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 $A[L..U]$ is just a constant expression of the original ordering of $a$.

$$
\text{begin} \\
\quad \{ \text{L \leq U} \} \\
\quad \text{begin} \\
\quad \quad k := U; \\
\quad \quad \text{while} \ L < k \ \{ \text{INV\_2 \equiv ?} \} \\
\quad \quad \quad \text{begin} \\
\quad \quad \quad \quad j := L; \\
\quad \quad \quad \quad \text{while} \ j < k \ \{ \text{INV\_1 \equiv ?} \} \\
\quad \quad \quad \quad \quad \text{begin} \\
\quad \quad \quad \quad \quad \quad \text{if} \ a[J] > a[J + 1] \\
\quad \quad \quad \quad \quad \quad \quad \text{then} \ a[J], a[J + 1] := a[J + 1], a[J] \\
\quad \quad \quad \quad \quad \quad \quad \text{else} \ \text{skip}; \\
\quad \quad \quad \quad \quad \quad j := j + 1 \\
\quad \quad \quad \quad \quad \text{end} \\
\quad \quad \quad \quad k := k - 1 \\
\quad \quad \quad \text{end} \\
\quad \text{end} \\
\quad \{ \text{permutation?(a[L..U], A[L..U])} \} \\
\quad \{ \text{sorted?(a[L..U])} \} \\
\text{end}
$$

An explanation of how it works might read, “The inner loop moves the largest value in the range $a[L..k]$ into $a[k]$. The outer loop then sorts the elements in the range $a[U..k - 1]$, 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 $L < k < U$:

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,

$$
\text{INV\_1} \equiv \forall (L \leq i < j): a(i) \leq a(j + 1)
$$

\(^1\)Both $\text{INV\_2}$ and $\text{INV\_2}$ are derivable from the post-condition using the method of replacing a constant by a variable, using some additional knowledge about $\text{sorted?}$. 

1
Figure 1: Bubble Sort Diagrams
The outer loop’s invariant says that \( a \) is partially sorted.

\[
\text{INV}_2^1 \equiv \forall (k \leq i < U): a(i) \leq a(i + 1)
\]

Two support logical analysis, the invariants must “carry” additional information through the program, in order to sustain the \textit{permutation} property.

\[
\text{PERM}(l, u) \equiv
\begin{align*}
(a) & \quad \forall (i < l \lor u < i): a'(j) = a(j) \quad \text{a’s content outside the “current”} \\
\text{bounds is unchanged.} \\
(b) & \quad \text{permutation?}(a'[l..u], a[l..u]) \quad \text{Content inside “current” bounds is} \\
\text{permuted.} \\
(c) & \quad \text{permutation?}(a'[L..U], A[L..U]) \quad a \text{ preserves the content of} \ A.
\end{align*}
\]

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 \textit{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:

\[
\begin{align*}
\text{INV}_1 & \equiv \forall (L \leq i < j): a(i) \leq a(j) \land \text{PERM}(L,j) \\
\text{INV}_2 & \equiv \left[ \forall (k \leq i < U): a(i) \leq a(i + 1) \land \text{PERM}(k,U) \right]
\end{align*}
\]

2 \quad PVS Formulation

The accompanying source file, \texttt{sorting\_tutorial\_2.pvs}, contains an intermediate-level formulation of a sequential algorithm for \textit{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 \texttt{sorting\_tutorial\_2.prf}. It may be illuminating to step through some of these proofs while reading the discussions below.

2.1 \quad 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 \textit{explanation} that in no way reflects the chronological order in which supporting lemmas were introduced to solve sub-problems arising in the proof \textit{process}. And, of course, the mistakes and blind alleys are not documented.

Figure 2 shows a partial dependence graph (proof-chain) for \texttt{sorting\_tutorial\_2.prf}. It is partial in two respects: first, the proof dependencies do not include TCCs and results in the \texttt{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 on 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. For example, BS\_inner\_loop is defined,

  \[
  \text{BS\_inner\_loop}(l:\text{nat}, u:\{i:\text{nat} \ | \ l \leq i\}, a):
  \text{RECURSIVE ARRAY} \ [\text{nat} \rightarrow \text{int}] =
  \text{IF } l = u \cdots
  \]

  Subsequent theorems are likewise parameterized,

  \[
  \text{FORALL} \ (l:\text{nat}, u:\{i:\text{nat} \ | \ l \leq i\}, a): \cdots
  \]

  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,

  \[
  \forall l, u: \mathbb{N}, l \leq u \Rightarrow \cdots
  \]

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

- 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.

  \[
  \text{range: } [\# l:\text{nat}, u:\{i:\text{nat} \ | \ l \leq i\}]\#
  \]

  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.

\footnote{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\_TCC which asserts \(\forall l, u \in \mathbb{N}: u - l \geq 0\), that indicate an inconsistent formulation.}

\footnote{See footnote 2.}
2.2.1 Measure Induction

In keeping with the way ranges are declared, inductions (and function MEASUREs) involve not on a single variable, but the rather the term \((u - 1)\). A principle called measure-induction is provided by PVS, invoked by

\[
\text{(measure-induct+ "u-l" ("l" "u"))}
\]

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 strong 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:

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

2.3 Algorithm Models

PVS file sorting_tutorial2.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

```plaintext
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

```
BS_inner_loop(l: \mathbb{N}, u: \{i: \mathbb{N} \mid l \leq i\}, a):
  recursive array [\mathbb{N} \rightarrow \mathbb{Z}]=
  if l = u
    then a
    else if a[l] > a[l + 1]
      then BS_inner_loop(l + 1, u, swap(a, l, l + 1))
      else BS_inner_loop(l + 1, u, a)
  measure u - l by <

where \text{swap}(a, i, j) \equiv a \text{ with } [(i) := a(j), (j) := a(i)]
```