Assignment 2 CSSE3100/7100 Reasoning about Programs

Sample solution

(a) Proof of ComputeFusc

A sample solution is provided below. Each red number (1) represents 0.5 marks. (1^T denotes marks for the proof of termination.)

```
method ComputeFusc(N: int) returns (b: int)
    requires N >= 0
    ensures b == fusc(N)
{
    { N >= 0 } 1
    { fusc(N) == 1 * fusc(N) + 0 * fusc(N + 1) && N >= 0 }
    b := 0;
    { fusc(N) == 1 * fusc(N) + b * fusc(N + 1) && N >= 0 }
    { forall n, a :: fusc(N) == 1 * fusc(N) + b * fusc(N + 1) && N >= 0 }
    var n, a := N, 1;
    { fusc(N) == a * fusc(n) + b * fusc(n + 1) && n >= 0 }
    while n != 0
        invariant fusc(N) == a * fusc(n) + b * fusc(n + 1)
        invariant n >= 0 1^T
        decreases n 2^T
    {
        { n != 0 && fusc(N) == a * fusc(n) + b * fusc(n + 1) && n >= 0} 2
        { fusc(N) == a * fusc(n) + b * fusc(n + 1) && n > 0 }
        { forall d :: fusc(N) == a * fusc(n) + b * fusc(n + 1) && n > 0 }
        { forall d :: (n % 2 == 0 ==> a * fusc(n) + b * fusc(n + 1) && n > 0) &&
                (n % 2 != 0 ==> a * fusc(n) + b * fusc(n + 1) && n > 0) } 3^T
        ghost var d := n;
        { (n % 2 == 0 ==> a * fusc(n) + b * fusc(n + 1) && d > n / 2 >= 0) &&
          (n % 2 != 0 ==> a * fusc(n) + b * fusc(n + 1) && d > (n - 1) / 2 >= 0) } 3
        if n % 2 == 0 {
            { fusc(N) = a * fusc(n) + b * fusc(n + 1) && d > n / 2 > = 0 }
            { fusc(N) == a * fusc(2 * n / 2) +                        rule (iii) 4
                        b * fusc( 2 * n / 2 + 1) &&                   rule (iv) 5
                    d > n / 2 >= 0 }
            { fusc(N) == a * fusc(n / 2) + b * (fusc(n / 2) + fusc(n / 2 + 1)) }
            { fusc(N) == (a + b) * fusc(n / 2) + b * fusc(n / 2 + 1) && d > n / 2 >= 0 } 6
            a := a + b;
            { fusc(N) == a * fusc(n / 2) + b * fusc(n / 2 + 1) && d > n / 2 >= 0 } 7
            n := n / 2;
```

```
                    { fusc(N) == a * fusc(n) + b * fusc(n + 1) && d > n >= 0 }
            } else {
                    { fusc(N) == a * fusc(n) + b * fusc(n + 1) && d > (n - 1) / 2 >= 0 }
                    { fusc(N) == a * fusc(2 * (n - 1) / 2 + 1) +                       rule (iv) 8
                                    b * fusc(2 * (n + 1) / 2) &&                       rule (iii) 9
                            d > (n - 1) / 2 >= 0 }
                    { fusc(N) == a * (fusc((n - 1) / 2) + fusc((n - 1) / 2 + 1)) + b * ((n + 1) / 2) &&
                            d > (n - 1) / 2 >= 0 }
                    { fusc(N) == a * fusc((n - 1) / 2) +  (b + a) * fusc((n - 1) / 2 + 1) &&
                            d > (n - 1) / 2 >= 0 } 10
                    b := b + a;
                    { fusc(N) == a * fusc((n - 1) / 2) +  b * fusc((n - 1) / 2 + 1) &&
                            d > (n - 1) / 2 >= 0 } 11
                    n := (n - 1) / 2;
                    { fusc(N) == a * fusc(n) + b * fusc(n + 1) && d > n >= 0 }
            }
            { fusc(N) == a * fusc(n) + b * fusc(n + 1) && d > n >= 0 }
            { fusc(N) == a * fusc(n) + b * fusc(n + 1) && n >= 0 && d > n && d >= 0 } 4T
    }
    { n == 0 && fusc(N) == a * fusc(n) + b * fusc(n + 1) && n >= 0 } 12
                                        strengthening (by introducing n == 0), rules (i) and (ii)
    { b == fusc(N) }
}
```

The program is correct since the calculated precondition is implied by the state precondition.

(b) Derivation of ComputePos

A sample solution is provided below. Each red number (1) represents 0.5 marks. (1[C] denotes marks for derived code.) The solution uses the "Replace a constant with a variable" loop design technique applied to 2 constants, num and den.

```
method ComputePos(num: int, den: int) returns (n: int)
    requires num > 0 && den > 0
    ensures n > 0 && num == fusc(n) && den == fusc(n+1)
{
    { true }                                                               rules (ii) and (iii)
    { 1 > 0 && 1= fusc(1) && 1 = fusc(2) } 1
    n := 1; 1C
    { n > 0 && 1 = fusc(n) && 1 == fusc(n + 1) }
    { forall a, b :: n > 0 && 1 = fusc(n) && 1 == fusc(n + 1) }
    var a, b := 1, 1; 2C
    { n > 0 && a == fusc(n) && b == fusc(n + 1) }
    while (a != num || b != den) 3C
```

```
        invariant n > 0 && a == fusc(n) && b == fusc(n + 1) 2
    {
        { n > 0 && (a != num || b != den) && a == fusc(n) && b == fusc(n + 1) }
                                                                strengthening 3
        { n + 1 > 0 && b == fusc(n+1) } 4
        a := b; 4C
        { n + 1 > 0 && a == fusc(n + 1) }                        one-point rule 5
        { n + 2 > 0 &&
          forall b' :: b' = fusc(n + 2) ==> n + 1 > 0 && a == fusc(n + 1) && b' = fusc(n + 2) } 6
        b := ComputeFusc(n + 2); 5C
        { n + 1 > 0 && a == fusc(n + 1) && b == fusc(n + 2) } 7
        n := n + 1; 6C
        { n > 0 && a == fusc(n) && b == fusc(n + 1) }
    }
    { n > 0 && a == num && b == den && a == fusc(n) && b == fusc(n + 1) }
                          strengthening (by introducing a == num && b == den) 8
    { n > 0 && num == fusc(n) && den == fusc(n + 1) }
}
```