

```

1  // a)
2  method ComputeFusc(N: int) returns (b: int)
3      requires N >= 0                                     // P
4      ensures b == fusc(N)                               // R
5  {
6      b := 0;
7      var n, a := N, 1;
8
9      assert
10         0 <= n <= N;                                     // J
11
12     assert
13         fusc(N) == a * fusc(n) + b * fusc(n + 1); // J
14
15     while (n != 0)                                       // B
16         invariant 0 <= n <= N                          // J
17         invariant fusc(N) == a * fusc(n) + b * fusc(n + 1) // J
18         decreases n // D
19     {
20         ghost var d := n; // D
21
22         assert
23             fusc(N) == a * fusc(n) + b * fusc(n + 1); // J
24
25         assert
26             n != 0; // B
27         assert
28             (n % 2 != 0 && n % 2 == 0) || fusc(N) == a * fusc(n) + b * fusc(n + 1);
29         assert
30             (n % 2 != 0 || n % 2 == 0) ==> fusc(N) == a * fusc(n) + b * fusc(n + 1);
31
32         assert
33             n % 2 != 0 || fusc(N) == a * fusc(n) + b * fusc(n + 1);
34         assert
35             n % 2 == 0 || fusc(N) == a * fusc(n) + b * fusc(n + 1);
36
37         assert
38             n % 2 == 0 ==> fusc(N) == a * fusc(n) + b * fusc(n + 1);
39         assert
40             n % 2 != 0 ==> fusc(N) == a * fusc(n) + b * fusc(n + 1);
41
42         if (n % 2 == 0)
43         {
44             assert
45                 fusc(N) == a * fusc(n) + b * fusc(n + 1); // J
46
47             rule3(n/2);
48             assert
49                 fusc(n/2) == fusc(n);
50
51             assert
52                 fusc(N) == a * fusc(n/2) + b * fusc(n + 1);
53
54             assert
55                 fusc(N) == a * fusc(n/2) + b * fusc(n/2) + b * fusc(n + 1) - b * fusc(n/2);
56
57             assert
58                 fusc(N) == a * fusc(n/2) + b * fusc(n/2) + b * (fusc(n + 1) - fusc(n/2));
59
60             rule4(n/2);
61             assert
62                 fusc((n/2) + 1) == fusc(n + 1) - fusc(n/2);
63

```

```

64  assert
65      fusc(N) == a * fusc(n/2) + b * fusc(n/2) + b * fusc((n/2) + 1);
66
67  assert
68      fusc(N) == (a + b) * fusc(n/2) + b * fusc((n/2) + 1);
69
70  a := a + b;
71
72  assert
73      fusc(N) == a * fusc(n/2) + b * fusc((n/2) + 1);
74
75  n := n / 2;
76
77  assert
78      fusc(N) == a * fusc(n) + b * fusc(n + 1);
79  } else {
80      assert
81          fusc(N) == a * fusc(n) + b * fusc(n + 1);    // J
82
83      rule3((n + 1)/2);
84      assert
85          fusc((n + 1)/2) == fusc(n + 1);
86
87      assert
88          fusc(N) == b * fusc(((n - 1)/2) + 1) + a * fusc(n);
89
90      assert
91          fusc(N) ==
92              b * fusc(n) - b * fusc(n) + b * fusc(((n - 1)/2) + 1) + a * fusc(n);
93
94      assert
95          fusc(N) ==
96              b * fusc(n) - b * (fusc(n) - fusc(((n - 1)/2) + 1)) + a * fusc(n);
97
98      rule4((n - 1)/2);
99      assert
100          fusc((n - 1)/2) == fusc(n) - fusc(((n - 1)/2) + 1);
101
102      assert
103          fusc(N) == b * fusc(n) - b * fusc((n - 1)/2) + a * fusc(n);
104
105      rule3((n - 1)/2);
106      assert
107          fusc(n-1) == fusc((n - 1)/2);
108
109      assert
110      fusc(N) == b * fusc(n) - b * fusc(n - 1) + a * fusc(n);
111
112      assert
113          fusc(N) == b * fusc(n) - b * fusc(n - 1) + a * fusc(n);
114
115      assert
116          fusc(N) ==
117              a * fusc(n - 1) + b * fusc(n) - b * fusc(n - 1)
118              + a * fusc(n) - a * fusc(n - 1);
119
120      assert
121          fusc(N) == a * fusc(n - 1) + (b + a) * (fusc(n) - fusc(n - 1));
122
123      rule3((n - 1)/2);
124      assert
125          fusc(n - 1) == fusc((n - 1) / 2);
126

```

```

127     assert
128         fusc(N) == a * fusc(n - 1) + (b + a) * (fusc(n) - fusc((n - 1)/2));
129
130 rule3((n - 1)/2);
131     assert
132         fusc(n - 1) == fusc((n - 1) / 2);
133
134     assert
135         fusc(N) == a * fusc((n - 1) / 2) + (b + a) * (fusc(n) - fusc((n - 1)/2));
136
137 rule4((n - 1)/2);
138     assert
139         fusc(((n - 1)/2) + 1) == fusc(n) - fusc((n - 1)/2);
140
141     assert
142         fusc(N) == a * fusc((n - 1) / 2) + (b + a) * fusc(((n - 1)/2) + 1);
143
144     b := b + a;
145
146     assert
147         fusc(N) == a * fusc((n - 1) / 2) + b * fusc(((n - 1)/2) + 1);
148
149     n := (n - 1) / 2;
150
151     assert
152         fusc(N) == a * fusc(n) + b * fusc(n + 1);
153 }
154
155     assert
156         n < d; // D < d
157
158     assert
159         fusc(N) == a * fusc(n) + b * fusc(n + 1); // J
160 }
161     assert
162         fusc(N) == a * fusc(n) + b * fusc(n + 1); // J
163
164     assert
165         n == 0; // !B
166
167     assert
168         fusc(N) == a * fusc(0) + b * fusc(0 + 1); // J
169
170     assert
171         fusc(N) == a * fusc(0) + b * fusc(1); // J
172
173 rule1();
174     assert
175         fusc(0) == 0;
176
177     assert
178         fusc(N) == a * 0 + b * fusc(1); // J
179
180 rule2();
181     assert
182         fusc(1) == 1;
183
184     assert
185         fusc(N) == a * 0 + b * 1; // J
186
187     assert
188         fusc(N) == b; // R
189 }

```

```

1 // b)
2 method ComputePos(num: int, den: int) returns (n: int)
3   requires num > 0 && den > 0 // P
4   ensures n > 0 && num == fusc(n) && den == fusc(n + 1) // R
5 {
6   var nu, de := 1, 1;
7   n := 1;
8
9   assert
10    n == 1;
11
12   rule2();
13   assert
14    fusc(n) == nu;
15
16   rule3(n);
17   assert
18    fusc(n + 1) == de;
19
20   assert
21    nu == fusc(n) && de == fusc(n + 1); // J
22
23   while !(nu == num && de == den) // B
24     invariant n > 0 // J
25     invariant nu == fusc(n) // J
26     invariant de == fusc(n + 1) // J
27   {
28     assert nu == fusc(n); // J
29     assert de == fusc(n + 1); // J
30
31     var t := ComputeFusc(n+2);
32
33     // Method Call Rule
34
35   /*
36   method call
37   method ComputeFusc(N: int) returns (b: int)
38   requires N >= 0 // P
39   ensures b == fusc(N) // R
40
41   WP[t := ComputeFusc(E), Q] =
42   P[N \ E] && forall b' :: R[N, b \ E, b'] ==> Q[t \ b']
43
44   WP[t := ComputeFusc(E), Q] =
45   P[N \ (n+2)] && forall b' :: R[N, b \ (n+2), b'] ==> Q[t \ b']
46   N >= 0[N \ (n+2)] && forall b' :: b == fusc(N)[N, b \ (n+2), b'] ==> Q[t \ b']
47   (n+2) >= 0 && forall b' :: b' == fusc((n+2)) ==> Q[t \ b']
48
49   // One-point rule
50
51   (n+2) >= 0 && b' == fusc((n+2)) ==> Q[t \ b']
52
53   (n+2) >= 0 && b' == fusc((n+2)) ==> t == fusc(n+2)[t \ b']
54   (n+2) >= 0 && b' == fusc((n+2)) ==> b' == fusc(n+2)
55 */
56   assert
57    t == fusc(n+2);
58
59   nu, de := de, t;
60
61   assert
62    nu == fusc(n + 1);
63

```

```
64     assert
65         de == fusc(n + 2);
66
67     assert
68         nu == fusc(n + 1) && de == fusc(n + 2);
69
70     n := n + 1;
71
72     assert
73         nu == fusc(n) && de == fusc(n + 1);
74 }
75
76 assert
77     (nu == num && de == den);           // !B
78
79 assert
80     nu == fusc(n) && de == fusc(n + 1); // J
81
82 assert
83     nu == num && de == den;           // R
84 }
```