Forkable-Strings

By Kawin

March 23, 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*

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-empty-branch-list* [*simp*]: *increasing-tree* (*Node n* [])
  by *simp*

lemma *increasing-tree-ind* [*simp*] : (∀ *x* ∈ *set l*. *increasing-tree x* ∧ *lt-nat-tree n x*) ⟷ *increasing-tree* (*Node n l*)
  by (*smt increasing-tree.elims*(*2*) *increasing-tree.elims*(*3*) *length-greater-0-conv length-pos-if-in-set nattree.distinct*(*1*) *nattree.inject*)

definition *ListMax* :: *nat list* ⇒ *nat* **where**
  ListMax l = *foldr max l 0*

lemma *max-of-the-list-0* [*simp*]: *ListMax* [] = *0*

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

lemma *max-of-the-list* [*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 $ListMax\ l \leq ListMax\ (Cons\ a\ l) \wedge a \leq ListMax\ (Cons\ a\ l)$
by (*simp add*: ‹$ListMax\ (a\ \#\ l) = max\ a\ (ListMax\ l)$›)
then show *?case*
using *Cons.IH* **by** *auto*
qed

fun *height-branch* :: $nattree \Rightarrow nat$ **where**
height-branch Empty = *0*
| *height-branch* (*Node* - *l*) = *1* + *ListMax* (*map height-branch l*)

lemma *height-branch-empty-list* [*simp*]: *height-branch* (*Node n* []) = *1*
by *simp*

lemma *height-branch-lt* [*simp*]: $\forall x \in set\ l.$ *height-branch x* < *height-branch* (*Node n l*)
by (*simp add*: *le-imp-less-Suc*)

fun *height* :: $nattree \Rightarrow nat$ **where**
height Empty = *0*
| *height* (*Node n bl*) = *ListMax* (*map height-branch bl*)

lemma *height-empty-list* [*simp*]: *height* (*Node n* []) = *0*
by *simp*

lemma *height-ge* [*simp*]: $\forall x \in set\ l.$ *height-branch x* $\leq$ *height* (*Node n l*)
by *auto*

fun *depth-branch* :: $nat \Rightarrow nattree \Rightarrow nat$ **where**
depth-branch m Empty = *0*
| *depth-branch m* (*Node n l*) = (*if m* = *n then 1 else*
(*case ListMax* (*map* (*depth-branch m*) *l*) *of*
0 $\Rightarrow$ *0* |
Suc n $\Rightarrow$ *Suc* (*Suc n*)))

lemma *depth-branch-eq* [*simp*] : *depth-branch m* (*Node m l*) = *1*
by *simp*

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) ∧ (∀ x ∈ set l. f x = 0) ⟶ ListMax (map f (Cons a l))
= 0
    using Cons.IH by linarith
  then show ?case
    by simp
qed

lemma depth-branch-empty-branch-list [simp]: depth-branch m (Node n []) = of-bool
(m = n)
proof (cases m = n)
  case True
  then show ?thesis by simp
next
  case False
  then show ?thesis by auto
qed

lemma depth-branch-ne-nf [simp]:
(∀ x ∈ set l. depth-branch m x = 0) ∧ m ≠ n ⟶ depth-branch m (Node n l) =
0
by simp

fun depth :: nattree ⇒ nat ⇒ nat where
  depth - 0 = 0
| depth Empty n = 0
| depth (Node n bl) m = ListMax (map (depth-branch m) bl)

lemma depth-root-ge [simp] : m > 0 ⟶ (∀ x ∈ set l. depth-branch m x ≤ depth
(Node n l) m)
  using gr0-conv-Suc image-eqI by auto


fun getFrom′ :: bool list ⇒ nat ⇒ bool where
  getFrom′ [] n = True
| getFrom′ (x#xs) 0 = x
| getFrom′ (x#xs) (Suc n) = getFrom′ xs n



definition H :: bool list ⇒ nat set where
  H l = {x. ¬ getFrom′ (False#l) x}
```

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

lemma *getFrom-suc-eq-H* [*simp*]: ¬ *getFrom′ l x* ⟷ *Suc x* ∈ *H l*
  using *H-def* **by** *auto*

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

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-empty-branch-list* [*simp*] : *count-node m* (*Node n* []) = *of-bool* (*m* = *n*) **by** *simp*

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

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 *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

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 *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* ‹*max-node* (*Node x1 x2*) = *ListMax* (*x1* # *map max-node x2*)› *add.commute add-cancel-left-right count-node.simps*(*2*) *less-irrefl-nat less-le-trans list.set-intros*(*1*) *max-of-the-list of-bool-def*)
    then show *?case*
      using *ListSum-0* ‹∀ *x*. ∀ *y*∈*set x2*. *max-node* (*Node x1 x2*) < *x* ⟶ *count-node x y* = *0*› **by** *auto*
  qed

fun *increasing-depth-H* :: *nattree* ⇒ *bool list* ⇒ *bool* **where**
  increasing-depth-H t l = (∀ *x* ∈ *H l*. ∀ *y* ∈ *H l*. *x* < *y* ⟶ *depth t x* < *depth t y*)

fun *root-label0* :: *nattree* ⇒ *bool* **where**
  root-label0 Empty = *False*
| *root-label0* (*Node 0* -) = *True*
| *root-label0* (*Node* (*Suc n*) -) = *False*

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-label0 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-label0 F* **by** *simp*

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

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

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

fun *flatTree* :: *nattree* $\Rightarrow$ *bool* **where**
  flatTree Empty = *False*
| *flatTree* (*Node* - []) = *True*
| *flatTree* (*Node 0 l*) = (*Suc* (*Suc 0*) $\leq$
foldr ((λ*x*.(λ*y*.(*if x* = (*height* (*Node 0 l*)) *then Suc y else y*)))) (*map height-branch l*) *0*)
| *flatTree* (*Node* (*Suc n*) *l*) = *False*

definition *isForkable* :: *bool list* $\Rightarrow$ *bool* **where**
  isForkable w = ($\exists$ *F*. (*isFork w F*) $\wedge$ *flatTree F*)

fun *order-map* :: (*nat* $\Rightarrow$ *'b* $\Rightarrow$ *'c list*) $\Rightarrow$ *'b list* $\Rightarrow$ *nat* $\Rightarrow$ *'c list* **where**
  order-map f [] - = []
| *order-map f* (*x*#*xs*) *n* = *f n x* @ (*order-map f xs* (*Suc n*))

fun *order-map-disjoint* :: (*nat* $\Rightarrow$ *'b* $\Rightarrow$ *'c list*) $\Rightarrow$ *nat* $\Rightarrow$ *'b list* $\Rightarrow$ (*'c list*) *list* **where**
  order-map-disjoint f - [] = []
| *order-map-disjoint f n* (*x*#*xs*) = *f n x* # (*order-map-disjoint f* (*Suc n*) *xs*)

function *firstN-R* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat list* **where**
  firstN-R i n = (*if i* $\geq$ *n then* [] *else i*#(*firstN-R* (*Suc i*) *n*))
   apply *auto*[*1*] **by** *blast*

termination *firstN-R*
  apply (*relation measure* (λ(*i*,*n*). *n* − *i*))
  apply *simp*
  by *simp*

definition *firstN* :: *nat* $\Rightarrow$ *nat list* **where**
  firstN n = *firstN-R 0 n*

function *tinelist-nat* :: *nat* $\Rightarrow$ *nattree* $\Rightarrow$ (*nat list*) *list* **where**
  tinelist-nat - *Empty* = [[]]
| *tinelist-nat m* (*Node* - []) = [[*m*]]
| *tinelist-nat m* (*Node* - (*x*#*xs*)) = *map* (*Cons m*)

(*foldr append* (*map* (λ(*n*,*t*). *tinelist-nat n t*) (*zip* (*firstN* (*length* (*x*#*xs*))) (*x*#*xs*)))
[])
  apply (*metis depth-branch.cases neq-Nil-conv*)
  by *auto*

fun *total-node* :: *nattree* ⇒ *nat* **where**
  total-node Empty = *0*
| *total-node* (*Node n l*) = *Suc* (*ListSum* (*map total-node l*))

lemma *total-nod-dec* [*simp*] : ∀ *x* ∈ *set l*. *total-node x* < *total-node* (*Node n l*)
proof (*induction l*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons a l*)
  then show *?case* **by** *auto*
qed

lemma [*simp*]: (*a*, *b*) ∈ *set* (*zip* (*firstN* (*Suc* (*length xs*))) (*x* # *xs*)) ⟹
      total-node b < *Suc* (*total-node x* + *foldr op* + (*map total-node xs*) *0*)
using *total-nod-dec set-zip-rightD* **by** *force*

termination *tinelist-nat*
  apply (*relation measure* (λ(*n*,*nt*). *total-node nt*))
  apply *auto*
  done

fun *tinelist* :: *nattree* ⇒ (*nat list*) *list* **where**
  tinelist Empty = []
| *tinelist* (*Node 0 l*) = (*foldr append* (*map* (λ(*n*,*t*). *tinelist-nat n t*) (*zip* (*firstN*
(*length l*)) *l*)) [])
| *tinelist* (*Node* (*Suc n*) *l*) = []

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

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

definition *isPrefix-tines* :: *nattree* ⇒ *nattree* ⇒ *nat list* ⇒ *nat list* ⇒ *bool* **where**
 isPrefix-tines nt1 nt2 t1 t2 =

$(isPrefix\text{-}lists\ t1\ t2 \wedge isPrefix\text{-}lists\ (getLabelFromTine\ nt1\ t1)\ (getLabelFromTine\ nt2\ t2))$

definition $isPrefix\text{-}trees :: nattree \Rightarrow nattree \Rightarrow bool$ **where**
$isPrefix\text{-}trees\ nt1\ nt2 =$
$(\forall t1. (\forall t2. ListMem\ t1\ (tinelist\ nt1) \wedge ListMem\ t2\ (tinelist\ nt2) \longrightarrow isPrefix\text{-}tines\ nt1\ nt2\ t1\ t2))$

definition $isPrefix\text{-}forks :: bool\ list \Rightarrow bool\ list \Rightarrow nattree \Rightarrow nattree \Rightarrow bool$ **where**
$isPrefix\text{-}forks\ w1\ w2\ nt1\ nt2 =$
$(isFork\ w1\ nt1 \wedge isFork\ w2\ nt2 \wedge isPrefix\text{-}lists\ w1\ w2 \wedge isPrefix\text{-}trees\ nt1\ nt2)$

inductive $ListOfEmpty :: nattree\ list \Rightarrow bool$ **where**
$Nil : ListOfEmpty\ []$
$|\ Cons : ListOfEmpty\ l \Longrightarrow ListOfEmpty\ (Empty\#l)$

fun $closedFork\text{-}Hgiven :: nattree \Rightarrow nat\ set \Rightarrow bool$ **where**
$closedFork\text{-}Hgiven\ Empty\ \text{-} = True$
$|\ closedFork\text{-}Hgiven\ (Node\ n\ l)\ h = (if\ ListOfEmpty\ l$
$then\ (n \in h)$
$else\ foldr\ conj\ (map\ (\lambda x.\ closedFork\text{-}Hgiven\ x\ h)\ l)\ True)$

definition $closedFork :: nattree \Rightarrow bool\ list \Rightarrow bool$ **where**
$closedFork\ t\ w = closedFork\text{-}Hgiven\ t\ (H\ w)$

definition $gap :: nattree \Rightarrow nat\ list \Rightarrow nat$ **where**
$gap\ nt\ tine = height\ nt - (length\ tine)$

definition $reserve :: bool\ list \Rightarrow nat\ list \Rightarrow nat$ **where**
$reserve\ w\ labeledTine = foldr\ (\lambda x.(plus\ (of\text{-}bool\ x)))\ (drop\ (ListMax\ labeledTine)\ w)\ 0$

definition $reach :: nattree \Rightarrow bool\ list \Rightarrow nat\ list \Rightarrow nat$ **where**
$reach\ nt\ w\ tine = reserve\ w\ tine - gap\ nt\ tine$

definition $lambda :: nattree \Rightarrow bool\ list \Rightarrow nat$ **where**
$lambda\ t\ w = ListMax\ (map\ (reach\ t\ w)\ (tinelist\ t))$

fun $crosslist :: 'a\ list \Rightarrow 'a\ list \Rightarrow (('a, 'a)\ prod)\ list$ **where**
$crosslist\ []\ \text{-} = []$
$|\ crosslist\ (x\#xs)\ ys = (map\ (Pair\ x)\ ys)\ @\ (crosslist\ xs\ ys)$

fun *cross-all-pair′* :: (′*a list*) *list* ⇒ (′*a list*,((′*a*, ′*a*) *prod*) *list*) *prod* **where**
cross-all-pair′ [] = ([],[])
| *cross-all-pair′* (*x*#*xs*) = (*x* @ *fst* (*cross-all-pair′ xs*),
(*crosslist x* (*fst* (*cross-all-pair′ xs*))) @ *snd* (*cross-all-pair′ xs*))

definition *cross-all-pair* :: (′*a list*) *list* ⇒ ((′*a*, ′*a*) *prod*) *list* **where**
cross-all-pair l = *snd* (*cross-all-pair′ l*)

fun *list-of-disjoint-edged-tines* :: *nattree* ⇒ ((*nat list*, *nat list*) *prod*) *list* **where**
list-of-disjoint-edged-tines Empty = []
| *list-of-disjoint-edged-tines* (*Node n l*)
= *cross-all-pair* (*order-map-disjoint* (λ*x*. *map* (*Cons x*)) *0* (*map tinelist l*))

definition *margin* :: *nattree* ⇒ *bool list* ⇒ *nat* **where**
margin t w = *foldr* (λ(*a*,*b*).*max* (*min* (*reach t w a*) (*reach t w b*))) (*list-of-disjoint-edged-tines t*) (*0* − (*height t*))

proposition *proposition-4-17* : *isForkable w* ⟷ (∃ *F*.(*isFork w F* ∧ *margin F w* ≥ *0*))
sorry

definition *lambda-of-string* :: *bool list* ⇒ *nat* **where**
lambda-of-string w = (*GREATEST t*. (∃*F*.(*isFork w F* ∧ *closedFork F w* ∧ *t* = *lambda F w*)))

definition *margin-of-string* :: *bool list* ⇒ *nat* **where**
margin-of-string w = (*GREATEST t*. (∃*F*.(*isFork w F* ∧ *closedFork F w* ∧ *t* = *margin F w*)))

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

lemma *lemma-4-18* : (*m* [] = (*0*,*0*)) ∧
(∀ *w*. ((*length w* > *0*) ⟶ (
(*m* (*w* @ [*True*]) = (*lambda-of-string w* + *1*, *margin-of-string w* + *1*))
∧ ((*lambda-of-string w* > *margin-of-string w*) ∧ (*margin-of-string w* = *0*)
⟶ (*m* (*w* @ [*False*]) = (*lambda-of-string w* − *1*, *0*)))
∧ (*lambda-of-string w* = *0* ⟶ (*m* (*w* @ [*False*]) = (*0*, *margin-of-string w* − *1*)))
∧ (*lambda-of-string w* > *0* ∧ *margin-of-string w* ≠ *0* ⟶ (*m* (*w* @ [*False*])
= (*lambda-of-string w* − *1*, *margin-of-string w* − *1*)))))
∧ (∃ *F*. (*isFork w F* ∧ *closedFork F w* ∧ (*m w* = (*lambda F w*, *margin F w*)))))
sorry

end