CIS 120 Standard exam II

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 mutable.

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 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 Java, if `s` and `t` are variables of type `String` such that `s == t`, then `s.equals(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) (m: int list) : int list =
  if x = 0 then m else loop x (x :: m)
```

h. [✓ T] F The `loop` function above is tail recursive.

i. [✓ 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 -> x + 1) :: x.contents)
```

j. [✓ 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 stored `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. ASM, structural and reference equality (24 points total)

Consider the code and ASM shown in Appendix B on page 12. For convenience, you may carefully remove the Appendices from the main exam.

a. (2 points) Does `q` satisfy the queue invariant given in class?

Circle: [✓ **yes**] or **no**.

If your answer is no, also briefly explain how the invariant is violated below.

b. (10 points) For each of the following expressions, use the heap diagram to circle whether they evaluate to **`true`**, **`false`**, *loop forever*, or *do not type check.*

i. `qn2.next == q.tail`

`true` [✓ **`false`**] *loops forever* *doesn't typecheck*

ii. `qn2.next = q.tail`

[✓ **`true`**] **`false`** *loops forever* *doesn't typecheck*

iii. `q.head.next = q.tail`

`true` **`false`** *loops forever* [✓ *doesn't typecheck*]

iv. `qn1 = q.tail`

`true` **`false`** *loops forever* [✓ *doesn't typecheck*]

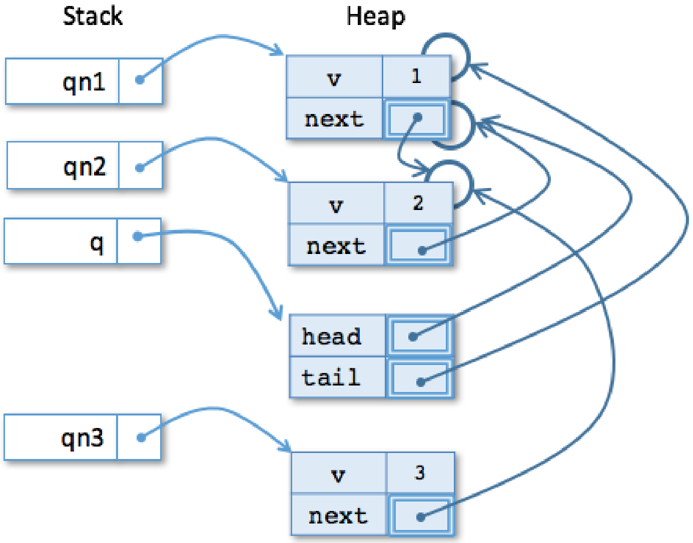
v. `qn2 == qn2`

[✓ **`true`**] **`false`** *loops forever* *doesn't typecheck*

c. (10 points) Now, suppose the following code is placed on the workspace, starting from the configuration shown in the diagram in Appendix B on page 12.

```
let qn3 : int qnode = { v = 3; next = Some qn2 } in
q.head <- qn2.next;
qn1.next <- qn3.next
```

Complete the missing parts of the diagram below, showing the final state of the heap after these operations have executed. Don't forget to draw your "Some bubbles" clearly!



Grading Scheme: 2 points per box

d. (2 points) Does `q` satisfy the queue invariant given in class after this code has executed?

Circle: **yes** or [✓ **no**].

If your answer is no, also briefly explain how the invariant is violated below.

Answer: `q.tail` *points to* `Some qn1`*, but* `qn1.next` *is not* `None`.

3. Queue implementation (20 points)

This problem uses the OCaml mutable queue data structures shown in Appendix A.

Complete the implementation of a function `insert_before q x y`, which adds a new element `x` to a queue `q` *immediately before* the first occurrence of a given element `y`.

For example, if the queue `q` contains the elements `1,2,4` (in that order), then `insert_before q 3 4` should modify `q` so that it contains the elements `1,2,3,4` (in that order). If the given value `y` is not present, then `q` should not be modified. More test cases demonstrating the behavior of this function appear below.

You should use structural equality (`=`) when comparing values in the queue for equality with `y`. Your answer **may not** use functions in Appendix A (such as `enq` or `deq`) or list library functions.

```
;; run_test "insert_before" (fun () ->
      let q = from_list [1;2;4] in
      insert_before q 3 4;
      to_list q = [1;2;3;4])

;; run_test "insert_before beg" (fun () ->
      let q = from_list [1;2;3] in
      insert_before q 0 1;
      to_list q = [0;1;2;3])

;; run_test "insert_before none" (fun () ->
      let q = from_list [1;2;3] in
      insert_before q 5 4;
      to_list q = [1;2;3])

      ;; run_test "insert_before empty" (fun () ->
      let q = from_list [] in
      insert_before q 5 4;
      to_list q = [])
```

(Use the next page for your answer.)

Complete the implementation below. Don't forget to fill in a type annotation for `prev`!

```
let rec insert_before (q : 'a queue) (x : 'a) (y : 'a) =
  let rec loop (prev : 'a qnode) (qno : 'a qnode option) : unit =
    begin match qno with
    | Some qn -> if qn.v = y then
        prev.next <- Some {v = x; next = qno}
      else
        loop qn qn.next
    | None -> ()
    end in
  begin match q.head with
  | Some qn ->
      if qn.v = y then
        q.head <- Some { v = x; next = q.head }
      else loop qn qn.next
  | None -> ()
  end
```

or

```
let insert_before (q : 'a queue) (x : 'a) (y : 'a) : unit =
  let rec loop (prev : 'a qnode option) (qno : 'a qnode option) : unit =
    begin match qno with
      | Some qn ->
          if qn.v = y then
            let new_q = Some { v = x; next = qno } in
            begin match prev with
              | None -> q.head <- new_q
              | Some n -> n.next <- new_q
            end
          else loop qno qn.next
      | None -> ()
    end
  in
  loop None q.head
```

Grading Scheme: 2 points each

- *type for* `prev`*, consistent with implementation*
- *Check qn.v for equality with y (both at head and each qno)*
- *Create new qnode containing x (both at head and each qno)*
- *new qnode's next reference is qno (or Some qn) for each qno*
- *new qnode's next reference is q.head (or Some qn)*
- *update prev.next when value at qno*
- *update q.head when value at first node*
- *recursive call to loop with correct arguments*
- *initial call to loop with correct arguments*
- *do nothing at end of queue*

Other errors at discretion.

4. (20 points) **Object encoding**

The Java class `Unique`, shown below, implements a generator for unique strings sharing some common prefix. If this common prefix is not specified, then the prefix `"x"` is used instead. (This class might be used in a compiler to generate temporary variable names.)

```
public class Unique {

  private final String prefix; // cannot be changed after initialization
  private int i = 0;      // updates with each new String generated

  public Unique (String p) {
    if (p == null) {
      prefix = "x";
    } else {
      prefix = p;
    }
  }

  // generates a new string by incrementing the index and concatenating it to
  // the prefix string
  public String generate() {
    i = i+1;
    return (prefix + Integer.toString(i));
  }
}
```

For example, we can use this class to generate unique strings as follows:

```
Unique utemp = new Unique("temp");
String s0 = utemp.generate(); // returns ”temp1”
String s1 = utemp.generate(); // returns ”temp2”

Unique ux = new Unique(null);
String t0 = ux.generate(); // returns ”x1”
String t1 = ux.generate(); // returns ”x2”
```

Your job in this problem is to translate the `Unique` class into an OCaml “object”, making sure to encapsulate the private state and provide equivalent functionality. In particular, you should define an OCaml type, called `unique`, corresponding to the type of `Unique` objects and a function `make_unique` that can construct values of this type. You may also wish to define a type of `local_state` for use in the `make_unique` function.

Recall also that `string_of_int` converts integers to strings in OCaml and that `^` is the OCaml operator for string concatenation.

(Use the next page for your answers.)

```
type unique = { generate : unit -> string }

type local_state = { prefix : string ; mutable i : int }

let make_unique (p : string option) : unique =
 let s = { prefix = begin match p with
   Some p -> p | None -> "x" end ; i = 0 } in
 { generate = (fun () ->
    s.i <- s.i + 1;
    s.prefix ^ string_of_int i)
 }
```

Grading Scheme: 2 points each:

- *unique contains generate function of type* `unit -> string` *or* `unit -> string option`
- `local_state` *contains mutable int, should be encapsulated, can't be in* `unique`
- `prefix` *string is not mutable, may or may not be stored in* `local_state`
- `make_unique` *takes string option as argument*
- `make_unique` *initializes int (outside of* `generate` *function)*
- `make_unique` *uses pattern matching to select the correct prefix string (outside of* `generate` *function). No credit for comparison with* `null`*.*
- `make_unique` *creates the* `unique` *record, containing* `generate` *function*
- `generate` *function type matches type definition in* `unique`
- `generate` *function increments int, no credit for increment during* `make_unique`
- `generate` *function creates correct string, no credit for string creation during* `make_unique`

5. Higher-order programming (14 points total)

Suppose the following two functions are added to the mutable queue module, shown in Appendix A.

```
let fold1 (combine : 'a -> 'b -> 'b) (base : 'b) (q: 'a queue) : 'b =
      let rec loop qno =
            begin match qno with
            | None -> base
            | Some qn -> combine qn.v (loop qn.next)
            end in
      loop q.head

let fold2 (combine : 'a -> 'b -> 'b) (base : 'b) (q: 'a queue) : 'b =
      let rec loop qno acc =
            begin match qno with
            | None -> acc
            | Some qn -> loop qn.next (combine qn.v acc)
      end in
      loop q.head base
```

a. (4 points) Fill in the value computed for `ans` in the following code snippet (or write *infinite loop* if the code does not terminate).

```
let q = from_list [1;2;3;4] in
let ans = fold1 (fun x y -> x :: y) [] q in


ans = ______[1;2;3;4]______________________________________
```

b. (4 points) Fill in the value computed for `ans` in the following code snippet (or write *infinite loop* if the code does not terminate.)

```
let q = from_list [1;2;3;4] in
fold2 (fun x _ -> enq x q) () q;
let ans = to_list q in


ans = ______infinite loop________________________________
```

(Problem continues on next page.)

c. (6 points) Implement a `transform` function for queues using either `fold1` or `fold2`. This function should have the following behavior: when given a queue `q` containing the numbers `1,2,3,4` (in that order), the call `transform (fun x -> x + 1) q` should produce a *new queue* containing `2,3,4,5` (in that order) and should not modify `q`.

You *may* use other queue functions defined in Appendix A in your definition, but you *must* use either `fold1` or `fold2`. You may not define a recursive helper function in your solution.

```
let transform (f : ’a -> ’b) (q : ’a queue) : ’b queue =
  let q2 = create () in
  fold2 (fun x _ -> enq q2 (f x)) () q;
  q2

or

let transform (f : ’a -> unit) (q : ’a queue) : ’b queue =
  let l = fold1 (fun x y -> f x :: y) [] q in
  from_list l
```

A Appendix: 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 ->
        q.head <- qn.next;
        (if qn.next = None then q.tail <- None);
        qn.v
  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 []

let from_list (xs : 'a list) =
   let q = create () in
   List.iter (fun x -> enq x q) xs;
   q
```

B Appendix: Example Abstract Stack Machine Diagram

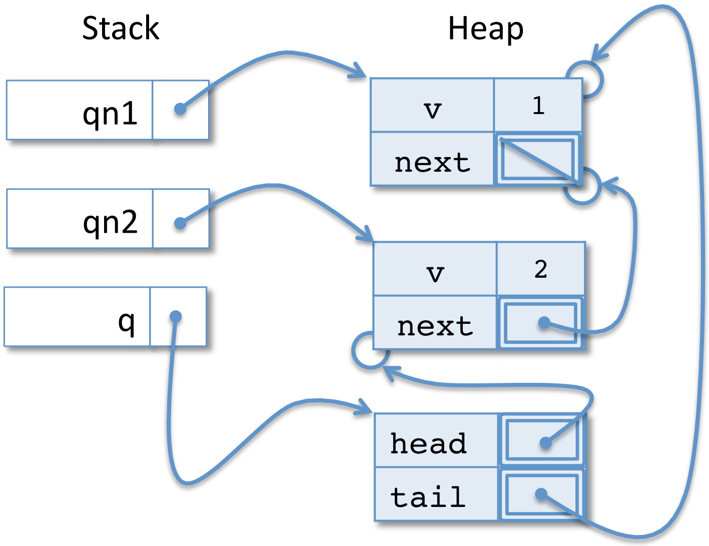
An example of the Stack and Heap components of the OCaml Abstract Stack Machine. Your diagram should use similar "graphical notation" for `Some v` and `None` values.

```
(* 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 ∗)*.


Stack
Heap
qn1
qn2
q
v
1
next
v
2
next
head
tail