

## lib/memory-range/count-range0.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 :=
35      (forall r 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 correctness :=
40      (forall r M x i j .
41        (range i j) = SOME r ==>
42        M \ (Forward-Iterator.count x i j) = (count x (collect M r)))
43
44    define theorems := [correctness1 correctness]
45
46    define (correctness1-prop r) :=
47      (forall M x i j A .
48        (range i j) = SOME r ==>
49        M \ (count1 x i j A) = (count x (collect M r)) + A)
50
51    define proofs :=
52      method (theorem adapt)
53        let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
54            deref := (adapt deref)}
55        match theorem {
56          (val-of correctness1) =>
57          by-induction (adapt theorem) {
58            (stop h:(It 'X 'S)) =>
59            pick-any M:(Memory 'S) x:'S i:(It 'X 'S) j:(It 'X 'S) A:N
60              assume I := ((range i j) = (SOME stop h))
61              let {EL1 := (!prove empty-range1);
62                  _ := (!chain-> [I ==> (i = j) [EL1]])}
63              (!combine-equations
64                (!chain [(M \ (count1 x i j A))
65                          = A
66                          [if-empty]])
67                (!chain [((count x (collect M (stop h))) + A)
68                          = ((count x nil) + A)
69                          [collect.of-stop]

```

```

68         = (zero + A)                                [List.count.empty]
69         = A                                           [N.Plus.left-zero]]))
70 | (r as (back r':(Range 'X 'S))) =>
71   let {ind-hyp := (correctness1-prop r')}
72   pick-any M:(Memory 'S) x:'S i:(It 'X 'S) j:(It 'X 'S) A:N
73     assume I := ((range i j) = SOME back r)
74     let {goal := (M \ (count1 x i j A) =
75       (count x (collect M (back r))) + A);
76       NB1 := (!prove nonempty-back1);
77       LB := (!prove range-back);
78       II := conclude (i != j)
79         (!chain-> [I ==> (i != j) [NB1]]);
80       III := (!chain->
81         [I ==> ((range (successor i) j) = SOME r)
82           [LB]]);
83       IV := conclude (i = (start (back r)))
84         (!chain->
85           [(range i j)
86             = (SOME (back r)) [I]
87             = (range (start back r)
88               (finish back r)) [range.collapse]
89             ==> (i = start back r &
90               j = finish back r) [range.injective]
91             ==> (i = start back r) [left-and]]])
92   (!two-cases
93     assume case1 := (M at deref i = x)
94     conclude goal
95       (!combine-equations
96         (!chain
97           [(M \ (count1 x i j A))
98             = (M \ (count1 x (successor i) j (S A))) [if-equal]
99             = ((count x (collect M r)) + (S A)) [III ind-hyp]
100             = (S ((count x (collect M r)) + A))
101               [N.Plus.right-nonzero]]))
102         (!chain
103           [((count x (collect M (back r))) + A)
104             = ((count x (M at (deref i)) :: (collect M r)) + A)
105               [IV collect.of-back]
106             = ((S (count x (collect M r))) + A)
107               [case1 List.count.more]
108             = (S ((count x (collect M r)) + A))
109               [N.Plus.left-nonzero]]))
110     assume case2 := (M at deref i != x)
111     conclude goal
112     let {_ := (!sym case2)}
113     (!combine-equations
114       (!chain
115         [(M \ (count1 x i j A))
116           = (M \ (count1 x (successor i) j A)) [if-unequal]
117           = ((count x (collect M r)) + A) [III ind-hyp]]))
118       (!chain
119         [((count x (collect M (back r))) + A)
120           = ((count x (M at deref i) :: (collect M r)) + A)
121             [IV collect.of-back]
122           = ((count x (collect M r)) + A)
123             [case2 List.count.same]]))
124   } # by-induction
125 | (val-of correctness) =>
126   let {L1 := (!prove correctness)}
127   pick-any r:(Range 'X 'S) M:(Memory 'S) x:'S
128     i:(It 'X 'S) j:(It 'X 'S)
129     assume ((range i j) = SOME r)
130     (!chain
131       [(M \ (Forward-Iterator.count x i j))
132         = (M \ (count1 x i j zero)) [definition]
133         = ((count x (collect M r)) + zero) [L1]
134         = (count x (collect M r)) [N.Plus.right-zero]]
135     ) # match theorem
136
137 (add-theorems theory |{theorems := proofs}|)

```

```
138 } # count  
139 } # Forward-Iterator
```