Final exam, February 27, 2012

Duration: 3 hours. The four exercises are independent.
Allowed documents: lecture notes, personal notes, **but no electronic device.**
Mobile phones must be switched off.

Exercise 1 In this exercise, we consider the simple language IMP of the first lectures of this course, where expressions do not have any side effect.

1. *Prove that the triple*

$$\{P\}x := e\{\exists v,\ e[x \leftarrow v] = x \wedge P[x \leftarrow v]\}$$

   is valid with respect to the operational semantics.

2. *Show that the triple above can be proved using the rules of Hoare logic.*

Let us assume that we replace the standard Hoare rule for assignment by the rule

$$\frac{}{\{P\}x := e\{\exists v,\ e[x \leftarrow v] = x \wedge P[x \leftarrow v]\}}$$

3. *Show that the triple* $\{P[x \leftarrow e]\}x := e\{P\}$ *can be proved with the new set of rules.*

Exercise 2 In this exercise, we consider the programming language involving expressions with side effects but no pointers, as defined in Section 5.2 of the lecture notes. We propose to add to this language the construct of `for` loops, with a syntax of the form

$$\texttt{for}\ i = e_1\ \texttt{to}\ e_2\ \texttt{do}\ e$$

This expression introduces a new variable i which is visible only in e, and is immutable in the sense that it cannot be assigned in e. Expressions e_1 and e_2 must be of type `int`, whereas e must be of type `unit`. Informally, the operational semantics is that e_1 and e_2 must be evaluated first, in this order, to values v_1 and v_2. If $v_1 > v_2$ then nothing is executed. Otherwise the loop body e is executed iteratively for i taking each value from v_1 to v_2. It is formalized using the following rules:

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma, \Pi, e_1'}{\Sigma, \Pi, \texttt{for}\ i = e_1\ \texttt{to}\ e_2\ \texttt{do}\ e \rightsquigarrow \Sigma, \Pi, \texttt{for}\ i = e_1'\ \texttt{to}\ e_2\ \texttt{do}\ e}$$

$$\frac{\Sigma, \Pi, e_2 \rightsquigarrow \Sigma, \Pi, e_2'}{\Sigma, \Pi, \texttt{for}\ i = v_1\ \texttt{to}\ e_2\ \texttt{do}\ e \rightsquigarrow \Sigma, \Pi, \texttt{for}\ i = v_1\ \texttt{to}\ e_2'\ \texttt{do}\ e}$$

$$\frac{v_1 > v_2}{\Sigma, \Pi, \texttt{for}\ i = v_1\ \texttt{to}\ v_2\ \texttt{do}\ e \rightsquigarrow \Sigma, \Pi, ()}$$

$$\frac{v_1 \leq v_2}{\Sigma, \Pi, \texttt{for}\ i = v_1\ \texttt{to}\ v_2\ \texttt{do}\ e \rightsquigarrow \Sigma, \Pi, (\texttt{let}\ i = v_1\ \texttt{in}\ e); (\texttt{for}\ i = v_1 + 1\ \texttt{to}\ v_2\ \texttt{do}\ e)}$$

An example program is as follows (where arrays are modeled by references to pure maps of type map α indexed by integers):

```
procedure square(a : ref (map real), n:int)
 requires n ≥ 0
 writes a
 ensures forall i:int. 0 ≤ i < n → select(a,i) = select(a@Old,i) ∗ select(a@Old,i)
 body
  for i=0 to n−1 do
   let x = select(a,i) in a := store(a,i,x∗x)
```

We propose the following Hoare logic rule for the `for` loop:

$$\frac{\{I \wedge v_1 \leq i \leq v_2\} e \{I[i \leftarrow i+1]\}}{\{I[i \leftarrow v_1] \wedge v_1 \leq v_2\} \texttt{for}\ i = v_1\ \texttt{to}\ v_2\ \texttt{do}\ e \{I[i \leftarrow v_2+1]\}}$$

1. *Prove that the rule is correct for partial correctness.*
2. *Is it correct for total correctness?*
3. *Propose an additional rule that can deal with the case $v_1 > v_2$.*
4. *Propose a formula for computing the weakest precondition of a `for` loop, when the loop bounds are values, assuming that an invariant I is given:*

$$\mathrm{WP}(\texttt{for}\ i = v_1\ \texttt{to}\ v_2\ \texttt{invariant}\ I\ \texttt{do}\ e, Q) = ?$$

5. *Propose an extended formula for the case where loop bounds are any expressions, possibly with side effects:*

$$\mathrm{WP}(\texttt{for}\ i = e_1\ \texttt{to}\ e_2\ \texttt{invariant}\ I\ \texttt{do}\ e, Q) = ?$$

6. *Propose a suitable loop invariant for the example program* square *above, and prove it using* WP.

Exercise 3 The following program takes a list l of integers as input and returns two lists (l_1, l_2) where l_1 contains the nonnegative elements of l, and l_2 contains the negative ones.

```
record IntList = { data : int ; next: IntList; }
function split(l:IntList):(IntList,IntList)
 body
  let l1 = ref null in let l2 = ref null in let p = ref l in
  while p ≠ null do
   let n = p in
   p := p→next;
   if n→data ≥ 0 then
    n→next := l1; l1 := n
   else
    n→next := l2; l2 := n
  done;
  (l1,l2)
```

Informally, the specification we want is

- the input list must be null-terminated
- the two output lists are null-terminated
- the list l_1 contains only nonnegative values
- the list l_2 contains only negative values
- all values appearing in l_1 and l_2 must already appear in the input list.

Notice that this specification does not require that all values of the input list should appear either in l_1 or l_2.

1. *Propose an equivalent program without pointers, using the Component-as-Array model, with appropriate pre- and postconditions. It is recommended to use the predicate* mem(x:α,l:list α) *that tells whether a given element x appears in a pure logic list l.*

2. *Propose an appropriate loop invariant for the program of the previous question. Explain informally why your loop invariant is enough to prove the program, in particular regarding separation issues.*

We now consider the Separation Logic approach instead of the Component-as-Array model.

3. *Propose inductive predicates to represent list segments containing respectively any values, nonnegative values, and negative values.*

4. *Specify the program in Separation Logic. Propose a loop invariant and explain informally how the proof proceeds, in particular regarding separation issues.*

Exercise 4 In this exercise, we want to compute on integers modulo p for some $p \geq 2$. We denote by $\mathsf{mod}(v, p)$ the remainder of the division of v by p, and we write $\mathsf{eq\_p}(x, y)$ if $\mathsf{mod}(x, p) = \mathsf{mod}(y, p)$.

We want to represent these integers modulo p by machine integers, which are signed 32-bits integers, denoted by int32. We recall that int32 values range from -2^{31} to $2^{31} - 1$, and that operations of addition, subtraction, and multiplication, respectively denoted add_32, sub_32, and mul_32, are required not to overflow.

To have efficient computations, we do not want to represent integers modulo p only by values from 0 to $p - 1$. Instead, any int32 value v can be an acceptable representation of the number $\mathsf{mod}(v, p)$.

We propose the following annotated function for addition modulo p, with some unknown constant C (with $C > 0$) and a predicate $P(n) = A \leq n \leq B$ for some constants A and B (with $A < B$).

```
function add_p(x:int32,y:int32):int32 =
  requires P(x) ∧ P(y)
  ensures eq_p(result,x+y) ∧ P(result)
  body
    let z = add_32(x,y) in if z < C then z else sub_32(z,C)
```

1. *Find the necessary conditions on A, B, and C to make the postcondition* P(**result**) *valid, assuming there are no integer overflows.*

2. *Find extra conditions on A, B, and C, depending on p, that make the* add_p *function above correct, including absence of integer overflow.*

3. *Propose a body for the function below for subtracting modulo p. Explain why it satisfies the contract.*

```
function sub_p(x:int32,y:int32):int32 =
  requires P(x) ∧ P(y)
  ensures eq_p(result,x−y) ∧ P(result)
  body ?
```

We now fix $B = C - 1$. We propose the following function for multiplication modulo p.

```
function mul_p(x:int32,y:int32):int32 =
  requires P(x) ∧ P(y)
  ensures eq_p(result,x∗y) ∧ P(result)
  body
    let z = mul_32(x,y) in if z < C then z else mod(z,p)
```

4. *What additional conditions on C are required to make this function correct?*

To improve efficiency of multiplication, we propose the following modified body.

```
let z = mul_32(x,y) in if z < C then z else
sub_32(z, mul_32(p, shr_32(mul_32(z, D), K)))
```

where shr_32(x, y) computes $\lfloor x/2^y \rfloor$ for $y \geq 0$.

5. *Assuming the modified* mul_p *function returns the same value as the original function, find the necessary condition on D and K such that there are no integer overflows in the modified function.*

Let $Q(z)$ the formula $f(z) \leq z(D/2^k - 1/p) < f(z)$ with $f(z) = \lfloor z/p \rfloor - z/p$.

6. *Explain why, if $Q(z)$ holds for all $0 \leq z \leq B^2$, then the modified* mul_p *function returns the same value as the original function.*

7. *Give some tight bounds on $f(z)$ that do not depend on z. Deduce from these bounds a simpler condition $Q'(z)$ that implies $Q(z)$. Find a particular value z_0 such that if $Q'(z_0)$ holds then for all z, $0 \leq z \leq B^2$ implies $Q(z)$.*

8. *For $p = 17$ and $C = 170$, verify that $D = 30841$ and $K = 19$ satisfy the required conditions.*