lib/search/binary-search-nat0.ath

```
# Binary search function for searching in a binary search tree (here
# restricted to natural number elements) and correctness theorems.

load "search/binary-search-tree-nat"

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

extend-module BinTree {

declare binary-search: [N (BinTree N)] -> (BinTree N)

module binary-search {

define (axioms as [at-root go-left go-right empty]) :=
 (fun
  [(binary-search x (node L y R)) =
       [(node L y R)          when (x = y)
        (binary-search x L)   when (x < y)
        (binary-search x R)   when (x =/= y & ~ x < y)]
   (binary-search x null) = null])

assert axioms

define found :=
  (forall T .
    BST T ==>
      forall x L y R .
        (binary-search x T) = (node L y R) ==> x = y & x in T)

define not-found :=
  (forall T .
    BST T ==>
      forall x . (binary-search x T) = null ==> ~ x in T)

define binary-search-found-base :=
 method ()
  conclude (BST null ==>
             (forall x L y R .
                (binary-search x null) = (node L y R) ==>
                x = y & x in null))
    assume (BST null)
      pick-any x:N L:(BinTree N) y:N R:(BinTree N)
        assume i := ((binary-search x null) = (node L y R))
          let {A := conclude (null = (node L y R))
                      (!chain
                       [null:(BinTree N) = (binary-search x null)
                                           [empty]
                        = (node L y R)       [i]]);
               B := (!chain-> [true ==> (null =/= (node L y R))
                               [(exclusive-constructors "BinTree")]])}
           (!from-complements (x = y & x in null) A B)

define binary-search-found-step :=
 method (T)
  match T {
    (node L:(BinTree N) y:N R:(BinTree N)) =>
      let {p1 := (forall ?x ?L1 ?y1 ?R1 .
                   (binary-search ?x L) = (node ?L1 ?y1 ?R1) ==>
                   ?x = ?y1 & ?x in L);
           p2 := (forall ?x ?L1 ?y1 ?R1 .
                   (binary-search ?x R) = (node ?L1 ?y1 ?R1) ==>
                   ?x = ?y1 & ?x in R);
           ind-hyp1 := (BST L ==> p1);
           ind-hyp2 := (BST R ==> p2)}
      assume i := (BST (node L y R))
           conclude
             (forall ?x ?L1 ?y1 ?R1 .
```

```
68                  (binary-search ?x (node L y R)) = (node ?L1 ?y1 ?R1)
69                      ==> ?x = ?y1 & ?x in (node L y R))
70              let {
71                p0 := (BST L & (forall ?x . ?x in L ==> ?x <= y) &
72                        BST R & (forall ?z . ?z in R ==> y <= ?z));
73                _ := (!chain-> [i ==> p0 [BST.nonempty]]);
74                _ := (!chain-> [p0 ==> (BST L) [prop-taut]
75                                   ==> p1      [ind-hyp1]]);
76                _ := (!chain-> [p0 ==> (BST R) [prop-taut]
77                                   ==> p2      [ind-hyp2]])}
78              pick-any x:N L1 y1:N R1
79                assume p3 := ((binary-search x (node L y R)) =
80                              (node L1 y1 R1))
81                  conclude (x = y1 & x in (node L y R))
82                    (!two-cases
83                      assume (x = y)
84                        let {A := conclude (x = y1)
85                                    (!chain->
86                                     [(node L y R)
87                                     = (binary-search x (node L y R))
88                                        [at-root]
89                                     = (node L1 y1 R1) [p3]
90                                     ==> (& (L = L1) (y = y1) (R = R1))
91                                         [(datatype-axioms "BinTree")]
92                                     ==> (y = y1)
93                                         [(compose right-and left-and)]
94                                     ==> (x = y1)   [(x = y)]]);
95                             B := conclude (x in (node L y R))
96                                    (!chain->
97                                     [(x = y) ==> (x in (node L y R))
98                                                  [in.root]])}
99                        (!both A B)
100                     assume (x =/= y)
101                       (!two-cases
102                         assume (x < y)
103                           (!chain->
104                             [(binary-search x L)
105                              = (binary-search x (node L y R))
106                                        [go-left]
107                              = (node L1 y1 R1)                [p3]
108                              ==> (x = y1 & x in L)            [p1]
109                              ==> (x = y1 & x in (node L y R))
110                                             [in.left]])
111                         assume (~ x < y)
112                           (!chain->
113                            [(binary-search x R)
114                             = (binary-search x (node L y R))
115                                    [go-right]
116                             = (node L1 y1 R1)              [p3]
117                             ==> (x = y1 & x in R)          [p2]
118                             ==> (x = y1 & x in (node L y R))
119                                     [in.right]])))
120     }
121
122 by-induction found {
123   null => (!binary-search-found-base)
124 | (node L y:N R) => (!binary-search-found-step (node L y R))
125 }
126
127 #.....................................................................
128 #
129 by-induction not-found {
130   null =>
131     assume (BST null)
132       conclude (forall ?x .
133                   (binary-search ?x null) = null ==> ~ ?x in null)
134         pick-any x:N
135           assume ((binary-search x null) = null)
136             (!chain-> [true ==> (~ x in null) [in.empty]])
137 | (node L y:N R) =>
```

```
138     let {p1 := (forall ?x . (binary-search ?x L) = null ==> ~ ?x in L);
139          p2 := (forall ?x . (binary-search ?x R) = null ==> ~ ?x in R);
140          ind-hyp1 := (BST L ==> p1);
141          ind-hyp2 := (BST R ==> p2)}
142     assume i := (BST (node L y R))
143       conclude (forall ?x . (binary-search ?x (node L y R)) = null ==>
144                             ~ ?x in (node L y R))
145         let {
146          smaller-in-left := (forall ?x . ?x in L ==> ?x <= y);
147          larger-in-right := (forall ?z . ?z in R ==> y <= ?z);
148          p0 := (BST L & smaller-in-left & BST R &
149                 larger-in-right);
150          _ := (!chain-> [i ==> p0        [BST.nonempty]]);
151          _ := (!chain-> [p0 ==> (BST L)          [prop-taut]
152                             ==> p1               [ind-hyp1]]);
153          _ := (!chain-> [p0 ==> smaller-in-left [prop-taut]]);
154          _ := (!chain-> [p0 ==> (BST R)          [prop-taut]
155                             ==> p2               [ind-hyp2]]);
156          _ := (!chain-> [p0 ==> larger-in-right [prop-taut]])}
157         pick-any x
158           assume p3 := ((binary-search x (node L y R)) = null)
159             (!by-contradiction (~ x in (node L y R))
160               assume (x in (node L y R))
161                 let {p4 := (!chain-> [(x in (node L y R))
162                                       ==> (x = y | x in L | x in R)
163                                             [in.nonempty]])}
164                 (!two-cases
165                  assume (x = y)
166                    let {A := (!chain
167                               [null:(BinTree N)
168                                = (binary-search x (node L y R))   [p3]
169                                = (node L y R) [at-root]]);
170                         B := (!chain->
171                               [true ==> (null =/= (node L y R))
172                                  [(exclusive-constructors "BinTree")]])}
173                    (!absurd  A B)
174                  assume (x =/= y)
175                    (!two-cases
176                     assume (x < y)
177                       (!cases p4
178                        assume (x = y)
179                          (!absurd (x = y) (x =/= y))
180                        assume A := (x in L)
181                          let {B := (!chain->
182                                     [(binary-search x L)
183                                     = (binary-search x (node L y R))
184                                         [go-left]
185                                     = null:(BinTree N)        [p3]
186                                     ==> (~ x in L)            [p1]])}
187                          (!absurd A B)
188                        assume (x in R)
189                          let {_ := (!chain->
190                                     [(x in R)
191                                      ==> (y <= x)    [larger-in-right]
192                                      ==> (~ x < y)   [N.Less=.trichotomy4]])}
193                          (!absurd (x < y) (~ x < y)))
194                     assume (~ x < y)
195                       (!cases p4
196                        assume (x = y)
197                          (!absurd (x = y) (x =/= y))
198                        assume (x in L)
199                          let {_ :=
200                            (!chain->
201                             [(x in L)
202                              ==> (x <= y)        [smaller-in-left]
203                              ==> (x < y | x = y) [N.Less=.definition]
204                              ==> (~ x < y & (x < y | x = y))
205                                                  [augment]
206                              ==> (x = y)         [prop-taut]])}
207                          (!absurd (x = y) (x =/= y))
```

```
208                       assume (x in R)
209                         let {_ :=
210                           (!chain->
211                            [(binary-search x R)
212                             = (binary-search x (node L y R))
213                                       [go-right]
214                             = null:(BinTree N)          [p3]
215                             ==> (~ x in R)              [p2]])}
216                          (!absurd (x in R) (~ x in R))))))
217 } # by-induction
218
219 #.....................................................................
220 # Alternative proof of not-found with top level case analysis based
221 # on cases (x = y | x in L | x in R):
222 by-induction not-found {
223   null => assume (BST null)
224             conclude (forall ?x .
225                         (binary-search ?x null) = null ==> ~ ?x in null)
226               pick-any x:N
227                 assume ((binary-search x null) = null)
228                   (!chain-> [true ==> (~ x in null) [BinTree.in.empty]])
229 | (node L y:N R) =>
230     let {p1 := (forall ?x . (binary-search ?x L) = null ==> ~ ?x in L);
231          p2 := (forall ?x . (binary-search ?x R) = null ==> ~ ?x in R);
232          ind-hyp1 := (BST L ==> p1);
233          ind-hyp2 := (BST R ==> p2)}
234     assume i := (BST (node L y R))
235       conclude (forall ?x . (binary-search ?x (node L y R)) = null ==>
236                             ~ ?x in (node L y R))
237         let {smaller-in-left := (forall ?x . ?x in L ==> ?x <= y);
238              larger-in-right := (forall ?z . ?z in R ==> y <= ?z);
239              p0 := (BST L & smaller-in-left & BST R &
240                     larger-in-right);
241              _ := (!chain-> [i ==> p0 [BinTree.BST.nonempty]]);
242              _ := (!chain-> [p0 ==> (BST L) [prop-taut]]);
243              _ := (!chain-> [p0 ==> smaller-in-left [prop-taut]]);
244              _ := (!chain-> [p0 ==> (BST R) [prop-taut]]);
245              _ := (!chain-> [p0 ==> larger-in-right [prop-taut]])}
246         pick-any x
247           let {_ := (!chain-> [(BST L) ==> p1    [ind-hyp1]]);
248                _ := (!chain-> [(BST R) ==> p2    [ind-hyp2]])}
249           assume p3 := ((binary-search x (node L y R)) = null)
250             let {_ :=
251                  (!by-contradiction (x =/= y)
252                   assume (x = y)
253                     (!absurd
254                      (!chain
255                       [null:(BinTree N)
256                             = (binary-search x (node L y R))   [p3]
257                             = (node L y R) [at-root]])
258                      (!chain-> [true ==> (~ (null = (node L y R)))
259                             [(first (datatype-axioms "BinTree"))]]))))}
260              (!by-contradiction (~ x in (node L y R))
261               assume (x in (node L y R))
262                 let {to-consider :=
263                      (!chain-> [(x in (node L y R))
264                                 ==> (x = y | x in L | x in R)
265                                           [BinTree.in.nonempty]])}
266                   (!cases to-consider
267                    assume (x = y)
268                      (!absurd (x = y) (x =/= y))
269                    assume (x in L)
270                      let {p4 := ((binary-search x (node L y R))
271                                  = (binary-search x L));
272                           _ := (!chain->
273                                 [(x in L)
274                                  ==> (x <= y)        [smaller-in-left]
275                                  ==> (x < y | x = y) [Less=.definition]]);
276                           _ := conclude p4
277                                  (!cases (x < y | x = y)
```

```
                                (!chain
                                 [(x < y) ==> p4 [go-left]])
                                assume (x = y)
                                  (!from-complements p4 (x = y) (x =/= y)));
                      _ := conclude (~ x in L)
                             (!chain->
                              [(binary-search x L)
                               = (binary-search x (node L y R)) [p4]
                               = null:(BinTree N)               [p3]
                               ==> (~ x in L)                   [p1]])}
                   (!absurd (x in L) (~ (x in L)))
                 assume (x in R)
                   let {p5 := ((binary-search x (node L y R))
                               = (binary-search x R));
                        _ := conclude p5
                               (!chain->
                                [(x in R)
                                 ==> (y <= x)    [larger-in-right]
                                 ==> (~ x < y)   [Less=.trichotomy4]
                                 ==> p5          [go-right]]);
                        _ := conclude (~ x in R)
                               (!chain->
                                [(binary-search x R)
                                 = (binary-search x (node L y R)) [p5]
                                 = null:(BinTree N)               [p3]
                                 ==> (~ (x in R))                 [p2]])}
                   (!absurd (x in R) (~ x in R))))
}

#.....................................................................
# Converse of binary-search.not-found follows from
# binary-search.found:
define not-in-implies-null-result :=
  (forall T .
    BST T ==> forall x . ~ x in T ==> (binary-search x T) = null)

conclude not-in-implies-null-result
  pick-any T:(BinTree N)
    assume (BST T)
      let {exhaustive := (!constructor-exhaustiveness "BinTree")}
      pick-any x:N
        assume (~ x in T)
          (!by-contradiction ((binary-search x T) = null)
           assume ii := ((binary-search x T) =/= null)
             let {p := (exists ?L ?y ?R .
                          (binary-search x T) = (node ?L ?y ?R));
                  _ := (!chain->
                        [true
                         ==> ((binary-search x T) = null | p) [exhaustive]
                         ==> p                                [(dsyl with ii)]])}
             pick-witnesses L y R for p p'
               let {_ := (!chain-> [p' ==> (x = y & x in T) [found]
                                   ==> (x in T)    [right-and]])}
                (!absurd (x in T) (~ x in T)))

#.....................................................................
# Combining the implications:
define not-found-iff-not-in :=
  (forall T .
    BST T ==> forall x . (binary-search x T) = null <==> ~ x in T)

conclude not-found-iff-not-in
  pick-any T:(BinTree N)
    assume (BST T)
      pick-any x:N
        let {A := (!chain
                   [((binary-search x T) = null) ==> (~ x in T)
                    [not-found]]);
             B := (!chain
                   [(~ x in T) ==> ((binary-search x T) = null)
```

```
348                     [not-in-implies-null-result]])}
349         (!equiv A B)
350 #.....................................................................
351 define in-implies-node-result :=
352   (forall T .
353     BST T ==>
354       forall x .
355         x in T ==> exists L R . (binary-search x T) = (node L x R))
356
357 conclude in-implies-node-result
358   pick-any T:(BinTree N)
359     assume (BST T)
360       pick-any x:N
361         assume (x in T)
362           let {p := (exists ?L ?y ?R .
363                       (binary-search x T) = (node ?L ?y ?R));
364                q := ((binary-search x T) =/= null);
365                _ := (!by-contradiction q
366                      assume i := ((binary-search x T) = null)
367                        let {_ := (!chain->
368                                   [i ==> (~ x in T)  [not-found]])}
369                         (!absurd (x in T) (~ x in T)));
370                exhaustive := (!constructor-exhaustiveness "BinTree");
371                _ := (!chain->
372                      [true
373                   ==> ((binary-search x T) = null | p) [exhaustive]
374                   ==> p                              [(dsyl with q)]])}
375           pick-witnesses L y R for p p'
376             let {_ := (!chain->
377                        [(binary-search x T)
378                         = (node L y R) [p']
379                         ==> (x = y & x in T)  [found]
380                         ==> (x = y)           [left-and]])}
381               (!chain->
382                [(binary-search x T)
383                 = (node L y R) [p']
384                 = (node L x R) [(x = y)]
385                 ==> (exists ?L ?R .
386                        (binary-search x T) = (node ?L x ?R))
387                                [existence]])
388 } # binary-search
389 } # BinTree
```