Assignment 2 CSSE3100/7100 Reasoning about Programs

Personal feedback ***Student's Name***

Question 1 [6 marks]

(a) Your friend (yes, the same one!) claims that the method below always terminates and returns X * Y, provided X and Y are non-negative. Prove that your friend is either right or wrong. To do so, complete the specification below and use weakest precondition reasoning to show whether or not the implementation satisfies the specification.

If the specification is not correct, explain the cases when it will fail to compute X * Y and provide a precondition for which it is correct.

```
method mult2(X: int, Y: int)  returns (r: int)
    requires ...
    ensures ...
{
    var x, y, c := X, Y, 0;
    while y > 1
        invariant …
        decreases …
    {
        if y % 2 == 0 {
            y := y/2;
        } else {
            y := (y-1)/2;
            c := c + x;
        }
        x := 2*x;
    }
    r := x + c;
}
```

(b) Encode the method (with corrected precondition, if necessary) along with your invariant and decreases clauses in Dafny. Unfortunately, Dafny doesn't know everything you do about integer and modulo division. Provide lemmas so that Dafny can verify the code. If Dafny can't prove your lemmas automatically, provide a proof by induction.

Feedback:

(a) Loop invariant (1 mark): *Comment*

Your mark: *1*

Termination metric (0.5 marks): *Comment*

Your mark: *0.5*

Weakest precondition proof (3 marks): *Comment*

Your mark: *3*

Dafny encoding and lemmas (1.5 marks): *Comment*

Your mark: *1.5*

A sample solution is shown below (0.5 marks was given for each line indicated with ← and 0.5 marks for stating the required precondition to make the method correct).

```
method mult2(X: int, Y: int) returns (r: int)
    requires X >= 0 && Y >= 0
    ensures r == X * Y
{
    {  Y >= 1  }
    {  X * Y  == X * Y && Y >= 1  }
    {  forall x, y, c :: X * Y  == X * Y && Y >= 1  }
    var x, y, c := X, Y, 0;
    {  x  * y + c == X * Y && y >= 1   }
    while y > 1
            invariant x * y + c == X * Y && y >= 1
            decreases y
    {
            {  x  * y + c == X * Y  && y >= 1 && y > 1  }
            {  x  * y + c == X * Y && (y % 2 == 0 ==> y > 1) && (y % 2 != 0 ==>  y > 1) }
            { x  * y + c == X * Y && (y % 2 == 0 ==> y >= 2) && (y % 2 != 0 ==> y >= 3) }
            { (y % 2 == 0 ==> x * y + c == X * Y && y >= 2 && true) &&
                    (y % 2 != 0 ==> x * y + c == X * Y && y >= 3 && true)
            { (y % 2 == 0 ==> x * y + c == X * Y && y >=2 && y > y/2) &&
                    (y % 2 != 0 ==> x * y + c == X * Y && y >= 3 && y > (y-1)/2)
            ghost var d := y;
            { (y % 2 == 0 ==> x * y + c == X * Y && y >=2 && d > y/2) &&
                    (y % 2 != 0 ==> x * y + c == X * Y && y >= 3 && d > (y-1)/2) }
            if y % 2 == 0 {
                    {  x * y + c == X * Y  && y >= 2 && d > y/2 }          ←
                    {  2 * x * y/2 + c == X * Y && y/2 >= 1 && d > y/2 }
                    y := y/2;
                    {  2 * x * y + c == X * Y && y >= 1 && d > y }
            } else {
                    {  x * y + c == X * Y && y >= 3 && d > (y-1)/2 }              ←
                    {  x * (y-1) + c + x == X * Y &&  y >= 3 && d > (y-1)/2 }
                    {  2 * x * (y-1)/2 + c + x == X * Y  && (y-1)/2 >= 1 && d > (y-1)/2 }
```

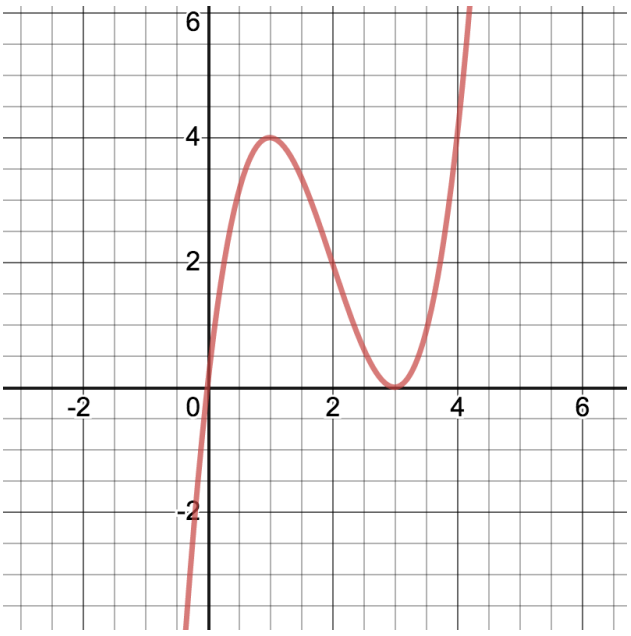
```
            y := (y-1)/2;
            { 2 * x * y + c + x == X * Y && y >= 1 && d > y }          ←
            c := c + x;
            { 2 * x * y + c == X * Y  && y >= 1 && d > y }

        }
        { 2 * x * y + c == X * Y && y >= 1 && d > y }                  ←
        x := 2*x;
        { x  * y + c == X * Y && y >= 1 && d > y }
    }
    { x * y + c == X * Y  && y >= 1 && y <= 1 }      ←
    { x + c == X * Y }
    r := x + c;
    { r == X * Y }
}
```

Note that y >= 1 is necessary in the invariant to make the postcondition hold when the guard is false. Hence, the weakest precondition is stronger than the stated precondition and the method is incorrect. It doesn't always result in X * Y. It only does so when Y >= 1. It also only terminates when Y >= 1.

(b) See a2q1.dfy. Both lemmas are proved automatically by Dafny. 1 mark was given for the lemma(s), and 0.5 marks for the placement of the lemma(s) in the code.

Question 2 [9 marks]

The graph below plots the equation $y = x^3 - 6x^2 + 9x$.



(a) Derive an iterative program which, for a given $N \geq 0$, computes the smallest integer x that satisfies $x^3 - 6x^2 + 9x \geq N$. Your program must satisfy the following requirements for efficiency:

1. No loop iterations for values which (given the graph above) cannot correspond to the return value.
2. No multiplication in loop iterations. You must use the "wishing" method from the lectures and textbook to achieve this.

Your derivation must be based on a pre/postcondition specification and use weakest precondition reasoning.

(b) Encode your method in Dafny to check your working. Use calc statements to check the predicate simplifications you made when deriving the loop body.

Feedback:

(a) Precondition (0.5 marks): *Comment*

Your mark: *0.5*

Postcondition (1 mark): *Comment*

Your mark: *1*

Loop invariant (1 mark): *Comment*

Your mark: *1*

Termination metric (0.5 marks): *Comment*

Your mark: *0.5*

Derivation (5 marks): *Comment*

Your mark: *5*

Dafny encoding (1 mark): *Comment*

Your mark: *1*

A sample solution is shown below (0.5 marks was given for each line indicated with ← and 0.5 for a correct initialisation of the variables in the loop frame).

Since I asked for a solution set out like Exercise 6.2 and that solution did not prove termination, I have not required you prove termination in this question.

```
function exp(x: int): int {

        x*x*x – 6*x*x + 9*x
}

method min(N: int) returns (r: int)
        requires  N >= 0
        ensures  (N == 0 ==> r == 0)  && (0 < N <= 4 ==> r == 1)
        ensures  N > 4 ==> exp(r) >= N && exp(r-1) < N
{
        {  N >= 0  }
        {  (N == 0 ==> N == 0) && (0 < N <= 4 ==> 0 < N <= 4) &&
           (!(0 <= N <= 4) ==> !(0 <= N <= 4) && N >= 0)  }
        if N == 0 {                                                          ←
                {  N == 0  }                                                 ←
                r := 0;
                {  N== 0 && r == 0  }
                {  (N == 0 ==> r == 0)  && (0 < N <= 4 ==> r == 1) &&
                   (N > 4 ==> exp(r) >= N && exp(r-1) < N)  }

        } else if 0 < N <= 4 {                                               ←
                {  0 < N <= 4  }                                             ←
                r := 1;
                {  0 < N <= 4 && r == 1  }
                {  (N == 0 ==> r == 0)  && (0 < N <= 4 ==> r == 1) &&
                   (N > 4 ==> exp(r) >= N && exp(r-1) < N)  }

        } else {
                {  !(0 <= N <= 4) && N >= 0}
                {  !(0 <= N <= 4) && 20 == exp(5) && exp(4) < N  }
                r := 5;
                {  !(0 <= N <= 4) && 20 == exp(r)  && exp(r-1) < N  }
                var z := 20;
                {  !(0 <= N <= 4) && z == exp(r) && exp(r-1) < N  }
                …                                  // see “initialisation” below
                while z < N                                                          ←
                        invariant  !(0 <= N <= 4) && z == exp(r) && exp(r-1) < N
                        invariant  …               // see “additional invariant” below
                        decreases  N – z
                {

                        …                          // see “loop body” below

                }
                {  !(0 <= N <= 4) && z == exp(r)  && exp(r-1) < N && z >= N  }
                {  !(0 <= N <= 4) && exp(r) >= N && exp(r-1) < N  }
                {  (N == 0 ==> r == 0)  && (N <= 4 ==> r == 1) &&
                   (N > 4 ==> exp(r) >= N && exp(r-1) < N)  }
        }
```

{ (N == 0 ==> r == 0) && (N <= 4 ==> r == 1) &&
(N > 4 ==> exp(r) >= N && exp(r-1) < N) }
}

1) Loop body required for the invariant given in the proof above:

{ !(0 <= N <= 4) && z == exp(r) && exp(r-1) < N && z < N } ←
{ !(0 <= N <= 4) && z + k == exp(r) + (3*r*r – 9*r + 4) && exp(r) < N }
z := z + k; // where k == 3*r*r – 9*r + 4
{ !(0 <= N <= 4) && z == exp(r) + (3*r*r – 9*r + 4) && exp(r) < N } ←
{ !(0 <= N <= 4) && z == (r*r*r - 6*r*r + 9*r) + (3*r*r – 9*r + 4) && exp(r) < N }
{ !(0 <= N <= 4) &&
z == r*r*r + 2*r*r + r + r*r + 2*r + 1 – 6*r*r – 12*r – 6 + 9*r + 9 && exp(r) < N }
{ !(0 <= N <= 4) && z == (r+1)*(r*r + 2*r + 1) – 6*(r*r + 2*r + 1) + 9*r + 9 &&
exp(r) < N }
{ !(0 <= N <= 4) && z == (r+1)*(r+1)*(r+1) – 6*(r+1)*(r+1) + 9*(r+1) &&
exp(r) < N }
{ !(0 <= N <= 4) && z == exp(r+1) && exp(r) < N }
r := r + 1;
{ !(0 <= N <= 4) && z == exp(r) && exp(r-1) < N }

2) Loop body required for invariant k == 3*r*r – 9*r + 4:

{ k == 3*r*r – 9*r + 4 && z < N }
{ k == 3*r*r – 9*r + 4 } ←
{ k + m == 3*r*r – 9*r + 4 +m }
k := k + m; // where m == 6*r - 6
{ k == 3*r*r – 9*r + 4 + 6*r - 6 }
{ k == 3*r*r + 6*r + 3 – 9*r - 9 + 4 }
{ k == 3*(r+1)*(r+1) – 9*(r+1) + 4 }
r := r+1;
{ k == 3*r*r – 9*r + 4 }

3) Loop body required for invariant m == 6*r - 6:

{ m == 6*r – 6 && z < N }
{ m == 6*r – 6 } ←
{ m + 6 == 6*r -6 + 6 }
m := m + 6;
{ m == 6*r + 6 – 6 }
{ m == 6*(r+1) – 6 }
r := r +1;
{ m == 6*r – 6 }

From 1), 2) and 3) we can deduce that

```
{  !(0 <= N <= 4) && z == exp(r)  && exp(r-1) < N &&
k == 3*r*r – 9*r + 4 && m == 6*r – 6 && z < N }
z, k, m := z + k, k + m, m+ 6;
r := r + 1;
{  !(0 <= N <= 4) && z == exp(r)  && exp(r-1) < N &&
   k == 3*r*r – 9*r + 4 && m == 6*r – 6  }
```

Hence, the "additional invariant" is

```
k == 3*r*r – 9*r + 4 && m == 6*r – 6
```

the "loop body" is

```
z, k, m := z + k, k + m, m+ 6;
r := r + 1;
```

and the "initialisation" is

```
var k := 34;                // since 3*5*5 – 9*5 + 4 == 34
var m := 24;                // since 6*5 – 6 == 24
```

(b) See a2q2.dfy. 0.5 marks was given for the calcs before assignments to z, k and m, and 0.5 marks for the calcs after the assignments to z, k and m.

Total Mark: *15*