lib/memory-range/replace-range.ath

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

load "forward-iterator"

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

extend-module Forward-Iterator {

  declare replace: (S, X) [(It X S) (It X S) S S] -> (Memory.Change S)

  module replace {

  define axioms :=
    (fun
     [(M \ (replace i j x y)) =
         [M                       when (i = j)
          ((M \ (deref i) <- y) \ (replace (successor i) j x y))
                                  when (i =/= j & M at deref i = x)
          (M \ (replace (successor i) j x y))
                                  when (i =/= j & M at deref i =/= x)]])
  define [if-empty if-equal if-unequal] := axioms

  (add-axioms theory axioms)

define replace' := List.replace
define M' := ?M':(Memory 'S)
define q := ?q:(It 'Z 'S)

define (correctness-prop r) :=
  (forall M M' i j x y .
    (range i j) = SOME r &
    M' = (M \ (replace i j x y))
    ==> (collect M' r) = (replace' (collect M r) x y) &
        forall q . ~ q *in r ==> M' at deref q = M at deref q)

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 h:(It 'X 'S)) =>
        pick-any M:(Memory 'S) M':(Memory 'S) i:(It 'X 'S) j:(It 'X 'S)
                               x:'S y:'S
          let {A1 := ((range i j) = SOME stop h);
               A2 := (M' = (M \ (replace i j x y)))}
           assume (A1 & A2)
            let {ER1 := (!prove empty-range1);
                 _ := conclude (i = j)
                        (!chain-> [A1 ==> (i = j)      [ER1]]);
                 _ := conclude (M' = M)
                        (!chain
                         [M' = (M \ (replace i j x y))  [A2]
                             = M                        [(i = j) if-empty]]);
                 B1 := conclude ((collect M' stop h) =
                                 (replace' (collect M stop h) x y))
                         (!chain
                          [(collect M' stop h)
                           = (collect M stop h)         [(M' = M)]
                           = nil:(List 'S)              [collect.of-stop]
                           = (replace' nil x y)         [List.replace.empty]
                           = (replace' (collect M stop h) x y)
                                                        [collect.of-stop]]);
                 B2 := conclude
```

```
                           (forall ?k:(It 'Z 'S) . ~ ?k *in stop h ==>
                              M' at deref ?k = M at deref ?k)
                         pick-any k:(It 'Z 'S)
                           assume (~ k *in stop h)
                             (!chain [(M' at deref k) = (M at deref k)
                                                       [(M' = M)]])}
            (!both B1 B2)
        | (r as (back r':(Range 'X 'S))) =>
           let {ind-hyp := (correctness-prop r')}
             pick-any M:(Memory 'S) M':(Memory 'S) i:(It 'X 'S) j:(It 'X 'S)
                      x:'S y:'S
               let {A1 := ((range i j) = SOME r);
                    A2 := (M' = (M \ (replace i j x y)))}
                assume (A1 & A2)
                 let {B1 := ((collect M' r) =
                             (replace' (collect M r) x y));
                      B2 := (forall h . ~ h *in r ==>
                                        M' at deref h = M at deref h);
                      NB1 := (!prove nonempty-back1);
                      _ := conclude (i =/= j)
                             (!chain-> [A1 ==> (i =/= j)  [NB1]]);
                      LB := (!prove range-back);
                      B3 := (!chain->
                               [A1 ==> ((range (successor i) j) = SOME r')
                                                          [LB]]);
                      B4 := conclude (i = start r)
                              (!chain->
                               [(range i j)
                                = (SOME r)  [A1 A2]
                                = (range (start r)
                                         (finish r))      [range.collapse]
                                ==> (i = start r &
                                     j = finish r)        [range.injective]
                                ==> (i = start r)         [left-and]]);
                      FNIR := (!prove *in.first-not-in-rest);
                      RR := (!prove *in.range-reduce)}
                 conclude (B1 & B2)
                   (!two-cases
                     assume (M at deref i = x)
                       let
                        {M1 := (M \ (deref i) <- y);
                         C1 :=
                          (!chain
                           [M' = (M \ (replace i j x y))  [A2]
                               = (M1 \ (replace (successor i) j x y))
                                                          [if-equal]]);
                         (and C2a C2b) :=
                           (!chain->
                            [C1 ==> (B3 & C1) [augment]
                                ==> ((collect M' r') =
                                     (replace' (collect M1 r') x y) &
                                     forall h . ~ h *in r' ==>
                                        M' at deref h =
                                        M1 at deref h)   [ind-hyp]]);
                         C3 := (!chain->
                                [true
                                 ==> (~ start r *in r')   [FNIR]
                                 ==> (~ i *in r')         [B4]]);
                         _ := (!sym (M at deref i = x));
                         CU := (!prove collect.unchanged);
                         _ := conclude B1
                               (!combine-equations
                                (!chain
                                 [(collect M' r)
                                  = ((M' at deref i) :: (collect M' r'))
                                                          [B4
                                                           collect.of-back]
                                  = ((M1 at deref i) ::
                                     (replace' (collect M1 r') x y))
                                                          [C2a C2b]
```

```
             = (y :: (replace' (collect M1 r') x y))
                                     [assign.equal]
             = (y :: (replace' (collect M r') x y))
                                     [CU]])
             (!chain
              [(replace' (collect M r) x y)
               = (replace' ((M at deref i) ::
                            (collect M r'))
                          x y)       [B4
                                      collect.of-back]
               = (y :: (replace' (collect M r') x y))
                               [List.replace.equal]]));
    _ := conclude B2
          pick-any h
           assume D := (~ h *in r)
            let {E :=
              (!chain->
               [D ==> (~ (deref h =
                          deref start r |
                          h *in r')) [*in.of-back]
                  ==> (~ (deref h = deref i |
                          h *in r')) [B4]
                  ==> (deref h =/= deref i &
                       ~ h *in r')   [dm]
                  ==> (deref h =/= deref i)
                                     [left-and]
                  ==> (deref i =/= deref h)
                                     [sym]])}
            (!chain->
             [D ==> (~ h *in r')      [RR]
                ==> (M' at deref h =
                     M1 at deref h)  [C2b]
                ==> (M' at deref h =
                     M at deref h)   [E
                                      assign.unequal]])}
  (!both B1 B2)
assume (M at deref i =/= x)
  let {M1 := M;
       C1 := (!chain
              [M' = (M \ (replace i j x y)) [A2]
                  = (M \ (replace
                          (successor i) j x y))
                                     [if-unequal]]);
       (and C2a C2b) :=
         (!chain->
          [C1 ==> (B3 & C1)          [augment]
              ==> ((collect M' r') =
                   (replace' (collect M r') x y) &
                   forall h . ~ h *in r' ==>
                     M' at deref h =
                     M at deref h)   [ind-hyp]]);
       C3 := (!chain->
              [true ==> (~ start r *in r')
                                     [FNIR]
                    ==> (~ i *in r') [B4]]);
       _ := (!sym (M at deref i =/= x));
       _ := conclude B1
             (!combine-equations
              (!chain
               [(collect M' r)
                = ((M' at deref i) ::
                   (collect M' r'))  [B4
                                      collect.of-back]
                = ((M at deref i) ::
                   (replace' (collect M r') x y))
                                     [C2a C2b]])
              (!chain
               [(replace' (collect M r) x y)
                = (replace' ((M at deref i) ::
                            (collect M r'))        x y)
```

```
                                                                   [B4
                                                                    collect.of-back]
                                           = ((M at deref i) ::
                                              (replace' (collect M r') x y))
                                                       [List.replace.unequal]]));
                                  _ := conclude B2
                                        pick-any h
                                         assume D := (~ h *in r)
                                          (!chain->
                                           [D ==> (~ h *in r')  [RR]
                                              ==> (M' at deref h = M at deref h)
                                                                   [C2b]])}
                             (!both B1 B2))
         } # by-induction
     } # match theorem

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