

lib/memory-range/count-range.ath

```

1 load "forward-iterator"
2
3 #.....
4 extend-module Forward-Iterator {
5
6   define collect := Trivial-Iterator.collect
7
8   declare count1: (S, X) [S (It X S) (It X S) N] -> N
9
10  declare count: (S, X) [S (It X S) (It X S)] -> N
11
12  module count {
13
14    define A := ?A:N
15
16    define axioms :=
17      (fun
18        [(M \\< (count1 x i j A)) =
19          [A
20            (M \\< (count1 x (successor i) j (S A))) when (i = j)
21              M at deref i = x)
22            (M \\< (count1 x (successor i) j A)) when (i /= j &
23              M at deref i /= x)]
24
25          (M \\< (count x i j)) = (M \\< (count1 x i j zero))])
26
27    define [if-empty if-equal if-unequal definition] := axioms
28
29    (add-axioms theory axioms)
30
31    define count' := List.count
32    overload + N.+
33
34    define (correctness1-prop r) :=
35      (forall M x i j A .
36        (range i j) = SOME r ==>
37          M \\< (count1 x i j A) = (count' x (collect M r)) + A)
38
39    define correctness1 := (forall r . correctness1-prop r)
40
41    define correctness :=
42      (forall r M x i j .
43        (range i j) = SOME r ==>
44          M \\< (count x i j) = (count' x (collect M r)))
45
46    define proofs :=
47      method (theorem adapt)
48        let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
49            [deref successor] := (adapt [deref successor])}
50        match theorem {
51          (val-of correctness1) =>
52            by-induction (adapt theorem) {
53              (stop h:(It 'X 'S)) =>
54                pick-any M:(Memory 'S) x:'S i:(It 'X 'S) j:(It 'X 'S) A:N
55                  assume I := ((range i j) = (SOME stop h))
56                  let {ER1 := (!prove empty-rangel);
57                      _ := (!chain-> [I ==> (i = j) [ER1]])}
58                  (!combine-equations
59                    (!chain [(M \\< (count1 x i j A))
60                              = A [if-empty]])
61                      (!chain [((count' x (collect M (stop h))) + A)
62                                = ((count' x nil) + A) [collect.of-stop]
63                                = (zero + A) [List.count.empty]
64                                = A [N.Plus.left-zero]]))
65                | (r as (back r':(Range 'X 'S))) =>
66                  let {ind-hyp := (correctness1-prop r')}
67                  pick-any M:(Memory 'S) x:'S i:(It 'X 'S) j:(It 'X 'S) A:N

```

```

68     assume I := ((range i j) = SOME r)
69     let {goal := (M \\< (count1 x i j A) =
70           (count' x (collect M r)) + A);
71         NB1 := (!prove nonempty-back1);
72         LB := (!prove range-back);
73         II := conclude (i != j)
74           (!chain-> [I ==> (i != j) [NB1]]);
75         III := (!chain->
76             [I ==> ((range (successor i) j) = SOME r')
77                  [LB]]);
78         IV := conclude (i = start r)
79             (!chain->
80             [(range i j)
81              = (SOME r) [I]
82              = (range (start r)
83                    (finish r)) [range.collapse]
84              ==> (i = start r &
85                  j = finish r) [range.injective]
86              ==> (i = start r) [left-and]]))
87     (!two-cases
88     assume case1 := (M at deref i = x)
89     conclude goal
90     (!combine-equations
91     (!chain
92     [(M \\< (count1 x i j A))
93      = (M \\< (count1 x (successor i) j (S A))] [if-equal]
94      = ((count' x (collect M r')) + (S A)) [III ind-hyp]
95      = (S ((count' x (collect M r')) + A))
96          [N.Plus.right-nonzero]))
97     (!chain
98     [((count' x (collect M r)) + A)
99      = ((count' x (M at (deref i)) :: (collect M r')) + A)
100        [IV collect.of-back]
101      = ((S (count' x (collect M r')) + A)
102        [case1 List.count.more]
103      = (S ((count' x (collect M r')) + A))
104          [N.Plus.left-nonzero]))))
105     assume case2 := (M at deref i != x)
106     conclude goal
107     let {_ := (!sym case2)}
108     (!combine-equations
109     (!chain
110     [(M \\< (count1 x i j A))
111      = (M \\< (count1 x (successor i) j A)) [if-unequal]
112      = ((count' x (collect M r')) + A) [III ind-hyp]))
113     (!chain
114     [((count' x (collect M r)) + A)
115      = ((count' x (M at deref i) :: (collect M r')) + A)
116        [IV collect.of-back]
117      = ((count' x (collect M r')) + A)
118        [case2 List.count.same]))))
119     } # by-induction
120 | (val-of correctness) =>
121     let {L1 := (!prove correctness1)}
122     pick-any r:(Range 'X 'S) M:(Memory 'S) x:'S
123             i:(It 'X 'S) j:(It 'X 'S)
124     assume ((range i j) = SOME r)
125     (!chain
126     [(M \\< (count x i j))
127      = (M \\< (count1 x i j zero)) [definition]
128      = ((count' x (collect M r)) + zero) [L1]
129      = (count' x (collect M r)) [N.Plus.right-zero]])
130     } # match theorem
131
132 (add-theorems theory |{[correctness1 correctness] := proofs}|)
133 } # count
134 } # Forward-Iterator

```