Lectures Notes on Searching and Sorting

Construction and Verification of Software
FCT-NOVA
Bernardo Toninho

29 March, 2022

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 Dafny.

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 maxArray(a:array<int>, n:int, m:int)
  requires 0 < n ≤ a.Length
  reads a
{ ∀ k : int • 0 ≤ k < n ⟹ a[k] ≤ m }

method Max(a:array<int>) returns (m:int)
    requires 0 < a.Length
    ensures maxArray(a,a.Length,m)
{
    m := a[0];
    var i := 1;
    while i < a.Length
        decreases a.Length - i
        invariant 1 ≤ i ≤ a.Length
        invariant maxArray(a,i,m)
    {
        if m < a[i]
        { m := a[i]; }
        i := i + 1;
    }
}
```

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

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

```
forall k : int :: 0 <= k < n ==> a[k] <= m
```

Notice also that there is a clause in the predicate specification, **`reads`** `a`, to indicate to the caller context that the predicate will depend on the mutable positions of the array. Which means that all its post-conditions will have to be revoked whenever the array changes. This is used in a derived rule called, the framing rule,

$$\frac{\{A\}\ P\ \{B\}}{\{A \wedge C\}\ P\ \{B \wedge C\}} \quad \text{P does not change } vars(C)$$

Notice that this rule is derived in *Hoare Logic* although not explicitly included. The **`reads`** `a` clause helps to determine, that `maxArray` depends on the values of `a` and cannot be "framed out", or considered in formula C above, if P changes the array.

The post-condition of the method is written to capture the full extent of the array (`maxArray(a,a.Length,m)`), up to position `a.Length-1`. On the other hand, the loop invariant used in the method (`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

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

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

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

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

3 Searching

Another example we can explore is a linear search in an array. See the appendix for the concrete syntax of dafny. The specification of any `indexOf` method is to return the index of the sought element or to return `-1` 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

```
method indexOf(a:array<int>, n:int, x:int) returns (z:int)
  requires 0 ≤ n ≤ a.Length
  ensures z = -1 ⟹ ∀ k : int • 0 ≤ k < n ⟹ a[k] ≠ x
  ensures z ≠ -1 ⟹ ∃ k : int • 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 SMT used by the Dafny compiler (Z3) may fail to find the proof for some correct assertions, especially when using quantifiers.

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

```
1  method indexOf(a:array<int>, n:int, x:int) returns (z:int)
2    requires 0 ≤ n ≤ a.Length
3    ensures z = -1 ⟹ ∀ k : int • 0 ≤ k < n ⟹ a[k] ≠ x
4    ensures z ≠ -1 ⟹ ∃ k : int • 0 ≤ k < n ∧ a[k] = x
5  {
6    var i := 0;
7    assert ∀ k : int • 0 ≤ k < i ⟹ a[k] ≠ x;
8    while i < n
9      decreases n - i
10     invariant 0 ≤ i ≤ n
11     invariant ∀ k : int • 0 ≤ k < i ⟹ a[k] ≠ x
12   {
13     if a[i] = x {
14       return i;
15     }
16     i := i + 1;
17   }
18   return -1;
19 }
```

This method follows the common programming pattern of exiting the loop and the method 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 ≥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 ≤i < n`. Thus, the existential quantifier is satisfied by given a witness to its formula ($k = i$).

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  method BSearch(a:array<char>, n:int, x:char) returns (z:int)
```

```
2   requires 0 ≤ n ≤ a.Length
3   requires sorted(a, n)
4   ensures z = -1 ⟹ ∀ i : int • (0 ≤ i < n) ⟹ a[i] ≠ x
5   ensures z ≠ -1 ⟹ 0 ≤ z < n ∧ a[z] = x
```

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<char>, n:int)
  requires 0 ≤ n ≤ a.Length
  reads a
  { ∀ i, j • (0 ≤ i < j < n) ⟹ 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  method BSearch(a:array<char>, n:int, x:char) returns (z:int)
2    requires 0 ≤ n ≤ a.Length ∧ sorted(a, n)
3    ensures z = -1 ⟹ ∀ i • (0 ≤ i < n) ⟹ a[i] ≠ x
4    ensures z ≠ -1 ⟹ 0 ≤ z < n ∧ a[z] = x
5  {
6      var low, high := 0, n;
7      while low < high
8          decreases high - low
9          invariant 0 ≤ low ≤ high ≤ n
10         invariant ∀ i • 0 ≤ i < n ∧ i < low ⟹ a[i] ≠ x
11         invariant ∀ i • 0 ≤ i < n ∧ high ≤ i ⟹ a[i] ≠ x
12     {
13         var mid := low + (high-low)/2;
14         if a[mid] < x          { low := mid + 1; }
15         else if x < a[mid]     { high := mid; }
16         else  { return mid; }
17     }
18     return -1;
19 }
```

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 ≤low ≤high ≤n` is kept by the computation of the `mid` value.

One more detail, note the parallel assignment in line 6, which declares and initializes two variables in one step.

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

4.1 Boolean results

A similar pattern can be found in boolean methods whose result is the truth value for a given assertion. In that case, an equivalence can be established. In the following definition, we use

method `BSearch`, defined above, to implement the method `Contains` below

```
method Contains(a:array<char>, n:int, x:char) returns (b:bool)
  requires 0 ≤ n ≤ a.Length ∧ sorted(a, n)
  ensures b ⟺ ∃ k • (0 ≤ k < n) ∧ a[k] = x
{
    var i := BSearch(a,n,x);
    return i ≠ -1;
}
```

Notice the use of logical equivalence in the post-condition and that some information was lost between the results of `BSearch` and `Contains`. The actual position of the element is not passed to the results of this method. 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 method can be expressed using the `sorted` predicate defined earlier. In this case, the `sorted` predicate is used in a post-condition.

```
method sort(a:array<char>)
  ensures sorted(a,a.Length)
  modifies a
```

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. The sub-problem of selection sort is represented by an auxiliary method `selectSmaller`, with the following signature,

```
method selectSmaller(a:array<int>,i:int)
    requires 0 ≤ i < a.Length
    requires sorted(a,i)
    requires partitioned(a,i,a.Length)
    modifies a
    ensures sorted(a,i+1)
    ensures partitioned(a,i+1,a.Length)
```

Notice that the method assumes that the array is sorted, up to a given point (cursor `i`), and that all the values left of `i` are smaller than all the elements to the right of the cursor, given by predicate `partitioned`, defined below.

```
predicate partitioned(a:array<char>,i:int,n:int)
  requires 0 ≤ n ≤ a.Length
  reads a;
{ ∀ k, l • 0 ≤ k < i ≤ l < n ⟹ (a[k] ≤ a[l]) }
```

Notice that `selectSmaller` performs one single step in the sorting of the whole array. The implementation of this method that completes the selection for all positions of the array, is the following

```
method selectSmaller(a:array<int>,i:int)
    requires 0 ≤ i < a.Length
```

```
    requires sorted(a,i)
    requires partitioned(a,i,a.Length)
    modifies a
    ensures sorted(a,i+1)
    ensures partitioned(a,i+1,a.Length)
{
    var jMin := i;
    var j := i+1;
    while (j < a.Length)
        invariant i+1 ≤ j ≤ a.Length
        invariant i ≤ jMin < j
        invariant ∀ k • i ≤ k < j ⟹ a[jMin] ≤ a[k]
        invariant sorted(a, i)
        invariant partitioned(a, i, a.Length)
    {
        if (a[j] < a[jMin]) {
            jMin := j;
        }
        j := j+1;
    }
    if (jMin ≠ i) {
        a[i] , a[jMin] := a[jMin] , a[i];
    }
}
```

The method calculates the index `jMin` of the least element of the array, starting from `i`, and then swaps the element at `jMin` with that of position `i`. This means that all the elements up to position `i` are now sorted, since the element at position `i` is now provably greater or equal to all others in greater indices. Also, that all the remaining elements, in higher positions, are larger than all the elements on lower positions.

```
method selectionSort(a:array<char>)
  requires 0 ≤ n ≤ a.Length
  ensures sorted(a,a.Length)
  modifies a
{
    var i := 0;
    while i < a.Length
        invariant 0 ≤ i ≤ a.Length
        invariant sorted(a,i)
        invariant partitioned(a,i,a.Length)
    {
        selectSmaller(a,i);
        i := i+1;
    }
}
```

This final step provides the result (`sorted(a,a.Length)`) for the whole array.

The key in verifying these algorithms is to explore the different (intuitive) invariants that must be kept in all iterations.

6 Inserting into a sorted array

Verifying the sorted insertion of an element in a sorted array is also an interesting challenge. The minimal specification of such a method is given by the following signature

```
method insert(a:array<int>, n:int, e:int)
  requires 0 ≤ n < a.Length
  requires sorted(a, n)
  ensures sorted(a, n+1)
```

However, this is not the strongest post-condition possible. Why not?
Note that any method that maintains the order of the elements and adds an extra one satisfies the specification given.

```
method wrongInsert(a:array<int>, n:int, e:int)
  requires 0 ≤ n < a.Length
  requires sorted(a, n)
  ensures sorted(a, n+1)
  modifies a
{
  if( n > 0 )
  { a[n] := a[n-1]; }
}
```

If one wants to specify that the all the elements in the array were maintained, then a more specific post-condition is necessary. The following specification and implementation provide more detail on how the elements are arranged in the final state of the array. Notice that there is the need to also return the position into which the element was inserted in order to be able to talk about it in the post-condition.

```
1  method insert(a:array<int>, n:int, e:int) returns (pos:int)
2    requires 0 ≤ n < a.Length
3    requires sorted(a, n)
4    ensures sorted(a, n+1)
5    ensures 0 ≤ pos ≤ n ∧ a[pos] = e
6    ensures ∀ k • 0 ≤ k < pos ⟹ a[k] = old(a[k])
7    ensures ∀ k • pos < k ≤ n ⟹ a[k] = old(a[k-1])
8    modifies a
9  {
10   var i := n;
11   if( n > 0 )
12   { a[n] := a[n-1]; }
13   while 0 < i ∧ e < a[i-1]
14     invariant 0 ≤ i ≤ n
15     invariant sorted(a, n+1)
16     invariant ∀ k • i < k < n+1 ⟹ e ≤ a[k]
17     invariant ∀ k • 0 ≤ k < i ⟹ a[k] = old(a[k])
18     invariant ∀ k • i < k ≤ n ⟹ a[k] = old(a[k-1])
19   {
20     a[i] := a[i-1];
21     i := i - 1;
22   }
```

```
23    a[i] := e;
24    return i;
25  }
```

The post-condition in line 5 specifies that the element was inserted in position `pos`, and that post-conditions in lines 6 and 7 specify that the elements to the "left" of the element are the same as before and those to the "right" of the new element are shifted in one position, respectively.

This implementation covers for the case where the array had no elements, and initialize a new position of the array with a copy of the value in the last position.

This initialization is admitely only necessary for verification purposes. This assignment is repeated by the first iteration of the loop. Thus, the invariant of the loop is established before the loop starts. The technique applied here is similar to all previous examples, taking the post-conditions as the starting to design the loop invariant. The extra condition to support the fact that the assignment in line 24 maintains the order in the array is the loop invariant in line 17, that lets the prover know that all elements to the "right" are greater or equal to the new element.

7 Exercises

1. Specify and implement method `fillK(a,n,k,c)`. This method returns **`true`** if and only if the first `c` elements, up to `n`, of array `a` are equal to `k`.

   Define the weakest pre-condition and the strongest post-condition possible. Implement the method so that it verifies.

   ```
   method fillK(a: array<int>, n: int, k: int, c: int)
     returns (b: bool)
   ```

2. Specify and implement the method `containsSubString`. This method tests whether or not the array of characters `a` contains the elements of array `b`. If `a` contains `b`, then the method returns the offset of `b` in `a`. If `a` does not contain `b` then the method returns an illegal index (e.g. `-1`).

   Define the weakest pre-condition and the strongest post-condition possible. Implement the method so that it verifies.

   Hint: you may want to define auxiliary functions and methods. */

   ```
   method containsSubString(a: array<char>, b: array<char>)
     returns (pos: int)
   ```

3. Specify and implement the method `resize`. This method returns a new array whose length is double of the length of the array given as argument (`a`). If the length of the array supplied as an argument is zero, then set the length of the resulting array (`b`) to a constant of your choice.

   All the elements of array `a` should be inserted, in the same order, in array `b`.

   Define the weakest pre-condition and the strongest post-condition possible. Implement the method so that it verifies.

   ```
   method resize(a: array<int>) returns (z: array<int>)
   ```

4. Specify and implement method `reverse`. This method receives an array `a` and returns a new array (`b`) in which the elements of a appear in the inverse order.

   For instance, the inverse of array `a =[0, 1, 5, *, *]`, where `'*'` denotes an uninitialized array position, results in `b =[5, 1, 0, *, *]`.

   Define the weakest pre-condition and the strongest post-condition possible. Implement the method so that it verifies.

```
method reverse(a:array<int>, n:int) returns (z:array<int>)
```