Assignment 1 – Software Analysis

Deductive verification with Dafny

Claudio Maggioni

Contents

1 Choice of Sorting Algorithm

I decide to implement and verify the correctness of selection sort. The algorithm sorts a given list in place with average and worst-case complexity $O(n^2)$. It works by iteratively finding either the minimum or maximum element in the list, pushing it to respectively either the beginning or the end of the list, and subsequently running the next iteration over the remaining $n - 1$ elements.

For the sake of this assignment, I choose to implement and analyze the variant where the minimum element is computed. The pseudocode of selection sort is given in algorithm 1.

Algorithm 1 Selection sort

Require: a list of values
Ensure: a is sorted in-place
1: **if** $a = \emptyset$ **then**
2: **return**
3: **end if**
4: $s \leftarrow 0$
5: **while** $s < |a|$ **do**
6: $m \leftarrow \arg\min_x a[x]$ for $s \leq x < |a|$
7: **swap** $a[x]$, $a[s]$
8: $s \leftarrow s + 1$
9: **end while**

I choose this algorithm due to its procedural nature, since I feel more comfortable tackling loops instead of recursive calls when writing verification code as we already covered them in class.

Additionally, given the algorithm incrementally places a ever-growing portion of sorted elements at the beginning of the list as s increases, finding a loop invariant for the **while** loop shown in the pseudocode should be simple as I can formalize this fact into a predicate.

1.1 Dafny implementation

To implement and verify the algorithm I use Dafny, a programming language that is verification-aware and equipped with a static program verifier.

I first write an implementation of the pseudocode, listed below.

Listing 1 Implementation of selection sort in Dafny

```
1  method SelectionSort(a: array<int>)
2  {
3    if (a.Length == 0) {
4      return;
5    }
6
7    var s := 0;
8
9    while (s < a.Length - 1)
10   {
11     var min: int := s;
12     var i: int := s + 1;
13
14     while (i < a.Length)
15     {
16       if (a[i] < a[min]) {
17         min := i;
18       }
19       i := i + 1;
20     }
21
22     a[s], a[min] := a[min], a[s];
23     s := s + 1;
24   }
25 }
```

The implementation is only slightly different from the pseudocode. The biggest difference lies in the inner **while** loop at lines 14-20. This is just a procedural implementation of the assignment

$$m \leftarrow \arg\min_{x} l[x] \qquad \text{for} \qquad s \leq x < |l|$$

at line 6 of the pseudocode.

2 Verification

I now verify that the implementation in listing 1 is correct by adding a specification to it, namely a method precondition, a method postcondition, and invariants and variants for the outer and inner loop.

2.1 Method precondition and postcondition

Aside the `array<int>` type declaration, no other condition is needed to constrain the input parameter `a` as a sorting algorithm should sort any list. Therefore, the method precondition is

```
requires true
```

which can just be omitted.

Regarding postconditions, as the assignment description suggests, we need to verify that the method indeed sorts the values, while preserving the values in the list (i.e. without adding or deleting values).

We can define the sorted condition by saying that for any pair of monotonically increasing indices of a the corresponding elements should be monotonically non-decreasing. This can be expressed with the predicate:

```
predicate sorted(s: seq<int>)
{
  forall i,j: int :: 0 <= i < j < |s| ==> s[i] <= s[j]
}
```

According to advice given during lecture, we can express order-indifferent equality with the predicate:

```
predicate sameElements(a: seq<int>, b: seq<int>)
{
  multiset(a) == multiset(b)
}
```

Therefore, the method signature including preconditions and postconditions is:

```
method SelectionSort(a: array<int>)
  modifies a
  ensures sorted(a[..])
  ensures sameElements(a[..], old(a[..]))
```