Forkable-Strings

By Kawin

April 6, 2017

Contents

theory *Forkable-Strings* **imports** *Main ~~/src/HOL/List* **begin**

We will use True as 1 and False as 0 in characteristic strings; we might think about each bool value answers to the question 'is this slot controlled by an adversarial player?'.

datatype *nattree* = *Empty* | *Node nat nattree list*

One reason why we don't have Leaves here is that we have to define prefixes of tines carefully so that we don't consider having a leaf with cannot be continue by a list of trees but instead we can have a list of Emptys in order to extend each Empty by a tree

inductive *ListOfEmpty* :: *nattree list* ⇒ *bool* **where**
  Nil : *ListOfEmpty* []
| *Cons* : *ListOfEmpty l* ⟹ *ListOfEmpty* (*Empty#l*)

inductive *Leaf* :: *nattree* ⇒ *bool* **where**
ListOfEmpty l ⟹ *Leaf* (*Node n l*)

fun *lt-nat-tree* :: *nat* ⇒ *nattree* ⇒ *bool* **where**
  lt-nat-tree n Empty = *True*
| *lt-nat-tree n* (*Node m -*) = (*n* < *m*)

lemma *lt-nat-tree-lt* [*simp*]: (*n* < *m*) ⟷ *lt-nat-tree n* (*Node m l*)
  by *simp*

lemma *lt-nat-tree-ge* [*simp*]: (*n* ≥ *m*) ⟷ ¬ *lt-nat-tree n* (*Node m l*)
  by *auto*

fun *increasing-tree* :: *nattree* ⇒ *bool* **where**
  increasing-tree Empty = *True*
| *increasing-tree* (*Node -* []) = *True*
| *increasing-tree* (*Node n l*) = (∀ *x* ∈ *set l*. *increasing-tree x* ∧ *lt-nat-tree n x*)

lemma *increasing-tree-branch-list-of-empty* [*simp*]: *ListOfEmpty x* ⟶ *increasing-tree* (*Node n x*)

proof (*induction x*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a x*)
  then show *?case*
  proof (*cases a*)
    case *Empty*
    then show *?thesis*
    proof −
      obtain *nn* :: *nattree list* ⇒ *nattree* ⇒ *nat* ⇒ *nattree* **where**
       ∀ *x0 x1 x2*. (∃ *v3*. *v3* ∈ *set* (*x1* # *x0*) ∧ (¬ *increasing-tree v3* ∨ ¬ *lt-nat-tree x2 v3*)) = (*nn x0 x1 x2* ∈ *set* (*x1* # *x0*) ∧ (¬ *increasing-tree* (*nn x0 x1 x2*) ∨ ¬ *lt-nat-tree x2* (*nn x0 x1 x2*)))
        by *moura*
      then have *f1*: ∀ *n na ns*. (¬ *increasing-tree* (*Node n* (*na* # *ns*)) ∨ (∀ *nb*. *nb* ∉ *set* (*na* # *ns*) ∨ *increasing-tree nb* ∧ *lt-nat-tree n nb*)) ∧ (*increasing-tree* (*Node n* (*na* # *ns*)) ∨ *nn ns na n* ∈ *set* (*na* # *ns*) ∧ (¬ *increasing-tree* (*nn ns na n*) ∨ ¬ *lt-nat-tree n* (*nn ns na n*)))
        by (*meson increasing-tree.simps*(*3*))
      obtain *nns* :: *nattree list* ⇒ *nattree list* **where**
        f2: ∀ *ns*. (¬ *ListOfEmpty ns* ∨ *ns* = [] ∨ *ns* = *Empty* # *nns ns* ∧ *ListOfEmpty* (*nns ns*)) ∧ (*ListOfEmpty ns* ∨ *ns* ≠ [] ∧ (∀ *nsa*. *ns* ≠ *Empty* # *nsa* ∨ ¬ *ListOfEmpty nsa*))
        by (*metis ListOfEmpty.simps*)
      have (*Empty* # *x* = *Empty* # *nns* (*a* # *x*)) = (*x* = *nns* (*a* # *x*))
        by *blast*
      then show *?thesis*
      using *f2 f1* **by** (*metis* (*no-types*) *Cons.IH Empty in-set-member increasing-tree.simps*(*1*) *lt-nat-tree.simps*(*1*) *member-rec*(*1*) *member-rec*(*2*))
    qed
    next
    case (*Node x21 x22*)
    then show *?thesis*
      using *ListOfEmpty.simps* **by** *blast*
  qed

qed

lemma *increasing-tree-ind* [*simp*] : (∀ *x* ∈ *set l*. *increasing-tree x* ∧ *lt-nat-tree n x*) ⟷ *increasing-tree* (*Node n l*)
proof −
  { **fix** *nn* :: *nattree*
    obtain *nna* :: *nattree* ⇒ *nat* **and** *nnb* :: *nattree* ⇒ *nattree* **and** *nns* :: *nattree* ⇒ *nattree list* **and** *nnc* :: *nattree* ⇒ *nattree* **where**
      ff1: ∀ *n*. *increasing-tree n* ∨ *Node* (*nna n*) (*nnb n* # *nns n*) = *n* ∧ *nnc n* ∈ *set* (*nnb n* # *nns n*) ∧ (¬ *increasing-tree* (*nnc n*) ∨ ¬ *lt-nat-tree* (*nna n*) (*nnc n*))
      using *increasing-tree.elims*(*3*) **by** *moura*
    have ∀ *n ns*. (*n*::*nattree*) ∉ *set ns* ∨ (∃ *nsa*. *n* # *nsa* = *ns*) ∨ (∃ *na nsa*. *na* #

$nsa = ns \wedge n \in set\ nsa)$
by $(metis\ list.set\text{-}cases)$
then obtain $nnsa :: nattree \Rightarrow nattree\ list \Rightarrow nattree\ list$ **and** $nnd :: nattree \Rightarrow nattree\ list \Rightarrow nattree$ **and** $nnsb :: nattree \Rightarrow nattree\ list \Rightarrow nattree\ list$ **where**
$ff2: \forall n\ ns.\ n \notin set\ ns \vee n \# nnsa\ n\ ns = ns \vee nnd\ n\ ns \# nnsb\ n\ ns = ns \wedge n \in set\ (nnsb\ n\ ns)$
by $moura$
obtain $nne :: nat \Rightarrow nattree \Rightarrow nattree\ list \Rightarrow nattree$ **where**
$ff3: \forall n\ na\ ns.\ (\neg\ increasing\text{-}tree\ (Node\ n\ (na \# ns)) \vee (\forall nb.\ nb \notin set\ (na \# ns) \vee increasing\text{-}tree\ nb \wedge lt\text{-}nat\text{-}tree\ n\ nb)) \wedge (nne\ n\ na\ ns \in set\ (na \# ns) \wedge (\neg\ increasing\text{-}tree\ (nne\ n\ na\ ns) \vee \neg\ lt\text{-}nat\text{-}tree\ n\ (nne\ n\ na\ ns)) \vee increasing\text{-}tree\ (Node\ n\ (na \# ns)))$
by $moura$
then have $ff4: \forall n\ na\ nb\ ns.\ lt\text{-}nat\text{-}tree\ nb\ n \vee n \notin set\ (nnd\ na\ ns \# nnsb\ na\ ns) \vee \neg\ increasing\text{-}tree\ (Node\ nb\ ns) \vee na \# nnsa\ na\ ns = ns \vee na \notin set\ ns$
using $ff2$ **by** $metis$
{ assume $nn \# nnsa\ nn\ l \neq l$
{ assume $\exists na.\ nn \# nnsa\ nn\ l \neq l \wedge increasing\text{-}tree\ (Node\ na\ (nnd\ nn\ l \# nnsb\ nn\ l)) \wedge nn \# nnsa\ nn\ l \neq l \wedge increasing\text{-}tree\ (Node\ n\ l)$
moreover
{ assume $\exists na\ nb.\ increasing\text{-}tree\ (Node\ n\ l) \wedge nn \# nnsa\ nn\ l \neq l \wedge increasing\text{-}tree\ (Node\ na\ (nnd\ nb\ l \# nnsb\ nb\ l)) \wedge nb \# nnsa\ nb\ l \neq l \wedge nb \in set\ l \wedge increasing\text{-}tree\ (Node\ n\ l)$
moreover
{ assume $\exists na\ nb\ nc.\ nb \in set\ l \wedge increasing\text{-}tree\ (Node\ n\ l) \wedge nb \# nnsa\ nb\ l \neq l \wedge increasing\text{-}tree\ (Node\ n\ l) \wedge increasing\text{-}tree\ (Node\ nc\ (nnd\ na\ l \# nnsb\ na\ l)) \wedge na \# nnsa\ na\ l \neq l \wedge na \in set\ l \wedge increasing\text{-}tree\ (Node\ n\ l)$
moreover
{ assume $\exists na\ nb\ nc\ ns.\ nb \# nnsa\ nb\ ns \neq ns \wedge nb \in set\ ns \wedge increasing\text{-}tree\ (Node\ n\ l) \wedge nn \in set\ ns \wedge increasing\text{-}tree\ (Node\ n\ ns) \wedge increasing\text{-}tree\ (Node\ na\ (nnd\ nc\ l \# nnsb\ nc\ l)) \wedge nc \# nnsa\ nc\ l \neq l \wedge nc \in set\ l \wedge increasing\text{-}tree\ (Node\ n\ l)$
then have $(nn \notin set\ l \vee increasing\text{-}tree\ nn \wedge lt\text{-}nat\text{-}tree\ n\ nn) \wedge increasing\text{-}tree\ (Node\ n\ l) \vee \neg\ increasing\text{-}tree\ (Node\ n\ l) \wedge (\exists na.\ na \in set\ l \wedge (\neg\ increasing\text{-}tree\ na \vee \neg\ lt\text{-}nat\text{-}tree\ n\ na))$
using $ff4\ ff3\ ff2$ **by** $(metis\ (no\text{-}types))$ **}**
ultimately have $(nn \notin set\ l \vee increasing\text{-}tree\ nn \wedge lt\text{-}nat\text{-}tree\ n\ nn) \wedge increasing\text{-}tree\ (Node\ n\ l) \vee \neg\ increasing\text{-}tree\ (Node\ n\ l) \wedge (\exists na.\ na \in set\ l \wedge (\neg\ increasing\text{-}tree\ na \vee \neg\ lt\text{-}nat\text{-}tree\ n\ na))$
by $blast$ **}**
ultimately have $(nn \notin set\ l \vee increasing\text{-}tree\ nn \wedge lt\text{-}nat\text{-}tree\ n\ nn) \wedge increasing\text{-}tree\ (Node\ n\ l) \vee \neg\ increasing\text{-}tree\ (Node\ n\ l) \wedge (\exists na.\ na \in set\ l \wedge (\neg\ increasing\text{-}tree\ na \vee \neg\ lt\text{-}nat\text{-}tree\ n\ na))$
by $blast$ **}**
ultimately have $(nn \notin set\ l \vee increasing\text{-}tree\ nn \wedge lt\text{-}nat\text{-}tree\ n\ nn) \wedge increasing\text{-}tree\ (Node\ n\ l) \vee \neg\ increasing\text{-}tree\ (Node\ n\ l) \wedge (\exists na.\ na \in set\ l \wedge (\neg\ increasing\text{-}tree\ na \vee \neg\ lt\text{-}nat\text{-}tree\ n\ na))$
by $blast$ **}**
then have $(nn \notin set\ l \vee increasing\text{-}tree\ nn \wedge lt\text{-}nat\text{-}tree\ n\ nn) \wedge increasing\text{-}tree$

$(Node\ n\ l) \lor \neg$ *increasing-tree* $(Node\ n\ l) \land (\exists\, na.\ na \in set\ l \land (\neg$ *increasing-tree* $na \lor \neg$ *lt-nat-tree* $n\ na))$

using *ff3 ff2 ff1* **by** (*metis* (*no-types*) *nattree.inject*) **}**

then have $(nn \notin set\ l \lor$ *increasing-tree* $nn \land$ *lt-nat-tree* $n\ nn) \land$ *increasing-tree* $(Node\ n\ l) \lor \neg$ *increasing-tree* $(Node\ n\ l) \land (\exists\, na.\ na \in set\ l \land (\neg$ *increasing-tree* $na \lor \neg$ *lt-nat-tree* $n\ na))$

using *ff3 ff1* **by** (*metis* (*no-types*) *list.set-intros*(*1*) *nattree.inject*) **}**

then show *?thesis*

by *auto*

qed

definition *ListMax* :: *nat list* ⇒ *nat* **where**

ListMax l = *foldr max l 0*

lemma *ListMax-0* [*simp*]: *ListMax* [] = *0*

by (*simp add*: *ListMax-def*)

lemma *Listmax-ge* [*simp*]: $\forall\, x \in set\ l.\ x \leq ListMax\ l$

proof (*induction l*)

case *Nil*

then show *?case*

by *auto*

next

case (*Cons a l*)

have *ListMax* (*Cons a l*) = *max a* (*ListMax l*)

using *ListMax-def* **by** *auto*

have $ListMax\ l \leq ListMax\ (Cons\ a\ l) \land a \leq ListMax\ (Cons\ a\ l)$

by (*simp add*: ‹*ListMax* (*a* # *l*) = *max a* (*ListMax l*)›)

then show *?case*

using *Cons.IH* **by** *auto*

qed

fun *height* :: *nattree* ⇒ *nat* **where**

height Empty = *0*

| *height* (*Node n bl*) = (*if Leaf* (*Node n bl*) *then 0 else Suc* (*ListMax* (*map height bl*)))

lemma *height-Leaf* [*simp*]: *Leaf n* ⟶ *height n* = *0*

by (*metis height.elims*)

lemma *Leaf-ind* [*simp*]: *Leaf* (*Node n l*) = *Leaf* (*Node n* (*Empty*#*l*))

by (*metis Leaf.simps ListOfEmpty.simps list.distinct*(*1*) *list.sel*(*3*) *nattree.inject*)

lemma *not-ListOfEmpty-imp-not-Empty-existence* [*simp*] :¬ *ListOfEmpty l* ⟶ $(\exists\, x \in set\ l.\ x \neq Empty)$

proof (*induction l*)

case *Nil*

then show *?case*

by (*simp add*: *ListOfEmpty.Nil*)

next
  case $(Cons\ a\ l)$
  then have $(\forall x \in set\ l.\ x = Empty) \longrightarrow ListOfEmpty\ l$
    by *auto*
  then have $a = Empty \wedge (\forall x \in set\ l.\ x = Empty) \longrightarrow ListOfEmpty\ (a\#l)$
    using *ListOfEmpty.Cons* **by** *blast*
  then have $\neg\ ListOfEmpty\ (a\#l) \longrightarrow (a \neq Empty \vee (\exists x \in set\ l.\ x \neq Empty))$
    by *blast*
  then show *?case* **by** *simp*
qed

lemma *not-Leaf-imp-not-List-of-empty* [*simp*]:
$\neg\ Leaf\ (Node\ n\ l) \longrightarrow (\exists x \in set\ l.\ x \neq Empty)$
proof –
  have $\neg\ Leaf\ (Node\ n\ l) \longrightarrow \neg\ ListOfEmpty\ l$
    using *Leaf.intros* **by** *blast*
  then show *?thesis* **using** *not-ListOfEmpty-imp-not-Empty-existence*
    by *blast*
qed

lemma *Leaf-non-ListOfEmpty* [*simp*]:
$(\exists x \in set\ l.\ x \neq Empty) = (\neg\ Leaf\ (Node\ n\ l))$
proof –
  have $(\exists x \in set\ l.\ x \neq Empty) \longrightarrow (\neg\ Leaf\ (Node\ n\ l))$
    by (*metis Leaf.cases increasing-tree-branch-list-of-empty increasing-tree-ind*
lt-nat-tree.elims(*2*) *nat-less-le nattree.inject*)
  then show *?thesis* **using** *not-Leaf-imp-not-List-of-empty* **by** *blast*
qed

lemma *height-ge* [*simp*]: $\forall x \in set\ l.\ height\ x\ \leq height\ (Node\ n\ l)$
proof (*induction l*)
  case *Nil*
  then show *?case*
    by (*metis empty-iff empty-set*)
next
  case $(Cons\ a\ l)$
  have *a1*: $height\ (Node\ n\ (Cons\ a\ l)) = (if\ Leaf\ (Node\ n\ (Cons\ a\ l))\ then\ 0\ else$
$Suc\ (ListMax$
$(map\ height\ (Cons\ a\ l))))$
    using *height.simps*(*2*) **by** *blast*
  then show *?case*
  proof (*cases a*)
    case *Empty*
    then show *?thesis*
    by (*metis* (*no-types, lifting*) *Leaf-non-ListOfEmpty Listmax-ge a1 height.simps*(*1*)
image-eqI
le-SucI list.set-map order-refl)
  next
    case (*Node x21 x22*)

then show *?thesis*
by (*metis* (*no-types*, *lifting*) *Leaf-non-ListOfEmpty Listmax-ge a1 height.simps*(*1*) *image-eqI*
le-SucI le-numeral-extra(*3*) *list.set-map*)
qed
qed

lemma *listmax-0* [*simp*]: $(\forall\ x \in set\ l.\ f\ x = 0) \longrightarrow ListMax\ (map\ f\ l) = 0$
proof (*induction l*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a l*)
have $ListMax\ (map\ f\ (Cons\ a\ l)) = max\ (f\ a)\ (ListMax\ (map\ f\ l))$
using *ListMax-def* **by** *auto*
then have $(f\ a = 0) \wedge (\forall\ x \in set\ l.\ f\ x = 0) \longrightarrow ListMax\ (map\ f\ (Cons\ a\ l)) = 0$
using *Cons.IH* **by** *linarith*
then show *?case*
by *simp*
qed

I use Type nat option to screen out a branch without a labelled node; however I still use ListMax assuming there is only one node labelled by the second argument.

inductive *ListOfNone* :: (*'a option*) *list* ⇒ *bool* **where**
Nil: *ListOfNone* []
| *Cons* : *ListOfNone n* ⟹ *ListOfNone* (*None#n*)

fun *maxOption* :: *nat option* ⇒ *nat option* ⇒ *nat option* **where**
maxOption None x = *x*
| *maxOption* (*Some n*) *x* = (*case x of Some m* ⇒ *Some* (*max n m*) | *None* ⇒ *Some n*)

definition *ListMaxOption* :: (*nat option*) *list* ⇒ *nat option* **where**
ListMaxOption l = *foldr maxOption l None*

definition *SucOption* :: *nat option* ⇒ *nat option* **where**
SucOption n = (*case n of None* ⇒ *None* | *Some n* ⇒ *Some* (*Suc n*))

fun *le-option* :: *nat option* ⇒ *nat option* ⇒ *bool* **where**
le-option None - = *True*
| *le-option* (*Some n*) *x* = (*case x of None* ⇒ *False* | *Some m* ⇒ *n* ≤ *m*)

We don't care None cases

fun *lt-option* :: *nat option* ⇒ *nat option* ⇒ *bool* **where**
lt-option None - = *False*
| *lt-option* - *None* = *False*
| *lt-option* (*Some m*) (*Some n*) = (*m* < *n*)

fun *depth* :: *nattree* ⇒ *nat* ⇒ *nat option* **where**
  depth Empty n = *None*
| *depth* (*Node n bl*) *m* = (*if n* = *m*
    then (*Some 0*)
    else SucOption (*ListMaxOption* (*map* (λ*x*. *depth x m*) *bl*)))

definition *H* :: *bool list* ⇒ *nat set* **where**
  H l = {*x*. *x* ≤ *length l* ∧ ¬ (*nth* (*False#l*) *x*)}

definition *isHonest* :: *bool list* ⇒ *nat* ⇒ *bool* **where**
  isHonest l x = (¬ (*nth* (*False#l*) *x*))

lemma *H-0* [*simp*]: *0* ∈ *H l*
  by (*simp add*: *H-def*)

lemma *getFrom-suc-eq-H* [*simp*]: *x* < *length l* ∧ ¬ *nth l x* ⟷ *Suc x* ∈ *H l*
  by (*simp add*: *H-def less-eq-Suc-le*)

fun *ListSum* :: *nat list* ⇒ *nat* **where**
  ListSum l = *foldr plus l 0*

lemma *ListSum-0* [*simp*] :(∀ *x* ∈ *set l*. *x* = *0*) ⟶ *ListSum l* = *0*
  proof (*induction l*)
    case *Nil*
    then show *?case* **by** *simp*
  next
    case (*Cons a l*)
    then show *?case*
      by *simp*
  qed

No prunning used as we don't yet have an increasing tree in argument, but can improve it later

fun *count-node* :: *nat* ⇒ *nattree* ⇒ *nat* **where**
  count-node - Empty = *0*
| *count-node m* (*Node n bl*) = (*of-bool* (*m* = *n*)) + *ListSum* (*map* (*count-node m*) *bl*)

lemma *count-node-Leaf* [*simp*] : *Leaf* (*Node n l*) ⟶ *count-node m* (*Node n l*) = *of-bool* (*m* = *n*)
proof −
  have *Leaf* (*Node n l*) ⟶ (∀ *x* ∈ *set l*. *count-node m x* = *0*)
    by (*metis Leaf-non-ListOfEmpty count-node.simps*(*1*))
  then have *Leaf* (*Node n l*) ⟶ *ListSum* (*map* (*count-node m*) *l*) = *0*
    by (*metis ListSum-0 Listmax-ge le-zero-eq listmax-0*)
  then show *?thesis*
    using *count-node.simps*(*2*) **by** *presburger*

qed

definition *unique-node* :: *nattree* ⇒ *nat* ⇒ *bool* **where**
unique-node t n = (*count-node n t* = *1*)

This function returns true only if each member in a set has one and only associated node.

fun *unique-nodes-by-nat-set* :: *nattree* ⇒ *nat set* ⇒ *bool* **where**
unique-nodes-by-nat-set t s = (∀ *x* ∈ *s*. *unique-node t x*)

definition *uniqueH-tree* :: *nattree* ⇒ *bool list* ⇒ *bool* **where**
uniqueH-tree t l = *unique-nodes-by-nat-set t* (*H l*)

lemma *uniqueH-tree-in-imp-l* [*simp*]: ∀ *x* ∈ *H l*. *uniqueH-tree t l* ⟶ *unique-node t x*
using *uniqueH-tree-def* **by** *auto*

lemma *uniqueH-tree-in-imp-r* [*simp*]: (∀ *x* ∈ *H l*. *unique-node t x*) ⟶ *uniqueH-tree t l*
using *uniqueH-tree-def unique-nodes-by-nat-set.simps* **by** *blast*

fun *max-node* :: *nattree* ⇒ *nat* **where**
max-node Empty = *0*
| *max-node* (*Node n bl*) = *ListMax* (*n* # (*map max-node bl*))

lemma *max-node-max* [*simp*]: ∀ *m*. *max-node t* < *m* ⟶ *count-node m t* = *0*
proof (*induction t*)
case *Empty*
then show *?case*
by *simp*
next
case (*Node x1 x2*)
have *a*: *max-node* (*Node x1 x2*) = *ListMax* (*x1* # (*map max-node x2*))
by *simp*
then have ∀ *x* ∈ *set x2*. *max-node x* ≤ *max-node* (*Node x1 x2*) ∧ *x1* ≤ *max-node* (*Node x1 x2*)
by *simp*
then have ∀ *x*. ∀ *y* ∈ *set x2* . *max-node* (*Node x1 x2*)< *x* ⟶ *max-node y* < *x* ∧ *x1* < *x*
using *le-less-trans* **by** *blast*
then have ∀ *x*. ∀ *y* ∈ *set x2* . *max-node* (*Node x1 x2*)< *x* ⟶ *count-node x y* = *0*
by (*simp add*: *Node.IH*)
then have ∀ *x*. *max-node* (*Node x1 x2*)< *x* ⟶ *count-node x* (*Node x1 x2*) = *ListSum* (*map* (*count-node x*) *x2*)
by (*smt Listmax-ge a add.commute add-cancel-left-right count-node.simps*(*2*) *list.set-intros*(*1*)
not-le of-bool-def)
then show *?case*

using *ListSum-0* ‹$\forall x.\ \forall y{\in}set\ x2.\ max\text{-}node\ (Node\ x1\ x2) < x \longrightarrow count\text{-}node\ x\ y = 0$› **by** *auto*
qed

fun *increasing-depth-H* :: *nattree* ⇒ *bool list* ⇒ *bool* **where**
increasing-depth-H t l = $(\forall x \in H\ l.\ \forall y \in H\ l.\ x < y \longrightarrow lt\text{-}option\ (depth\ t\ x)\ (depth\ t\ y))$

inductive *root-label-0* :: *nattree* ⇒ *bool* **where**
root-label-0 (*Node 0 l*)

lemma *root-label-0-depth-0* [*simp*] : *root-label-0 n* ⟶ *depth n 0 = Some 0*
by (*metis depth.simps*(2) *root-label-0.cases*)

F —- w

fun *isFork* :: *bool list* ⇒ *nattree* ⇒ *bool* **where**
isFork w F = ((*length w* ≥ *max-node F*)
∧ (*increasing-tree F*)
∧ (*uniqueH-tree F w*)
∧ (*increasing-depth-H F w*)
∧ *root-label-0 F*)

lemma *isFork-max-not-exceed* [*simp*] : *isFork w F* ⟶ *length w* ≥ *max-node F* **by** *simp*

lemma *isFork-root-0* [*simp*] : *isFork w F* ⟶ *root-label-0 F* **by** *simp*

lemma *isFork-increasing-tree* [*simp*] : *isFork w F* ⟶ *increasing-tree F*
using *isFork.simps* **by** *blast*

lemma *isFork-uniqueH-tree* [*simp*] : *isFork w F* ⟶ $(\forall x \in H\ w.\ unique\text{-}node\ F\ x)$
by (*meson isFork.elims*(2) *uniqueH-tree-in-imp-l*)

lemma *isFork-increasing-depth-H* [*simp*] :
isFork w F ⟶ $(\forall\ x \in H\ w.\ \forall\ y \in H\ w.\ x < y \longrightarrow lt\text{-}option\ (depth\ F\ x)\ (depth\ F\ y))$
by (*meson increasing-depth-H.elims*(2) *isFork.elims*(2))

fun *getLabelFromTine* :: *nattree* ⇒ *nat list* ⇒ *nat list* **where**
getLabelFromTine Empty l = []
| *getLabelFromTine* - [] = []
| *getLabelFromTine* (*Node* - *l*) (*x#xs*) = (*if x* ≥ *length l then* [] *else*
(*case nth l x of*
Empty ⇒ [] | (∗*it runs out of nodes before we can trace down all paths*∗)
Node n - ⇒ *n* # *getLabelFromTine* (*hd* (*drop x l*)) *xs*))

This function provides a set of all path possible, starting from a root by comparing between the length of lists of all choices of edges and lists of their labels.

fun *set-of-tines* :: *nattree* ⇒ (*nat list*) *set* **where**
set-of-tines t = {*tine*. *length tine* = *length* (*getLabelFromTine t tine*)}

fun *edge-disjoint-tines* :: *nat list* ⇒ *nat list* ⇒ *bool* **where**
edge-disjoint-tines [] - = *True*
| *edge-disjoint-tines* - [] = *True*
| *edge-disjoint-tines* (*x*#*xs*) (*y*#*ys*) = (*x*≠*y*)

Definition 4.11: flatTree

fun *flatTree* :: *nattree* ⇒ *bool* **where**
flatTree F =
(∃ *t1* ∈ *set-of-tines F*.
∃ *t2* ∈ *set-of-tines F*.
length t1 = *length t2*
∧ *length t1* = *height F*
∧ *edge-disjoint-tines t1 t1*)

lemma *Leaf-imp-nil-label-tine* [*simp*]: **assumes** *Leaf* (*Node n l*) **shows** *getLabelFromTine* (*Node n l*) *t* = []
proof (*cases t*)
case *Nil*
then show *?thesis*
using *getLabelFromTine.simps*(*2*) **by** *blast*
next
case (*Cons a list*)
then show *?thesis*
proof (*cases a* ≥ *length l*)
case *True*
then show *?thesis*
using *getLabelFromTine.simps*(*3*) *local.Cons* **by** *presburger*
next
case *False*
have *a* < *length l*
using *False* **by** *auto*
then have *nth l a* = *Empty*
using *Leaf-non-ListOfEmpty assms nth-mem* **by** *blast*
then show *?thesis*
by (*simp add*: *local.Cons*)
qed
qed

lemma *flatTree-trivial* [*simp*]: **assumes** *Leaf* (*Node n l*) **shows** *flatTree* (*Node n l*)
proof −
have *set-of-tines* (*Node n l*) = {*tine*. *length tine* = *length* (*getLabelFromTine* (*Node n l*) *tine*)}
by (*metis set-of-tines.elims*)

then have *set-of-tines* (*Node n l*) = {*tine*. *length tine* = *length* []}
by (*metis* (*no-types*, *lifting*) *Collect-cong Leaf-imp-nil-label-tine assms list.size*(*3*))
then have *set-of-tines* (*Node n l*) = {*tine*. *length tine* = *0*}
by (*metis* (*no-types*) ‹*set-of-tines* (*Node n l*) = {*tine*. *length tine* = *length* []}› *list.size*(*3*))
then have *set-of-tines* (*Node n l*) = {[]}
by *blast*
then show *flatTree* (*Node n l*)
by (*metis assms edge-disjoint-tines.simps*(*1*) *flatTree.simps height.simps*(*2*) *list.size*(*3*) *singletonI*)
qed

definition *isForkable* :: *bool list* ⇒ *bool* **where**
isForkable w = (∃ *F*. *isFork w F* ∧ *flatTree F*)

definition *flatFork* :: *bool list* ⇒ *nattree* ⇒ *bool* **where**
flatFork w F = (*isFork w F* ∧ *flatTree F*)

inductive *ListOfAdverse* :: *bool list* ⇒ *bool* **where**
Nil : *ListOfAdverse* []
| *Cons* : *ListOfAdverse xs* ⟹ *ListOfAdverse* (*True#xs*)

lemma *ListOfAdverse-all-True* [*simp*]: *ListOfAdverse w* ⟶ (∀ *x* ∈ *set w*. *x*)
proof (*induction w*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a w*)
have *ListOfAdverse* (*a#w*) ⟶ *a*
using *ListOfAdverse.cases* **by** *blast*
then show *?case*
using *Cons.IH ListOfAdverse.cases* **by** *auto*
qed

lemma *all-True-ListOfAdverse* [*simp*]: (∀ *x* ∈ *set w*. *x*) ⟶ *ListOfAdverse w*
proof (*induction w*)
case *Nil*
then show *?case*
by (*simp add*: *ListOfAdverse.Nil*)
next
case (*Cons a w*)
then have *a* = *True* ∧ (∀ *x* ∈ *set w*. *x*) ⟶ *ListOfAdverse* (*a#w*)
using *ListOfAdverse.Cons* **by** *blast*
then show *?case* **by** *simp*
qed

lemma *singleton-H-ListOfAdverse* [*simp*]: *ListOfAdverse w* ⟶ *H w* = {*0*}
proof (*induction w*)
case *Nil*

then show *?case*
using *H-def* **by** *auto*
next
case (*Cons a w*)
have *ListOfAdverse* $(a\#w) \longrightarrow a$
using *ListOfAdverse.cases* **by** *blast*
then have *ListOfAdverse* $(a\#w) \longrightarrow (\forall\ x.\ x \leq$ *length* $w \longrightarrow$ *nth* (*False*#(*a*#*w*)) $x =$ *nth* (*False*#*w*) *x*)
by (*smt ListOfAdverse-all-True add.right-neutral add-Suc-right insert-iff le-SucI list.simps*(*15*) *list.size*(*4*) *nth-equal-first-eq*)
have *ListOfAdverse* $(a\#w) \longrightarrow$ (*nth* (*False*#(*a*#*w*)) (*length* (*a*#*w*)))
by (*smt ListOfAdverse-all-True length-0-conv linear list.distinct*(*1*) *nth-equal-first-eq*)
then show *?case*
by (*smt Collect-cong H-0 H-def ListOfAdverse-all-True mem-Collect-eq nth-equal-first-eq singleton-conv*)
qed

lemma *ListOfEmpty-max-node-ListMax-0* [*simp*]:
assumes *ListOfEmpty l*
shows *ListMax* (*map max-node l*) = *0*
by (*metis Leaf.simps Leaf-non-ListOfEmpty assms listmax-0 map-eq-map-tailrec max-node.simps*(*1*))

lemma *max-node-Leaf* [*simp*]:
assumes *Leaf* (*Node n l*)
shows *max-node* (*Node n l*) = *n*
proof −
have *max-node* (*Node n l*) = *ListMax* (*n*#(*map max-node l*)) **by** *simp*
then have *max-node* (*Node n l*) = *max n* (*ListMax* (*map max-node l*))
using *ListMax-def* **by** *auto*
then show *max-node* (*Node n l*) = *n*
using *Leaf.simps assms* **by** *auto*
qed

lemma *flatFork-Trivial* : **assumes** *Leaf* (*Node 0 l*) **and** *ListOfAdverse w* **shows** *flatFork w* (*Node 0 l*)
proof −
have *flatTree* (*Node 0 l*)
using *assms*(*1*) *flatTree-trivial* **by** *blast*
have *prem1*: *length* $w \geq$ *max-node* (*Node 0 l*)
using *assms*(*1*) *max-node-Leaf* **by** *presburger*
have *prem2*: *increasing-tree* (*Node 0 l*)
using *Leaf.cases assms*(*1*) *increasing-tree-branch-list-of-empty* **by** *blast*
have *count-node 0* (*Node 0 l*) = *1*
by (*metis* (*full-types*) *assms*(*1*) *count-node-Leaf of-bool-eq*(*2*))
have *H w* = {*0*} **using** *assms*(*2*) *singleton-H-ListOfAdverse* **by** *blast*
then have *prem3*: *uniqueH-tree* (*Node 0 l*) *w*
by (*smt assms*(*1*) *count-node-Leaf of-bool-eq*(*2*) *singletonD uniqueH-tree-in-imp-r unique-node-def*)

have *prem4*:*increasing-depth-H* (*Node 0 l*) *w*
by (*simp add*: ‹*H w* = {*0*}›)
have *root-label-0* (*Node 0 l*)
by (*simp add*: *root-label-0.intros*)
then show *?thesis*
using ‹*flatTree* (*Node 0 l*)› *flatFork-def isFork.elims*(*3*) *prem1 prem2 prem3 prem4* **by** *blast*
qed

lemma *forkable-eq-exist-flatfork* [*simp*] : *isForkable w* ⟷ (∃ *F*. *flatFork w F*)
using *flatFork-def isForkable-def* **by** *blast*

Definition 4.13 is really tricky as we have to traverse F and F' whether it holds that F subseteq¿ F' at the same time.

fun *isPrefix-list* :: *'a list* ⇒ *'a list* ⇒ *bool* **where**
isPrefix-list [] - = *True*
| *isPrefix-list* (*l#ls*) [] = *False*
| *isPrefix-list* (*l#ls*) (*r#rs*) = ((*l*=*r*) ∧ *isPrefix-list ls rs*)

definition *isPrefix-tine* :: *nattree* ⇒ *nattree* ⇒ *nat list* ⇒ *nat list* ⇒ *bool* **where**
isPrefix-tine nt1 nt2 t1 t2 =
(*isPrefix-list t1 t2* ∧ *isPrefix-list* (*getLabelFromTine nt1 t1*) (*getLabelFromTine nt2 t2*))

definition *isPrefix-tree* :: *nattree* ⇒ *nattree* ⇒ *bool* **where**
isPrefix-tree nt1 nt2 =
(∀ *t1* ∈ *set-of-tines nt1*. ∀ *t2* ∈ *set-of-tines nt2*. *isPrefix-list t1 t2*
⟶ *isPrefix-tine nt1 nt2 t1 t2*)

as this can consider from any list of natural numbers.

definition *isPrefix-fork* :: *bool list* ⇒ *bool list* ⇒ *nattree* ⇒ *nattree* ⇒ *bool* **where**
isPrefix-fork w1 w2 nt1 nt2 =
(*isFork w1 nt1* ∧ *isFork w2 nt2* ∧ *isPrefix-list w1 w2* ∧ *isPrefix-tree nt1 nt2*)

Definition 4.14

fun *closedFork-Hgiven* :: *nattree* ⇒ *nat set* ⇒ *bool* **where**
closedFork-Hgiven Empty - = *True*
| *closedFork-Hgiven* (*Node n l*) *h* = (*if ListOfEmpty l*
then (*n* ∈ *h*)
else foldr conj (*map* (λ*x*. *closedFork-Hgiven x h*) *l*) *True*)

A closed fork has to be a fork of a certain string and closed in regard to that string.

definition *closedFork* :: *nattree* ⇒ *bool list* ⇒ *bool* **where**
closedFork F w = (*isFork w F* ∧ *closedFork-Hgiven F* (*H w*))

lemma *closedFork-ListOfAdverse* [*simp*]:

assumes *Leaf* (*Node 0 l*) **and** *ListOfAdverse w*
shows *closedFork* (*Node 0 l*) *w*
proof −
have *closedFork-Hgiven* (*Node 0 l*) (*H w*)
by (*metis H-0 Leaf.cases assms*(*1*) *closedFork-Hgiven.simps*(*2*) *nattree.inject*)

then show *?thesis*
using *assms*(*1*) *assms*(*2*) *closedFork-def flatFork-Trivial flatFork-def* **by** *blast*
qed

lemma *not-ListOfAdverse-not-trivial-fork* [*simp*]:
assumes *Leaf* (*Node 0 l*) **and** ¬ *ListOfAdverse w*
shows ¬ *isFork w* (*Node 0 l*)
proof −
have $\exists\ x \in set\ w.\ \neg\ x$
using *all-True-ListOfAdverse assms*(*2*) **by** *blast*
then have $\exists\ x.\ x > 0 \wedge x \leq length\ w \wedge \neg\ (nth\ (False\#w)\ x)$
by (*metis Suc-leI in-set-conv-nth nth-Cons-Suc zero-less-Suc*)
then have $\exists\ x.\ x > 0 \wedge x \in H\ w$
by (*simp add*: *H-def*)
then have ¬ *uniqueH-tree* (*Node 0 l*) *w*
by (*metis One-nat-def assms*(*1*) *max-node-Leaf max-node-max nat.simps*(*3*) *uniqueH-tree-in-imp-l unique-node-def*)
then show *?thesis*
using *isFork.simps* **by** *blast*
qed

lemma *Leaf-inp-ListOfAdverse-trivial-fork* [*simp*]:
assumes *Leaf* (*Node 0 l*)
shows *ListOfAdverse w* ⟷ *isFork w* (*Node 0 l*)
using *assms flatFork-Trivial flatFork-def not-ListOfAdverse-not-trivial-fork* **by** *blast*

From Definition 4.15, gap reserve and reach depend on a fork and a characteristic string.

A gap of a tine is a difference between its length and the longest tine's.

definition *gap* :: *nattree* ⇒ *nat list* ⇒ *nat* **where**
gap nt tine = *height nt* − *length tine*

A reserve of a tine is the number of adversarial nodes after the last node of the tine.

definition *reserve* :: *bool list* ⇒ *nat list* ⇒ *nat* **where**
reserve w labeledTine = *foldr* (λ*x*.(*plus* (*of-bool x*))) (*drop* (*ListMax labeledTine*) *w*) *0*

A reach of a tine is simply a difference between its reserve and gap.

definition *reach* :: *nattree* ⇒ *bool list* ⇒ *nat list* ⇒ *int* **where**
reach nt w tine = *int* (*reserve w* (*getLabelFromTine nt tine*)) − *int* (*gap nt tine*)

lambda and mu (or called margin) from Definition 4.16.

definition *lambda* :: *nattree* ⇒ *bool list* ⇒ *int* **where**
lambda t w = *Max* {*r*. ∃ *x* ∈ *set-of-tines t*. *r* = *reach t w x*}

lemma *ListOfAdverse-count-eq-length* :
ListOfAdverse w ⟶ (*foldr* (λ*x*.(*plus* (*of-bool x*))) *w 0*) = *length w*
proof (*induction w*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a w*)
have *ListOfAdverse* (*Cons a w*) ⟶ *a* ∧ *ListOfAdverse w*
using *ListOfAdverse.cases* **by** *blast*
then have *ListOfAdverse* (*Cons a w*) ⟶ *a* ∧ (*foldr* (λ*x*.(*plus* (*of-bool x*))) *w 0*) = *length w*
using *Cons.IH* **by** *blast*
then have *ListOfAdverse* (*Cons a w*) ⟶ *foldr* (λ*x*.(*plus* (*of-bool x*))) (*a* # *w*) *0* = (λ*x*.(*plus* (*of-bool x*))) *a* (*length w*)
by (*metis foldr-Cons o-apply*)
then show *?case*
by (*metis* (*full-types*) *One-nat-def* ‹*ListOfAdverse* (*a* # *w*) ⟶ *a* ∧ *ListOfAdverse w*› *list.size*(*4*) *of-bool-eq*(*2*) *semiring-normalization-rules*(*24*))
qed

lemma *lambda-no-honest* : **assumes** *ListOfAdverse w* **shows** ∃*t*. *isFork w t* ∧ *lambda t w* ≥ *0*
proof −
obtain *l* **where** *ListOfEmpty l*
using *ListOfEmpty.Nil* **by** *auto*
obtain *t* **where** *a*:*Leaf t* ∧ *t* = *Node 0 l* ∧ *isFork w t*
using *Leaf.intros Leaf-inp-ListOfAdverse-trivial-fork* ‹*ListOfEmpty l*› *assms* **by** *blast*
have *b*:*gap t* [] = *0*
by (*metis* ‹*Leaf t* ∧ *t* = *Node 0 l* ∧ *isFork w t*› *gap-def height-Leaf list.size*(*3*) *minus-nat.diff-0*)
have *reserve w* [] ≥ *0*
by *simp*
have *reachge0*: *reach t w* [] ≥ *0*
using ‹*gap t* [] = *0*› *reach-def* **by** *auto*
have ∀ *tine*. *getLabelFromTine t tine* = []
using *Leaf-imp-nil-label-tine a* **by** *blast*
then have ∀ *tine*. *length* (*getLabelFromTine t tine*) = *0*
by *simp*
then have *set-of-tines t* = {[]}
by *simp*
then have *exist*: ∃ *x* ∈ *set-of-tines t*. *reach t w x* ≥ *0*
using *reachge0* **by** *blast*
then have *all*: ∀ *x* ∈ *set-of-tines t*. *reach t w x* ≥ *0*
using ‹*set-of-tines t* = {[]}› **by** *fastforce*

then have $\{r.\ \exists\ x \in \textit{set-of-tines}\ t.\ r = \textit{reach}\ t\ w\ x\} = \{r.\ r = \textit{reach}\ t\ w\ []\}$
using ‹*set-of-tines t* = {[]}› **by** *auto*
hence $\{r.\ \exists\ x \in \textit{set-of-tines}\ t.\ r = \textit{reach}\ t\ w\ x\} = \{r.\ r = \textit{int}\ (\textit{reserve}\ w\ (\textit{getLabelFromTine}\ t\ [])) - \textit{int}\ (\textit{gap}\ t\ [])\}$
using *reach-def* **by** *auto*
hence $\{r.\ \exists\ x \in \textit{set-of-tines}\ t.\ r = \textit{reach}\ t\ w\ x\} = \{r.\ r = \textit{int}\ (\textit{reserve}\ w\ []) - 0\}$
by (*metis a b getLabelFromTine.simps*(*2*) *of-nat-0*)
hence $\{r.\ \exists\ x \in \textit{set-of-tines}\ t.\ r = \textit{reach}\ t\ w\ x\} = \{r.\ r = \textit{int}\ (\textit{foldr}\ (\lambda x.(\textit{plus}\ (\textit{of-bool}\ x)))\ w\ 0)\}$
using *reserve-def* **by** *auto*
hence $\{r.\ \exists\ x \in \textit{set-of-tines}\ t.\ r = \textit{reach}\ t\ w\ x\} = \{\textit{int}\ (\textit{length}\ w)\}$
by (*simp add*: *ListOfAdverse-count-eq-length assms*)
hence *lambda t w* ≥ *0*
using *lambda-def* **by** *auto*
thus *?thesis*
using *a* **by** *blast*
qed

definition *set-of-edge-disjoint-tines* :: *nattree* ⇒ ((*nat list*, *nat list*) *prod*) *set* **where**
set-of-edge-disjoint-tines t
= {(*x*,*y*). *x* ∈ *set-of-tines t*
∧ *y* ∈ *set-of-tines t*
∧ *edge-disjoint-tines x y*}

definition *margin* :: *nattree* ⇒ *bool list* ⇒ *int* **where**
margin t w = *Max* {*r*. (∃ (*a*,*b*) ∈ *set-of-edge-disjoint-tines t*. *r* = *min* (*reach t w a*) (*reach t w b*))}

lemma *margin-no-honest* : **assumes** *ListOfAdverse w* **shows** ∃ *t*. *isFork w t* ∧ *margin t w* ≥ *0*
proof −
obtain *l* **where** *ListOfEmpty l*
using *ListOfEmpty.Nil* **by** *auto*
obtain *t* **where** *a*:*Leaf t* ∧ *t* = *Node 0 l* ∧ *isFork w t*
using *Leaf.intros Leaf-inp-ListOfAdverse-trivial-fork* ‹*ListOfEmpty l*› *assms* **by** *blast*
have *b*:*gap t* [] = *0*
by (*metis* ‹*Leaf t* ∧ *t* = *Node 0 l* ∧ *isFork w t*› *gap-def height-Leaf list.size*(*3*) *minus-nat.diff-0*)
have *reserve w* [] ≥ *0*
by *simp*
have *reachge0*: *reach t w* [] ≥ *0*
using ‹*gap t* [] = *0*› *reach-def* **by** *auto*
have ∀ *tine*. *getLabelFromTine t tine* = []
using *Leaf-imp-nil-label-tine a* **by** *blast*
then have ∀ *tine*. *length* (*getLabelFromTine t tine*) = *0*
by *simp*

then have *set-nil*:*set-of-tines* t = {[]}
by *simp*
then have c:∃ x ∈ *set-of-tines* t. *reach* t w x ≥ *0*
using *reachge0* **by** *blast*
then have d: *set-of-edge-disjoint-tines* t
= {(x,y). x ∈ {[]}
∧ y ∈ {[]}
∧ *edge-disjoint-tines* x y}
using *set-nil set-of-edge-disjoint-tines-def* **by** *auto*
then have ∀ (a,b) ∈ *set-of-edge-disjoint-tines* t.
a = [] ∧ b= []
by *simp*
then have ([],[]) ∈ *set-of-edge-disjoint-tines* t
by (*simp add*: *case-prodI* d)
then have *set-of-edge-disjoint-tines* t = {([],[])}
using ‹∀ (a, b)∈*set-of-edge-disjoint-tines* t. a = [] ∧ b = []› **by** *blast*
then have ∃ (a,b) ∈ *set-of-edge-disjoint-tines* t.
min (*reach* t w a) (*reach* t w b) ≥ *0*
by (*simp add*: *reachge0*)
then have ∀ (a,b) ∈ *set-of-edge-disjoint-tines* t.
min (*reach* t w a) (*reach* t w b) ≥ *0*
using ‹*set-of-edge-disjoint-tines* t = {([], [])}› **by** *auto*
then have {r. ∃ (a,b) ∈ *set-of-edge-disjoint-tines* t. r = *min* (*reach* t w a) (*reach* t w b)} = {r. r = *reach* t w []}
using ‹*set-of-edge-disjoint-tines* t = {([], [])}› **by** *auto*
hence {r. ∃ (a,b) ∈ *set-of-edge-disjoint-tines* t. r = *min* (*reach* t w a) (*reach* t w b)} = {r. r = *int* (*reserve* w (*getLabelFromTine* t [])) − *int* (*gap* t [])}
using *reach-def* **by** *auto*
hence {r. ∃ (a,b) ∈ *set-of-edge-disjoint-tines* t. r = *min* (*reach* t w a) (*reach* t w b)} = {r. r = *int* (*reserve* w []) − *0*}
by (*metis* a b *getLabelFromTine.simps*(*2*) *of-nat-0*)
hence {r. ∃ (a,b) ∈ *set-of-edge-disjoint-tines* t. r = *min* (*reach* t w a) (*reach* t w b)} = {r. r = *int* (*foldr* (λx.(*plus* (*of-bool* x))) w *0*)}
using *reserve-def* **by** *auto*
hence {r. ∃ (a,b) ∈ *set-of-edge-disjoint-tines* t. r = *min* (*reach* t w a) (*reach* t w b)} = {*int* (*length* w)}
by (*simp add*: *ListOfAdverse-count-eq-length assms*)
hence *margin* t w ≥ *0*
using *margin-def* **by** *auto*
thus *?thesis*
using a **by** *blast*
qed

This function is to construct, from an increasing tree, a tree not containing greater-labelled nodes than a certain number.

fun *remove-greater* :: *nat* ⇒ *nattree* ⇒ *nattree* **where**
remove-greater - *Empty* = *Empty*
| *remove-greater* m (*Node* n l) = (*if* n < m *then Node* n (*map* (*remove-greater* m) l) *else Empty*)

definition *max-honest-node* :: *bool list* ⇒ *nat* **where**
max-honest-node w = *Max* {*r*. *r* ∈ *H w*}

fun *count-node-by-set* :: *nat set* ⇒ *nattree* ⇒ *nat* **where**
count-node-by-set - *Empty* = *0*
| *count-node-by-set s* (*Node n l*) = (*of-bool* (*n* ∈ *s*)) + *ListSum* (*map* (*count-node-by-set s*) *l*)

definition *count-honest-node* :: *bool list* ⇒ *nattree* ⇒ *nat* **where**
count-honest-node w t = *count-node-by-set* (*H w*) *t*

lemma *map-ListOfEmpty* [*simp*]: *ListOfEmpty* (*map* (λ*x*. *Empty*) *l*)
apply (*induction l*)
apply (*simp add*: *ListOfEmpty.Nil*)
by (*simp add*: *ListOfEmpty.Cons*)

fun *toClosedFork* :: *bool list* ⇒ *nattree* ⇒ *nattree* **where**
toClosedFork - *Empty* = *Empty*
| *toClosedFork w* (*Node n l*) =
(*if* *count-honest-node w* (*Node n l*) = *of-bool* (*isHonest w n*)
then
(*if isHonest w n then Node n* (*map* (λ*x*. *Empty*) *l*) *else Empty*)
else Node n (*map* (*toClosedFork w*) *l*)
)

lemma *isFork-toClosedFork-isFork* [*simp*]: *isFork w F* ⟶ *isFork w* (*toClosedFork w F*)
sorry

lemma *closedFork-eq-toClosedFork* [*simp*]: *isFork w F* ⟶ *F* = (*toClosedFork w F*)
sorry

lemma *toClosedFork-prefixFork* [*simp*]: *isFork w F* ⟶ *isPrefix-fork w w F* (*toClosedFork w F*)
sorry

lemma *closedFork-deepest-honest-node-eq-height* [*simp*]: *isFork w F* ∧ *closedFork F w* ⟶
depth (*ClosedFork w F*) (*max-honest-node w*) = *Some* (*height F*)
sorry

lemma *obtain-two-non-negative-reach-tines-toClosedFork* [*simp*]:
assumes *isFork w F* ∧ *flatFork w F*
shows *t1* ∈ *set* (*tinelist F*) ∧ *t2* ∈ *set* (*tinelist F*)
∧ *length t1* = *length t2* ∧ *length t1* = *height F*
⟶

($\exists$ *t1'* $\in$ *set* (*tinelist* (*toClosedFork w F*)).
$\exists$ *t2'* $\in$ *set* (*tinelist* (*toClosedFork w F*)).
isPrefix-tine (*toClosedFork w F*) *F t1' t1*
$\wedge$ *isPrefix-tine* (*toClosedFork w F*) *F t2' t2*
$\wedge$ *edge-disjoint-tines t1' t2'*
$\wedge$ *reach* (*toClosedFork w F*) *w t1'* $\geq$ *0*
$\wedge$ *reach* (*toClosedFork w F*) *w t2'* $\geq$ *0*)
sorry

lemma *if-4-17* [*simp*]: **assumes** *isForkable w* **shows** ($\exists$ *F*.(*isFork w F* $\wedge$ *margin F w* $\geq$ *0*))
proof (*cases ListOfAdverse w*)
case *True*
then show *?thesis*
using *margin-no-honest* **by** *blast*
next
case *False*
then show *?thesis* **sorry**
qed

lemma *only-if-4-17* [*simp*]: **assumes** ($\exists$ *F*.(*isFork w F* $\wedge$ *margin F w* $\geq$ *0*))
shows *isForkable w*
proof (*cases ListOfAdverse w*)
case *True*
then show *?thesis*
using *Leaf.intros ListOfEmpty.Nil flatFork-Trivial forkable-eq-exist-flatfork* **by** *blast*
next
case *False*
then show *?thesis* **sorry**
qed

proposition *proposition-4-17* : *isForkable w* $\longleftrightarrow$ ($\exists$ *F*.(*isFork w F* $\wedge$ *margin F w* $\geq$ *0*))
using *if-4-17 only-if-4-17* **by** *blast*

definition *lambda-of-string* :: *bool list* $\Rightarrow$ *int* **where**
lambda-of-string w = *Max* {*t*. ($\exists$ *F*.(*isFork w F* $\wedge$ *closedFork F w* $\wedge$ *t* = *lambda F w*))}

lemma *max-node-lowerbound* : *max-node* (*Node n l*) $\geq$ *n* **by** *simp*

lemma *max-node-lowerbound-branch* : ($\exists$ *x* $\in$ *set l*. *x* = *Node n ll*) $\longrightarrow$ *max-node* (*Node m l*) $\geq$ *n*
by (*metis Listmax-ge dual-order.trans image-eqI list.set-intros*(*2*) *max-node.simps*(*2*) *max-node-lowerbound set-map*)

lemma *isFork-Nil* : **assumes** *isFork* [] *F* **shows** *Leaf F* $\wedge$ *root-label-0 F*
proof –

have *inc* : *increasing-tree F*
  using *assms isFork-increasing-tree* **by** *blast*
have *root0*: *root-label-0 F*
  using *assms isFork-root-0* **by** *blast*
then obtain *l* **where** *Fnode*: $F = Node\ 0\ l$
  using *root-label-0.cases* **by** *blast*
then have $\neg$ *ListOfEmpty l* $\longrightarrow$ ($\exists$ $x \in$ *set l*. $x \neq$ *Empty*) **by** *simp*
then have $\neg$ *ListOfEmpty l* $\longrightarrow$ ($\exists$ *n*. ($\exists$ *ll*. *Node n ll* $\in$ *set l*))
  by (*metis* ‹$F = Node\ 0\ l$› *assms increasing-tree-ind isFork-increasing-tree lt-nat-tree.elims*(*2*))
then have $\neg$ *ListOfEmpty l* $\longrightarrow$ ($\exists$ *n*. $n > 0$ $\wedge$ ($\exists$ *ll*. *Node n ll* $\in$ *set l*))
  using *Fnode inc increasing-tree-ind lt-nat-tree.simps*(*2*) **by** *blast*
then have $\neg$ *ListOfEmpty l* $\longrightarrow$ (*max-node F* $>$ *0*)
  by (*metis Fnode gr0I max-node-lowerbound-branch not-le*)
then show *?thesis*
  by (*metis Fnode Leaf.intros assms isFork-max-not-exceed list.size*(*3*) *not-le root0*)
qed

lemma *label-from-Leaf-eq-nil* : **assumes** *Leaf t* **shows** *getLabelFromTine t x* = []
  by (*metis Leaf.cases Leaf-imp-nil-label-tine assms*)

lemma *reserve-nil-nil* : *reserve* [] [] = *0*
  by (*simp add*: *reserve-def*)

lemma *lambda-of-nil-aux* : **assumes** *isFork* [] *F* $\wedge$ *closedFork F* [] **shows** *lambda F* [] = *0*
proof −
have *f1*: *Leaf F* $\wedge$ *root-label-0 F*
  using *assms isFork-Nil* **by** *blast*
then have *reach F* [] [] = *int* (*reserve* [] (*getLabelFromTine F* [])) − *int* (*gap F* [])
  using *reach-def* **by** *blast*
then have *reach F* [] [] = *int* (*reserve* [] []) − *int 0*
  using *f1 gap-def height-Leaf label-from-Leaf-eq-nil* **by** *presburger*
then have *reach F* [] [] = *0*
  using *reserve-nil-nil* **by** *presburger*
then show *lambda F* [] = *0*
proof −
  have $\forall$ *x*. *getLabelFromTine F x* = []
    using *f1 label-from-Leaf-eq-nil* **by** *blast*
  then have *set-of-tines F* = {*tine*. *length tine* = *0*}
    by (*metis* (*no-types*, *lifting*) *Collect-cong list.size*(*3*) *set-of-tines.elims*)
  then have *set-nil*:*set-of-tines F* = {[]}
    by (*smt Collect-cong length-greater-0-conv list.size*(*3*) *singleton-conv*)
  then have ($\forall$ $x \in$ *set-of-tines F*. *reach F* [] *x* = *0*)
    using *f1* **by** (*metis* ‹*reach F* [] [] = *0*› *singletonD*)
  have ($\exists$ $x \in$ *set-of-tines F*. *reach F* [] *x* = *0*)
    using *set-nil* ‹*reach F* [] [] = *0*› **by** *blast*

have *Max* {*r*. ∃ *x* ∈ *set-of-tines F*. *r* = *reach F* [] *x*} = *0*
using ‹*reach F* [] [] = *0*› *set-nil* **by** *auto*
then show *?thesis*
using *lambda-def* **by** *auto*
qed
qed

lemma *lambda-of-nil* : *lambda-of-string* [] = *0*
proof −
obtain *F* **where** *F*:*isFork* [] *F*
using *ListOfAdverse.Nil margin-no-honest* **by** *blast*
then have *f1*: *Leaf F* ∧ *root-label-0 F*
by (*metis isFork-Nil*)
have *closedFork F* []
using *f1 ListOfAdverse.Nil closedFork-ListOfAdverse root-label-0.cases* **by** *blast*

obtain *ss* **where** *ss*:*ss* = {*t*. ∃*f*. *isFork* [] *f* ∧ *closedFork f* [] ∧ *t* = *lambda f* []}
by *blast*
then have *zero-in*: *ss* = {*0*}
by (*smt Collect-cong F* ‹*closedFork F* []› *lambda-of-nil-aux singleton-conv2*)
then have *Max ss* = *0*
by *simp*
have *Max* {*i*. ∃*n*. *isFork* [] *n* ∧ *closedFork n* [] ∧ *i* = *lambda n* []} = *0*
using ‹*Max ss* = *0*› *ss* **by** *blast*
then show *?thesis*
using *lambda-of-string-def* **by** *presburger*
qed

definition *margin-of-string* :: *bool list* ⇒ *int* **where**
margin-of-string w = *Max* {*t*. (∃*F*.(*isFork w F* ∧ *closedFork F w* ∧ *t* = *margin F w*))}

lemma *margin-of-nil-aux* : **assumes** *isFork* [] *F* ∧ *closedFork F* [] **shows** *margin F* [] = *0*
proof −
have *f1*: *Leaf F* ∧ *root-label-0 F*
using *assms isFork-Nil* **by** *blast*
then have *reach F* [] [] = *int* (*reserve* [] (*getLabelFromTine F* [])) − *int* (*gap F* [])
using *reach-def* **by** *blast*
then have *reach F* [] [] = *int* (*reserve* [] []) − *int 0*
using *f1 gap-def height-Leaf label-from-Leaf-eq-nil* **by** *presburger*
then have *reach0*:*reach F* [] [] = *0*
using *reserve-nil-nil* **by** *presburger*
have ∀*x*. *getLabelFromTine F x* = []
using *f1 label-from-Leaf-eq-nil* **by** *blast*
then have *set-of-tines F* = {*tine*. *length tine* = *0*}
by (*metis* (*no-types*, *lifting*) *Collect-cong list.size*(*3*) *set-of-tines.elims*)
then have *set-of-tines F* = {[]}

by *auto*
then have $\forall x \in$ *set-of-tines F* . $x =$ []
by *auto*
have *edge-disjoint-tines* [] []
by *simp*
then have *set-of-edge-disjoint-tines F* =
{(*x*,*y*). *x* ∈ *set-of-tines F* ∧ *y* ∈ *set-of-tines F*}
using ‹*set-of-tines F* = {[]}› *set-of-edge-disjoint-tines-def* **by** *auto*
then have *all*:∀ (*a*,*b*) ∈ *set-of-edge-disjoint-tines F*. (*a*,*b*) = ([],[])
using ‹∀ *x*∈*set-of-tines F*. *x* = []› **by** *auto*
have *exist*: ∃ (*a*,*b*) ∈ *set-of-edge-disjoint-tines F*. (*a*,*b*) = ([],[])
using ‹*set-of-edge-disjoint-tines F* = {(*x*, *y*). *x* ∈ *set-of-tines F* ∧ *y* ∈ *set-of-tines F*}› ‹*set-of-tines F* = {[]}› **by** *blast*
hence *set-nil-pair*:*set-of-edge-disjoint-tines F*={([],[])}
using *all* **by** *blast*
have *Max* {*r*. (∃ (*a*,*b*) ∈ *set-of-edge-disjoint-tines F*. *r* = *min* (*reach F* [] *a*) (*reach F* [] *b*))} = *0*
using *reach0 set-nil-pair* **by** *auto*
thus *?thesis* **using** *margin-def*
by *simp*
qed

lemma *margin-of-nil*: *margin-of-string* [] = *0*
proof −
obtain *F* **where** *isFork* [] *F*
using *ListOfAdverse.Nil margin-no-honest* **by** *blast*
then have *f1*: *Leaf F* ∧ *root-label-0 F*
by (*metis isFork-Nil*)
have *closedFork F* []
using *f1 ListOfAdverse.Nil closedFork-ListOfAdverse root-label-0.cases* **by** *blast*

obtain *ss* **where** *ss*:*ss* = {*t*. ∃*f*. *isFork* [] *f* ∧ *closedFork f* [] ∧ *t* = *margin f* []}
by *blast*
then have *zero-in*: *ss* = {*0*}
by (*smt Collect-cong ListOfAdverse.Nil closedFork-ListOfAdverse isFork-Nil margin-no-honest margin-of-nil-aux root-label-0.cases singleton-conv*)
then have *Max ss* = *0*
by *simp*
have *Max* {*i*. ∃*n*. *isFork* [] *n* ∧ *closedFork n* [] ∧ *i* = *margin n* []} = *0*
using ‹*Max ss* = *0*› *ss* **by** *blast*
then show *?thesis* **using** *margin-of-string-def*
by *presburger*
qed

definition *m* :: *bool list* ⇒ (*int*, *int*) *prod* **where**
m w = (*lambda-of-string w*, *margin-of-string w*)

lemma *lemma-4-18-trivial-case-m* : *m* [] = (*0*,*0*)

by (*simp add*: *lambda-of-nil m-def margin-of-nil*)

lemma *lemma-4-18-trivial-case-exist-Fork* :
$\exists$ *F*. (*isFork* [] *F* $\wedge$ *closedFork F* [] $\wedge$ (*m* [] = (*lambda F* [], *margin F* [])))
by (*metis ListOfAdverse.Nil closedFork-ListOfAdverse isFork-Nil lambda-of-nil-aux lemma-4-18-trivial-case-m margin-no-honest margin-of-nil-aux root-label-0.cases*)

lemma *lemma-4-18* : (*m* [] = (*0*,*0*)) $\wedge$
($\forall$ *w*. ((*length w* > *0*) $\longrightarrow$ (
(*m* (*w* @ [*True*]) = (*lambda-of-string w* + *1*, *margin-of-string w* + *1*))
$\wedge$ ((*lambda-of-string w* > *margin-of-string w*) $\wedge$ (*margin-of-string w* = *0*)
$\longrightarrow$ (*m* (*w* @ [*False*]) = (*lambda-of-string w* − *1*, *0*)))
$\wedge$ (*lambda-of-string w* = *0* $\longrightarrow$ (*m* (*w* @ [*False*]) = (*0*, *margin-of-string w*
− *1*)))
$\wedge$ (*lambda-of-string w* > *0* $\wedge$ *margin-of-string w* $\neq$ *0* $\longrightarrow$ (*m* (*w* @ [*False*])
= (*lambda-of-string w* − *1*, *margin-of-string w* − *1*)))))
$\wedge$ ($\exists$ *F*. (*isFork w F* $\wedge$ *closedFork F w* $\wedge$ (*m w* = (*lambda F w*, *margin F w*)))))
sorry

end