# 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 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)

let {A := (!chain [null:(BinTree N) = (binary-search x null) [empty]]);
    B := (!chain-> [true ==>
      (null /= (node L y R)) [tree-axioms]]))}

(!from-complements (x = y & x in null) A B)

(binary-search-found-base)

define [x1 y1 L1 R1] := [?x1:N ?y1:N ?L1:(BinTree N) ?R1:(BinTree N)]

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)
        conclude (found-property T)
        let {p0 := (BST L &
          (forall x . x in L ==> x <= y) &
          BST R &
          (forall z . z in R ==> y <= z));
lib/search/binary-search-nat.ath

_ := (!chain-> [hyp => p0 [BST.nonempty]]);
fpl := (!chain-> [p0 => (BST L) [prop-taut]]);
  ==> (found-property L) [ind-hyp1]);
fpr := (!chain-> [p0 => (BST R) [prop-taut]]);
  ==> (found-property R) [ind-hyp2]));

pick-any x:N L1 y1:N R1
let {subtree := (node L1 y1 R1)}
assume hyp' := ((binary-search x T) = subtree)
conclude (x = y1 & x in T)
  (!two-cases
   (!both conclude (x = y1)
    (!chain->
     [T = (binary-search x T) [at-root]
      = subtree [hyp']
      ==> y = y1 [tree-axioms]
      ==> (x = y1) [(x = y)]))
   conclude (x in T)
    (!chain-> [(x = y)
      ==> (x in T) [in.root]]))
  assume (x /= y)
  (!two-cases
   assume (x < y)
   (!chain- (binary-search x L)
     = (binary-search x T) [go-left]
     = subtree [hyp']
     ==> (x = y1 & x in L) [fpl]
     ==> (x = y1 & x in T) [in.left]))
   assume (~ x < y)
   (!chain- (binary-search x R)
     = (binary-search x T) [go-right]
     = subtree [hyp']
     ==> (x = y1 & x in R) [fpr]
     ==> (x = y1 & x in T) [in.right]))
  )

by-induction found 
  null => (!binary-search-found-base)
  | (node L y:N R) => (!binary-search-found-step (node L y R))
|

by-induction not-found 
  null => (forall x . (binary-search x T) = null ==~ x in T)

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

by-induction not-found 
  null => (forall x . (binary-search x null) = null)
pick-any x
assume hyp' := \((binary-search x T) = \text{null}\)

{!by-contradiction (~ x in T)
assume (x in T)
}

{!two-cases
assume (x = y)
{!absurd
{!chain [null:(BinTree N)
= (binary-search x T) [hyp']
= T [at-root]]
{!chain-> [true
=> (null =/= T) [tree-axioms]]])
assume (x /= y)
{!two-cases
assume (x < y)
{!cases disj
assume (x = y)
{!absurd (x = y) (x /= y))
assume (x in L)
{!absurd
(x in L)
{!chain-> [(binary-search x L)
= (binary-search x T) [go-left]
= null [hyp']
=> (~ x in L) [pl]])
assume (x in R)
{!absurd
(x < y)
{!chain-> [(x in R)
=> (y <= x) [larger-in-right]
=> (~ x < y) [N.Less=.trichotomy4]]})
assume (~ x < y)
{!cases disj
assume (x = y)
{!absurd (x = y) (x /= y))
assume (x in L)
{!absurd
(x /= y)
{!chain-> [(x in L)
=> (x <= y) [smaller-in-left]
=> (x < y | x = y) [N.Less=.definition]
=> (~ x < y &
(x < y | x = y)) [augment]
=> (x = y) [prop-taut]]})
assume (x in R)
{!absurd
(x in R)
{!chain-> [(binary-search x R)
= (binary-search x T) [go-right]
= null [hyp']
=> (~ x in R) [p2]])})

} # by-induction

#..........................................................................
# 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)
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) [tree-axioms]
===> 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)
[not-in-implies-null-result]])
!equiv A B)

define in-implies-node-result :=
(forall T .
BST T ==>
forall x .
x in T ===> exists L R . (binary-search x T) = (node L x R))
conclude in-implies-node-result
pick-any T:(BinTree N)
assume (BST T)
pick-any x:N
assume (x in T)
let [p := (exists L y R .
(binary-search x T) = (node L y R));
q := ((binary-search x T) =/= null);
_ := (by-contradiction q)
assume i := ((binary-search x T) = null)
let (_, := (!chain->
[i ===> (~ x in T) [not-found]])
(!absurd (x in T) (~ x in T)));
_ := (!chain->
true
===> ((binary-search x T) = null | p) [tree-axioms]
===> p [dsyl with ql]])
pick-witnesses L y R for p p'
let (_, := (!chain->
[(binary-search x T)
= (node L y R) [p']
===> (x = y) [found left-and]])
(!chain->
[(binary-search x T)
= (node L y R) [p']
===> (x = y)]
===> (exists L R .
(binary-search x T) = (node L x R)) [existence]])
778  |  # binary-search
779  |  # BinTree