Formal Methods in Software Development
Exercise 4 (December 10)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

October 14, 2007

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

- A PDF file with
  - a cover page with the title of the course, your name, Matrikelnummer, and email-address,
  - a copy of the ProofNavigator file used in the exercise,
  - for each proof of a formula F, a screenshot of the RISC ProofNavigator after executing the command `proof` F,
  - optionally any explanations or comments you would like to make;
- the RISC ProofNavigator (.pn) files used for the proofs;
- the proof directories generated by the RISC ProofNavigator.

4(KV4): Partitioning an Array

Take the program

```
i = l; j = r;
while (i <= j)
{
  if (a[i] < v)
    i = i+1;
  else if (a[j] >= v)
    j = j-1;
  else
  {
    t = a[i]; a[i] = a[j]; a[j] = t;
    i = i+1; j = j-1;
  }
}
m = i;
```

which describes the *partitioning phase* of the Quicksort algorithm: given an array a, indices $l \leq r$ within that array, and a partitioning value v, the program reshuffles the array $a[l \ldots r]$ such that, for some index $l \leq m \leq r$, $a[l \ldots m-1]$ holds all elements less than v and $a[m \ldots r]$ holds all the others.

Develop an adequate specification of this piece of code as a Hoare triple, find a suitable loop invariant, produce the necessary conditions for verifying the partial correctness of the code, and prove these conditions with the help of the RISC ProofNavigator (as in the previous exercises).

Please note that the index j may become less than l and may thus even become *negative*; likewise the index i may exceed the bounds of the array. Please also note that the specification must state that after the execution of the program the array holds the same elements as before (but possibly in a different order).

Hint: two arrays a and b of length n hold the same values if and only if there exists an index permutation p such that $a[i] = b[p(i)]$. A function $p : \mathbb{N}_n \to \mathbb{N}_n$ is an index permutation if and only if it is bijective, i.e.

$$\forall i \in \mathbb{N}_n : \exists j \in \mathbb{N}_n : i = p(j) \wedge$$
$$\forall i_1, i_2 \in \mathbb{N}_n : p(i_1) = p(i_2) \Rightarrow i_1 = i_2$$

This may be expressed in a RISC ProofNavigator theory as follows:

```
% basis type of permutations
PTYPE: TYPE = ARRAY NAT OF NAT;

% Permutation(p,n) <=> p is a permutation of length n
Permutation: (PTYPE,NAT) -> BOOLEAN =
  LAMBDA(p:PTYPE,n:NAT):
    (FORALL(i:NAT): i<n => (EXISTS(j:NAT): j<n AND i=p[j])) AND
    (FORALL(i1,i2:NAT): i1<n AND i2<n AND p[i1]=p[i2] => i1=i2);

% SameElements(a,b) <=> arrays a and b have same elements
SameElements: (ARR,ARR) -> BOOLEAN =
  LAMBDA(a,b:ARR):
    length(a) = length(b) AND
    (EXISTS(p:PTYPE): Permutation(p, length(a)) AND
      (FORALL(i:NAT): i<length(a) => get(a,i)=get(b,p[i])));
```

You may also make use of the following auxiliary definitions:

```
% the identity permutation
identity: PTYPE;
identityaxiom: AXIOM FORALL(i:NAT): identity[i]=i;

% swap(p,i,j) = the permutation constructed from
%   permutation p by swapping the positions i and j
swap: (PTYPE,NAT,NAT)->PTYPE;
swapaxiom: AXIOM
```

```
  FORALL(p:PTYPE, i:NAT, j:NAT):
    swap(p,i,j)[i]=p[j] AND
    swap(p,i,j)[j]=p[i] AND
    (FORALL(k:NAT): k/=i AND k/=j => swap(p,i,j)[k]=p[k]);

% swapping two elements preserves the permutation properties
swapisperm: FORMULA
  FORALL(p:PTYPE, n:NAT, i:NAT, j:NAT):
    Permutation(p,n) AND i<n AND j<n
     => Permutation(swap(p,i,j), n);
```

The permutations denoted by *identity* and $swap(p, i, j)$ will be the only ones needed in the verification. They are characterized by the axioms *identityaxiom* and *swapaxiom* that will be automatically imported into the proof. Furthermore, you may use the command `lemma` to introduce *swapisperm* as additional knowledge into your proof (the formula itself needs not be proved).

The verification consists of the proof of five conditions A, B1, B2, B3, and C, corresponding to the input condition, the three branches in the loop body, and the output condition:

- The proof of A needs scattering, expansion of definitions, manual and/or automatic instantiations.
- The proofs of B1, B2, and C proceed mainly automatically.
- The proof of B3 has various branches that just need expansions of definitions and instantiations; only the proof that the property *SameElements* is preserved requires more effort: after expanding the definition (both in the knowledge and in the goal), one has to find a suitable permutation p and prove that

  1. p has the properties expected of a permutation (here the lemma stated above may be used) and that
  2. p indeed describes the exchange of elements in the array performed by the loop body. Here a suitable case distinction has to be made.

  In this proof, most branches are finally closed by expanding the definitions of the array operations.