Reasoning About Programs

Week 9 PMT - Binary Search
To discuss during PMT - do NOT hand in

Sophia Drossopoulou and Mark Wheelhouse

1st Question:

Assume that `search` has the same specification as in the lecture slides, that is:

```
int search(char[] a, char x)
// PRE: a ≠ null  ∧  Sorted(a)
// POST: a₀[0..r) < x ≤ a₀[r..a₀.length)
{ ... }
```

Which of the following statements hold for sorted character arrays `a` and characters `x`?
Justify your answers: prove the true statements and give counterexamples for the false ones.

i) `search(a,x)` returns the index of the first occurrence of `x` in `a`.

ii) if `x` is in `a` then `search(a,x)` returns the index of the first occurrence of `x` in `a`.

A possible answer:

i) **Does not hold**

For example, consider the case where $\texttt{a}[0..\texttt{a.length}) < \texttt{x}$.
i.e. every element of `a` is smaller than `x`.

Then $\texttt{search(a,x)} = \texttt{a.length}$ satisfies the specification,
but `a.length` is not an index of `a`.

ii) **Does not hold**

For a counterexample, consider an array `b = ['d','e','n','t']`.
Then, the first occurrence of `'d'` in `b` is at index 0.

On the other hand $\texttt{b}[0..-2) < \texttt{'d'} \leq \texttt{b}[-2..4)$.

Therefore, $\texttt{search(b,'d')} = -2$ satisfies the specification.
(In fact, any value smaller than 1 satisfies the specification.)

2nd Question:

Assume that `search` now has the following specification:

```
int search(char[] a, char x)
// PRE: a ≠ null  ∧  Sorted(a)
// POST: a₀[0..r) < x ≤ a₀[r..a₀.length)  ∧  r ∈ [0..a₀.length]
{ ... }
```

Which of the following statements hold for sorted character arrays `a` and characters `x` and `y`? Justify your answers: prove the true statements and give counterexamples for the false ones.

i) `search(a,x)` returns the index of the first occurrence of `x` in `a`

ii) if `x` is in `a` then `search(a,x)` returns the index of the first occurrence of `x` in `a`

iii) if all entries in `a` are equal, then `search(a,x)` is either 0 or `a.length`

iv) if all entries in `a` are different, then `search(a,x)` returns a different value for each `x` (i.e. if $\mathtt{x} \neq \mathtt{y}$ then $\mathtt{search(a,x)} \neq \mathtt{search(a,y)}$)

v) if $\mathtt{x} < \mathtt{y}$ then $\mathtt{search(a,x)} < \mathtt{search(a,y)}$

vi) if $\mathtt{x} \leq \mathtt{y}$ then $\mathtt{search(a,x)} \leq \mathtt{search(a,y)}$

A possible answer:

...

3rd Question:

The function `d(a,x)` is defined as follows:

$$\mathtt{d}(\mathtt{a},\mathtt{x}) \quad = \quad \mathtt{search}(\mathtt{a},\mathtt{x}+1) - \mathtt{search}(\mathtt{a},\mathtt{x})$$

What is the significance of `d(a,x)` for an arbitrary ordered, non-empty array `a` and integer `x`? Justify your answer.

A possible answer:

...

4th Question:

Consider the specification of `search` as given in Question 1 and the following invariant:

$$\mathtt{a} \approx \mathtt{a}_0 \;\wedge\; 0 \;??\; \mathtt{left} \;??\; \mathtt{right} \;??\; \mathtt{a.length} \;\wedge\; \mathtt{a}[0..\mathtt{left}) < \mathtt{x} \leq \mathtt{a}(\mathtt{right}..\mathtt{a.length})$$

Note that this invariant differs from the one chosen in the lecture slides.

The invariant is not fully specified yet, as we have not set exact bounds for `left` and `right`. Therefore, the code will have the following outline, where `t?` indicates some term which has not yet been determined.

```
int search(char[] a, char x)
// PRE: a ≠ null  ∧  Sorted(a)
// POST: a_0[0..r) < x ≤ a_0[r..a_0.length)
{
  ...
  // INV: a ≈ a_0  ∧  0 ?? left ?? right ?? a.length
  //          ∧  a[0..left) < x ≤ a(right..a.length)
  // VAR: ???
  while( ??? ) {
    int middle = (left + right) / 2;
    ...
  }
  // MID: a ≈ a_0  ∧  a[0..t?) < x ≤ a(t?..a.length)
  return t?;
}
```

In the following, we shall refine the invariant and systematically develop the code:

i) **From invariant to loop condition**: Find the loop condition, the term `t?`, and refine the unspecified parts of the invariant.

ii) **From invariant to loop code**: Write some code for the loop's body that uses `middle`, preserves the invariant when the loop condition holds, and which also "makes progress".

iii) **From invariant to initialization**: Write initialization code which establishes the invariant.

iv) **Array Accesses**: Informally check that any array accesses in the code will be valid.

v) **Termination**: Find a variant which decreases with *every* loop iteration and give its lower bound.

vi) **Putting it all together**: Write out the full code and give in comments all invariants, mid-conditions and variants.

vii) **Prove that the code satisfies its specification**:

  a) Prove that the initialization code establishes the invariant.

  b) Prove that the loop body preserves the invariant.

  c) Prove that the midcondition holds immediately on termination of the loop.

  d) Prove that the variant is bounded and decreases on *every* loop iteration.

  e) Prove that all array access in the method are valid.

A possible answer:

(see secondary answer-sheet)