PV: Exercises on Hoare Logic

Wishnu Prasetya

February 21, 2012

1 Hoare Logic

1. Write Hoare-triple specifications for the following programs:

   (a) A program that takes two array a and b as parameters and checks if they have a common element.

   Answer:

   Some technical issue first: the arrays in uPL are infinite; this is an assumption made to simplify the logic. We can think them as representing actual arrays, but we don't care about their actual size.

   Let us assume that we pass two extra parameters `M` and `N` representing the actual size of `a` and `b`.

   $$\{\ \mathtt{M} \geq 0 \wedge \mathtt{N} \geq 0\ \}\ \ \mathtt{haveCommon(a,b,M,N)}\ \ \{\ (\exists \mathtt{i} : 0 \leq \mathtt{i} < \mathtt{M}\ \mathtt{min}\ \mathtt{N} : \mathtt{a[i]} = \mathtt{b[i]})\ \}$$

   (b) A program that takes two arrays a and b as parameters and returns the longest common prefix of a and b.

   Answer:

   The program below will do a side effect on the array `c`, and returns an integer which is the actual size of `c`.

   $$\{\ \mathtt{M} \geq 0 \wedge \mathtt{N} \geq 0\ \}$$

   $$\mathtt{longestPref(a,b,M,N,OUT\ c)}$$

   $$\left\{\begin{array}{l} 0 \leq \mathtt{return} \leq \mathtt{M}\ \mathtt{min}\ \mathtt{N} \\ \wedge\ (\forall \mathtt{i} : \mathtt{i} \leq \mathtt{i} < \mathtt{return} : \mathtt{a[i]} = \mathtt{b[i]}) \\ \wedge\ (\mathtt{return} < \mathtt{M}\ \mathtt{min}\ \mathtt{N}\ \Rightarrow\ \mathtt{a[return]} \neq \mathtt{b[return]}) \\ \wedge\ (\forall \mathtt{i} : \mathtt{i} \leq \mathtt{i} < \mathtt{return} : \mathtt{a[i]} = \mathtt{c[i]}) \end{array}\right\}$$

   (c) A program that sorts an array a.

   Answer:

   The following program will directly sort the array a itself (so it is destructive). I'll leave the specification of a non-destructive version to you as an exercise.

   To express the specification we need to relate the final a to the old a. This means that we need a notation to be able to refer to the old a in the post-condition. This can be done in several ways. Below we'll just use a so-called auxilliary variable A to freeze the value of a just before it is passed to `sort`.

Note that auxilliary variable is not an actual program variable. It is introduced just to allow us to express certain aspects in a specification. So it is a part of a specification.

$$\{\ \mathtt{M} \geq \mathtt{0}\ \}$$

$$\mathtt{A} := \mathtt{a};\ \mathtt{sort}(\mathtt{M}, \mathtt{OUT\ a})$$

$$\{\ (\forall \mathtt{i} : \mathtt{0} \leq \mathtt{i} < \mathtt{M} - \mathtt{1} : \mathtt{a}[\mathtt{i}] \leq \mathtt{a}[\mathtt{i}+\mathtt{1}])\ \wedge\ \mathtt{isPermutation\ a\ A\ M}\ \}$$

where `isPermutation a b N` means that the list formed by the array segment `a[0..N)` is a permutation of that of `b[0..N)`. You still need a formal definition of this concept, so that a tool can check it; I leave it to you to give such a formal definition.

2. Let P, Q be arbitrary predicates, and S be an arbitrary statement.

   What do the following specifications mean: (a) if you use the partial correctness interpretation of Hoare triples, and (b) if you use total correctness interpretation instead?

   (a) $\{\ P\ \}\ \ S\ \ \{\ \mathtt{true}\ \}$
   Answer:
   With the partial correctness interpretation the above specification is valid; however it does not give any useful information to us.
   If we take the total correctness interpretation the above specification does say that S terminates, which is a useful fact to know. It does not put a constraint on the terminal state though (so S can teriminate in any state, as long as it terminates).

   (b) $\{\ P\ \}\ \ S\ \ \{\ \mathtt{false}\ \}$
   Answer:
   If P itself is non-empty (it is not `false`), no real program can establish `false` as the post-condition, because no (end) state can satisfy `false`.
   So, the specification is invalid (in both partial of total correctness) unless P is equivalent to `false`. In the latter case, see below.

   (c) $\{\ \mathtt{false}\ \}\ \ S\ \ \{\ Q\ \}$
   Answer:
   This specification is valid in both partial and total correctness, but gives no useful information.

3. Give for each statement below a reasonable pre-condition that is sufficient to guarantee the post-condition. Try to come up with the weakest possible pre-condition.

   (a) $\{\ ?\ \}\ \ \mathtt{x{+}{+};\ s{:=}s{+}x}\ \ \{\ \mathtt{s{>}0}\ \}$ **Answer:**
   Use the given inference rules to calculate a sufficient pre-condition. Except when you have a loop, they will actually give you the weakest precondiotion. Applying the rules gives us:

   $$\mathtt{s} + \mathtt{x} + \mathtt{1} > \mathtt{0}$$

   (b) $\{\ ?\ \}\ \ \mathtt{yold{:=}y;}\ \ \mathtt{if\ i = 0\ then\ y{:=}0\ else\ y := y/x}\ \ \{\ \mathtt{y{<}yold}\ \}$
   Answer:
   Also by applying our inference rules we obtain:

   $$((\mathtt{i{=}0}) \Rightarrow \mathtt{0{<}y})\ \wedge\ ((\mathtt{i{\neq}0}) \Rightarrow (\mathtt{y/x} < \mathtt{y}))$$

4. We will extend uPL with the following statements. Let P be a predicate.

   (a) `assert` P does nothing (skip) if executed on a state that satisfies P. Otherwise it will abort.

   (b) `miracle` P will 'magically' jump to a state that satisfies P. If there are multiple states satisfying P, one will be chosen non-deterministically.

   (c) Let's define `miracle false` in our semantic as follows:

   $$(\texttt{miracle}\,\texttt{false})\ s \quad = \quad \emptyset$$

   We can now define `assume` as follows:

   $$\begin{array}{lll}\texttt{assume}\,P & = & \texttt{if}\ P\ \ \texttt{then}\,\texttt{skip} \\ & & \qquad\ \ \texttt{else}\ \ \texttt{miracle}\,\texttt{false}\end{array}$$

   You can see this construct as the dual of `assert`.

   Give inference rules to deal with `assert`, `miracle`, and `assume`. Your rules don't have to be *complete*, but they have to be *sound*.

   A rule is complete if it allows you to infer all conclusions that are supposedly inferable with the rule. It is sound if it only allows you to infer valid conclusions.

   Answer:

   - You may want to distinguish when a program terminates normally and abnormally because it aborts. You may want to say that in the first case the terminal state should satisfy some post-condition Q_n and in the latter case another predicate Q_a. However such a distinction cannot be expressed with a plain Hoare triple nor with our abstract semantic (it is too simplistic for that). So, we cannot give a complete rule for `assert`.

     We can still give a sound rule by specifying those pre-conditions that will prevent `assert` to abort. It is this rule:

     $$\frac{P \Rightarrow (P' \wedge Q)}{\{\ P\ \}\ \ \texttt{assert}\ P'\ \ \{\ Q\ \}}$$

   - In terms of our abstract semantic, we can model `miracle` as follows:

     $$\texttt{miracle}\,P\ s \quad = \quad \{t \mid P\ t\}$$

     Note that `miracle false` is also defined (notice that this leads to the definition given in (c)), though the interpretation of this in term of real execution is somewhat strange.

     Given the above model, we can derive this theorem (which you can use as an inference rule):

     $$\frac{P \Rightarrow Q}{\{\ \texttt{true}\ \}\ \ \texttt{miracle}\,P\ \ \{\ Q\ \}}$$

   - By combining the inference rules for `if` and `assume` we can obtain the following rule/theorem:

     $$\frac{P \Rightarrow (P' \Rightarrow Q)}{\{\ P\ \}\ \ \texttt{assume}\ P'\ \ \{\ Q\ \}}$$

5. The basic Hoare logic we have so far is not be able to deal with the following situations. Propose extensions to deal with them.

(a) $\{\ ?\ \}$ `a[i] := x` $\{$ `even j` $\Rightarrow$ `(a[j]=0)` $\}$
Answer:
Treat the assignment to an array element like `a[i] = x` as an assignment that replaces the whole array:

`a := a(i repby x)`

Now we can fall back to the know our old rule to handle assignment.
The expression $a(e_1\ \texttt{repby}\ e_2)$ denotes a new array that replaces the e_1-th element of a with e_2. It is defined indirectly by the following property:

$$a(e_1\ \texttt{repby}\ e_2)[k] \quad = \quad (k = e_1) \ \rightarrow\ e_2 \mid a[k]$$

The RHS is just an if-then-else expression.
So, via the above described transformation we can calculate a sufficient pre-condition for the above statement, namely:

$$\texttt{even j} \quad \Rightarrow \quad (((\texttt{j=i}) \ \rightarrow\ \texttt{x} \mid \texttt{a[j]}) = \texttt{0})$$

(b) $\{\ ?\ \}$ `x := y/x` $\{$ `x < y` $\}$
Answer:
Transform the statement first to:

$$\texttt{assert (x} \neq \texttt{0); x:=y/x}$$

Now we can calculate a sufficient pre-condition, namely:

$$(\texttt{x} \neq \texttt{0}) \ \wedge\ \texttt{y/x < y}$$

Alternatively, if you are pretty sure that in the context where you use the expression `y/x` that `x` will not be 0, you can explicitly add this as an assumption by transforming the statement to this instead:

$$\texttt{assume (x} \neq \texttt{0); x:=y/x}$$

Note that without such an assumption you may not be able to prove your post-condition. So, either you assume it (that $\texttt{x} \neq \texttt{0}$), or prove it.

(c) Let `a` be a finite array whose size is denoted by `#a`. The indices of this array range from 0 up to (but exclusive) `#a`.

$$\{\ ?\ \}\ \ \texttt{x := a[i]}\ \ \{\ (\exists \texttt{k} : \texttt{0} \leq \texttt{k} < \texttt{\#a} : \texttt{x = a[k]})\ \}$$

Answer:
Transform the statement first to:

$$\texttt{assert (0} \leq \texttt{i} < \texttt{\#a); x:=a[i]}$$

Now we can calculate a sufficient pre-condition, namely:

$$(\texttt{0} \leq \texttt{i} < \texttt{\#a}) \ \wedge\ (\exists \texttt{k} : \texttt{0} \leq \texttt{k} < \texttt{\#a} : \texttt{a[i] = a[k]})$$

which can be (equivalently) simplified to just the first conjunct.

6. Sketch a proof showing the validity of each of the specification below. You will need an invariant, and when asked also a termination metric.

(a) This specification in both partial and total correctness interpretation:

$$\{\ \texttt{i}\geq\texttt{0}\ \}\ \ \texttt{while i>0 do i} := \texttt{i}-\texttt{1}\ \ \{\ \texttt{i} = \texttt{0}\ \}$$

Answer:

Use $\texttt{i}\geq\texttt{0}$ as the invariant.

In total correctness we also have to prove termination. Use `i` as the termination metric.

Verify it yourself that these invariant and metric satisfy all the conditions of the Loop Rule.

(b) The following in partial correctness:

$\{\ \texttt{n>0}\ \}$

$\texttt{s} := \texttt{0};\ \texttt{k} := \texttt{1};$
$\texttt{while k<n do}\ \{\ \texttt{s} := \texttt{s+k};\ \texttt{k++}\ \}$

$\{\ \texttt{s} = \texttt{n} * (\texttt{n}-\texttt{1})/\texttt{2}\ \}$

Answer:

We can use this invariant:

$$\texttt{I}:\quad \texttt{s} = \texttt{k} * (\texttt{k}-\texttt{1})/\texttt{2}\ \ \wedge\ \ \texttt{0<k}\leq\texttt{n}$$

- Upon termination (so, $\texttt{k} \geq \texttt{n}$) it will implies that $\texttt{k} = \texttt{n}$, and thus the required post-condition.
- The initialization code before the loop can also setup this `I`.
- It is re-established by each iteration. That is, we need to show:

  $\{\ \texttt{I} \wedge \texttt{k<n}\ \}$ loop's body $\{\ \texttt{I}\ \}$

  If you calculate the pre-condition of:

  $\{\ ?\ \}$ loop's body $\{\ \texttt{I}\ \}$

  using our inference rules, you will come up with this:

  $(\texttt{s}+\texttt{k}\ =\ (\texttt{k}+\texttt{1}) * \texttt{k}/\texttt{2})\ \ \wedge\ \ \texttt{0<k+1}\leq\texttt{n}$

  Now we need to show that the above is implied by $\texttt{I} \wedge \texttt{k<n}$. The second conjunct is trivial.
  For the first conjunct, we will show it below:

  $(\texttt{k}+\texttt{1}) * \texttt{k}/\texttt{2}$
  $=$
  $((\texttt{k}-\texttt{1})+\texttt{2}) * \texttt{k}/\texttt{2}$
  $=$
  $(\texttt{k} * (\texttt{k}-\texttt{1})/\texttt{2})\ +\ \texttt{k}$
  $=$ // `I` was assumed, which says that $\texttt{k} * (\texttt{k}-\texttt{1})/\texttt{2} = \texttt{s}$
  $\texttt{s}+\texttt{k}$

  (Done)

7. Let's take a look at a bit bigger program now. *Bubble sort* is an in-situ[1] algorithm to sort an array.

   Its worst case run time is linear to $\mathtt{n}^2$, where `n` is the size of its input array, compared to `n` $*$ `log n` of some other algoritms. So, it is not the fastest algorithm, but it will do for our example here.

   Suppose a is our input array. The idea is to grow a section of a which is already sorted. We'll keep this section in the left part of a, and let it grow to the right.

   At each iteration we keep swapping elements in a's right-section (the still unsorted section) until the smallest element x of this right-section is moved to the most left part of this right-section. Then we can simply add this x to our left-section, and thus growing it.

   If you think the value of each element as the element weight, the 'swapping'-phase can be seen as a phase to 'bubble' the lightest element to the surface. Hence the name 'bubble sort'.

   Below is an implementation of the bubble sort algorithm in uPL. How to proceed now if we want to convince our customers of its correctness?

```
BUBBLESORT (a:[]int, n:int) : ()
{ var i,j:int ;
  i:=0 ;
  while i<n do
    { j:=n-1 ;
      while i<j do
         { j:=j-1
           if a[j+1]<a[j]
              then { tmp := a[j]    ;
                     a[j] := a[j+1] ;
                     a[j+1] := tmp  }
              else skip
         } ;
      i:=i+1 } }
```

   Answer:

   You need to come-up with invariants of the loops. If termination is not obvious, also come up with the needed termination metrics. In this case termination is quite obvious, so I'll skip this part.

   You need to argue as the Loop Rule tells you: (1) argue that when each loop terminates its invariant implies the loop's post-condition, (2) argue that the invariant can indeed be setup when the loop is started, and (3) argue that the invariant can be re-established by your iterations.

   Let us introduce the following abbreviation:

$$\mathsf{sorted\ a\ i} \;=\; (\forall \mathsf{p},\mathsf{q} :\; 0 \leq \mathsf{p},\mathsf{q} < \mathsf{i} : \mathsf{p} \leq \mathsf{q} \;\Rightarrow\; \mathsf{a}[\mathsf{p}] \leq \mathsf{a}[\mathsf{q}])$$

   Below I have annotated the needed invariants of both the outer and inner loops:

[1] 'In-situ' means that the algorithm directly manipulates its target data structure (in this case an array). So it is more space efficient, at the expense of being destructive.

```
{ 0≤n }  // pre-condition

 i:=0 ;

// Inv outer loop:
{ sorted a i  ∧  0≤i≤n }

 while i<n do
   { j:=n-1 ;

      // Inv inner loop:
      { sorted a i  ∧  0≤i<n  ∧  i≤j<n  ∧  a[j] = MIN(a[j...n)) }

     while i<j do
        { j:=j-1
          if a[j+1]<a[j]
             then { tmp := a[j]    ;
                    a[j] := a[j+1] ;
                    a[j+1] := tmp  }
             else skip
        } ;

       // Post-cond. of inner loop:
       { sorted a i  ∧  0≤i<n  ∧  a[i] = MIN(a[i...n)) }

     i:=i+1 }

{ sorted a n }  // Post-condition
```

8. Here is a uPL program to do a binary search in an array. The parameter `n` is assumed to be the size of the array `a`, and the elements `a` are stored in `a[0]...a[n − 1]`. The program returns the index in `a` can be found, if it is in the array. Else -1 is returned. Note that the program requires `a` to be *ascendingly sorted.*

```
binarySearch(n:int, a:int[], x:int) : int {

    int low,high  ;

    low  := 0 ;
    high := n ;
    while low+1<high ∧ a[low] ≠ x {
       // find a middle-point mid between low and high:
       mid := (low + high) div 2 ;
       if (a[mid] > x) then high := mid
                       else low  := mid ;
       } ;

   // Now either x is found or there is no more element
   // between low and high.

   return (a[low]=x  →  low | −1)
}
```

Write a specification for this program. How do you proceed now if we are to prove the its correctness? What is your loop invariant?

Answer:

For save some writing let me first introduce this abbreviation to express that an array `a` is sorted:

$$\texttt{sorted a n} \;=\; (\forall \texttt{i} : 0 \le \texttt{i} < \texttt{n}-1 : \texttt{a[i]} \le \texttt{a[i+1]})$$

And this abbreviation too, to express that a value `x` is in an array:

$$\texttt{x} \in \texttt{a[0..n)} \;=\; (\exists \texttt{k} : 0 \le \texttt{i} < \texttt{n} : \texttt{a[k]} = \texttt{x})$$

Now the requested specification:

$$\{\ \texttt{n}>0 \ \wedge\ \texttt{sorted a n}\ \}$$

$$\texttt{binarySearch(n : int, a : int[], x : int)}$$

$$\{\ \texttt{x} \in \texttt{a[0..n)} \ \rightarrow\ (\texttt{a[return]}=\texttt{x}) \mid (\texttt{return} = -1)\ \}$$

Use the following loop invariant *I*:

$$0 \le \texttt{low} < \texttt{high} \le \texttt{n}$$
$$\wedge$$
$$\texttt{x} \in \texttt{a[0..n)} \ = \ \texttt{x} \in \texttt{a[low..high)}$$

This invariant is obviously setup by the initialization code.

When the loop terminates because `a[low]` = `in`, then arguing the post-condition is not difficult.

Now, if the loop terminates because `low+1` = `high`, notice that the segment `a[low..high)` will therefore be empty. So, `bx` ∈ `a[low..high)` cannot be true. It follows, according to the above invariant, that therefore `x` is also not in `a[0..n)`, in which case the program correctly return -1.

I leave it to you to argue that each iteratition of the loop does indeed re-establish this I.

9. Write a simple class `SortedArray` that maintains a list of sorted Integers, equiped with the following methods:

   (a) `add` to add a new element into your array. Enlarge the size of your array when necessary.

   (b) `get` to retrieve the greatest element from the array (this element will also be removed from the array).

   (c) `max` to return the greatest element.

   (d) `contains` to check if an Integer is in the array.

   Use Esc/Java to debug your class. Also lear to use the tool to enhance your class with minimalistic specifications to get rid of its false positive warnings (that is, warnings that are actually not errors but more because you didn't tell the tool of some hidden assumption you had about your class, e.g. that you won't pass a null to a certain method).

   Next, you can try to add more complete specifications to your class. You can't really rely on Esc/Java to verify your specifications (the logic of Esc/Java is incomplete), but you now will at least have formal specifications.

   Can you suggest how to proceed now? Is there anything else we can do to improve the trustworthiness of your class?