

lib/memory-range/copy-range.ath

```

1 load "forward-iterator"
2 #.....
3
4 extend-module Forward-Iterator {
5
6   declare copy-memory: (S, X, Y) [(It X S) (It X S) (It Y S)] -> (Change S)
7   declare copy: (S, X, Y) [(It X S) (It X S) (It Y S)] -> (It Y S)
8
9   define [M i j k M' k'] :=
10     [?M:(Memory 'S) ?i:(It 'X 'S) ?j:(It 'X 'S) ?k:(It 'Y 'S)
11       ?M':(Memory 'S) ?k':(It 'Y 'S)]
12
13   module copy-memory {
14
15     define axioms :=
16       (fun
17         [(M \ (copy-memory i j k)) =
18           [M
19             ((M \ (deref k) <- (M at (deref i)))
20               \ (copy-memory (successor i) j (successor k)))
21               when (i =/= j)]]])
22
23     define [empty nonempty] := axioms
24
25     (add-axioms theory axioms)
26   }
27
28   module copy {
29
30     define axioms :=
31       (fun
32         [(M \ \ (copy i j k)) =
33           [k
34             ((M \ (deref k) <- (M at (deref i)))
35               \ \ (copy (successor i) j (successor k)))
36               when (i =/= j)]]])
37
38     define [empty nonempty] := axioms
39
40     (add-axioms theory axioms)
41
42     define r1 := ?r1
43
44     define (correctness-prop r) :=
45       (forall i j M k M' k' .
46         (range i j) = SOME r &
47         ~ k *in r &
48         M' = M \ (copy-memory i j k) &
49         k' = M \ \ (copy i j k)
50         ==> exists r1 .
51           (range k k') = SOME r1 &
52           (collect M' r1) = (collect M r) &
53           forall h . ~ h *in r1 ==> M' at deref h = M at deref h)
54
55     define correctness := (forall r . correctness-prop r)
56
57     define proof :=
58       method (theorem adapt)
59         let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
60             [deref *in successor] := (adapt [deref *in successor])}]
61         match theorem {
62           (val-of correctness) =>
63             by-induction (adapt theorem) {
64               (stop q:(It 'X 'S)) =>
65                 pick-any i:(It 'X 'S) j:(It 'X 'S)
66                   M:(Memory 'S) k:(It 'Y 'S)
67                   M':(Memory 'S) k':(It 'Y 'S)
68                 assume A := ((range i j) = SOME stop q &
69                             ~ k *in stop q &

```

```

68         M' = M \ (copy-memory i j k) &
69         k' = M \ \ (copy i j k)
70     conclude goal :=
71     (exists r1 .
72     (range k k') = SOME r1 &
73     (collect M' r1) = (collect M stop q) &
74     forall h . ~ h *in r1 ==> M' at deref h = M at deref h)
75     let {ER1 := (!prove empty-rangel);
76     _ := conclude (i = j)
77     (!chain-> [(range i j)
78     = (SOME stop q) [A]
79     ==> (i = j) [ER1]]);
80     _ := conclude (M' = M)
81     (!chain->
82     [M' = (M \ (copy-memory i j k))
83     = M [A]
84     [copy-memory.empty]]);
85     _ := conclude (k' = k)
86     (!chain->
87     [k' = (M \ \ (copy i j k)) [A]
88     = k [empty]]);
89     _ := (!chain [(start stop k)
90     = k [start.of-stop]
91     = (finish stop k) [finish.of-stop]]);
92     protected := pick-any h
93     assume (~ h *in stop k)
94     (!chain
95     [(M' at deref h)
96     = (M at deref h) [(M' = M)]]);
97     ER := (!prove empty-range);
98     B := (!both
99     (!chain [(range k k')
100     = (range k k) [(k' = k)]
101     = (SOME stop k) [ER]]
102     (!both (!chain
103     [(collect M' stop k)
104     = nil:(List 'S) [collect.of-stop]
105     = (collect M stop q) [collect.of-stop]]
106     protected)))
107     (!chain-> [B ==> goal [existence]])
108 | (r as (back r':(Range 'X 'S))) =>
109     let {ind-hyp := (correctness-prop r')}
110     pick-any i:(It 'X 'S) j:(It 'X 'S) M:(Memory 'S) k:(It 'Y 'S)
111     M':(Memory 'S) k':(It 'Y 'S)
112     let {M1 := (M \ deref k <- M at deref i);
113     A1 := ((range i j) = SOME r);
114     A2 := (~ k *in r);
115     A3 := (M' = M \ (copy-memory i j k));
116     A4 := (k' = M \ \ (copy i j k));
117     assume (A1 & A2 & A3 & A4)
118     conclude
119     goal := (exists r1 .
120     (range k k') = SOME r1 &
121     (collect M' r1) = (collect M r) &
122     forall h . ~ h *in r1 ==>
123     M' at deref h = M at deref h)
124     let {(and B1 B2) :=
125     (!chain->
126     [(range i j)
127     = (SOME r) [A1]
128     = (range (start r)
129     (finish r)) [range.collapse]
130     ==> (i = (start r) &
131     j = (finish r)) [range.injective]]);
132     NB := (!prove nonempty-back);
133     _ := (!chain->
134     [true
135     ==> ((start r) /= (finish r)) [NB]
136     ==> (i /= j) [B1 B2]]);
137     RR := (!prove *in.range-reduce);

```

```

138 CU := (!prove collect.unchanged);
139 B3 := (!chain->
140   [A2 ==> (~ k *in r') [RR]
141   ==> ((collect M1 r') =
142   (collect M r')) [CU]]);
143 B4 := conclude (M' = (M1 \ (copy-memory
144   (successor i) j
145   (successor k))))
146   (!chain
147   [M' = (M \ (copy-memory i j k))
148   [A3]
149   = (M1 \ (copy-memory (successor i) j
150   (successor k)))
151   [copy-memory.nonempty]]);
152 B5 := conclude (k' = (M1 \ \ (copy (successor i) j
153   (successor k))))
154   (!chain
155   [k' = (M \ \ (copy i j k)) [A4]
156   = (M1 \ \ (copy (successor i) j
157   (successor k)))
158   [nonempty]]);
159 LB := (!prove range-back);
160 A1' := (!chain->
161   [A1 ==> ((range (successor i) j) =
162   SOME r') [LB]]);
163 RS2 := (!prove *in.range-shift2);
164 B6 := (!chain->
165   [A2
166   ==> (~ (successor k) *in r') [RS2]
167   ==> (A1' & ~ (successor k) *in r' & B4 & B5)
168   [augment]
169   ==> (exists r1 .
170   (range (successor k) k') = SOME r1 &
171   (collect M' r1) = (collect M1 r') &
172   forall h . ~ h *in r1 ==>
173   M' at deref h =
174   M1 at deref h) [ind-hyp]]))
175 pick-witness r1 for B6 B6-w
176 let {C1 := (!chain->
177   [(range (successor k) k') = SOME r1]
178   ==> ((range k k') = SOME back r1)
179   [LB])};
180 C2 := (!chain->
181   [(range k k')
182   = (SOME back r1) [C1]
183   = (range (start back r1)
184   (finish back r1)) [range.collapse]
185   ==> (k = start back r1 &
186   k' = finish back r1) [range.injective]
187   ==> (k = start back r1) [left-and]]);
188 FNIR := (!prove *in.first-not-in-rest);
189 C3 := (!chain->
190   [true
191   ==> (~ start back r1 *in r1) [FNIR]
192   ==> (~ k *in r1) [C2]]);
193 C4 := conclude ((collect M' r1) = (collect M r'))
194   (!chain
195   [(collect M' r1)
196   = (collect M1 r') [B6-w]
197   = (collect M r') [B3]]);
198 C5 := conclude ((collect M' (back r1)) =
199   (collect M r))
200   (!chain
201   [(collect M' (back r1))
202   = ((M' at deref start back r1)
203   :: (collect M' r1)) [collect.of-back]
204   = ((M' at deref k) :: (collect M' r1))
205   [C2]
206   = ((M1 at deref k) :: (collect M' r1))
207   [C3 B6-w]

```

```

208         = ((M at deref i) :: (collect M r'))
209             [assign.equal C4]
210         = (collect M r) [collect.of-back B1]];
211 C6 := conclude (forall h . ~ h *in back r1 ==>
212             M' at deref h =
213             M at deref h)
214 pick-any h:(It 'X 'S)
215 assume D1 := (~ h *in back r1)
216 let {_ := (!chain-> [D1 ==> (~ h *in r1)
217             [RR]])};
218     D2 :=
219         (!chain->
220             [D1
221             ==> (deref h /= deref start back r1 &
222                 ~ h *in r1) [*in.of-back dm]
223             ==> (deref h /= deref k &
224                 ~ h *in r1) [C2]
225             ==> (deref h /= deref k &
226                 M' at deref h =
227                 M1 at deref h) [B6-w]]);
228     D3 := (!right-and D2);
229     _ := (!sym (!left-and D2));
230     (!chain
231         [(M' at deref h)
232          = (M1 at deref h) [D3]
233          = (M at deref h) [assign.unequal]]);
234     C7 := (!both C1 (!both C5 C6))}
235     (!chain-> [C7 ==> goal [existence]])
236 }
237 }
238
239 (add-theorems theory |{[correctness] := proof}|)
240 } # copy
241 } # Forward-Iterator

```