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.

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

   (c) A program that sorts an array a.

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 { `true` }

   (b) { P } S { `false` }

   (c) { `false` } S { Q }

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) { ? } `x++; s:=s+x` { `s>0` }

   (b) { ? } `yold:=y; if i = 0 then y:=0 else y := y/x` { `y<yold` }

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:

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

   We can now define `assume` as follows:

   $$\texttt{assume}\ P \quad = \quad \texttt{if}\ P\ \texttt{then skip else miracle false}$$

   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.

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)` $\}$

   (b) $\{ ? \}$ `x := y/x` $\{$ `x < y` $\}$

   (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`.

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

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:

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

   (b) The following in partial correctness:

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

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

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

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 } }
```

[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.

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?

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?