lib/search/binary-search1-nat.ath

```
# Binary search function for searching in a binary search tree (here
# restricted to natural number elements) and correctness theorems.
# Optimized version that uses about 25% fewer comparisions, on average,
# versus the version in binary-search.ath.

load "binary-search-tree-nat.ath"

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

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

module binary-search {

define [x y L R L1 y1 R1] := [?x:N ?y:N ?L:(BinTree N) ?R:(BinTree N)
                              ?L1:(BinTree N) ?y1:N ?R1:(BinTree N)]

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

define (binary-search-found-base) :=
 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 := (!chain [null:(BinTree N)
                             = (binary-search x null)     [empty]
                             = (node L y R)               [i]]);
               B := (!chain-> [true
                               ==> (null =/= (node L y R)) [tree-axioms]])}
           (!from-complements (x = y & x in null) A B)

(!binary-search-found-base)

define (found-property T) :=
  (forall x L1 y1 R1 .
     (binary-search x T) = (node L1 y1 R1) ==> x = y1 & x in T)

define binary-search-found-step :=
 method (T)
  match T {
    (node L:(BinTree N) y:N R:(BinTree N)) =>
      let {[ind-hyp1 ind-hyp2] := [(BST L ==> found-property L)
                                   (BST R ==> found-property R)]}
       assume hyp := (BST T)
```

```
68          conclude (found-property T)
69           let {p0 := (BST L &
70                       (forall x . x in L ==> x <= y) &
71                       BST R &
72                       (forall z . z in R ==> y <= z));
73                _ := (!chain-> [hyp ==> p0                    [BST.nonempty]]);
74                fpl := (!chain-> [p0 ==> (BST L)              [prop-taut]
75                                     ==> (found-property L)   [ind-hyp1]]);
76                fpr := (!chain-> [p0 ==> (BST R)              [prop-taut]
77                                     ==> (found-property R)   [ind-hyp2]])}
78             pick-any x:N L1 y1:N R1
79               let {subtree := (node L1 y1 R1)}
80               assume hyp' := ((binary-search x T) = subtree)
81                 conclude (x = y1 & x in T)
82                   (!two-cases
83                    assume (x < y)
84                      (!chain->
85                       [(binary-search x L)
86                        = (binary-search x T) [go-left]
87                        = subtree             [hyp']
88                       ==> (x = y1 & x in L)  [fpl]
89                       ==> (x = y1 & x in T)  [in.left]])
90                    assume (~ x < y)
91                      (!two-cases
92                        assume (y < x)
93                          (!chain->
94                           [(binary-search x R)
95                            = (binary-search x T)  [go-right]
96                            = subtree              [hyp']
97                            ==> (x = y1 & x in R)  [fpr]
98                            ==> (x = y1 & x in T)  [in.right]])
99                        assume (~ y < x)
100                         let {_ := (!chain->
101                                    [(~ x < y & ~ y < x)
102                                  ==> (x = y) [Less.trichotomy1]]);
103                              i := conclude (x = y1)
104                                     (!chain->
105                                      [T
106                                      = (binary-search x T)
107                                                          [at-root]
108                                      = subtree           [hyp']
109                                    ==> (y = y1)          [tree-axioms]
110                                    ==> (x = y1)          [(x = y)]]);
111                              ii := conclude (x in T)
112                                      (!chain->
113                                       [(x = y)
114                                      ==> (x in T)        [in.root]])}
115                       (!both i ii)))
116   }
117
118 by-induction found {
119   null => (!binary-search-found-base)
120 | (node L y:N R) => (!binary-search-found-step (node L y R))
121 }
122
123 #.....................................................................
124
125 define (not-found-prop T) :=
126   (forall x . (binary-search x T) = null ==> ~ x in T)
127
128 by-induction not-found {
129   null =>
130     assume (BST null)
131       conclude (not-found-prop null)
132         pick-any x:N
133           assume ((binary-search x null) = null)
134             (!chain-> [true ==> (~ x in null)    [in.empty]])
135 | (T as (node L y:N R)) =>
136     let {p1 := (not-found-prop L);
137          p2 := (not-found-prop R);
```

```
138           [ind-hyp1 ind-hyp2] := [(BST L ==> p1) (BST R ==> p2)]}
139       assume hyp := (BST T)
140         conclude (not-found-prop T)
141           let {smaller-in-left := (forall x . x in L ==> x <= y);
142                larger-in-right := (forall z . z in R ==> y <= z);
143                p0 := (BST L &
144                       smaller-in-left &
145                       BST R &
146                       larger-in-right);
147                _ := (!chain-> [hyp ==> p0                [BST.nonempty]]);
148                _ := (!chain-> [p0 ==> smaller-in-left    [prop-taut]]);
149                _ := (!chain-> [p0 ==> larger-in-right    [prop-taut]]);
150                _ := (!chain-> [p0
151                           ==> (BST L)                    [prop-taut]
152                           ==> (not-found-prop L)         [ind-hyp1]]);
153                _ := (!chain-> [p0
154                            ==> (BST R)                   [prop-taut]
155                            ==> (not-found-prop R)        [ind-hyp2]])}
156           pick-any x
157             assume hyp' := ((binary-search x T) = null)
158               (!by-contradiction (~ x in T)
159                 assume (x in T)
160                   let {C := (!chain->
161                              [(x in T)
162                               ==> (x = y | x in L | x in R)
163                                                  [in.nonempty]])}
164                     (!two-cases
165                      assume (x < y)
166                        let {_ := (!chain->
167                                   [(binary-search x L)
168                                    = (binary-search x T)   [go-left]
169                                    = null:(BinTree N)      [hyp']
170                                    ==> (~ x in L)          [p1]])}
171                          (!cases C
172                           assume (x = y)
173                             (!absurd
174                              (x = y)
175                              (!chain->
176                               [(x < y) ==> (x =/= y) [Less.not-equal]]))
177                           assume (x in L)
178                             (!absurd (x in L) (~ x in L))
179                           assume (x in R)
180                             (!absurd
181                              (x < y)
182                              (!chain->
183                               [(x in R)
184                                ==> (y <= x)  [larger-in-right]
185                                ==> (~ x < y) [Less=.trichotomy4]]))))
186                      assume (~ x < y)
187                        (!two-cases
188                         assume (y < x)
189                           let {_ := (!chain->
190                                      [(binary-search x R)
191                                       = (binary-search x T)[go-right]
192                                       = null:(BinTree N)   [hyp']
193                                       ==> (~ x in R)       [p2]])}
194                             (!cases C
195                              assume (x = y)
196                                (!absurd
197                                 (!chain [y = x [(x = y)]])
198                                 (!chain-> [(y < x) ==> (y =/= x)
199                                                   [Less.not-equal]]))
200                              assume (x in L)
201                                (!absurd
202                                 (y < x)
203                                 (!chain->
204                                  [(x in L)
205                                   ==> (x <= y)  [smaller-in-left]
206                                   ==> (~ y < x) [Less=.trichotomy4]]))
207                              assume (x in R)
```

```
208                        (!absurd (x in R) (~ x in R)))
209                    assume (~ y < x)
210                      (!absurd
211                       (!chain->
212                        [null:(BinTree N)
213                         = (binary-search x T) [hyp']
214                         = T                   [at-root]])
215                       (!chain->
216                        [true ==> (null =/= T) [tree-axioms]])))))
217  }
218
219  #------------------------------------------------------------------------
220  # Corollary:
221
222  define in-iff-result-not-null :=
223    (forall T .
224      (BST T) ==>
225        forall x . x in T <==> (binary-search x T) =/= null)
226
227  conclude in-iff-result-not-null
228    pick-any T:(BinTree N)
229      assume (BST T)
230        pick-any x:N
231          let {right :=
232                assume (x in T)
233                  (!by-contradiction ((binary-search x T) =/= null)
234                   assume i := ((binary-search x T) = null)
235                     (!absurd (x in T)
236                              (!chain->
237                               [i ==> (~ x in T) [not-found]])));
238               left :=
239                 assume ii := ((binary-search x T) =/= null)
240                   let {p := (exists ?L ?y ?R .
241                                (binary-search x T) = (node ?L ?y ?R));
242                        ex := (!constructor-exhaustiveness "BinTree");
243                        _ := (!chain->
244                              [true
245                               ==> ((binary-search x T) = null | p) [ex]
246                               ==> p                  [(dsyl with ii)]])}
247                   pick-witnesses L y R for p p'
248                     (!chain-> [p' ==> (x = y & x in T) [found]
249                                   ==> (x in T)         [right-and]])}
250          (!equiv right left)
251
252  } # binary-search
253  } # BinTree
```