

lib/memory-range/random-access-iterator.ath

```

1 load "bidirectional-iterator"
2 load "nat-minus"
3 #.....
4
5 module Random-Access-Iterator {
6   open Bidirectional-Iterator
7   overload + N.+
8
9   declare I+N: (X, S) [(It X S) N] -> (It X S) [+]
10  declare I-N: (X, S) [(It X S) N] -> (It X S) [-]
11  declare I-I: (X, S) [(It X S) (It X S)] -> N [-]
12
13  define [m n] := [?m:N ?n:N]
14
15  define I+0 := (forall i . i + zero = i)
16  define I+pos := (forall i n . i + (S n) = (successor i) + n)
17  define I-0 := (forall i . i - zero = i)
18  define I-pos := (forall i n . i - (S n) = predecessor (i - n))
19  define I-I := (forall i j n . i - j = n <==> j = i - n)
20
21  define theory :=
22    (make-theory ['Bidirectional-Iterator] [I+0 I+pos I-0 I-pos I-I])
23
24 #.....
25  define I-I-self := (forall i . i - i = zero)
26  define I+N-cancellation := (forall n i . (i + n) - n = i)
27  define I-I-cancellation := (forall n i . (i + n) - i = n)
28  define successor-in :=
29    (forall n i . successor (i + n) = (successor i) + n)
30  define I-M-N :=
31    (forall n m i . (i - m) - n = i - (m + n))
32
33  define proofs :=
34    method (theorem adapt)
35      let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
36          [successor predecessor I+N I-N I-I] :=
37            (adapt [successor predecessor I+N I-N I-I])}
38      match theorem {
39        (val-of I-I-self) =>
40          pick-any i:(It 'X 'S)
41            (!chain->
42              [(i - i = zero) <== (i = i - zero) [I-I]
43                <== (i = i) [I-0]])
44        | (val-of I+N-cancellation) =>
45          by-induction (adapt theorem) {
46            zero =>
47              pick-any i:(It 'X 'S)
48                (!chain->
49                  [(i + zero) - zero
50                    = (i - zero) [I+0]
51                    = i [I-0]])
52            | (n as (S n')) =>
53              let {ind-hyp := (forall i . (i + n') - n' = i)}
54                pick-any i:(It 'X 'S)
55                  (!chain->
56                    [(i + n) - n
57                      = ((successor i) + n') - n [I+pos]
58                      = (predecessor ((successor i) + n') - n') [I-pos]
59                      = (predecessor successor i) [ind-hyp]
60                      = i [predecessor.of-successor]])
61              }
62        | (val-of I-I-cancellation) =>
63          let {IC := (!prove I+N-cancellation)}
64            pick-any n i:(It 'X 'S)
65              (!chain->
66                [(i = i)
67                  ==> (i = (i + n) - n) [IC]

```

```

68     ==> ((i + n) - i = n)                                [I-I]]
69 # Following version fails when tracing rewriting, but works when not
70 # tracing rewriting!
71 | (val-of I-I-cancellation) =>
72   let {IC := (!prove I+N-cancellation)}
73     pick-any n i:(It 'X 'S)
74     (!chain->
75       [(i + n) - i = n] <== (i = (i + n) - n)           [I-I]
76         <== (i = i)                                       [IC]))
77 | (val-of successor-in) =>
78   by-induction (adapt theorem) {
79     zero => pick-any i
80       (!chain [(successor (i + zero))
81               = (successor i)                             [I+0]
82               = ((successor i) + zero)                   [I+0]])
83 | (n as (S n')) =>
84   let {ind-hyp :=
85     (forall i . successor (i + n') = (successor i) + n')}
86     pick-any i
87     (!chain
88       [(successor (i + n))
89        = (successor ((successor i) + n'))               [I+pos]
90        = ((successor successor i) + n')                 [ind-hyp]
91        = ((successor i) + n)                             [I+pos]])
92   }
93 | (val-of I-M-N) =>
94   by-induction (adapt theorem) {
95     zero =>
96       pick-any m:N i:(It 'X 'S)
97       (!chain
98         [(i - m) - zero]
99         = (i - m)                                       [I-0]
100        = (i - (m + zero))                               [N.Plus.right-zero])
101 | (n as (S n')) =>
102   let {ind-hyp := (forall ?m ?i .
103     (?i:(It 'X 'S) - ?m:N) - n' =
104     ?i:(It 'X 'S) - (?m:N + n'))}
105     pick-any m:N i:(It 'X 'S)
106     (!combine-equations
107       (!chain
108         [(i - m) - n]
109         = (predecessor ((i - m) - n'))                 [I-pos]
110         = (predecessor (i - (m + n'))))                 [ind-hyp]])
111       (!chain
112         [(i - (m + n))
113          = (i - S (m + n'))                             [N.Plus.right-nonzero]
114          = (predecessor (i - (m + n'))))                 [I-pos]]))
115   }
116 }
117
118 (add-theorems theory |{[I-I-self I+N-cancellation I-I-cancellation
119   successor-in I-M-N] := proofs}|)
120 } # Random-Access-Iterator

```