15-122: Principles of Imperative Computation

Recitation 8 Solutions

Elias Szabo-Wexler, Nivedita Chopra

Consider the code for quicksort, with the partition step implemented as follows.

```
#use "sortutil.c0"
#use <rand>

int partition(int[] A, int lower, int pivot_index, int upper)
//@requires 0 <= lower && lower <= pivot_index;
//@requires pivot_index < upper && upper <= \length(A);
//@ensures lower <= \result && \result < upper;
//@ensures gt_seg(A[\result], A, lower, \result);
//@ensures le_seg(A[\result], A, \result, upper);
{
  // Hold the pivot element off to the left at "lower"
  int pivot = A[pivot_index];
  swap(A, lower, pivot_index);

  int left = lower+1; // Inclusive lower bound (lower+1, pivot's at lower)
  int right = upper; // Exclusive upper bound

  while (left < right)
    //@loop_invariant lower+1 <= left && left <= right && right <= upper;
    //@loop_invariant gt_seg(pivot, A, lower+1, left); // Not lower!
    //@loop_invariant le_seg(pivot, A, right, upper);
    {
      if (A[left] < pivot) {
        left++;
      } else {
        //@assert A[left] >= pivot;
        swap(A, left, right-1); // right-1 because of exclusive upper bound
        right--;
      }
    }
  //@assert left == right;

  swap(A, lower, left-1);
  return left-1;
}

void sort(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{
  if (upper - lower <= 1) return;
  int pivot_index = lower + rand(init_rand(234234)) % (upper - lower); // Pivot at midpoint

  int new_pivot_index = partition(A, lower, pivot_index, upper);
  sort(A, lower, new_pivot_index);
  //@assert is_sorted(A, lower, new_pivot_index + 1);
  sort(A, new_pivot_index + 1, upper);
}
```

Checkpoint 0

What would go wrong if the `partition` function ignored `pivot_index` and picked a new pivot?

The pivot indices can be thought of as the anchor points for the algorithm – we sort around the chosen pivots. If the partition stop ignored the pivots selected by the sort function, then sort would continue operating with incorrect assumptions about what, exactly, was sorted! In the end, the array would not necessarily be sorted (and, in fact, some contracts might fail!)

Checkpoint 1

Why is swapping the pivot with `left-1` the right thing? Why is `left` wrong? Why is `left-1` safe?

The loop invariants imply that that whatever is located *at* `left` is actually greater than or equal to the pivot (since `left == right` by line 28 and the invariant on line 18 holds). If we swap there, we might unsort the left side.
We know `left-1` is safe from the loop invariant on line 16 – `left` is at least `lower+1`

Checkpoint 2

Prove that the that `partition` function is correct.

We seek to prove the correctness of the partition step, as follows.

1. **Loop invariants hold initially**

   (1) `lower+1 <= left && left <= right && right <= upper`

$$
\begin{aligned}
\texttt{left} &= \texttt{lower+1} && \text{(Line 12)} \\
\Rightarrow \texttt{lower+1} &<= \texttt{left} &&
\end{aligned}
$$

$$
\begin{aligned}
\texttt{lower} &<= \texttt{pivot\_index} && \text{(Line 2)} \\
\texttt{pivot\_index} &< \texttt{upper} && \text{(Line 3)} \\
\Rightarrow \texttt{lower} &< \texttt{upper} && \text{(Transitivity)} \\
\Rightarrow \texttt{lower+1} &<= \texttt{upper} && \text{(Addition)} \\
\Rightarrow \texttt{left} &<= \texttt{right} && \text{(Substitution; Line 12, Line 13)}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{right} &= \texttt{upper} && \text{(Line 13)} \\
\Rightarrow \texttt{right} &<= \texttt{upper} &&
\end{aligned}
$$

(2) `ge_seg(pivot, A, lower+1, left)`

`ge_seg(pivot, A, left, left)`	(Trivially true)
⇒ `ge_seg(pivot, A, lower+1, left)`	(Line 12)

(3) `le_seg(pivot, A, right, upper)`

`le_seg(pivot, A, right, right)`	(Trivially true)
⇒ `le_seg(pivot, A, right, upper)`	(Line 13)

2. **Loop invariants are preserved**

Case I: `A[left] <= pivot`

(1) `lower+1 <= left && left <= right && right <= upper`

`lower+1 <= left && left <= right && right <= upper`	(line 16)
`left' = left + 1`	(line 21)
`lower+1 <= left`	(line 16)
⇒ `lower+1 <= left'`	(Preceding facts)
`left < right`	(line 15)
`right' = right`	(no assignment)
⇒ `left' <= right'`	(Preceding facts)
⇒ `right' <= upper`	(Preceding facts)

(2) `ge_seg(pivot, A, lower+1, left)`

`ge_seg(pivot, A, lower+1, left)`	(line 17)
`A[left] <= pivot`	(line 20)
`left' = left+1`	(line 21)
⇒ `ge_seg(pivot, A, lower+1, left')`	(Preceding facts)

(3) `le_seg(pivot, A, right, upper)`

`le_seg(pivot, A, right, upper)`	(line 18)
`right' = right`	(no assignment)
⇒ `le_seg(pivot, A, right', upper)`	(Preceding facts)

Case II: `A[left] > pivot`

(1) `lower+1 <= left && left <= right && right <= upper`

`lower+1 <= left && left <= right && right <= upper`	(Line 16)
`left'` = `left`	(no assignment)
⇒ `lower+1` <= `left'`	(Preceding facts)
`right'` = `right - 1`	(line 25)
⇒ `right'` <= `upper`	(Preceding facts)
`left` < `right`	(line 15)
⇒ `left'` <= `right'`	(Preceding facts)

(2) `ge_seg(pivot, A, lower+1, left)`

`ge_seg(pivot, A, lower+1, left)`	(line 17)
`left'` = `left`	(no assignment)
⇒ `ge_seg(pivot, A, lower+1, left')`	(Preceding facts)

(3) `le_seg(pivot, A, right, upper)`

`le_seg(pivot, A, right, upper)`	(Line 18)
`right'` = `right-1`	(Line 25)
`A[left]` > `pivot`	(Line 23)
`A[left]` $\underset{\text{swap}}{\Leftrightarrow}$ `A[right']`	(Line 24)
⇒ `le_seg(pivot, A, right', upper)`	(Preceding facts)

3. **Negation of loop guard and loop invariants imply postcondition**

(1) `lower <= \result && \result < upper`

`left` $>=$ `right`	(Line 15; Negation)
`left` $<=$ `right`	(Line 16)
$\Rightarrow$ `left` $=$ `right`	
`lower+1` $<=$ `left`	(Line 16)
`\result` $=$ `left-1`	(Line 31)
$\Rightarrow$ `lower` $<=$ `\result`	
`right` $<=$ `upper`	(Line 16)
`\result` $=$ `right-1`	(Substitution)
$\Rightarrow$ `\result` $<$ `upper`	

(2) `ge_seg(A[\result], A, lower, \result)`

`ge_seg(pivot, A, lower+1, left)`	(Line 17)
`A[left-1]` $=$ `pivot`	(Line 30)
`A[\result]` $=$ `pivot`	(Line 31)
$\Rightarrow$ `ge_seg(A[\result], A, lower, \result)`	

(3) `le_seg(A[\result], A, \result, upper)`

`le_seg(pivot, A, right, upper)`	(Line 18)
`A[left-1]` $=$ `pivot`	(Line 30)
`A[\result]` $=$ `pivot`	Line 31)
$\Rightarrow$ `le_seg(A[\result], A, \result, upper)`	

4. **Loop terminates**

$$\texttt{left} < \texttt{right} \qquad \text{(line 15)}$$

Case I:

$$\texttt{left'} = \texttt{left+1} \qquad \text{(line 21)}$$
$$\texttt{right'} = \texttt{right} \qquad \text{(no assignment)}$$

Case II:

$$\texttt{left'} = \texttt{left} \qquad \text{(no assignment)}$$
$$\texttt{right'} = \texttt{right-1} \qquad \text{(line 25)}$$

Observe that, regardless of case,

$$\Rightarrow \texttt{right' - left' < right - left} \qquad \text{(substitution)}$$

Thus, `right - left` is the quantity that is getting smaller with every iteration of the loop. Since `right, left > 0` (line 2, 12, 13), we see that eventually `right'-left' = 0`. Therefore the loop *must* terminate.

Checkpoint 3

Could you change `partition` (code and/or loop invariants) in order to justify one of the postconditions on line 5 or 6 being `gt_seg` or `lt_seg`?

Yes. We can actually change the left side, such that all elements are strictly less than the pivot. This requires a minor modification to the sorting comparison. We therefore need to modify the ensures condition on line 6, the loop invariant on line 18, and the conditional on line 2. Consider the code below:

```
1 #use "sortutil.c0"
2
3 int partition(int[] A, int lower, int pivot_index, int upper)
4 //@requires 0 <= lower && lower <= pivot_index;
5 //@requires pivot_index < upper && upper <= \length(A);
6 //@ensures lower <= \result && \result < upper;
7 //@ensures ge_seg(A[\result], A, lower, \result);
8 //@ensures lt_seg(A[\result], A, \result, upper);
9 {
10   // Hold the pivot element off to the left at "lower"
11   int pivot = A[pivot_index];
12   swap(A, lower, pivot_index);
13
14   int left = lower+1; // Inclusive lower bound (lower+1, pivot's at lower)
15   int right = upper; // Exclusive upper bound
16
```

```
  while (left < right)
    //@loop_invariant lower+1 <= left && left <= right && right <= upper;
    //@loop_invariant ge_seg(pivot, A, lower+1, left); // Not lower!
    //@loop_invariant lt_seg(pivot, A, right, upper);
    {
      if (A[left] < pivot) {
        left++;
      } else {
        //@assert A[left] >= pivot;
        swap(A, left, right-1); // right-1 because of exclusive upper bound
        right--;
      }
    }
  //@assert left == right;

  swap(A, lower, left-1);
  return left-1;
}

void sort(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{
  if (upper - lower <= 1) return;
  int pivot_index = lower + (upper - lower)/2; // Pivot at midpoint

  int new_pivot_index = partition(A, lower, pivot_index, upper);
  sort(A, lower, new_pivot_index);
  //@assert is_sorted(A, lower, new_pivot_index + 1);
  sort(A, new_pivot_index + 1, upper);
}
```

Checkpoint 4

Using the `rand` library in C0, modify this code to select a random pivot.

Only the `sort` function is to be modified here, in order to choose a random pivot.

```
#use "sortutil.c0"

void sort(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{
  if (upper - lower <= 1) return;
  rand_t gen = init_rand(15122);
  int pivot_index = lower + (rand(gen) % (upper-lower-1)) // somewhere in [lower, upper-1]

  int new_pivot_index = partition(A, lower, pivot_index, upper);
  sort(A, lower, new_pivot_index);
  //@assert is_sorted(A, lower, new_pivot_index + 1);
  sort(A, new_pivot_index + 1, upper);
}
```