Assignment 1 – Software Analysis

Deductive verification with Dafny

Claudio Maggioni

1 Choice of Sorting Algorithm

In order to choose I browse the implementations of some classical sorting algorithms to find one that strikes a good tradeoff between ease-of-proof and a solid understanding of how it works on my part.

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: $s \leftarrow 0$
2: **while** $s < |a|$ **do**
3: $\quad m \leftarrow \arg\min_x a[x]$ for $s \leq x < |a|$
4: $\quad$ **swap** $a[x]$, $a[s]$
5: $\quad s \leftarrow s + 1$
6: **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 3 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[..]))
```

2.2 Outer loop variant and invariant

As mentioned already, the outer **while** loop in selection sort strongly relates with the incremental selection of minimum values from the non-processed part of the list (i.e. for indices in $[s, |a|)$) and them being moved to the beginning of the list in the correct order. Indeed, the outer loop maintains two main properties:

- The processed elements (i.e. indices $[0, s)$) are sorted, as the elements in them are the minimum, the second-minimum, the third-minimum and so on in this order;
- As all the processed elements have been selected as minimums, all of them are by definition greater than or equal than the non-processed elements. Even if this property seems trivial, it is quite important to ensure we can simply "append" elements to the processed portion as they will for sure be greater than all the values in indices $[0, s)$.

We can formalize these two facts respectively with two Dafny loop invariants on the outer while loop:

```
invariant s >= 1 ==> sorted(a[0..s]) // saying one item is sorted makes little sense
invariant forall i,j: int :: 0 <= i < s <= j < a.Length ==> a[i] <= a[j]
```

Since this loop has an index s iterating over the values in $[0, |a| - 1]$ (note that we condiser the value of s after loop termination as well) in steps of one the corresponding loop invariant and variant on the index are quite straightforward[1]. Indeed, they are respectively:

```
invariant s <= a.Length - 1
decreases a.Length - s
```

Finally, since the only mutation performed on the list is the **swap** operation at line 22 of the code, and a swap operation does not create or destroy values (either it swaps the position of two values or it swaps the same position with itself – i.e. doing nothing), the order-indifferent equality predicate can be simply added to this loop as an invariant

```
invariant sameElements(a[..], old(a[..]))
```

However trivial of a fact as it is, Dafny requires it as an invariant to complete the proof of correctness, and therefore we need to be a little redundant.

[1] We specify loop variants as well for completeness' sake, even if Dafny is able to infer them in these circumnstances.

2.3 Inner loop variant and invariant

The inner loop implements the *arg min* expression used in the assignment at line 6 of the pseudocode. From this fact we can easily say this loop does not mutate a in any array and therefore the order-indifferent equality predicate holds:

```
invariant sameElements(a[..], old(a[..]))
```

Then, we can define the invariant for the index i, which iterates over the values in $[s+1, |a|]$ in single increasing steps. We also know that the loop condition is based on the upper bound of this interfal, so we can define the loop variant as well:

```
invariant s + 1 <= i <= a.Length
decreases a.Length - i
```

The only assignments to variable *min* are at line 11 and line 17 of the pseudocode, where *min* is initialized to s and *min* is assigned the value of i respectively. Therefore we can say the value of *min* at any point where is in scope is either s or $\in [s+1, |a|)$ as it is assigned a possible value of i *inside* the loop. We can formalize this fact with an invariant:

```
invariant s <= min < a.Length
```

Finally we can define a loop invariant for the values of the list element referenced by *min*. As the loop scans for values referenced by i less than `a[min]`, updating *min* with the value of i when this is true, we can say that for all values smaller than i in the scanned region `a[min]` indeed contains the smallest value. This fact in predicate form results in the following loop invariant:

```
invariant forall j: int :: s <= j < i ==> a[min] <= a[j]
```

2.4 Dafny implementation with specification

The Dafny implementation of selection sort complete with the specification explained so far can be found in the file `selectionsort.dfy` found in the repository:

gitlab.com:usi-si-teaching/msde/2022-2023/software-analysis/maggioni/assigment-1

on the *main* branch.

3 Conclusions

Writing specifications with Dafny turned out to indeed be a challenge. Choosing the right predicates and the right places to put them to satisfy Danfy and Boogie, its embedded theorem prover, was an iterative process requiring many attempts and careful scrutiny at the reported errors.

As part of this process, I have introduced many `assert` and `assume` clauses to tackle the algorithm portion by portion. As `assume`s are basically "cheats" w.r.t. the correctness proving process and `assert`s were redundant, no such statement survives in my implementation.

Lines 3-5 of the algorithm's implementation (a simple check of the list a being empty, and an immediate `return` if this is the case) are not strictly needed according to the pseudocode. However, handling this edge case specifically simplifies somewhat the process of determining the loop invariants as we can assume in all further statements that the list is non-empty.

Indeed, the hardest step in the verification process was determining appropriate loop invariants to satisfy the theorem proven. The need for redundant order-indifferent equality predicates in both loops is quite counter-intuitive as those invariants merely re-state a property that is true across the method.