lib/memory-range/forward-iterator.ath

```
load "trivial-iterator"
#..........................................................................

module Forward-Iterator {
  open Trivial-Iterator

  define theory := (make-theory ['Trivial-Iterator] [])

  declare successor: (X, S) [(It X S)] -> (It X S)

  module successor {

    define of-start := (forall r . successor start back r = start r)

    define injective := (forall i j . successor i = successor j ==> i = j)

    define deref-of :=
      (forall i r . deref successor i = deref start r
                    ==> deref i = deref start back r)

    (add-axioms theory [of-start injective deref-of])
  }

define start-shift :=
  (forall i r . successor i = start r ==> i = start back r)

define range-back :=
  (forall i j r . (range (successor i) j) = SOME r
                 <==> (range i j) = SOME (back r))

define (finish-not-*in-prop r) :=
  (forall i j k . (range i j) = SOME r & k *in r ==> k =/= j)

define finish-not-*in := (forall r . finish-not-*in-prop r)

define proofs :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
         [deref *in successor] := (adapt [deref *in successor])}
      match theorem {
        (val-of start-shift) =>
        pick-any i:(It 'X 'S) r:(Range 'X 'S)
          assume I := (successor i = start r)
            (!chain->
             [(successor i)
              = (start r)                        [I]
              = (successor start back r)         [successor.of-start]
              ==> (i = start back r)             [successor.injective]])
      | (val-of range-back) =>
        pick-any i:(It 'X 'S) j:(It 'X 'S) r:(Range 'X 'S)
         (!equiv
          assume I := ((range (successor i) j) = SOME r)
           let {SS1 := (!prove start-shift);
                II := (!chain->
                        [(range (successor i) j)
                         = (SOME r)              [I]
                         = (range (start r)
                                  (finish r))    [range.collapse]
                         ==> (successor i = start r & j = finish r)
                                                 [range.injective]])}
            (!chain [(range i j)
                   = (range (start back r)
                            (finish back r))     [II SS1 finish.of-back]
                   = (SOME back r)               [range.collapse]])
          assume I := ((range i j) = (SOME back r))
           let {II := (!chain->
                       [(range i j)
```

```
                      = (SOME back r)            [I]
                      = (range (start back r)
                               (finish back r))  [range.collapse]
                      ==> (i = start back r &
                           j = finish back r)    [range.injective]])}
            (!chain [(range (successor i) j)
                   = (range (start r) (finish r))
                                  [II successor.of-start finish.of-back]
                   = (SOME r)                    [range.collapse]]))
      | (val-of finish-not-*in) =>
         by-induction (adapt theorem) {
           (stop h) =>
             pick-any i j k
               assume ((range i j) = SOME stop h & k *in stop h)
                 (!from-complements (k =/= j)
                  (k *in stop h)
                  (!chain->
                   [true ==> (~ k *in stop h)    [*in.of-stop]]))
         | (r as (back r':(Range 'X 'S))) =>
             let {ind-hyp := (finish-not-*in-prop r')}
              pick-any i:(It 'X 'S) j:(It 'X 'S) k:(It 'X 'S)
               let {A1 := ((range i j) = SOME r);
                    A2 := (k *in r);
                    NB := (!prove nonempty-back)}
                assume (A1 & A2)
                 let {B1 := (!chain->
                             [A2 ==> (deref k = deref start r |
                                      k *in r')    [*in.of-back]])}
                  (!cases B1
                   assume B1a := (deref k = deref start r)
                     let {C1 := (!chain->
                                 [B1a ==> (k = start r)
                                                  [deref.injective]]);
                          (and C2 C3) :=
                            (!chain->
                             [(range i j)
                            = (SOME r)      [A1]
                            = (range (start r) (finish r))
                                                  [range.collapse]
                            ==> (i = start r &
                                 j = finish r)
                                                  [range.injective]])}
                     (!chain->
                      [true ==> (start r =/= finish r)
                                                  [NB]
                            ==> (k =/= j)         [C1 C3]])
                   assume B1b := (k *in r')
                     let {RB := (!prove range-back);
                          C1 := (!chain->
                                 [A1
                                  ==> ((range (successor i) j) =
                                       SOME r')   [RB]]);
                          _ := (!both C1 B1b)}
                       (!fire ind-hyp [(successor i) j k]))
         } # by-induction
      } # match theorem

(add-theorems theory |{[start-shift range-back finish-not-*in] := proofs}|)
#.......................................................................

define range-shift1 :=
  (forall r i . (successor i) in r ==> i in back r)
define range-shift2 :=
  (forall i r . ~ i in back r ==> ~ (successor i) in r)

define proofs :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
         successor := (adapt successor)}
     match theorem {
```

```
      (val-of range-shift1) =>
      by-induction (adapt theorem) {
        (stop h) =>
        pick-any i
          assume I := ((successor i) in stop h)
            let {II := (!chain->
                        [true
                         ==> (~ (successor i) in stop h) [in.of-stop]])}
                 (!from-complements (i in back stop h) I II)
         | (r as (back r':(Range 'X 'S))) =>
            let {ind-hyp := (forall i . (successor i) in r' ==> i in r)}
             pick-any i
               assume A := ((successor i) in r)
                let {case1 := (successor i = start r);
                     case2 := ((successor i) in r');
                     goal := (i in back r);
                     B := (!chain->
                           [A ==> (case1 | case2)         [in.of-back]]);
                     SS := (!prove start-shift)}
                  (!cases B
                   assume case1
                     (!chain->
                      [case1
                       ==> (i = start back r)             [SS]
                       ==> (i = start back r | i in r)    [alternate]
                       ==> goal                           [in.of-back]])
                   assume case2
                     (!chain->
                      [case2
                   ==> (i in r)                           [ind-hyp]
                   ==> (i = start back r | i in r)        [alternate]
                   ==> goal                               [in.of-back]]))
         }
      | (val-of range-shift2) =>
        pick-any i r
          let {RS1 := (!prove range-shift1);
               p := (!chain [((successor i) in r)
                             ==> (i in back r)            [RS1]])}
            (!contra-pos p)
      }

(add-theorems theory |{[range-shift1 range-shift2] := proofs}|)

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


  module *in {
    open Trivial-Iterator.*in

    define range-shift1 :=
      (forall r i . (successor i) *in r ==> i *in back r)
    define range-shift2 :=
      (forall i r . ~ i *in back r ==> ~ (successor i) *in r)

define proofs :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory)}
    match theorem {
      (val-of range-shift1) =>
      by-induction (adapt theorem) {
        (stop h) =>
        pick-any i
          assume I := ((successor i) *in stop h)
            let {II := (!chain->
                        [true ==> (~ (successor i) *in stop h)
                                                            [of-stop]])}
              (!from-complements (i *in back stop h) I II)
      | (back r:(Range 'Y 'S)) =>
        let {ind-hyp := (forall ?i:(It 'X 'S) .
                          (successor ?i) *in r ==> ?i *in back r)}
```

```
          pick-any i
           assume A := ((successor i) *in back r)
             let {case1 := (deref successor i = deref start back r);
                  case2 := ((successor i) *in r);
                  goal := (i *in back back r);
                  B := (!chain->
                        [A ==> (case1 | case2)             [of-back]]);
                  DO := (!prove successor.deref-of)}
            (!cases (case1 | case2)
              assume case1
                (!chain->
                 [case1
              ==> (deref i = deref start back back r)      [DO]
              ==> (deref i = deref start back back r |
                   i *in back r)                            [alternate]
              ==> goal                                      [of-back]])
              assume case2
                (!chain->
                 [case2
              ==> (i *in back r)                            [ind-hyp]
              ==> (deref  i = deref start back back r |
                   i *in back r)                            [alternate]
              ==> goal                                      [of-back]]))
      }
      | (val-of range-shift2) =>
        pick-any i r
          let {RS1 := (!prove range-shift1);
               p := (!chain [((successor i) *in r)
                             ==> (i *in back r)   [RS1]])}
          (!contra-pos p)
      }

(add-theorems theory |{[range-shift1 range-shift2] := proofs}|)
} # close module *in
} # close module Forward-Iterator
```