lib/memory-range/copy-range.ath

```
load "forward-iterator"
#...............................................................

extend-module Forward-Iterator {

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

  define [M i j k M' k'] :=
         [?M:(Memory 'S) ?i:(It 'X 'S) ?j:(It 'X 'S) ?k:(It 'Y 'S)
          ?M':(Memory 'S) ?k':(It 'Y 'S)]

  module copy-memory {

    define axioms :=
      (fun
       [(M \ (copy-memory i j k)) =
          [M                              when (i = j)
           ((M \ (deref k) <- (M at (deref i)))
                   \ (copy-memory (successor i) j (successor k)))
                                          when (i =/= j)]])
    define [empty nonempty] := axioms

    (add-axioms theory axioms)
  }

  module copy {

    define axioms :=
      (fun
       [(M \\ (copy i j k)) =
          [k                              when (i = j)
           ((M \ (deref k) <- (M at (deref i)))
                   \\ (copy (successor i) j (successor k)))
                                          when (i =/= j)]])
    define [empty nonempty] := axioms

    (add-axioms theory axioms)

define r1 := ?r1

define (correctness-prop r) :=
  (forall i j M k M' k' .
    (range i j) = SOME r &
    ~ k *in r &
    M' = M \ (copy-memory i j k) &
    k' = M \\ (copy i j k)
    ==> exists r1 .
          (range k k') = SOME r1 &
          (collect M' r1) = (collect M r) &
          forall h . ~ h *in r1 ==> M' at deref h = M at deref h)

define correctness := (forall r . correctness-prop r)

define proof :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
         [deref *in successor] := (adapt [deref *in successor])}
    match theorem {
      (val-of correctness) =>
      by-induction (adapt theorem) {
        (stop q:(It 'X 'S)) =>
        pick-any i:(It 'X 'S) j:(It 'X 'S)
                 M:(Memory 'S) k:(It 'Y 'S)
                 M':(Memory 'S) k':(It 'Y 'S)
          assume A := ((range i j) = SOME stop q &
                       ~ k *in stop q &
```

```
                       M' = M \ (copy-memory i j k) &
                       k' = M \\ (copy i j k))
            conclude goal :=
                (exists r1 .
                  (range k k') = SOME r1 &
                  (collect M' r1) = (collect M stop q) &
                  forall h . ~ h *in r1 ==> M' at deref h = M at deref h)
              let {ER1 := (!prove empty-range1);
                   _ := conclude (i = j)
                          (!chain-> [(range i j)
                                     = (SOME stop q)  [A]
                                     ==> (i = j)      [ER1]]);
                   _ := conclude (M' = M)
                          (!chain->
                           [M' = (M \ (copy-memory i j k))
                                                      [A]
                               = M                    [copy-memory.empty]]);
                   _ := conclude (k' = k)
                         (!chain->
                          [k' = (M \\ (copy i j k))   [A]
                              = k                     [empty]]);
                   _ := (!chain [(start stop k)
                                 = k                  [start.of-stop]
                                 = (finish stop k)    [finish.of-stop]]);
                   protected := pick-any h
                                  assume (~ h *in stop k)
                                    (!chain
                                     [(M' at deref h)
                                     = (M at deref h) [(M' = M)]]);
                   ER := (!prove empty-range);
                   B := (!both
                         (!chain [(range k k')
                                  = (range k k)       [(k' = k)]
                                  = (SOME stop k)     [ER]])
                         (!both (!chain
                                 [(collect M' stop k)
                                = nil:(List 'S)       [collect.of-stop]
                                = (collect M stop q)  [collect.of-stop]])
                                protected))}
              (!chain-> [B ==> goal                   [existence]])
        | (r as (back r':(Range 'X 'S))) =>
          let {ind-hyp := (correctness-prop r')}
           pick-any i:(It 'X 'S) j:(It 'X 'S) M:(Memory 'S) k:(It 'Y 'S)
                    M':(Memory 'S) k':(It 'Y 'S)
             let {M1 := (M \ deref k <- M at deref i);
                  A1 := ((range i j) = SOME r);
                  A2 := (~ k *in r);
                  A3 := (M' = M \ (copy-memory i j k));
                  A4 := (k' = M \\ (copy i j k))}
              assume (A1 & A2 & A3 & A4)
               conclude
                 goal := (exists r1 .
                           (range k k') = SOME r1 &
                           (collect M' r1) = (collect M r) &
                           forall h . ~ h *in r1 ==>
                                      M' at deref h = M at deref h)
                 let {(and B1 B2) :=
                         (!chain->
                          [(range i j)
                           = (SOME r)                 [A1]
                           = (range (start r)
                                    (finish r))       [range.collapse]
                           ==> (i = (start r) &
                                j = (finish r))       [range.injective]]);
                      NB := (!prove nonempty-back);
                      _ := (!chain->
                             [true
                          ==> ((start r) =/= (finish r))   [NB]
                          ==> (i =/= j)                    [B1 B2]]);
                      RR := (!prove *in.range-reduce);
```

```
     CU := (!prove collect.unchanged);
     B3 := (!chain->
            [A2 ==> (~ k *in r')        [RR]
                ==> ((collect M1 r') =
                     (collect M r'))    [CU]]);
     B4 := conclude (M' = (M1 \ (copy-memory
                                   (successor i) j
                                   (successor k))))
              (!chain
               [M' = (M \ (copy-memory i j k))
                                        [A3]
                   = (M1 \ (copy-memory (successor i) j
                                        (successor k)))
                              [copy-memory.nonempty]]);
     B5 := conclude (k' = (M1 \\ (copy (successor i) j
                                       (successor k))))
             (!chain
              [k' = (M \\ (copy i j k)) [A4]
                  = (M1 \\ (copy (successor i) j
                                  (successor k)))
                                        [nonempty]]);
     LB := (!prove range-back);
     A1' := (!chain->
             [A1 ==> ((range (successor i) j) =
                      SOME r')          [LB]]);
     RS2 := (!prove *in.range-shift2);
     B6 := (!chain->
            [A2
         ==> (~ (successor k) *in r')   [RS2]
         ==> (A1' & ~ (successor k) *in r' & B4 & B5)
                                        [augment]
         ==> (exists r1 .
              (range (successor k) k') = SOME r1 &
              (collect M' r1) = (collect M1 r') &
              forall h . ~ h *in r1 ==>
                         M' at deref h =
                         M1 at deref h) [ind-hyp]])}
pick-witness r1 for B6 B6-w
  let {C1 := (!chain->
              [((range (successor k) k') = SOME r1)
               ==> ((range k k') = SOME back r1)
                                        [LB]]);
       C2 := (!chain->
              [(range k k')
             = (SOME back r1)           [C1]
             = (range (start back r1)
                      (finish back r1)) [range.collapse]
             ==> (k = start back r1 &
                  k' = finish back r1)  [range.injective]
               ==> (k = start back r1)  [left-and]]);
       FNIR := (!prove *in.first-not-in-rest);
       C3 := (!chain->
              [true
           ==> (~ start back r1 *in r1) [FNIR]
           ==> (~ k *in r1)             [C2]]);
       C4 := conclude ((collect M' r1) = (collect M r'))
               (!chain
                [(collect M' r1)
                 = (collect M1 r')      [B6-w]
                 = (collect M r')       [B3]]);
       C5 := conclude ((collect M' (back r1)) =
                       (collect M r))
               (!chain
                [(collect M' (back r1))
                 = ((M' at deref start back r1)
                    :: (collect M' r1)) [collect.of-back]
                 = ((M' at deref k) :: (collect M' r1))
                                        [C2]
                 = ((M1 at deref k) :: (collect M' r1))
                                        [C3 B6-w]
```

```
                                  = ((M at deref i) :: (collect M r'))
                                                        [assign.equal C4]
                                  = (collect M r)   [collect.of-back B1]]);
                        C6 := conclude (forall h . ~ h *in back r1 ==>
                                                   M' at deref h =
                                                   M at deref h)
                               pick-any h:(It 'X 'S)
                                assume D1 := (~ h *in back r1)
                                 let {_ := (!chain-> [D1 ==> (~ h *in r1)
                                                         [RR]]);
                                      D2 :=
                                       (!chain->
                                        [D1
                                     ==> (deref h =/= deref start back r1 &
                                          ~ h *in r1)    [*in.of-back dm]
                                     ==> (deref h =/= deref k &
                                          ~ h *in r1)    [C2]
                                     ==> (deref h =/= deref k &
                                          M' at deref h =
                                          M1 at deref h) [B6-w]]);
                                      D3 := (!right-and D2);
                                      _ := (!sym (!left-and D2))}
                                 (!chain
                                  [(M' at deref h)
                                 = (M1 at deref h)       [D3]
                                 = (M at deref h)        [assign.unequal]]);
                        C7 := (!both C1 (!both C5 C6))}
                   (!chain-> [C7 ==> goal                [existence]])
      }
    }

  (add-theorems theory |{[correctness] := proof}|)
 } # copy
} # Forward-Iterator
```