Formal Methods in Software Development
Exercise 6 (December 5)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

The result is to be submitted by the deadline stated above *via the Moodle interface* of the course as a *.zip or .tgz* file which contains

1. a PDF file with
   - a cover page with the course title, your name, Matrikelnummer, and email address,
   - a (nicely formatted) copy of the *.java/.theory* file(s) used in the exercise,
   - the deliverables requested in the description of the exercise,
   - for each program method, a screenshot of the "Analysis" view of the RISC ProgramExplorer with the specification/implementation of the method and the (expanded) tree of all (non-optional) tasks generated from the method,
   - for each program method, a screenshot of the corresponding "Semantics" view and an informal interpretation of the method semantics;
   - for each task an explicit statement whether the goal of the task was achieved or not and, if yes, how (fully automatic proof, immediate completion after starting an interactive proof, complete or incomplete interactive proof),
   - for each truly interactive proof, a screenshot of the corresponding "Verify" view with the proof tree,
   - optionally any explanations or comments you would like to make;
2. the *.java/.theory* file(s) used in the exercise,
3. the task directory (*.PETASKS**) generated by the RISC ProgramExplorer.

Email submissions are *not* accepted.

Exercise 6: Insertion Sort

Use the RISC ProgramExplorer to specify the following program, analyze its semantics, and verify its correctness with respect to its specification:

```
class Exercise6
{
  // sort array a in ascending order by the "insertion sort" algorithm
  public static void sort(int[] a)
  {
    int n = a.length;
    int i = 1;
    while (i < n)
    {
      int x = a[i];
      int j = shift(a, i, x);
      a[j] = x;
      i = i+1;
    }
  }

  // shift all elements a[j..i-1] greater than x by one element
  // return position j into which x is to be inserted after the shift
  public static int shift(int[] a, int i, int x)
  {
    while (i > 0 && a[i-1] > x)
    {
      a[i] = a[i-1];
      i = i-1;
    }
    return i;
  }
}
```

First, create a separate directory in which you place the file *Exercise6.java*, `cd` to this directory, and start `ProgramExplorer&` from there. The task directory *.PETASKS** is then generated as a subdirectory of this directory.

Then perform the following tasks:

1. (35P) Derive a suitable specification of `sort` (clauses `requires`, `assignable`, `ensures`) and annotate the loop in the body of the method appropriately (clauses `invariant` and `decreases`). Based on these annotations analyze the semantics of `sort` (of the whole loop and of the method body) and verify the correctness of the method with respect to its specification.

   > Since `shift` has not yet been specified, the semantics of the loop body cannot yet be analyzed and it cannot yet be verified that the loop invariant is preserved by every loop iteration; however, all other tasks can be solved.

For the purpose of this exercise, in the specification of `sort` it suffices to state that the result array is sorted; it is not necessary to establish a relationship between the elements of the pre-state array and those of the post-state array. However, do not forget to specify the non-nullness status and the length of the post-state array.

In the invariant of the loop, specify which part of the array has already been sorted and which has remained unchanged (along with all necessary minor conditions on a, n and i).

2. (35P) Develop a suitable specification of `shift` and complete the analysis of the semantics and the verification of all tasks of `sort`.

   The specification of `shift` must describe the return value j and the post-state value of a in terms of j (along with all necessary minor conditions that concern the non-nullness status and the length of a and the range of j): all elements from j to $i-1$ are shifted by one position and, unless j is zero, $j-1$ denotes a position at which a holds an element not smaller than x (while all shifted elements are bigger than x); all elements up to j or after i are not shifted.

3. (30P) Annotate the loop in the body of `shift`, analyze the semantics of the method, and verify its correctness with respect to its specification.

   The loop invariant is essentially a generalization of the postcondition; it describes that part of a that has been changed and that which has remained unchanged.

By above tasks, you ultimately deliver for `sort` and `shift` all results that have been requested in Exercise 5 (if you cannot show all required verification conditions for one part of the exercise, you may nevertheless continue with the subsequent parts).

Among all verification tasks, the only complicated one is to show in Part 2 that the invariant of the loop in `sort` is preserved by every iteration of the loop (which invokes `shift`): the core is to show that if the loop was sorted at the entry of the loop body up to position $i-1$, it is sorted at the end of the loop body up to position i, which involves a comparison of elements at positions k and $k+1$. This proof proceeds by case distinction whether $k+1$ is less than j, equal to j, equal to $j+1$ or bigger than $j+1$ (command `case`; since the return value j of `shift` is indicated in the proof by the term $value_1(s)$ for some program state s, the command has form `case k+1 ... value__1(...) ...`).

Otherwise, all proofs may proceed by application of `decompose`, `split`, `instantiate`, `scatter`, and `auto`.

Bonus (10 Points): give an exact specification of `sort` (no verification is required).

Hint: in addition to specify that the result array is sorted, you must specify the existence of a one-to-one mapping of indices from the original array to indices of the result array (such mappings are just values of type `ARRAY INT OF INT`).