lib/search/binary-search-tree1.ath

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

load "ordered-list"
load "binary-tree"

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

extend-module SWO {
open BinTree

#..........................................................................
declare BST: (S) [(BinTree S)] -> Boolean

module BST {

define in' := BinTree.in

define empty := (BST null)
define nonempty :=
  (forall L y R .
    BST (node L y R) <==>
     BST L & (forall x . x in' L ==> x <E y) &
     BST R & (forall z . z in' R ==> y <E z))

(evolve Theory [[empty nonempty] Definition])

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

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

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]])
        | (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 in (inorder L) & ?y in (y :: (inorder R))
                       ==> ?x <E ?y);
               goal := (ordered (inorder (node L y R)));
               ET := (!lemma <E-Transitive);
               OA := (!lemma ordered.append);
               OC := (!lemma ordered.cons)}
          conclude (BST (node L y R)
                    ==> ordered (inorder (node L y R)))
            assume i := (BST (node L y R))
              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 in (inorder L) &
                                          v in (y :: (inorder R)))
                              let {C := (!chain->
                                         [ii ==> (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 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])

#......................................................................
declare in: (S) [S (BinTree S)] -> Boolean

module in {

define empty := (forall x . ~ x in null)
define nonempty :=
  (forall x L y R .  x in (node L y R) <==> x E y | x in L | x in R)

(evolve Theory [[empty nonempty] Definition])

define root := (forall x L y R . x E y ==> x in (node L y R))
define left := (forall x L y R . x in L ==> x in (node L y R))
define right := (forall x L y R . x in R ==> x in (node L y R))

define proofs :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt Theory);
```

```
         [E in] := (adapt [E in])}
    match theorem {
      (val-of root) =>
       pick-any x L y R
         (!chain
          [(x E y) ==> (x E y | x in L | x in R)      [alternate]
                   ==> (x in (node L y R))            [nonempty]])
    | (val-of left) =>
      pick-any x L y R
        (!chain
         [(x in L) ==> (x in L | x in R)            [alternate]
                   ==> (x E y | x in L | x in R)    [alternate]
                   ==> (x in (node L y R))          [nonempty]])
    | (val-of right) =>
      pick-any x L y R
        assume (x in R)
          (!chain->
           [(x in R) ==> (x in L | x in R)          [alternate]
                     ==> (x E y | x in L | x in R)  [alternate]
                     ==> (x in (node L y R))        [nonempty]])
     }

(evolve Theory [[root left right] proofs])

define exists-equivalent :=
  (forall T x . x in T ==> (exists z . x E z & z in' T))

define characterization :=
  (forall L y R .
    BST (node L y R)
     ==> BST L & (forall x . x in L ==> x <E y) &
         BST R & (forall z . z in R ==> y <E z))

define lemmas := [exists-equivalent characterization]

define proofs :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt Theory);
         [< <E E in BST] := (adapt [< <E E in BST])}
    match theorem {
     (val-of exists-equivalent) =>
     by-induction (adapt theorem) {
       null =>
       pick-any x
         assume is-in := (x in null)
           let {is-not := (!chain->
                                [true ==> (~ (x in null)) [empty]])}
            (!from-complements
             (exists ?z . x E ?z & ?z in' null)
             is-in is-not)
     | (node L y R) =>
       pick-any x
         assume is-in := (x in (node L y R))
           let {ind-hyp1 := (forall ?x . ?x in L ==>
                                         exists ?z . ?x E ?z & ?z in' L);
                ind-hyp2 := (forall ?x . ?x in R ==>
                                         exists ?z . ?x E ?z & ?z in' R);
                goal := (exists ?z . x E ?z & ?z in' (node L y R));
                possibilities := (x E y | x in L | x in R);
                i := (!chain-> [is-in ==> possibilities [nonempty]])}
            (!cases possibilities
             assume ii := (x E y)
               (!chain->
                [(y = y) ==> (y in' (node L y R)) [BinTree.in.root]
                        ==> (ii & y in' (node L y R)) [augment]
                        ==> goal                      [existence]])
             assume iv := (x in L)
               let {v := (!chain->
                          [iv ==> (exists ?z . x E ?z & ?z in' L)
                              [ind-hyp1]])}
```

```
                pick-witness z for v v'
                  (!chain->
                   [v' ==> (x E z & (z in' (node L y R)))
                                    [BinTree.in.left]
                       ==> goal     [existence]])
              assume iv := (x in R)
                let {v := (!chain->
                            [iv ==> (exists ?z . x E ?z & ?z in' R)
                                [ind-hyp2]])}
                pick-witness z for v v'
                  (!chain->
                   [v' ==> (x E z & (z in' (node L y R)))  [in.right]
                       ==> goal  [existence]]))
        } # by-induction
      | (val-of characterization) =>
        pick-any L:(BinTree 'S) y:'S R:(BinTree 'S)
          assume i := (BST (node L y R))
            let {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);
                 _ := (!chain-> [i ==> p0 [nonempty]]);
                 _ := (!chain-> [p0 ==> (BST L)  [prop-taut]]);
                 _ := (!chain-> [p0 ==> (BST R)  [prop-taut]]);
                 _ := (!chain-> [p0 ==> smaller-in-left  [prop-taut]]);
                 _ := (!chain-> [p0 ==> larger-in-right  [prop-taut]]);
                 EE := (!prove exists-equivalent);
                 ET := (!prove <E-Transitive);
                 C := conclude (forall ?x . ?x in L ==> ?x <E y)
                        pick-any x
                          let {ex := (exists ?x' . x E ?x' & ?x' in' L)}
                          assume ii := (x in L)
                            let {_ := (!chain-> [ii ==> ex [EE]])}
                            pick-witness x' for ex
                              conclude (x <E y)
                                (!chain->
                                 [(x E x' & x' in' L)
                                  ==> (x E x' & x' <E y)      [smaller-in-left]
                                  ==> ((~ (x < x') & ~ (x' < x)) & x' <E y)
                                  [E-Definition]
                                  ==> (~ (x' < x) & x' <E y)  [prop-taut]
                                  ==> (x <E x' & x' <E y)     [<E-Definition]
                                  ==> (x <E y)                [ET]]);
                 D := conclude (forall ?z . ?z in R ==> y <E ?z)
                        pick-any z
                          let {ex := (exists ?z' . z E ?z' & ?z' in' R)}
                          assume ii := (z in R)
                            let {_ := (!chain-> [ii ==> ex [EE]])}
                            pick-witness z' for ex
                              conclude (y <E z)
                                (!chain->
                                 [(z E z' & z' in' R)
                                  ==> (z E z' & y <E z')      [larger-in-right]
                                  ==> ((~ (z < z') & ~ (z' < z)) & y <E z')
                                  [E-Definition]
                                  ==> (y <E z' & ~ (z < z'))  [prop-taut]
                                  ==> (y <E z' & z' <E z)     [<E-Definition]
                                  ==> (y <E z)                [ET]])}
              (!both (BST L) (!both C (!both (BST R) D)))
      } # match theorem

(evolve Theory [lemmas proofs])
} # in
} # BST
} # SWO
```