Formal Methods in Software Development
Exercise 6 (December 7)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

September 3, 2015

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: Transforming an Array into a Set

Take the following program whose function `toSet` "compacts" an array *a* (with possibly duplicate elements) into a "set" by moving the distinct elements of *a* to the front and returning the number of these elements.

```
// has(a, n, x) <=> x occurs among the first n elements of a
/*@
  theory uses Base {
    int: TYPE = Base.int;
    intArray: TYPE = Base.IntArray;
    has: PREDICATE(intArray, int, int) =
      PRED(a:intArray, n:int, x:int):
        EXISTS(k:INT): 0 <= k AND k < n AND a.value[k] = x;
  }
@*/
class Exercise6
{
  // removes duplicates from 'a' moving the non-duplicates to the front of 'a'
  // returns the number of distinct elements (all now at the beginning of 'a')
  public static int toSet(int[] a)
  {
    int n = 0;
    int i = 0;
    while (i < a.length)
    {
      boolean h = has(a, n, a[i]);
      if (!h)
      {
        a[n] = a[i];
        n = n+1;
      }
      i = i+1;
    }
    return n;
  }

  // true if and only if 'x' occurs among the first n elements of 'a'
  public static boolean has(int[] a, int n, int x)
  {
    int i = 0;
    boolean result = false;
    while (i < n && !result)
    {
      if (a[i] == x)
        result = true;
      else
        i = i+1;
    }
    return result;
  }
}
```

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.

Now use the RISC ProgramExplorer to specify the program, analyze its semantics, and verify its correctness with respect to its specification. In more detail, perform the following tasks:

1. (35P) Derive a suitable specification of program function `toSet` (clauses `requires`, `assignable`, `ensures`) and annotate the loop in the body of the method appropriately (clauses `invariant` and `decreases`). For this purpose, use the logical function `has` provided in the local theory of the program file.

   Based on these annotations analyze the semantics of `toSet` (of the method body and of the loop) and verify the correctness of the method with respect to its specification.

   > Since the program function `has` is not yet 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.
   >
   > In the postcondition of `toSet` you must ensure the following:
   >
   > a) the new value of a is not null and has the same length as the original value;
   >
   > b) the function result r is in a certain range;
   >
   > c) the first r elements of the new a are all different;
   >
   > d) the first r elements of the new a are contained somewhere in the whole range of the old a;
   >
   > e) the elements in the whole range of the old a are contained among the first r elements of the new a.
   >
   > In the loop invariant you have to state the following:
   >
   > a) both the old and the new a are not null and have the same length;
   >
   > b) range conditions for i and n;
   >
   > c) appropriate generalizations of the conditions (c-e) stated above (think about in which range of the new and the old a these conditions hold);
   >
   > d) the not yet processed part of a has not changed.

2. (10P) Develop a suitable specification of program function `has`.

3. (25P) Annotate the loop in the body of `has` with an invariant and termination term, analyze the semantics of `has` (method body, loop, and loop body), and and verify its correctness with respect to its specifi cation.

4. (30P) Analyze the semantics of the body of the loop in `toSet` and verify that the loop invariant is preserved by every loop iteration.

The last two tasks are independent of each other and can be performed in any order.

By above tasks, you ultimately deliver for `toSet` and `has` all results that have also 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). To get full credit for the proof in task (4), it suffices to show that at all formulas in parts (a,b,d) of the invariant and at least one of the clauses in part (c) is preserved (by both branches of the loop body). For the proof of every additional clause of (c), 10 bonus points will be issued.

The proofs of the verification tasks can be performed by application of the commands `expand`, `scatter`, `decompose`, `goal`, `split`, `case`, `instantiate`, and `auto` (respectively `auto` *`label`* to instantiate a single formula with more terms than the general `auto` does).

Some recommendations for the proofs (in particular for task (4), the only difficult one):

1. The definition of `has` has to be unfolded by `expand has`.
2. To close most branches with quantified goals, `auto` L (with label L of the goal) can be ultimately applied.
3. If you encounter conditional formulas or terms (`if` F `then ... else ... endif`), apply command `case` F (or `split`) to split the proof into two branches.

   In particular, the proof of (4) leads to a conditional assumption corresponding to the two branches in the loop body. For this proof, it is recommended to first apply `decompose` (rather than `scatter`), then split the proof into two branches, and only afterwards apply `scatter`; focus first on the second branch which shows that the invariant is preserved if the body of the `if` statement is not executed (this branch is easier to solve).
4. In the proof of the preservation of an invariant formula $\forall k : P(k)$, this formula is used as an assumption and a modified formula $\forall k : P'(k)$ is used as a goal; this amounts to a proof of $P'(k_0)$ for some constant k_0. Identify the label L of the corresponding assumption and apply `instantiate` k_0 `in` L before continuing.

   Also some other quantified assumptions may need to be explicitly instantiated, in particular those corresponding to the invariant that no two elements in the overwritten part of the array are the same and that the elements in the not yet processed part of the array have remained unchanged.