Formal Methods in Software Development
Exercise 4 (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 section for each part of the exercise with the requested deliverables and optionally any explanations or comments you would like to make;
2. the RISCAL specification (`.txt`) file(s) used in the exercise;
3. the `.java`/`.theory` file(s) used in the exercise,
4. the task directory (`.PETASKS*`) generated by the RISC ProgramExplorer.

Email submissions are *not* accepted.

Exercise 4: Verifying a Program by Checking and Proving

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

```
class Exercise4
{
  // returns true if "x" occurs among the first "n" elements of "a"
  public static boolean has(int[] a, int n, int x)
  {
    boolean r = false;
    for (int i = 0; i < n && !r; i++)
    {
      if (a[i] == x) r = true;
    }
    return r;
  }

  // compacts array "a" to a "set" that does not contain duplicate elements;
  // the distinct elements are moved to the initial portion of "a" and
  // their number is returned as a result.
  public static int toset(int[] a)
  {
    int n = a.length;
    int ia = 0;
    int ib = 0;
    while (ia < n)
    {
      boolean has = has(a,ib,a[ia]);
      if (!has)
      {
        a[ib] = a[ia];
        ib = ib+1;
      }
      ia = ia+1;
    }
    return ib;
  }
}
```

The function `toset` moves the different elements of array a to the initial positions of a and returns their number as a result; for instance, if we have initially $a = [2, 3, 5, 3, 2, 1, 3]$, the procedure returns with the new value $a = [\underline{2, 3, 5, 1}, 2, 1, 3]$ and result 4.

In detail, perform the following tasks:

1. (40P) For a first validation of specification and annotations, take the RISCAL specification file `toset.txt` which embeds an algorithmic version of above code. Please note that here `toset` is a function that takes the array a and its length n as an argument and returns as a result a record *result* whose component *result.a* holds the new array and whose component *result.n* holds the number of different elements in it.

   Equip the procedures with suitable pre-conditions and post-conditions. Validate the specification by checking the corresponding condition and interpret the results of the checks (are your specifications as expected?). Then equip the loops with invariants and termination terms and check the generated verification conditions. Also try to prove the conditions for arbitrary type bounds (but it may be the case that not all conditions can be automatically proved).

Hints: in the specification of `toset`, specify (1) that the initial portion of the result has only different elements, (2) that each of the *n* elements of *a* occurs in this initial portion and (3) that this initial portion holds no other values than the first *n* elements of *a*. In the loop invariant, specify (along with all necessary range conditions on *ia* and *ib*), appropriate generalizations of the three output conditions by considering which part of the array has been already processed before the current loop iteration.

2. (60P) Create a separate directory in which you place the file `Exercise4.java`, `cd` to this directory, and start `ProgramExplorer &` from there. The task directory `.PETASKS*` is then generated as a subdirectory of this directory.

   Derive suitable specifications and loop annotations of the functions, analyze and interpret their semantics (menu option "Show Semantics"), and prove the generated verification conditions.

   - In the specification of function `has`, do not forget to specify the non-nullness of *a*. In the loop annotation, do not forget to explicitly add all information from the input condition. The proofs should succeed by application of `scatter` (respectively `decompose` and `split`) and `auto` (respectively `auto` *`label`*).
   - In the specification of function `toset`, also specify the non-nullness of *a* and that *a* is modified by the procedure. In the postcondition, specify again all information from the precondition, and also the length of the array in the post-state (which in real Java can actually not change). In the invariant, specify all the information from the precondition, the range conditions on variables *n*, *ia*, and *ib*, which part of *a* has not changed yet, and the appropriately generalized versions of the output conditions.

   The deliverables are for both functions the same that have been requested in Exercise 3 (please note that the verification of `toset` is independent of the verification of `has`).

   Most verification tasks can be performed successfully by application of `scatter` (respectively `decompose` and `split`) and `auto` (respectively `auto` *`label`*). A minor exception is the proof of the postcondition of `toset` where an application of `instantiate` may be required.

   The only more complex proof is that the invariant of the loop of `toset` is preserved. Use here `decompose` and `split` to split the proof into two branches corresponding to the two branches of the conditional in the loop body; then proceed with the individual subproofs for the conditions in the invariant. Always think about which part of the invariant in which branch you consider to give you intuition about the proof.

   Some conditions can be solved by simple application of `scatter` and `auto` (or `auto` *`label`*), others require more work. If a disjunction or implication appears as an assumption, consider the application of `split` on that formula. If there appears a conditional formula `if` *F* `then ... else ... endif`, perform a case distinction by executing `case` *F*. Various branches require one or two applications of `instantiate`. To perform the whole proof the application of about 30 proof commands is needed.

   Complete as many proof branches as you can; analyze those where you failed and give some interpretation of what the proof branch indicates and what you think the reason of the failure is.