

lib/memory-range/replace-range.ath

```

1 #.....
2
3 load "forward-iterator"
4
5 #.....
6
7 extend-module Forward-Iterator {
8
9   declare replace: (S, X) [(It X S) (It X S) S S] -> (Memory.Change S)
10
11  module replace {
12
13   define axioms :=
14     (fun
15       [(M \ (replace i j x y)) =
16         [M
17           when (i = j)
18             ((M \ (deref i) <- y) \ (replace (successor i) j x y))
19             when (i /= j & M at deref i = x)
20             (M \ (replace (successor i) j x y))
21             when (i /= j & M at deref i /= x)]])
22
23   define [if-empty if-equal if-unequal] := axioms
24
25   (add-axioms theory axioms)
26
27  define replace' := List.replace
28  define M' := ?M':(Memory 'S)
29  define q := ?q:(It 'Z 'S)
30
31  define (correctness-prop r) :=
32    (forall M M' i j x y .
33      (range i j) = SOME r &
34      M' = (M \ (replace i j x y))
35      ==> (collect M' r) = (replace' (collect M r) x y) &
36      forall q . ~ q *in r ==> M' at deref q = M at deref q)
37
38  define correctness := (forall r . correctness-prop r)
39
40  define proof :=
41    method (theorem adapt)
42      let [[get prove chain chain-> chain<-] := (proof-tools adapt theory);
43          [deref *in successor] := (adapt [deref *in successor])]
44      match theorem {
45        (val-of correctness) =>
46        by-induction (adapt theorem) {
47          (stop h:(It 'X 'S)) =>
48          pick-any M:(Memory 'S) M':(Memory 'S) i:(It 'X 'S) j:(It 'X 'S)
49            x:'S y:'S
50          let {A1 := ((range i j) = SOME stop h);
51              A2 := (M' = (M \ (replace i j x y)))}
52          assume (A1 & A2)
53          let {ER1 := (!prove empty-rangel);
54              _ := conclude (i = j)
55                (!chain-> [A1 ==> (i = j) [ER1]]);
56              _ := conclude (M' = M)
57                (!chain
58                  [M' = (M \ (replace i j x y)) [A2]
59                   = M [ (i = j) if-empty]]);
60              B1 := conclude ((collect M' stop h) =
61                            (replace' (collect M stop h) x y))
62                (!chain
63                  [(collect M' stop h)
64                   = (collect M stop h) [(M' = M)]
65                   = nil:(List 'S) [collect.of-stop]
66                   = (replace' nil x y) [List.replace.empty]
67                   = (replace' (collect M stop h) x y)
68                   [collect.of-stop]]);
69              B2 := conclude

```

```

68         (forall ?k:(It 'Z 'S) . ~ ?k *in stop h ==>
69           M' at deref ?k = M at deref ?k)
70       pick-any k:(It 'Z 'S)
71       assume (~ k *in stop h)
72         (!chain [(M' at deref k) = (M at deref k)
73                 [(M' = M)]])
74     (!both B1 B2)
75 | (r as (back r':(Range 'X 'S))) =>
76   let {ind-hyp := (correctness-prop r')}
77   pick-any M:(Memory 'S) M':(Memory 'S) i:(It 'X 'S) j:(It 'X 'S)
78     x:'S y:'S
79   let {A1 := ((range i j) = SOME r);
80         A2 := (M' = (M \ (replace i j x y)))}
81   assume (A1 & A2)
82   let {B1 := ((collect M' r) =
83             (replace' (collect M r) x y));
84         B2 := (forall h . ~ h *in r ==>
85               M' at deref h = M at deref h);
86         NB1 := (!prove nonempty-back1);
87         _ := conclude (i != j)
88             (!chain-> [A1 ==> (i != j) [NB1]]);
89         LB := (!prove range-back);
90         B3 := (!chain->
91               [A1 ==> ((range (successor i) j) = SOME r')
92                       [LB]]);
93         B4 := conclude (i = start r)
94             (!chain->
95               [(range i j)
96                = (SOME r) [A1 A2]
97                = (range (start r)
98                      (finish r)) [range.collapse]
99                ==> (i = start r &
100                   j = finish r) [range.injective]
101                ==> (i = start r) [left-and]]);
102         FNIR := (!prove *in.first-not-in-rest);
103         RR := (!prove *in.range-reduce)}
104   conclude (B1 & B2)
105   (!two-cases
106     assume (M at deref i = x)
107     let
108       {M1 := (M \ (deref i) <- y);
109         C1 :=
110           (!chain
111             [M' = (M \ (replace i j x y)) [A2]
112              = (M1 \ (replace (successor i) j x y))
113              [if-equal]]);
114         (and C2a C2b) :=
115           (!chain->
116             [C1 ==> (B3 & C1) [augment]
117                  ==> ((collect M' r') =
118                      (replace' (collect M1 r') x y) &
119                      forall h . ~ h *in r' ==>
120                        M' at deref h =
121                        M1 at deref h) [ind-hyp]]);
122         C3 := (!chain->
123               [true
124                ==> (~ start r *in r') [FNIR]
125                ==> (~ i *in r') [B4]]);
126         _ := (!sym (M at deref i = x));
127         CU := (!prove collect.unchanged);
128         _ := conclude B1
129             (!combine-equations
130               (!chain
131                 [(collect M' r)
132                  = ((M' at deref i) :: (collect M' r'))
133                    [B4
134                     collect.of-back]
135                  = ((M1 at deref i) ::
136                    (replace' (collect M1 r') x y))
137                  [C2a C2b]

```

```

138 = (y :: (replace' (collect M1 r') x y))
139           [assign.equal]
140 = (y :: (replace' (collect M r') x y))
141           [CU]])
142 (!chain
143 [(replace' (collect M r) x y)
144  = (replace' ((M at deref i) ::
145              (collect M r'))
146            x y) [B4
147              collect.of-back]
148  = (y :: (replace' (collect M r') x y))
149          [List.replace.equal]]);
150 _ := conclude B2
151     pick-any h
152     assume D := (~ h *in r)
153     let {E :=
154         (!chain->
155          [D ==> (~ (deref h =
156                  deref start r |
157                  h *in r')) [*in.of-back]
158          ==> (~ (deref h = deref i |
159                  h *in r')) [B4]
160          ==> (deref h /= deref i &
161              ~ h *in r') [dm]
162          ==> (deref h /= deref i)
163              [left-and]
164          ==> (deref i /= deref h)
165              [sym]]})
166         (!chain->
167          [D ==> (~ h *in r') [RR]
168          ==> (M' at deref h =
169              M1 at deref h) [C2b]
170          ==> (M' at deref h =
171              M at deref h) [E
172                assign.unequal]]})
173     (!both B1 B2)
174     assume (M at deref i /= x)
175     let {M1 := M;
176         C1 := (!chain
177              [M' = (M \ (replace i j x y)) [A2]
178              = (M \ (replace
179                  (successor i) j x y))
180                [if-unequal]]);
181         (and C2a C2b) :=
182         (!chain->
183          [C1 ==> (B3 & C1) [augment]
184          ==> ((collect M' r') =
185              (replace' (collect M r') x y) &
186              forall h . ~ h *in r' ==>
187                  M' at deref h =
188                  M at deref h) [ind-hyp]]);
189         C3 := (!chain->
190              [true ==> (~ start r *in r')
191                  [FNIR]
192              ==> (~ i *in r') [B4]]);
193     _ := (!sym (M at deref i /= x));
194     _ := conclude B1
195         (!combine-equations
196         (!chain
197          [(collect M' r)
198          = ((M' at deref i) ::
199              (collect M' r')) [B4
200                collect.of-back]
201          = ((M at deref i) ::
202              (replace' (collect M r') x y))
203              [C2a C2b]])
204         (!chain
205          [(replace' (collect M r) x y)
206          = (replace' ((M at deref i) ::
207                      (collect M r'))
208                    x y)

```

```

208                                     [B4
209                                     collect.of-back]
210     = ((M at deref i) ::
211         (replace' (collect M r') x y)
212         [List.replace.unequal]));
213     _ := conclude B2
214         pick-any h
215         assume D := (~ h *in r)
216         (!chain->
217           [D ==> (~ h *in r') [RR]
218             ==> (M' at deref h = M at deref h)
219               [C2b]])}
220     (!both B1 B2))
221   } # by-induction
222 } # match theorem
223
224 (add-theorems theory |{[correctness] := proof}|)
225 } # replace
226 } # Forward-Iterator

```