MPRI 2:30: Semester II Exam (F* Part)

Question 1 (Abstract queue) Consider the following module implementing (FIFO) queues using lists, but keeping this representation of queues and all the operations abstract.

```
module AbstractQueue
  abstract type queue = list int
  abstract val is_empty : queue → bool
  let is_empty = Nil?
  abstract val empty : q:queue{is_empty q}
  let empty = []
  abstract val enq : int → queue → q:queue{¬(is_empty q)}
  let enq x xs = Cons x xs
  abstract val deq : q:queue{¬(is_empty q)} → queue
  let rec deq xs = match xs with
                    | [x] → []
                    | x1::x2::xs → x1 :: deq (x2::xs)
  abstract val front : q:queue{¬(is_empty q)} → int
  let rec front xs = match xs with
                      | [x] → x
                      | x1::x2::xs → front (x2::xs)
```

To allow client code to use this interface and reason about the results we export lemmas providing an algebraic specification of queues. For instance, the following front_enq lemma

```
let front_enq (i:int) (q:queue) : Lemma (front (enq i q) = (if is_empty q then i else front q)) = ()
```

allows us to prove the following assertion *outside* the AbstractQueue module:

```
assert(front (enq 3 (enq 2 (enq 1 empty))) = 1)
```

(a) How many instances of the front_enq lemma are needed to show the assert above? Write down these instances.

(b) Write another lemma that can be exposed by the AbstractQueue module to prove the assertion:

```
assert(deq (enq 3 (enq 2 (enq 1 empty))) == enq 3 (enq 2 empty))
```

Question 2 (Imperative count) Write down a valid specification for the following stateful function:

```
let rec count r (n:nat) : ST unit (requires (λ h0 → ..........................................................................))

                                   (ensures (λ h0 () h1 → ................................................................)) =
  if n > 0 then (r := !r + 1; count r (n − 1))
```

Your specification should capture what this function does in terms of the usual sel and upd specification-level functions:

```
val sel : #a:Type → heap → ref a → GTot a
val upd : #a:Type → heap → ref a → a → GTot heap
```

and can assume the usual theory of arrays for sel and upd.

Question 3 (Removing from lists) The following F* function removes all occurrences of an integer y from a list xs:

```
let rec remove (y:int) (xs:list int) : Tot (list int) (decreases xs) =
  match xs with
  | [] → []
  | x :: xs' → if x = y then remove x xs' else x :: remove y xs'
```

We can use remove to implement another function that removes all elements of a list ys from a list xs:

```
let rec remove_list (ys:list int) (xs:list int) : Tot (list int) (decreases ys) =
  match ys with
  | [] → xs
  | y'::ys' → remove y' (remove_list ys' xs)
```

You will have to prove that remove_list indeed removes all the elements of ys from xs. More precisely, given a function that counts all occurrences of an element y in a list xs:

```
let rec count (y:int) (xs:list int) : Tot nat (decreases xs) =
  match xs with
  | [] → 0
  | x :: xs' → (if x = y then 1 else 0) + count y xs'
```

prove the following main lemma:

```
val count_remove_list (y:int) (ys:list int) (xs:list int) :
    Lemma (requires (count y ys > 0)) (ensures (count y (remove_list ys xs) = 0))
```

Write down any intermediate lemmas you use in your proof in F* syntax and prove those as well. The proofs of the main and intermediate lemmas can be in whatever syntax you like (F*, math, a mix) and for each of these proofs make it explicit over what you are doing induction, what is the case structure, where you are using other lemmas, etc.