2IMP10 Program Verification Techniques
Assignment 2

Rick van Oosterhout, 1450131, r.h.m.v.oosterhout@student.tue.nl

Quint Bakens, 1454315, q.bakens@student.tue.nl

Dafny

For this assignment we were required to verify programs using Dafny. We did this by using the Dafny extension for Visual Studio Code. We installed this using the extension browser within Visual Studio Code, which provided us with Dafny version 3.0.3. Verification was done automatically within Visual Studio Code, which runs Dafny on .dfy files after changes have been made.

Prime Assignment

To write a method that checks whether a number is prime, we first want to specify the postcondition that should hold for this function. In this case, this was already specified by the `prime?(n: nat)` predicate. All we had to do was uncomment the postcondition for the method itself, which was now specified as:

```
method CheckPrime(n: nat) returns (p: bool)
    ensures p <==> prime?(n)
```

Next, we implemented the body of the method. The easiest way to check whether n is prime is to go over all values i where $1 < i < n$ and check whether any of them divides n. So we first make sure that i starts from 2. After which we check for each $i < n$ if it divides n, if it does then n is not prime, if no i divides n, it is prime.

```
1    var i: nat;
2
3    if (n <= 1) {
4      return false;
5    }
6
7    i := 2;
8    while (i < n)
9      // explicit termination
10     decreases n - i
11     // head invariant, every checked value (d) til now does not
       ↪  divide n
12     invariant (forall d | 1 < d < i :: n % d != 0)
13   {
14     if (n % i == 0) {
15       return false;
16     }
17     i := i + 1;
18   }
19
20   return true;
```

To make sure that Dafny verifies successfully, we need to specify that no value d that we have checked so far ($1 < d < i$) is a divisor of n. This is done using the invariant on line 12. Furthermore we also included the `decreases` statement, however Dafny also obtains this one implicitly.

The postcondition holds for all paths, because the precondition, case distinction and loop invariant together cover all the cases of the method. For which they ensure that `p <==> prime?(n)`.

SumPositives Assignment

For this exercise, we wrote an inductive function *sumpos*, we tested it using assert statements at the end of the code. Usuing this function, we created a method *SumPositives* that calculates the sum of positive numbers in an int sequence passed as a paramter, equivalent to *sumpos*.

```
function sumpos(xs: seq<int>): int
{
  if |xs| == 0 then 0
  else if xs[0] > 0 then xs[0] + sumpos(xs[1..])
  else sumpos(xs[1..])
}

method SumPositives(xs: seq<int>) returns (s: int)
  ensures s == sumpos(xs)
{
  s := 0;
  if |xs| == 0 {
    return s;
  }

  var i: int;
  for i := 0 to |xs|
    invariant s + sumpos(xs[i..]) == sumpos(xs)
  {
    if (xs[i] > 0) {
      s := s + xs[i];
    }
  }

  return s;
}

method SpecTest() {
  assert sumpos([]) == 0;
  assert sumpos([-1]) == 0;
  assert sumpos([0]) == 0;
  assert sumpos([1]) == 1;
  assert sumpos([0,1,2,3]) == 6;
  assert sumpos([0,-1,2,-3]) == 2;
  assert sumpos([-40,10,0,0,-3,20,-3]) == 30;
}
```

The function $sumpos(xs) : seq < int >$. There are 3 cases:

- Either the size of the array xs is 0, return 0 in that case;
- Or if the value at $xs[0] > 0$ then return $xs[0] + sumpos(xs[1..]$;
- Or the value at $xs[0] \leq 0$, then simply return $sumpos(xs[1..])$.

We tested the induction function using the method $SpecTest$ in which several cases were tested to see if the function correctly computes the sum of positive integers in the array. To this end, several assert statements with inputs for the *sumpos* function were provided, which did indeed all verify, as expected.

Lastly, the method $SumPositives$ was created which also takes an array $xs : seq < int >$ as input. The postcondition of this method is simply the function *sumpos*. The method $SumPositives$ starts by creating variable s, which is set to 0, and checking if the size of the input sequence xs is equal to 0, in which case it will return 0. Afterwards, the method iterates over all values in the sequence, and if $xs[i] > 0$, with i being the index, then we do $s := s + xs[i]$.

In the for-loop we used, an invariant was added to let Dafny correctly verify the program. This invariant `s + sumpos(xs[i..])  == sumpos(xs)`. This states that at the start of every iteration in the loop, the value of the up untill now found sum summed with the *sumpos* value of the remaining sequence is equal to the *sumpos* value of the entire sequence.

As Dafny correctly verifies the program, we have proven that our implementation of the method $SumPositives$ is correct.

QuadraticSort Assignment

The full QuadraticSort algorithm, along with its verification is provided below.

```
method {:verify true} QuadraticSort(a: array<int>)
   modifies a;
   ensures Sorted(a, 0, a.Length);
   ensures multiset(a[..]) == old(multiset(a[..]));
{
   if (a.Length == 0) {return;}
   var i := 1;
   while (i < a.Length)
      decreases a.Length - i
      invariant 0 < i <= a.Length
      invariant Sorted(a, 0, i)
      invariant multiset(a[..]) == old(multiset(a[..]))
   {
      var j := i-1;
      while (0 <= j && a[j] > a[j+1])
         decreases j
         invariant Sorted(a, 0, j+1)
         invariant 0 < j+1 < i ==> a[j] <= a[j + 2]
         invariant Sorted(a, j+1, i+1)
         invariant multiset(a[..]) == old(multiset(a[..]))
      {
         a[j], a[j+1] := a[j+1], a[j];
         j := j - 1;
      }
      i := i + 1;
   }
}
```

In order to verify the entire algorithm, we had to take care of 2 postconditions.

1. `Sorted(a, 0, a.Length)`
2. `multiset(a[..]) == old(multiset(a[..])`

We can simply solve the first postcondition by pushing it down to the while loops as an invariant. This is logically equivalent, as we only swap elements, thus the multiset of the array a contains the same values.

For the second postcondition, regarding the sortedness of the array a, a more complex solution was necessary. First, we know that after each iteration of the outer while loop, the array is sorted upto and not including the index i. Thus this is our invariant in the outer while loop. To ensure that the precondition of the aforementioned invariant holds, we must also ensure that the index i is in a valid range. To this end, we also add the invariant `0 < i <= a.Length` to the outer while loop.

These invariants pushed down the issue to the inner while loop. In the inner while loop, elements with indices j and $j+1$ get swapped, but only if these are not yet sorted. In each iteration of this inner loop, the array is guaranteed to be sorted up to and not including index $j+1$. This is true, as we start this inner loop with a sorted array up to index i and $j = i - 1$. After each iteration, the values at indices j and $j+1$ are swapped if these are not sorted. Logically at this point it is true that the array is sorted up to and including index j. Besides, at all times during the inner while loop, except for the first iteration, we know that $a[j] \leq a[j+2]$, as before the swap these used to be sorted. Finally, as we know that when entering the inner loop, the array is sorted upto and not including index i, after swapping two arbitrary values at indices j and $j+1$, we still know that the array from index $j+1$ is sorted upto and not including index $i+1$.

Adding these 3 invariants to the inner while loop, finished the verification by Dafny.

SlopeSearch Assignment

To implement `SlopeSearch` as specified in the assignment. We need to make sure that our implementation (and specification) match with the description of the algorithm that is presented in the book.

Firstly, we make sure that we implement the matching precondition. The precondition of the algorithm that is presented in the book consists of the following properties:

Let M and N be natural numbers and let array $f : [0..M] \times [0..N] \to Z$ be ascending in both arguments, i.e.,

$$(\forall i : 0 \leq i \leq M : (\forall j : 0 \leq j < N : f.i.j \leq f.i.(j+1)))$$
$$\wedge(\forall j : 0 \leq i \leq N : (i : 0 \leq i < M : f.i.j \leq f.(i+1).j))$$

Assume that a value X occurs in f,i.e.,

$$(\exists i, j : 0 \leq i \leq M \wedge 0 \leq i \leq N : f.i.j = X)$$

We implement this as a precondition for the method in Dafny using a predicate in our requires statement.

```
predicate precon(f: seq<seq<nat>>, X: int)
{
  // F is non-empty and rectangular
  |f| > 0 && (forall i | 0 <= i < |f| :: |f[0]| == |f[i]| > 0)

  // All neighbors are increasing
  && (forall i | 0 <= i < |f| :: (forall j | 0 <= j < |f[i]|-1
  ↪  :: f[i][j] <= f[i][j+1]))
  && (forall i | 0 <= i < |f|-1 :: (forall j | 0 <= j < |f[i]|
  ↪  :: f[i][j] <= f[i+1][j]))

  // Cells below (>i) and/or to the right (>j) are increasing
  // Equivalent to the properties above due to transitivity
  && (forall i,j | 0 <= i < |f| && 0 <= j < |f[i]| :: (forall r
  ↪  | 0 <= r < i:: f[r][j] <= f[i][j]))
  && (forall i,j | 0 <= i < |f| && 0 <= j < |f[i]| :: (forall c
  ↪  | 0 <= c < j:: f[i][c] <= f[i][j]))

  // X exists within f
  && (exists i,j | 0 <= i < |f| && 0 <= j < |f[i]| :: f[i][j] ==
  ↪  X)
}
```

We need the first condition because we use a nested sequence data type for f. For which we also need the requirement that f is rectangular. So we make sure that all sub-sequences contain the same number of elements. The second set of conditions states that each element with an increment on either of the indices, is larger or equal than the previous element. This condition is followed by an equivalent condition, which uses the transitivity of $\leq$ to prove that any element $f[i'][j']$ with either $i' < i$ or $j' < j$ is smaller or equal than $f[i][j]$. Lastly, we have the condition that there exists an element with $f[i][j] = X$, meaning that X occurs at least once in f. We add this predicate to the method using the requires statement. Such that inputs must satisfy the predicate to be valid inputs with regards to the precondition.

Besides the precondition, we need to implement the postcondition that is specified by the book. This postcondition states:

We are asked to derive a program that establishes for integer variables a and b

$$0 \leq a \leq M \wedge 0 \leq b \leq N \wedge f.a.b = X$$

We implement this postcondition in a similar way to our precondition, using a predicate that is used in the specification of our method. This `postcon` function verifies that the combination of inputs f, X and return values a and b, satisfy the postcondition given by the book.

```
predicate postcon(f: seq<seq<nat>>, X:int, a: nat, b: nat)
{
  0 <= a < |f|
  && 0 <= b < |f[a]|
  && f[a][b] == X
}
```

After we implemented the specification of the pre- and postcondition, we created the implementation of the algorithm. This implementation is a direct copy of the pseudocode presented in the book. Where the search is started at $(0, N)$ and each time we either increment a or decrement b. To make sure that dadny is able to verify the method. We already implemented an equivalent precondition, which helps Dafny proving further steps. Besides the precondition, we also need to add a decreases statement and loop invariant. These are both also copied from the book's pseudocode directly. Presented as the bound and invariant, these are given on lines 8 and 9 in the following code block.

```
1  method SlopeSearch(f: seq<seq<nat>>, X: int) returns (a: nat, b: nat)
2    requires precon(f, X)
3    ensures postcon(f, X, a, b)
4  {
5    a, b := 0, |f[0]|-1;
6
7    while (true)
```

```
    decreases |f| - a + b
    invariant exists i,j | 0 <= i < |f| && 0 <= j < |f[i]| &&
    ↪  f[i][j] == X :: 0 <= a <= i && j <= b < |f[i]|
  {
    if f[a][b] < X {
      a := a + 1;
    } else if f[a][b] > X {
      b := b - 1;
    } else {
      break;
    }
  }

  return a,b;
}
```

HeapSort Assignment

Heapify

In order to create a heap from an input array using the bubble up algorithm, we iteratively check whether the element at *index*, starting from the second element, its parent is smaller. If this is the case, we swap the value of the parent of *index* and the value of at *index*. If a swap occurs, we recursively check whether the parents of the children at *index* are smaller than their children, and if so swap them as well. This in essence, is the bubbling down. The algorithm, with its verification code can be seen below.

```
// turn a into a heap by bubbling up
method {:verify true} Heapify(a: array<int>)
  modifies a;
  requires a.Length > 0;
  ensures heap(a, a.Length - 1);
  ensures multiset(a[..]) == multiset(old(a[..]));
{
  var index: int := 1;

  // Bubble all the elements up, starting from the second
  ↪  element
  while (index < a.Length)
    decreases a.Length - index;
    invariant multiset(a[..]) == multiset(old(a[..]));
    invariant index - 1 < a.Length;
    invariant heap(a, index - 1);
  {
    var updateIndex := index;

    // Recursively bubble the current index up if necessary
    while (updateIndex >= 0)
      decreases updateIndex;
      invariant multiset(a[..]) == multiset(old(a[..]));
      invariant 0 <= updateIndex < a.Length;
      invariant heapSpecial(a, index, updateIndex);

    {
      // Break from loop if we arrive at the first element
      if (updateIndex <= 0)
      {
        break;
      }

      // Swap the current updateIndex with its parent if it is
      ↪  smaller and recursively try again on the parent of
      ↪  updateIndex
      if (a[updateIndex] > a[parent(updateIndex)])
      {
        a[parent(updateIndex)], a[updateIndex] :=
         ↪  a[updateIndex], a[parent(updateIndex)];
        updateIndex := parent(updateIndex);
      }
      // Break if element is not smaller
      else
      {
        break;
      }
    }
    index := index + 1;
  }
}
```

To let Dafny verify the program, we iterate over all the elements of the array, starting from the second element, as the first element by itself already constitutes a valid heap. we added the following three invariants to the outer while loop:

- `invariant multiset(a[..])  == multiset(old(a[..]))`, this ensures that the input array contains equivalent values at every iteration of the outer while loop;
- `invariant index - 1 < a.Length`, this ensures that we can always index on the value of index on the array;
- `invariant heap(a, index - 1)`, also it is necessary that at all entries of the outer while loop we start with a heap up to the value of *index*, so that by adding a single element and correctly bubbling it up, we still end with a valid heap.

In the inner while loop, we first create a variable *updateIndex*, this keeps track of the index which we, possibly, need to update. As long as this value is greater than 0, it is possible that the value at index *updateIndex* needs to be swapped with its parent. This is not possible at index 0, thus we break in that case. If the value at index *updateIndex* is greater than the value of its parent, we swap it. and update the value of *updateIndex* to the parent's index. From this point we recursively bubble up the value. This stops, as mentioned before, when either the value of *updateIndex* reaches 0, or when no update is necessary in one iteration.

To verify the inner while loop, a new predicate was necessary. This new predicate is *heapSpecial* and its code is as follows.

```
predicate heapSpecial(a: array<int>, end: int, updateIndex: int)
  reads a;
  requires 0 <= end < a.Length;
  requires 0 <= updateIndex <= a.Length;
{
  (forall i | 0 < i <= end :: (i != updateIndex) ==>
  ↪  a[parent(i)] >= a[i])
  && (forall i :: 0 < i <= end && updateIndex > 0 && parent(i)
  ↪  == updateIndex ==> a[parent(parent(i))] >= a[i])
}
```

This predicate simply checks whether a heap is valid upto index *end*, when we do not factor in the value we are currently updating, being *updateIndex*.

This predicate is added as an invarint to the inner while loop. This now holds, because at the start of every iteration of this loop we have a valid heap upto index *index* and after, possibly, swapping the value at index *updateIndex*, the heap remains valid, except for the values at *parent*(*updateIndex*), *updateIndex* and *children*(*updateIndex*). As these are still being bubbled. We also added an invariant to ensure that at the start of every iteration *updateIndex* had a valid value, which is necessary for the *heapSpecial* predicate. This invariant is `0 <= updateIndex < a.Length`. Finally, the invariant `multiset(a[..]) == multiset(old(a[..]))` was added, to again ensure that the input array contains equivalent values at stages of the loop.

This in total verifies the Heapify method.

Heapsort

```
// sort a according to the heapsort algorithm
method {:verify true} HeapSort(a: array<int>)
  modifies a
  requires a.Length > 0
  ensures multiset(a[..]) == multiset(old(a[..]));
  ensures sorted(a, 0, a.Length - 1);
{
  Heapify(a);

  UnHeapify(a);
}
```

In order to sort the array a, we simply first ensure that a is a heap by building it using the `Heapify` algorithm created earlier. Afterwrads we can run `UnHeapify` to sort it. As `UnHeapify` is a method without a body, the postconditions of the method will be assumed to be correct. Thus the postcondition of `HeapSort` is also statisfied, as the postcondition of both methods are equal.

HeapMax

The lemma *HeapMax* was already provided, it was up to us to implement the body of the lemma.

```
// if a[..i+1] contains a heap, then a[0] is a maximal element
↪  of a[..i+1]
lemma {:verify true} HeapMax(a: array<int>, i: int)
  requires 0 <= i < a.Length
  requires heap(a, i)
  ensures firstmax(a, i)
{
  if (parent(i) >= 0) {
    assert a[parent(i)] >= a[i];
    HeapMax(a, i-1);
  } else {
    // No help needed
  }
```

Above is the code we created to prove the lemma. In order to ensure that the first element of the heap is indeed we use a dynamic programming strategy. As long as the parent of the maximal index provided i is greater than 0, then we know that i is not the root. In this case, the index i should be greater or equal to its parent. From this point we prove that the *HeapMax* lemma holds for the same input, except we decrement the index i. This leads to a proof by Dafny.

UnHeapify

we did not implement this function for the bonus point.