

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

```

1 # Binary search function for searching in a binary search tree (here
2 # restricted to natural number elements) and correctness theorems.
3
4 load "search/binary-search-tree-nat"
5
6 #-----
7
8 extend-module BinTree {
9
10 declare binary-search: [N (BinTree N)] -> (BinTree N)
11
12 module binary-search {
13
14 define (axioms as [at-root go-left go-right empty]) :=
15   (fun
16     [(binary-search x (node L y R)) =
17       [(node L y R)           when (x = y)
18        (binary-search x L)    when (x < y)
19        (binary-search x R)    when (x /= y & ~ x < y)]
20     (binary-search x null) = null])
21
22 assert axioms
23
24 define found :=
25   (forall T . BST T ==>
26     forall x L y R . (binary-search x T) = (node L y R) ==> x = y & x in T)
27
28 define not-found :=
29   (forall T . BST T ==> forall x . (binary-search x T) = null ==> ~ x in T)
30
31 define tree-axioms := (datatype-axioms "BinTree")
32
33 define (binary-search-found-base) :=
34   conclude (BST null ==>
35     forall x L y R .
36       (binary-search x null) = (node L y R)
37       ==> x = y & x in null)
38   assume (BST null)
39     pick-any x:N L:(BinTree N) y:N R:(BinTree N)
40     assume i := ((binary-search x null) = (node L y R))
41     let {A := (!chain [null:(BinTree N)
42                       = (binary-search x null)      [empty]
43                       = (node L y R)                [i]]);
44         B := (!chain-> [true
45                       ==> (null /= (node L y R)) [tree-axioms]])}
46     (!from-complements (x = y & x in null) A B)
47
48 (!binary-search-found-base)
49
50 define [x1 y1 L1 R1] := [?x1:N ?y1:N ?L1:(BinTree N) ?R1:(BinTree N)]
51
52 define (found-property T) :=
53 (forall x L1 y1 R1 .
54   (binary-search x T) = (node L1 y1 R1) ==> x = y1 & x in T)
55
56 define binary-search-found-step :=
57   method (T)
58     match T {
59       (node L:(BinTree N) y:N R:(BinTree N)) =>
60         let {[ind-hyp1 ind-hyp2] := [(BST L ==> found-property L)
61                                     (BST R ==> found-property R)]}
62         assume hyp := (BST T)
63         conclude (found-property T)
64         let {p0 := (BST L &
65                   (forall x . x in L ==> x <= y) &
66                   BST R &
67                   (forall z . z in R ==> y <= z));}

```

```

68     _ := (!chain-> [hyp ==> p0                                [BST.nonempty]]);
69     fpl := (!chain-> [p0 ==> (BST L)                          [prop-taut]
70                    ==> (found-property L) [ind-hyp1]]);
71     fpr := (!chain-> [p0 ==> (BST R)                          [prop-taut]
72                    ==> (found-property R) [ind-hyp2]]);
73
74 pick-any x:N L1 y1:N R1
75 let {subtree := (node L1 y1 R1)}
76 assume hyp' := ((binary-search x T) = subtree)
77 conclude (x = y1 & x in T)
78   (!two-cases
79     assume (x = y)
80     (!both conclude (x = y1)
81       (!chain->
82         [T = (binary-search x T) [at-root]
83          = subtree [hyp']
84          ==> (y = y1) [tree-axioms]
85          ==> (x = y1) [(x = y)]]
86       conclude (x in T)
87         (!chain-> [(x = y)
88                  ==> (x in T) [in.root]]))
89     assume (x /= y)
90     (!two-cases
91       assume (x < y)
92       (!chain-> [(binary-search x L)
93                = (binary-search x T) [go-left]
94                = subtree [hyp']
95                ==> (x = y1 & x in L) [fpl]
96                ==> (x = y1 & x in T) [in.left]])
97       assume (~ x < y)
98       (!chain-> [(binary-search x R)
99                = (binary-search x T) [go-right]
100               = subtree [hyp']
101               ==> (x = y1 & x in R) [fpr]
102               ==> (x = y1 & x in T) [in.right]]))
103   }
104
105 by-induction found {
106   null => (!binary-search-found-base)
107   | (node L y:N R) => (!binary-search-found-step (node L y R))
108 }
109
110 define (not-found-prop T) :=
111   (forall x . (binary-search x T) = null ==> ~ x in T)
112
113 by-induction not-found {
114   null =>
115     assume (BST null)
116     conclude (not-found-prop null)
117     pick-any x:N
118     assume ((binary-search x null) = null)
119     (!chain-> [true ==> (~ x in null) [in.empty]])
120   | (T as (node L y:N R)) =>
121     let {p1 := (not-found-prop L);
122          p2 := (not-found-prop R);
123          [ind-hyp1 ind-hyp2] := [(BST L ==> p1) (BST R ==> p2)]}
124     assume hyp := (BST T)
125     conclude (not-found-prop T)
126     let {smaller-in-left := (forall x . x in L ==> x <= y);
127          larger-in-right := (forall z . z in R ==> y <= z);
128          p0 := (BST L &
129               smaller-in-left &
130               BST R &
131               larger-in-right);
132          _ := (!chain-> [hyp ==> p0                                [BST.nonempty]]);
133          _ := (!chain-> [p0 ==> smaller-in-left [prop-taut]]);
134          _ := (!chain-> [p0 ==> larger-in-right [prop-taut]]);
135          _ := (!chain-> [p0
136                       ==> (BST L) [prop-taut]
137                       ==> (not-found-prop L) [ind-hyp1]]);
138          _ := (!chain-> [p0

```

```

138             ==> (BST R)                                [prop-taut]
139             ==> (not-found-prop R)                    [ind-hyp2]])}
140 pick-any x
141   assume hyp' := ((binary-search x T) = null)
142   (!by-contradiction (~ x in T)
143     assume (x in T)
144       let {disj := (!chain-> [(x in T)
145                             ==> (x = y |
146                                 x in L |
147                                 x in R)          [in.nonempty]])}
148     (!two-cases
149       assume (x = y)
150       (!absurd
151         (!chain [null:(BinTree N)
152                 = (binary-search x T)    [hyp']
153                 = T                      [at-root]])
154         (!chain-> [true
155                  ==> (null /= T)         [tree-axioms]]))
156       assume (x /= y)
157       (!two-cases
158         assume (x < y)
159         (!cases disj
160           assume (x = y)
161             (!absurd (x = y) (x /= y))
162           assume (x in L)
163             (!absurd
164               (x in L)
165               (!chain->
166                 [(binary-search x L)
167                  = (binary-search x T)    [go-left]
168                  = null                    [hyp']
169                  ==> (~ x in L)          [p1]]))
170             assume (x in R)
171               (!absurd
172                 (x < y)
173                 (!chain->
174                   [(x in R)
175                    ==> (y <= x)          [larger-in-right]
176                    ==> (~ x < y)        [N.Less=.trichotomy4]]))
177             assume (~ x < y)
178             (!cases disj
179               assume (x = y)
180                 (!absurd (x = y) (x /= y))
181               assume (x in L)
182                 (!absurd
183                   (x /= y)
184                   (!chain-> [(x in L)
185                             ==> (x <= y)    [smaller-in-left]
186                             ==> (x < y | x = y) [N.Less=.definition]
187                             ==> (~ x < y &
188                                   (x < y | x = y)) [augment]
189                             ==> (x = y)        [prop-taut]]))
190               assume (x in R)
191                 (!absurd
192                   (x in R)
193                   (!chain->
194                     [(binary-search x R)
195                      = (binary-search x T)    [go-right]
196                      = null                    [hyp']
197                      ==> (~ x in R)          [p2]]))))))
198 } # by-induction
199
200 #.....
201 # Converse of binary-search.not-found follows from
202 # binary-search.found:
203 define not-in-implies-null-result :=
204   (forall T .
205     BST T ==> forall x . ~ x in T ==> (binary-search x T) = null)
206
207 conclude not-in-implies-null-result

```

```

208 pick-any T:(BinTree N)
209   assume (BST T)
210   pick-any x:N
211     assume (~ x in T)
212       (!by-contradiction ((binary-search x T) = null)
213         assume ii := ((binary-search x T) /= null)
214           let {p := (exists L y R .
215             (binary-search x T) = (node L y R));
216             _ := (!chain->
217               [true
218                 ==> ((binary-search x T) = null | p) [tree-axioms]
219                 ==> p [dsyl with ii])]}
220         pick-witnesses L y R for p p'
221         let {_ := (!chain-> [p' ==> (x = y & x in T) [found]
222           ==> (x in T) [right-and]])}
223         (!absurd (x in T) (~ x in T))
224
225 #.....
226 # Combining the implications:
227 define not-found-iff-not-in :=
228   (forall T .
229     BST T ==> forall x . (binary-search x T) = null <==> ~ x in T)
230
231 conclude not-found-iff-not-in
232 pick-any T:(BinTree N)
233   assume (BST T)
234   pick-any x:N
235     let {A := (!chain
236       [(binary-search x T) = null] ==> (~ x in T)
237       [not-found]);
238       B := (!chain
239         [(~ x in T) ==> ((binary-search x T) = null)
240         [not-in-implies-null-result])}
241     (!equiv A B)
242 #.....
243 define in-implies-node-result :=
244   (forall T .
245     BST T ==>
246     forall x .
247     x in T ==> exists L R . (binary-search x T) = (node L x R))
248
249 conclude in-implies-node-result
250 pick-any T:(BinTree N)
251   assume (BST T)
252   pick-any x:N
253   assume (x in T)
254   let {p := (exists L y R .
255     (binary-search x T) = (node L y R));
256     q := ((binary-search x T) /= null);
257     _ := (!by-contradiction q
258       assume i := ((binary-search x T) = null)
259         let {_ := (!chain->
260           [i ==> (~ x in T) [not-found]])}
261         (!absurd (x in T) (~ x in T)));
262     _ := (!chain->
263       [true
264         ==> ((binary-search x T) = null | p) [tree-axioms]
265         ==> p [dsyl with q])]}
266   pick-witnesses L y R for p p'
267   let {_ := (!chain->
268     [(binary-search x T)
269     = (node L y R) [p']
270     ==> (x = y) [found left-and]])}
271   (!chain->
272     [(binary-search x T)
273     = (node L y R) [p']
274     = (node L x R) [(x = y)]
275     ==> (exists L R .
276       (binary-search x T) = (node L x R))
277     [existence]])}

```

```
278 } # binary-search  
279 } # BinTree
```