lib/memory-range/trivial-iterator.ath

```
load "range"
load "memory"
load "list-of"
#..................................................................

module Trivial-Iterator {
  open Range, Memory

  define theory := (make-theory ['Range 'Memory] [])

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

  declare deref: (X, S) [(It X S)] -> (Memory.Loc S)

  module deref {

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

    (add-axioms theory [injective])
  }

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

  declare *in: (X, Y, S) [(It X S) (Range Y S)] -> Boolean

  module *in {

    define of-stop := (forall i k . ~ i *in (stop k))
    define of-back :=
      (forall i r . i *in (back r)
                  <==> deref i = deref start back r | i *in r)

    define first-not-in-rest := (forall r . ~ start back r *in r)

    (add-axioms theory [of-stop of-back first-not-in-rest])

define range-expand := (forall i r . i *in r ==> i *in (back r))
define range-reduce := (forall i r . ~ i *in (back r) ==> ~ i *in r)
define theorems := [range-expand range-reduce]
define proofs :=
  method (theorem adapt)
   let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
        [deref *in] := (adapt [deref *in])}
    match theorem {
      (val-of range-expand) =>
      pick-any i:(It 'X 'S) r:(Range 'Y 'S)
        (!chain
         [(i *in r)
          ==> (deref i = deref start back r | i *in r)  [alternate]
          ==> (i *in back r)                            [of-back]])
    | (val-of range-reduce) =>
      pick-any i r
        let {RE := (!prove range-expand);
             p := (!chain [(i *in r) ==> (i *in back r) [RE]])}
        (!contra-pos p)
    }

    (add-theorems theory |{theorems := proofs}|)
  } # close module *in

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

  declare collect: (S, X) [(Memory S) (Range X S)] -> (List S)

  module collect {
```

```
    define axioms :=
     (fun
      [(collect M (stop h)) = nil
       (collect M (back r)) = ((M at deref start back r) :: (collect M r))])

    define [of-stop of-back] := axioms

    (add-axioms theory axioms)

    define (unchanged-prop r) :=
     (forall M i v .
        ~ i *in r ==> (collect (M \ (deref i) <- v) r) = (collect M r))

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

define proof :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
         [deref *in] := (adapt [deref *in])}
     match theorem {
      (val-of unchanged) =>
      by-induction (adapt theorem) {
        (stop h:(It 'Y 'S)) =>
        pick-any M:(Memory 'S) i:(It 'X 'S) v:'S
          assume (~ i *in stop h)
            let {M1 := (M \ (deref i) <- v)}
              (!chain [(collect M1 (stop h))
                     = nil:(List 'S)        [of-stop]
                     = (collect M (stop h)) [of-stop]])
      | (r as (back r':(Range 'Y 'S))) =>
        pick-any M:(Memory 'S) i:(It 'X 'S) v:'S
          let {ind-hyp := (unchanged-prop r');
                k' := (start r);
                M1 := (M \ (deref i) <- v)}
            assume A := (~ i *in r)
              let {B1 := (!chain->
                          [A ==> (~ (deref i = deref k' |
                                     i *in r'))                 [*in.of-back]
                             ==> (deref i =/= deref k' &
                                  ~ i *in r')                   [dm]
                             ==> (deref i =/= deref k')         [left-and]]);
                   RR := (!prove *in.range-reduce);
                   B2 := (!chain->
                          [A
                       ==> (~ i *in r')                         [RR]
                       ==> ((collect M1 r') = (collect M r'))   [ind-hyp]])}
                 (!chain [(collect M1 r)
                        = ((M1 at deref k') :: (collect M1 r')) [of-back]
                        = ((M at deref k') :: (collect M1 r'))
                                                       [B1 assign.unequal]
                        = ((M at deref k') :: (collect M r'))   [B2]
                        = (collect M r)                         [of-back]])
     }
   }

   (add-theorems theory |{[unchanged] := proof}|)
 } # close modlule collect
} # close module Trivial-Iterator
```