

lib/search/binary-tree.ath

```

1  # Binary tree datatype
2
3  load "list-of"
4
5  #-----
6
7  datatype (BinTree S) := null | (node (BinTree S) S (BinTree S))
8  assert (datatype-axioms "BinTree")
9
10 module BinTree {
11   open List
12
13   define [x x' y T L R] :=
14     [?x:'S ?x':'S ?y:'S
15      ?T:(BinTree 'S) ?L:(BinTree 'S) ?R:(BinTree 'S)]
16
17   declare in: (S) [S (BinTree S)] -> Boolean
18
19   module in {
20
21     assert empty := (forall x . ~ x in null)
22     assert nonempty :=
23       (forall x L y R . x in (node L y R) <==> x = y | x in L | x in R)
24
25     #.....
26     # Lemmas:
27
28     define root := (forall x L y R . x = y ==> x in (node L y R))
29     define left := (forall x L y R . x in L ==> x in (node L y R))
30     define right := (forall x L y R . x in R ==> x in (node L y R))
31
32     #.....
33     # Proofs:
34
35     conclude root
36     pick-any x L y R
37     (!chain
38      [(x = y) ==> (x = y | x in L | x in R)      [alternate]
39       ==> (x in (node L y R))                  [nonempty]])
40
41     conclude left
42     pick-any x L y R
43     (!chain
44      [(x in L) ==> (x in L | x in R)            [alternate]
45       ==> (x = y | x in L | x in R)            [alternate]
46       ==> (x in (node L y R))                  [nonempty]])
47
48     conclude right
49     pick-any x L y R
50     assume (x in R)
51     (!chain->
52      [(x in R) ==> (x in L | x in R)            [alternate]
53       ==> (x = y | x in L | x in R)            [alternate]
54       ==> (x in (node L y R))                  [nonempty]])
55   } # in
56
57   #-----
58   # inorder: applied to a binary-tree, produces a list of the tree elements
59   # ordered so that the root element appears between the elements
60   # of the left subtree and those of the right subtree (and recursively
61   # the elements are in this order within each subtree).
62
63
64   declare inorder: (S) [(BinTree S)] -> (List S)
65
66   define join := List.join
67

```

```

68 module inorder {
69   assert empty := (inorder null = nil)
70   assert nonempty :=
71     (forall L R x .
72       inorder (node L x R) = (inorder L) join (x :: inorder R))
73 }
74
75 overload BinTree.in List.in
76
77 extend-module inorder {
78   define in-correctness-1 := (forall T x . x in inorder T ==> x in T)
79   define in-correctness-2 := (forall T x . x in T ==> x in inorder T)
80
81   by-induction in-correctness-1 {
82     null =>
83       pick-any x
84         assume (x in inorder null)
85           let {A := (!chain->
86             [(x in inorder null)
87              ==> (x in nil) [empty]]);
88             B := (!chain-> [true ==> (~ x in nil) [List.in.empty]])}
89             (!from-complements (x in null) A B)
90   | (node L y R) =>
91     let {ind-hyp1 := (forall ?x . ?x in inorder L ==> ?x in L);
92         ind-hyp2 := (forall ?x . ?x in inorder R ==> ?x in R)}
93     pick-any x
94       assume A := (x in (inorder (node L y R)))
95       let {B := (!chain->
96         [A ==> (x in ((inorder L) join (y :: inorder R)))
97          [nonempty]
98          ==> (x in inorder L |
99             (x in (y :: inorder R))) [List.in.of-join]
100         ==> (x in inorder L | x = y | x in inorder R)
101            [List.in.nonempty]])}
102         (!cases B
103          (!chain [(x in inorder L)
104                  ==> (x in L) [ind-hyp1]
105                  ==> (x in (node L y R)) [in.left]])
106          (!chain [(x = y)
107                  ==> (x in (node L y R)) [in.root]])
108          (!chain [(x in inorder R)
109                  ==> (x in R) [ind-hyp2]
110                  ==> (x in (node L y R)) [in.right]])
111        )
112
113   by-induction in-correctness-2 {
114     null =>
115       pick-any x
116         assume (x in null)
117         (!from-complements (x in inorder null)
118          (x in null)
119          (!chain-> [true ==> (~ x in null) [in.empty]]))
120   | (node L y R) =>
121     let {ind-hyp1 := (forall ?x . ?x in L ==> ?x in inorder L);
122         ind-hyp2 := (forall ?x . ?x in R ==> ?x in inorder R)}
123     pick-any x
124       assume A := (x in (node L y R))
125       conclude (x in (inorder (node L y R)))
126       let {C := (!chain-> [A ==> (x = y | x in L | x in R)
127                          [in.nonempty]])}
128         (!cases C
129          assume (x = y)
130            (!chain->
131             [(x = y)
132              ==> (x in (x :: inorder R)) [List.in.head]
133              ==> (x in (y :: inorder R)) [(x = y)]
134              ==> (x in inorder L | x in (y :: inorder R))
135                 [alternate]
136              ==> (x in ((inorder L) join (y :: inorder R)))
137                 [List.in.of-join]

```

```

138     ==> (x in (inorder (node L y R))) [nonempty]])
139   (!chain [(x in L)
140     ==> (x in inorder L)           [ind-hyp1]
141     ==> (x in inorder L | x in (y :: inorder R))
142         [alternate]
143     ==> (x in ((inorder L) join (y :: inorder R)))
144         [List.in.of-join]
145     ==> (x in (inorder (node L y R)))
146         [nonempty]])
147   (!chain [(x in R)
148     ==> (x in inorder R)           [ind-hyp2]
149     ==> (x in (y :: inorder R)) [List.in.tail]
150     ==> (x in inorder L | x in (y :: inorder R))
151         [alternate]
152     ==> (x in ((inorder L) join (y :: inorder R)))
153         [List.in.of-join]
154     ==> (x in (inorder (node L y R))) [nonempty]])
155 }
156
157 define in-correctness := (forall T x . x in (inorder T) <=> x in T)
158
159 conclude in-correctness
160   pick-any T:(BinTree 'S) x
161     (!equiv
162       (!chain [(x in inorder T) ==> (x in T)   [in-correctness-1]])
163       (!chain [(x in T) ==> (x in inorder T) [in-correctness-2]]))
164
165 } # inorder
166
167 #-----
168 # count: given a value x and a binary tree, returns the number
169 # of occurrences of x in the tree.
170
171 declare count: (S) [S (BinTree S)] -> N
172 overload BinTree.count List.count
173
174 define + := N.+
175
176 module count {
177   define (axioms as [empty more same]) :=
178     (fun
179       [(count x null) = zero
180        (count x (node L x' R)) =
181          [(S ((count x L) + (count x R))) when (x = x')
182           ((count x L) + (count x R))   when (x /= x')])
183   assert axioms
184 } # count
185
186 extend-module inorder {
187
188   define count-correctness :=
189     (forall T x . (count x (inorder T)) = (count x T))
190
191 #-----
192 # Proof:
193
194 by-induction count-correctness {
195   null =>
196     conclude (forall ?x . (count ?x inorder null) =
197       (BinTree.count ?x null))
198     pick-any x
199       let {A := (!chain [(count x inorder null)
200         = (count x nil) [empty]
201         = zero [List.count.empty]]);
202         B := (!chain [(count x null)
203         = zero [count.empty]})
204         (!combine-equations A B)
205   | (node L y R) =>
206     let {ind-hyp1 := (forall ?x . (count ?x inorder L) = (count ?x L));
207         ind-hyp2 := (forall ?x . (count ?x inorder R) = (count ?x R))}

```

```

208 conclude (forall ?x .(count ?x (inorder (node L y R))) =
209             (count ?x (node L y R)))
210 pick-any x
211   (!two-cases
212     assume (x = y)
213     (!combine-equations
214       (!chain
215         [(count x (inorder (node L y R)))
216          = (count x ((inorder L) join (y :: inorder R)))
217            [nonempty]
218          = ((count x inorder L) +
219            (count x (y :: inorder R)))
220            [List.count.of-join]
221          = ((count x inorder L) + (S (count x inorder R)))
222            [List.count.more]
223          = (S ((count x inorder L) + (count x inorder R)))
224            [N.Plus.right-nonzero]])
225       (!chain
226         [(count x (node L y R))
227          = (S ((count x L) + (count x R)))
228            [count.more]
229          = (S ((count x inorder L) + (count x inorder R)))
230            [ind-hyp1 ind-hyp2]]))
231     assume (x /= y)
232     (!combine-equations
233       (!chain
234         [(count x (inorder (node L y R)))
235          = (count x ((inorder L) join (y :: inorder R)))
236            [nonempty]
237          = ((count x inorder L) +
238            (count x (y :: inorder R)))
239            [List.count.of-join]
240          = ((count x inorder L) + (count x inorder R))
241            [List.count.same]])
242       (!chain
243         [(count x (node L y R))
244          = ((count x L) + (count x R))
245            [count.same]
246          = ((count x inorder L) + (count x inorder R))
247            [ind-hyp1 ind-hyp2]])))
248 } # by-induction
249
250 } # inorder
251 } # BinTree
252
253 EOF
254 (load "c:\\np\\lib\\search\\binary-tree")

```