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

Program Verification – Assignment 6

- 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.

T18: Warmup: Triangle sum (10 Points)

In this exercise, we use Viper for reasoning about triangle numbers.

(a) *Prove the first assertion in the* `client` *method, without using* `lemmaTriangleSum`*.*

(b) *Prove the lemma* `lemmaTriangleSum`*, and verify the second assertion in* `client`*. Remember that you must also prove termination.*

```
function triangleSum(n: Int): Int
    requires n >= 0
{
    n == 0 ? 0 : n + triangleSum(n-1)
}

method client(n: Int)
    requires n >= 0
{
    // TODO: verify this
    assert triangleSum(5) == 15

    // TODO: verify this
    assert triangleSum(n) == n * (n + 1) / 2
}

// TODO: verify this
method lemmaTriangleSum(n: Int)
    requires n >= 0
    ensures triangleSum(n) == n * (n + 1) / 2
```

T19: Computing sums (40 Points)

In this task, our goal is to develop an approach for reasoning about sums of the form

$$\sum_{i=n}^{<m} f(i) = \begin{cases} f(n) + f(n+1) + \cdots + f(m-1) & \text{if } n < m \\ 0 & \text{otherwise} \end{cases}$$

in Viper, where n and m are natural numbers and f is some (uninterpreted) function mapping integers to integers.
We consider two different ways for computing the above sum: The function `sumUp` computes the sum from the first element to the last element. By contrast, the function `sumDown` computes the sum from the last element to the first element.

(a) *Implement the functions* `sumUp` *and* `sumDown`*.*

(b) *Show that both functions compute the same sum by proving* `lemmaSameSum`*. Remember to also prove termination. Hint: You may want to introduce additional lemmas.*

(c) *Verify the method* `client`*.*

```
// uninterpreted function (do not change)
function F(x: Int): Int

// TODO: implement
function sumUp(n: Int, m: Int): Int

// TODO: implement
function sumDown(n: Int, m: Int): Int

// TODO: implement
method lemmaSameSum(n: Int, m: Int)
   requires n <= m
   ensures sumUp(n,m) == sumDown(n,m)

// TODO: verify this
method client(n: Int, m: Int)
    returns (sum: Int)
    ensures sum == sumUp(n, m)
{
   sum := 0
   var i: Int := m
   while (i != n) {
       i := i - 1
       sum := sum + F(i)
   }
}
```

T20: Reversing sequences (50 Points)

This exercise concerns reasoning about sequences in Viper.

(a) *Prove that reversing a sequence twice yields the original sequence by implementing the method* `lemma_reverse_reverse(xs:Seq[Int])`.

(b) *Implement the method* `reverse_method`, *and verify it with Viper*

(c) *Extend the method* `client` *such that it verifies.*

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

    axiom def_reverse_nil {
        reverse(Seq()) == Seq()
    }

    axiom def_reverse_cons {
        forall x: Int, xs: Seq[Int] ::
            reverse(Seq(x) ++ xs) == reverse(xs) ++ Seq(x)
    }
}

method client(xs: Seq[Int]) {
    var ys: Seq[Int]
    ys := reverse_method(xs)
    var zs: Seq[Int]
    zs := reverse_method(ys)

    // TODO: verify this
    assert xs == zs
}

// TODO: implement
method reverse_method(xs: Seq[Int])
    returns (zs: Seq[Int])
    ensures zs == reverse(xs)

// TODO: implement
method lemma_reverse_reverse(xs: Seq[Int])
    ensures reverse(reverse(xs)) == xs
```