load "random-access-iterator"

extend-module Random-Access-Iterator {
  define join := List.join
  overload <= N <=

  define length1 := (forall r . start r = (finish r) - (length r))
  define length2 := (forall r . length r = (finish r) - (start r))

  define length3 := (forall i j r . (range i j) = SOME r ==> length r = j - i)
  define r' := ?r'; (Range 'X 'S)

  define length4 :=
    (forall i j n r r' r'' .
      (range i j) = SOME r &
      (range i i + n) = SOME r' &
      (range i + n j) = SOME r''
      ==> length r = (length r') + (length r''))

  define (contained-range-prop n) :=
    (forall r i j k .
      (range i j) = SOME r &
      k = i + n &
      n <= length r
      ==> exists r'. (range i k) = SOME r')

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

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

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

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

  method [n0 r0 r0'] :=
    method (theorem adapt)
      let {get prove chain chain-> chain<-} := (proof-tools adapt theory);
      [successor predecessor I+N I-N I-I] :=
      (adapt [successor predecessor I+N I-N I-I])

      match theorem {
      (val-of length1) =>
        by-induction (adapt theorem) {
        (stop j) =>
          conclude (start stop j = (finish stop j) - (length stop j))
            ![combine-equations
            ![chain [(start stop j) = j [start.of-stop]]]
            ![chain [((finish stop j) - (length stop j))
            = (j - zero) [finish.of-stop
            length.of-stop]
            ![I-0])]
          = j
        | (r as (back r')) =>
          conclude (start r = (finish r) - (length r))
            ![combine-equations
            ![chain [(start r) = (predecessor start r') [predecessor.of-start]
            | (r as (back r')) =>
          conclude (start r = (finish r) - (length r))
            ![combine-equations
            ![chain [(start r) = (predecessor start r') [predecessor.of-start]


val-of length2 =>
pick-any r: (Range 'X 'S)
let [RL1 := (!prove length1)]
(chain->
[true
  ==> (start r = ((finish r) - length r)) [RL1]
  ==> ((finish r) - (start r) = length r) [I-I]
  ==> (length r = (finish r) - start r) [sym]]
)

val-of length3 =>
pick-any i; (It 'X 'S) j; (It 'X 'S) r: (Range 'X 'S)
assume A := (range i j) = SOME r
let [B := (!chain->
  (range (start r) (finish r)) = (SOME r) [range.collapse]
  = (range i j) [A]
  ==> (start r = i & finish r = j) [range.injective]]);
RL2 := (!prove length2)
(chain->
[length r] = ((finish r) - start r) [RL2]
= (j - i) [B]]

val-of length4 =>
conclude
(forall i j n r r' r'' .
  (range i j) = SOME r &
  (range i i + n) = SOME r' &
  (range i + n j) = SOME r''
  ==> length r = (length r') + length r'')
pick-any i; (It 'X 'S) j; (It 'X 'S) n r: (Range 'X 'S) r': (Range 'X 'S) r": (Range 'X 'S)
let [A1 := (range i j) = SOME r];
A2 := (range i i + n) = SOME r';
A3 := (range i + n j) = SOME r'';
RL3 := (!prove length3);
IIMN := (!prove I-M-N);
IIC := (!prove I-I-cancellation)
assume (A1 & A2 & A3)
let [k := (i + n);
  B0 := (!chain-> [A1 ==> (length r = j - i) [RL3]]);
  B1 := (!chain->
    [A3 ==> (length r'' = j - k) [RL3]
    ==> (j - k = length r'') [nym]
    ==> (k = j - length r'') [I-I]]);
  B2 := (!chain->
    [A2 ==> (length r' = k - i) [RL3]
    ==> (k - i = length r') [sym]
    ==> (i = k - length r') [I-I]]))
(chain->
[i = (k - length r') [B2]
  = ((j - (length r'')) - length r') [B1]
  = ((j - (length r'') + length r')) [I-M-N]
  ==> (j - i = (length r'')) + length r' [I-I]
  ==> (length r = (length r'') + length r') [B0]
  ==> (length r = (length r') + length r'') [N.Plus.commutative]])

val-of contained-range =>
by-induction (adapt theorem) {
zero =>
pick-any r: (Range 'X 'S) i; (It 'X 'S) j; (It 'X 'S) k: (It 'X 'S)
let [A1 := (range i j) = SOME r];
A2 := (k = i + zero);
A3 := (zero <= length r);
EL := (!prove empty-range)
assume (A1 & A2 & A3)
let {C1 := (!chain \[k = (i + zero) \[A2\]
       = i \[I+0\]\])}
(!chain->
[(range i k)
  = (range i i) \[C1\]
  = (SOME stop i) \[EL\]
=> (exists r' . (range i k) = SOME r') \[existence]\])
| (n as (S n')) =>
pick-any r:(Range 'X 'S) i:(It 'X 'S) j:(It 'X 'S) k:(It 'X 'S)
let {A1 := ((range i j) = SOME r);
A2 := (k = i + n);
A3 := (n <= length r);
goal := (exists r'. (range i k) = SOME r');
NL := (!prove nonzero-length)}
assume (A1 & A2 & A3)
let {B0 := (!chain->
    [A3
    => (exists n0 . length r = S n0) \[N.Less=.S4]\])}
pick-witness n0 for B0 B0-w
let {B := (!chain->
    \[true
    => (S n0 /= zero) \[N.S-not-zero\]
    => (length r /= zero) \[B0-w\]
    => (exists r0 . r = (back r0)) \[NL\]);
LB := (!prove range-back)}
pick-witness r0 for B B-w
let {C0 := (!chain->
    [(range i j)
     = (SOME r) \[A1\]
     = (SOME back r0) \[B-w\]
     => (range (successor i) j) = SOME r0 \[LB\]);
C1 := (!chain [k = (i + n) \[A2\]
    = ((successor i) + n') \[I+pos\]);
C2 := (!chain->
    [A3
    => (n <= length back r0) \[B-w\]
    => (n <= S length r0) \[length.of-back]
    => (n' <= length r0) \[N.Less=.injective]\]);
C3 := (!chain->
    [(C0 & C1 & C2)
     => (exists r'. (range (successor i) k) = SOME r')
     \[ind-hyp\]])}
pick-witness r' for C3 C3-w
(!chain->
  [C3-w
  => (range i k) = SOME (back r') \[LB\]
  => goal \[existence]\])
| (val-of collect-split-range) =>
by-induction (adapt theorem) {
    zero =>
pick-any i:(It 'X 'S) j:(It 'X 'S) r:(Range 'X 'S)
let {A1 := ((range i j) = SOME r);
A2 := (zero <= length r))
assume (A1 & A2)
let {goal := (exists r' r'') .
    (range i i + zero) = SOME r' &
    (range i + zero j) = SOME r'' &
    (forall M . (collect M r) =
      (collect M r') join (collect M r'')));
EL := (!prove empty-range);
B1 := (!chain
    [(range i i + zero)
     = (range i i) \[I+0\]
     = (SOME stop i) \[empty-range]\]);
B2 := (!chain

[(range i + zero j)
  = (range i j) [I+0]
  = (SOME r) [A1]);

B3 := pick-any M

(!sym (!chain

[({(collect M stop i) join (collect M r))
  = (nil join (collect M r))
  = (collect.of-stop)
  = (collect M r) [List.join.left-empty]]))

(!chain-> [(B1 & B2 & B3) ==> goal [existence]])

| (n as (S n')) =>

pick-any i:(It 'X 'S) j:(It 'X 'S) r:(Range 'X 'S)

let {A1 := ((range i j) = SOME r);
  A2 := (S n' <= length r)}

assume (A1 & A2)

let {goal := (exists r (S n' r)

(range i i + n) = SOME r &
  (range (i + n) j) = SOME r'' &
  (forall M .
    (collect M r) =
    (collect M r') join (collect M r'')));

ind-hyp := (collect-split-range-prop n');

B1 := (!chain->

[A2
  ==> (exists n0 . length r = S n0)
  [N.Less=.S4]])

pick-witness n0 for B1 B1-w

let {NL := (!prove nonzero-length);
  C1 := (!chain->

  [true
  ==> (S n0 /= zero) [N.S-not-zero]
  ==> (length r /= zero) [B1-w]
  ==> (exists r0 . r = (back r0)) [NL]])

pick-witness r0 for C1 C1-w

let {LB := (!prove range-back);
  D1 := (!chain->

  [(range i j)
    = (SOME r) [A1]
    = (SOME back r0) [C1-w]
    ==> ((range (successor i) i + n) =
      SOME r0) [LB]);

D2 := (!chain->

[A2 ==> (n <= length back r0) [C1-w]
  ==> (n <= S length r0) [length.of-back]
  ==> (n' <= length r0) [N.Less=.injective]];)

D3 := (!chain->

[(D1 & D2)
  ==> ((exists r0' r'' .
      (range (successor i) (successor i) + n') =
      SOME r0' &
      (range (successor i) + n' j) = SOME r'' &
      (forall M .
        (collect M r0) =
        (collect M r0') join (collect M r''))))
  [ind-hyp]]);

pick-witnesses r0' r'' for D3 D3-w

let {D3-w1 := ((range (successor i) (successor i) + n') = SOME r0');
  D3-w2 := ((range (successor i) + n' j) = SOME r'');
  D3-w3 := (forall M .
    (collect M r0) =
    (collect M r0') join (collect M 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 := pick-any M
let SB := (!prove range.start-back);
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
= (start back r0') [F2]]
[!chain
((collect M r)
= (collect M (back r0)) [C1-w]
= ((M at deref start back r0) ::
(collect M r0)) [collect.of-back]
= (((M at deref start back r0) ::
(collect M r0')) join
(collect M r'')) [D3-w3]
= (((M at deref start back r0) ::
(collect M r'))
join (collect M r''))
[List.join.left-nonempty]
= (((M at deref start back r0') ::
(collect M r0'))
join (collect M r')) [F3]
(join (collect M r')) [collect.of-back]])
[!chain-> [(E1 & E2 & E3) ==> goal [existence]]]
}
}
(add-theorems theory |[length1 length2 length3 length4 contained-range
collect-split-range] := proofs|)}
| # Random-Access-Iterator