Lectures Notes on
Searching and Sorting

Software Verification

Nova School of Science and Technology
Mário Pereira   mjp.pereira@fct.unl.pt

Version of October 29, 2025

These notes are based on lecture notes by Bernardo Toninho.

1 Introduction

In these notes, we analyze a series of examples and present a strategy for specifying and verifying iterative algorithms, and in particular, we will address some sorting algorithms using Why3.

2 Loop invariants using arrays

Let us start with an example of a simple algorithm to determine the maximum value stored in an array. Consider the following listing defining a function, which specifies that a given value is greater than all the values in a given array, up to a given position.

```
predicate max_array_spec (a: array int) (n m: int)
= forall k. 0 <= k < n -> a[k] <= m

let max_array_for (a: array int) : int
  requires { 0 < a.length }
  ensures  { max_array_spec a a.length result }
= let ref m = a[0] in
  for i = 1 to a.length - 1 do
    invariant { 1 <= i <= a.length }
    invariant { forall k. 0 <= k < i -> a[k] <= m }
     invariant { max_array_spec a i m }
     if m < a[i] then
      m <- a[i]
  done;
  m
```

Exercise: The post-condition above is not the strongest possible. Do you understand that you can be more specific in the specification of this function?

Observe the expression in predicate `max_array_spec`, here stripped of any extra formatting. It universally quantifies variable `k` with values of type `int`.

```
  forall k. 0 <= k < n -> a[k] <= m
```

The post-condition of the function is written to capture the full extent of the array (`max_array_spec(a,a.Length,m)`), up to position `a.Length-1`. On the other hand, the loop invariant used in the function (`maxArray(a,i,m)`) is a general formulation of the post-condition, which captures that the value of `m` is greater than all the values in the array up to position `i-1`.

Notice that the invariant holds at the start of the loop and should be proven valid in the end. This implication is visible in the following Hoare triple

$$\{maxArray(\mathsf{a}, \mathsf{i}, \mathsf{m}) \wedge \mathsf{i} < \mathsf{a.Length}\}$$

$$\texttt{if } \mathsf{m} < \mathsf{a}[\mathsf{i}] \texttt{ then } \{\mathsf{m} := \mathsf{a}[\mathsf{i}]\} \texttt{ else skip;}$$
$$\mathsf{i} := \mathsf{i} + 1$$

$$\{maxArray(\mathsf{a}, \mathsf{i}, \mathsf{m})\}$$

The backward assignment rule produces the weakest pre-condition for $\mathsf{i} := \mathsf{i}+1$, which is $maxArray(\mathsf{a}, \mathsf{i}+1, \mathsf{m})$, and is implied by $maxArray(\mathsf{a}, \mathsf{i}, \mathsf{m}) \wedge \mathsf{a}[\mathsf{i}] \leq \mathsf{m}$. Such assertion, when used as the post-condition of the conditional statement, is supported in both branches by the pre-condition $maxArray(\mathsf{a}, \mathsf{i}, \mathsf{m})$. Notice that at the end of the if branch we have that $\mathsf{m} = \mathsf{a}[\mathsf{i}]$. The else branch is statement `skip`, and therefore the pre-condition and the negated condition holds at the end ($\mathsf{a}[\mathsf{i}] \leq \mathsf{m}$).

3 Searching

Another example we can explore is a linear search in an array. The specification of any `index_of` function is to return the index of the sought element or to return a negative value if the element does not exist in the array. The precondition serves only the purpose of restricting the inputs so that the search is maintained within legal bounds. The complete signature is the following

```
let index_of (a: array int) (n x: int) : int
  requires { 0 <= n <= a.length }
  ensures  { result < 0 -> forall k. 0 <= k < n -> a[k] <> x }
  ensures  { result >= 0 -> exists k. 0 <= k < n /\ a[k] = x }
```

The post-condition is comprised of two complementary cases, both expressed with an implication, using an universal quantifier to express that all elements are different from the one being sought, and an existential quantifier to express that one element matched the search criteria. Remember that first-order logic is undecidable and that the SMTs used by the Why3 may fail to find the proof for some correct assertions, especially when using quantifiers.

Examine now the implementation of the function, which is based on the invariant that states that *all elements "to the left" of the cursor are different from parameter* ***x***.

```
1 let index_of (a: array int) (n x: int) : int
2   requires { 0 <= n <= a.length }
3   ensures  { result < 0 -> forall k. 0 <= k < n -> a[k] <> x }
4   ensures  { result >= 0 -> exists k. 0 <= k < n /\ a[k] = x }
5 = for i = 0 to n - 1 do
6     invariant { forall j. 0 <= j < i -> a[j] <> x }
7     if a[i] = x then return i
8   done;
9   -1
```

This function follows the common programming pattern of exiting the loop and the function when encountering the first occurence of an element in an array. Notice the loop invariant in line 11, which is a general form of the post-condition on line 3, assuming that the element was not yet found. It is not obtained by a direct replacing of the limit variable by the cursor, but it is still a sub-formula and more general formulation. Notice also that it is valid at the beginning of the loop (the assert on line 7). Such invariant will be valid at the end of the loop, and when combined with the invariant in line 10, and the negated condition (`i` $\geq$`n`), supports the post-condition linked to the result (`-1`). The post-condition in line 4 is supported by the condition of the if statement and the loop invariant and loop condition that says that `0` $\leq$`i < n`. Thus, the existential quantifier is satisfied by given a witness to its formula ($\mathsf{k} = \mathsf{i}$).

This form of specification is very common, linking a special result value to a given condition. The need to frame the value of the cursor variables is also common when using integers as cursors.

Exercise: Note that the specification can be stronger and that the use of an existential quantifier is less informative than an alternative formulation. Can you find the alternative?

4 Binary Search

A more efficient searching algorithm such as a binary search in a sorted array has the following signature:

```
1  let bin_search (a: array int) (value: int) : (pos: int)
2    requires { 0 <= a.length /\ sorted a }
3    ensures  { 0 <= pos -> pos < a.length /\ a[pos] = value }
4    ensures  { pos < 0 -> forall i. 0 <= i < a.length -> a[i] <> value }
```

Notice that the application of a binary search algorithm assumes that the array is sorted. Notice that no runtime errors occur if the array is not sorted. Nevertheless, the post-condition of finding the element in the array is not at all guaranteed. The assumption of a sorted array is given by the pre-condition in line 3, using the `sorted` predicate with the following definition

```
predicate sorted (a: array int)
= forall i j. 0 <= i <= j < a.length -> a[i] <= a[j]
```

The pre-condition in line 3 above will support the invariant of a loop in a binary search using two cursors, that *the element being sought is always between the low cursor and the high cursor.*

Observe the implementation below, that starts with two limits and uses a middle value to obtain the next low or high cursor.

```
1  let bin_search (a: array int) (value: int) : (pos: int)
2    requires { 0 <= a.length /\ sorted a }
3    ensures  { 0 <= pos -> pos < a.length /\ a[pos] = value }
4    ensures  { pos < 0 -> forall i. 0 <= i < a.length -> a[i] <> value }
5  = let ref low = 0 in
6    let ref high = a.length in
7    while low < high do
8      variant   { high - low }
9      invariant { 0 <= low <= high <= a.length }
10     invariant { forall k. 0 <= k < low -> a[k] <> value }
11     invariant { forall k. high <= k < a.length -> a[k] <> value }
12     let ref mid = div (low + high) 2 in
13     if a[mid] < value then low <- mid + 1
14     else if value < a[mid] then high <- mid
15     else (* value = a[mid] *) return mid
16   done;
17   -1
```

Notice the invariant, stating that all the values below the low limit and higher than the high limit are all different from the element we are looking for. Notice also that the invariant `0` $\leq$`low` $\leq$`high` $\leq$`n` is kept by the computation of the `mid` value.

Notice that the array is not modified inside the loop, and therefore the property `sorted a` is still valid, which is essential to maintain the invariant.

4.1 Boolean results

A similar pattern can be found in boolean functions whose result is the truth value for a given assertion. In that case, an equivalence can be established. In the following definition, we use function `bin_search`, defined above, to implement the function `contains` below

```
let contains (a: array int) (x: int) : bool
  requires { sorted a }
  ensures  { result <-> exists k. 0 <= k < a.length /\ a[k] = x }
= let i = bin_search a x in
  i >= 0
```

Notice the use of logical equivalence in the post-condition and that some information was lost between the results of `bin_search` and `contains`. The actual position of the element is not passed to the results of this function. This is captured anyway by the existential quantifier in the post-condition.

5 Sorting

Let's now take a look at a sorting algorithm. The signature of such a function can be expressed using the `sorted` predicate defined earlier. In this case, the `sorted` predicate is used in a post-condition.

```
let sort (a: array int) : array int
  ensures { sorted result }
```

A simple algorithm to decompose and analyse is a quadratic selection sort algorithm, that starts in the beginning of an array, and iteratively selecting the smallest element of the remaining array.

```
let selection_sort (a: array int)
  ensures { sorted a }
= let min = ref 0 in
  for i = 0 to length a - 1 do
    (* a[0..i[ is sorted; now find minimum of a[i..n[ *)
    invariant { sorted_sub a 0 i /\
        forall k1 k2: int. 0 <= k1 < i <= k2 < length a -> a[k1] <= a[k2] }
    min := i;
    for j = i + 1 to length a - 1 do
      invariant {
        i <= !min < j && forall k: int. i <= k < j -> a[!min] <= a[k] }
      if a[j] < a[!min] then min := j
    done;
    if !min <> i then swap a !min i;
  done
```

Notice that the outer loop assumes that the array is sorted, up to a given point (cursor `i`), expressed via `sorted_sub a 0 i`. The outer invariant also states that all the values left of `i` are smaller than all the elements to the right of the cursor, expressed in the

```
  forall k1 k2: int. 0 <= k1 < i <= k2 < length a -> a[k1] <= a[k2]
```

condition.
Notice that the inner loop performs one single step in the sorting of the whole array. This loop is only concerned about finding the minimum element among the values of the array that remain to be sorted.

This post-condition asserts that the whole array is sorted, as expressed in `sorted a`.

The presented specification is not strong enough for a sorting algorithm. If one completely ignores the input array and just return a fresh one containing all values equal to a certain constant, than array would also be sorted. The missing piece is to also state that the *sequence* of elements in the final array are a permutation of the *sequence* of elements in the initial array. The complete specification is as follows:

```
let selection_sort (a: array int)
  ensures { sorted a /\ permut_all (old a) a } =
  let min = ref 0 in
  for i = 0 to length a - 1 do
    (* a[0..i[ is sorted; now find minimum of a[i..n[ *)
    invariant { sorted_sub a 0 i /\ permut_all (old a) a /\
        forall k1 k2: int. 0 <= k1 < i <= k2 < length a -> a[k1] <= a[k2] }
    min := i;
    for j = i + 1 to length a - 1 do
      invariant {
        i <= !min < j && forall k: int. i <= k < j -> a[!min] <= a[k] }
      if a[j] < a[!min] then min := j
    done;
    label L in
    if !min <> i then swap a !min i;
  done
```

The `permut_all (old a) a` post-condition asserts that every element in `a`, the final array, should also occur with the same number of times as in the initial array. This is a very simple, practical definition of what it means *two arrays to be a permutation of one another*. Finally, the loop invariant also states that at every single iteration one preserves the permutation relation between the current state of the array and the initial one.