lib/main/ordered-list-nat.ath

# Properties of ordered lists of natural numbers

load "nat-less.ath"
load "list-of.ath"

# <=L: is a natural number less than or equal to the first element of
# a list of natural numbers (true if the list is empty).

extend-module List {
  open N

declare <=L: [N (List N)] -> Boolean

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

define left-transitive :=
  (forall L x y . x <= y & y <=L L ==> x <=L L)

define before-all-implies-before-first :=
  (forall L x . (forall y . y in L ==> x <= y) ==> x <=L L)

define append :=
  (forall L M x . x <=L L & x <=L M ==> x <=L (L join M))

datatype-cases left-transitive {
  nil =>
    pick-any x y
      assume (x <= y & y <=L nil)
      (true ==> (x <=L nil) [empty])
  | (z::M) =>
    pick-any x y
      assume (x <= y & y <=L (z::M))
      conclude (x <=L (z::M))

    (true ==> (x <=L nil) [empty])
  |
}

datatype-cases before-all-implies-before-first {
  nil =>
    pick-any x
      assume (forall y . y in nil ==> x <= y)
      conclude (x <=L nil)

    (true ==> (x <=L nil) [empty])
  | (z:N::L) =>
    pick-any x
      assume (forall y . y in (z::L) ==> x <= y)
      conclude (x <=L (z::L))

    (true ==> (x <=L nil) [empty])
  |
}

datatype-cases append {
  nil =>
    pick-any M x
      (!chain [(x <=L nil & x <=L M) ==> (x <=L (nil join M)) [join.left-empty]])
  | (u::N) =>
    pick-any M x

assume (x <=L (u :: N) & (x <=L M))

{|chain->
  |(x <=L (u :: N))
  ==(>(x <= u) | [nonempty]
  ==(>(x <=L (u :: (N join M))) | [nonempty]
  ==(>(x <=L ((u :: N) join M)) | [join.left-nonempty])

# List.ordered: are the natural numbers in a list in order?

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

module ordered {
  assert empty := (ordered nil)
  assert nonempty :=
    (forall L x . ordered (x :: L) <=> x <=L L & ordered L)

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

  conclude head
  pick-any L x
    {|chain|
    ==(>(ordered (x :: L))
    ==(>(x <=L L & ordered L) | [nonempty]
    ==(>(x <=L L) | [left-and])}

  conclude tail
  pick-any L x
    {|chain|
    ==(>(ordered (x :: L))
    ==(>(x <=L L & ordered L) | [nonempty]
    ==(>(ordered L) | [right-and])}

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

  by-induction first-to-rest-relation {
    nil =>
    pick-any x:N y:N
      assume i := ((ordered (x :: nil)) & y in nil)
      let (not-in := (!chain->
        [true => (~ y in nil) | [in.empty]])
        |![from-complements (x <= y) (y in nil) not-in])
      | (z:N :: M:(List N)) =>
      let (ind-hyp := (forall ?x ?y .
        ordered (?x :: M) & ?y in M => ?x <= ?y))
      conclude (forall ?x ?y .
        ordered (?x :: (z :: M)) & ?y in (z :: M)
      => ?x <= ?y)
    pick-any x:N y:N
      assume ((ordered (x :: (z :: M))) & y in (z :: M))
      let (p0 :=
        (!chain->
        ==(>(ordered (x :: (z :: M)))
        ==(>(x <=L (z :: M) & ordered (z :: M))
        | [nonempty]
        ==(>(x <=L (z :: M) & z <=L M & ordered M)
        | [nonempty])


139  =>  (x <= z & z <=L M & ordered M)
"[<=L.nonempty])];
140 p1 := (!chain-> [p0 ==> (ordered M) [prop-taut])];
141 p2 := (!chain->
142 [p0 ==> (x <= z & z <=L M) [prop-taut]
143 => (x <=L M) [<=L.left-transitive]
144 ==> (x <=L M & ordered M) [augment]
145 =>> (ordered (x :: M)) [nonempty]));
146 p3 := (!chain->
147 [(y in (z :: M))
148 =>> (y = z | y in M) {in.nonempty})
149 (!cases (y = z | y in M)
150 assume (y = z)
151 (!chain-> [p0 ==> (x <= z) [left-and]
152 ==> (x <= y) {y = z}])
153 (!chain (y in M)
154 ==> (p2 & y in M) [augment]
155 ==> (x <= y) [ind-hyp]])}
156 )
157 conclude cons
158 pick-any L x
159 let {p := (forall ?y . ?y in L => x <= ?y)}
160 assume (ordered L & p)
161 (!chain->
162 [p ==> (x <=L L) [<=L.before-all-implies-before-first]
163 =>> (x <=L L & ordered L ) [augment]
164 ==> (ordered (x :: L)) [nonempty])}
165 by induction append {
166 nil:(List N) =>
167 conclude (forall ?R .
168 ordered nil & ordered ?R &
170 =>> (ordered (nil join ?R)))
171 pick-any R
172 assume ((ordered nil) & (ordered R) &
174 (!chain->
175 [ordered R]
176 ==> (ordered (nil join R)) [join.left-empty])
177 | (z :: L:(List N)) =>
178 let {ind-hyp :=
179 (forall ?R .
180 ordered L & ordered ?R &
182 =>> (ordered (L join ?R)))}
183 conclude (forall ?R .
184 ordered (z :: L) & ordered ?R &
186 =>> (ordered ((z :: L) join ?R)))
187 pick-any R:(List N)
188 let (A1 := (ordered (z :: L));
189 A2 := (ordered R);
190 A3 := (forall ?x ?y .
191 ?x in (z :: L) & ?y in R => ?x <= ?y))
191 assume (A1 & A2 & A3)
192 let {C1 := (!chain->
193 (ordered (z :: L)) ==> (ordered L) [tail])};
195 pick-any x:N y:N
196 assume D := (x in L & y in R)
197 (!chain->
198 [D ==> (x <= y) [in.tail]
199 =>> (x <= y) [A3]])
200 C3 := conclude (ordered (L join R))
\[\begin{align*}
\text{assume} & \quad (R = \text{nil}) \\
\implies & \quad (z = \text{nil}) \quad (\text{nil} \implies z) \\
\text{assume} & \quad (R \neq \text{nil}) \\
\text{let} & \quad (D1) := \text{conclude} \quad (z = z) \\
& \quad \text{conclude} \quad (z \in (z :: L)) \\
& \quad \text{pick-witnesses} \quad u \ M \text{ for } D3 \\
\text{conclude} & \quad (z :: L) \text{ join } R)
\end{align*}\]