lib/main/ordered-list.ath

```
# Properties of ordered lists

load "list-of"
load "order"

extend-module SWO {
open List
#.....................................................................
# <EL: is a T value < or E the first element of a list of T
# values (true if the list is empty)

declare <EL:  (T) [T (List T)] -> Boolean

module <EL {
define empty := (forall x . x <EL nil)
define nonempty :=
  (forall x y L . x <EL (y :: L) <==> x <E y)

(add-axioms theory [empty nonempty])

define left-transitive := (forall L x y . x <E y & y <EL L ==> x <EL L)
define before-all-implies-before-first :=
  (forall L x . (forall y . y in L ==> x <E y) ==> x <EL L)
define append := (forall L M x . x <EL L & x <EL M ==> x <EL (L join M))
define append-2 := (forall L M x . x <EL (L join M) ==> x <EL L)

define theorems := [left-transitive before-all-implies-before-first
                    append append-2]
define proofs :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
         [< <E <EL] := (adapt [< <E <EL])}
    match theorem {
      (val-of left-transitive) =>
      datatype-cases theorem {
        nil =>
          pick-any x y
            assume (x <E y & y <EL nil)
              (!chain-> [true ==> (x <EL nil) [empty]])
      | (z :: M) =>
         let {ET := (!prove <E-transitive)}
          pick-any x y
            assume (x <E y & y <EL (z :: M))
              conclude (x <EL (z :: M))
                (!chain-> [(x <E y & y <EL (z :: M))
                           ==> (x <E y & y <E z)       [nonempty]
                           ==> (x <E z)                [ET]
                           ==> (x <EL (z :: M))        [nonempty]])
      }
    | (val-of before-all-implies-before-first) =>
      datatype-cases theorem {
        nil =>
        pick-any x
          assume (forall ?y . ?y in nil ==> x <E ?y)
            conclude (x <EL nil)
              (!chain-> [true ==> (x <EL nil) [empty]])
      | (z :: L) =>
        pick-any x
          assume i := (forall ?y . ?y in (z :: L) ==> x <E ?y)
            conclude (x <EL (z :: L))
              (!chain-> [(z = z) ==> (z = z | z in L)  [alternate]
                                 ==> (z in (z :: L))   [in.nonempty]
                                 ==> (x <E z)          [i]
                                 ==> (x <EL (z :: L))  [nonempty]])
      }
    | (val-of append) =>
       datatype-cases theorem {
         nil =>
```

```
           pick-any M x
             (!chain
              [(x <EL nil & x <EL M)
               ==> (x <EL M)            [right-and]
               ==> (x <EL (nil join M)) [join.left-empty]])
       | (u :: N) =>
           pick-any M x
             assume (x <EL (u :: N) & (x <EL M))
               (!chain-> [(x <EL (u :: N))
                          ==> (x <E u)                  [nonempty]
                          ==> (x <EL (u :: (N join M))) [nonempty]
                          ==> (x <EL ((u :: N) join M))
                                            [join.left-nonempty]])
       }
    | (val-of append-2) =>
      datatype-cases theorem {
        nil =>
        pick-any M x
          assume (x <EL (nil join M))
            (!chain-> [true ==> (x <EL nil)    [empty]])
      | (y :: L) =>
        pick-any M x
          (!chain [(x <EL ((y :: L) join M))
                   ==> (x <EL (y :: (L join M))) [join.left-nonempty]
                   ==> (x <E y)                  [nonempty]
                   ==> (x <EL (y :: L))          [nonempty]])
      }
    }

(add-theorems theory |{theorems := proofs}|)
} # <EL
#.........................................................................
# ordered: are the elements of a list in (nondecending) order?

declare ordered: (T) [(List T)] -> Boolean

module ordered {
open <EL

define empty := (ordered nil)
define nonempty :=
  (forall L x . ordered (x :: L) <==> x <EL L & ordered L)

(add-axioms theory [empty nonempty])

define head := (forall L x . ordered (x :: L) ==> x <EL L)
define tail := (forall L x . ordered (x :: L) ==> ordered L)

define proofs :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
         [< ordered <EL] := (adapt [< ordered <EL])}
   match theorem {
     (val-of head) =>
     pick-any L x
       (!chain [(ordered (x :: L))
                ==> (x <EL L & ordered L) [nonempty]
                ==> (x <EL L)             [left-and]])

   | (val-of tail) =>
     pick-any L x
       (!chain [(ordered (x :: L))
                ==> (x <EL L & ordered L) [nonempty]
                ==> (ordered L)           [right-and]])
   }

(add-theorems theory |{[head tail] := proofs}|)

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

```
define first-to-rest-relation :=
  (forall L x y . ordered (x :: L) & y in L ==> x <E y)
define cons := (forall L x . ordered L & (forall y . y in L ==> x <E y)
                             ==> (ordered (x :: L)))
define append :=
  (forall L M . ordered L & ordered M &
                (forall x y . x in L & y in M ==> x <E y)
                ==> ordered (L join M))
define append-2 :=
  (forall L M . ordered (L join M) ==> ordered L & ordered M)

define theorems := [first-to-rest-relation cons append append-2]
define proofs :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
         [ordered <EL] := (adapt [ordered <EL])}
    match theorem {
      (val-of first-to-rest-relation) =>
      by-induction (adapt theorem) {
        nil =>
        pick-any x y
          assume i := (ordered (x :: nil) & y in nil)
            let {not-in := (!chain-> [true ==> (~ y in nil) [in.empty]])}
            (!from-complements (x <E y) (y in nil) not-in)
      | (z :: M) =>
        let {ind-hyp := (forall ?x ?y .
                          ordered (?x :: M) & ?y in M
                          ==> ?x <E ?y);
             goal := (forall ?x ?y .
                       ordered (?x :: (z :: M)) &
                       ?y in (z :: M)
                       ==> x <E ?y);
             transitive := (!prove <EL.left-transitive)}
        conclude goal
          pick-any x y
            assume (ordered (x :: (z :: M)) & y in (z :: M))
              let {B1 := (x <E z & z <EL M & (ordered M));
                   B2 := (!chain->
                          [(ordered (x :: (z :: M)))
                       ==> (x <EL (z :: M) & ordered (z :: M))
                                            [nonempty]
                       ==> (x <EL (z :: M) & z <EL M & ordered M)
                                            [nonempty]
                       ==> B1               [<EL.nonempty]]);
                   B3 := (!chain->
                          [B1 ==> (ordered M) [prop-taut]]);
                   B4 := (!chain->
                          [B1
                       ==> (x <E z & z <EL M)    [prop-taut]
                       ==> (x <EL M)             [transitive]
                       ==> (x <EL M & ordered M) [augment]
                       ==> (ordered (x :: M))    [nonempty]]);
                   B4 := (!chain->
                          [(y in (z :: M))
                           ==> (y = z | y in M) [in.nonempty]])}
              (!cases (y = z | y in M)
                assume (y = z)
                  (!chain-> [B1 ==> (x <E z)  [left-and]
                                 ==> (x <E y)  [(y = z)]])
                (!chain
                 [(y in M)
                  ==> (ordered (x :: M) & y in M)  [augment]
                  ==> (x <E y)                     [ind-hyp]]))
      }
    | (val-of cons) =>
      pick-any L x
        let {A1 := (ordered L);
             A2 := (forall ?y . ?y in L ==> x <E ?y);
             BAIBF := (!prove <EL.before-all-implies-before-first)}
        assume (A1 & A2)
```

```
          (!chain-> [A2 ==> (x <EL L)                   [BAIBF]
                       ==> ((x <EL L) & A1)             [augment]
                       ==> (ordered (x :: L))           [nonempty]])
    | (val-of append) =>
      by-induction (adapt theorem) {
        nil =>
         conclude (forall ?M .
                    ordered nil & ordered ?M &
                    (forall ?x ?y . ?x in nil & ?y in ?M ==> ?x <E ?y)
                    ==> ordered (nil join ?M))
           pick-any M
             assume (ordered nil & ordered M &
                      (forall ?x ?y . ?x in nil & ?y in M ==> ?x <E ?y))
               (!chain->
                [(ordered M)
                 ==> (ordered (nil join M)) [join.left-empty]])
      | (z :: L:(List 'S)) =>
        let {ind-hyp :=
                (forall ?M .
                  (ordered L) & (ordered ?M) &
                  (forall ?x ?y . ?x in L & ?y in ?M ==> ?x <E ?y)
                  ==> (ordered (L join ?M)));
             goal :=
                (forall ?M .
                  ordered (z :: L) & ordered ?M &
                  (forall ?x ?y . ?x in (z :: L) & ?y in ?M ==> ?x <E ?y)
                  ==> (ordered ((z :: L) join ?M)));
             OLT := (!prove tail);
             ELA := (!prove <EL.append)}
        pick-any M:(List 'S)
          let {A1 := (ordered (z :: L));
               A2 := (ordered M);
               A3 := (forall ?x ?y .
                       ?x in (z :: L) & ?y in M ==> ?x <E ?y)}
          assume (A1 & A2 & A3)
            let {C1 := (!chain-> [A1 ==> (ordered L)    [OLT]]);
                 C2 := conclude (forall ?x ?y .
                                  ?x in L & ?y in M ==> ?x <E ?y)
                         pick-any x y
                           assume A4 := (x in L & y in M)
                             (!chain->
                              [A4 ==> (x in (z :: L) & y in M)
                                                          [in.tail]
                                  ==> (x <E y)            [A3]]);
                 C3 := (!chain->
                         [(ordered L)
                          ==> (ordered L &
                               ordered M & C2)      [augment]
                          ==> (ordered (L join M)) [ind-hyp]]);
                 C4 := conclude (z <EL M)
                         (!two-cases
                          assume (M = nil)
                            (!chain->
                             [true ==> (z <EL nil) [<EL.empty]
                                   ==> (z <EL M)    [(M = nil)]])
                          assume (M =/= nil)
                            let {D1 := conclude (z in (z :: L))
                                         (!chain->
                                          [(z = z)
                                           ==> (z = z | z in L) [alternate]
                                           ==> (z in (z :: L))
                                                              [in.nonempty]]);
                                 D2 := (exists ?u ?P . M = (?u :: ?P));
                                 D3 := conclude D2
                                         (!chain->
                                          [true
                                           ==> (M = nil | D2)
                                           [(datatype-axioms "List")]
                                           ==> (M =/= nil &
                                                (M = nil | D2)) [augment]
```

```
                                           ==> D2           [prop-taut]])}
                            pick-witnesses u P for D2
                              (!chain->
                               [true
                                ==> (u in (u :: P))     [in.head]
                                ==> (u in M)            [(M = u :: P)]
                                ==> (z in (z :: L) & u in M)    [augment]
                                ==> (z <E u)            [A3]
                                ==> (z <EL (u :: P)) [<EL.nonempty]
                                ==> (z <EL M)         [(M = u :: P)]]));
                 OLH := (!prove head)}
            conclude (ordered ((z :: L) join M))
              (!chain->
               [A1
                ==> (z <EL L)                       [OLH]
                ==> ((z <EL L) & C4)                [augment]
                ==> (z <EL (L join M))              [ELA]
                ==> ((z <EL (L join M)) & C3)       [augment]
                ==> (ordered (z :: (L join M)))     [nonempty]
                ==> (ordered ((z :: L) join M))     [join.left-nonempty]])
      }
    | (val-of append-2) =>
      by-induction (adapt theorem) {
        nil => pick-any M
                assume A := (ordered (nil join M))
                 let {goal := (ordered nil & ordered M);
                      B := (!chain->
                            [true ==> (ordered nil)
                                         [empty]])}
                 (!chain-> [A ==> (ordered M)      [join.left-empty]
                              ==> goal             [augment]])
      | (x :: L) =>
        pick-any M
         assume A := (ordered (x :: L) join M)
          let {goal := (ordered (x :: L) & ordered M);
               ind-hyp := (forall ?M .
                           (ordered (L join ?M)) ==>
                           (ordered L & ordered ?M));
               ELA := (!prove <EL.append-2)}
          (!chain->
           [A ==> (ordered (x :: (L join M)))     [join.left-nonempty]
              ==> (x <EL (L join M) & ordered (L join M))  [nonempty]
              ==> (x <EL L & ordered (L join M))           [ELA]
              ==> (x <EL L & ordered L & ordered M)        [ind-hyp]
              ==> ((x <EL L & ordered L) & ordered M)      [prop-taut]
              ==> goal                                     [nonempty]])
      }
    }

(add-theorems theory |{theorems := proofs}|)
} # ordered
} # SWO
```