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

```
load "bidirectional-iterator"
load "nat-minus"
#..................................................................

module Random-Access-Iterator {
  open Bidirectional-Iterator
  overload + N.+

  declare I+N: (X, S) [(It X S) N] -> (It X S) [+]
  declare I-N: (X, S) [(It X S) N] -> (It X S) [-]
  declare I-I: (X, S) [(It X S) (It X S)] -> N [-]

  define [m n] := [?m:N ?n:N]

  define I+0 := (forall i . i + zero = i)
  define I+pos := (forall i n . i + (S n) = (successor i) + n)
  define I-0 := (forall i . i - zero = i)
  define I-pos := (forall i n . i - (S n) = predecessor (i - n))
  define I-I := (forall i j n . i - j = n <==> j = i - n)

  define theory :=
    (make-theory ['Bidirectional-Iterator] [I+0 I+pos I-0 I-pos I-I])

#..................................................................
  define I-I-self := (forall i . i - i = zero)
  define I+N-cancellation := (forall n i . (i + n) - n = i)
  define I-I-cancellation := (forall n i . (i + n) - i = n)
  define successor-in :=
    (forall n i . successor (i + n) = (successor i) + n)
  define I-M-N :=
    (forall n m i .(i - m) - n = i - (m + n))

define proofs :=
  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 I-I-self) =>
      pick-any i:(It 'X 'S)
        (!chain->
         [(i - i = zero) <== (i = i - zero)              [I-I]
                         <== (i = i)                     [I-0]])
    | (val-of I+N-cancellation) =>
      by-induction (adapt theorem) {
        zero =>
        pick-any i:(It 'X 'S)
          (!chain->
           [((i + zero) - zero)
            = (i - zero)                                 [I+0]
            = i                                          [I-0]])
      | (n as (S n')) =>
        let {ind-hyp := (forall i . (i + n') - n' = i)}
         pick-any i:(It 'X 'S)
          (!chain->
           [((i + n) - n)
            = (((successor i) + n') - n)                 [I+pos]
            = (predecessor (((successor i) + n') - n'))  [I-pos]
            = (predecessor successor i)                  [ind-hyp]
            = i                         [predecessor.of-successor]])
      }
    | (val-of I-I-cancellation) =>
      let {IC := (!prove I+N-cancellation)}
       pick-any n i:(It 'X 'S)
        (!chain->
         [(i = i)
          ==> (i = (i + n) - n)                          [IC]
```

```
          ==> ((i + n) - i = n)                          [I-I]])
# Following version fails when tracing rewriting, but works when not
# tracing rewriting!
    | (val-of I-I-cancellation) =>
      let {IC := (!prove I+N-cancellation)}
       pick-any n i:(It 'X 'S)
        (!chain->
         [((i + n) - i = n) <== (i = (i + n) - n)        [I-I]
                            <== (i = i)                  [IC]])
    | (val-of successor-in) =>
      by-induction (adapt theorem) {
        zero => pick-any i
                  (!chain [(successor (i + zero))
                          = (successor i)                [I+0]
                          = ((successor i) + zero)       [I+0]])
      | (n as (S n')) =>
        let {ind-hyp :=
              (forall i . successor (i + n') = (successor i) + n')}
         pick-any i
          (!chain
           [(successor (i + n))
            = (successor ((successor i) + n'))           [I+pos]
            = ((successor successor i) + n')             [ind-hyp]
            = ((successor i) + n)                        [I+pos]])
      }
    | (val-of I-M-N) =>
      by-induction (adapt theorem) {
        zero =>
        pick-any m:N i:(It 'X 'S)
          (!chain
           [((i - m) - zero)
            = (i - m)                                    [I-0]
            = (i - (m + zero))            [N.Plus.right-zero]])
      | (n as (S n')) =>
        let {ind-hyp := (forall ?m ?i .
                          (?i:(It 'X 'S) - ?m:N) - n' =
                          ?i:(It 'X 'S) - (?m:N + n'))}
         pick-any m:N i:(It 'X 'S)
          (!combine-equations
           (!chain
            [((i - m) - n)
             = (predecessor ((i - m) - n'))              [I-pos]
             = (predecessor (i - (m + n')))              [ind-hyp]])
           (!chain
            [(i - (m + n))
             = (i - S (m + n'))          [N.Plus.right-nonzero]
             = (predecessor (i - (m + n')))              [I-pos]]))
      }
    }

 (add-theorems theory |{[I-I-self I+N-cancellation I-I-cancellation
                        successor-in I-M-N] := proofs}|)
} # Random-Access-Iterator
```