AdaCore | Build Software that Matters

Tech Paper

Quicksort Verified by SPARK

DAVID EASTERLING,
University of Dayton Research Institute, USA

adacore.com

1 About The Author

David Easterling is an engineer with more than 25 years of programming experience in a host of languages, primarily C, C++, Java, Fortran95, and Python. He works for the University of Dayton Research Institute, contracting to the Verification and Validation of Complex and Autonomous Systems team in the Aerospace Systems team in the Aerospace Systems Directorate of the Air Force Research Laboratory.

2 Introduction

Recently, I endeavored to start learning Ada and SPARK for a new research task, focusing on the process of developing verifiable programs for autonomous applications. In order to learn the intricacies of the language, I took on a task to implement the Quicksort algorithm and to prove properties on its functional behavior. I chose Quicksort for its simplicity and its role in the heart of algorithmic theory – the rare implementation with a poor worst-case performance but highly desirable standard-case. Quicksort is used in a wide variety of applications, and as such having an implementation verified to be bug-free and correct could be quite useful.

While having a bug-free implementation is the threshold of this effort, some thought was taken in what level of rigor to attain for the SPARK version of Quicksort. There are increasing levels of rigor [1] allowed by analyses that SPARK enables. These levels referenced here are often referred to as bronze, silver, gold, and platinum, with platinum referring to full functional verification. The goal of this effort is to produce a platinum level implementation of Quicksort as a way to build a knowledge base with the SPARK suite of automatic verification tools. While existing permutation proofs, such as the one in the SPARK User's Manual, could have short-circuited the process, the goal was to develop such a proof from scratch. For the sake of simplicity it is assumed that this implementation is only accepting integer type variables within a narrowly defined subset, although the same proofs will apply for any orderable arrays as inputs, and the algorithm has no optimizations for large data sets.

In Section 3, the Quicksort algorithm will be described, detailing the properties to be proven for this implementation with SPARK. Section 4 develops the variable types to be used in the implementation for context. Section 5 develops the functions that are not used directly in the program flow but are used in the proofing steps. Section 6 details the specification of the Quicksort implementation, QS(). Section 7 describes the operations involved in implementing and proving the Partition step of the Quicksort algorithm. Section 8 describes the operations involved in implementing and proving the Recursive step of the Quicksort Algorithm. Finally, Section 9 concludes with a discussion on the lessons learned in this process.

3 Quicksort

3.1 A brief introduction to Quicksort.

A sorting algorithm that is widely used for its low growth rate related to problem size in the average case, Quicksort can be divided into three basic operations. The first operation is a partition of all elements around a pivot element... usually the element in a preselected position of the array to be sorted, although more advanced strategies exist.

The second operation is to recursively sort the low part of the array. The final operation is to recursively sort the elements in the high part of the array in the same way.

Figure 1. Quicksort Pseudocode

```
procedure quicksort(array, low, high):
   if low < high:
     pivotIndex = partition(array, low, high)
     quickSort(array, low, pivotIndex - 1)
     quickSort(array, pivotIndex + 1, high)
```

The partitioning strategy utilized here is Lomuto's partitioning scheme[2], choosing the last element of the input array as the "pivot" around which the array is to be recursively sorted.

Figure 2. Partition Pseudocode

```
function partition(array, low, high):
  pivot = array[high]
  k = low
  for i = low to high - 1:
    if array[k] <= pivot:
      swap array[k] with array[i]
      k = k + 1
  swap array[k] with array[pivot]
  return k
```

In the pseudocode presented in Figures 1 and 2, "low" is the index of the first element of the region of the array "array" to be sorted, and "high" is the index of the last element. Similarly, the "swap" operation simply exchanges the location of two elements of "array." At the end of the partitioning operation, "array" has three distinct sections: the chosen "pivot" is in the correct location, while every element before it in "array," those with lesser indices, has a lower value than "pivot" and every element after it in the array, with greater indices, has a higher value than "pivot."

3.2 Properties of the Quicksort Implementation.

In creating a SPARK implementation of the Quicksort algorithm, I endeavored to maintain four properties of that implementation. These properties are the absence of runtime exceptions, termination of the recursive execution, the output array being a permutation of the input array and output array resulting in a total ordering of the elements of the input array from lowest to highest.

3.2.1 Absence of Runtime Exceptions.

For a SPARK implementation to have an absence of runtime exceptions means that there must be no computation errors (overflow, divide-by-zero, etc.), memory access errors, or any other problems that might demonstrate proper syntax but would result in an exception at runtime. Any code being checked by SPARK ("in SPARK mode," achieved by utilizing the SPARK_Mode pragma) will be checked to ensure that no runtime exceptions will be generated.

The absence of runtime exceptions ("AoRTE") is the hallmark of the silver level of proof. Maintaining a silver-level Quicksort implementation requires the use of preconditions to limit input arguments to a computable range, both for QS() itself and for the Swap procedure (to prevent bad memory accesses).

3.2.2 Termination.

A termination proof for recursion requires two elements: A base case or cases that will always converge when used as the argument for a function, and a condition that any case will monotonically transform to that base case as inputs derived from that case are recursively called. In the case of QS(), the base case is any input with less than two elements, as these are

already sorted and will result in a return of the input. The monotonic property in this case exists because one element, the pivot element, is never sorted on the recursive call, so the recursion always reduces in length by at least one element from the original input; therefore, the length of the input array strictly decreases to the base case across recursive calls.

Since proving recursive termination is so formulaic, SPARK has a straightforward statement that is used with recursive functions that states the condition that must hold for termination to be proven. The Subprogram_Variant statement includes both the monotonic direction of the termination and the variable that follows this relationship (in our case, the length of our input array). SPARK will check that this condition holds in any program path that includes the recursive call, and thus will guarantee that the recursion will not recur indefinitely.

3.2.3 Permutation.

It is desired that any output of QS() is a permutation of the input, which means that each element of the input is present in the output with equal cardinality to the input; more intuitively, a permutation is simply a reordering. Permutations are always reflexive and transitive. The SPARK User's Guide [3] includes the development of a rigorous proof of permutation directly based on the counts of elements in the proof of a different sorting algorithm, Selection Sort. This implementation deliberately avoids copying that approach in order to develop an independent proof of permutation, which utilizes the concept of permutation maps.

Permutation maps are maps of one array to its permutations, and are simply arrays of indices that map each element of the initial array to a single position in a permutation. Stating that array "A" is a permutation of array "B" is equivalent to stating the existence of at least one permutation map that maps "B" to "A". While permutations are reflexive, these maps are not, so utilizing this fact in an implementation to prove permutation requires careful ordering of arrays to ensure validity. SPARK will verify the construction of a permutation map across QS()'s operations to prove that QS()'s output is a permutation of its input.

3.2.4 Ordering.

The output array of QS() should increase monotonically. Since the base case of the Quicksort recursion (an array with two or more elements) is always sorted, it suffices to show that the pivot element is sorted to the proper location in each non-base example and that larger arrays will be broken down to strictly smaller arrays around pivot elements. Induction then indicates that eventually every element will be sorted to its proper place, resulting in an ordered output.

4 Variable Types

Bearing in mind the goals and properties to be achieved by QS(), it is now time to systematically develop a QS() implementation that meets those properties. In this section, the variables that will be utilized by QS() are developed to provide context for its operations.

4.1 Index and Index_Array.

Index types are used directly in the basic implementation of Quicksort for the partitioning and swap operations, and indirectly to define the element arrays to be operated on. The range of the index type *IndexType* here is arbitrary, and may be altered to any necessary size, although QS() is not optimized for big data arrays. Index arrays of the *Index_Array* type are used exclusively for the partitioning proofs for this implementation.

Figure 3. Index Definitions

```
1     subtype IndexType is Integer range 1..2999;
2     type Index_Array is array(IndexType range <>) of IndexType;
```

4.2 Element and Element_Array.

As with indices, the range of allowed elements is arbitrary. *Element's* underlying type is also arbitrary, and can be modified to any sortable type. The *Element_Array* type is used to define the input and output types for QS().

Figure 4. Element Definitions

```
1     subtype Element is Integer range -1000 .. 1000;
2     type Element_Array is array(IndexType range <>) of Element;
```

4.3 The Permutation_Array Type.

In order to introduce the permutation map concept, it is necessary to define the conditions under which an array may be considered a permutation map. A permutation array has the following properties: it must map onto itself, and each element of the array must be unique. Any permutation array *P* that contains as its elements the indices with an arbitrary array *A* and is the same length as *A* will form a permutation of *A, B,* by mapping indices in the following manner: *A(P (I)) = B(I)*. Conversely, if B is a permutation of *A*, then there will exist at least one permutation map $P_{A->B}$ that satisfies this relationship.

4.3.1 Type definition.

For the purposes of QS(), the permutation map is proved by construction, which means that it is convenient to define a permutation array type that maintains the two properties listed.

Figure 5. Spanning Predicate

```
1  function Spanning_Predicate(Perm_Map : Index_Array) return Boolean is
2    (for all I in Perm_Map'Range => --Perm_Map(I) in Perm_Map'Range
3      (for some J in Perm_Map'Range =>
4        Perm_Map(J) = I
5      )
6    );
7  --with Ghost;
```

The spanning predicate illustrated in Figure 5 ensures that each element of the permutation array is mapped to by at least one of its indices. For simple integer index types this is sufficient for the permutation map properties already stated, but to ensure a more generally applicable type it is necessary to also establish uniqueness directly.

Figure 6. Uniqueness Predicate

```
1  function Is_Unique(Perm_Map : Index_Array) return Boolean is
2    (for all I in Perm_Map'Range =>
3      (for all J in Perm_Map'Range =>
4        (if not (I=J) then
5          Perm_Map(I) /= Perm_Map(J)
6        )
7      )
8    );
9  --with Ghost;
```

Figure 7. Permutation Array Predicate

```
function Perm_Map_Properties_Hold(Perm_Map : Index_Array) return Boolean is
  (--Uniqueness Property
  Is_Unique(Perm_Map) and then
  --Spanning Property
  Spanning_Predicate(Perm_Map)
  );
```

The uniqueness predicate presented in Figure 6 ensures that no two elements are identical in the permutation array. QS() often uses both the Spanning_Predicate() predicate and the Is_Unique() predicate simultaneously. It is thus convenient to combine them into a single predicate, Perm_Map_Properties_Hold(), illustrated in Figure 7.

Finally, the *Permutation_Array* type illustrated in Figure 8 incorporates both predicates with a simple combined function. Since all permutation arrays are by definition index arrays, it uses the *Index_Array* type as its supertype and includes the combined Perm_Map_Properties_Hold() as its only predicate. The predicates on types and subtypes are checked at assignment, including on the return from subprograms where they are declared as "out" or "in out" variables, and on declaration if there's any initialization component. The predicate of the *Permutation_Array* type guarantees that if two arrays form a permutation, then the permutation array that maps the first of them to the second is a permutation map for that permutation.

Figure 8. Permutation Array

```
subtype Permutation_Array is Index_Array with
--Predicates:  These are enforced in and out of all
--subprograms and at assignment
  Predicate => (
    Perm_Map_Properties_Hold(Permutation_Array)
  );
--To be converted to Ghost_Predicate upon ghost argument implementation
```

5 Helper Functions

QS() includes the frequent use of helper functions to bundle common operations and information that is used frequently, such as statements on preconditions and postconditions or for the logical grouping of related formalisms.

5.1 Perm_Lengths_Match.

Perm_Lengths_Match(), in Figure 9, is a helper function that enforces index equivalence between arrays that are to be checked as permutations, and serves as the precondition for the actual check on permutation, Is_Permutation().

It ensures an absence of runtime exceptions by rigorously enforcing that the indices are all identical, and, so long as the array isn't empty, the indices all fall in the legal range for index types. It is overspecified by design, as having each of these statements as unique statements that can be relied upon by the SPARK provers aids the speed of proof when any given individual one is required.

Figure 9. Perm_Lengths_Match() Declaration

```
function Perm_Lengths_Match (Left, Right : Element_Array;
  Perm_Map : Index_Array) return Boolean is
    (
    (if Perm_Map'Length > 0 then (
      Perm_Map'First >= IndexType'First and then
      Perm_Map'Last <= IndexType'Last
      )
    ) and then
      Perm_Map'Length = Left'Length and then
      Perm_Map'Length = Right'Length and then
      Right'Length = Left'Length and then
      Perm_Map'First = Left'First and then
      Perm_Map'Last = Left'Last and then
      Perm_Map'First = Right'First and then
      Perm_Map'Last = Right'Last and then
      Right'First = Left'First and then
      Right'Last = Left'Last
    ) with
      Ghost;
```

5.2 Is_Permutation.

Is_Permutation(), in figure 10, is the fundamental boolean function that determines whether or not two arrays are permutations of each other, essential for one of our four key properties for QS(). Between the restrictions of Perm_Lengths_Match() and the predicates on the *Permutation_Array* type, Is_Permutation() simply needs to affirm that the actual mapping between the two arrays is correct; if so, then a valid permutation map has been demonstrated and it can be affirmed that the two arrays form a permutation.

Figure 10. Is_Permutation() Declaration

```
function Is_Permutation(Left, Right : Element_Array;
  Perm_Map : Permutation_Array) return Boolean is
  --Perm_Map has unique entries, (Predicate of permutation maps)
  --Left and Right have the same length,
  --Perm_Map(I) spans the range of Left for I in the range of Perm_Map,
  --I spans the range of Right for I in the range of Perm_Map
  --  (logical result of spanning + uniqueness)
  --I maps each entry of Left onto Right
  (
    (for all I in Perm_Map'Range => Perm_Map(I) in Left'Range) and then
        (for all I in Perm_Map'Range => Left(Perm_Map(I)) = Right(I)
  )
)
with
  Pre => (Perm_Lengths_Match(Left, Right, Perm_Map)),
  Ghost,
  Global => Null;
```

5.3 IndexArrayOf.

The Index_Array_Of() function (Figure 11) takes in an arbitrary element array and returns an index array of equivalent size and index bounds where each index is mapped to itself. This is the identity permutation map $P_{A->A}$, since $A(P_{A->A}(I)) = A(I =)$ for all I .

IndexArrayOf()'s postconditions guarantee to SPARK that the lengths of the arrays match and that they share their first and last index, and that the index array is in fact a permutation of the input. The actual function body itself consists of a loop doing the index assignments and the Loop_Invariant verifying to SPARK the contents of the array. Broadly speaking, a Loop_Invariant can be seen as an assertion that exists only in loops and can be proved inductively.

IndexArrayOf()'s postconditions are important because they establish a base case for a permutation map that will be modified later. As long as those modifications are tracked to ensure the result remains a valid permutation, a chain can be established between a final permutation and an original array. This is precisely what needs to happen for QS() to prove that its final permutation map forms a valid permutation between its input and its output.

6 Qs Specification

Much of the context is now set to describe a specification of QS() which attempts to demonstrate the absence of runtime exceptions, termination of the implementation, total ordering of the resulting output array, and that the output array is a permutation of the input array. Of the properties discussed, there are two (the permutation and ordering properties) that are fully described in the postconditions, one that is implicit to the use of SPARK_Mode (absence of runtime exceptions), and one that is captured by the Subprogram_Variant statement (termination). The first stated postcondition ensures than any output of QS() will be sorted, while the second ensures that any output of QS() will be a permutation of the input. Thus, all four of the desired properties are covered.

7 Partition

The first step of the Quicksort algorithm is the Partition step. Leaving this step, it is necessary that four things be true: First, the pivot element must be properly placed within the array, which means that each element preceding the pivot within the array must have a value less than the pivot (the "low" property), and each element subsequent to the pivot within the array must have a value greater than or equal to the pivot (the "high" property). Secondly, there must be no runtime exceptions during the execution of this step. Third, the program must proceed through the step without exiting prematurely. Finally, the array leaving this step must be a permutation of the array that entered it.

7.2 Code Preamble.

Looking now at the implementation details, QS() begins by setting up important variables and initializing the permutation array, then asserting that it forms a permutation map from the beginning (the identity map), and establishing the conditional escape for the base case to ensure termination of any recursion. Figure 13 shows the creation and initialization of the initial variables. "Pivot" is the index of the Quicksort pivot element; "K" is the first index after the end of the "low" part of the array. "Init" is the stand-in for the input array in the QS() postcondition; it's a constant to allow SPARK to make that logical leap easily. The "A_Previous" array is also initialized to "A." The "A_Previous" array exists to store "A" before each modification of "A" so that statements about that modification can be automatically verified by SPARK. "Init," similarly, stores the input array so that statements about the input array can be made in the body of QS(), and serves the same role as SPARK's automatically defined postcondition-scope "A'Old." When statements are verified about the relationship of "A" to "Init", the same statements hold for the relationship of "A" to "A'Old" in the postcondition (and are recognized to do so by the automatic verifier): notably, that "A" is a permutation of "A'Old."

Figure 11. IndexArrayOf() Definition

```
   function IndexArrayOf(A : Element_Array) return Permutation_Array
   --A helper function that populates an array with the indices of A
   -- This is the identity case of a permutation array, which maps A to itself.
     with
     Global => Null,
     Post => (
                IndexArrayOf'Result'First = A'First
              and then IndexArrayOf'Result'Last = A'Last
              and then IndexArrayOf'Result'Length = A'Length
              and then (for all J in A'Range => IndexArrayOf'Result(J) = J)
              and then Perm_Map_Properties_Hold(IndexArrayOf'Result)
              and then Is_Permutation(A, A, IndexArrayOf'Result)
             );

   function IndexArrayOf(A : Element_Array) return Permutation_Array is
      Ind : Index_Array(A'Range) := (others => IndexType'First);
   begin
       for I in A'Range loop
         Ind(I) := I;
         pragma Loop_Invariant(for all J in Ind'First .. I => Ind(J) = J);
      end loop;
      return Ind;
   end IndexArrayOf;
```

Figure 12. QS() Specification

```
pragma SPARK_Mode (On);
--...
  procedure QS(A : in out Element_Array;
               Ghost_Perm_Map : out Permutation_Array
              )
    with
      --Converges
    Subprogram_Variant => (Decreases => A'Last-A'First),
    Pre => (A'First in IndexType'Range and then
                  A'Last in IndexType'Range and then
                 Ghost_Perm_Map'Length = A'Length and then
                   Ghost_Perm_Map'First = A'First and then
                     Ghost_Perm_Map'Last = A'Last
           ),
    Post => (  --Permutation Property
               Is_Permutation(A, A'Old, Ghost_Perm_Map) and then
               --Ascending or 1 or 0 elements property
               (if (A'Last > A'First) then
                    (for all I in A'First .. A'Last - 1 =>
                         A(I) <= A(I + 1))
               )
             );
```

Maintaining the mapping of “A” to “Init” as “A” is modified via “Ghost_Perm_Map” will thus satisfy the permutation postcondition. “Ghost_LPM” is an index array that can be modified without breaking the permutation map predicate; it will used to store updates to “Ghost_Perm_Map” in preparation of those predicates being proven to allow the assignment. In general, therefore, the program flow will consist of modifications to “A” via the Swap() procedure or QS() recursion, with the permutation map for those modifications being stored in “Ghost_LPM” and the previous state

of “A” being stored in “A’Previous.” Some work will then be done to update “Ghost_Perm_Map” to maintain it as the proper permutation map between “A” and “Init.”

Figure 14 shows the initialization of several variables after the “begin” statement. The body of QS() will then follow the steps laid out in the pseudocode. First will be the partitioning loop, which will systematically alter the contents of “A” until “A” is properly partitioned, maintaining the permutation link from “A” to “Init” in the meantime. Finally, the recursive steps will be invoked to complete the algorithm, using the postconditions of the recursion to aid in establishing the postconditions of the algorithm.

Figure 13. QS() Body - Local Variables

```
1  procedure QS (A : in out Element_Array;
2                  Ghost_Perm_Map : out Permutation_Array) is
3        Pivot       : IndexType;
4        K           : IndexType;
5        --Init is used for the permutation property preservation.
6        --A_Previous is the bridge for that property.
7        A_Previous      : Element_Array := A; --with Ghost;
8        Init      : constant Element_Array := A; -- with Ghost;
9        --Ghost_LPM receives records of permutations from Swap and Quicksort
10       --(Index_Array for partial changes)
11       Ghost_LPM : Index_Array := IndexArrayOf(Init); --with Ghost
12
```

Figure 14. QS() Body -- Initialization

```
13    begin
14       Ghost_Perm_Map := IndexArrayOf(A);
15       pragma Assert(Is_Permutation(A, Init, Ghost_Perm_Map));
16       if A'Length > 1 then -- Empty and one-element arrays are sorted.
17          Pivot := A'Last;
18          K := A'First;
19          pragma Assert(Is_Permutation(A, A_Previous, Ghost_LPM));
```

The output permutation map is set to the identity permutation map for “A,” and that permutation relationship is immediately asserted for SPARK to verify via the postconditions on Index_Array_Of() (Figure 11), using the “Init” array as the target. Maintaining this relationship between each state of “A” and “Init” is sufficient for the permutation postcondition. “Pivot” is set to the last element of “A,” an example of the Lomuto partitioning scheme [2]. “K,” as the end of the “low” end, is set to the first element of “A.” Finally, the relationship between “A” and “A_Previous” via “Ghost_LPM,” the identity permutation, is established.

The if statement (Line 16) of Figure 14 provides an escape for the base case; if the check yields false, the input array is already sorted. The partitioning and recursion steps are only invoked if the array needs further sorting after this point.

7.3 Partition Loop.

The partitioning step of Quicksort (Figure 2) involves a loop followed by a final swap to set the pivot in the correct place. QS() follows the logic listed in the pseudocode in Figure 19, swapping the element at position “K” with the element at position “I” (the loop index) if the element at “I” is less than the pivot, and then advancing “K”. By the time “I” has traversed the indices of the

input array, “K” will be in the correct position for the pivot, the element at “K” will be greater than or equal to the pivot, the pivot element will still be the last element of the array, every element of index less than “K” will be less than the pivot element and every element between “K” and the pivot element’s index will be greater than the pivot element. At that point, it only takes a swap operation between “K” and the pivot’s index to complete the partitioning. However, the properties maintained in this loop need to established for SPARK to verify them.

7.3.1 Loop_Invariant.

SPARK’s powerful Loop_Invariant pragma allows SPARK to verify information about a loop through each iteration, by verifying it first at the first iteration and then on an arbitrary iteration, given its truth in a previous iteration, by induction. Loop_Invariants are the only way to inform the solver of the “work” done by the loop. A simple rule of thumb is that if an element is unmodified by the loop its relevant properties should be contained in an assertion, if necessary, but if the element is modified by an arbitrary iteration of the loop then the relevant properties should be established in a loop invariant. The Loop_Invariant will be presented first here, for clarity.

Figure 15. Loop_Invariant - Bounded K

```
20    for I in A'First..(Pivot - 1) loop
21       pragma Loop_Invariant(
22    --K stays in "bounds"
23       K >= A'First
24       and then K <= A'Last
```

The Loop_Invariant pragma in the Quicksort partition step provided in Figure 15 maintains that “K,” the index of the element after the “low” end, stays in bounds. This ensures that “K” will not drift out of the range of “A,” regardless of the branches taken within the loop, and enables SPARK to ensure an absence of runtime exceptions regarding access to “A.”

Figure 16. Loop_Invariant - Partition Area Logic

```
25    --End of Low End <= End of High End
26      and then K <= I
```

The Loop Invariant (Figure 16) also maintains that “K” stays less than or equal to “I,” the loop variable, in order to maintain the two divisions of the partition (since “K” is the end of the “low” part and “I” is the end of the “high” part).

The Loop Invariant (Figure 17) maintains that the elements in the “low” part stay less than the pivot and elements in the “high” part stay greater than or equal to the pivot. Since the Loop_Invariant is placed at the start of the loop, SPARK will need to carry that logic forward through the last iteration of the loop in order to satisfy the ordering postcondition.

Finally, the Loop Invariant (Figure 18) ensures that “Ghost_Perm_Map” is being maintained as a permutation from the updated array to the array’s initial state after each loop iteration. This ensures the permutation postcondition is maintained throughout the loop.

7.3.2 Loop Body.

With the Loop Invariant understood, the remainder of the loop, Figure 19 can now be shown directly. The loop checks if the conditions for a swap hold, then initializes “A_Previous” to “A,” performs the swap, and then calls “Trans_Perm_Update” to update the permutation map. The swap only occurs if “K” is not equal to “I” (since nothing is gained by swapping an element with itself). “K” is then advanced.

Figure 17. Loop_Invariant - Partition Properties

```
27    --Low End has Lo Property
28      and then (for all J in A'First..K-1 =>
29        A(J) < A(Pivot))
30    --Hi End has Hi Property
31      and then (for all J in K .. I-1 =>
32        A(J) >= A(Pivot))
```

Figure 18. Loop_Invariant - Permutation Properties

```
33    -- Permutation Property preserved (Global)
34      and then
35        (Perm_Map_Properties_Hold(Ghost_Perm_Map))
36      and then (
37        Is_Permutation(A, Init, Ghost_Perm_Map)
38      )
39      );
```

Figure 19. Loop Body

```
40              if (A(I) < A(Pivot) and Pivot > K) then
41                 if (K < I) then
42                    A_Previous := A;
43                    Swap (A, Ghost_LPM, I, K);
44                    Ghost_Perm_Map :=
45                      Trans_Perm_Update(Ghost_Perm_Map, Ghost_LPM,
46                                        A, A_Previous, Init);
47                 end if;
48                 K := K + 1;
49              end if;
50           end loop;
```

There are two subprograms invoked here that are yet to be defined: Swap(), which swaps two elements within an array (and provides the permutation map between its input and output), and Trans_Perm_Update(), which updates the global permutation map "Ghost_Perm_Map" and proves that "A" and "Init" form a permutation by the transitive property. The following examines these subprograms in detail to show how they are used to establish the abscence of runtime exceptions, the relative ordering of elements and the permutation between the current "A", the previous "A", and the initial "A" input to QS().

7.3.3 Swap - Specification.

Swap() is the fundamental unit of work in the Quicksort sorting algorithm. As such, it is necessary to completely define it for SPARK.

In the code block presented in Figure 20, the preconditions ensure that there are no access errors within Swap() due to either of the swap variables "I" or "K" being out of range of "A," and that input array "A" and the permutation array "Ghost_Perm_Map" are indexed equally.

In the code block presented in Figure 21, the first postcondition ensures that Swap() generates a permutation. The second defines that only the values at the K and I indices are exchanged between the input array and the output array and that all other elements are untouched, which is used to prove the ordering Loop_Invariants of the partitioning step (and later orderings).

Figure 20. Swap() Specification

```
1     procedure Swap(A : in out Element_Array;
2                    Ghost_Perm_Map : out Permutation_Array;
3                    I, K: in IndexType)
4       with
5         Pre => (
6                   I in A'Range and then
7                     K in A'Range and then
8                       A'First = Ghost_Perm_Map'First and then
9                         A'Last = Ghost_Perm_Map'Last
10                ),
```

Figure 21. Swap() Specification - Postconditions

```
11          Post =>(
12                    --Permutation Property
13                    Is_Permutation(A, A'Old, Ghost_Perm_Map) and then
14                    --Swap Property
15                    A(K) = A'Old(I) and then A(I) = A'Old(K) and then
16                      (for all J in A'Range =>
17                         (if not((J=I) or (J=K)) then A'Old(J) = A(J))
18                      )
19                );
```

7.3.4 Swap - Body.

With the contracts in place, the crucial properties can be proved for the Swap() subprogram based on its body implementation. As seen in Figure 22, Swap() utilizes the postconditions of the IndexArrayOf() function, along with the predicate functions of a permutation array upon the final assignment (Figure 7), to ensure the permutation postcondition.

The local variable that stores the permutation map until the swap is completed, "GP_Map," must be an index array, not a permutation array, because the permutation predicates will be violated after the swap operation is begun but before it is finalized. In order to prove to SPARK that Swap properly spans, the SwapSpanning() lemma is employed (see the next section).

It is of particular convenience that there is no question that Swap() has been properly implemented here, because SPARK() has the full swap operation definition as a required postcondition. Should some human error have caused the steps of Swap() to be taken out of order, for example, SPARK would not verify that postcondition upon subprogram exit. The presence of a possible error both in the postcondition and the implementation would, in turn, cause other proofs to fail down the line. Only the ultimate postconditions in QS() must be satisfactory to the skeptic; SPARK's attempts to verify those will fail if anything else fails to uphold the logical structure.

7.3.5 SwapSpanning.

SwapSpanning() is a lemma that states that if a subset of an Index_Array that contains two fewer elements than the original spans, and that those two other indices contain each other as elements, then the entire array spans. The code for this lemma does not require any statements in the body, because devoid of other context SPARK can easily prove this assertion. Creating this lemma simplifies the Swap proof to showing that the preconditions hold, and invoking this lemma.

7.3.6 Trans_Perm_Update - Specification.

Now that the details of the Swap() subprogram have been fully explicated, it is time to turn back to the Trans_Perm_Update() function used in the loop body to perform the partition step of

Figure 22. Swap() Body

```
procedure Swap (A : in out Element_Array;
                Ghost_Perm_Map : out Permutation_Array;
                I, K : in IndexType
               )
is
   GP_Map : Index_Array := IndexArrayOf(A);
   Temp : Element;

begin
   Temp := A (I);
   A (I) := A (K);
   A (K) := Temp;
   GP_Map(I) := K;
   pragma Assert(for all J in GP_Map'Range =>
                   (if not (J = I) then GP_Map(J) = J));
   pragma Assert(GP_Map(I) = K);
   GP_Map(K) := I;
   pragma Assert(for all J in GP_Map'Range =>
                   (if not (J = I or J = K) then GP_Map(J) = J));
   pragma Assert(GP_Map(K) = I);
   pragma Assert(for all J in GP_Map'Range =>
             (if not (J = I or J = K) then
                  (for some Z in GP_Map'Range =>
                        GP_Map(Z) = J and not (Z = I or Z = K)))
                  );
   SwapSpanning(GP_Map, I, K);
   pragma Assert(Spanning_Predicate(GP_Map));
   Ghost_Perm_Map := GP_Map;
end Swap;
```

Figure 23. SwapSpanning() Lemma

```
procedure SwapSpanning(P: Index_Array;
                       I: IndexType;
                       K: IndexType) with
  Ghost,
  Pre => (
            I in P'Range and then
              K in P'Range and then
           (for all J in P'Range =>
             (if not (J = I or J = K) then
                  (for some Z in P'Range =>
                        P(Z) = J and not (Z = I or Z = K)))
                  )
            and then P(I) = K and then P(K) = I
           ),
  Post => (
            Spanning_Predicate(P)
          );
```

QS(). Trans_Perm_Update() acts as a combination lemma (asserting the transitive property of permutations) and update function to transfer the elements of the Index_Array "Ghost_LPM" to the Permutation_Array "Ghost_Perm_Map." In doing so it proves to SPARK that there is still a permutation map from the updated "A" to Init," and it is necessary to invoke it upon each call to Swap().

Besides ensuring an absence of runtime exceptions from bad array access, the preconditions to Trans_Perm_Update() in Figure 24 are that "A" and "B" are permutations (by the "AToB" permutation map) and "B" and "C" are permutations (by the "BToC" permutation map). Similarly, the postconditions provided in Figure 25 ensure that the indices are properly set up for the output permutation and then state the transitive result, that "A" and "C" are permutations mapped by the output.

Additionally, the statements on equivalent first index and length is used to ensure absence of runtime exceptions when using a single iterator to reference both the resulting permutation map and the input permutation map "BToC," which is necessary in establishing the permutation link between the input to QS(), "Init," the previous partition loop iteration value "A_Previous," and the current value of "A" after the swap operation.

Examining the preconditions and postconditions of Trans_Perm_Update() thus yields the classic definition of transitivity: If there's a map from "A" to "B," and a map from "B" to "C," there's a map from "A" to "C," which is then returned by this function.

Figure 24. SwapSpanning() Lemma

```
1     function Trans_Perm_Update(BToC, AToB : in Permutation_Array;
2                                A, B, C : Element_Array) return Permutation_Array
3     --Returns the permutation array that maps the transitive permutation from
4     -- A to C.
5       with
6       Pre => (Perm_Map_Properties_Hold(AToB) and then
7                 Perm_Map_Properties_Hold(BToC) and then
8                 Perm_Lengths_Match(A, B, AToB) and then
9                 Perm_Lengths_Match(B, C, BToC) and then
10              (for all I in BToC'Range =>
11                   I in AToB'Range) and then
12              (for all I in BToC'Range =>
13                BToC(I) in AToB'Range) and then
14              Is_Permutation(A, B, AToB) and then
15              Is_Permutation(B, C, BToC)),
```

Figure 25. Trans_Perm_Update() Specification-Postconditions

```
17      Post => (Trans_Perm_Update'Result'First = BToC'First and then
18                 Trans_Perm_Update'Result'Length = BToC'Length and then
19                 Perm_Map_Properties_Hold(Trans_Perm_Update'Result) and then
20                 Is_Permutation(A, C, Trans_Perm_Update'Result));
```

7.3.7 Trans_Perm_Update - Body.

Given the preconditions, the only actual work to be done by the update is to map each element of the output (the Index_Array "AToC") via the application of the transitive property. This work will be done in a loop that constructs "AToC" by referencing "AToB" via the elements of "BToC." "AToC" is then returned on function exit.

In Figure 26, “AToC” must be an index array, instead of the expected permutation array, because it violates the permutation array predicates while it is being constructed; after it is constructed, proving it to satisfy the permutation predicates allows it to be assigned to the output. It is easily shown that while it is under construction, each entry of “AToC” satisfies the condition that it maps an element of the “A” array to the “C” array, and the assertions placed here maintain that.

The loop invariant, shown in Figure 27, shows that the growing sub-array maintains almost all the necessary conditions for a permutation from “A” to “C” are observed, saving only the full spanning proof. First, it shows tha “AToC” assigns elements from “A” to “C,” converting the assertions from the loop body (which apply only to the latest element) into a general statement about “AToC.” After confirming inductively that the entire permutation array being built is being constructed from the concatenation of the two provided to the function (lines 14 – 16), the loop invariant establishes uniqueness (lines 18–25) and the permutation property (28 – 30). However, spanning can not be proven for an array as it is being built, since it requires access to all the elements to check against the range. A weaker but necessary precondition, that the elements of the array do fall in the range, is instead proven inductively (lines 32 – 34).

Figure 26. Trans_Perm_Update() Body Part 1

```
1     function Trans_Perm_Update(BToC, AToB : in Permutation_Array;
2                                A, B, C : Element_Array) return Permutation_Array
3     is
4        AToC : Index_Array := BToC;
5     begin
6        for I in BToC'Range loop
7           AToC(I) := AToB(BToC(I));
8               pragma Assert(for all J in AtoC'First..I =>
9                             A(AToB(J)) = B(J) and then
10                            B(BToC(J)) = C(J) and then
11                            A(AToC(J)) = C(J));
```

Figure 27. Trans_Perm_Update() Body Part 2

```
12          pragma Loop_Invariant(
13                                --Assignment proof
14                                (for all J in BToC'First..I =>
15                                   AToC(J) = AToB(BToC(J)))
16                                and then
17                                --Uniqueness proof
18                                (for all J in BToC'First..I =>
19                                       (for all K in BToC'First..I =>
20                                            (
21                                             if not (J = K) then not
22                                               (AToC(J) = AToC(K))
23                                            )
24                                       )
25                                )
26                                and then
27                                  --Perm map property
28                                (for all J in AToC'First..I =>
29                                   A(AToC(J)) = C(J)
30                                     ) and then
31                                  --Spanning Property necessary
32                                  (for all J in AToC'First..I =>
33                                     AToC(J) in AToC'Range
34                                  )
35                               );
36       end loop;
```

Exiting the loop, a lemma is invoked to prove the spanning predicate for "AToC." This is an example of "lemma offloading," where statements that could logically be made at a point in execution are instead invoked in a procedure, to clear the context for SPARK and allow for faster proving.

7.3.8 Spanning_Trans_Lemma Specification.

SPARK will, when proving Trans_Perm_Update(), prove only the preconditions for Spanning_Trans_Lemma() and assume the postcondition to be true. Then, it will independently prove Spanning_Trans_Lemma(), assuming the preconditions at that time and proving the postconditions.

First, the specification (Figure 29) contains some access assurances and context from the invocation point that might be useful to SPARK.

The relevant spanning properties, including the one established in the Loop_Invariant, are given as preconditions next (Figure 30). The postcondition is the one sought for "AtoC"'s spanning predicate, which is the last step necessary to assign it to the output of Trans_Perm_Update().

7.3.9 Spanning_Trans_Lemma Body.

Two assertions are included in the body of Spanning_Trans_Lemma() (Figure 29) as guideposts to SPARK to allow the proof to complete in a timely manner. These assertions are what would be required where Spanning_Trans_Lemma() was invoked, if the context were not overwhelming.

Figure 28. Trans_Perm_Update() Body Part 3

```
37          Spanning_Trans_Lemma(AToB, BToC, AToC);
38          return AToC;
39       end;
```

Figure 29. Spanning_Trans_Lemma() - Specification

```
1     procedure Spanning_Trans_Lemma(AtoB, BtoC : Permutation_Array;
2                                    AtoC : Index_Array)
3       --Spanning property is transitive.
4       with Ghost,
5       Always_Terminates,
6       Pre => (
7                 AtoB'Length = AtoC'Length and then
8                 BtoC'Length = AtoC'Length and then
9                 AtoB'Length = BToC'Length and then
10                Spanning_Predicate(AtoB) and then
11                Spanning_Predicate(BtoC) and then
12                Is_Unique(AtoB) and then
13                Is_Unique(BtoC) and then
14                Is_Unique(AtoC) and then
```

The first assertion (lines 4 – 8) is that every element of "AtoB" is mapped to by some index of "AtoB." The second assertion (lines 9 – 13) is that every element of "AtoB" is mapped to by some index of "AtoC." Together with the inherited context from the preconditions, this lets SPARK prove that "AtoC" satisfies the spanning predicate.

7.3.10 Post-Loop Swap.

Returning now to the QS() partitioning step, it has been established that the elements of the "low" section have values less than the pivot and elements of the "high" section have values greater than or equal to the pivot. The pivot remains the last element of "A," so the only remaining operation to be completed from the partitioning pseudocode (Figure 2, line 8) is to swap the K^{th}

element with the pivot. It has also been established that Ghost_Perm_Map shows that the value of the updated "A" is a permutation of the initial input array "Init."

The last swap places the pivot element in its correct position. SPARK is capable of following the logic involved with "K"'s placement without any guideposting assertions, thanks to the Swap() operation being fully defined in its postcondition (Figure 21). The pivot element is now positioned between the "low" part and the "high" part.

Figure 30. Spanning_Trans_Lemma() - Specification cont.

```
15                (for all I in BtoC'Range =>
16                     BtoC(I) in AtoB'Range) and then
17                (for all I in AtoC'Range =>
18                     I in BtoC'Range) and then
19                (for all I in AtoC'Range =>
20                     AtoC(I) = AtoB(BtoC(I))
21                )
22             ),
23       Post => (Spanning_Predicate(AtoC));
```

Figure 31. Spanning_Trans_Lemma() - Body

```
1     procedure Spanning_Trans_Lemma(AtoB, BtoC : Permutation_Array;
2                                    AtoC : Index_Array) is
3     begin
4        pragma Assert(for all I in AtoB'Range =>
5                         (for some J in AtoB'Range =>
6                            AtoB(J) = I
7                         )
8                      );
9        pragma Assert(for all I in AtoB'Range =>
10                        (for some J in AtoC'Range =>
11                           AtoC(J) = AtoB(I)
12                        )
13                     );
14    end;
```

7.3.11 Leaving the Partition Step.

Exiting the partition step, the properties that must be maintained can be affirmed. First, with regards to ordering, the pivot element has been properly placed between the "low" and "high" parts, and for the "low" part, all elements have a value less than the value of the pivot and for the "high" part all elements have a value greater than or equal to the value of the pivot. Secondly, SPARK has automatically verified the absence of runtime exceptions based on the preconditions on the input. Finally, thanks to the invocations of the Trans_Perm_Update() function, the fact that "A" remains a permutation of "Init" has been maintained for SPARK.

Figure 32. QS() - Post-loop swap

```
51          --Swap K and Pivot
52            A_Previous := A;
53            Swap (A, Ghost_LPM, K, Pivot);
54            Ghost_Perm_Map :=
55              Trans_Perm_Update(Ghost_Perm_Map, Ghost_LPM,
56            A, A_Previous, Init);
```

8 Recursion

8.1 Introduction.

The final step of the Quicksort algorithm is to invoke it recursively on both the "low" part (those elements, now before the pivot, that have value less than the pivot) and the "high" part (those elements, now after the pivot, that have value greater than the pivot). Because QS() will be invoked on sub-arrays of the array "A", it is also at this point that those sub-arrays must satisfy the preconditions for QS(), along with the permutation map that is sent to record the permutation.

8.2 "Low" Recursion.

The first branch of the recursive step (Figure 33) begins with a conditional check on the length of the low part. This is done to simplify range checks on K, ensuring the absence of runtime exceptions.

In Figure 33 "Ghost_LPM" is set to "A"'s identity permutation map to ensure that the indices match properly for the QS() call. "A_Previous" is set to "A" to have a comparison of the entire working array both before and after the call, which will be used to establish the chain of permutations. SPARK always assumes that subprogram postconditions are true at the point of subprogram invocation, so after the call to QS(), it is established that the low slice of the new "A" is a permutation of the low slice of "A_Previous," that it is properly ordered, and that the permutation map is now stored in the passed slice of "Ghost_LPM," since these are all yielded by the QS() postconditions (Figure 12). Utilizing these facts going forward makes it possible to establish those same postconditions on the entirety of "A," but it requires a few steps to show SPARK how to integrate from the slice to the whole.

There are two things that need to happen to integrate these results into the full verification the of QS() postconditions. The first is to assert that the "low" part of the array, combined with the pivot, is properly ordered. Yielding to a moment of anthropomorphization, SPARK knows that "K" (now the pivot index) is greater than anything that was in the "low" part before it was sorted (established in the partition step), it knows that the "low" part is properly sorted now (thanks to the QS() postconditions), and it knows that the new "low" is a permutation of the old "low" (again, from the QS() postconditions). What needs to be shown to SPARK to prove the full ordering of the "low" part with the pivot is that permutations preserve maximums, that is, if a number is greater than or equal to every element of an array, then it is greater than or equal to every element of a permutation of that array.

This is where the Max_Perm_Lemma() comes in.equal to every element of an array, then it is greater than or equal to every element of a permutation of that array. This is where the Max_Perm_Lemma() comes in.

Figure 33. QS() Body - Recursive Step, first branch

```
57            if (K > A'First) then -- Don't need to sort 0 elements
58               A_Previous := A;
59               Ghost_LPM := IndexArrayOf(A);
60               QS(A(A'First..K-1), Ghost_LPM(Ghost_LPM'First..K-1));
61               Max_Perm_Lemma(A(A'First..K-1), A_Previous(A_Previous'First..K-1),
62                              Ghost_LPM(Ghost_LPM'First..K-1), A_Previous(K));
```

Figure 34. QS() Body - Recursive Step, first branch

```
1  procedure Max_Perm_Lemma(Left, Right : Element_Array;
2      Perm_Map : Permutation_Array;
3      K : Element) with
4    --Permutations preserve maximums.
5    Ghost,
6    Always_Terminates,
7    Pre => (
8      Perm_Lengths_Match(Left, Right, Perm_Map) and then
9      Is_Permutation(Left, Right, Perm_Map) and then
10     (for all I in Right'Range =>
11       Right(I) <= K)
12     ),
13     Post => (
14     (for all I in Left'Range =>
15       Left(I) <= K)
16     );
```

8.2.1 Max_Perm_Lemma.

The specification for Max_Perm_Lemma() is shown in Figure 34. First, the preconditions establish that "Left" and "Right" are a permutation (lines 8–9) and that "K" is a maximum of "Right" (lines 10 – 12). Then, the postcondition (lines 13 – 16) establishes that "K" is also a maximum of "Left." Note the "Ghost" property declared in the "with" block (line 5). This aspect will cause SPARK to ensure that this subprogram is not compiled unless a specific compiler flag is set, preventing it from interfering with QS() at runtime while allowing its use in the verification of QS().

Thanks to the definitions established, SPARK is able to prove the postcondition of this lemma directly from the preconditions, obviating the need for a body beyond a "null" statement (not shown).

8.2.2 Sub_Perm_Left_Update.

Returning briefly to the body of QS(), now that QS() itself is being invoked on portions of the initial array, additional logic (Figure 35) is needed to prove to SPARK that the result still has a permutation map mapping the resulting semi-sorted array to the original initial value of the input array.

Figure 35. QS() Body - Recursive Step, First Branch cont.

```
63              Ghost_Perm_Map :=
64                Trans_Perm_Update(Ghost_Perm_Map,
65                                  Sub_Perm_Left_Update(
66                                    Ghost_LPM(Ghost_LPM'First..K-1),
67                                    A, A_Previous),
68                                  A, A_Previous, Init);
```

The permutation map returned by the recurring QS() doesn't contain all the original indices, so this permutation has two parts. The elements that weren't passed to QS() are mapped by the identity permutation map (i.e., they are identical), and the ones that were passed are mapped by QS()'s returned permutation map. Linking the two together involves the use of another update function that also proves to SPARK that the result is a valid permutation map.

This function is split into two parts for ease of use, with the "left" update being invoked where the first recursion returns, and the "right" update being invoked after the second recursion. In Figure 36, after the carefully preventing access errors (lines 13–21) by enforcing that the first and last elements of the arrays are the same and that the ranges are identical, the next precondition of Sub_Perm_Left_Update() (lines 24–28) is that the first part of the array "Left" (as defined by the range

Figure 36. Sub_Perm_Left_Update() Specification

```
1     function Sub_Perm_Left_Update(Perm_Map : Permutation_Array;
2                                   Left, Right : Element_Array)
3                                   return Permutation_Array
4       --If the first part of Left has a permutation map that maps to the
5       --  first part of Right and all elements outside that region are identical
6       --  between the two arrays, those arrays are permutations.
7       with
8         --Ghost,
9         Global => Null,
10      Pre => (
11              --Index Equality, with the caveat that Perm_Map applies only
12              -- to the first part of Left and Right
13                Perm_Map'Length > 0 and then
14                  Right'Length > 0 and then
15                      Left'Length = Right'Length and then
16              (for all I in Perm_Map'Range =>
17                   I in Right'Range) and then
18                Left'First = Perm_Map'First and then
19                  Right'First = Perm_Map'First and then
20                           Right'Last >= Perm_Map'Last and then
21                (for all I in Right'Range => I in Left'Range) and then
22                --The first part of Perm_Map is a permutation map for the
23              -- first part of Left to the first part of Right
24                  Perm_Lengths_Match(Left(Perm_Map'First..Perm_Map'Last),
25                    Right(Perm_Map'First..Perm_Map'Last), Perm_Map) and then
26                  Is_Permutation(Left(Perm_Map'First..Perm_Map'Last),
27                    Right(Perm_Map'First..Perm_Map'Last),
28                Perm_Map) and then
```

Figure 37. Sub_Perm_Left_Update() Specification cont.

```
29
30                --The remainder of Left and Right are equivalent
31                  (for all I in Perm_Map'Last + 1 .. Left'Last =>
32                       Left(I) = Right(I))
33              ),
```

of the input “Perm_Map”) is mapped to the first part of the array “Right,” which is inherited from the postcondition of the QS() recursion, along with the precondition required for that declaration.

In Figure 37, the second part of the “Left” array is identical to the “Right” array (as neither were altered in the QS() recursive call). That relationship is presented in the final precondition.

Finally, in Figure 38, the postcondition states that “Sub_Perm_Left_Update’Result” is, in fact, a permutation map from “Left” to “Right,” which is what SPARK needs to proceed with the QS() proof.

To satisfy this postcondition, the construction of the full permutation array is carried out in two loops. The first (Figure 39, line 7– Figure 41, line 26) copies over the permutation map inherited from the recursive QS() call, and the second (Figure 42, lines 31 – 48) fills out the remainder with the identity map.

The Index_Array “Perm_Map_Update” array is initialized to inherit the necessary length and indices of “Right.” Two assertions are then made to reiterate the important properties for SPARK that will be used in the loop invariant to follow. The first loop, Figure 39 which copies over the inherited permutation map, has its own loop invariant.

The first part of the Loop_Invariant (Figure 40) establishes the uniqueness predicate for “Perm_

Figure 38. Sub_Perm_Left_Update() Specification Postcondition

```
34       Post => (
35                  --The output is a full permutation map from Left to Right
36                  Perm_Map_Properties_Hold(Sub_Perm_Left_Update'Result) and then
37                    Perm_Lengths_Match(Left, Right,
38                      Sub_Perm_Left_Update'Result) and then
39                    Is_Permutation(Left, Right, Sub_Perm_Left_Update'Result)
40                );
```

Figure 39. Sub_Perm_Left_Update() Body

```
1    function Sub_Perm_Left_Update(Perm_Map : Permutation_Array;
2                                  Left, Right : Element_Array)
3                                  return Permutation_Array is
4        Perm_Map_Update : Index_Array(Right'First..Right'Last) :=
5          (others => IndexType'First);
6     begin
7        for I in Perm_Map'First..Perm_Map'Last loop
8           Perm_Map_Update(I) := Perm_Map(I);
```

Figure 40. Sub_Perm_Left_Update() Body - First Loop Invariant

```
9            pragma Loop_Invariant(--Perm_Map_Update's Uniqueness
10                                 (for all J in Right'First..I =>
11                                      Perm_Map_Update(J) = Perm_Map(J))
12                                   and
13                                   (for all J in Right'First..I =>
14                                      (for all K in Right'First..I =>
15                                         (if not (J = K) then not (
16                                          Perm_Map_Update(J) =
17                                               Perm_Map_Update(K))
18                                         )
19                                    )
20                                   ) and
```

Map_Update," because it establishes that the growing array is just a copy of the corresponding elements of "Perm_Map," and thus it simply inherits uniqueness from the already constructed permutation map output by QS().

The fact that Perm_Map_Update() maps "Left" to "Right" is then established (lines 22 – 23, Figure 41). After the loop, the full spanning predicate is checked as a guidepost to SPARK to show that this property was properly verified (lines 27 – 30).

Figure 41. Sub_Perm_Left_Update() Body - First Loop Invariant Part 3

```
21                                   --Permutation condition
22                                   (for all J in Right'First..I =>
23                                      Left(Perm_Map_Update(J)) = Right(J)
24                                   )
25                                 );
26       end loop;
27       pragma Assert(Spanning_Predicate(Perm_Map(Perm_Map'First..
28                     Perm_Map'Last)));
29       pragma Assert(Spanning_Predicate(Perm_Map_Update(Perm_Map_Update'First..
30                     Perm_Map'Last)));
```

The second loop (Figure 42) is more straightforward, as the guiding permutation map is just the identity map, but now that spanning has been established from the first loop updating it becomes slightly more complex. Adding a single element that indexes itself to a spanning map is a specific example of the more general case of concatenating two spanning maps, which has been offloaded to the Spanning_Update_Lemma() (line 33), described below. The final return (line 49), since it invokes the Permutation_Map predicates, when combined with the mapping loop invariants serves as a robust check to ensure that Is_Permutation() is satisfied.

Figure 42. Sub_Perm_Left_Update() Body - Second Loop

```
31       for I in Perm_Map'Last+1..Right'Last loop
32           Perm_Map_Update(I) := I;
33           Spanning_Update_Lemma(Perm_Map_Update(Perm_Map_Update'First..I), I);
34           pragma Loop_Invariant(--Uniqueness
35                                 (for all J in Perm_Map'Last+1..I =>
36                                    Perm_Map_Update(J) = J) and
37                                 --Spanning
38                                 Spanning_Predicate(Perm_Map_Update(
39                                                    Perm_Map_Update'First..I))
40                                 and
41                                 --Permutation Condition
42                                   (for all J in Perm_Map'Last+1..I =>
43                                      J in Left'Range and then
44                                        Perm_Map_Update(J) in Left'Range and then
45                                           Left(Perm_Map_Update(J)) = Right(J)
46                                     )
47                                );
48        end loop;
49        return Perm_Map_Update;
50     end;
```

8.2.3 Spanning Update Lemma.

The Spanning_Update_Lemma() (Figure 43) procedure is used to verify to SPARK that if an array is a concatenation of two sub-arrays that span when constricted to their respective ranges, the

Figure 43. Spanning_Update_Lemma()

```
1    procedure Spanning_Update_Lemma(Perm_Map : Index_Array;
2                                    PartPoint : IndexType)
3        --If two partitions of a full map span their indepedent ranges,
4        --  then the full map spans.  Partpoint denotes the separation of the
5        --  partitions (as the first element of the second partition)
6        with Ghost,
7        Always_Terminates,
8        Pre => (Perm_Map'First <= PartPoint and then
9                  Perm_Map'Last >= PartPoint and then
10                 Spanning_Predicate(Perm_Map(Perm_Map'First..PartPoint-1))
11                 and then
12                 Spanning_Predicate(Perm_Map(PartPoint..Perm_Map'Last))
13               ),
14       Post => (Spanning_Predicate(Perm_Map));
15
16
17     procedure Spanning_Update_Lemma(Perm_Map : Index_Array;
18                                     PartPoint : IndexType) is
19     begin
20        null;
21     end;
```

full array spans. The full array is passed with a partition point ("PartPoint"), which is verified to be a legitimate index (lines 8 – 9) and which marks the beginning of the second sub-array, and the spanning property is asserted on each subsection (lines 11 – 12). The postcondition, that the entire initial array spans, is verifiable by SPARK without the need for any statements in the body.

8.2.4 Exiting the "Low" Recursion Step.

Returning to QS() in Figure 44, there are three guideposting assertions at the end of the first branch of the QS() recursive step that SPARK needs going forward to aid it in proving the ordering and permutation postconditions.

Figure 44. Spanning_Update_Lemma()

```
69              --Ordering of first K elements has now been established between the
70              --max and QS postcondition.
71              pragma Assert(for all I in A'First..K-1 => A(I) <= A(I+1));
72              pragma Assert(Perm_Map_Properties_Hold(Ghost_Perm_Map));
73              pragma Assert(Is_Permutation(A, Init, Ghost_Perm_Map));
74           end if;
```

The Max_Perm_Lemma() shows SPARK that the K^{th} element is greater than the first K - 1 elements, where K is the pivot element's index, and the QS() postcondition shows that ordering has been maintained for the first K - 1 elements. SPARK then requires this final ordering assertion (Figure 44) to tie those two elements together, and then two signposting assertions are given to SPARK to affirm that the current state of the output array is still a permutation of the input.

8.3 "High" Recursion.

With the "low" part of the post-partition array having been fully sorted recursively, all that remains to be done is to do the same thing for the "high" part.

The second recursive branch (Figure 45) is a mirror of the first, although notably it does not

Figure 45. QS() Body - Recursive Step Second Branch

```
75           if (K < A'Last) then  -- Don't need to sort 0 elements.
76              A_Previous := A;
77              Ghost_LPM := IndexArrayOf(A);
78              QS(A(K+1..A'Last), Ghost_LPM(K+1..Ghost_LPM'Last));
79              Min_Perm_Lemma(A(K+1..A'Last), A_Previous(K+1..A_Previous'Last),
80                             Ghost_LPM(K+1..Ghost_LPM'Last), A_Previous(K));
81
82              Ghost_Perm_Map :=
83                Trans_Perm_Update(Ghost_Perm_Map,
84                                  Sub_Perm_Right_Update(
85                                    Ghost_LPM(K+1..Ghost_LPM'Last),
86                                 A, A_Previous),
87                                  A, A_Previous, Init);
88
89              -- Ordering of elements from K to the end has now been established
90              -- from the previous min and QS postcondition.
91              pragma Assert(Perm_Map_Properties_Hold(Ghost_Perm_Map));
92              pragma Assert(Is_Permutation(A, Init, Ghost_Perm_Map));
93           end if;
94           pragma Assert(Is_Permutation(A, Init, Ghost_Perm_Map));
95        end if;
96        pragma Assert(Is_Permutation(A, Init, Ghost_Perm_Map));
97     end QS;
```

require the final guideposting assertions. It may seem odd that the returning "Ghost_LPM" will work as a permutation map, since the sub-array of "A" sent to QS() does not start with the first index of "A". Understanding why this works requires understanding a key property of the language: Indices in Ada are not changed across subprogram calls, and SPARK inherits this property. Thus, the input array for this recursive QS() call will maintain the indices that are sent to it with "A", allowing "Ghost_LPM" to function properly as a permutation map with identically offset indices (especially important when you consider how the Sub_Perm_Right_Update() will construct its own permutation map).

The only remaining differences from the "Low" part are in the lemma and the update invoked. Concluding the code are a series of signposting assertions to affirm to SPARK that the permutation postcondition is met through all branches.

8.3.1 Min_Perm_Lemma.

After the recursive QS() call on the "high" part (Figure 45, line 78), Min_Perm_Lemma() is used to relate that the reordered "high" part still contains elements that are of greater than or equal value to the pivot value. The construction of Min_Perm_Lemma() (Figure 46) simply involves reversing the inequalities in the precondition and the postcondition from those of Max_Perm_Lemma(). To summarize, the preconditions ensure access to arrays can happen without raising an exception, the "high" part after the recursive call is a permutation of the "high" part before the call, and all elements of the "high" part before the call are greater than or equal to the value of the pivot. The postcondition affirms that the reordering of the "high" part after the call maintains that relationship with the pivot.

Figure 46. Min_Perm_Lemma()

```
1     procedure Min_Perm_Lemma(Left, Right : Element_Array;
2                              Perm_Map : Permutation_Array;
3                           K : Element) with
4       --Permutations preserve minimums.
5       Ghost,
6       Always_Terminates,
7       Pre => (
8                 Perm_Lengths_Match(Left, Right, Perm_Map) and then
9                 Is_Permutation(Left, Right, Perm_Map) and then
10                  (for all I in Right'Range =>
11                       Right(I) >= K)
12            ),
13      Post => (
14                 (for all I in Left'Range =>
15                      Left(I) >= K)
16             );
```

8.3.2 Sub_Perm_Right_Update.

After establishing the ordering of the "high" part after the recursive call, Sub_Perm_Right_Update is invoked (Figure 45, line 88) to establish that the array on the return is a permutation of the array that was sent to QS(), in the same way that Sub_Perm_Left_Update() was used in the first recursion. The specification for the Sub_Perm_Right_Update() function is presented in Figure 47 to clarify the differences between it and the Sub_Perm_Left_Update(). As the only difference in implementation is that the body reverses the order of the two loops, the body has been omitted for brevity.

8.4 Exiting.

Finally, everything is tied together. The last thing to do (Figure 48) is escape the conditional branch that guarantees the base case arrays will not be sorted.

When SPARK is run on this subprogram and all of the assertions pass, the absence of runtime exceptions property is verified because the entire QS() implementation and all of the subprograms it invokes exist in SPARK_Mode. The termination property is checked and passes because each recursive call to QS() omits at least one value (the pivot element) from the input array, which satisfies the Subprogram_Variant condition. The permutation postcondition is satisfied by the Sub_Perm_Lemma() invocations, combined with the permutation postcondition of QS() itself, and the ordering postcondition is satisfied by itself combined with the relationship of the pivot to the rest of the partition from the partition step; the latter is shown via the Max_Perm_Lemma() and Min_Perm_Lemma() subprograms.

With all properties of Quicksort verified to hold for QS(), SPARK will verify that the relationship between the QS() preconditions and postconditions holds: QS() sorts.

Figure 47. Sub_Perm_Lemma_Right() Specification

```
function Sub_Perm_Right_Update(Perm_Map : Permutation_Array;
                               Left, Right : Element_Array)
      return Permutation_Array
      --If the second part of Left has a permutation map that maps to the
      --  second part of Right and all elements outside that region are
      --  identical between the two arrays, those arrays are permutations.
      with
        --Ghost,
        Global => Null,
      Pre => (
              Perm_Map'Length > 0 and then
                Right'Length > 0 and then
                   Left'Length = Right'Length and then
              (for all I in Perm_Map'Range =>
                   I in Right'Range) and then
                Left'Last = Perm_Map'Last and then
                  Right'Last = Perm_Map'Last and then
                      Left'First <= Perm_Map'First and then
                          Right'First <= Perm_Map'First and then
                (for all I in Right'Range => I in Left'Range) and then
                --The first part of Perm_Map is a permutation map for the
              -- first part of Left to the first part of Right
                   Perm_Lengths_Match(Left(Perm_Map'First..Perm_Map'Last),
                     Right(Perm_Map'First..Perm_Map'Last), Perm_Map) and then
                   Is_Permutation(Left(Perm_Map'First..Perm_Map'Last),
                     Right(Perm_Map'First..Perm_Map'Last),
                Perm_Map) and then
                --The remainder of Left and Right are equivalent
                  (for all I in Left'First .. Perm_Map'First - 1 =>
                       Left(I) = Right(I))
               ),


      Post => (Perm_Map_Properties_Hold(Sub_Perm_Right_Update'Result) and then
                   Perm_Lengths_Match(Left, Right,
                                      Sub_Perm_Right_Update'Result) and then
                   Is_Permutation(Left, Right, Sub_Perm_Right_Update'Result)
              );
```

Figure 48. QS Body() - Final Lines

```
84          end if; -- Array'Length < 2
85       end QS;
```

9 Conclusions

The project to create a fully-verified implementation of Quicksort was a fascinating one when it came to learning the processes employed by an auto-active verifier. However, this implementation proof relies on the updating and passing of a permutation map that will essentially double the work involved in the sorting algorithm, as well as quite a bit of overhead in maintaining it. This is because the permutation maps can't be passed as arguments to non-Ghost subprograms and declared to be Ghost variables at the same time. The good news is that Adacore is hard at work fixing this problem, and its slated to be a feature in 2026. The bad news is that, at the time of writing this paper, it is currently 2025. Nevertheless, as an exercise in semi-automatic verification, this experiment is a success, developing an independent definition of permutations and utilizing them in a platinum-level proof of QS() as an implementation of Quicksort.

While this experience presented the code guided by the program flow, this is not the way the project developed. When new languages are being learned, it is frequently done by iteration, where attempts are made and refined in reaction to the syntax checker, the compiler, and test problems showing errors in logic and implementation.

Learning SPARK was similarly iterative, except that it was a higher-level iteration, with SPARK acting as a logic checker, revealing to me areas where my implementation fell short and guiding me about what it needed to proceed logically. Just as a syntax checker is an invaluable tool to novice and expert coders alike in checking syntax, a verifier like SPARK is invaluable in checking logic, an area that is not nearly so tractable to developers (as the frequency of bugs even in mature code demonstrates).

Some lessons were not intuitive. It is, for example, possible to combine the two subarray permutation lemmas (Sub_Perm_Lemma_Left() and Sub_Perm_Lemma_Right()) into one more general Sub_Perm_Lemma(). While that results in fewer lines of code overall, the resulting conditional preconditions and postconditions are much more complex. Of the four resulting possible verifiable statements (with two preconditions and two postconditions),

SPARK is forced to attempt proof across three false statements(with two wrong preconditions and one correct precondition attached to the incorrect postcondition) disjunctively combined with the the single correct statement, resulting in a much larger search space for the automatic provers to explore. Splitting the general case into two simpler and more straightforward lemmas results in an easier program both for humans to read and for SPARK to prove. Coding for proof can lead to concessions being made when viewed through the lens of coding for succinctness.

The brilliance of SPARK, as mentioned in the discussion of the Sort() implementation, is that all the skeptic needs to believe is that, for a section fully implemented in SPARK_Mode, the preconditions and postconditions describe what they need to to ensure the relationship between inputs and outputs. This is reminiscent of, and allows the verification to work hand-in-hand with, traditional "black box" implementation strategies. The strength of this property is hard to overstate: The only true weakness of a "black box" implementation is in the debugging, where one must navigate the layers of black boxes to discover which contains a bug of either syntax or logic.

SPARK does that for you, meaning that finally the programmer can truly focus only on the section that requires development and trust the black boxes to do their job, so long as they prove.

References

[1] Roderick Chapman et al. Co-Developing Programs and Their Proof of Correctness. url: https://www.adacore.com/papers/co-developingprograms-and-their-proof-of-correctness.

[2] Thomas H. Cormen et al. Introduction to Algorithms, Third Edition. 3rd. The MIT Press, 2009. isbn: 0262033844.

[3] SPARK User's Guide. url: https://docs.adacore.com/spark2014-docs/html/ug/en/source/manual_proof.html.

AdaCore | Build Software that Matters

adacore.com