lib/memory-range/reverse-range.ath

```
#..........................................................................

load "bidirectional-iterator"

#..........................................................................

extend-module Bidirectional-Iterator {

  module Reversing {
    define join := List.join
    define reverse := List.reverse

    declare reverse-iterator: (X, S) [(It X S)] -> (It (It X S) S)
    declare reverse-range: (X, S) [(Range X S)] -> (Range (It X S) S)
    declare base-iterator: (X, S) [(It (It X S) S)] -> (It X S)
    declare base-range: (X, S) [(Range (It X S) S)] -> (Range X S)

    define deref-reverse :=
      (forall i . deref reverse-iterator i = deref predecessor i)

    define *reverse-in :=
      (forall i r . (reverse-iterator i) *in (reverse-range r) <==>
                    (predecessor i) *in r)

    define base-reverse-range :=
      (forall r . base-range reverse-range r = r)

    define reverse-base-range :=
      (forall r . reverse-range base-range r = r)

    define reverse-of-range :=
      (forall i j r .
        (range (reverse-iterator j) (reverse-iterator i)) = SOME r
        <==> (range i j) = SOME base-range r)

    define reverse-base :=
      (forall i . reverse-iterator base-iterator i = i)

    define collect-reverse-stop :=
      (forall M i . (collect M (reverse-range stop i)) = nil)

    define collect-reverse-back :=
      (forall M r .
        (collect M reverse-range back r) =
        (collect M reverse-range r) join ((M at deref start back r) :: nil))

    (add-axioms theory [deref-reverse *reverse-in base-reverse-range
                        reverse-base-range reverse-of-range reverse-base
                        collect-reverse-stop collect-reverse-back])
#..........................................................................

define reverse-range-reverse :=
  (forall i j r .
    (range (reverse-iterator j)
           (reverse-iterator i)) = SOME (reverse-range r)
    <==> (range i j) = SOME r)

define collect-reverse :=
  (forall r M .
    (collect M reverse-range r) = reverse (collect M r))

define [M' r'] := [?M':(Memory 'S) ?r':(Range (It 'Y 'S) 'S)]

define collect-reverse-corollary :=
  (forall M r M' r' .
    (collect M' r') = (collect M reverse-range r)
    ==> (collect M' base-range r') = (collect M r))
```

```

define proofs :=
 method (theorem adapt)
  let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
       deref := (adapt deref)}
    match theorem {
      (val-of reverse-range-reverse) =>
      pick-any i:(It 'X 'S) j:(It 'X 'S) r:(Range 'X 'S)
        (!chain
         [((range (reverse-iterator j) (reverse-iterator i)) =
           SOME (reverse-range r))
          <==> ((range i j) = SOME (base-range (reverse-range r)))
                                                    [reverse-of-range]
          <==> ((range i j) = SOME r)              [base-reverse-range]])
    | (val-of collect-reverse) =>
      by-induction (adapt theorem) {
        (stop i) =>
        pick-any M
          (!combine-equations
           (!chain->
            [(collect M (reverse-range (stop i)))
             = nil                                  [collect-reverse-stop]])
           (!chain
            [(reverse (collect M (stop i)))
             = (reverse nil)                        [collect.of-stop]
             = nil                                  [List.reverse.empty]]))
      | (r as (back r')) =>
        let {ind-hyp := (forall M .
                          (collect M reverse-range r') =
                          reverse (collect M r'))}
         pick-any M
          (!combine-equations
           (!chain
            [(collect M reverse-range r)
             = ((collect M reverse-range r')
                join
                ((M at deref start r) :: nil))    [collect-reverse-back]
             = ((reverse (collect M r'))
                join
                ((M at deref start r) :: nil))    [ind-hyp]])
           (!chain
            [(reverse (collect M r))
             = (reverse (M at deref start r
                         :: (collect M r'))        [collect.of-back]
             = ((reverse (collect M r'))
                join
                ((M at deref start r) :: nil))    [List.reverse.nonempty]]))
       }
    | (val-of collect-reverse-corollary) =>
       pick-any M:(Memory 'S) r:(Range 'X 'S)
                M':(Memory 'S) r':(Range (It 'Y 'S) 'S)
        assume A := ((collect M' r') = (collect M (reverse-range r)))
         let {CR := (!prove collect-reverse)}
          (!chain
           [(collect M' (base-range r'))
          = (reverse reverse (collect M' (base-range r')))
                                                    [List.reverse.of-reverse]
          = (reverse (collect M' reverse-range base-range r'))
                                                    [CR]
          = (reverse (collect M' r'))              [reverse-base-range]
          = (reverse (collect M reverse-range r))  [A]
          = (reverse reverse (collect M r))        [CR]
          = (collect M r)                          [List.reverse.of-reverse]])
    }

  (add-theorems theory |{[reverse-range-reverse collect-reverse
                          collect-reverse-corollary] := proofs}|)
  } # Reversing
} # Bidirectional-Iterator
```