Formal Methods in Software Development
Exercise 3 (November 29)

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 3a (40P): Proving Verification Conditions

Take the six verification conditions A, T, B1, B2, B3, and C manually derived in Exercise 2 and checked there with RISCAL (if you did not solve that exercise, you may ask a colleague for these conditions or take them from the distributed sample solution). Simplify these conditions such that they do not any more contain occurrences of function num(.,.):

- In the invariants, simply drop the corresponding parts (but keep everything else);
- In the postcondition say that there exists *some* value n in the range $0 \ldots N$ such that b contains 0 in the first n positions and 1 everywhere else ($\exists$n:int. 0≤n ∧ n≤N ∧ ...).

The goal of this exercise is to *prove* the simplified conditions with the RISC ProofNavigator (in the style of the verification of the "linear search" algorithm presented in class) for arrays of *arbitrary* length N.

For this purpose, write a declaration file with the following structure

```
newcontext "exercise3a";

// arrays as presented in class (except ELEM = INT)
...
ELEM: TYPE = INT;

// program variables and mathematical constants
N:INT; a:ARR; b:ARR; i:INT; j:INT;

// make the precondition on parameter "a" an axiom
Parameters: AXIOM N = length(a) AND ... ;

// precondition of loop, postcondition of loop, loop invariant and measure
Input: BOOLEAN = ... ;
Output: BOOLEAN = ... ;
Invariant: (ARR,INT,INT)->BOOLEAN = LAMBDA(b:ARR,i:INT,j:INT): N = length(b) AND ... ;
Termination: (ARR,INT,INT)->INT = LAMBDA(b:ARR,i:INT,j:INT): ... ;
...
```

Formulate in this file the precondition on the parameters a as an "axiom"; this is a formula that will be automatically added as an assumption to every subsequent proof. The information that constant N is the length of array a should be added to the precondition; likewise the information that N is the length of b should be added to the invariant.

Please note that the condition B3 corresponding to the third branch of the loop body does *not* involve any occurrence of the auxiliary variable e (which is therefore also not declared above); this variable vanishes by the three applications of the rule for (array) assignments corresponding to the assignments that perform the swap of the array elements.

It should be possible to perform the proofs of conditions A, T, B1, B2, and B3 with the commands expand, scatter, and auto; in the proof of B3 it is necessary to expand the definitions of the array operations get, put, length and content.

The only proof where some more insight is required is the proof of condition C where an explicit instantiation of the existentially quantified postcondition formula EXISTS(n:INT):... is required (command instantiate...in...). However, before doing so, one must distinguish between the case

that the loop terminates with $i = j$ and the other case (command `case i=0`). Furthermore, in the case $i = 0$ one must distinguish between case $b[i] = 0$ and the other case (command `case get(b,i)=0`). This gives in total three proof branches; after the right instantiation each branch can be closed by application of `scatter` and `auto`.

If you get lost in a proof, perform `decompose` rather than `scatter` and possibly delay the application of `expand`.

The deliverables for this exercise consists of the following items:

1. a (nicely formatted) copy of the ProofNavigator file (included as text, not as screenshots);
2. for each proof of a formula F, a readable screenshot of the RISC ProofNavigator after executing the command `proof` F (displaying the proof tree) with explicit statements whether the proof succeeded;
3. if any check gives an error respectively any proof fails, a detailed explanation of the estimated reason of the failure.

Exercise 3b (60P): Verifying a Program by Checking and Proving

We consider the following problem: given an array a of non-negative integer elements, find the maximum element m of a; if a is empty, this shall be indicated by $m = -1$ (which is not a possible array element).

The goal of this exercise is to take the following Java program that solves this problem, and to use the RISC ProgramExplorer to annotate the program with specification and annotations, analyze its semantics, and verify its correctness with respect to its specification:

```
class Exercise3
{
  // returns 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 m = -1;
      for (int i = 0; i < n; i++)
      {
        if (a[i] > m) m = a[i];
      }
      return m;
    }
  }
}
```

In detail, perform the following tasks:

1. (20P) For a first validation of specification and invariants, take the file `maximum.txt` which embeds the RISCAL version of above code in a procedure `maximumElement` and equip this procedure with suitable pre-conditions, post-conditions, invariants, and termination term. Validate (for small values N and M) the annotations (check the procedure and check the *automatically* generated verification conditions; it is not necessary to manually derive them). These validated annotations shall then serve as the basis of the further proof-based verification.
2. (20P) Create a separate directory in which you place the file `Exercise3.java` that contains above Java procedure, `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).

   Specify the method by an appropriate contract (clauses `requires` and `ensures`) and annotate the loop with an appropriate invariant and termination term (do not forget the non-null status of the array). In contrast to RISCAL, you have to specify here also all available information you have from the precondition about the input array (because in Java procedure parameters are not constants).

   Investigate (by application of menu option "Show Semantics" for the procedure) the computed semantics (transition relation and termination condition) of the method and give an informal interpretation of the semantics (and your explanation whether respectively why it seems adequate) in sufficient detail.
3. (20P) 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 proofs should be 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).

The deliverables of this exercise consist of

1. a nicely formatted copy of the RISCAL specification (included as text, not as screenshots);
2. the outputs of the checks (included as text, not as screenshots) with explicit statements whether the checks succeeded;
3. a (nicely formatted) copy of the annotated `.java` file used in the exercise,
4. a screenshot of the corresponding "Semantics" view and an informal interpretation of the method semantics;
5. 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,
6. for each task generated by the RISC ProgramExplorer 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),
7. for each truly interactive proof, a screenshot of the corresponding "Verify" view with the proof tree.