lib/memory-range/copy-range-backward.ath

```
load "copy-range"
load "reverse-range"
#..................................................................

extend-module Bidirectional-Iterator {
  open Reversing

  declare copy-memory-backward: (S, X, Y) [(It X S) (It X S) (It Y S)] ->
                                           (Change S)
  declare copy-backward: (S, X, Y) [(It X S) (It X S) (It Y S)] -> (It Y S)

  module copy-memory-backward {
    define [def] :=
      (fun
       [(M \ (copy-memory-backward i j k)) =
               (M \ (copy-memory (reverse-iterator j)
                                 (reverse-iterator i)
                                 (reverse-iterator k)))])
  }

  module copy-backward {
    define [def] :=
      (fun
       [(M \\ (copy-backward i j k)) =
                 (base-iterator (M \\ (copy (reverse-iterator j)
                                             (reverse-iterator i)
                                             (reverse-iterator k))))])
  }

  (add-axioms theory [copy-memory-backward.def
                      copy-backward.def])

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

extend-module copy-backward {
define [r r'] := [?r:(Range 'X 'S) ?r':(Range 'Y 'S)]

define correctness :=
  (forall r i j M k M' k' .
    (range i j) = (SOME r) &
    ~ (predecessor k) *in r &
    M' = (M \ (copy-memory-backward i j k)) &
    k' = (M \\ (copy-backward i j k))
    ==> exists r' .
          (range k' k) = SOME r' &
          (collect M' r') = (collect M r) &
          forall h . ~ h *in r' ==> M' at deref h = M at deref h)

define proof :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
         [deref *in successor predecessor] :=
           (adapt [deref *in successor predecessor])}
    match theorem {
      (val-of correctness) =>
      pick-any r:(Range 'X 'S) i:(It 'X 'S) j:(It 'X 'S)
               M:(Memory 'S) k:(It 'Y 'S)
               M':(Memory 'S) k':(It 'Y 'S)
        let {A1 := ((range i j) = SOME r);
             A2 := (~ (predecessor k) *in r);
             A3 := (M' = (M \ (copy-memory-backward i j k)));
             A4 := (k' = (M \\ (copy-backward i j k)));
             goal := (exists r' .
                        (range k' k) = SOME r' &
                        (collect M' r') = (collect M r) &
                        forall h . ~ h *in r' ==>
                           M' at deref h = M at deref h)}
```

```
        assume (A1 & A2 & A3 & A4)
         let {ri := reverse-iterator;
              RLR := (!prove reverse-range-reverse);
              B1 := (!chain->
                     [A1 ==> ((range (ri j) (ri i)) =
                              SOME reverse-range r)      [RLR]]);
              B2 := (!chain->
                     [A2
                  ==> (~ (ri k) *in reverse-range r)     [*reverse-in]]);
              B3 := (!chain
                     [M'
                   = (M \ (copy-memory-backward i j k))  [A3]
                   = (M \ (copy-memory (ri j) (ri i) (ri k)))
                                            [copy-memory-backward.def]]);
              B4 := (!chain
                     [(ri k')
                    = (ri (M \\ (copy-backward i j k)))  [A4]
                    = (ri (base-iterator
                           (M \\ (copy (ri j) (ri i) (ri k)))))
                                            [copy-backward.def]
                    = (M \\ (copy (ri j) (ri i) (ri k))) [reverse-base]]);
               CC := (!prove copy.correctness);
               B5 := (!chain->
                     [(B1 & B2 & B3 & B4)
                  ==> (exists r' .
                        (range (ri k) (ri k')) = SOME r' &
                        (collect M' r') = (collect M reverse-range r) &
                        forall h .
                          ~ h *in r' ==>
                          M' at deref h = M at deref h)  [CC]])}
          pick-witness r' for B5
           let {B5-w1 := ((range (ri k) (ri k')) = SOME r');
                B5-w2 := ((collect M' r') =
                          (collect M (reverse-range r)));
                B5-w3 := (forall h .
                           ~ h *in r' ==>
                           M' at deref h = M at deref h);
                C1 := (!chain->
                       [B5-w1
                    ==> ((range k' k) = SOME base-range r')
                                             [reverse-of-range]]);
                CRC := (!prove collect-reverse-corollary);
                C2 := (!chain->
                       [B5-w2
                    ==> ((collect M' base-range r') = (collect M r))
                                             [CRC]]);
                C3 :=
                 conclude (forall h . ~ h *in base-range r' ==>
                                        M' at deref h = M at deref h)
                  pick-any h
                    (!chain
                     [(~ h *in base-range r')
                      ==> (~ predecessor successor h *in
                                 base-range r')   [predecessor.of-successor]
                      ==> (~ (reverse-iterator successor h) *in
                                 reverse-range base-range r')
                                             [*reverse-in]
                      ==> (~ (reverse-iterator successor h) *in r')
                                             [reverse-base-range]
                      ==> (M' at deref reverse-iterator successor h =
                           M at deref reverse-iterator successor h)
                                             [B5-w3]
                      ==> (M' at deref predecessor successor h =
                           M at deref predecessor successor h)
                                             [deref-reverse]
                      ==> (M' at deref h = M at deref h)
                                             [predecessor.of-successor]])}
             (!chain->
              [(C1 & C2 & C3) ==> goal        [existence]])
    }
```

```
  (add-theorems theory |{[correctness] := proof}|)
 } # close module copy-backward
} # close module Bidirectional-Iterator
```