Handin I4

Jens Kristian Refsgaard Nielsen
201303862

9. maj 2019

1 Merge two arrays

We are to fix the provided Merge method in dafny. We add two requires statements, the first to ensure that there is a non trivial task ahead of us. The second secures that there is exactly enough space in c to merge a and b into it.

```
method merge(a:array<int>,b:array<int>,c:array<int>)
modifies c // declares that merge writes in array c
requires a.Length >0 && b.Length >0 && c.Length >0
requires a.Length+b.Length == c.Length
{ var i:int := 0;
  var j:int := 0;
  var k:int := 0;
  while (k<c.Length)
  invariant i+j==k
  { if (i>=a.Length || (j<b.Length && b[j]<=a[i]))
    { c[k] := b[j];
      j := j+1;
    }
    else
    { c[k] := a[i];
      i := i+1;
    }
    k := k+1;
} }
```

2 Merge two sorted arrays

2.1 Question 2

We wish to define a function that checks whether the input array is sorted or not, and returns a boolean. This is equivivalent to making a predicate, because these are simply functions that always return a boolean.

```
function sorted(a:array<int>):bool
reads a
{
forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}
predicate sorted(a:array<int>)
reads a
{
forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}
```

The quantifiers used is at glance a stronger predicate than the standard notion of being sorted $a[i] <= a[i+1]$ for relevant i, however let j, k be given such that $0 <= j < k < a.Length$ then by definition of $<$ we have $m > 0$ such that $j + m = k$ and then by repeted application we get $a[j] <= a[j+1] <= \cdots <= a[k-1] <= a[k]$ and by transitivity $a[j] <= a[k]$. So the predicate is a good way to define sortedness of arrays.

2.2 Question 3

We wish to use dafny to prove that the result of our merge method is sorted. We add the ensures keyword that tells dafny to check whether the following predicate is true after running the method.

```
method merged(a:array<int>,b:array<int>,c:array<int>)
modifies c // declares that merge writes in array c
requires sorted(a)
requires sorted(b)
ensures sorted(c)
requires a.Length >0 && b.Length >0 && c.Length >0
requires a.Length + b.Length == c.Length
{ var i:int := 0;
  var j:int := 0;
  var k:int := 0;
  while (k<c.Length)
  invariant i+j==k
  invariant 0<=k<=c.Length
  invariant forall g :: i<=g<a.Length&&k>0 ==> c[k-1]<=a[g]
  invariant forall g :: j<=g<b.Length&&k>0 ==> c[k-1]<=b[g]
  invariant forall g :: ((0<=g<k)) ==> c[g]<=c[k-1]
  invariant forall h,g :: 0<=h<g<k ==> c[h]<=c[g]
  { if (i>=a.Length || (j<b.Length && b[j]<=a[i]))
    { c[k] := b[j];
      j := j+1;
    }
    else
    { c[k] := a[i];
      i := i+1;
    }
    k := k+1;
} }
```

Regarding the invariants:

$i + j == k$: this ensures that every element is touched by the procedure as k and (i or j) is raised by 1 each iteration.

$0 <= k <= c.Length$: this is just an out of bounds checker.

forall g :: $i <= g < a.$Length&&$k > 0 ==> c[k-1] <= a[g]$: here we check that the remaining part of a is bigger than the element we just inserted.

forall g :: $i <= g < a.$Length&&$k > 0 ==> c[k-1] <= b[g]$: here we check that the remaining part of b is bigger than the element we just inserted.

forall g :: $((0 <= g < k)) ==> c[g] <= c[k-1]$: this is the crucial part where we check that all previous elements are smaller than the one we just inserted.

forall h, g :: $0 <= h < g < k ==> c[h] <= c[g]$: it follows immediately that the elements we have inserted in c are sorted.