Formal Methods in Software Development
Exercise 3 (November 14)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

October 25, 2016

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 3: Searching for the Maximum

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

```
class Exercise3
{
  // returns index of maximum element in array a
  // of non-negative integers (-1, if a is empty)
  public static int maximum(int[] a)
  {
    int n = a.length;
    if (n == 0)
      return -1;
    else
    {
      int j = 0;
      int m = a[0];
      int i = 1;
      while (i < n)
      {
        if (a[i] > m)
        {
          j = i;
          m = a[j];
        }
        i = i+1;
      }
      return j;
    }
  }
}
```

Create a separate directory in which you place the file *Exercise3.java*, `cd` to this directory, and start `ProgramExplorer &` from there. The task directory *.PETASKS** is then generated as a subdirectory of this directory. (If you use the virtual course machine, place the directory for this exercise into the home directory of the guest user; in particular, do not place it into the directory shared with the host computer).

Then perform the following tasks:

1. (25P) Specify the method by an appropriate contract (clauses `requires`, `assignable`, and `ensures`).

   Do not forget the non-null status of the array.

2. (25P) Annotate the loop with an appropriate invariant and termination term (clauses `invariant` and `decreases`).

   In the invariant, start with that part of the precondition that is still true and then focus on your knowledge about the values of the variables changed in the loop.

3. (25P) Investigate the computed semantics of the method, in particular the transition relations and termination conditions of

   - the loop body
   - the loop itself,
   - the whole method.

   in order to judge the adequacy of your annotations. Give an informal interpretation of the semantics (and your detailed explanation whether respectively why it seems adequate).

   > Take the interpretation and explanation serious; not only will they be judged for the credit of this part of the exercise, they will also help you to detect errors that would be hard to find in the later proofs. In particular, a description in just two sentences will not do to get full credit.

4. (25P) Verify all (non-optional) tasks generated from the method. Only few of them should require interactive proofs; most of these can probably be performed just by application of `decompose`, `split`, `scatter` and `auto`.

   > The only more complex cases should be the proofs that the invariant is preserved and that the method body ensures the postcondition; here it is wise to first perform a `decompose` and then a `split` corresponding to the two branches in the method respectively the loop body (if you immediately perform a `scatter`, you have to make a `split` in each of the resulting proof obligations which considerably blows up the proof).
   >
   > Furthermore, if a proof results in the knowledge *executes_*(s) and *returns_*(s), this indicates that program state s is both the result of a normal execution and of executing the `return` statement, which is a contradiction; this contradiction can be exhibited and the the proof situation thus closed by executing the proof command `expand executes_, returns_;`.