

## lib/memory-range/reverse-range.ath

```

1 #.....
2
3 load "bidirectional-iterator"
4
5 #.....
6
7 extend-module Bidirectional-Iterator {
8
9   module Reversing {
10    define join := List.join
11    define reverse := List.reverse
12
13    declare reverse-iterator: (X, S) [(It X S)] -> (It (It X S) S)
14    declare reverse-range: (X, S) [(Range X S)] -> (Range (It X S) S)
15    declare base-iterator: (X, S) [(It (It X S) S)] -> (It X S)
16    declare base-range: (X, S) [(Range (It X S) S)] -> (Range X S)
17
18    define deref-reverse :=
19      (forall i . deref reverse-iterator i = deref predecessor i)
20
21    define *reverse-in :=
22      (forall i r . (reverse-iterator i) *in (reverse-range r) <==>
23        (predecessor i) *in r)
24
25    define base-reverse-range :=
26      (forall r . base-range reverse-range r = r)
27
28    define reverse-base-range :=
29      (forall r . reverse-range base-range r = r)
30
31    define reverse-of-range :=
32      (forall i j r .
33        (range (reverse-iterator j) (reverse-iterator i)) = SOME r
34        <==> (range i j) = SOME base-range r)
35
36    define reverse-base :=
37      (forall i . reverse-iterator base-iterator i = i)
38
39    define collect-reverse-stop :=
40      (forall M i . (collect M (reverse-range stop i)) = nil)
41
42    define collect-reverse-back :=
43      (forall M r .
44        (collect M reverse-range back r) =
45        (collect M reverse-range r) join ((M at deref start back r) :: nil))
46
47      (add-axioms theory [deref-reverse *reverse-in base-reverse-range
48        reverse-base-range reverse-of-range reverse-base
49        collect-reverse-stop collect-reverse-back])
50 #.....
51
52 define reverse-range-reverse :=
53   (forall i j r .
54     (range (reverse-iterator j)
55       (reverse-iterator i)) = SOME (reverse-range r)
56     <==> (range i j) = SOME r)
57
58 define collect-reverse :=
59   (forall r M .
60     (collect M reverse-range r) = reverse (collect M r))
61
62 define [M' r'] := [?M':(Memory 'S) ?r':(Range (It 'Y 'S) 'S)]
63
64 define collect-reverse-corollary :=
65   (forall M r M' r' .
66     (collect M' r') = (collect M reverse-range r)
67     ==> (collect M' base-range r') = (collect M r))

```

```

68
69 define proofs :=
70 method (theorem adapt)
71 let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
72   deref := (adapt deref)}
73 match theorem {
74   (val-of reverse-range-reverse) =>
75   pick-any i:(It 'X 'S) j:(It 'X 'S) r:(Range 'X 'S)
76     (!chain
77       [(range (reverse-iterator j) (reverse-iterator i)) =
78         SOME (reverse-range r)
79         <==> ((range i j) = SOME (base-range (reverse-range r)))
80           [reverse-of-range]
81         <==> ((range i j) = SOME r)
82           [base-reverse-range]])
83 | (val-of collect-reverse) =>
84   by-induction (adapt theorem) {
85     (stop i) =>
86     pick-any M
87       (!combine-equations
88         (!chain->
89           [(collect M (reverse-range (stop i)))
90            = nil
91            [collect-reverse-stop]]
92         (!chain
93           [(reverse (collect M (stop i)))
94            = (reverse nil)
95            [collect.of-stop]
96            = nil
97            [List.reverse.empty]]))
98 | (r as (back r')) =>
99   let {ind-hyp := (forall M .
100     (collect M reverse-range r') =
101     reverse (collect M r'))}
102   pick-any M
103     (!combine-equations
104       (!chain
105         [(collect M reverse-range r)
106          = ((collect M reverse-range r')
107            join
108            ((M at deref start r) :: nil))
109          [collect-reverse-back]
110          = ((reverse (collect M r'))
111            join
112            ((M at deref start r) :: nil))
113          [ind-hyp]])
114       (!chain
115         [(reverse (collect M r))
116          = (reverse (M at deref start r)
117            :: (collect M r'))
118          [collect.of-back]
119          = ((reverse (collect M r'))
120            join
121            ((M at deref start r) :: nil))
122          [List.reverse.nonempty]]))
123     }
124 | (val-of collect-reverse-corollary) =>
125   pick-any M:(Memory 'S) r:(Range 'X 'S)
126     M':(Memory 'S) r':(Range (It 'Y 'S) 'S)
127   assume A := ((collect M' r') = (collect M (reverse-range r)))
128   let {CR := (!prove collect-reverse)}
129     (!chain
130       [(collect M' (base-range r'))
131        = (reverse reverse (collect M' (base-range r'))))
132        [List.reverse.of-reverse]
133       = (reverse (collect M' reverse-range base-range r'))
134         [CR]
135       = (reverse (collect M' r'))
136         [reverse-base-range]
137       = (reverse (collect M reverse-range r))
138         [A]
139       = (reverse reverse (collect M r))
140         [CR]
141       = (collect M r)
142         [List.reverse.of-reverse]])
143   }
144 }
145
146 (add-theorems theory |[reverse-range-reverse collect-reverse
147   collect-reverse-corollary] := proofs)|)
148 } # Reversing
149 } # Bidirectional-Iterator

```