1 Sequential Programs – Sorting

One way to analyze sequential programs in PVS is construct models that represent those programs as recursive functions. We explore this approach here, using *Bubble Sort* as a running example. Sequential sorting entails rearranging the content of an array "in place." So we shall also be exploring how to do that in PVS. Section 1.1 presents with a sorting program and its semi-formal correctness proof. Section 1.2 discusses how the program is translated to PVS. Section 1.3 lists a initial PVS formulation of specification, implementation and correctness.

1.1 Informal Proof of *Bubble Sort*

Below is a sequential program claimed to perform a bubble sort. Let us first fix an input array content A and bounds L and U. Since the sorting is done in place, the input array a is changed in the course of the program. So the initial value $\mathsf{A}[\mathsf{L} .. \mathsf{U}]$ is just an expression representing the inital ordering of a within index range $[\mathsf{L} .. \mathsf{U}]$, where it is implicitly understood that $\mathsf{L} \leq \mathsf{U}$..

```
{ a[L..U] = A[L..U] }
begin
k := U;
while L < k {INV_2 ≡?}
   begin
   j := L;
   while j < k {INV_1 ≡?}
      begin
      if a[J] > a[J+1]
         then a[J], a[J+1] := a[J+1], a[J]
         else skip;
      j:= j+1
      end;
   k := k − 1
   end
end
{ sorted?(a[L..U], A[L..U]) }
```

The postcondition consists of two parts,

$$sorted?(a[\mathsf{L} .. \mathsf{U}],\ \mathsf{A}[\mathsf{L} .. \mathsf{U}]) \equiv \begin{cases} permutation?(a[\mathsf{L} .. \mathsf{U}],\ \mathsf{A}[\mathsf{L} .. \mathsf{U}]) \\ \wedge \\ ordered?(a[\mathsf{L} .. \mathsf{U}]) \end{cases}$$

Saying that the elements of $a[\mathsf{L} .. \mathsf{U}]$ just a rearrangement of the elements of $\mathsf{A}[\mathsf{L} .. \mathsf{U}]$, and in addition, the elements of $a[\mathsf{L} .. \mathsf{U}]$ are in non-decreasing order.

An explanation of how the program works might read, "The inner loop moves the largest value in the range $a[\mathsf{L} .. k]$ into $a[k]$. The outer loop then

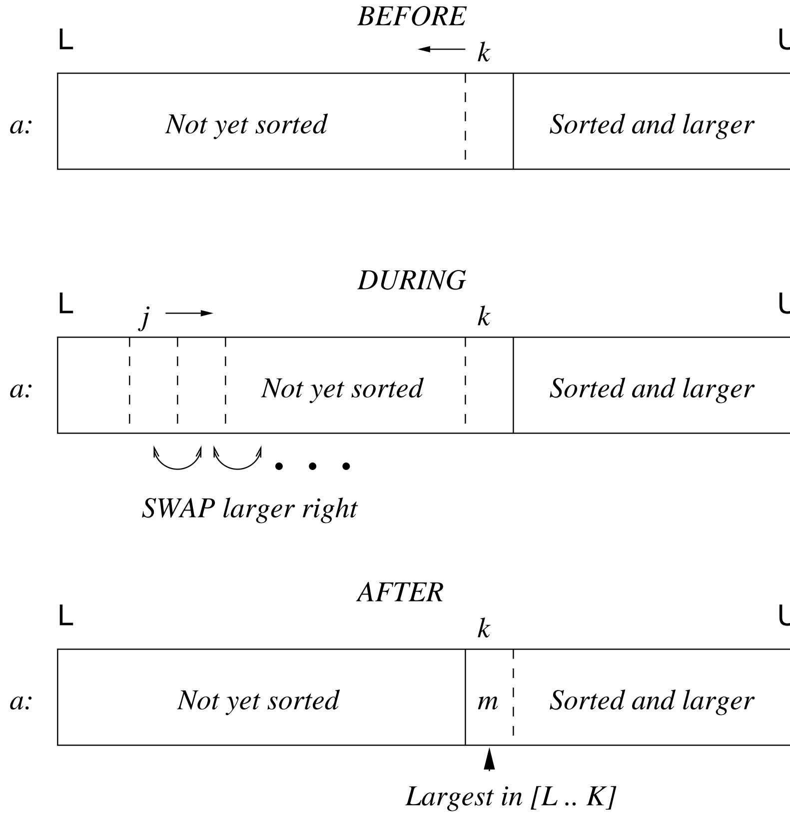

Figure 1: Bubble Sort Diagrams

sorts the elements in the range $a[k \mathrel{..} \mathsf{U}]$, decrementing k by 1 until it reaches L. This description is often accompanied by diagrams (Fig. 1), depicting what is happening "during" the loop, that is, while $\mathsf{L} < k < \mathsf{U}$:

1.1.1 Formulation of Invariants

The diagrams are actually expressions of loop invariants; however, as always, these pictures leave a lot unsaid. The outer loop's invariant says that a is partially sorted.

$$\textsc{inv}_2^\dagger \equiv \forall (k \leq i < \mathsf{U})\colon a(i) \leq a(i+1)$$

The inner loop's, invariant says that in addition to $\textsc{inv}_2^\dagger$, that the largest value in $a[\mathsf{L} \mathrel{..} j]$ is at index j. Formally,

$$\textsc{inv}_1^\dagger \equiv \textsc{inv}_2^\dagger \wedge \forall (\mathsf{L} \leq i < j)\colon a(i) \leq a(j+1)$$

Two support logical analysis, the invariants must "maintain" certain conditions in order to sustain the *permuation*? property.

$\text{PERM}(l, u) \equiv$

(a)	$\forall (i < l \vee u < i)\colon a'(j) = a(j)$	*a's content* outside *the "current" bounds is unchanged.*
(b)	$\mathit{permutation?}(a'[l\ ..\ u],\ a[l\ ..\ u])$	*Content* inside *"current" bounds is permuted.*
(c)	$\mathit{permutation?}(a'[\mathsf{L}\ ..\ \mathsf{U}],\ \mathsf{A}[\mathsf{L}\ ..\ \mathsf{U}])$	*a preserves the content of* A.

where a' denotes the effect of the loop's body.

Depending on how the proof is done, condition (c) may be subsumed by (a) and (b). Condition (a), saying the loops *have no side effects*; and (b), saying initial content is preserved; are irksome. Such intuitively obvious details are ignored textbook explanations, of course. However, they are necessary in a formal proof. So the actual loop invariants will look something like:

$$\begin{array}{lcll} \text{INV}_1 & \equiv & & \forall (\mathsf{L} \leq i < j)\colon a(i) \leq a(j+1) \quad \wedge\ \text{PERM}(\mathsf{L}, j) \\ \text{INV}_2 & \equiv & \text{INV}_1 \wedge & \forall (k \leq i < \mathsf{U})\colon a(i) \leq a(i+1) \quad \wedge\ \text{PERM}(k, \mathsf{U}) \end{array}$$

Both INV_1 and INV_2 are derivable from the post-condition using the method of replacing a constant by a variable, together with "common knowledge" about *sortedness*.

1.2 PVS Formulation

One approach to verifying the sorting algorithm is to formulate and prove the "informal proof" in Section 1.1 by synthesizing verfication in the logic of program correctness assertions. The difficulties lie in the manual translation and dealing with array assignment, which we did not see in the previous homework exercise.

This section centers on the alternative approach of modeling the program as a system of recursive function. The accompanying PVS source file contains an intermediate-level formulation of a sequential algorithm for *Bubble Sort*. I've tried to strike a balance between a naive logical representation and a more generic one. It may be illuminating to step through some of these proofs while reading the discussions below.

1.2.1 Order of Results

The resulting `sort.pvs` file is an artifact of the proof process in which the accumlated definitions, axioms, lemmas and theorems are listed in dependence order. In other words, just as in ordinary mathematical discourse, the definitions, etc., make up an organized *explanation* that in no way reflects the chronological order in which supporting lemmas were introduced to solve subproblems arising in the proof *process*. And, of course, all the mistakes and blind alleys are not mentioned.

1.2.2 Range Restriction and Measure Induction

All of the inductive arguments are based on the size of the array region, that is the difference $u - l$ between upper and lower bounds of the region. There are numerous ways to set this up, of course, and the consequences of the set-up can manifest themselves as unexpected sub-goals or TCCs. In particular, we want to avoid cases in which the range is negative, that is, the lower bound exceeds the upper bound.

- A seemingly straightforward approach is to explicitly restrict the range in the premises of all theorems,

  $$\forall l, u \colon \mathbb{N}, l \leq u \Rightarrow \cdots$$

  Although this can be made to work, it induces distracting proof sub-goals, and complicates the specification of `MEASURE` terms in recursive definitions.

- On can use dependent subtyping to restrict the upper bound to be greater than the lower bound. For example, the inner loop of the program might be modeled as `BS_inner_loop` is defined,

  ```
  InnerLoop(a, l:nat, u:{i:nat | l <= i}, a):
  RECURSIVE ARRAY [nat -> int] = ...
  ```

  Theorems are likewise parameterized,

  ```
  FORALL (l:nat):  FORALL (u:{i:nat | l <= i}, a)): ···
  ```

  Not only is this more in the spirit of an ordinary mathematical explanation, but it also discharges the range restrictions as TCCs, most of which are automatically proven. When it is necessary to invoke the restriction on `u` in a proof, it is readily done with `typepred`.

- See the parameterized `subrange` type in the `Prelude` is defined

  ```
  subrange(l, u):  TYPE = {k | l <= k AND k <= u}
  ```

  includes a built-in theory of induction, but it cannot be used directly in function calls, nor does it restrict `l` $\leq$ `k`/

- In the PVS source file accompanying this tutorial, record type used:

  ```
  RNG: TYPE = [# l:nat, u:{i:nat | l <= i}#]
  RNG_sz(r:  RNG): nat = r`u - r`l
  ```

  It isn't obvious that this approach is any better than dependent typing, but it can be made to work.

1.2.3 Measure Induction

The `measure-induct+` command is used for doing induction over terms, rather than single variables. In the case of Bubble Sort, those terms express the size of the range over which a function is operating. In the accompanying PVS file induction is invoked with

```
(measure-induct+ "r‘u-r‘l" ("r"))
```

where `r` is a variable of type `RNG`.[1]

The `measure-induct` principle takes the form of a stong induction, so no subgoal for the base case is generated. Nevertheless, the proof will inevitably compel base case(s), so it is usually a good idea to discharge it immediately.

1	`(measure-induct+ "r‘u-r‘l" ("r"))`	*invoke induction, introducing skolem constant* ***x!1*** *of type* `RNG`
2	`(case "x!1‘u = x!’1")`	*discharge the base case.*
2.1	*. . . direct proof . . .*	*typically does not use the induction hypothesis*
2.2	`(skolem` f `"R")`	*Induction case,* $l \neq u$*.*
	⋮	
	`(inst` $i.h.$ t_l t_u`)`	*Suitable instantiation of the induction hypothesis*
	⋮	

1.2.4 Algorithm Models

PVS file `sort.pvs` uses standard techniques for translating sequential programs to their models as recursive functions. Each loop is developed as an iterative (tail-recursive) function. The program in Section 1.1 translates to

```
BS_inner_loop(a, r): RECURSIVE ARRAY [nat -> int] =
   IF r‘l = r‘u
   THEN a
   ELSE IF a(r‘l) > a(r‘l + 1)
        THEN BS_inner_loop( swap(a, r‘l, r‘l+1), (# l:= r‘l + 1, u:=r‘u #))
        ELSE BS_inner_loop( a,                   (# l:= r‘l + 1, u:=r‘u #))
       ENDIF
   ENDIF
   MEASURE RNG_sz(r) by <

BS_outer_loop(a, r): RECURSIVE ARRAY [nat -> int] =
  IF r‘u = r‘l
  THEN a
```

[1] `(measure-induct+ "RNG_sz(r), ("r"))` would work just as well.

```
    ELSE BS_outer_loop( BS_inner_loop(a, r), (# l:= r`l, u := r`u - 1 #))
    ENDIF
    MEASURE RNG_sz(r)  by <

  BSort(a, r): ARRAY [nat -> int] = BS_outer_loop(a, r)
```

In general, this is not entirely a mechanical translation, but it is straightforward and could be mechanized.

1.3 A Starting Formulation

```
sort: THEORY
BEGIN

RNG: TYPE = [# l:nat, u:i:nat | l <= i #]
RNG_sz(r:RNG):nat = r`u - r`l

%%%
%%% Reserve some variable names to save having to type them "in line".
%%%

a, b:    VAR ARRAY [nat -> int] %
r, s:    VAR RNG                %
z:       VAR int                % always used for some item in an array
l, u:    VAR nat                % always used for lower bound of a range
k, j, i: VAR nat                % always used as quantified index within a range

%%%
%%% Specification
%%% A region is sorted if it's elements are in non-decreasing order
%%% and the result is a simple rearrangement of the original elements.

ordered?(a, r):bool =
  forall( j,k:subrange(r`l, r`u)): j<=k IMPLIES a(j) <= a(k)

occurrences(z, a, r): RECURSIVE nat =
 (IF (a(r`l) = z) THEN 1 ELSE 0 ENDIF)
 +
 (IF (r`l = r`u)
    THEN 0
    ELSE occurrences(z, a, (# l:= (r`l + 1), u:= r`u #))
 ENDIF)
 MEASURE r`u - r`l

permutation?(a, r, b, s):bool =
  forall z: occurrences(z, a, r) = occurrences(z, b, s)
```

```
sorted?(a, r, b, s): boolean =
  ordered?(a, r)
  AND
  permutation?(a, r, b, s)

%%%%
%%%% PVS model of the sequential BubbleSort program
%%%%

swap(a:[nat->int], j,k:nat): ARRAY[nat -> int] =
  a WITH [(j):= a(k), (k):= a(j)]

BS_inner_loop(a, r): RECURSIVE ARRAY [nat -> int] =
   IF r`l = r`u
   THEN a
   ELSE IF a(r`l) > a(r`l + 1)
        THEN BS_inner_loop( swap(a, r`l, r`l+1), (# l:= r`l + 1, u:=r`u #))
        ELSE BS_inner_loop( a,                   (# l:= r`l + 1, u:=r`u #))
       ENDIF
   ENDIF
   MEASURE RNG_sz(r) by <

BS_outer_loop(a, r): RECURSIVE ARRAY [nat -> int] =
  IF r`u = r`l
  THEN a
  ELSE BS_outer_loop( BS_inner_loop(a, r), (# l:= r`l, u := r`u - 1 #))
  ENDIF
  MEASURE RNG_sz(r)  by <

BSort(a, r): ARRAY [nat -> int] = BS_outer_loop(a, r)

%%%
%%% Correctness
%%%

BSort_sorts: THEOREM
  FORALL (a, r): sorted?(BSort(a, r), r, a, r)

END sort
```

1.4 Scale of the Exercise

The diagram below shows the number of lemmas introduced and their dependences to prove the *Ordered?* property of `BubbleSort`. It is included to give you an idea of the scale of the this proof exercise.