local
  val counter : int ref = ref 0
in
  fun tick (): int = (counter := !counter + 1; !counter)
  fun reset (): unit = (counter := 0)
end;

fun new_counter (): (unit → int) * (unit → unit) =
  let
    val counter : int ref = ref 0
    fun tick (): int = (counter := !counter + 1; !counter)
    fun reset (): unit = (counter := 0)
  in
    (tick, reset)
  end;

val c1 = new_counter ();
val c2 = new_counter ();

fun do_tick (x: unit → int, _: unit → unit): int = x();
fun do_reset (_: unit → int, x: unit → unit): unit = x();
do_tick c1;
do_tick c1;
do_reset c1;
do_tick c1;
do_tick c2;

fun rot3 (a, b, c) =
  let
    val t = !a
  in
    a := !b; b := !c; c := t
  end
val x1 = ref 1;
val x2 = ref 2;
val x3 = ref 3;
(!x1, !x2, !x3);
rot3 (x1, x2, x3);
(!x1, !x2, !x3);
rot3 (x1, x2, x3);
(!x1, !x2, !x3);
rot3 (x1, x2, x3);
(!x1, !x2, !x3);
rot3 (x1, x2, x3);
(!x1, !x2, !x3);

val a1 = ref 1;
val a2 = ref 3;
val a3 = a1;
(!a1, !a2, !a3);
rot3 (a1, a2, a3);
(!a1, !a2, !a3);
rot3 (a1, a2, a3);
(!a1, !a2, !a3);

fun rot3' (a, b, c) =
    let
        val (x, y, z) = (!a, !b, !c)
    in
        a := y; b := z; c := x
    end

val b1 = ref 1;
val b2 = ref 3;
val b3 = b1;

(!b1, !b2, !b3);
rot3' (b1, b2, b3);
(!b1, !b2, !b3);
rot3' (b1, b2, b3);
(!b1, !b2, !b3);

structure Array =
    struct
        type 'a array = 'a ref list
        fun array (0: int, v: 'a): 'a array = []
            | array (n: int, v: 'a): 'a array = (ref v)::(array (n-1, v))

        fun tabulate (n: int, f: int -> 'a): 'a array =
            let
                fun range (0: int): int list = [0]
                    | range (n: int): int list = n::(range (n-1))
            in
                List.map (fn n => ref (f (n))) (range n)
            end

        fun fromList (l: 'a list): 'a array =
            List.map (fn v => ref v) l

        fun sub (a: 'a array, i: int): 'a = !(List.nth (a, i))

        fun update (a: 'a array, i: int, v: 'a): unit =
            (List.nth (a, i)) := v

        fun length (a: 'a array): int = length a

        fun modify (f: 'a -> 'a) (a: 'a array): unit =
            List.app (fn r => r := f (!r)) a

        fun app (f: 'a -> unit) (a: 'a array): unit =
            List.app (fn r => f (!r)) a

        fun foldr (f: 'a * 'b -> 'b) (v: 'b) (a: 'a array): 'b =
            List.foldr (fn (r, v) => f (!r, v)) v a
    end;

val a = Array.fromList ["a", "b", "c"];
Array.sub (a, 2);
Array.update (a,1,"p");

(* Signature of a FIFO queue *)
signature QUEUE =
  sig
    (* Abstract type representing the queue of elements of type 'a *)
    type 'a q

    (* Exception raised when trying to dequeue from an empty queue *)
    exception Empty

    (* Value of an empty queue *)
    val empty : unit → 'a q

    (* Adds an element to the queue *)
    val enqueue : 'a q * 'a → 'a q

    (* Removes an element from the queue *)
    val dequeue : 'a q → 'a q * 'a
  end

(* Implementation of a FIFO queue using mutation *)

(* Abstraction function: A queue is represented as a linked list of *)
(* cells which can be altered using mutation (linked list). *)
(* The representation are references to the first and last elements *)
(* of the list. The queue corresponding to such a representation is *)
(* the one made of the elements contained in the linked list. *)

(* Representation invariants: The front is a reference to the first *)
(* element of the linked list. The back is a reference to the last *)
(* (empty) element of the list. *)

structure MutationQueue :> QUEUE =
  struct
    (* A cell is made of a value and a pointer to the next or NIL *)
    datatype 'a cell = NIL | CELL of 'a * 'a cell ref

    (* Define a type for a reference to a cell *)
    type 'a cref = 'a cell ref

    (* Representation of the abstract type 'a q *)
    (* It is a part of cell references, corresponding to the front *)
    (* and the back of the linked list *)
    type 'a q = 'a cref * 'a cref

    (* Exception raised when trying to dequeue from an empty queue *)
    exception Empty

    (*
    NOTE: Value restriction
    
    Error: explicit type variable cannot be generalized at its
    binding declaration: 'a
    
    val empty': 'a q =
    let
      val r = ref NIL
    *)
(* Value of an empty queue *)
fun empty (): 'a q =
  let
    val r = ref NIL
  in
    (r, r)
  end

(* Adds an element to the queue *)
fun enqueue ((f,b): 'a q, v: 'a): 'a q =
  let
    val b' = ref NIL
    val _ = b := CELL (v, b')
  in
    (f, b')
  end

(* Removes an element from the queue *)
fun dequeue (ref NIL, b): 'a q = 'a q * 'a =
  raise Empty
| dequeue (ref (CELL (v, f')), b): 'a q = 'a q * 'a =
    ((f', b), v)
end;

structure M = MutationQueue;

(*
NOTE: Value restriction

Warning: type vars not generalized because of
value restriction are instantiated to dummy types (X1,X2,...)
*)
val q1' = M.empty ();

val q1 = M.enqueue (q1', 1);

val q2 = M.enqueue (q1, 1);

val q3 = M.enqueue (q2, 2);

val (q4, v1) = M.dequeue q3;

val (q5, v2) = M.dequeue q4;

val q6 = M.enqueue (q5, 3);

val (q7, v3) = M.dequeue q6;

val (q4', v1') = M.dequeue q3;

val (q5', v2') = M.dequeue q4';

val (q6', v3') = M.dequeue q5';