

Program Verification – Assignment 5

- 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.

T15: Prime numbers (20 Points)

Find a suitable loop invariant to verify that the method `isPrime(n)` checks whether the number `n` is a prime number.

```
domain X {
    function mathIsPrime(n: Int): Bool
    axiom mathIsPrime_def { forall n: Int :: mathIsPrime(n) <==>
        2 <= n && (forall d: Int :: 2 <= d < n ==> n % d != 0)
    }
}

method isPrime(n: Int)
    returns (res: Bool)
    requires n >= 0
    ensures res == mathIsPrime(n)
{
    res := (2 <= n)
    var i: Int := 2
    while (res && i < n) { // TODO: add invariants
        if (n % i == 0) {
            res := false
        }
        i := i + 1
    }
}
```

T16: Unbounded for loops

(40 Points)

In this task, we consider verification of *unbounded* for-loops (for-loops that cannot be unrolled statically) of the form

```
for i in [L..U] invariants { body }
```

where L and U are expressions of type `Int`.

- (a) Extend our operational semantics by providing inference rules for unbounded for loops.
- (b) Define Hoare-style proof rules for reasoning about unbounded for loops.
- (c) Define an encoding of unbounded for-loops into IVL1. You may want to add (in)variants to your encoding that are shared among all for-loops
- (d) Use your insights from (a)-(c) to encode the program below into Viper and verify it.

Note: We updated the postcondition to match the provided Viper code.

```
method triangleSum(n: Int)
  returns (res: Int)
  requires n >= 0
  ensures res == n * (n+1) / 2
{
  res := 0
  for i in [1..n]
    // invariant ....
  {
    res := res + i
  }
}
```

T17: Breaking loops

(40 Points)

We have seen how `while` loops can be verified using loop invariants. However, many programming languages offer additional commands to manipulate a loop's control flow. For example, the `break` command can be used to leave a loop without executing the rest of the loop body (i.e. the code following `break`).

- (a) Define an encoding for while loops containing `break` commands in their body into IVL1. That is, define `encode[while (G) invariant I { C }]` and `encode[break]`. You are allowed to introduce a new variable in your encoding to keep track of whether a `break` command was executed (and use it in your specifications), if necessary.

- (b) Apply your encoding to the body of the method `findOccurrence`, adding any invariants that are necessary for verification. Check your results with Viper.

```
method findOccurrence(s: Seq[Int], e: Int) returns (res: Int)
  ensures res > -1 ==> res < |s| && s[res] == e
{
  res := -1
  var i: Int := 0
  while (i < |s|) // TODO: apply your encoding to this loop
    invariant 0 <= i <= |s|
    // invariant ...
  {
    if (s[i] == e) {
      res := i
      break
    }
    i := i + 1
  }
}
```