lib/memory-range/collect-locs.ath

```
load "random-access-iterator"
#.....................................................................
extend-module Random-Access-Iterator {
  define join := List.join
  define in := List.in

  declare collect-locs: (S, X) [(Range X S)] -> (List (Memory.Loc S))

  module collect-locs {
    define [< <=] := [N.< N.<=]

    define axioms :=
     (fun
      [(collect-locs stop h) = nil
       (collect-locs back r) = (deref start back r :: (collect-locs r))])
    define [of-stop of-back] := axioms

    (add-axioms theory axioms)

  define [r''] := [?r'':(Range 'X 'S)]

  define (split-range-prop n) :=
    (forall i j r .
      (range i j) = SOME r & n <= length r
      ==>
      exists r' r'' .
        (range i i + n) = SOME r' &
        (range i + n j) = SOME r'' &
        (collect-locs r) = (collect-locs r') join (collect-locs r''))

  define split-range := (forall n . split-range-prop n)

  define *in-relation :=
    (forall r i . i *in r <==> deref i in collect-locs r)

  define all-*in :=
    (forall n i j r .
      (range i j) = SOME r & n < length r ==> (i + n) *in r)

  define *in-whole-range :=
    (forall n i j k r r' .
      (range i j) = SOME r &
      n < length r &
      (range i i + n) = SOME r' &
      (k *in r' | k = i + n)
      ==> k *in r)

  define *in-whole-range-2 :=
    (forall n i j k r r' .
      (range i j) = SOME r &
      n <= length r &
      (range i + n j) = SOME r' &
      (k *in r' | k = j)
      ==> k *in r | k = j)

define [n0 r0 r0'] := [?n0 ?r0 ?r0']

define proofs :=
  method (theorem adapt)
   let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
        [successor *in predecessor I+N I-N I-I] :=
          (adapt [successor *in predecessor I+N I-N I-I]);
        DAO := (datatype-axioms "Option")}
    match theorem {
      (val-of split-range) =>
      by-induction (adapt theorem) {
        zero =>
```

```
68          pick-any i:(It 'X 'S) j:(It 'X 'S) r:(Range 'X 'S)
69            let {A1 := ((range i j) = SOME r);
70                 A2 := (zero <= length r)}
71             assume (A1 & A2)
72              let {goal := (exists r' r'' .
73                              (range i i + zero) = SOME r' &
74                              (range i + zero j) = SOME r'' &
75                              (collect-locs r) =
76                              (collect-locs r') join (collect-locs r''));
77                   EL := (!prove empty-range);
78                   B1 := (!chain
79                          [(range i i + zero)
80                           = (range i i)                          [I+0]
81                           = (SOME stop i)                        [EL]]);
82                   B2 := (!chain
83                          [(range i + zero j)
84                           = (range i j)                          [I+0]
85                           = (SOME r)                             [A1]]);
86                   B3 := (!sym
87                          (!chain
88                           [((collect-locs stop i) join (collect-locs r))
89                            = (nil join (collect-locs r))       [of-stop]
90                            = (collect-locs r)     [List.join.left-empty]]))}
91               (!chain-> [(B1 & B2 & B3) ==> goal [existence]])
92        | (n as (S n')) =>
93          pick-any i:(It 'X 'S) j:(It 'X 'S) r:(Range 'X 'S)
94           let {A1 := ((range i j) = SOME r);
95                A2 := (n <= length r)}
96            assume (A1 & A2)
97              let {goal := (exists r' r'' .
98                              (range i i + n) = SOME r' &
99                              (range (i + n) j) = SOME r'' &
100                             (collect-locs r) =
101                             (collect-locs r') join (collect-locs r''));
102                  ind-hyp := (split-range-prop n');
103                  B1 := (!chain->
104                         [A2
105                      ==> (exists n0 . length r = S n0)    [N.Less=.S4]])}
106               pick-witness n0 for B1 B1-w
107                let {NL := (!prove nonzero-length);
108                     C1 := (!chain->
109                            [true ==> (S n0 =/= zero)       [N.S-not-zero]
110                                  ==> (length r =/= zero)   [B1-w]
111                                  ==> (exists r0 .
112                                         r = (back r0))     [NL]])}
113                 pick-witness r0 for C1 C1-w
114                  let {LB := (!prove range-back);
115                       D1 := (!chain->
116                              [(range i j)
117                               = (SOME r)                    [A1]
118                               = (SOME (back r0))            [C1-w]
119                               ==> ((range (successor i) j) =
120                                    SOME r0)                 [LB]]);
121                       D2 := (!chain->
122                              [A2 ==> (n <= length back r0) [C1-w]
123                                  ==> (n <= S length r0)    [length.of-back]
124                                  ==> (n' <= length r0)
125                                                        [N.Less=.injective]]);
126                       D3 := (!chain->
127                              [(D1 & D2)
128                               ==>
129                               (exists r0' r'' .
130                                 (range (successor i) (successor i) + n') =
131                                        SOME r0' &
132                                 (range (successor i) + n' j) = SOME r'' &
133                                 (collect-locs r0) =
134                                 (collect-locs r0') join (collect-locs r''))
135                                                        [ind-hyp]])}
136                   pick-witnesses r0' r'' for D3 D3-w
137                     let {D3-w1 := ((range (successor i) (successor i) + n')
```

```
                              = SOME r0');
                    D3-w2 := ((range (successor i) + n' j) = SOME r'');
                    D3-w3 :=
                         ((collect-locs r0) =
                          (collect-locs r0') join (collect-locs r''));
                    E1 := (!chain->
                           [D3-w1
                            ==> ((range (successor i) i + n) =
                                 SOME r0')                    [I+pos]
                            ==> ((range i i + n) =
                                 SOME back r0')               [LB]]);
                    E2 := (!chain->
                           [D3-w2
                            ==> ((range i + n j) =
                                 SOME r'')                    [I+pos]]);
                    E3 :=  let {F1 := (!chain->
                                       [E1 ==> (i = start back r0')
                                               [range.start-back]]);
                                F2 := (!chain->
                                       [(range i j)
                                        = (SOME r)            [A1]
                                        = (SOME (back r0))    [C1-w]
                                        ==> (i = start back r0)
                                                 [range.start-back]]);
                                F3 := (!chain
                                       [(start back r0)
                                        = i                   [F2]
                                        = (start back r0')    [F1]])}
                           (!chain
                            [(collect-locs r)
                             = (collect-locs (back r0))       [C1-w]
                             = ((deref start back r0) ::
                                (collect-locs r0))            [of-back]
                             = ((deref start back r0) ::
                                ((collect-locs r0') join
                                 (collect-locs r'')))         [D3-w3]
                             = (((deref start back r0) ::
                                 (collect-locs r0')) join
                                 (collect-locs r''))
                                            [List.join.left-nonempty]
                             = (((deref start back r0') ::
                                 (collect-locs r0')) join
                                 (collect-locs r''))          [F3]
                             = ((collect-locs back r0') join
                                (collect-locs r''))      [of-back]])}
                  (!chain-> [(E1 & E2 & E3) ==> goal      [existence]])
    }
  | (val-of *in-relation) =>
    by-induction (adapt theorem) {
      (stop h) =>
      pick-any i
        let {B1 := (!chain->
                    [true ==> (~ i *in stop h)        [*in.of-stop]
                          ==> (i *in stop h <==> false) [prop-taut]]);
             B2 := (!chain->
                    [true ==> (~ deref i in nil)      [List.in.empty]
                          ==> (deref i in nil <==> false)
                                                      [prop-taut]])}
          (!chain
           [(i *in stop h)
            <==> false                                [B1]
            <==> (deref i in nil)                     [B2]
            <==> (deref i in (collect-locs stop h))   [of-stop]])
    | (r as (back r')) =>
      let {ind-hyp := (forall i .
                         i *in r' <==> deref i in (collect-locs r'))}
       pick-any i
        (!chain
         [(i *in r) <==> (deref i = deref start r |
                                i *in r')             [*in.of-back]
```

```
208                          <==> (deref i = deref start r |
209                                deref i in (collect-locs r'))
210                                                    [ind-hyp]
211                          <==> (deref i in (deref start r) ::
212                                collect-locs r')       [List.in.nonempty]
213                          <==> (deref i in (collect-locs r))
214                                                    [of-back]])
215        }
216      | (val-of all-*in) =>
217        by-induction (adapt theorem) {
218          zero =>
219          pick-any i:(It 'X 'S) j:(It 'X 'S) r:(Range 'X 'S)
220           let {A1 := ((range i j) = SOME r);
221                A2 := (zero < length r);
222                NL := (!prove nonzero-length)}
223            assume (A1 & A2)
224              let {B1 := (!chain->
225                          [A2
226                           ==> (zero =/= length r)        [N.Less.not-equal]
227                           ==> (length r =/= zero)        [sym]
228                           ==> (exists r0 . r = back r0)  [NL]])}
229               pick-witness r0 for B1 B1-w
230                let {C1 := (!chain->
231                            [(range i j)
232                             = (SOME r)                [A1]
233                             = (SOME back r0)          [B1-w]
234                             ==> (i = start back r0)   [range.start-back]]);
235                     C2 := (!chain->
236                            [(deref i) = (deref start back r0)      [C1]])}
237                 (!chain->
238                  [C2 ==> (C2  | i *in r0)             [alternate]
239                      ==> (i *in back r0)              [*in.of-back]
240                      ==> (i *in r)                    [B1-w]
241                      ==> ((i + zero) *in r)           [I+0]])
242        | (n as (S n')) =>
243         let {ind-hyp := (forall i j r .
244                           (range i j) = SOME r & n' < length r
245                           ==> (i + n') *in r)}
246          pick-any i:(It 'X 'S) j:(It 'X 'S) r:(Range 'X 'S)
247           let {A1 := ((range i j) = SOME r);
248                A2 := (S n' < length r)}
249            assume (A1 & A2)
250             conclude (i + n *in r)
251              let {NL := (!prove nonzero-length);
252                   B1 := (!chain->
253                          [true ==> (zero < n)           [N.Less.zero<S]
254                                ==> (zero < n & A2)      [augment]
255                                ==> (zero < length r)    [N.Less.transitive]
256                                ==> (zero =/= length r)  [N.Less.not-equal]
257                                ==> (length r =/= zero)  [sym]
258                                ==> (exists r0 .
259                                       r = back r0)      [NL]])}
260               pick-witness r0 for B1 B1-w
261                 let {LB := (!prove range-back);
262                      C1 := (!chain->
263                             [A1 ==> ((range i j) = SOME back r0) [B1-w]
264                                 ==> ((range (successor i) j) =
265                                      SOME r0)                   [LB]]);
266                      C2 := (!chain->
267                             [A2 ==> (n < length back r0)         [B1-w]
268                                 ==> (n < S length r0)  [length.of-back]
269                                 ==> (n' < length r0)   [N.Less.injective]]);
270                      RE := (!prove *in.range-expand)}
271                     (!chain->
272                      [(C1 & C2)
273                       ==> (((successor i) + n') *in r0) [ind-hyp]
274                       ==> ((i + n) *in r0)              [I+pos]
275                       ==> ((i + n) *in back r0)         [RE]
276                       ==> ((i + n) *in r)               [B1-w]])
277        }
```

```
278       | (val-of *in-whole-range) =>
279         pick-any n i:(It 'X 'S) j:(It 'X 'S) k:(It 'X 'S)
280                  r:(Range 'X 'S) r':(Range 'X 'S)
281           let {A1 := ((range i j) = SOME r);
282                A2 := (n < length r);
283                A3 := ((range i i + n) = SOME r');
284                A4 := (k *in r' | k = i + n)}
285           assume (A1 & A2 & A3 & A4)
286             let {B1 := (!chain-> [A2 ==> (n <= length r)
287                                          [N.Less=.Implied-by-<]]);
288                  SR := (!prove split-range);
289                  B2 := (!chain->
290                         [(A1 & B1)
291                          ==> (exists r' r'' .
292                                 (range i i + n) = SOME r' &
293                                 (range i + n j) = SOME r'' &
294                                 (collect-locs r) =
295                                 (collect-locs r') join (collect-locs r''))
296                                                        [SR]])}
297               pick-witnesses r1 r2 for B2 B2-w
298                 let {B2-w1 := ((range i i + n) = SOME r1);
299                      B2-w2 := ((range i + n j) = SOME r2);
300                      B2-w3 := ((collect-locs r) =
301                                (collect-locs r1) join (collect-locs r2));
302                      C1 := (!chain->
303                             [(SOME r')
304                              = (range i i + n)           [A3]
305                              = (SOME r1)                 [B2-w1]
306                              ==> (r' = r1)               [DAO]]);
307                     ICL := (!prove *in-relation)}
308                 (!cases A4
309                   (!chain
310                    [(k *in r')
311                     <==> (k *in r1)   [C1]
312                     <==> (deref k in (collect-locs r1))  [ICL]
313                     ==> (deref k in (collect-locs r1) |
314                          deref k in (collect-locs r2))   [alternate]
315                     ==> (deref k in ((collect-locs r1) join
316                                      (collect-locs r2))) [List.in.of-join]
317                     <==> (deref k in (collect-locs r))   [B2-w3]
318                     <==> (k *in r)                       [ICL]])
319                  assume (k = i + n)
320                    let {AI := (!prove all-*in)}
321                      (!chain->
322                       [(A1 & A2) ==> (i + n *in r)       [AI]
323                                  ==> (k *in r)           [(k = i + n)]]))
324       | (val-of *in-whole-range-2) =>
325         pick-any n i:(It 'X 'S) j:(It 'X 'S) k:(It 'X 'S)
326                  r:(Range 'X 'S) r':(Range 'X 'S)
327           let {A1 := ((range i j) = SOME r);
328                A2 := (n <= length r);
329                A3 := ((range i + n j) = SOME r');
330                A4 := (k *in r' | k = j)}
331           assume (A1 & A2 & A3 & A4)
332             let {SR := (!prove split-range);
333                  B2 := (!chain->
334                         [(A1 & A2)
335                          ==> (exists r' r'' .
336                                 (range i i + n) = SOME r' &
337                                 (range i + n j) = SOME r'' &
338                                 (collect-locs r) =
339                                 (collect-locs r') join (collect-locs r''))
340                                                        [SR]])}
341               pick-witnesses r1 r2 for B2 B2-w
342                 let {B2-w1 := ((range i i + n) = SOME r1);
343                      B2-w2 := ((range i + n j) = SOME r2);
344                      B2-w3 := ((collect-locs r) =
345                                (collect-locs r1) join (collect-locs r2));
346                      C1 := (!chain->
347                             [(SOME r')
```

```
                          = (range i + n j)           [A3]
                          = (SOME r2)                 [B2-w2]
                          ==> (r' = r2)               [DAO]]);
                   ICL := (!prove *in-relation)}
               (!cases A4
                (!chain
                 [(k *in r')
                  <==> (k *in r2)   [C1]
                  <==> (deref k in (collect-locs r2))  [ICL]
                  ==> (deref k in (collect-locs r1) |
                       deref k in (collect-locs r2))   [alternate]
                  ==> (deref k in ((collect-locs r1) join
                                   (collect-locs r2))) [List.in.of-join]
                  <==> (deref k in (collect-locs r))   [B2-w3]
                  <==> (k *in r)                       [ICL]
                  ==> (k *in r | k = j)                [alternate]])
                (!chain
                 [(k = j) ==> (k *in r | k = j)        [alternate]]))
     }

  (add-theorems theory |{[split-range *in-relation all-*in *in-whole-range
                          *in-whole-range-2] := proofs}|)

 } # collect-locs
} # Random-Access-Iterator
```