Prof. Dr. Christoph Matheja
Theory of Correct Systems @ UOL
Software Systems Engineering @ DTU

Program Verification – Assignment 4

- Submit your solutions online in groups of 1-3 before the deadline.
- The tasks are numbered consecutively across all assignments. If you have questions, you can thus refer to the task number.
- You can use the concepts and notation from the course material without explanation. Any other definitions or notations must be defined before they are used. Similarly, theorems or mathematical properties that are neither introduced in the course nor well-known from typical undergraduate courses, for example discrete math or logic, need to be introduced and proven first. When in doubt about using a property you know from another course, ask us first.

T12: Sorting sequences (30 Points)

The goal of this exercise is to verify a recursive version of insertion sort. The Viper code below implements insertion sort in Viper. It also contains method contracts describing a first specification.

```
domain X {
    function sorted(xs: Seq[Int]): Bool

    axiom {
        true // TODO: define sorted such that
             // sorted(xs) holds iff
             // xs is sorted in ascending order
    }
}

// TODO: should verify
method insertion_sort(xs: Seq[Int])
    returns (ys: Seq[Int])
    ensures sorted(ys)
    ensures |ys| == |xs|
{
    if (|xs| == 0) {
        ys := Seq[Int]() // empty sequence
    } else {
        // xs[1..] is xs without the first element
        ys := insertion_sort(xs[1..])
        ys := insert(xs[0], ys)
    }
}
```

```
// TODO: should verify
// You can add postconditions
method insert(x: Int, xs: Seq[Int])
    returns (ys: Seq[Int])
    requires sorted(xs)
    ensures sorted(ys)
{
    if (|xs| == 0) {
        ys := Seq(x) // sequence containing just x
    } else {
        var y:Int := xs[0]
        if (x < y) {
            // ++ is concatenation of sequences
            ys := Seq(x) ++ xs
        } else {
            ys := insert(x, xs[1..])
            ys := Seq(y) ++ ys
        }
    }
}
```

(a) *Define the uninterpreted function* ***sorted(xs)*** *such that it returns* ***true*** *if and only if the sequence* ***xs*** *is sorted in ascending order.*

(b) *Verify insertion sort by extending the contracts in the above methods. You can add postconditions and assertions. However, your final solutions should contain no assumptions and no additional preconditions.*

(c) *Give alternative implementations of the above methods that satisfy the given contracts, but do* ***not*** *sort the given input sequence.*

(d) *How can we strengthen the given contracts such that our method for insertion sort indeed returns a sorted version of the original input sequence* ***xs****? Formalize a proposal by providing an abstract method equipped with an improved contract. You do not have to verify that the given implementation satisfies your updated contract.*

T13: Filtering sequences (30 Points)

In this exercise you will implement and verify a function that removes all instances of an element from a list.

```
domain X {
    function seq_to_set(xs: Seq[Int]): Set[Int]

    axiom def_seq_to_set {
        // TODO
    }
}

method remove_all(xs: Seq[Int], x: Int)
    returns (ys: Seq[Int])
    ensures !(x in seq_to_set(ys))
{
    // TODO
}
```

(a) *Axiomatize the function* `seq_to_set(xs:Seq[Int]):Set[Int]` *such that it creates a set that has all the elements in the list* `xs`.

(b) *Implement the method* `remove_all(xs,x)` *such that it returns the list* `xs` *with all the instances of* `x` *removed.*

(c) *Verify the method according to the given contract.*

Hint: You can create empty sequences and sets with `Seq()` and `Set()` and singleton sequences and sets with `Seq(x)` and `Set(x)`, respectively. You can concatenate sequences with `++` and compute the union of sets with `union`. You can also compute the size of a sequence or set `xs` with `|xs|`, access the i-th element of a sequence with `xs[i]`, and obtain a subsequence with `xs[from..to]`. Further details on the supported operations are found in the Viper tutorial: https://viper.ethz.ch/tutorial/

T14: Method Unfoldings (40 Points)

We discussed in class why modular verification of procedure calls is sound for partial correctness by considering finite inlinings C^n of procedure calls.
One of the key steps in the proof argument is that validity of all finite inlinings implies validity of the original program. More formally:

$$\text{If, for all } n \in \mathbb{N},\ \models \{\{\ F\ \}\}\ C^n\ \{\{\ H\ \}\} \text{ then } \models \{\{\ F\ \}\}\ C\ \{\{\ H\ \}\}.$$

Explain (i.e. you can make reasonably simple observations that can be proven by induction on the program structure without giving a detailed proof) why the above statement is correct.
Hint: Notice that the statement concerns semantic judgements, not syntactic ones.