

lib/memory-range/forward-iterator.ath

```

1 load "trivial-iterator"
2 #.....
3
4 module Forward-Iterator {
5   open Trivial-Iterator
6
7   define theory := (make-theory ['Trivial-Iterator] [])
8
9   declare successor: (X, S) [(It X S)] -> (It X S)
10
11  module successor {
12
13    define of-start := (forall r . successor start back r = start r)
14
15    define injective := (forall i j . successor i = successor j ==> i = j)
16
17    define deref-of :=
18      (forall i r . deref successor i = deref start r
19        ==> deref i = deref start back r)
20
21    (add-axioms theory [of-start injective deref-of])
22  }
23
24  define start-shift :=
25    (forall i r . successor i = start r ==> i = start back r)
26
27  define range-back :=
28    (forall i j r . (range (successor i) j) = SOME r
29      <==> (range i j) = SOME (back r))
30
31  define (finish-not-*in-prop r) :=
32    (forall i j k . (range i j) = SOME r & k *in r ==> k /= j)
33
34  define finish-not-*in := (forall r . finish-not-*in-prop r)
35
36  define proofs :=
37    method (theorem adapt)
38      let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
39          [deref *in successor] := (adapt [deref *in successor])}
40      match theorem {
41        (val-of start-shift) =>
42          pick-any i:(It 'X 'S) r:(Range 'X 'S)
43            assume I := (successor i = start r)
44              (!chain->
45                [(successor i)
46                 = (start r) [I]
47                 = (successor start back r) [successor.of-start]
48                 ==> (i = start back r) [successor.injective]])
49          | (val-of range-back) =>
50            pick-any i:(It 'X 'S) j:(It 'X 'S) r:(Range 'X 'S)
51              (!equiv
52                assume I := ((range (successor i) j) = SOME r)
53                  let {SS1 := (!prove start-shift);
54                      II := (!chain->
55                        [(range (successor i) j)
56                         = (SOME r) [I]
57                         = (range (start r)
58                           (finish r)) [range.collapse]
59                         ==> (successor i = start r & j = finish r)
60                           [range.injective]])}
61                  (!chain [(range i j)
62                           = (range (start back r)
63                             (finish back r)) [II SS1 finish.of-back]
64                           = (SOME back r) [range.collapse]])
65                assume I := ((range i j) = (SOME back r))
66                let {II := (!chain->
67                  [(range i j)

```

```

68         = (SOME back r) [I]
69         = (range (start back r)
70             (finish back r)) [range.collapse]
71         ==> (i = start back r &
72             j = finish back r) [range.injective]]}
73     (!chain [(range (successor i) j)
74             = (range (start r) (finish r))
75                 [II successor.of-start finish.of-back]
76             = (SOME r) [range.collapse]])
77 | (val-of finish-not-*in) =>
78   by-induction (adapt theorem) {
79     (stop h) =>
80     pick-any i j k
81     assume ((range i j) = SOME stop h & k *in stop h)
82     (!from-complements (k /= j)
83      (k *in stop h)
84      (!chain->
85       [true ==> (~ k *in stop h) [*in.of-stop]]))
86 | (r as (back r':(Range 'X 'S))) =>
87   let {ind-hyp := (finish-not-*in-prop r')}
88   pick-any i:(It 'X 'S) j:(It 'X 'S) k:(It 'X 'S)
89   let {A1 := ((range i j) = SOME r);
90       A2 := (k *in r);
91       NB := (!prove nonempty-back)}
92   assume (A1 & A2)
93   let {B1 := (!chain->
94           [A2 ==> (deref k = deref start r |
95                 k *in r') [*in.of-back]])}
96   (!cases B1
97    assume Bla := (deref k = deref start r)
98    let {C1 := (!chain->
99            [Bla ==> (k = start r)
100                 [deref.injective]])}
101    (and C2 C3) :=
102    (!chain->
103     [(range i j)
104      = (SOME r) [A1]
105      = (range (start r) (finish r))
106              [range.collapse]
107      ==> (i = start r &
108          j = finish r)
109          [range.injective]])}
110   (!chain->
111    [true ==> (start r /= finish r)
112             [NB]
113             ==> (k /= j) [C1 C3]])
114   assume B1b := (k *in r')
115   let {RB := (!prove range-back);
116       C1 := (!chain->
117             [A1
118              ==> ((range (successor i) j) =
119                  SOME r') [RB]])}
120   _ := (!both C1 B1b)
121   (!fire ind-hyp [(successor i) j k])
122 } # by-induction
123 } # match theorem
124
125 (add-theorems theory |[start-shift range-back finish-not-*in] := proofs)|)
126 #.....
127
128 define range-shift1 :=
129   (forall r i . (successor i) in r ==> i in back r)
130 define range-shift2 :=
131   (forall i r . ~ i in back r ==> ~ (successor i) in r)
132
133 define proofs :=
134   method (theorem adapt)
135     let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
136         successor := (adapt successor)}
137   match theorem {

```

```

138 (val-of range-shift1) =>
139 by-induction (adapt theorem) {
140 (stop h) =>
141 pick-any i
142 assume I := ((successor i) in stop h)
143 let {II := (!chain->
144 [true
145 ==> (~ (successor i) in stop h) [in.of-stop]])}
146 (!from-complements (i in back stop h) I II)
147 | (r as (back r':(Range 'X 'S))) =>
148 let {ind-hyp := (forall i . (successor i) in r' ==> i in r)}
149 pick-any i
150 assume A := ((successor i) in r)
151 let {case1 := (successor i = start r);
152 case2 := ((successor i) in r');
153 goal := (i in back r);
154 B := (!chain->
155 [A ==> (case1 | case2) [in.of-back]]);
156 SS := (!prove start-shift)}
157 (!cases B
158 assume case1
159 (!chain->
160 [case1
161 ==> (i = start back r) [SS]
162 ==> (i = start back r | i in r) [alternate]
163 ==> goal [in.of-back]])
164 assume case2
165 (!chain->
166 [case2
167 ==> (i in r) [ind-hyp]
168 ==> (i = start back r | i in r) [alternate]
169 ==> goal [in.of-back]]))
170 }
171 | (val-of range-shift2) =>
172 pick-any i r
173 let {RS1 := (!prove range-shift1);
174 p := (!chain [(successor i) in r
175 ==> (i in back r) [RS1]])}
176 (!contra-pos p)
177 }
178
179 (add-theorems theory |[range-shift1 range-shift2] := proofs|)
180
181 #.....
182
183
184 module *in {
185 open Trivial-Iterator.*in
186
187 define range-shift1 :=
188 (forall r i . (successor i) *in r ==> i *in back r)
189 define range-shift2 :=
190 (forall i r . ~ i *in back r ==> ~ (successor i) *in r)
191
192 define proofs :=
193 method (theorem adapt)
194 let {[get prove chain chain-> chain<-] := (proof-tools adapt theory)}
195 match theorem {
196 (val-of range-shift1) =>
197 by-induction (adapt theorem) {
198 (stop h) =>
199 pick-any i
200 assume I := ((successor i) *in stop h)
201 let {II := (!chain->
202 [true ==> (~ (successor i) *in stop h)
203 [of-stop]])}
204 (!from-complements (i *in back stop h) I II)
205 | (back r:(Range 'Y 'S)) =>
206 let {ind-hyp := (forall ?i:(It 'X 'S) .
207 (successor ?i) *in r ==> ?i *in back r)}

```

```

208     pick-any i
209     assume A := ((successor i) *in back r)
210     let {case1 := (deref successor i = deref start back r);
211         case2 := ((successor i) *in r);
212         goal := (i *in back back r);
213         B := (!chain->
214             [A ==> (case1 | case2)           [of-back]]);
215         DO := (!prove successor.deref-of)}
216     (!cases (case1 | case2)
217         assume case1
218             (!chain->
219                 [case1
220                 ==> (deref i = deref start back back r)   [DO]
221                 ==> (deref i = deref start back back r |
222                     i *in back r)                       [alternate]
223                 ==> goal                                 [of-back]])
224         assume case2
225             (!chain->
226                 [case2
227                 ==> (i *in back r)                       [ind-hyp]
228                 ==> (deref i = deref start back back r |
229                     i *in back r)                       [alternate]
230                 ==> goal                                 [of-back]]))
231     }
232 | (val-of range-shift2) =>
233     pick-any i r
234     let {RS1 := (!prove range-shift1);
235         p := (!chain [((successor i) *in r)
236                     ==> (i *in back r)   [RS1]])}
237     (!contra-pos p)
238 }
239
240 (add-theorems theory |[range-shift1 range-shift2] := proofs|)
241 } # close module *in
242 } # close module Forward-Iterator

```