load "forward-iterator"

extend-module Forward-Iterator {

  define collect := Trivial-Iterator.collect

  declare count1: (S, X) [S (It X S) (It X S) N] -> N

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

  module count {
    define A := ?A:N
    define axioms :=
    (fun
    [(M \ (count1 x i j A)) =
    [A when (i = j)
    (M \ (count1 x (successor i) j (S A))) when (i /= j &
    M at deref i = x)
    (M \ (count1 x (successor i) j A)) when (i /= j &
    M at deref i /= x)]
    (M \ (count x i j)) = (M \ (count1 x i j zero)))
    define [if-empty if-equal if-unequal definition] := axioms
    (add-axioms theory axioms)
    define count' := List.count
    overload + N.+
    define (correctness1-prop r) :=
    (forall M x i j A .
    (range i j) = SOME r ==>
    M \ (count1 x i j A) = (count' x (collect M r)) + A)
    define correctness1 := (forall r . correctness1-prop r)
    define correctness :=
    (forall r M x i j .
    (range i j) = SOME r ==> 
    M \ (count x i j) = (count' x (collect M r)))
    define proofs :=
    method (theorem adapt)
    let [{[get prove chain chain-> chain<-] := (proof-tools adapt theory);
    [deref successor] := (adapt [deref successor])}]
    match theorem {
    (val-of correctness1) =>
    by-induction (adapt theorem) {
    (stop h:(It 'X 'S)) =>
    pick-any M:(Memory 'S) x:'S i:(It 'X 'S) j:(It 'X 'S) A:N
    assume I := ((range i j) = (some stop h))
    let (ER1 := (!prove empty-range1);
    _ := (!chain-> [I ==> (i = j) [ER1]])}
    (!combine-equations
    (!chain ((count' x (collect M (stop h))) + A)
    = A
    [if-empty]))
    (!chain (((count' x (collect M (stop h))) + A)
    = (zero + A) [collect.of-stop]
    = A [N.Plus.left-zero]))}
  }
  }
}
assume I := ((range i j) = SOME r)
let (goal := (M \ (count x i j A)) =
  (count' x (collect M r)) + A);
NB1 := (!prove nonempty-back);
LB := (!prove range-back);
II := conclude (i /= j)
  ((chain-> [I => (i /= j) [NB1]]));
III := (!chain->
  [I => ((range (successor i) j) = SOME r')
   [LB]]);
IV := conclude (i = start r)
  ((chain->
    (range i j) = (SOME r) [I]
    = (range (start r) (finish r)) [range.collapse]
    ==> (i = start r & j = finish r) [range.injective]
    ==> (i = start r) [left-and]))
(!two-cases
assume casel := (M at deref i = x)
conclude goal
(!combine-equations
(!chain
  [(M \ (count1 x i j A))
    = (M \ (count1 x (successor i) j (S A))) [if-equal]
    = ((count' x (collect M r')) + (S A)) [III ind-hyp]
    = (S ((count' x (collect M r')) + A))
    [N.Plus.right-nonzero]])
(!chain
  [((count' x (collect M r)) + A)
    = ((count' x (M at deref i)) :: (collect M r')) + A]
  [IV collect.of-back]
  = (S ((count' x (collect M r')) + A))
  [casel List.count.same]
  [N.Plus.left-nonzero]])
assume case2 := (M at deref i /= x)
conclude goal
let ___ := (!sym case2)
(!combine-equations
(!chain
  [(M \ (count1 x i j A))
    = (M \ (count1 x (successor i) j A)) [if-unequal]
    = ((count' x (collect M r')) + A) [III ind-hyp]])
(!chain
  [((count' x (collect M r)) + A)
    = ((count' x (M at deref i)) :: (collect M r')) + A]
  [IV collect.of-back]
  = ((count' x (collect M r')) + A)]
  [case2 List.count.same]])
)
# by-induction
| (val-of correctness) =>
let [LL := (!prove correctness1)]
pick-any r:(Range 'X 'S) M:(Memory 'S) x:'S
i:(It 'X 'S) j:(It 'X 'S)
assume ((range i j) = SOME r)
(!chain
  [(M \ (count x i j))
    = (M \ (count1 x i j) zero) [definition]
    = ((count' x (collect M r)) + zero) [L1]
    = (count' x (collect M r)) [N.Plus.right-zero]])
)
# match theorem

(add-theorems theory |{[correctness1 correctness] := proofs}|)
)
# count
| # Forward-Iterator