# Binary tree datatype

`load "list-of"

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

datatype (BinTree S) := null | (node (BinTree S) S (BinTree S))
assert (datatype-axioms "BinTree")

module BinTree {
open List

define [x x' y T L R] :=
  [x:'S ?x':S ?y:'S
   ?T:(BinTree 'S) ?L:(BinTree 'S) ?R:(BinTree 'S)]
declare in: (S) [S (BinTree S)] -> Boolean

module in {
assert empty := (forall x . x in null)
assert nonempty :=
  (forall x L y R . x in (node L y R) ==> x = y | x in L | x in R)
# ..........................................................................
# Lemmas:
define root := (forall x L y R . x = y ==> x in (node L y R))
define left := (forall x L y R . x in L ==> x in (node L y R))
define right := (forall x L y R . x in R ==> x in (node L y R))
# ..........................................................................
# Proofs:
conclude root
pick-any x L y R
  (!chain
   [(x = y) ==> (x = y | x in L | x in R) [alternate]
   ==> (x in (node L y R)) [nonempty]])

conclude left
pick-any x L y R
  (!chain
   [(x in L) ==> (x in L | x in R) [alternate]
   ==> (x = y | x in L | x in R) [alternate]
   ==> (x in (node L y R)) [nonempty]])

conclude right
pick-any x L y R
  assume (x in R)
  (!chain->
   [(x in R) ==> (x in L | x in R) [alternate]
   ==> (x = y | x in L | x in R) [alternate]
   ==> (x in (node L y R)) [nonempty]])
}
# in

# inorder: applied to a binary-tree, produces a list of the tree elements
# ordered so that the root element appears between the elements
# of the left subtree and those of the right subtree (and recursively
# the elements are in this order within each subtree).
declare inorder: (S) [(BinTree S)] -> (List S)
define join := List.join
module inorder {

assert empty := (inorder null = nil)
assert nonempty :=
  (forall L R x . inorder (node L x R) = (inorder L) join (x :: inorder R))
}

overload BinTree.in List.in

extend-module inorder {

define in-correctness-1 := (forall T x . x in inorder T ==> x in T)
define in-correctness-2 := (forall T x . x in T ==> x in inorder T)

by-induction in-correctness-1 {
  null =>
  pick-any x
    assume (x in inorder null)
    let (A := (![chain->
               [(x in inorder null)
                ==> (x in nil) [empty]]));
        B := (![chain-> [true ==> (~ x in nil) [List.in.empty]]])
    ){from-complements (x in null) A B}
  | (node L y R) =>
    let {ind-hyp1 := (forall ?x . ?x in inorder L ==> ?x in L);
        ind-hyp2 := (forall ?x . ?x in inorder R ==> ?x in R)}
    pick-any x
    assume A := (x in (inorder (node L y R)))
    let (B := (![chain->
               [A ==> (x in ((inorder L) join (y :: inorder R)))
                [nonempty]
               ==> (x in inorder L | (x in (y :: inorder R))) [List.in.of-join]
               ==> (x in inorder L | x = y | x in inorder R)
                [List.in.nonempty]]))
    ){cases B
      (!chain [(x in inorder L)
        ==> (x in L) [ind-hyp1]
        ==> (x in (node L y R)) [in.left]])
      (!chain [(x = y)
        ==> (x in (node L y R)) [in.root]])
      (!chain [(x in inorder R)
        ==> (x in R) [ind-hyp2]
        ==> (x in (node L y R)) [in.right]])}
  }

by-induction in-correctness-2 {
  null =>
  pick-any x
  (!from-complements (x in inorder null)
   (x in null)
   (!chain-> [true ==> (~ x in null) [in.empty]])
  ){node L y R} =>
  let {ind-hyp1 := (forall ?x . ?x in inorder L ==> ?x in L);
        ind-hyp2 := (forall ?x . ?x in inorder R ==> ?x in R)}
  pick-any x
  assume A := (x in (node L y R))
  conclude (x in (inorder (node L y R)))
  let (C := (![chain-> [A ==> (x = y | x in L | x in R)
                        [in.nonempty]]}))
  ){cases C
    assume (x = y)
    (!chain->
     ![x = y]
     ==> (x in (inorder R)) [List.in.head]
     ==> (x in (y :: inorder R)) [x = y]
     ==> (x in inorder L | x in (y :: inorder R)) [alternate]
     ==> (x in ((inorder L) join (y :: inorder R))) [List.in.of-join]}
  }
}
define in-correctness := (forall T x . x in (inorder T) <=> x in T)

conclude in-correctness
  pick-any T:(BinTree 'S) x
  (!equiv
    (!chain [(x in inorder T) ==> (x in T) [in-correctness-1]])
    (!chain [(x in T) ==> (x in inorder T) [in-correctness-2]])
  )
# inorder

# count: given a value x and a binary tree, returns the number of occurrences of x in the tree.
declare count: (S) [S (BinTree S)] -> N
overload BinTree.count List.count

module count {
  define (axioms as [empty more same]) :=
    (fun
      [S (null) = zero]
      [S ((count x L) + (count x R)) when (x = x')]
      [S (count x L) + (count x R)) when (x /= x')]])
  assert axioms}
# count

extend-module inorder {
  define count-correctness :=
    (forall T x . (count x (inorder T)) = (count x T))
# Proof:
by-induction count-correctness {
  null =>
    conclude (forall ?x . (count ?x inorder null) =
      (BinTree.count ?x null))
    pick-any x
    let {A := (!chain [(count x inorder null) =
                      [count x nil] [empty] =
                      zero [List.count.empty]]);
         B := !chain [(count x null) =
                      zero [count.empty]])
      (!combine-equations A B)}
  | (node L y R) =>
    let {ind-hyp1 := (forall ?x . (count ?x inorder L) = (count ?x L));
       ind-hyp2 := (forall ?x . (count ?x inorder R) = (count ?x R))}
conclude (forall ?x . (count ?x (inorder (node L y R))) =
          (count ?x (node L y R)))

pick-any x

!two-cases

  assume (x = y)

  (!combine-equations

    (!chain
      [(count x (inorder (node L y R)))
       = (count x ((inorder L) join (y :: inorder R)))
       [nonempty]
       = ((count x inorder L) +
          (count x (y :: inorder R)))
       [list.count.of-join]
       = ((count x inorder L) + (S (count x inorder R)))
       [list.count.more]
       = (S ((count x inorder L) + (count x inorder R)))
       [N.Plus.right-nonzero])]

    (!chain
      [(count x (node L y R))
       = (S ((count x L) + (count x R)))
       [count.more]
       = (S ((count x inorder L) + (count x inorder R)))
       [ind-hyp1 ind-hyp2]])])

  assume (x != y)

  (!combine-equations

    (!chain
      [(count x (inorder (node L y R)))
       = (count x ((inorder L) join (y :: inorder R)))
       [nonempty]
       = ((count x inorder L) +
          (count x (y :: inorder R)))
       [list.count.of-join]
       = ((count x inorder L) + (count x inorder R))
       [list.count.same]])

    (!chain
      [(count x (node L y R))
       = ((count x L) + (count x R))
       [count.same]
       = ((count x inorder L) + (count x inorder R))
       [ind-hyp1 ind-hyp2]])])

) # by-induction

| # inorder
| # BinTree

EOF

(load "c:\\np\\lib\\search\\binary-tree")