Sorting Arrays

1 Informal Proof of Bubble Sort

Here 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 a constant expression of the original ordering of a.

```
{ L ≤ U }
{ a[L..U] = A[L..U] }
begin
k := U;
while L < k {INV₂ ≡?}
   begin
   j := L;
   while j < k {INV₁ ≡?}
      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
{ permutation?(a[L..U], A[L..U]) }
{ sorted?(a[L..U] }
```

An explanation of how it works might read, "The inner loop moves the largest value in the range $a[\mathsf{L} .. k]$ into $a[k]$. The outer loop then sorts the elements in the range $a[\mathsf{U} .. k-1]$, diminishing k by 1 until it reaches L. This description is often accompanied by some diagrams (Fig. 1), depicting what is happening "during" the loop, that is, while $\mathsf{L} < k < \mathsf{U}$:

1.1 Formulation of Invariants

The diagrams are actually expressions of loop invariants[1] However, these pictures leave a lot unsaid. For the inner loop, that invariant says, in part, that the largest value is at index $j+1$. Formally,

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

[1] Both $\textsc{inv}_2$ and INV_2 are derivable from the post-condition using the method of replacing a constant by a variable, using some additional knowledge about *sorted*?.

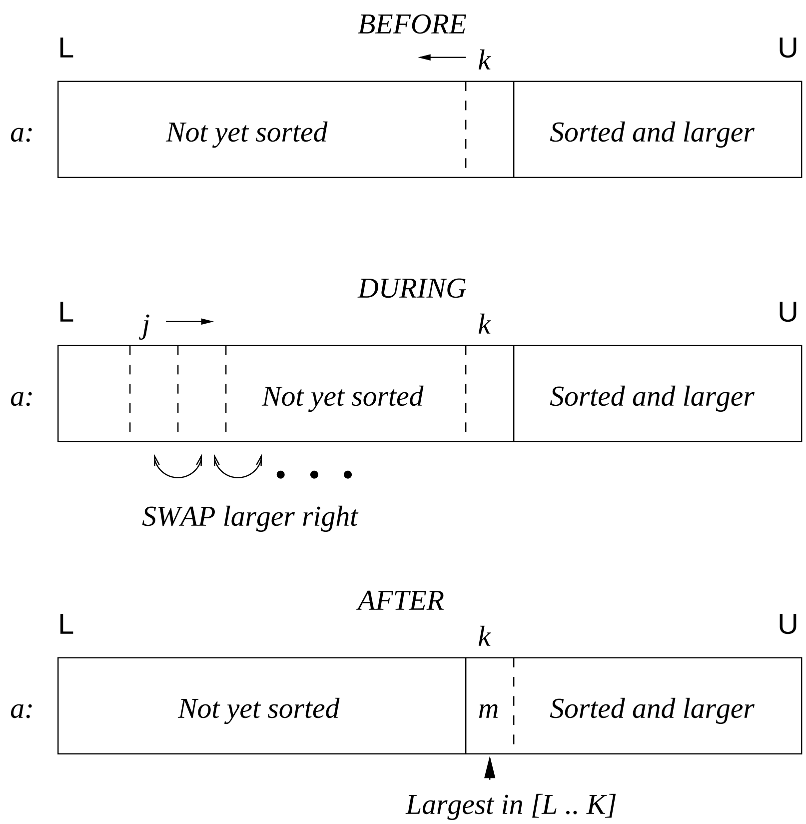

Figure 1: Bubble Sort Diagrams

The outer loop's invariant says that a is partially sorted.

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

Two support logical analysis, the invariants must "carry" additional information through the program, 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)	$\mathit{permutation?}(a'[l \mathrel{..} u],\ a[l \mathrel{..} u])$	*Content* inside *"current" bounds is permuted.*
(c)	$\mathit{permutation?}(a'[\mathsf{L} \mathrel{..} \mathsf{U}],\ \mathsf{A}[\mathsf{L} \mathrel{..} \mathsf{U}])$	*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. However, they are essential in a formal proof. So the actual loop invariants will look something like:

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

2 PVS Formulation

The accompanying source file, `sorting_tutorial_2.pvs`, 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. Proofs, some partial, are included in `sorting_tutorial_2.prf`. It may be illuminating to step through some of these proofs while reading the discussions below.

2.1 Order of Results

The `.pvs` file is an artifact of the proof process in which definitions, axioms 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 sub-problems arising in the proof *process*. And, of course, the mistakes and blind alleys are not documented.

Figure 2 shows a partial dependence graph (proof-chain) for `sorting_tutorial_2.prf`. It is partial in two respects: first, the proof dependencies do not include TCCs and results in the `Prelude`; second, the proof status is not complete, so some dependencies are yet to be discovered. The boxed nodes are roughly the starting point in the correctness formulation. Oval nodes are new definitions and theorems introduced (so far) during proof development.

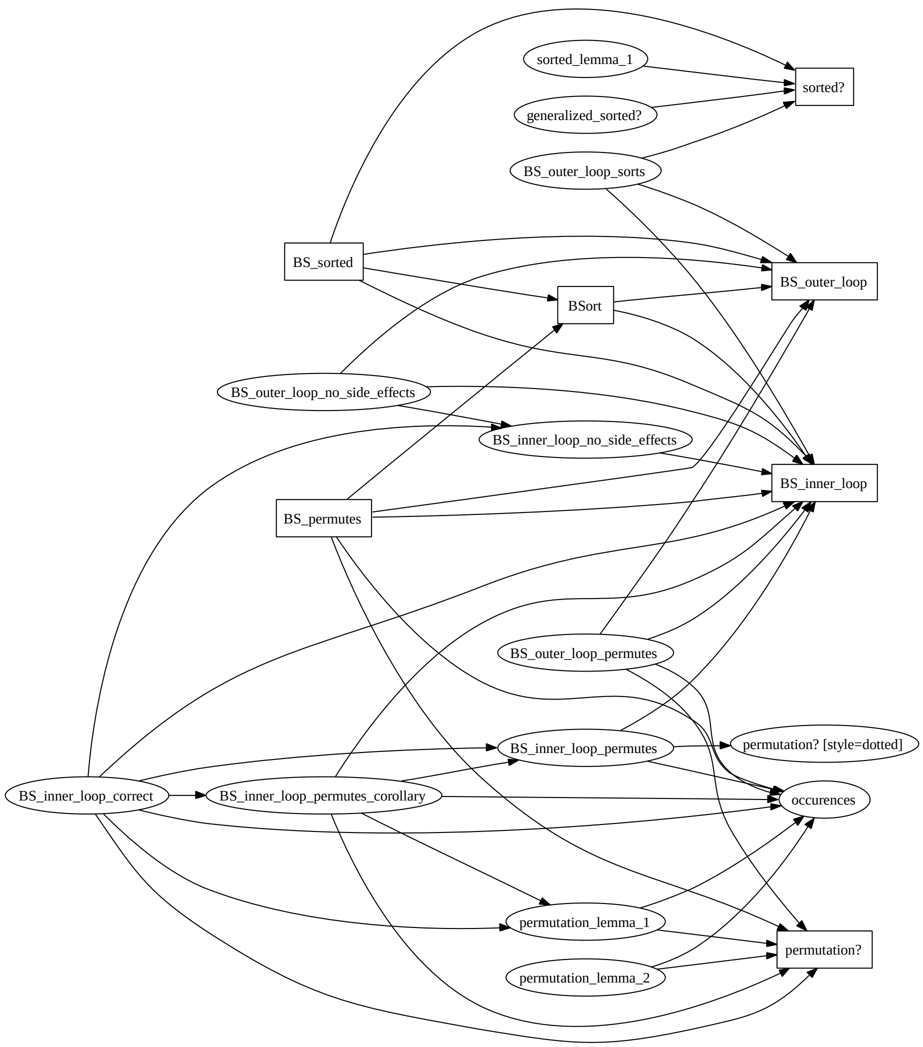

Figure 2: Partial dependence graph for `sorting_tutorial_2.prf`

2.2 Range Restriction and Measure Induction

Formal verification of *Bubble Sort* is based the formulation of the algorithm, and its specification. All of the inductive arguments are based on the "size" of the array region. Most proofs depend on the *range*—the distance between upper and lower bounds—getting smaller. 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.

- In the tutorial PVS file, I use dependent typing to gain some control over some of the complications.[2] For example, `BS_inner_loop` is defined,

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

  Subsequent theorems are likewise parameterized,

  ```
  FORALL (l:nat, 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` during a proof, it can be done readily with `TYPEPRED`.

- 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 subgoals, and complicates the specification of `MEASURE` terms in recursive definitions.[3]

- See the `subrange` type and `subrange_inductions` theory in the Prelude.

- One could declare a record type of upper- and lower-bound pairs, e.g.

  ```
  range:  [# l:nat, u:{i:nat | l <= i}#]
  ```

  to modularize and abbreviate parameter type declarations.

- *and so on.*

It is worthwhile experimenting with different ways to deal with array-range specifications, early on. However, the final formulation should use a consistent logical representation throughout.

[2] This form of parameter typing is not used in some of the earlier definitions of `sorting_tutorial_2.pvs`. Because of this, there are unprovable TCCs, such as `occurrences_TCC1` which asserts $\forall l, u \in \mathbb{N}\colon u - l \geq 0$, that indicate an inconsistent formulation.

[3] See footnote 2.

2.2.1 Measure Induction

In keeping with the way ranges are declared, inductions (and function `MEASURE`s) involve not on a single variable, but the rather the *term* `(u - l)`. A principle called `measure-induction` is provided by PVS, invoked by

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

that instantiates the general principle according to the type of the expression and possibly multiple induction variables.

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 a good idea to discharge it immediately. The generic sequence of proof commands is:[4]

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

2.3 Algorithm Models

PVS file `sorting_tutorial_2.pvs` uses standard techniques for translating sequential programs to their models as recursive functions. Each loop is developed as an iterative function, for instance, program fragment

```
while j < k
   begin
   if a[J] > a[J+1]
      then a[J],  a[J+1] := a[J+1],  a[J]
      else skip;
   j:= j+1
   end;
end
```

[4]It may eventually be worthwhile to encode this as a `pvs-strategy`

Translates to

$$
\begin{array}{l}
\mathit{BS\_inner\_loop}(l\colon \mathbb{N}, u\colon \{i\colon \mathbb{N} \mid l \leq i\}, a)\colon \\
\text{RECURSIVE ARRAY } [\mathbb{N}- > \mathbb{Z}] = \\
\quad \text{IF } l = u \\
\qquad \text{THEN } a \\
\quad \text{ELSE IF } a[l] > a[l+1] \\
\qquad \text{THEN } \mathit{BS\_inner\_loop}(l+1, u, \text{SWAP}(a, l, l+1) \\
\qquad \text{ELSE } \mathit{BS\_inner\_loop}(l+1, u, a) \\
\mathit{measure}\ u - l\ \mathit{by} <
\end{array}
$$

$$\mathit{where}\ \mathit{swap}(a,i,j) \equiv a \text{ WITH } [(i) := a(j),\ (j) := a(i)]$$

In general, this is not entirely a mechanical translation, but it is straightforward and could be mechanized.[5]

2.4 Theory of *Sortedness*

As the program proofs develop, you are likely to find a need to teach PVS much more than you anticipate about the *sorted?* and *permuation?*. When you are in the middle of a proof, it is very tempting to "push things through," but this is almost always a mistake. It is far better to stop and introduce lemmas about how basic operations like swapping and range-restriction preserve these properties. For instance, the lemma below says that swapping within range preserves the *permutation?* property.

LEMMA. $\forall l \leq j, k \leq u$ *let*

$$\mathit{swap}(a,j,k) \equiv a \text{ WITH } [(i) := a(j),\ (j) := a(i)]$$

Then permuation?$(a, \mathit{swap}(a,j,k), l, u)$.

Lemmas of this kind should be generalized to the extent that they do not refer to algorithms being verified. It may be better to modularize by developing a separate PVS Theory for specification properties. Assign one team member to this task, while another works on implementation proofs; but swap roles from time to time.

A truly generic Theory of sorted-ness would abstract content (`int`) to a type parameter, and '$\leq$' to an arbitrary total order. It might even abstract from arrays to some generic sorting structure. I strongly recommend against going that far in the direction of genericity, at least until the implementation proofs for ARRAYs of `int`s are finished.

[5] One goal of this project is to learn more about logical representation techniques, which includes formulating models that interact well with verification proofs.