

lib/search/binary-search-tree1.ath

```

1  # Binary search trees, a subset of binary trees defined by a
2  # predicate, BST
3
4  load "ordered-list"
5  load "binary-tree"
6
7  #-----
8
9  extend-module SWO {
10 open BinTree
11
12 #.....
13 declare BST: (S) [(BinTree S)] -> Boolean
14
15 module BST {
16
17   define in' := BinTree.in
18
19   define empty := (BST null)
20   define nonempty :=
21     (forall L y R .
22       BST (node L y R) <==>
23         BST L & (forall x . x in' L ==> x <E y) &
24         BST R & (forall z . z in' R ==> y <E z))
25
26   (evolve Theory [[empty nonempty] Definition])
27
28   #-----
29
30   # Theorem: the inorder function applied to a binary search tree
31   # produces an ordered list.
32
33   define ordered-inorder :=
34     (forall T . BST T ==> (ordered (inorder T)))
35
36   define proof :=
37     method (theorem adapt)
38       let {lemma := method (P) (!property P adapt Theory);
39           given := lambda (P) (get-property P adapt Theory);
40           chain := method (L) (!chain-help given L 'none);
41           chain-> := method (L) (!chain-help given L 'last);
42           [< <E ordered BST] := (adapt [< <E ordered BST])}
43       match theorem {
44         (val-of ordered-inorder) =>
45         by-induction theorem {
46           null =>
47             assume (BST null)
48               (!chain-> [(ordered nil) ==> (ordered (inorder null))
49                           [inorder.empty]])
50         | (node L y R) =>
51           let {ind-hyp1 := (BST L ==> ordered (inorder L));
52               ind-hyp2 := (BST R ==> ordered (inorder R));
53               smaller-in-left := (forall ?x . ?x in L ==> ?x <E y);
54               larger-in-right := (forall ?z . ?z in R ==> y <E ?z);
55               p0 := (BST L & smaller-in-left &
56                     BST R & larger-in-right);
57               p1 := (forall ?x ?y .
58                     ?x in (inorder L) & ?y in (y :: (inorder R))
59                     ==> ?x <E ?y);
60               goal := (ordered (inorder (node L y R)));
61               ET := (!lemma <E-Transitive);
62               OA := (!lemma ordered.append);
63               OC := (!lemma ordered.cons)}
64         conclude (BST (node L y R)
65                   ==> ordered (inorder (node L y R)))
66         assume i := (BST (node L y R))
67         let {_ := (!chain-> [i ==> p0 [nonempty]])};

```

```

68   _ := (!chain->
69     [p0 ==> (BST L)      [prop-taut]
70     ==> (ordered (inorder L)) [ind-hyp1]]);
71   _ := (!chain->
72     [p0 ==> (BST R)      [prop-taut]
73     ==> (ordered (inorder R)) [ind-hyp2]]);
74   _ := (!chain-> [p0 ==> smaller-in-left [prop-taut]]);
75   _ := (!chain-> [p0 ==> larger-in-right [prop-taut]]);
76   _ := conclude p1
77     pick-any u v
78     assume ii := (u in (inorder L) &
79       v in (y :: (inorder R)))
80     let {C := (!chain->
81       [ii ==> (u in (inorder L) &
82         (v = y | v in (inorder R)))
83       [List.in.nonempty]
84       ==> (u in L & (v = y | v in R))
85       [inorder.in-correctness]
86       ==> ((u in L & v = y) |
87         (u in L & v in R))
88       [prop-taut]])}
89     (!cases C
90       assume (u in L & v = y)
91       (!chain->
92         [(u in L) ==> (u <E y) [smaller-in-left]
93         ==> (u <E v) [(v = y)]]
94       (!chain [(u in L & v in R)
95         ==> (u <E y & y <E v) [smaller-in-left
96         larger-in-right]
97         ==> (u <E v) [ET]]));
98
99   iii := conclude (forall ?z . ?z in (inorder R) ==> y <E ?z)
100     pick-any z
101     (!chain [(z in (inorder R))
102       ==> (z in R) [inorder.in-correctness]
103       ==> (y <E z) [larger-in-right]])
104   conclude goal
105     (!chain->
106       [(ordered (inorder R))
107       ==> (ordered (inorder R) & iii) [augment]
108       ==> (ordered (y :: (inorder R))) [OC]
109       ==> (ordered (inorder L) &
110         (ordered (y :: (inorder R)))) [augment]
111       ==> (ordered (inorder L) &
112         ordered (y :: (inorder R)) & p1) [augment]
113       ==> (ordered ((inorder L) join (y :: (inorder R)))) [OA]
114       ==> goal [inorder.nonempty]])
115     }
116   }
117
118 (evolve Theory [[ordered-inorder] proof])
119
120 #.....
121 declare in: (S) [S (BinTree S)] -> Boolean
122
123 module in {
124
125   define empty := (forall x . ~ x in null)
126   define nonempty :=
127     (forall x L y R . x in (node L y R) <==> x E y | x in L | x in R)
128
129   (evolve Theory [[empty nonempty] Definition])
130
131   define root := (forall x L y R . x E y ==> x in (node L y R))
132   define left := (forall x L y R . x in L ==> x in (node L y R))
133   define right := (forall x L y R . x in R ==> x in (node L y R))
134
135   define proofs :=
136     method (theorem adapt)
137     let {[get prove chain chain-> chain<-] := (proof-tools adapt Theory);

```

```

138     [E in] := (adapt [E in])}
139 match theorem {
140   (val-of root) =>
141     pick-any x L y R
142     (!chain
143       [(x E y) ==> (x E y | x in L | x in R)      [alternate]
144                 ==> (x in (node L y R))          [nonempty]])
145 | (val-of left) =>
146   pick-any x L y R
147   (!chain
148     [(x in L) ==> (x in L | x in R)              [alternate]
149               ==> (x E y | x in L | x in R)      [alternate]
150               ==> (x in (node L y R))          [nonempty]])
151 | (val-of right) =>
152   pick-any x L y R
153   assume (x in R)
154   (!chain->
155     [(x in R) ==> (x in L | x in R)              [alternate]
156               ==> (x E y | x in L | x in R)      [alternate]
157               ==> (x in (node L y R))          [nonempty]])
158 }
159
160 (evolve Theory [[root left right] proofs])
161
162 define exists-equivalent :=
163   (forall T x . x in T ==> (exists z . x E z & z in' T))
164
165 define characterization :=
166   (forall L y R .
167     BST (node L y R)
168     ==> BST L & (forall x . x in L ==> x <E y) &
169     BST R & (forall z . z in R ==> y <E z))
170
171 define lemmas := [exists-equivalent characterization]
172
173 define proofs :=
174   method (theorem adapt)
175     let {[get prove chain chain-> chain<-] := (proof-tools adapt Theory);
176         [< <E E in BST] := (adapt [< <E E in BST])}
177   match theorem {
178     (val-of exists-equivalent) =>
179     by-induction (adapt theorem) {
180       null =>
181         pick-any x
182         assume is-in := (x in null)
183         let {is-not := (!chain->
184           [true ==> (~ (x in null)) [empty]])}
185         (!from-complements
186           (exists ?z . x E ?z & ?z in' null)
187           is-in is-not)
188 | (node L y R) =>
189   pick-any x
190   assume is-in := (x in (node L y R))
191   let {ind-hyp1 := (forall ?x . ?x in L ==>
192     exists ?z . ?x E ?z & ?z in' L);
193       ind-hyp2 := (forall ?x . ?x in R ==>
194     exists ?z . ?x E ?z & ?z in' R);
195       goal := (exists ?z . x E ?z & ?z in' (node L y R));
196       possibilities := (x E y | x in L | x in R);
197       i := (!chain-> [is-in ==> possibilities [nonempty]])}
198   (!cases possibilities
199     assume ii := (x E y)
200     (!chain->
201       [(y = y) ==> (y in' (node L y R)) [BinTree.in.root]
202               ==> (ii & y in' (node L y R)) [augment]
203               ==> goal                      [existence]])
204     assume iv := (x in L)
205     let {v := (!chain->
206       [iv ==> (exists ?z . x E ?z & ?z in' L)
207         [ind-hyp1]])}

```

```

208     pick-witness z for v v'
209     (!chain->
210       [v' ==> (x E z & (z in' (node L y R)))
211               [BinTree.in.left]
212               ==> goal [existence]])
213   assume iv := (x in R)
214   let {v := (!chain->
215     [iv ==> (exists ?z . x E ?z & ?z in' R)
216           [ind-hyp2]])}
217   pick-witness z for v v'
218   (!chain->
219     [v' ==> (x E z & (z in' (node L y R))) [in.right]
220           ==> goal [existence]])
221 } # by-induction
222 | (val-of characterization) =>
223 pick-any L:(BinTree 'S) y:'S R:(BinTree 'S)
224 assume i := (BST (node L y R))
225 let {smaller-in-left := (forall ?x . ?x in' L ==> ?x <E y);
226     larger-in-right := (forall ?z . ?z in' R ==> y <E ?z);
227     p0 := (BST L & smaller-in-left &
228           BST R & larger-in-right);
229     _ := (!chain-> [i ==> p0 [nonempty]]);
230     _ := (!chain-> [p0 ==> (BST L) [prop-taut]]);
231     _ := (!chain-> [p0 ==> (BST R) [prop-taut]]);
232     _ := (!chain-> [p0 ==> smaller-in-left [prop-taut]]);
233     _ := (!chain-> [p0 ==> larger-in-right [prop-taut]]);
234     EE := (!prove exists-equivalent);
235     ET := (!prove <E-Transitive);
236     C := conclude (forall ?x . ?x in L ==> ?x <E y)
237         pick-any x
238         let {ex := (exists ?x' . x E ?x' & ?x' in' L)}
239         assume ii := (x in L)
240         let {_ := (!chain-> [ii ==> ex [EE]])}
241         pick-witness x' for ex
242         conclude (x <E y)
243         (!chain->
244           [(x E x' & x' in' L)
245            ==> (x E x' & x' <E y) [smaller-in-left]
246            ==> ((~ (x < x')) & ~ (x' < x)) & x' <E y)
247            [E-Definition]
248            ==> (~ (x' < x) & x' <E y) [prop-taut]
249            ==> (x <E x' & x' <E y) [<E-Definition]
250            ==> (x <E y) [ET]])
251     D := conclude (forall ?z . ?z in R ==> y <E ?z)
252         pick-any z
253         let {ex := (exists ?z' . z E ?z' & ?z' in' R)}
254         assume ii := (z in R)
255         let {_ := (!chain-> [ii ==> ex [EE]])}
256         pick-witness z' for ex
257         conclude (y <E z)
258         (!chain->
259           [(z E z' & z' in' R)
260            ==> (z E z' & y <E z') [larger-in-right]
261            ==> ((~ (z < z')) & ~ (z' < z)) & y <E z')
262            [E-Definition]
263            ==> (y <E z' & ~ (z < z')) [prop-taut]
264            ==> (y <E z' & z' <E z) [<E-Definition]
265            ==> (y <E z) [ET]])
266     (!both (BST L) (!both C (!both (BST R) D)))
267 } # match theorem
268
269 (evolve Theory [lemmas proofs])
270 } # in
271 } # BST
272 } # SWO

```