lib/search/ordered-inorder.ath

```
load "binary-search-tree"

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

# Theorem: the inorder function applied to a binary search tree
# produces an ordered list.

# Needs repair.

extend-module SWO {
extend-module BST {

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

define proof :=
  method (theorem adapt)
    let {lemma := method (P) (!property P adapt Theory);
         given := lambda (P) (get-property P adapt Theory);
         chain := method (L) (!chain-help given L 'none);
         chain-> := method (L) (!chain-help given L 'last);
         [< <E ordered BST] := (adapt [< <E ordered BST])}
    match theorem {
      (val-of ordered-inorder) =>
        by-induction theorem {
          null =>
            assume (BST null)
              (!chain-> [(ordered nil) ==> (ordered (inorder null))
                                         [inorder.empty]])
        | (T as (node L y R)) =>
          let {ind-hyp1 := (BST L ==> ordered (inorder L));
               ind-hyp2 := (BST R ==> ordered (inorder R));
               smaller-in-left := (forall ?x . ?x in L ==> ?x <E y);
               larger-in-right := (forall ?z . ?z in R ==> y <E ?z);
               p0 := (BST L & smaller-in-left &
                      BST R & larger-in-right);
               p1 := (forall ?x ?y .
                       ?x List.in (inorder L) & ?y List.in (y :: (inorder R))
                       ==> ?x <E ?y);
               goal := (ordered (inorder T));
               ET := (!lemma <E-Transitive);
               OA := (!lemma ordered.append);
               OC := (!lemma ordered.cons)}
          conclude (BST T
                    ==> ordered (inorder T))
            assume i := (BST T)
              let {_ := (!chain-> [i ==> p0 [nonempty]]);
                   _ := (!chain->
                          [p0 ==> (BST L)    [prop-taut]
                              ==> (ordered (inorder L)) [ind-hyp1]]);
                   _ := (!chain->
                          [p0 ==> (BST R)    [prop-taut]
                              ==> (ordered (inorder R)) [ind-hyp2]]);
                   _ := (!chain-> [p0 ==> smaller-in-left  [prop-taut]]);
                   _ := (!chain-> [p0 ==> larger-in-right  [prop-taut]]);
                   _ := conclude p1
                          pick-any u v
                            assume ii := (u List.in (inorder L) &
                                          v List.in (y :: (inorder R)))
                              let {C := (!chain->
                                         [ii ==> (u List.in (inorder L) &
                                                  (v = y | v List.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 C
                               assume (u in L & v = y)
                                 (!chain->
                                  [(u in L) ==> (u <E y)  [smaller-in-left]
                                            ==> (u <E v)  [(v = y)]])
                               (!chain [(u in L & v in R)
                                        ==> (u <E y & y <E v) [smaller-in-left
                                                               larger-in-right]
                                        ==> (u <E v)          [ET]]));

                   iii := conclude (forall ?z . ?z in (inorder R) ==> y <E ?z)
                            pick-any z
                            (!chain [(z in (inorder R))
                                    ==> (z in R) [inorder.in-correctness]
                                    ==> (y <E z) [larger-in-right]])}
              conclude goal
                (!chain->
                 [(ordered (inorder R))
                  ==> (ordered (inorder R) & iii)        [augment]
                  ==> (ordered (y :: (inorder R)))       [OC]
                  ==> (ordered (inorder L) &
                       (ordered (y :: (inorder R))))     [augment]
                  ==> (ordered (inorder L) &
                       ordered (y :: (inorder R)) & p1)  [augment]
                  ==> (ordered ((inorder L) join (y :: (inorder R)))) [OA]
                  ==> goal                          [inorder.nonempty]])
        }
    }

(evolve Theory [[ordered-inorder] proof])

} # BST
} # SWO
```