let q") and allocate queue record*
- *2 code to put qn1 on stack ("let qn1")* after *q on stack*

- *1 allocate record with v=1*
- *1 allocate record with v=2, next = None*
- *1 q.head points to Some qn1*
- *1 update qn1.next*
- *1 q.tail points to Some (second queue node)*
- *2 no stack variable for qn2*

c. Suppose we start executing from the ASM configuration shown on the previous page. Modify the template stack and heap diagram below to show what it will look like after the following code executes on the workspace.

```
let qn0 = { v = 0; next = q.tail } in
qn1.next <- Some qn0
```

Note that you may need to allocate new stack variables, heap nodes or add `Some` bubbles in appropriate places. If you need to erase a line, mark it clearly with an "X".



Grading Scheme: 6 points.

- *2 qn1.next reference + some bubble*
- *1 qn0 stack variable*
- *1 qnode with v=0 on heap*
- *2 qn0.next reference + some bubble*

d. Does `q` satisfy the queue invariant after the code in part (c) has executed? Circle or **false**. **true**

Grading Scheme: 3 points

4. Mutable Queues Implementation (23 points)

Implement a function, called `intersperse`, that inserts a value *between* every value in a queue.

For example, if `q` contains the values `1,2,3` (in that order) then, after an execution of `intersperse 0 q`, the queue `q` should contain the values `1,0,2,0,3` (in that order). On the other hand, if `q` is empty or contains a single element, then a call to `intersperse` should not modify the queue. All calls to `intersperse` should leave `q` in a valid state.

Your implementation may define a single recursive helper function to traverse the queue. However it may not call any other functions, such as `from_list`, `to_list`, `deq` or `enq`.

```
let intersperse (x : 'a) (q : 'a queue) : unit =
  let rec loop (n : 'a qnode) : unit =
    begin match n.next with
    | None -> ()
    | Some nn ->
      n.next <- Some { v = x; next = Some nn };
      loop nn
  end in
  begin match q.head with
  | Some h -> loop h
  | None -> ()
  end
```

or

```
let intersperse (x :'a) (q : 'a queue) : unit =
  let rec loop (no : 'a qnode option) : unit =
    begin match no with
    | None -> ()
    | Some n ->
      begin match n.next with
      | Some nn ->
        let newnode = { v = x; next = Some nn } in
        n.next <- Some newnode;
        loop (Some nn)
      | None -> ()
      end
  end in
  loop q.head
```

Note: this next one is not quite correct as it compares two `Some` nodes with `==`. It should read `not (n.next == None)` instead. Or use pattern matching like above.

```
let intersperse (x : 'a) (q : 'a queue) : unit =
  let rec loop (qn: 'a qnode option) : unit =
    begin match qn with
     | None -> ()
     | Some n -> if not (qn == q.tail) then
                   let nxt = n.next in
                   n.next <- Some {v=x; next = nxt}; loop nxt
                 else ()
   end in
   loop q.head
```

Grading Scheme: 23 points total. Additions:

- *2 inner loop present, with appropriate type annotations*
- *3 type correct initial call to inner loop with q.head (or first qnode).*
- *4 allocate new qnode with v=x*
- *3 new qnode's next is to correct node*
- *4 update current node's next to point to new next*
- *5 correct recursive call to loop (2 if no newnode)*
- *2 return () for None*
- *(rest) no deductions*

Deductions:

- *putting an extra node at the end of the queue: -3*
- *putting an extra node at the end of the queue and not updating the tail: -5*
- *changing* `q.head`*: -2*
- `tail.next` *is not* `None`*: -2*
- *not using* `Some`*: -2*
- *using options without pattern matching (n.next.next): -3*
- *inappropriate use of reference equality (with options): -3*
- *infinite loop: -5*
- *correct, but uses prohibited helper functions: -12*
- *only allocates 1 new node: -7*
- *wrong value for new node: -1*

5. Listeners (18 points)

Recall the `value_controller` type from Homework 6.

```
type 'a value_controller = {
  get_value : unit -> 'a;
  set_value : 'a -> unit;
  add_change_listener : ('a -> unit) -> unit;
}
```

A value controller is an "object" that encapsulates a particular value. The methods of this object allow applications to access the value, change the value, and register change listeners. The listeners are called with the new value whenever the value is changed.

For example, if `vc` is an `int value_controller`, then the following code

```
vc.add_change_listener (fun (v:int) ->
       print_endline ("Value changed to: " ^ (string_of_int v)));
vc.set_value 10;
vc.set_value 20
```

prints the following text on the console when executed

```
Value changed to: 10
Value changed to: 20
```

On the next page, implement `make_value_controller`. This function should return a new value controller when given an initial value.

You are allowed to use the following helper function in your answer, but no other library functions.

```
let rec iter (f : 'a -> unit) (x : 'a list) : unit =
  begin match x with
  | [] -> ()
  | hd :: tl -> f hd ; iter f tl
  end
```

```
let make_value_controller (v : 'a) : 'a value_controller =
    let value = { contents = v } in
    let listeners = { contents = [] } in
    {
      add_change_listener =
        (fun f -> listeners.contents <- f :: listeners.contents);
      get_value =
        (fun () -> value.contents);
      set_value =
        (fun v' -> value.contents <- v';
         iter (fun f -> f v') listeners.contents)
    }
```

Grading Scheme: 18 points total

- ***a.*** *2 - mutable ref for value,*
- ***b.*** *1 - initialize value ref to v*
- ***c.*** *2 - mutable ref for listeners,*
- ***d.*** *1 - initialize listener ref to []*
- ***e.*** *1 - add_change_listener present and correct type*
- ***f.*** *2 - add_change_listener updates listeners*
- ***g.*** *1 - get_value present and correct type*
- ***h.*** *2 - get_value returns current value*
- ***i.*** *1 - set_value present and correct type*
- ***j.*** *1 - set_value updates value*
- ***k.*** *4 - set_value iterates through listeners*

Appendix A: OCaml Linked Queue implementation

```
type 'a qnode = { v : 'a; mutable next : 'a qnode option; }

type 'a queue = {
 mutable head : 'a qnode option;
 mutable tail : 'a qnode option;
}

let create () : 'a queue =
  { head = None; tail = None }

let is_empty (q:'a queue) : bool =
  q.head = None

let enq (x:'a) (q:'a queue) : unit =
  let newnode_opt = Some { v = x; next = None} in
  begin match q.tail with
    | None -> q.head <- newnode_opt;
     q.tail <- newnode_opt
    | Some qn2 ->
     qn2.next <- newnode_opt;
     q.tail <- newnode_opt
  end

let deq (q:'a queue) : 'a =
  begin match q.head with
    | None -> failwith "error: empty queue"
    | Some qn ->
     begin match q.tail with
       | Some qn2 ->
        if qn == qn2 then
          (* deq from 1-element queue *)
          (q.head <- None;
           q.tail <- None;
           qn2.v)
        else
          (q.head <- qn.next;
           qn.v) (* Make sure to use parens around ; expressions. *)
       | None -> failwith "invariant violation"
     end
  end


let to_list (q : 'a queue) : 'a list =
  let rec loop (qn : 'a qnode option) (acc : 'a list) : 'a list =
   begin match qn with
     | None -> List.rev acc
     | Some qn1 -> loop qn1.next (qn1.v :: acc)
   end in
  loop q.head []
```

Appendix B: Example Abstract Stack Machine Diagram

An example of the Stack and Heap components of the OCaml Abstract Stack Machine. Your diagram should use similar "graphical notation" for `Some v` and `None` values.

```
(* The types for mutable queues. *)
type 'a qnode = { v : 'a; mutable next : 'a qnode option; }

type 'a queue = {
  mutable head : 'a qnode option;
  mutable tail : 'a qnode option;
}

let qn1 : int qnode = {v = 1; next = None}
let qn2 : int qnode = {v = 2; next = Some qn1}
let q : int queue = {head = Some qn2; tail = Some qn1}
(* HERE *)
```

The OCaml program above yields the ASM Stack and Heap depicted below when the program execution reaches the point marked *(* HERE *)*.