lib/search/binary-search-tree-nat.ath

```
# Binary search trees, a subset of binary trees defined by a
# predicate, BST.

load "ordered-list-nat"
load "binary-tree"
#--------------------------------------------------------------------------

extend-module BinTree {

define [x y z T L R] := [?x:N ?y:N ?z:N ?T:(BinTree N) ?L:(BinTree N)
                         ?R:(BinTree N)]

define [< <=] := [N.< N.<=]

declare BST: [(BinTree N)] -> Boolean

declare no-smaller, no-larger: [(BinTree N) N] -> Boolean

assert* no-smaller-def :=
  [(null no-smaller _)
   ((node L y R) no-smaller x <==> x <= y &
                                   L no-smaller x &
                                   R no-smaller x)]

assert* no-larger-def :=
  [(null no-larger _)
   ((node L y R) no-larger x <==> y <= x &
                                  L no-larger x &
                                  R no-larger x)]

module BST {

assert* definition :=
  [(BST null)
   (BST (node L x R) <==> BST L & L no-larger x &
                              BST R & R no-smaller x)]

# Characterization properties:

assert empty := (BST null)

assert nonempty :=
  (forall L y R .
    BST (node L y R) <==> BST L & (forall x . x in L ==> x <= y) &
                          BST R & (forall z . z in R ==> y <= z))

# Though asserted here, empty and nonempty follow from no-smaller-def
# and no-larger-def. The proof is an exercise in the textbook.

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

# Theorem: the inorder function applied to a binary search tree
# produces an ordered list.  (Proved here only for natural number
# elements.)

define ordered := List.ordered

define is-ordered :=
  (forall T . BST T ==> (ordered (inorder T)))

by-induction is-ordered {
  null:(BinTree N) =>
  assume (BST null)
    (!chain->
     [true ==> (ordered nil:(List N)) [empty]
           ==> (ordered (inorder null:(BinTree N))) [inorder.empty]])
| (node L:(BinTree N) y:N R:(BinTree N)) =>
```

```
  let {ind-hyp1 := ((BST L) ==> (ordered inorder L));
       ind-hyp2 := ((BST R) ==> (ordered inorder R));
       smaller-in-left := (forall ?x . ?x in L ==> ?x <= y);
       larger-in-right := (forall ?z . ?z in R ==> y <= ?z)}
  assume A := (BST (node L y R))
    conclude goal := (ordered (inorder (node L y R)))
      let {C1 := (!chain->
                  [A ==>
                   ((BST L) & smaller-in-left & (BST R) & larger-in-right)
                                                 [nonempty]]);
           C2 := (!chain-> [C1 ==> (BST L)    [left-and]
                               ==> (ordered inorder L) [ind-hyp1]]);
           C3 := (!chain-> [C1 ==> (BST R)    [prop-taut]
                               ==> (ordered inorder R) [ind-hyp2]]);
           C4 := (!chain-> [C1 ==> smaller-in-left  [prop-taut]]);
           C5 := (!chain-> [C1 ==> larger-in-right  [prop-taut]]);
           C6 := conclude
                   (forall ?x ?y .
                     ?x in inorder L & ?y in (y :: inorder R)
                     ==> ?x <= ?y)
                   pick-any u v
                     assume A1 := (u in inorder L &
                                   v in (y :: inorder R))
                       let {D1 :=
                             (!chain->
                              [A1 ==> (u in inorder L &
                                       (v = y | v in inorder R))
                                           [List.in.nonempty]
                                  ==> (u in L & (v = y | v in R))
                                           [inorder.in-correctness]
                                  ==> ((u in L & v = y) |
                                       (u in L & v in R)) [prop-taut]])}
                       (!cases D1
                        assume (u in L & v = y)
                          (!chain->
                           [(u in L) ==> (u <= y) [smaller-in-left]
                                     ==> (u <= v) [(v = y)]])
                        (!chain [(u in L & v in R)
                                 ==> (u <= y & y <= v) [smaller-in-left
                                                        larger-in-right]
                                 ==> (u <= v)        [Less=.transitive]]));
           C7 := conclude (forall ?z . ?z in inorder R ==> y <= ?z)
                   pick-any z
                   (!chain [(z in inorder R)
                            ==> (z in R)       [inorder.in-correctness]
                            ==> (y <= z)       [larger-in-right]])}
      (!chain->
       [C3 ==> (C3 & C7)                   [augment]
           ==> (ordered (y :: inorder R)) [List.ordered.cons]
           ==> (C2 & (ordered (y :: inorder R)))       [augment]
           ==> (C2 & (ordered (y :: inorder R)) & C6)  [augment]
           ==> (ordered ((inorder L) join (y :: inorder R)))
                                            [List.ordered.append]
           ==> goal                         [inorder.nonempty]])
}
} # BST
} # BinTree
```