lib/main/list-of.ath

```
load "nat-plus"

#########################################################################
##  Polymorphic lists

#datatype (List T) := nil | (:: T (List T))

assert (datatype-axioms "List")

module List {
 open N

#-----------------------------------------------------------------------

 define [L L' L1 L2 L3 l l' l1 l2 p q r L M x y z x' h t t1 t2] :=
         [?L:(List 'S1) ?L':(List 'S2) ?L1:(List 'S3) ?L2:(List 'S4) ?L2:(List 'S5)
         ?l:(List 'S6) ?l':(List 'S7) ?l1:(List 'S8) ?l2:(List 'S9)
          ?p:(List 'S10) ?q:(List 'S11) ?r:(List 'S12) ?L:(List 'S13)
          ?M:(List 'S14) ?x ?y ?z ?x' ?h ?t ?t1 ?t2]


 declare join: (T) [(List T) (List T)] -> (List T) [[(alist->list id) (alist->list id)]]
 define ++ := join

 module join {

  assert left-empty := (forall q . nil join q = q)

  assert left-nonempty :=
    (forall x r q . (x :: r) join q = x :: (r join q))

  define right-empty := (forall p . p join nil = p)

  define right-nonempty :=
   (forall p y q .
     p join (y :: q) = (p join (y :: nil)) join q)

  by-induction right-empty {
    nil =>
    (!chain [(nil join nil) = nil [left-empty]])
  | (x :: p) =>
    let {induction-hypothesis := (p join nil = p)}
    (!chain [((x :: p) join nil)
          --> (x :: (p join nil))  [left-nonempty]
          --> (x :: p)             [induction-hypothesis]])
  }

  by-induction right-nonempty {
    nil =>
    pick-any y q
      (!combine-equations
       (!chain [(nil join (y :: q))
               --> (y :: q)  [left-empty]])
       (!chain [((nil join (y :: nil)) join q)
               --> ((y :: nil) join q)  [left-empty]
               --> (y :: (nil join q))  [left-nonempty]
               --> (y :: q)             [left-empty]]))
  | (x :: p) =>
    let {induction-hypothesis :=
            (forall ?y ?q .
              p join (?y :: ?q) = (p join (?y :: nil)) join ?q)}
    conclude (forall ?y ?q .
                (x :: p) join (?y :: ?q) =
                ((x :: p) join (?y :: nil)) join ?q)
      pick-any y q
        (!combine-equations
         (!chain [((x :: p) join (y :: q))
                 --> (x :: (p join (y :: q))) [left-nonempty]
```

```
                  --> (x :: ((p join (y :: nil)) join q))
                                        [induction-hypothesis]])
         (!chain [(((x :: p) join (y :: nil)) join q)
                  --> ((x :: (p join (y :: nil))) join q)
                                        [left-nonempty]
                  --> (x :: ((p join (y :: nil)) join q))
                                        [left-nonempty]]))
  }

  define Associative :=
    (forall p q r .
      (p join q) join r = p join (q join r))

  by-induction Associative {
    nil =>
    pick-any q r
      (!chain [((nil join q) join r)
               --> (q join r)            [left-empty]
               <-- (nil join (q join r)) [left-empty]])
  | (x :: p) =>
    let {induction-hypothesis :=
           (forall ?q ?r . (p join ?q) join ?r =
                           p join (?q join ?r))}
    conclude (forall ?q ?r .
               ((x :: p) join ?q) join ?r =
                (x :: p) join (?q join ?r))
      pick-any q r
        (!chain
         [(((x :: p) join q) join r)
          --> ((x :: (p join q)) join r) [left-nonempty]
          --> (x :: ((p join q) join r)) [left-nonempty]
          --> (x :: (p join (q join r))) [induction-hypothesis]
          <-- ((x :: p) join (q join r)) [left-nonempty]
          ])
  }

  define left-singleton :=
    (forall x p . (x :: nil) join p = x :: p)

  conclude left-singleton
    pick-any x p
      (!chain
       [((x :: nil) join p)
        = (x :: (nil join p))    [left-nonempty]
        = (x :: p)               [left-empty]])

 } # join

#------------------------------------------------------------------------
 declare reverse: (T) [(List T)] -> (List T) [[(alist->list id)]]

 module reverse {

  assert empty := ((reverse nil) = nil)
  assert nonempty :=
   (forall x r . (reverse (x :: r)) = (reverse r) join (x :: nil))

  define of-join :=
   (forall p q . (reverse (p join q)) = (reverse q) join (reverse p))

  define of-reverse := (forall p . (reverse (reverse p)) = p)

  by-induction of-join {
    nil =>
    conclude (forall q . (reverse (nil join q)) =
                         (reverse q) join (reverse nil))
      pick-any q
        (!combine-equations
         (!chain [(reverse (nil join q))
                  --> (reverse q)            [join.left-empty]])
```

```
         (!chain [((reverse q) join (reverse nil))
                  --> ((reverse q) join nil) [empty]
                  --> (reverse q)            [join.right-empty]]))
  | (x :: p) =>
    let {induction-hypothesis :=
           (forall ?q . (reverse (p join ?q)) =
                        (reverse ?q) join (reverse p))}
    conclude (forall ?q . (reverse ((x :: p) join ?q)) =
                          (reverse ?q) join (reverse (x :: p)))
      pick-any q
        (!chain [(reverse ((x :: p) join q))
                 --> (reverse (x :: (p join q)))  [join.left-nonempty]
                 --> ((reverse (p join q)) join (x :: nil))
                                             [nonempty]
                 --> (((reverse q) join (reverse p)) join (x :: nil))
                                             [induction-hypothesis]
                 --> ((reverse q) join ((reverse p) join (x :: nil)))
                                             [join.Associative]
                 <-- ((reverse q) join (reverse (x :: p)))
                                             [nonempty]])
  }

  by-induction of-reverse {
    nil =>
    conclude ((reverse (reverse nil)) = nil)
      (!chain [(reverse (reverse nil))
               --> (reverse nil)           [empty]
               --> nil                     [empty]])
  | (x :: p) =>
    conclude ((reverse (reverse (x :: p))) = (x :: p))
      let {induction-hypothesis := ((reverse (reverse p)) = p)}
      (!chain
       [(reverse (reverse (x :: p)))
        --> (reverse ((reverse p) join (x :: nil)))
                                             [nonempty]
        --> ((reverse (x :: nil)) join (reverse (reverse p)))
                                             [of-join]
        --> ((reverse (x :: nil)) join p)   [induction-hypothesis]
        --> (((reverse nil) join (x :: nil)) join p)
                                             [nonempty]
        --> ((nil join (x :: nil)) join p)  [empty]
        --> ((x :: nil) join p)             [join.left-empty]
        --> (x :: (nil join p))             [join.left-nonempty]
        --> (x :: p)                        [join.left-empty]])
  }

#----------------------------------------------------------------------------
# Another relationship between reverse and join:

  define join-singleton :=
    (forall p x . (reverse (p join (x :: nil))) =
                  x :: (reverse p))

  conclude join-singleton
    pick-any p x
      (!chain
       [(reverse (p join (x :: nil)))
        --> ((reverse (x :: nil)) join (reverse p))  [of-join]
        --> (((reverse nil) join (x :: nil)) join (reverse p))
                                                      [nonempty]
        --> ((nil join (x :: nil)) join (reverse p)) [empty]
        --> ((x :: nil) join (reverse p))  [join.left-empty]
        --> (x :: (nil join (reverse p)))  [join.left-nonempty]
        --> (x :: (reverse p))             [join.left-empty]])

#----------------------------------------------------------------------------
# Another proof of reverse, using join-singleton:

  by-induction of-reverse {
    nil =>
```

```
    conclude ((reverse (reverse nil)) = nil)
      (!chain [(reverse (reverse nil))
               --> (reverse nil)          [empty]
               --> nil                    [empty]])
  | (x :: p) =>
    conclude ((reverse (reverse (x :: p))) = (x :: p))
      let {induction-hypothesis := ((reverse (reverse p)) = p)}
      (!chain
       [(reverse (reverse (x :: p)))
        --> (reverse ((reverse p) join (x :: nil))) [nonempty]
        --> (x :: (reverse (reverse p)))  [join-singleton]
        --> (x :: p)                      [induction-hypothesis]])
  }
 } # reverse

#=====================================================================
declare length: (T) [(List T)] -> N  [[(alist->list id)]]

module length {

assert empty := (length nil = zero)
assert nonempty := (forall p x . length (x :: p) =  S length p)

define of-join := (forall p q .
                   length (p join q) = (length p) + (length q))
define of-reverse := (forall p . length reverse p = length p)

by-induction of-join {
  nil:(List 'S) =>
  conclude (forall ?q .
             length (nil:(List 'S) join ?q) = (length nil:(List 'S) + (length ?q)))
    pick-any q:(List 'S)
      (!combine-equations
       (!chain
        [(length (nil join q))
         --> (length q)             [join.left-empty]])
       (!chain
        [((length nil:(List 'S)) + (length q))
         --> (zero + (length q))    [empty]
         --> (length q)             [Plus.left-zero]]))
| (H:'S :: T:(List 'S)) =>
  conclude (forall ?q . length ((H :: T) join ?q) =
                        (length (H :: T)) + length ?q)
    let {induction-hypothesis :=
          (forall ?q . length (T join ?q) = (length T) + length ?q)}
    pick-any q:(List 'S)
      (!combine-equations
       (!chain
        [(length ((H :: T) join q))
         --> (length (H :: (T join q)))     [join.left-nonempty]
         --> (S (length (T join q)))        [nonempty]
         --> (S ((length T) + (length q)))  [induction-hypothesis]])
       (!chain
        [((length (H :: T)) + (length q))
         --> ((S (length T)) + (length q))  [nonempty]
         --> (S ((length T) + (length q)))  [Plus.left-nonzero]]))
}

by-induction of-reverse {
  nil =>
  (!chain [(length (reverse nil:(List 'S)))
           --> (length nil:(List 'S))       [reverse.empty]])
| (x :: p:(List 'S)) =>
  let {induction-hypothesis := ((length (reverse p)) = (length p))}
  conclude (length (reverse (x :: p)) = length (x :: p))
    (!chain
     [(length (reverse (x :: p)))
      --> (length ((reverse p) join (x :: nil)))
                                            [reverse.nonempty]
      --> ((length (reverse p)) + (length (x :: nil)))
```

```
                                                  [of-join]
      --> ((length p) + (length (x :: nil)))    [induction-hypothesis]
      --> ((length p) + (S (length nil:(List 'S)))) [nonempty]
      --> ((length p) + (S zero))               [empty]
      --> (S ((length p) + zero))               [Plus.right-nonzero]
      --> (S (length p))                        [Plus.right-zero]
      <-- (length (x :: p))                     [nonempty]])
}

} # length

#======================================================================
# List.count: given a value x and a list, returns the number
# of occurrences of x in the list.

declare count: (S) [S (List S)] -> N  [[id (alist->list id)]]

module count {
define [x x' L M] := [?x:'S ?x':'S ?L:(List 'S) ?M:(List 'S)]

assert axioms :=
  (fun
   [(count x nil)       = zero
    (count x (x' :: L)) = [(S (count x L))   when (x = x')
                           (count x L)       when (x =/= x')]])

define [empty more same] := axioms

define of-join :=
  (forall L M x . (count x (L join M)) = (count x L) + (count x M))
define of-reverse :=
  (forall L x . (count x (reverse L)) = (count x L))

by-induction of-join {
  nil =>
  pick-any M x
    (!combine-equations
     (!chain [(count x (nil join M))
              = (count x M)             [join.left-empty]])
     (!chain [((count x nil) + (count x M))
             = (zero + (count x M))     [empty]
             = (count x M)              [Plus.left-zero]]))
| (y :: L) =>
  let {ind-hyp := (forall ?M ?x . (count ?x (L join ?M)) =
                                  (count ?x L) + (count ?x ?M))}
  conclude (forall ?M ?x . (count ?x ((y :: L) join ?M)) =
                           (count ?x (y :: L)) + (count ?x ?M))
    pick-any M x
      (!two-cases
        assume (x = y)
          (!combine-equations
           (!chain
            [(count x ((y :: L) join M))
             = (count x (y :: (L join M)))     [join.left-nonempty]
             = (S (count x (L join M)))        [more]
             = (S ((count x L) + (count x M))) [ind-hyp]])
           (!chain
            [((count x (y :: L)) + (count x M))
             = ((S (count x L)) + (count x M)) [more]
             = (S ((count x L) + (count x M))) [Plus.left-nonzero]
             ]))
        assume (x =/= y)
          (!combine-equations
           (!chain
            [(count x ((y :: L) join M))
             = (count x (y :: (L join M)))    [join.left-nonempty]
             = (count x (L join M))           [same]
             = ((count x L) + (count x M))    [ind-hyp]])
           (!chain
            [((count x (y :: L)) + (count x M))
```

```
              = ((count x L) + (count x M))    [same]
              ])))
}

by-induction of-reverse {
  nil =>
  pick-any x
    (!chain [(count x (reverse nil))
              = (count x nil)          [reverse.empty]])
| (y :: L) =>
  let {ind-hyp := (forall ?x . (count ?x (reverse L)) = (count ?x L))}
  conclude (forall ?x . (count ?x (reverse (y :: L))) =
                        (count ?x (y :: L)))
    pick-any x
      (!two-cases
        assume (x = y)
          (!chain
           [(count x (reverse (y :: L)))
            = (count x ((reverse L) join (y :: nil)))
                                       [reverse.nonempty]
            = ((count x (reverse L)) + (count x (y :: nil)))
                                       [of-join]
            = ((count x L) + (S (count x nil)))
                                       [ind-hyp more]
            = ((count x L) + (S zero)) [empty]
            = (S ((count x L) + zero)) [Plus.right-nonzero]
            = (S (count x L))          [Plus.right-zero]
            = (count x (y :: L))       [more]])
        assume (x =/= y)
          (!chain
           [(count x (reverse (y :: L)))
            = (count x ((reverse L) join (y :: nil)))
                                       [reverse.nonempty]
            = ((count x (reverse L)) + (count x (y :: nil)))
                                       [of-join]
            = ((count x L) + (count x nil))
                                       [ind-hyp same]
            = ((count x L) + zero)     [empty]
            = (count x L)              [Plus.right-zero]
            = (count x (y :: L))       [same]]))
}
} # count

#======================================================================
# List.in (membership)

declare in: (T) [T (List T)] -> Boolean  [[id (alist->list id)]]

module in {

assert empty := (forall x . ~ x in nil)
assert nonempty := (forall x y L . x in (y :: L) <==> x = y | x in L)

#......................................................................
# Lemmas:

define head := (forall x L . x in (x :: L))
define tail := (forall x y L . x in L ==> x in (y :: L))

conclude head
  pick-any x L
    (!chain-> [(x = x)
               ==> (x = x | x in L) [alternate]
               ==> (x in (x :: L))  [nonempty]])

conclude tail
  pick-any x y L
    (!chain [(x in L)
             ==> (x = y | x in L)   [alternate]
             ==> (x in (y :: L))    [nonempty]])
```

```
define of-singleton :=
  (forall x y . x in (y :: nil) ==> x = y)

conclude of-singleton
  pick-any x y
    assume (x in (y :: nil))
      let {C := (!chain-> [(x in (y :: nil)) ==> (x = y | x in nil)
                                          [nonempty]])}
      (!cases C
       assume (x = y)
         (!claim (x = y))
       assume (x in nil)
         (!from-complements (x = y)
          (x in nil) (!chain-> [true ==> (~ x in nil) [empty]])))

#.......................................................................
# Theorem:

define of-join :=
  (forall L M x . x in (L join M) <==> x in L | x in M)

by-induction of-join {
  nil =>
  conclude (forall ?M ?x . ?x in (nil join ?M) <==>
                           ?x in nil | ?x in ?M)
    pick-any M x
      let {_ := (!chain->
                 [true ==> (~ x in nil)         [empty]
                       <==> (x in nil <==> false) [prop-taut]])}
      (!chain
       [(x in (nil join M))
        <==> (x in M)             [join.left-empty]
        <==> (false | x in M)     [prop-taut]
        <==> (x in nil | x in M)  [(x in nil <==> false)]])
| (y :: L) =>
  let {ind-hyp := (forall ?M ?x .
                    ?x in (L join ?M) <==> ?x in L | ?x in ?M)}
  conclude (forall ?M ?x .
              ?x in ((y :: L) join ?M) <==>
              ?x in (y :: L) | ?x in ?M)
    pick-any M x
      (!chain
       [(x in ((y :: L) join M))
        <==> (x in (y :: (L join M)))    [join.left-nonempty]
        <==> (x = y | x in (L join M))   [nonempty]
        <==> (x = y | x in L | x in M)   [ind-hyp]
        <==> ((x = y | x in L) | x in M) [prop-taut]
        <==> (x in (y :: L) | x in M)    [nonempty]])
}
} # in

#=======================================================================
# (List.replace L x y) returns a copy of L except that all
# occurrences of x are replaced by y

declare replace: (S) [(List S) S S] -> (List S) [[(alist->list id) id id]]

module replace {

assert axioms :=
  (fun
   [(replace nil x y) = nil
    (replace (x' :: L) x y) =
       [(y :: (replace L x y))    when (x = x')
        (x' :: (replace L x y))   when (x =/= x')]])

define [empty equal unequal] := axioms

define sanity-check1 :=
```

```
  (forall L x y .
    x =/= y  ==> (count x (replace L x y)) = zero)

define sanity-check2 :=
  (forall L x y .
    x =/= y  ==>
    (count y (replace L x y)) = (count x L) + (count y L))

by-induction sanity-check1 {
  nil =>
  pick-any x y
    assume (x =/= y)
      (!chain [(count x (replace nil x y))
                = (count x nil)           [empty]
                = zero                    [count.empty]])
| (z :: L) =>
  pick-any x y
    assume (x =/= y)
      let {ind-hyp := (forall ?x ?y .
                         ?x =/= ?y ==> (count ?x (replace L ?x ?y)) = zero);
           _ := (!sym (x =/= y))}
      (!two-cases
       assume (x = z)
         (!chain
          [(count x (replace (z :: L) x y))
           = (count x (y :: (replace L x y)))  [equal]
           = (count x (replace L x y))         [count.same]
           = zero                              [ind-hyp]])
       assume (x =/= z)
         (!chain
          [(count x (replace (z :: L) x y))
           = (count x (z :: (replace L x y)))  [unequal]
           = (count x (replace L x y))         [count.same]
           = zero                              [ind-hyp]]))
}

by-induction sanity-check2 {
  nil =>
  pick-any x y
    assume (x =/= y)
      (!combine-equations
       (!chain [(count y (replace nil x y))
                = (count y nil)           [empty]
                = zero                    [count.empty]])
       (!chain [((count x nil) + (count y nil))
                = (zero + zero)           [count.empty]
                = zero                    [Plus.right-zero]]))
| (z:'S :: L) =>
  pick-any x:'S y
    assume (x =/= y)
      let {ind-hyp := (forall ?x ?y .
                         ?x =/= ?y ==> (count ?y (replace L ?x ?y)) =
                                       (count ?x L) + (count ?y L));
           _ := (!sym (x =/= y))}
      (!two-cases
       assume (y = z)
         (!combine-equations
          (!chain
           [(count y (replace (z :: L) x y))
            = (count y (replace (y :: L) x y))  [(y = z)]
            = (count y (y :: (replace L x y)))  [unequal]
            = (S (count y (replace L x y)))     [count.more]
            = (S ((count x L) + (count y L)))   [ind-hyp]])
          (!chain
           [((count x (z :: L)) + (count y (z :: L)))
            = ((count x (y :: L)) + (count y (z :: L)))
            = ((count x L) + (count y (y :: L))) [count.same (y = z)]
            = ((count x L) + (S (count y L)))    [count.more]
            = (S ((count x L) + (count y L)))    [Plus.right-nonzero]]))
       assume (y =/= z)
```

```
         (!two-cases
          assume (x = z)
            (!combine-equations
             (!chain
              [(count y (replace (z :: L) x y))
               = (count y (y :: (replace L x y))) [equal]
               = (S (count y (replace L x y)))    [count.more]
               = (S ((count x L) + (count y L)))  [ind-hyp]])
             (!chain
              [((count x (z :: L)) + (count y (z :: L)))
               = ((S (count x L)) + (count y L))  [count.more count.same]
               = (S ((count x L) + (count y L)))  [Plus.left-nonzero]]))
          assume (x =/= z)
            (!combine-equations
             (!chain
              [(count y (replace (z :: L) x y))
               = (count y (z :: (replace L x y))) [unequal]
               = (count y (replace L x y))        [count.same]
               = ((count x L) + (count y L))      [ind-hyp]])
             (!chain
              [((count x (z :: L)) + (count y (z :: L)))
               = ((count x L) + (count y L))      [count.same]]))))
}
} # replace
} # List

define (alist->clist inner) :=
  letrec {loop := lambda (L acc)
                    match L {
                      (list-of x rest) => (loop rest ((inner x) :: acc))
                    | [] => acc
                    }}
    lambda (L)
      match L {
        (some-list _) => (loop (rev L) (nil))
      | _ => L
      }

define (clist->alist inner) :=
  letrec {loop := lambda (L acc)
                   match L {
                     (x :: rest) => (loop rest (add (inner x) acc))
                   | nil => (rev acc)
                   }}
    lambda (L)
      match L {
        (x :: rest) => (loop L [])
      | nil => []
      | _ => L
      }
```