Assignment 2

Due by Friday, February 19, at 5pm
Submission Via Gradescope

Fill in the four gaps in the following EASYCRYPT file, Assignment2.ec, which is available on the course website. Make sure EASYCRYPT is able to check your proofs.

(* ASSIGNMENT 2
Due on Gradescope by 5pm on Friday, February 19 *)

require import AllCore.

(* NOTE: in the following Hoare Logic proofs, you may *not* use the tactics ‘auto’ or ‘sp’, which we haven’t covered in Lab or the slides yet. You *may* use ‘smt’. *)

(* Any uses of the ‘smt’ tactic in your proof must be solvable by *both* Alt-Ergo and Z3: *)

prover quorum=2 ["Alt-Ergo" "Z3"].

(* QUESTION 1 (* 20 Points *))

module Swap = {
  var x, y : int
  proc f() : unit = {
    var z : int;
    z <- x;
    x <- y;
    y <- z;
  }
}.

lemma swapping (_x _y : int) :
  hoare
  [Swap.f :
    Swap.x = _x \ Swap.y = _y ==> Swap.x = _y \ Swap.y = _x].
proof.
(* BEGIN FILL IN *)

(* END FILL IN *)
qed.

(* QUESTION 2 (40 Points) *)

module M = {
    var x, y, z : int

    proc f() : unit = {
        if (x < y) {
            z <- x - y;
            if (x <= y) {
                while (false) {
                }
            }
        }
        else {
            z <- y - x - 1;
        }
    }
}.

lemma M1 :
    hoare [M.f : true ==> M.z < 0].
proof.
(* BEGIN FILL IN *)

(* END FILL IN *)
qed.

lemma M2 :
    hoare
        [M.f : 
            true ==> 
            (M.x < M.y => M.z = M.x - M.y) /
            (M.y <= M.x => M.z + 1 = M.y - M.x)].
proof.
(* BEGIN FILL IN *)

(* END FILL IN *)
qed.
(* QUESTION 3 (40 Points) *)

require import List.

(* This introduces a new type constructor, 'a list, which means that
for any type t, t list is the type of finite lists of elements of
type t. Lists are written [x1; x2; ...; xn]. E.g., *)

op xs = [1; 3; 5; 7].

(* is the value of type int list consisting of the first four odd
natural numbers.

If x has type 'a and ys has type 'a list, then x :: ys is the value
of type 'a list whose first element (head) is x, and whose
remaining elements (its tail) are those of ys. (We pronounce ::
"cons", for "construct".) E.g., *)

op zs = 1 :: [3; 5; 7].

lemma eq_xs_zs : xs = zs.
proof. by rewrite /xs /zs. qed.

(* size : 'a list -> int returns the number of elements of a
list. E.g., *)

lemma size_ex : size zs = 4.
proof. smt(). qed.

(* If we have lists xs and ys of type 'a list, then xs ++ ys is the
concatenation of xs and ys, i.e., the list consisting of the
elements of xs followed by the elements of ys. E.g., *)

op ws = xs ++ [9].

lemma ws_lem : ws = [1; 3; 5; 7; 9].
proof. smt(). qed.

(* rcons : 'a list -> 'a -> 'a list ("reverse cons") takes
xs : 'a list and y : 'a and returns xs ++ [y]. rcons is defined
recursively, and you can see how this is done by doing

print rcons. *)
proof.
  (*
  search rcons (++)
  *)
  by rewrite -cats1.
qed.

(* nth : 'a -> 'a list -> int -> 'a takes def : 'a, xs : 'a list, and
   i : int, and returns
   (+) the ith element of xs (counting from 0), if 0 <= i < size xs;
   (+) the default element, def, if i < 0 \ size xs <= i

E.g., *)

lemma nth_ex1 : nth (-2) ws 3 = 7.
proof. smt(). qed.

lemma nth_ex2 : nth (-2) ws (-1) = -2.
proof. smt(). qed.

lemma nth_ex3 : nth (-2) ws 5 = -2.
proof. smt(). qed.

(* take : 'a list -> int -> 'a list takes in a list xs and an integer
   n, and returns the list consisting of the first n elements of xs
   (it returns [] if n is negative, and returns xs if more then size
   xs elements are requested). E.g., *)

lemma take_ex1 : take 3 ws = [1; 3; 5].
proof. by rewrite /ws /xs. qed.

lemma take_ex2 : take (-1) ws = [].
proof. trivial. qed.

lemma take_ex3 : take 6 ws = [1; 3; 5; 7; 9].
proof. by rewrite /ws /xs. qed.

(* drop : 'a list -> int -> 'a list takes in a list xs and an integer
   n, and returns the list consisting of what’s left over if we remove
the first n elements of xs (it returns [] if more than size xs
   elements are dropped, and returns xs if n is negative). E.g., *)

lemma drop_ex1 : drop 3 ws = [7; 9].
proof. by rewrite /ws /xs. qed.

lemma drop_ex2 : drop (-1) ws = ws.
proof. trivial. qed.

lemma drop_ex3 : drop 6 ws = [].
proof. by rewrite /ws /xs. qed.

(* rev : 'a list -> 'a list reverses a list. E.g., *)

lemma rev_ex : rev ws = [9; 7; 5; 3; 1].
proof. by rewrite /ws /xs. qed.

(* You can search for combinations of (::), (++), size, rcons, nth,
   take, drop and rev to find numerous useful lemmas, which you can
   tell smt to try to use or you can use directly via apply or
   rewrite. *)

module Rev = {
   proc f(xs : int list) : int list = {
      var i : int;
      var ys : int list;
      i <- 0;
      ys <- [];
      while (i < size xs) {
         ys <- nth 0 xs i :: ys;
         i <- i + 1;
      }
   return ys;
   }.

lemma Rev_rev (_xs : int list) :
   hoare [Rev.f : xs = _xs ==> res = rev _xs].
proof.
   (* BEGIN FILL IN *)
   (* END FILL IN *)
qed.