lib/search/BST-theorems.ath

```
load "binary-search-tree"

# Needs repair.

extend-module SWO {
extend-module BST {

#.....................................................................
extend-module in {

define in' := BinTree.in

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 {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 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)))  [BinTree.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 := (!lemma exists-equivalent);
               ET := (!lemma <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
```