# Verified Enumeration of Trees

Nils Cremer

December 8, 2023

**Abstract**

This thesis presents the verification of enumeration algorithms for trees. The first algorithm is based on the well known Prüfer-correspondence and allows the enumeration of all possible labeled trees over a fixed finite set of vertices. The second algorithm enumerates rooted, unlabeled trees of a specified size up to graph isomorphisms. It allows for the efficient enumeration without the use of an intermediate encoding of the trees with level sequences, unlike the algorithm by Beyer and Hedetniemi [1] it is based on. Both algorithms are formalized and verified in Isabelle/HOL. The formalization of trees and other graph theoretic results is also presented.

## Contents

# 1 Graphs and Trees

**theory** *Tree-Graph*
**imports** *Undirected-Graph-Theory.Undirected-Graphs-Root*
**begin**

## 1.1 Miscellaneous

**definition** (**in** *ulgraph*) *loops* :: *'a edge set* **where**
*loops* = {*e*∈*E*. *is-loop e*}

**definition** (**in** *ulgraph*) *sedges* :: *'a edge set* **where**
*sedges* = {*e*∈*E*. *is-sedge e*}

**lemma** (**in** *ulgraph*) *union-loops-sedges*: *loops* ∪ *sedges* = *E*
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *disjnt-loops-sedges*: *disjnt loops sedges*
⟨*proof*⟩

**lemma** (**in** *fin-ulgraph*) *finite-loops*: *finite loops*
⟨*proof*⟩

**lemma** (**in** *fin-ulgraph*) *finite-sedges*: *finite sedges*
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *edge-incident-vert*: *e* ∈ *E* ⟹ ∃ *v*∈*V*. *vincident v e*
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *Union-incident-edges*: (⋃ *v*∈*V*. *incident-edges v*) = *E*
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *induced-edges-mono*: $V_1 \subseteq V_2 \Longrightarrow$ *induced-edges* $V_1 \subseteq$ *induced-edges* $V_2$
⟨*proof*⟩

**definition** (**in** *graph-system*) *remove-vertex* :: $'a \Rightarrow 'a$ *pregraph* **where**
*remove-vertex* $v = (V - \{v\}, \{e \in E.\ \neg$ *vincident* $v\ e\})$

**lemma** (**in** *ulgraph*) *ex-neighbor-degree-not-0*:
**assumes** *degree-non-0*: *degree* $v \neq 0$
**shows** $\exists\, u \in V.$ *vert-adj* $v\ u$
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *ex1-neighbor-degree-1*:
**assumes** *degree-1*: *degree* $v = 1$
**shows** $\exists!u.$ *vert-adj* $v\ u$
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *degree-1-edge-partition*:
**assumes** *degree-1*: *degree* $v = 1$
**shows** $E = \{\{THE\ u.$ *vert-adj* $v\ u,\ v\}\} \cup \{e \in E.\ v \notin e\}$
⟨*proof*⟩

**lemma** (**in** *sgraph*) *vert-adj-not-eq*: *vert-adj* $u\ v \Longrightarrow u \neq v$
⟨*proof*⟩

## 1.2 Degree

**lemma** (**in** *ulgraph*) *empty-E-degree-0*: $E = \{\} \Longrightarrow$ *degree* $v = 0$
⟨*proof*⟩

**lemma** (**in** *fin-ulgraph*) *handshaking*: $(\sum v \in V.$ *degree* $v) = 2 *$ *card* $E$
⟨*proof*⟩

**lemma** (**in** *fin-ulgraph*) *degree-remove-adj-ne-vert*:
**assumes** $u \neq v$
**and** *vert-adj*: *vert-adj* $u\ v$
**and** *remove-vertex*: *remove-vertex* $u = (V', E')$
**shows** *ulgraph.degree* $E'\ v =$ *degree* $v - 1$
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *degree-remove-non-adj-vert*:
**assumes** $u \neq v$
**and** *vert-non-adj*: $\neg$ *vert-adj* $u\ v$
**and** *remove-vertex*: *remove-vertex* $u = (V',\ E')$
**shows** *ulgraph.degree* $E'\ v =$ *degree* $v$
⟨*proof*⟩

## 1.3 Walks

**lemma** (**in** *ulgraph*) *walk-edges-induced-edges*: *is-walk p* $\Longrightarrow$ *set* (*walk-edges p*) $\subseteq$ *induced-edges* (*set p*)
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *walk-edges-in-verts*: *e* $\in$ *set* (*walk-edges xs*) $\Longrightarrow$ *e* $\subseteq$ *set xs*
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *is-walk-prefix*: *is-walk* (*xs*@*ys*) $\Longrightarrow$ *xs* $\neq$ [] $\Longrightarrow$ *is-walk xs*
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *split-walk-edge*: {*x*,*y*} $\in$ *set* (*walk-edges p*) $\Longrightarrow$
$\exists$ *xs ys*. *p* = *xs* @ *x* # *y* # *ys* $\vee$ *p* = *xs* @ *y* # *x* # *ys*
⟨*proof*⟩

## 1.4 Paths

**lemma** (**in** *ulgraph*) *is-gen-path-wf*: *is-gen-path p* $\Longrightarrow$ *set p* $\subseteq$ *V*
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *path-wf*: *is-path p* $\Longrightarrow$ *set p* $\subseteq$ *V*
⟨*proof*⟩

**lemma** (**in** *fin-ulgraph*) *length-gen-path-card-V*: *is-gen-path p* $\Longrightarrow$ *walk-length p* $\leq$ *card V*
⟨*proof*⟩

**lemma** (**in** *fin-ulgraph*) *length-path-card-V*: *is-path p* $\Longrightarrow$ *length p* $\leq$ *card V*
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *is-gen-path-prefix*: *is-gen-path* (*xs*@*ys*) $\Longrightarrow$ *xs* $\neq$ [] $\Longrightarrow$ *is-gen-path* (*xs*)
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *connecting-path-append*: *connecting-path u w* (*xs*@*ys*) $\Longrightarrow$ *xs* $\neq$ [] $\Longrightarrow$ *connecting-path u* (*last xs*) *xs*
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *connecting-path-tl*: *connecting-path u v* (*u* # *w* # *xs*) $\Longrightarrow$ *connecting-path w v* (*w* # *xs*)
⟨*proof*⟩

**lemma** (**in** *fin-ulgraph*) *obtain-longest-path*:
**assumes** *e* $\in$ *E*
**and** *sedge*: *is-sedge e*
**obtains** *p* **where** *is-path p* $\forall$ *s*. *is-path s* $\longrightarrow$ *length s* $\leq$ *length p*
⟨*proof*⟩

## 1.5 Cycles

**context** *ulgraph*
**begin**

**definition** *is-cycle2* :: *'a list* ⇒ *bool* **where**
*is-cycle2 xs* ⟷ *is-cycle xs* ∧ *distinct* (*walk-edges xs*)

**lemma** *loop-is-cycle2*: {*v*} ∈ *E* ⟹ *is-cycle2* [*v*, *v*]
⟨*proof*⟩

**end**

**lemma** (**in** *sgraph*) *cycle2-min-length*:
**assumes** *cycle*: *is-cycle2 c*
**shows** *walk-length c* ≥ *3*
⟨*proof*⟩

**lemma** (**in** *fin-ulgraph*) *length-cycle-card-V*: *is-cycle c* ⟹ *walk-length c* ≤ *Suc* (*card V*)
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *is-cycle-connecting-path*: *is-cycle* (*u*#*v*#*xs*) ⟹ *connecting-path v u* (*v*#*xs*)
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *cycle-edges-notin-tl*: *is-cycle2* (*u*#*v*#*xs*) ⟹ {*u*,*v*} ∉ *set* (*walk-edges* (*v*#*xs*))
⟨*proof*⟩

## 1.6 Subgraphs

**locale** *ulsubgraph* = *subgraph* $V_H$ $E_H$ $V_G$ $E_G$ +
*G*: *ulgraph* $V_G$ $E_G$ **for** $V_H$ $E_H$ $V_G$ $E_G$
**begin**

**interpretation** *H*: *ulgraph* $V_H$ $E_H$
⟨*proof*⟩

**lemma** *is-walk*: *H.is-walk xs* ⟹ *G.is-walk xs*
⟨*proof*⟩

**lemma** *is-closed-walk*: *H.is-closed-walk xs* ⟹ *G.is-closed-walk xs*
⟨*proof*⟩

**lemma** *is-gen-path*: *H.is-gen-path p* ⟹ *G.is-gen-path p*
⟨*proof*⟩

**lemma** *connecting-path*: *H.connecting-path u v p* ⟹ *G.connecting-path u v p*
⟨*proof*⟩

**lemma** *is-cycle*: $H.is\text{-}cycle\ c \Longrightarrow G.is\text{-}cycle\ c$
⟨*proof*⟩

**lemma** *is-cycle2*: $H.is\text{-}cycle2\ c \Longrightarrow G.is\text{-}cycle2\ c$
⟨*proof*⟩

**lemma** *vert-connected*: $H.vert\text{-}connected\ u\ v \Longrightarrow G.vert\text{-}connected\ u\ v$
⟨*proof*⟩

**lemma** *is-connected-set*: $H.is\text{-}connected\text{-}set\ V' \Longrightarrow G.is\text{-}connected\text{-}set\ V'$
⟨*proof*⟩

**end**

**lemma** (**in** *graph-system*) *subgraph-remove-vertex*: $remove\text{-}vertex\ v = (V', E') \Longrightarrow subgraph\ V'\ E'\ V\ E$
⟨*proof*⟩

## 1.7 Connectivity

**lemma** (**in** *ulgraph*) *connecting-path-connected-set*:
**assumes** *conn-path*: *connecting-path u v p*
**shows** *is-connected-set* (*set p*)
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *vert-connected-neighbors*:
**assumes** $\{v,u\} \in E$
**shows** *vert-connected v u*
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *connected-empty-E*:
**assumes** *empty*: $E = \{\}$
**and** *connected*: *vert-connected u v*
**shows** $u = v$
⟨*proof*⟩

**lemma** (**in** *fin-ulgraph*) *degree-0-not-connected*:
**assumes** *degree-0*: *degree v = 0*
**and** $u \neq v$
**shows** ¬ *vert-connected v u*
⟨*proof*⟩

**lemma** (**in** *fin-connected-ulgraph*) *degree-not-0*:
**assumes** *card V* ≥ *2*
**and** *inV*: $v \in V$
**shows** *degree v* ≠ *0*
⟨*proof*⟩

**lemma** (**in** *connected-ulgraph*) *V-E-empty*: $E = \{\} \Longrightarrow \exists\, v.\ V = \{v\}$
⟨*proof*⟩

**lemma** (**in** *connected-ulgraph*) *vert-connected-remove-edge*:
**assumes** *e*: $\{u,v\} \in E$
**shows** $\forall\, w \in V.$ *ulgraph.vert-connected* $V$ $(E - \{\{u,v\}\})$ $w$ $u \vee$ *ulgraph.vert-connected* $V$ $(E - \{\{u,v\}\})$ $w$ $v$
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *vert-connected-remove-cycle-edge*:
**assumes** *cycle*: *is-cycle2* $(u\#v\#xs)$
**shows** *ulgraph.vert-connected* $V$ $(E - \{\{u,v\}\})$ $u$ $v$
⟨*proof*⟩

**lemma** (**in** *connected-ulgraph*) *connected-remove-cycle-edges*:
**assumes** *cycle*: *is-cycle2* $(u\#v\#xs)$
**shows** *connected-ulgraph* $V$ $(E - \{\{u,v\}\})$
⟨*proof*⟩

**lemma** (**in** *connected-ulgraph*) *connected-remove-leaf*:
**assumes** *degree*: *degree* $l = 1$
**and** *remove-vertex*: *remove-vertex* $l = (V', E')$
**shows** *ulgraph.is-connected-set* $V'$ $E'$ $V'$
⟨*proof*⟩

**lemma** (**in** *connected-sgraph*) *connected-two-graph-edges*:
**assumes** $u \neq v$
**and** $V$: $V = \{u,v\}$
**shows** $E = \{\{u,v\}\}$
⟨*proof*⟩

## 1.8 Connected components

**context** *ulgraph*
**begin**

**abbreviation** *vert-connected-rel* $\equiv \{(u,v).$ *vert-connected* $u$ $v\}$

**definition** *connected-components* :: *'a set set* **where**
*connected-components* $=$ $V$ // *vert-connected-rel*

**definition** *connected-component-of* :: *'a* $\Rightarrow$ *'a set* **where**
*connected-component-of* $v$ $=$ *vert-connected-rel* `` $\{v\}$

**lemma** *vert-connected-rel-on-V*: *vert-connected-rel* $\subseteq$ $V \times V$
⟨*proof*⟩

**lemma** *vert-connected-rel-refl*: *refl-on* $V$ *vert-connected-rel*
⟨*proof*⟩

**lemma** *vert-connected-rel-sym*: *sym vert-connected-rel*
⟨*proof*⟩

**lemma** *vert-connected-rel-trans*: *trans vert-connected-rel*
⟨*proof*⟩

**lemma** *equiv-vert-connected*: *equiv V vert-connected-rel*
⟨*proof*⟩

**lemma** *connected-component-non-empty*: $V' \in$ *connected-components* $\Longrightarrow V' \neq \{\}$
⟨*proof*⟩

**lemma** *connected-component-connected*: $V' \in$ *connected-components* $\Longrightarrow$ *is-connected-set* $V'$
⟨*proof*⟩

**lemma** *connected-component-wf*: $V' \in$ *connected-components* $\Longrightarrow V' \subseteq V$
⟨*proof*⟩

**lemma** *connected-component-of-self*: $v \in V \Longrightarrow v \in$ *connected-component-of* $v$
⟨*proof*⟩

**lemma** *conn-comp-of-conn-comps*: $v \in V \Longrightarrow$ *connected-component-of* $v \in$ *connected-components*
⟨*proof*⟩

**lemma** *Un-connected-components*: *connected-components* = *connected-component-of* ‘ $V$
⟨*proof*⟩

**lemma** *connected-component-subgraph*: $V' \in$ *connected-components* $\Longrightarrow$ *subgraph* $V'$ (*induced-edges* $V'$) $V$ $E$
⟨*proof*⟩

**lemma** *connected-components-connected2*:
**assumes** *conn-comp*: $V' \in$ *connected-components*
**shows** *ulgraph.is-connected-set* $V'$ (*induced-edges* $V'$) $V'$
⟨*proof*⟩

**lemma** *vert-connected-connected-component*: $C \in$ *connected-components* $\Longrightarrow u \in C \Longrightarrow$ *vert-connected* $u$ $v \Longrightarrow v \in C$
⟨*proof*⟩

**lemma** *connected-components-connected-ulgraphs*:
**assumes** *conn-comp*: $V' \in$ *connected-components*
**shows** *connected-ulgraph* $V'$ (*induced-edges* $V'$)
⟨*proof*⟩

**lemma** *connected-components-partition-on-V*: *partition-on V connected-components*
⟨*proof*⟩

**lemma** *Union-connected-components*: $\bigcup$ *connected-components* = *V*
⟨*proof*⟩

**lemma** *disjoint-connected-components*: *disjoint connected-components*
⟨*proof*⟩

**lemma** *Union-induced-edges-connected-components*: $\bigcup$(*induced-edges* ‘ *connected-components*)
= *E*
⟨*proof*⟩

**lemma** *connected-components-empty-E*:
**assumes** *empty*: *E* = {}
**shows** *connected-components* = {{*v*} | *v*. *v*∈*V*}
⟨*proof*⟩

**lemma** *connected-iff-connected-components*:
**assumes** *non-empty*: *V* ≠ {}
**shows** *is-connected-set V* ⟷ *connected-components* = {*V*}
⟨*proof*⟩

**end**

**lemma** (**in** *connected-ulgraph*) *connected-components*[*simp*]: *connected-components*
= {*V*}
⟨*proof*⟩

**lemma** (**in** *fin-ulgraph*) *finite-connected-components*: *finite connected-components*
⟨*proof*⟩

**lemma** (**in** *fin-ulgraph*) *finite-connected-component*: *C* ∈ *connected-components*
⟹ *finite C*
⟨*proof*⟩

**lemma** (**in** *connected-ulgraph*) *connected-components-remove-edges*:
**assumes** *edge*: {*u*,*v*} ∈ *E*
**shows** *ulgraph.connected-components V* (*E* − {{*u*,*v*}}) =
{*ulgraph.connected-component-of V* (*E* − {{*u*,*v*}}) *u*, *ulgraph.connected-component-of*
*V* (*E* − {{*u*,*v*}}) *v*}
⟨*proof*⟩

**lemma** (**in** *ulgraph*) *connected-set-connected-component*:
**assumes** *conn-set*: *is-connected-set C*
**and** *non-empty*: *C* ≠ {}
**and** ⋀*u v*. {*u*,*v*} ∈ *E* ⟹ *u* ∈ *C* ⟹ *v* ∈ *C*
**shows** *C* ∈ *connected-components*

⟨*proof*⟩

**lemma** (**in** *ulgraph*) *subset-conn-comps-if-Union*:
**assumes** *A-subset-conn-comps*: $A \subseteq$ *connected-components*
**and** *Un-A*: $\bigcup A = V$
**shows** $A =$ *connected-components*
⟨*proof*⟩

**lemma** (**in** *connected-ulgraph*) *exists-adj-vert-removed*:
**assumes** $v \in V$
**and** *remove-vertex*: *remove-vertex* $v = (V',E')$
**and** *conn-component*: $C \in$ *ulgraph.connected-components* $V'$ $E'$
**shows** $\exists\, u{\in}C.$ *vert-adj* $v$ $u$
⟨*proof*⟩

## 1.9 Trees

**locale** *tree* = *fin-connected-ulgraph* +
**assumes** *no-cycles*: ¬ *is-cycle2* $c$
**begin**

**sublocale** *fin-connected-sgraph*
⟨*proof*⟩

**end**

**locale** *spanning-tree* = *ulgraph* $V$ $E$ + $T$: *tree* $V$ $T$ **for** $V$ $E$ $T$ +
**assumes** *subgraph*: $T \subseteq E$

**lemma** (**in** *fin-connected-ulgraph*) *has-spanning-tree*: $\exists\, T.$ *spanning-tree* $V$ $E$ $T$
⟨*proof*⟩

**context** *tree*
**begin**

**definition** *leaf* :: $'a \Rightarrow$ *bool* **where**
*leaf* $v \longleftrightarrow$ *degree* $v = 1$

**definition** *leaves* :: $'a$ *set* **where**
*leaves* $= \{v.$ *leaf* $v\}$

**definition** *non-trivial* :: *bool* **where**
*non-trivial* $\longleftrightarrow$ *card* $V \geq 2$

**lemma** *obtain-2-verts*:
**assumes** *non-trivial*
**obtains** $u$ $v$ **where** $u \in V$ $v \in V$ $u \neq v$
⟨*proof*⟩

**lemma** *leaf-in-V*: *leaf v* $\Longrightarrow$ $v \in V$
⟨*proof*⟩

**lemma** *exists-leaf*:
**assumes** *non-trivial*
**shows** $\exists\, v{\in}V.$ *leaf v*
⟨*proof*⟩

**lemma** *tree-remove-leaf*:
**assumes** *leaf*: *leaf l*
**and** *remove-vertex*: *remove-vertex l* = (*V'*,*E'*)
**shows** *tree V' E'*
⟨*proof*⟩

**end**

**lemma** *tree-induct* [*case-names singolton insert*, *induct set*: *tree*]:
**assumes** *tree*: *tree V E*
**and** *trivial*: $\bigwedge v.$ *tree* {*v*} {} $\Longrightarrow$ *P* {*v*} {}
**and** *insert*: $\bigwedge l\ v\ V\ E.$ *tree V E* $\Longrightarrow$ *P V E* $\Longrightarrow$ $l \notin V \Longrightarrow v \in V \Longrightarrow \{l,v\} \notin E \Longrightarrow$ *tree.leaf* (*insert* {*l*,*v*} *E*) *l* $\Longrightarrow$ *P* (*insert l V*) (*insert* {*l*,*v*} *E*)
**shows** *P V E*
⟨*proof*⟩

**context** *tree*
**begin**

**lemma** *card-V-card-E*: *card V* = *Suc* (*card E*)
⟨*proof*⟩

**end**

**lemma** *card-E-treeI*:
**assumes** *fin-conn-sgraph*: *fin-connected-ulgraph V E*
**and** *card-V-E*: *card V* = *Suc* (*card E*)
**shows** *tree V E*
⟨*proof*⟩

**context** *tree*
**begin**

**lemma** *add-vertex-tree*:
**assumes** $v \notin V$
**and** $w \in V$
**shows** *tree* (*insert v V*) (*insert* {*v*,*w*} *E*)
⟨*proof*⟩

**lemma** *tree-connected-set*:
**assumes** *non-empty*: $V' \neq \{\}$

**and** *subg*: $V' \subseteq V$
**and** *connected-V'*: *ulgraph.is-connected-set* $V'$ (*induced-edges* $V'$) $V'$
**shows** *tree* $V'$ (*induced-edges* $V'$)
⟨*proof*⟩

**lemma** *unique-adj-vert-removed*:
**assumes** $v \in V$
**and** *remove-vertex*: *remove-vertex* $v = (V',E')$
**and** *conn-component*: $C \in$ *ulgraph.connected-components* $V'$ $E'$
**shows** $\exists!u{\in}C.$ *vert-adj* $v$ $u$
⟨*proof*⟩

**lemma** *non-trivial-card-E*: *non-trivial* $\Longrightarrow$ *card* $E \geq 1$
⟨*proof*⟩

**lemma** *V-Union-E*: *non-trivial* $\Longrightarrow$ $V = \bigcup E$
⟨*proof*⟩

**end**

**lemma** *singleton-tree*: *tree* $\{v\}$ $\{\}$
⟨*proof*⟩

**lemma** *tree2*:
**assumes** $u \neq v$
**shows** *tree* $\{u,v\}$ $\{\{u,v\}\}$
⟨*proof*⟩

## 1.10 Graph Isomorphism

**locale** *graph-isomorphism* =
*G*: *graph-system* $V_G$ $E_G$ **for** $V_G$ $E_G$ +
**fixes** $V_H$ $E_H$ $f$
**assumes** *bij-f*: *bij-betw* $f$ $V_G$ $V_H$
**and** *edge-preserving*: ((‘) $f$) ‘ $E_G = E_H$
**begin**

**lemma** *inj-f*: *inj-on* $f$ $V_G$
⟨*proof*⟩

**lemma** $V_H$*-def*: $V_H = f$ ‘ $V_G$
⟨*proof*⟩

**definition** *inv-iso* ≡ *the-inv-into* $V_G$ $f$

**lemma** *graph-system-H*: *graph-system* $V_H$ $E_H$
⟨*proof*⟩

**interpretation** *H*: *graph-system* $V_H$ $E_H$ ⟨*proof*⟩

**lemma** *graph-isomorphism-inv*: *graph-isomorphism* $V_H$ $E_H$ $V_G$ $E_G$ *inv-iso*
⟨*proof*⟩

**interpretation** *inv-iso*: *graph-isomorphism* $V_H$ $E_H$ $V_G$ $E_G$ *inv-iso* ⟨*proof*⟩

**end**

**fun** *graph-isomorph* :: *'a pregraph* ⇒ *'b pregraph* ⇒ *bool* (**infix** ≃ *50*) **where**
($V_G$,$E_G$) ≃ ($V_H$,$E_H$) ⟷ (∃*f*. *graph-isomorphism* $V_G$ $E_G$ $V_H$ $E_H$ *f*)

**lemma** (**in** *graph-system*) *graph-isomorphism-id*: *graph-isomorphism V E V E id*
⟨*proof*⟩

**lemma** (**in** *graph-system*) *graph-isomorph-refl*: (*V*,*E*) ≃ (*V*,*E*)
⟨*proof*⟩

**lemma** *graph-isomorph-sym*: *symp* (≃)
⟨*proof*⟩

**lemma** *graph-isomorphism-trans*: *graph-isomorphism* $V_G$ $E_G$ $V_H$ $E_H$ *f* ⟹ *graph-isomorphism*
$V_H$ $E_H$ $V_F$ $E_F$ *g* ⟹ *graph-isomorphism* $V_G$ $E_G$ $V_F$ $E_F$ (*g o f*)
⟨*proof*⟩

**lemma** *graph-isomorph-trans*: *transp* (≃)
⟨*proof*⟩

**end**

# 2 Enumeration of Labeled Trees

**theory** *Labeled-Tree-Enumeration*
**imports** *Tree-Graph*
**begin**

**definition** *labeled-trees* :: *'a set* ⇒ *'a pregraph set* **where**
*labeled-trees V* = {(*V*,*E*)| *E*. *tree V E*}

## 2.1 Algorithm

Prüfer sequence to tree

**definition** *prufer-sequences* :: *'a list* ⇒ *'a list set* **where**
*prufer-sequences verts* = {*xs*. *length xs* = *length verts* − *2* ∧ *set xs* ⊆ *set verts*}

**fun** *tree-edges-of-prufer-seq* :: *'a list* ⇒ *'a list* ⇒ *'a edge set* **where**
*tree-edges-of-prufer-seq* [*u*,*v*] [] = {{*u*,*v*}}
| *tree-edges-of-prufer-seq verts* (*b*#*seq*) =
(*case find* (λ*x*. *x* ∉ *set* (*b*#*seq*)) *verts of*

*Some a ⇒ insert {a,b} (tree-edges-of-prufer-seq (remove1 a verts) seq))*

**definition** *tree-of-prufer-seq :: 'a list ⇒ 'a list ⇒ 'a pregraph* **where**
*tree-of-prufer-seq verts seq = (set verts, tree-edges-of-prufer-seq verts seq)*

**definition** *labeled-tree-enum :: 'a list ⇒ 'a pregraph list* **where**
*labeled-tree-enum verts = map (tree-of-prufer-seq verts) (List.n-lists (length verts − 2) verts)*

## 2.2 Correctness

Tree to Prüfer sequence

**definition** *remove-vertex-edges :: 'a ⇒ 'a edge set ⇒ 'a edge set* **where**
*remove-vertex-edges v E = {e∈E. ¬ graph-system.vincident v e}*

**lemma** *find-in-list*[*termination-simp*]: *find P verts = Some v ⟹ v ∈ set verts*
⟨*proof*⟩

**lemma** [*termination-simp*]: *find P verts = Some v ⟹ length verts − Suc 0 < length verts*
⟨*proof*⟩

**fun** *prufer-seq-of-tree :: 'a list ⇒ 'a edge set ⇒ 'a list* **where**
*prufer-seq-of-tree verts E =*
*(if length verts ≤ 2 then []*
*else (case find (tree.leaf E) verts of*
*Some leaf ⇒ (THE v. ulgraph.vert-adj E leaf v) # prufer-seq-of-tree (remove1 leaf verts) (remove-vertex-edges leaf E)))*

**locale** *valid-verts =*
**fixes** *verts*
**assumes** *length-verts*: *length verts ≥ 2*
**and** *distinct-verts*: *distinct verts*

**locale** *tree-of-prufer-seq-ctx = valid-verts +*
**fixes** *seq*
**assumes** *prufer-seq*: *seq ∈ prufer-sequences verts*

**lemma** (**in** *valid-verts*) *card-verts*: *card (set verts) = length verts*
⟨*proof*⟩

**lemma** *length-gt-find-not-in-ys*:
**assumes** *length xs > length ys*
**and** *distinct xs*
**shows** ∃ *x. find (λx. x ∉ set ys) xs = Some x*
⟨*proof*⟩

**lemma** (**in** *tree-of-prufer-seq-ctx*) *tree-edges-of-prufer-seq-induct'*:
**assumes** ⋀*u v. P* [*u, v*] []

**and** $\bigwedge$*verts b seq a.*
*find* ($\lambda x.\ x \notin set\ (b\ \#\ seq)$) *verts* = *Some a*
$\Longrightarrow$ $a \in set\ verts \Longrightarrow a \notin set\ (b\ \#\ seq) \Longrightarrow seq \in$ *prufer-sequences* (*remove1 a verts*)
$\Longrightarrow$ *tree-of-prufer-seq-ctx* (*remove1 a verts*) *seq* $\Longrightarrow$ *P* (*remove1 a verts*) *seq* $\Longrightarrow$ *P verts* (*b* # *seq*)
**shows** *P verts seq*
⟨*proof*⟩

**lemma** (**in** *tree-of-prufer-seq-ctx*) *tree-edges-of-prufer-seq-tree*:
**shows** *tree* (*set verts*) (*tree-edges-of-prufer-seq verts seq*)
⟨*proof*⟩

**lemma** (**in** *tree-of-prufer-seq-ctx*) *tree-of-prufer-seq-tree*: (*V*,*E*) = *tree-of-prufer-seq verts seq* $\Longrightarrow$ *tree V E*
⟨*proof*⟩

**lemma** (**in** *valid-verts*) *labeled-tree-enum-trees*:
**assumes** *VE-in-labeled-tree-enum*: (*V*,*E*) ∈ *set* (*labeled-tree-enum verts*)
**shows** *tree V E*
⟨*proof*⟩

## 2.3 Totality

**locale** *prufer-seq-of-tree-context* =
*valid-verts verts* + *tree set verts E* **for** *verts E*
**begin**

**lemma** *prufer-seq-of-tree-induct'*:
**assumes** $\bigwedge u\ v.\ P\ [u,v]\ \{\{u,v\}\}$
**and** $\bigwedge$*verts E l.* ¬ *length verts* ≤ *2* $\Longrightarrow$ *find* (*tree.leaf E*) *verts* = *Some l* $\Longrightarrow$ *tree.leaf E l*
$\Longrightarrow$ $l \in set\ verts \Longrightarrow$ *prufer-seq-of-tree-context* (*remove1 l verts*) (*remove-vertex-edges l E*)
$\Longrightarrow$ *P* (*remove1 l verts*) (*remove-vertex-edges l E*) $\Longrightarrow$ *P verts E*
**shows** *P verts E*
⟨*proof*⟩

**lemma** *prufer-seq-of-tree-wf*: *set* (*prufer-seq-of-tree verts E*) ⊆ *set verts*
⟨*proof*⟩

**lemma** *length-prufer-seq-of-tree*: *length* (*prufer-seq-of-tree verts E*) = *length verts* − *2*
⟨*proof*⟩

**lemma** *prufer-seq-of-tree-prufer-seq*: *prufer-seq-of-tree verts E* ∈ *prufer-sequences verts*
⟨*proof*⟩

**lemma** *count-list-prufer-seq-degree*: $v \in set\ verts \Longrightarrow Suc\ (count\text{-}list\ (prufer\text{-}seq\text{-}of\text{-}tree\ verts\ E)\ v) = degree\ v$
⟨*proof*⟩

**lemma** *not-in-prufer-seq-iff-leaf*: $v \in set\ verts \Longrightarrow v \notin set\ (prufer\text{-}seq\text{-}of\text{-}tree\ verts\ E) \longleftrightarrow leaf\ v$
⟨*proof*⟩

**lemma** *tree-edges-of-prufer-seq-of-tree*: $tree\text{-}edges\text{-}of\text{-}prufer\text{-}seq\ verts\ (prufer\text{-}seq\text{-}of\text{-}tree\ verts\ E) = E$
⟨*proof*⟩

**lemma** *tree-in-labeled-tree-enum*: $(set\ verts,\ E) \in set\ (labeled\text{-}tree\text{-}enum\ verts)$
⟨*proof*⟩

**end**

**lemma** (**in** *valid-verts*) *V-labeled-tree-enum-verts*: $(V,E) \in set\ (labeled\text{-}tree\text{-}enum\ verts) \Longrightarrow V = set\ verts$
⟨*proof*⟩

**theorem** (**in** *valid-verts*) *labeled-tree-enum-correct*: $set\ (labeled\text{-}tree\text{-}enum\ verts) = labeled\text{-}trees\ (set\ verts)$
⟨*proof*⟩

## 2.4 Distinction

**lemma** (**in** *tree-of-prufer-seq-ctx*) *count-prufer-seq-degree*:
**assumes** *v-in-verts*: $v \in set\ verts$
**shows** $Suc\ (count\text{-}list\ seq\ v) = ulgraph.degree\ (tree\text{-}edges\text{-}of\text{-}prufer\text{-}seq\ verts\ seq)\ v$
⟨*proof*⟩

**lemma** (**in** *tree-of-prufer-seq-ctx*) *notin-prufer-seq-iff-leaf*:
**assumes** $v \in set\ verts$
**shows** $v \notin set\ seq \longleftrightarrow tree.leaf\ (tree\text{-}edges\text{-}of\text{-}prufer\text{-}seq\ verts\ seq)\ v$
⟨*proof*⟩

**lemma** (**in** *valid-verts*) *inj-tree-edges-of-prufer-seq*: $inj\text{-}on\ (tree\text{-}edges\text{-}of\text{-}prufer\text{-}seq\ verts)\ (prufer\text{-}sequences\ verts)$
⟨*proof*⟩

**theorem** (**in** *valid-verts*) *distinct-labeld-tree-enum*: $distinct\ (labeled\text{-}tree\text{-}enum\ verts)$
⟨*proof*⟩

**lemma** (**in** *valid-verts*) *cayleys-formula*: $card\ (labeled\text{-}trees\ (set\ verts)) = length\ verts\ \widehat{}\ (length\ verts - 2)$
⟨*proof*⟩

**end**

# 3 Rooted Trees

**theory** *Rooted-Tree*
**imports** *Tree-Graph HOL−Library.FSet*
**begin**

**datatype** *tree* = *Node tree list*

**fun** *tree-size* :: *tree* ⇒ *nat* **where**
  *tree-size* (*Node ts*) = *Suc* (∑ *t*←*ts*. *tree-size t*)

**fun** *height* :: *tree* ⇒ *nat* **where**
  *height* (*Node* []) = *0*
| *height* (*Node ts*) = *Suc* (*Max* (*height* ‘ *set ts*))

Convenient case splitting and induction for trees

**lemma** *tree-cons-exhaust*[*case-names Nil Cons*]:
  (*t* = *Node* [] ⟹ *P*) ⟹ (⋀*r ts*. *t* = *Node* (*r* # *ts*) ⟹ *P*) ⟹ *P*
  ⟨*proof*⟩

**lemma** *tree-rev-exhaust*[*case-names Nil Snoc*]:
  (*t* = *Node* [] ⟹ *P*) ⟹ (⋀*ts r*. *t* = *Node* (*ts* @ [*r*]) ⟹ *P*) ⟹ *P*
  ⟨*proof*⟩

**lemma** *tree-cons-induct*[*case-names Nil Cons*]:
  **assumes** *P* (*Node* [])
    **and** ⋀*t ts*. *P t* ⟹ *P* (*Node ts*) ⟹ *P* (*Node* (*t*#*ts*))
  **shows** *P t*
⟨*proof*⟩

**fun** *lexord-tree* **where**
  *lexord-tree t* (*Node* []) ⟷ *False*
| *lexord-tree* (*Node* []) *r* ⟷ *True*
| *lexord-tree* (*Node* (*t*#*ts*)) (*Node* (*r*#*rs*)) ⟷ *lexord-tree t r* ∨ (*t* = *r* ∧ *lexord-tree*
(*Node ts*) (*Node rs*))

**fun** *mirror* :: *tree* ⇒ *tree* **where**
  *mirror* (*Node ts*) = *Node* (*map mirror* (*rev ts*))

**instantiation** *tree* :: *linorder*
**begin**

**definition**
  *tree-less-def*: (*t*::*tree*) < *r* ⟷ *lexord-tree* (*mirror t*) (*mirror r*)

**definition**
  *tree-le-def*: (*t* :: *tree*) ≤ *r* ⟷ *t* < *r* ∨ *t* = *r*

**lemma** *lexord-tree-empty2*[*simp*]: *lexord-tree* (*Node* []) $r \longleftrightarrow r \neq$ *Node* []
⟨*proof*⟩

**lemma** *mirror-empty*[*simp*]: *mirror* $t =$ *Node* [] $\longleftrightarrow t =$ *Node* []
⟨*proof*⟩

**lemma** *mirror-not-empty*[*simp*]: *mirror* $t \neq$ *Node* [] $\longleftrightarrow t \neq$ *Node* []
⟨*proof*⟩

**lemma** *tree-le-empty*[*simp*]: *Node* [] $\leq t$
⟨*proof*⟩

**lemma** *tree-less-empty-iff*: *Node* [] $< t \longleftrightarrow t \neq$ *Node* []
⟨*proof*⟩

**lemma** *not-tree-less-empty*[*simp*]: $\neg\ t <$ *Node* []
⟨*proof*⟩

**lemma** *tree-le-empty2-iff*[*simp*]: $t \leq$ *Node* [] $\longleftrightarrow t =$ *Node* []
⟨*proof*⟩

**lemma** *lexord-tree-antisym*: *lexord-tree* $t\ r \Longrightarrow \neg$ *lexord-tree* $r\ t$
⟨*proof*⟩

**lemma** *tree-less-antisym*: $(t$::*tree*$) < r \Longrightarrow \neg\ r < t$
⟨*proof*⟩

**lemma** *lexord-tree-not-eq*: *lexord-tree* $t\ r \Longrightarrow t \neq r$
⟨*proof*⟩

**lemma** *tree-less-not-eq*: $(t$::*tree*$) < r \Longrightarrow t \neq r$
⟨*proof*⟩

**lemma** *lexord-tree-irrefl*: $\neg$ *lexord-tree* $t\ t$
⟨*proof*⟩

**lemma** *tree-less-irrefl*: $\neg\ (t$::*tree*$) < t$
⟨*proof*⟩

**lemma** *lexord-tree-eq-iff*: $\neg$ *lexord-tree* $t\ r \wedge \neg$ *lexord-tree* $r\ t \longleftrightarrow t = r$
⟨*proof*⟩

**lemma** *mirror-mirror*: *mirror* (*mirror* $t$) $= t$
⟨*proof*⟩

**lemma** *mirror-inj*: *mirror* $t =$ *mirror* $r \Longrightarrow t = r$
⟨*proof*⟩

**lemma** *tree-less-eq-iff*: $\neg\ (t::tree) < r \wedge \neg\ r < t \longleftrightarrow t = r$
⟨*proof*⟩

**lemma** *lexord-tree-trans*: *lexord-tree t r* $\Longrightarrow$ *lexord-tree r s* $\Longrightarrow$ *lexord-tree t s*
⟨*proof*⟩

**instance**
⟨*proof*⟩

**end**

**lemma** *tree-size-children*: *tree-size* (*Node ts*) = *Suc n* $\Longrightarrow$ $t \in$ *set ts* $\Longrightarrow$ *tree-size* $t \leq n$
⟨*proof*⟩

**lemma** *tree-size-ge-1*: *tree-size t* $\geq$ *1*
⟨*proof*⟩

**lemma** *tree-size-ne-0*: *tree-size t* $\neq$ *0*
⟨*proof*⟩

**lemma** *tree-size-1-iff*: *tree-size t* = *1* $\longleftrightarrow$ *t* = *Node* []
⟨*proof*⟩

**lemma** *length-children*: *tree-size* (*Node ts*) = *Suc n* $\Longrightarrow$ *length ts* $\leq$ *n*
⟨*proof*⟩

**lemma** *height-Node-cons*: *height* (*Node* (*t*#*ts*)) $\geq$ *Suc* (*height t*)
⟨*proof*⟩

**lemma** *height-0-iff*: *height t* = *0* $\Longrightarrow$ *t* = *Node* []
⟨*proof*⟩

**lemma** *height-children*: *height* (*Node ts*) = *Suc n* $\Longrightarrow$ $t \in$ *set ts* $\Longrightarrow$ *height t* $\leq$ *n*
⟨*proof*⟩

**lemma** *height-children-le-height*: $\forall\, t \in$ *set ts*. *height t* $\leq$ *n* $\Longrightarrow$ *height* (*Node ts*) $\leq$ *Suc n*
⟨*proof*⟩

**lemma** *mirror-iff*: *mirror t* = *Node ts* $\longleftrightarrow$ *t* = *Node* (*map mirror* (*rev ts*))
⟨*proof*⟩

**lemma** *mirror-append*: *mirror* (*Node* (*ts*@*rs*)) = *Node* (*map mirror* (*rev rs*) @ *map mirror* (*rev ts*))
⟨*proof*⟩

**lemma** *lexord-tree-snoc*: *lexord-tree* (*Node ts*) (*Node* (*ts*@[*t*]))
⟨*proof*⟩

**lemma** *tree-less-cons*: *Node ts* < *Node* (*t*#*ts*)
⟨*proof*⟩

**lemma** *tree-le-cons*: *Node ts* ≤ *Node* (*t*#*ts*)
⟨*proof*⟩

**lemma** *tree-less-cons'*: *t* ≤ *Node rs* ⟹ *t* < *Node* (*r*#*rs*)
⟨*proof*⟩

**lemma** *tree-less-snoc2-iff*[*simp*]: *Node* (*ts*@[*t*]) < *Node* (*rs*@[*r*]) ⟷ *t* < *r* ∨ (*t* = *r* ∧ *Node ts* < *Node rs*)
⟨*proof*⟩

**lemma** *tree-le-snoc2-iff*[*simp*]: *Node* (*ts*@[*t*]) ≤ *Node* (*rs*@[*r*]) ⟷ *t* < *r* ∨ (*t* = *r* ∧ *Node ts* ≤ *Node rs*)
⟨*proof*⟩

**lemma** *lexord-tree-cons2*[*simp*]: *lexord-tree* (*Node* (*ts*@[*t*])) (*Node* (*ts*@[*r*])) ⟷ *lexord-tree t r*
⟨*proof*⟩

**lemma** *tree-less-cons2*[*simp*]: *Node* (*t*#*ts*) < *Node* (*r*#*ts*) ⟷ *t* < *r*
⟨*proof*⟩

**lemma** *tree-le-cons2*[*simp*]: *Node* (*t*#*ts*) ≤ *Node* (*r*#*ts*) ⟷ *t* ≤ *r*
⟨*proof*⟩

**lemma** *tree-less-sorted-snoc*: *sorted* (*ts*@[*r*]) ⟹ *Node ts* < *Node* (*ts*@[*r*])
⟨*proof*⟩

**lemma** *lexord-tree-comm-prefix*[*simp*]: *lexord-tree* (*Node* (*ss*@*ts*)) (*Node* (*ss*@*rs*)) ⟷ *lexord-tree* (*Node ts*) (*Node rs*)
⟨*proof*⟩

**lemma** *less-tree-comm-suffix*[*simp*]: *Node* (*ts*@*ss*) < *Node* (*rs*@*ss*) ⟷ *Node ts* < *Node rs*
⟨*proof*⟩

**lemma** *tree-le-comm-suffix*[*simp*]: *Node* (*ts*@*ss*) ≤ *Node* (*rs*@*ss*) ⟷ *Node ts* ≤ *Node rs*
⟨*proof*⟩

**lemma** *tree-less-comm-suffix2*: *t* < *r* ⟹ *Node* (*ts*@*t*#*ss*) < *Node* (*r*#*ss*)
⟨*proof*⟩

**lemma** *lexord-tree-append*[*simp*]: *lexord-tree* (*Node ts*) (*Node* (*ts*@*rs*)) ⟷ *rs* ≠ []
⟨*proof*⟩

**lemma** *tree-less-append*[*simp*]: *Node ts* < *Node* (*rs*@*ts*) ⟷ *rs* ≠ []
⟨*proof*⟩

**lemma** *tree-le-append*: *Node ts* ≤ *Node* (*ss*@*ts*)
⟨*proof*⟩

**lemma** *tree-less-singleton-iff*[*simp*]: *Node* (*ts*@[*t*]) < *Node* [*r*] ⟷ *t* < *r*
⟨*proof*⟩

**lemma** *tree-le-singleton-iff*[*simp*]: *Node* (*ts*@[*t*]) ≤ *Node* [*r*] ⟷ *t* < *r* ∨ (*t* = *r* ∧ *ts* = [])
⟨*proof*⟩

**lemma** *lexord-tree-nested*: *lexord-tree t* (*Node* [*t*])
⟨*proof*⟩

**lemma** *tree-less-nested*: *t* < *Node* [*t*]
⟨*proof*⟩

**lemma** *tree-le-nested*: *t* ≤ *Node* [*t*]
⟨*proof*⟩

**lemma** *lexord-tree-iff*:
*lexord-tree t r* ⟷ (∃ *ts t′ ss rs r′*. *t* = *Node* (*ss* @ *t′* # *ts*) ∧ *r* = *Node* (*ss* @ *r′* # *rs*) ∧ *lexord-tree t′ r′*) ∨ (∃ *ts rs*. *rs* ≠ [] ∧ *t* = *Node ts* ∧ *r* = *Node* (*ts* @ *rs*)) (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

**lemma** *tree-less-iff*: *t* < *r* ⟷ (∃ *ts t′ ss rs r′*. *t* = *Node* (*ts* @ *t′* # *ss*) ∧ *r* = *Node* (*rs* @ *r′* # *ss*) ∧ *t′* < *r′*) ∨ (∃ *ts rs*. *rs* ≠ [] ∧ *t* = *Node ts* ∧ *r* = *Node* (*rs* @ *ts*)) (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

**lemma** *tree-empty-cons-lt-le*: *r* < *Node* (*Node* [] # *ts*) ⟹ *r* ≤ *Node ts*
⟨*proof*⟩

**fun** *regular* :: *tree* ⇒ *bool* **where**
*regular* (*Node ts*) ⟷ *sorted ts* ∧ (∀ *t*∈*set ts*. *regular t*)

**definition** *n-trees* :: *nat* ⇒ *tree set* **where**
*n-trees n* = {*t*. *tree-size t* = *n*}

**definition** *regular-n-trees* :: *nat* ⇒ *tree set* **where**
*regular-n-trees n* = {*t*. *tree-size t* = *n* ∧ *regular t*}

## 3.1 Rooted Graphs

**type-synonym** *'a rpregraph* = (*'a set*) × (*'a edge set*) × *'a*

**locale** *rgraph* = *graph-system* +
  **fixes** *r*
  **assumes** *root-wf*: *r* ∈ *V*

**locale** *rtree* = *tree* + *rgraph*
**begin**

**definition** *subtrees* :: *'a rpregraph set* **where**
  *subtrees* =
    (*let* (*V'*,*E'*) = *remove-vertex r*
    *in* (λ*C*. (*C*, *graph-system.induced-edges E' C*, *THE r'*. *r'* ∈ *C* ∧ *vert-adj r r'*))
‘ *ulgraph.connected-components V' E'*)

**lemma** *rtree-subtree*:
  **assumes** *subtree*: (*S*,*E*$_S$,*r*$_S$) ∈ *subtrees*
  **shows** *rtree S E*$_S$ *r*$_S$
⟨*proof*⟩

**lemma** *finite-subtrees*: *finite subtrees*
⟨*proof*⟩

**lemma** *remove-root-subtrees*:
  **assumes** *remove-vertex*: *remove-vertex r* = (*V'*,*E'*)
    **and** *conn-component*: *C* ∈ *ulgraph.connected-components V' E'*
  **shows** *rtree C* (*graph-system.induced-edges E' C*) (*THE r'*. *r'* ∈ *C* ∧ *vert-adj r r'*)
⟨*proof*⟩

**end**

## 3.2 Rooted Graph Isomorphism

**fun** *app-rgraph-isomorphism* :: (*'a* ⇒ *'b*) ⇒ *'a rpregraph* ⇒ *'b rpregraph* **where**
  *app-rgraph-isomorphism f* (*V*,*E*,*r*) = (*f* ‘ *V*, ((‘) *f*) ‘ *E*, *f r*)

**locale** *rgraph-isomorphism* =
  *G*: *rgraph V*$_G$ *E*$_G$ *r*$_G$ + *graph-isomorphism V*$_G$ *E*$_G$ *V*$_H$ *E*$_H$ *f* **for** *V*$_G$ *E*$_G$ *r*$_G$ *V*$_H$ *E*$_H$ *r*$_H$ *f* +
  **assumes** *root-preserving*: *f r*$_G$ = *r*$_H$
**begin**

**interpretation** *H*: *graph-system V*$_H$ *E*$_H$ ⟨*proof*⟩

**lemma** *rgraph-H*: *rgraph V*$_H$ *E*$_H$ *r*$_H$
  ⟨*proof*⟩

**interpretation** $H$: *rgraph* $V_H$ $E_H$ $r_H$ ⟨*proof*⟩

**lemma** *rgraph-isomorphism-inv*: *rgraph-isomorphism* $V_H$ $E_H$ $r_H$ $V_G$ $E_G$ $r_G$ *inv-iso*

⟨*proof*⟩

**end**

**fun** *rgraph-isomorph* :: *'a rpregraph* ⇒ *'b rpregraph* ⇒ *bool* (**infix** $\simeq_r$ *50*) **where**
($V_G$,$E_G$,$r_G$) $\simeq_r$ ($V_H$,$E_H$,$r_H$) ⟷ (∃*f*. *rgraph-isomorphism* $V_G$ $E_G$ $r_G$ $V_H$ $E_H$ $r_H$ *f*)

**lemma** (**in** *rgraph*) *rgraph-isomorphism-id*: *rgraph-isomorphism V E r V E r id*
⟨*proof*⟩

**lemma** (**in** *rgraph*) *rgraph-isomorph-refl*: ($V$,$E$,$r$) $\simeq_r$ ($V$,$E$,$r$)
⟨*proof*⟩

**lemma** *rgraph-isomorph-sym*: $G \simeq_r H \Longrightarrow H \simeq_r G$
⟨*proof*⟩

**lemma** *rgraph-isomorphism-trans*: *rgraph-isomorphism* $V_G$ $E_G$ $r_G$ $V_H$ $E_H$ $r_H$ *f* ⟹ *rgraph-isomorphism* $V_H$ $E_H$ $r_H$ $V_F$ $E_F$ $r_F$ *g* ⟹ *rgraph-isomorphism* $V_G$ $E_G$ $r_G$ $V_F$ $E_F$ $r_F$ (*g o f*)
⟨*proof*⟩

**lemma** *rgraph-isomorph-trans*: *transp* ($\simeq_r$)
⟨*proof*⟩

**lemma** (**in** *rtree*) *rgraph-isomorphis-app-iso*: *inj-on f V* ⟹ *app-rgraph-isomorphism f* ($V$,$E$,$r$) = ($V'$,$E'$,$r'$) ⟹ *rgraph-isomorphism V E r V' E' r' f*
⟨*proof*⟩

**lemma** (**in** *rtree*) *rgraph-isomorph-app-iso*: *inj-on f V* ⟹ ($V$, $E$, $r$) $\simeq_r$ *app-rgraph-isomorphism f* ($V$, $E$, $r$)
⟨*proof*⟩

## 3.3 Conversion between unlabeled, ordered, rooted trees and tree graphs

**datatype** *'a ltree* = *LNode 'a 'a ltree list*

**fun** *ltree-size* :: *'a ltree* ⇒ *nat* **where**
*ltree-size* (*LNode r ts*) = *Suc* ($\sum$ *t*←*ts*. *ltree-size t*)

**fun** *root-ltree* :: *'a ltree* ⇒ *'a* **where**
*root-ltree* (*LNode r ts*) = *r*

**fun** *nodes-ltree* :: *'a ltree* ⇒ *'a set* **where**

*nodes-ltree* (*LNode r ts*) = {*r*} ∪ (⋃ *t*∈*set ts*. *nodes-ltree t*)

**fun** *relabel-ltree* :: (*'a* ⇒ *'b*) ⇒ *'a ltree* ⇒ *'b ltree* **where**
*relabel-ltree f* (*LNode r ts*) = *LNode* (*f r*) (*map* (*relabel-ltree f*) *ts*)

**fun** *distinct-ltree-nodes* :: *'a ltree* ⇒ *bool* **where**
*distinct-ltree-nodes* (*LNode a ts*) ⟷ (∀ *t*∈*set ts*. *a* ∉ *nodes-ltree t*) ∧ *distinct ts* ∧ *disjoint-family-on nodes-ltree* (*set ts*) ∧ (∀ *t*∈*set ts*. *distinct-ltree-nodes t*)

**fun** *postorder-label-aux* :: *nat* ⇒ *tree* ⇒ *nat* × *nat ltree* **where**
*postorder-label-aux n* (*Node* []) = (*n*, *LNode n* [])
| *postorder-label-aux n* (*Node* (*t*#*ts*)) =
(*let* (*n'*, *t'*) = *postorder-label-aux n t in*
*case postorder-label-aux* (*Suc n'*) (*Node ts*) *of*
(*n''*, *LNode r ts'*) ⇒ (*n''*, *LNode r* (*t'*#*ts'*)))

**definition** *postorder-label* :: *tree* ⇒ *nat ltree* **where**
*postorder-label t* = *snd* (*postorder-label-aux 0 t*)

**fun** *tree-ltree* :: *'a ltree* ⇒ *tree* **where**
*tree-ltree* (*LNode r ts*) = *Node* (*map tree-ltree ts*)

**fun** *regular-ltree* :: *'a ltree* ⇒ *bool* **where**
*regular-ltree* (*LNode r ts*) ⟷ *sorted-wrt* (λ*t s*. *tree-ltree t* ≤ *tree-ltree s*) *ts* ∧ (∀ *t*∈*set ts*. *regular-ltree t*)

**datatype** *'a stree* = *SNode 'a 'a stree fset*

**lemma** *stree-size-child-lt*[*termination-simp*]: *t* |∈| *ts* ⟹ *size t* < *Suc* (∑ *s*∈*fset ts*. *Suc* (*size s*))
⟨*proof*⟩

**lemma** *stree-size-child-lt'*[*termination-simp*]: *t* ∈ *fset ts* ⟹ *size t* < *Suc* (∑ *s*∈*fset ts*. *Suc* (*size s*))
⟨*proof*⟩

**fun** *stree-size* :: *'a stree* ⇒ *nat* **where**
*stree-size* (*SNode r ts*) = *Suc* (*fsum stree-size ts*)

**definition** *n-strees* :: *nat* ⇒ *'a stree set* **where**
*n-strees n* = {*t*. *stree-size t* = *n*}

**fun** *root-stree* :: *'a stree* ⇒ *'a* **where**
*root-stree* (*SNode a ts*) = *a*

**fun** *nodes-stree* :: *'a stree* ⇒ *'a set* **where**
*nodes-stree* (*SNode a ts*) = {*a*} ∪ (⋃ *t*∈*fset ts*. *nodes-stree t*)

**fun** *tree-graph-edges* :: *'a stree* ⇒ *'a edge set* **where**

*tree-graph-edges* (*SNode a ts*) = ((λ*t*. {*a*, *root-stree t*}) ‘ *fset ts*) ∪ (⋃ *t*∈*fset ts*. *tree-graph-edges t*)

**fun** *distinct-stree-nodes* :: *'a stree* ⇒ *bool* **where**
*distinct-stree-nodes* (*SNode a ts*) ⟷ (∀ *t*∈*fset ts*. *a* ∉ *nodes-stree t*) ∧ *disjoint-family-on nodes-stree* (*fset ts*) ∧ (∀ *t*∈*fset ts*. *distinct-stree-nodes t*)

**fun** *ltree-stree* :: *'a stree* ⇒ *'a ltree* **where**
*ltree-stree* (*SNode r ts*) = *LNode r* (*SOME xs*. *fset-of-list xs* = *ltree-stree* |‘| *ts* ∧ *distinct xs* ∧ *sorted-wrt* (λ*t s*. *tree-ltree t* ≤ *tree-ltree s*) *xs*)

**fun** *stree-ltree* :: *'a ltree* ⇒ *'a stree* **where**
*stree-ltree* (*LNode r ts*) = *SNode r* (*fset-of-list* (*map stree-ltree ts*))

**definition** *tree-graph-stree* :: *'a stree* ⇒ *'a rpregraph* **where**
*tree-graph-stree t* = (*nodes-stree t*, *tree-graph-edges t*, *root-stree t*)

**function** *stree-of-graph* :: *'a rpregraph* ⇒ *'a stree* **where**
*stree-of-graph* (*V*,*E*,*r*) =
(*if* ¬ *rtree V E r then undefined else*
*SNode r* (*Abs-fset* (*stree-of-graph* ‘ *rtree.subtrees V E r*)))
⟨*proof*⟩

**termination**
⟨*proof*⟩

**definition** *tree-graph* :: *tree* ⇒ *nat rpregraph* **where**
*tree-graph t* = *tree-graph-stree* (*stree-ltree* (*postorder-label t*))

**fun** *relabel-stree* :: (*'a* ⇒ *'b*) ⇒ *'a stree* ⇒ *'b stree* **where**
*relabel-stree f* (*SNode r ts*) = *SNode* (*f r*) ((*relabel-stree f*) |‘| *ts*)

**lemma** *root-ltree-wf*: *root-ltree t* ∈ *nodes-ltree t*
⟨*proof*⟩

**lemma** *root-relabel-ltree*[*simp*]: *root-ltree* (*relabel-ltree f t*) = *f* (*root-ltree t*)
⟨*proof*⟩

**lemma** *nodes-relabel-ltree*[*simp*]: *nodes-ltree* (*relabel-ltree f t*) = *f* ‘ *nodes-ltree t*
⟨*proof*⟩

**lemma** *finite-nodes-ltree*: *finite* (*nodes-ltree t*)
⟨*proof*⟩

**lemma** *root-stree-wf*: *root-stree t* ∈ *nodes-stree t*
⟨*proof*⟩

**lemma** *tree-graph-edges-wf*: *e* ∈ *tree-graph-edges t* ⟹ *e* ⊆ *nodes-stree t*
⟨*proof*⟩

**lemma** *card-tree-graph-edges-distinct*: *distinct-stree-nodes t* $\Longrightarrow$ *e* $\in$ *tree-graph-edges t* $\Longrightarrow$ *card e* = *2*
⟨*proof*⟩

**lemma** *nodes-stree-non-empty*: *nodes-stree t* $\neq$ {}
⟨*proof*⟩

**lemma** *finite-nodes-stree*: *finite* (*nodes-stree t*)
⟨*proof*⟩

**lemma** *finite-tree-graph-edges*: *finite* (*tree-graph-edges t*)
⟨*proof*⟩

**lemma** *root-relabel-stree*[*simp*]: *root-stree* (*relabel-stree f t*) = *f* (*root-stree t*)
⟨*proof*⟩

**lemma** *nodes-stree-relabel-stree*[*simp*]: *nodes-stree* (*relabel-stree f t*) = *f* ‘ *nodes-stree t*
⟨*proof*⟩

**lemma** *tree-graph-edges-relabel-stree*[*simp*]: *tree-graph-edges* (*relabel-stree f t*) = ((‘) *f*) ‘ *tree-graph-edges t*
⟨*proof*⟩

**lemma** *nodes-stree-ltree*[*simp*]: *nodes-stree* (*stree-ltree t*) = *nodes-ltree t*
⟨*proof*⟩

**lemma** *distinct-sorted-wrt-list*: $\exists$ *xs*. *fset-of-list xs* = *A* $\wedge$ *distinct xs* $\wedge$ *sorted-wrt* ($\lambda$*t s*. (*f t* :: *'b*::*linorder*) $\leq$ *f s*) *xs*
⟨*proof*⟩

**abbreviation** *ltree-stree-subtrees ts* $\equiv$ *SOME xs*. *fset-of-list xs* = *ltree-stree* |‘| *ts* $\wedge$ *distinct xs* $\wedge$ *sorted-wrt* ($\lambda$*t s*. *tree-ltree t* $\leq$ *tree-ltree s*) *xs*

**lemma** *fset-of-list-ltree-stree-subtrees*[*simp*]: *fset-of-list* (*ltree-stree-subtrees ts*) = *ltree-stree* |‘| *ts*
⟨*proof*⟩

**lemma** *set-ltree-stree-subtrees*[*simp*]: *set* (*ltree-stree-subtrees ts*) = *ltree-stree* ‘ *fset ts*
⟨*proof*⟩

**lemma** *distinct-ltree-stree-subtrees*: *distinct* (*ltree-stree-subtrees ts*)
⟨*proof*⟩

**lemma** *sorted-wrt-ltree-stree-subtrees*: *sorted-wrt* ($\lambda$*t s*. *tree-ltree t* $\leq$ *tree-ltree s*) (*ltree-stree-subtrees ts*)
⟨*proof*⟩

**lemma** *nodes-ltree-stree*[*simp*]: *nodes-ltree* (*ltree-stree t*) = *nodes-stree t*
⟨*proof*⟩

**lemma** *stree-ltree-stree*[*simp*]: *stree-ltree* (*ltree-stree t*) = *t*
⟨*proof*⟩

**lemma** *nodes-tree-graph-stree*: *tree-graph-stree t* = (*V*, *E*, *r*) ⟹ *V* = *nodes-stree t*
⟨*proof*⟩

**lemma** *relabel-stree-stree-ltree*: *relabel-stree f* (*stree-ltree t*) = *stree-ltree* (*relabel-ltree f t*)
⟨*proof*⟩

**lemma** *relabel-stree-relabel-ltree*: *relabel-ltree f t1* = *t2* ⟹ *relabel-stree f* (*stree-ltree t1*) = *stree-ltree t2*
⟨*proof*⟩

**lemma** *app-rgraph-iso-tree-graph-stree*: *app-rgraph-isomorphism f* (*tree-graph-stree t*) = *tree-graph-stree* (*relabel-stree f t*)
⟨*proof*⟩

**lemma** (**in** *rtree*) *root-stree-of-graph*[*simp*]: *root-stree* (*stree-of-graph* (*V*,*E*,*r*)) = *r*
⟨*proof*⟩

**lemma** (**in** *rtree*) *nodes-stree-stree-of-graph*[*simp*]: *nodes-stree* (*stree-of-graph* (*V*,*E*,*r*)) = *V*
⟨*proof*⟩

**lemma** (**in** *rtree*) *tree-graph-edges-stree-of-graph*[*simp*]: *tree-graph-edges* (*stree-of-graph* (*V*,*E*,*r*)) = *E*
⟨*proof*⟩

**lemma** (**in** *rtree*) *tree-graph-stree-of-graph*[*simp*]: *tree-graph-stree* (*stree-of-graph* (*V*,*E*,*r*)) = (*V*,*E*,*r*)
⟨*proof*⟩

**lemma** *postorder-label-aux-mono*: *fst* (*postorder-label-aux n t*) ≥ *n*
⟨*proof*⟩

**lemma** *nodes-postorder-label-aux-ge*: *postorder-label-aux n t* = (*n′*, *t′*) ⟹ *v* ∈ *nodes-ltree t′* ⟹ *v* ≥ *n*
⟨*proof*⟩

**lemma** *nodes-postorder-label-aux-le*: *postorder-label-aux n t* = (*n′*, *t′*) ⟹ *v* ∈ *nodes-ltree t′* ⟹ *v* ≤ *n′*

⟨*proof*⟩

**lemma** *distinct-nodes-postorder-label-aux*: *distinct-ltree-nodes* (*snd* (*postorder-label-aux n t*))
⟨*proof*⟩

**lemma** *distinct-nodes-postorder-label*: *distinct-ltree-nodes* (*postorder-label t*)
⟨*proof*⟩

**lemma** *distinct-nodes-stree-ltree*: *distinct-ltree-nodes t* ⟹ *distinct-stree-nodes* (*stree-ltree t*)
⟨*proof*⟩

**fun** *distinct-edges* :: *'a stree* ⇒ *bool* **where**
*distinct-edges* (*SNode a ts*) ⟷ *inj-on* (λ*t*. {*a*, *root-stree t*}) (*fset ts*)
∧ (∀ *t*∈*fset ts*. *disjnt* ((λ*t*. {*a*, *root-stree t*}) ‘ *fset ts*) (*tree-graph-edges t*))
∧ *disjoint-family-on tree-graph-edges* (*fset ts*)
∧ (∀ *t*∈*fset ts*. *distinct-edges t*)

**lemma** *distinct-nodes-inj-on-root-stree*: *distinct-stree-nodes* (*SNode r ts*) ⟹ *inj-on root-stree* (*fset ts*)
⟨*proof*⟩

**lemma** *distinct-nodes-disjoint-edges*:
**assumes** *distinct-nodes*: *distinct-stree-nodes* (*SNode a ts*)
**shows** *disjoint-family-on tree-graph-edges* (*fset ts*)
⟨*proof*⟩

**lemma** *card-nodes-edges*: *distinct-stree-nodes t* ⟹ *card* (*nodes-stree t*) = *Suc* (*card* (*tree-graph-edges t*))
⟨*proof*⟩

**lemma** *tree-tree-graph-edges*: *distinct-stree-nodes t* ⟹ *tree* (*nodes-stree t*) (*tree-graph-edges t*)
⟨*proof*⟩

**lemma** *rtree-tree-graph-edges*:
**assumes** *distinct-nodes*: *distinct-stree-nodes t*
**shows** *rtree* (*nodes-stree t*) (*tree-graph-edges t*) (*root-stree t*)
⟨*proof*⟩

**lemma** *rtree-tree-graph-stree*: *distinct-stree-nodes t* ⟹ *tree-graph-stree t* = (*V*,*E*,*r*) ⟹ *rtree V E r*
⟨*proof*⟩

**lemma** *rtree-tree-graph*: *tree-graph t* = (*V*,*E*,*r*) ⟹ *rtree V E r*
⟨*proof*⟩

Cardinality of the resulting rooted tree is correct

**lemma** *ltree-size-postorder-label-aux*: *ltree-size* (*snd* (*postorder-label-aux n t*)) = *tree-size t*
⟨*proof*⟩

**lemma** *ltree-size-postorder-label*: *ltree-size* (*postorder-label t*) = *tree-size t*
⟨*proof*⟩

**lemma** *distinct-nodes-ltree-size-card-nodes*: *distinct-ltree-nodes t* ⟹ *ltree-size t* = *card* (*nodes-ltree t*)
⟨*proof*⟩

**lemma** *distinct-nodes-stree-size-card-nodes*: *distinct-stree-nodes t* ⟹ *stree-size t* = *card* (*nodes-stree t*)
⟨*proof*⟩

**lemma** *stree-size-stree-ltree*: *distinct-ltree-nodes t* ⟹ *stree-size* (*stree-ltree t*) = *ltree-size t*
⟨*proof*⟩

**lemma** *card-tree-graph-stree*: *distinct-stree-nodes t* ⟹ *tree-graph-stree t* = (*V*,*E*,*r*) ⟹ *card V* = *stree-size t*
⟨*proof*⟩

**lemma** *card-tree-graph*: *tree-graph t* = (*V*,*E*,*r*) ⟹ *card V* = *tree-size t*
⟨*proof*⟩

**lemma** [*termination-simp*]: (*t*, *s*) ∈ *set* (*zip ts ss*) ⟹ *size t* < *Suc* (*size-list size ts*)
⟨*proof*⟩

**fun** *obtain-ltree-isomorphism* :: *'a ltree* ⇒ *'b ltree* ⇒ (*'a* ⇀ *'b*) **where**
*obtain-ltree-isomorphism* (*LNode r1 ts*) (*LNode r2 ss*) = *fold* (++) (*map2 obtain-ltree-isomorphism ts ss*) [*r1*↦*r2*]

**fun** *postorder-relabel-aux* :: *nat* ⇒ *'a ltree* ⇒ *nat* × (*nat* ⇀ *'a*) **where**
*postorder-relabel-aux n* (*LNode r* []) = (*n*, [*n* ↦ *r*])
| *postorder-relabel-aux n* (*LNode r* (*t*#*ts*)) =
(*let* (*n'*, *f*$_t$) = *postorder-relabel-aux n t*;
(*n''*, *f*$_{ts}$) = *postorder-relabel-aux* (*Suc n'*) (*LNode r ts*) *in*
(*n''*, *f*$_t$ ++ *f*$_{ts}$))

**definition** *postorder-relabel* :: *'a ltree* ⇒ (*nat* ⇀ *'a*) **where**
*postorder-relabel t* = *snd* (*postorder-relabel-aux 0 t*)

**lemma** *fst-postorder-label-aux-tree-ltree*: *fst* (*postorder-label-aux n* (*tree-ltree t*)) = *fst* (*postorder-relabel-aux n t*)
⟨*proof*⟩

**lemma** *dom-postorder-relabel-aux*: *dom* (*snd* (*postorder-relabel-aux n t*)) = *nodes-ltree* (*snd* (*postorder-label-aux n* (*tree-ltree t*)))
⟨*proof*⟩

**lemma** *ran-postorder-relabel-aux*: *ran* (*snd* (*postorder-relabel-aux n t*)) = *nodes-ltree t*
⟨*proof*⟩

**lemma** *relabel-ltree-eq*: ∀ *v*∈*nodes-ltree t*. *f v* = *g v* ⟹ *relabel-ltree f t* = *relabel-ltree g t*
⟨*proof*⟩

**lemma** *relabel-postorder-relabel-aux*: *relabel-ltree* (*the o snd* (*postorder-relabel-aux n t*)) (*snd* (*postorder-label-aux n* (*tree-ltree t*))) = *t*
⟨*proof*⟩

**lemma** *relabel-postorder-relabel*: *relabel-ltree* (*the o postorder-relabel t*) (*postorder-label* (*tree-ltree t*)) = *t*
⟨*proof*⟩

**lemma** *relabel-postorder-aux-inj*: *distinct-ltree-nodes t* ⟹ *inj-on* (*the o snd* (*postorder-relabel-aux n t*)) (*nodes-ltree* (*snd* (*postorder-label-aux n* (*tree-ltree t*))))
⟨*proof*⟩

**lemma** *relabel-postorder-inj*: *distinct-ltree-nodes t* ⟹ *inj-on* (*the o postorder-relabel t*) (*nodes-ltree* (*postorder-label* (*tree-ltree t*)))
⟨*proof*⟩

**lemma** (**in** *rtree*) *distinct-nodes-stree-of-graph*: *distinct-stree-nodes* (*stree-of-graph* (*V*,*E*,*r*))
⟨*proof*⟩

**lemma** *disintct-nodes-ltree-stree*: *distinct-stree-nodes t* ⟹ *distinct-ltree-nodes* (*ltree-stree t*)
⟨*proof*⟩

**lemma** (**in** *rtree*) *tree-graph-tree-of-graph*: *tree-graph* (*tree-ltree* (*ltree-stree* (*stree-of-graph* (*V*,*E*,*r*)))) ≃$_r$ (*V*,*E*,*r*)
⟨*proof*⟩

**lemma** (**in** *rtree*) *stree-size-stree-of-graph*[*simp*]: *stree-size* (*stree-of-graph* (*V*,*E*,*r*)) = *card V*
⟨*proof*⟩

**lemma** *inj-ltree-stree*: *inj ltree-stree*
⟨*proof*⟩

**lemma** *ltree-size-ltree-stree*[*simp*]: *ltree-size* (*ltree-stree t*) = *stree-size t*
⟨*proof*⟩

**lemma** *tree-size-tree-ltree*[*simp*]: *tree-size* (*tree-ltree t*) = *ltree-size t*
⟨*proof*⟩

**lemma** *regular-ltree-stree*: *regular-ltree* (*ltree-stree t*)
⟨*proof*⟩

**lemma** *regular-tree-ltree*: *regular-ltree t* ⟹ *regular* (*tree-ltree t*)
⟨*proof*⟩

**lemma** (**in** *rtree*) *tree-of-graph-regular-n-tree*: *tree-ltree* (*ltree-stree* (*stree-of-graph* (*V*,*E*,*r*))) ∈ *regular-n-trees* (*card V*) (**is** *?t* ∈ *?A*)
⟨*proof*⟩

**lemma** (**in** *rtree*) *ex-regular-n-tree*: ∃ *t*∈*regular-n-trees* (*card V*). *tree-graph t* ≃$_r$ (*V*,*E*,*r*)
⟨*proof*⟩

## 3.4 Injectivity with respect to isomorphism

**lemma** *app-rgraph-isomorphism-relabel-stree*: *app-rgraph-isomorphism f* (*tree-graph-stree t*) = *tree-graph-stree* (*relabel-stree f t*)
⟨*proof*⟩

Lemmas relating the connected components of the tree graph with the root removed to the subtrees of an stree.

**context**
**fixes** *t r ts V′ E′*
**assumes** *t*: *t* = *SNode r ts*
**assumes** *distinct-nodes*: *distinct-stree-nodes t*
**and** *remove-vertex*: *graph-system.remove-vertex* (*nodes-stree t*) (*tree-graph-edges t*) *r* = (*V′*,*E′*)
**begin**

**interpretation** *t*: *rtree nodes-stree t tree-graph-edges t r* ⟨*proof*⟩

**interpretation** *subg*: *ulsubgraph V′ E′ nodes-stree t tree-graph-edges t* ⟨*proof*⟩

**interpretation** *g′*: *ulgraph V′ E′* ⟨*proof*⟩

**lemma** *neighborhood-root*: *t.neighborhood r* = *root-stree* ‘ *fset ts*
⟨*proof*⟩

**lemma** *V′*: *V′* = *nodes-stree t* − {*r*}
⟨*proof*⟩

**lemma** *E′*: *E′* = ⋃ (*tree-graph-edges* ‘ *fset ts*)
⟨*proof*⟩

**lemma** *subtrees-not-connected*:
**assumes** *s-in-ts*: $s \in fset\ ts$
**and** *e*: $\{u, v\} \in E'$
**and** *u-in-s*: $u \in nodes\text{-}stree\ s$
**shows** $v \in nodes\text{-}stree\ s$
⟨*proof*⟩

**lemma** *subtree-connected-components*:
**assumes** *s-in-ts*: $s \in fset\ ts$
**shows** $nodes\text{-}stree\ s \in g'.connected\text{-}components$
⟨*proof*⟩

**lemma** *connected-components-subtrees*: $g'.connected\text{-}components = nodes\text{-}stree\ `\ fset\ ts$
⟨*proof*⟩

**lemma** *induced-edges-subtree*:
**assumes** *s-in-ts*: $s \in fset\ ts$
**shows** $graph\text{-}system.induced\text{-}edges\ E'\ (nodes\text{-}stree\ s) = tree\text{-}graph\text{-}edges\ s$
⟨*proof*⟩

**lemma** *root-subtree*:
**assumes** *s-in-ts*: $s \in fset\ ts$
**shows** $(THE\ r'.\ r' \in (nodes\text{-}stree\ s) \wedge t.vert\text{-}adj\ r\ r') = root\text{-}stree\ s$
⟨*proof*⟩

**lemma** *subtrees-tree-subtrees*: $t.subtrees = tree\text{-}graph\text{-}stree\ `\ fset\ ts$
⟨*proof*⟩

**end**

**lemma** *stree-of-graph-tree-graph-stree*[*simp*]: $distinct\text{-}stree\text{-}nodes\ t \Longrightarrow stree\text{-}of\text{-}graph\ (tree\text{-}graph\text{-}stree\ t) = t$
⟨*proof*⟩

**lemma** *distinct-nodes-relabel*: $distinct\text{-}stree\text{-}nodes\ t \Longrightarrow inj\text{-}on\ f\ (nodes\text{-}stree\ t) \Longrightarrow distinct\text{-}stree\text{-}nodes\ (relabel\text{-}stree\ f\ t)$
⟨*proof*⟩

**lemma** *relabel-stree-app-rgraph-isomorphism*:
**assumes** *distinct-stree-nodes t*
**and** $inj\text{-}on\ f\ (nodes\text{-}stree\ t)$
**shows** $relabel\text{-}stree\ f\ t = stree\text{-}of\text{-}graph\ (app\text{-}rgraph\text{-}isomorphism\ f\ (tree\text{-}graph\text{-}stree\ t))$
⟨*proof*⟩

**lemma** (**in** *rgraph-isomorphism*) *app-rgraph-isomorphism-G*: $app\text{-}rgraph\text{-}isomorphism\ f\ (V_G,E_G,r_G) = (V_H,E_H,r_H)$
⟨*proof*⟩

**lemma** *tree-graphs-iso-strees-iso*:
**assumes** *tree-graph-stree t1* $\simeq_r$ *tree-graph-stree t2*
**and** *distinct-t1*: *distinct-stree-nodes t1*
**and** *distinct-t2*: *distinct-stree-nodes t2*
**shows** $\exists f.$ *inj-on f* (*nodes-stree t1*) $\wedge$ *relabel-stree f t1* = *t2*
⟨*proof*⟩

Skip the ltree representation as it introduces complications with the proofs

**fun** *tree-stree* :: *'a stree* ⇒ *tree* **where**
*tree-stree* (*SNode r ts*) = *Node* (*sorted-list-of-multiset* (*image-mset tree-stree* (*mset-set* (*fset ts*))))

**fun** *postorder-label-stree-aux* :: *nat* ⇒ *tree* ⇒ *nat* × *nat stree* **where**
*postorder-label-stree-aux n* (*Node* []) = (*n*, *SNode n* {||})
| *postorder-label-stree-aux n* (*Node* (*t*#*ts*)) =
(*let* (*n'*, *t'*) = *postorder-label-stree-aux n t in*
*case postorder-label-stree-aux* (*Suc n'*) (*Node ts*) *of*
(*n''*, *SNode r ts'*) ⇒ (*n''*, *SNode r* (*finsert t' ts'*)))

**definition** *postorder-label-stree* :: *tree* ⇒ *nat stree* **where**
*postorder-label-stree t* = *snd* (*postorder-label-stree-aux 0 t*)

**lemma** *fst-postorder-label-stree-aux-eq*: *fst* (*postorder-label-stree-aux n t*) = *fst* (*postorder-label-aux n t*)
⟨*proof*⟩

**lemma** *postorder-label-stree-aux-eq*: *snd* (*postorder-label-stree-aux n t*) = *stree-ltree* (*snd* (*postorder-label-aux n t*))
⟨*proof*⟩

**lemma** *postorder-label-stree-eq*: *postorder-label-stree t* = *stree-ltree* (*postorder-label t*)
⟨*proof*⟩

**lemma** *postorder-label-stree-aux-mono*: *fst* (*postorder-label-stree-aux n t*) ≥ *n*
⟨*proof*⟩

**lemma** *nodes-postorder-label-stree-aux-ge*: *postorder-label-stree-aux n t* = (*n'*, *t'*) ⟹ *v* ∈ *nodes-stree t'* ⟹ *v* ≥ *n*
⟨*proof*⟩

**lemma** *nodes-postorder-label-stree-aux-le*: *postorder-label-stree-aux n t* = (*n'*, *t'*) ⟹ *v* ∈ *nodes-stree t'* ⟹ *v* ≤ *n'*
⟨*proof*⟩

**lemma** *distinct-nodes-postorder-label-stree-aux*: *distinct-stree-nodes* (*snd* (*postorder-label-stree-aux n t*))
⟨*proof*⟩

**lemma** *distinct-nodes-postorder-label-stree*: *distinct-stree-nodes* (*postorder-label-stree t*)
⟨*proof*⟩

**lemma** *tree-stree-postorder-label-stree-aux*: *regular t* ⟹ *tree-stree* (*snd* (*postorder-label-stree-aux n t*)) = *t*
⟨*proof*⟩

**lemma** *tree-ltree-postorder-label-stree*[*simp*]: *regular t* ⟹ *tree-stree* (*postorder-label-stree t*) = *t*
⟨*proof*⟩

**lemma** *inj-relabel-subtrees*:
**assumes** *distinct-nodes*: *distinct-stree-nodes* (*SNode r ts*)
**and** *inj-on-nodes*: *inj-on f* (*nodes-stree* (*SNode r ts*))
**shows** *inj-on* (*relabel-stree f*) (*fset ts*)
⟨*proof*⟩

**lemma** *inj-on-subtree*: *inj-on f* (*nodes-stree* (*SNode r ts*)) ⟹ *t* ∈ *fset ts* ⟹ *inj-on f* (*nodes-stree t*)
⟨*proof*⟩

**lemma** *tree-stree-relabel-stree*: *distinct-stree-nodes t* ⟹ *inj-on f* (*nodes-stree t*) ⟹ *tree-stree* (*relabel-stree f t*) = *tree-stree t*
⟨*proof*⟩

**lemma** *tree-ltree-relabel-ltree-postorder-label-stree*: *regular t* ⟹ *inj-on f* (*nodes-stree* (*postorder-label-stree t*)) ⟹ *tree-stree* (*relabel-stree f* (*postorder-label-stree t*)) = *t*
⟨*proof*⟩

**lemma** *postorder-label-stree-inj*: *regular t1* ⟹ *regular t2* ⟹ *inj-on f* (*nodes-stree* (*postorder-label-stree t1*)) ⟹ *relabel-stree f* (*postorder-label-stree t1*) = *postorder-label-stree t2* ⟹ *t1* = *t2*
⟨*proof*⟩

**lemma** *tree-graph-inj-iso*: *regular t1* ⟹ *regular t2* ⟹ *tree-graph t1* ≃$_r$ *tree-graph t2* ⟹ *t1* = *t2*
⟨*proof*⟩

**lemma** *tree-graph-inj*:
**assumes** *regular-t1*: *regular t1*
**and** *regular-t2*: *regular t2*
**and** *tree-graph-eq*: *tree-graph t1* = *tree-graph t2*
**shows** *t1* = *t2*
⟨*proof*⟩

**end**

# 4 Enumeration of Rooted Trees

**theory** *Rooted-Tree-Enumeration*
**imports** *Rooted-Tree*
**begin**

Algorithm inspired by works of Beyer and Hedetniemi [1], performing the same operations but directly on a recursive tree data structure instead of level sequences.

**definition** *n-rtree-graphs* :: *nat* ⇒ *nat rpregraph set* **where**
*n-rtree-graphs n* = {(*V*,*E*,*r*). *rtree V E r* ∧ *card V* = *n*}

Recursive definition on the tree structure without using level sequences

**fun** *trim-tree* :: *nat* ⇒ *tree* ⇒ *nat* × *tree* **where**
*trim-tree 0 t* = (*0*, *t*)
| *trim-tree* (*Suc 0*) *t* = (*0*, *Node* [])
| *trim-tree* (*Suc n*) (*Node* []) = (*n*, *Node* [])
| *trim-tree n* (*Node* (*t*#*ts*)) =
(*case trim-tree n* (*Node ts*) *of*
(*0*, *t'*) ⇒ (*0*, *t'*) |
(*n1*, *Node ts'*) ⇒
*let* (*n2*, *t'*) = *trim-tree n1 t*
*in* (*n2*, *Node* (*t'*#*ts'*)))

**lemma** *fst-trim-tree-lt*[*termination-simp*]: *n* ≠ *0* ⟹ *fst* (*trim-tree n t*) < *n*
⟨*proof*⟩

**fun** *fill-tree* :: *nat* ⇒ *tree* ⇒ *tree list* **where**
*fill-tree 0* - = []
| *fill-tree n t* =
(*let* (*n'*, *t'*) = *trim-tree n t*
*in fill-tree n' t'* @ [*t'*])

**fun** *next-tree-aux* :: *nat* ⇒ *tree* ⇒ *tree option* **where**
*next-tree-aux n* (*Node* []) = *None*
| *next-tree-aux n* (*Node* (*Node* [] # *ts*)) = *next-tree-aux* (*Suc n*) (*Node ts*)
| *next-tree-aux n* (*Node* (*Node* (*Node* [] # *rs*) # *ts*)) = *Some* (*Node* (*fill-tree* (*Suc n*) (*Node rs*) @ (*Node rs*) # *ts*))
| *next-tree-aux n* (*Node* (*t* # *ts*)) = *Some* (*Node* (*the* (*next-tree-aux n t*) # *ts*))

**fun** *next-tree* :: *tree* ⇒ *tree option* **where**
*next-tree t* = *next-tree-aux 0 t*

**lemma** *next-tree-aux-None-iff*: *next-tree-aux n t* = *None* ⟷ *height t* < *2*
⟨*proof*⟩

**lemma** *next-tree-Some-iff*: (∃ *t'*. *next-tree t* = *Some t'*) ⟷ *height t* ≥ *2*
⟨*proof*⟩

## 4.1 Enumeration is monotonically decreasing

**lemma** *trim-id*: *trim-tree n t* = (*Suc n'*, *t'*) ⟹ *t* = *t'*
⟨*proof*⟩

**lemma** *trim-tree-le*: (*n'*, *t'*) = *trim-tree n t* ⟹ *t'* ≤ *t*
⟨*proof*⟩

**lemma** *fill-tree-le*: *r* ∈ *set* (*fill-tree n t*) ⟹ *r* ≤ *t*
⟨*proof*⟩

**lemma** *next-tree-aux-lt*: *height t* ≥ *2* ⟹ *the* (*next-tree-aux n t*) < *t*
⟨*proof*⟩

**lemma** *next-tree-lt*: *height t* ≥ *2* ⟹ *the* (*next-tree t*) < *t*
⟨*proof*⟩

**lemma** *next-tree-lt'*: *next-tree t* = *Some t'* ⟹ *t'* < *t*
⟨*proof*⟩

## 4.2 Size preservation

**lemma** *size-trim-tree*: *n* ≠ *0* ⟹ *trim-tree n t* = (*n'*, *t'*) ⟹ *n'* + *tree-size t'* = *n*
⟨*proof*⟩

**lemma** *size-fill-tree*: *sum-list* (*map tree-size* (*fill-tree n t*)) = *n*
⟨*proof*⟩

**lemma** *size-next-tree-aux*: *height t* ≥ *2* ⟹ *tree-size* (*the* (*next-tree-aux n t*)) = *tree-size t* + *n*
⟨*proof*⟩

**lemma** *size-next-tree*: *height t* ≥ *2* ⟹ *tree-size* (*the* (*next-tree t*)) = *tree-size t*
⟨*proof*⟩

**lemma** *size-next-tree'*: *next-tree t* = *Some t'* ⟹ *tree-size t'* = *tree-size t*
⟨*proof*⟩

## 4.3 Setup for termination proof

**definition** *lt-n-trees n* ≡ {*t*. *tree-size t* ≤ *n*}

**lemma** *n-trees-eq*: *n-trees n* = *Node* ‘ {*ts*. *tree-size* (*Node ts*) = *n*}
⟨*proof*⟩

**lemma** *lt-n-trees-eq*: *lt-n-trees* (*Suc n*) = *Node* ‘ {*ts*. *tree-size* (*Node ts*) ≤ *Suc n*}
⟨*proof*⟩

**lemma** *finite-lt-n-trees*: *finite* (*lt-n-trees n*)
⟨*proof*⟩

**lemma** *n-trees-subset-lt-n-trees*: *n-trees n* $\subseteq$ *lt-n-trees n*
⟨*proof*⟩

**lemma** *finite-n-trees*: *finite* (*n-trees n*)
⟨*proof*⟩

## 4.4 Algorithms for enumeration

**fun** *greatest-tree* :: *nat* ⇒ *tree* **where**
*greatest-tree* (*Suc 0*) = *Node* []
| *greatest-tree* (*Suc n*) = *Node* [*greatest-tree n*]

**function** *n-tree-enum-aux* :: *tree* ⇒ *tree list* **where**
*n-tree-enum-aux t* =
(*case next-tree t of None* ⇒ [*t*] | *Some t′* ⇒ *t* # *n-tree-enum-aux t′*)
⟨*proof*⟩

**fun** *n-tree-enum* :: *nat* ⇒ *tree list* **where**
*n-tree-enum 0* = []
| *n-tree-enum n* = *n-tree-enum-aux* (*greatest-tree n*)

**termination** *n-tree-enum-aux*
⟨*proof*⟩

**definition** *n-rtree-graph-enum* :: *nat* ⇒ *nat rpregraph list* **where**
*n-rtree-graph-enum n* = *map tree-graph* (*n-tree-enum n*)

## 4.5 Regularity

**lemma** *regular-trim-tree*: *regular t* ⟹ *regular* (*snd* (*trim-tree n t*))
⟨*proof*⟩

**lemma** *regular-trim-tree′*: *regular t* ⟹ (*n′*, *t′*) = *trim-tree n t* ⟹ *regular t′*
⟨*proof*⟩

**lemma** *sorted-fill-tree*: *sorted* (*fill-tree n t*)
⟨*proof*⟩

**lemma** *regular-fill-tree*: *regular t* ⟹ *r* ∈ *set* (*fill-tree n t*) ⟹ *regular r*
⟨*proof*⟩

**lemma** *regular-next-tree-aux*: *regular t* ⟹ *height t* ≥ *2* ⟹ *regular* (*the* (*next-tree-aux n t*))
⟨*proof*⟩

**lemma** *regular-next-tree*: *regular t* ⟹ *height t* ≥ *2* ⟹ *regular* (*the* (*next-tree t*))
⟨*proof*⟩

**lemma** *regular-next-tree′*: *regular t* ⟹ *next-tree t* = *Some t′* ⟹ *regular t′*

⟨*proof*⟩

**lemma** *regular-n-tree-enum-aux*: *regular t* ⟹ *r* ∈ *set* (*n-tree-enum-aux t*) ⟹ *regular r*
⟨*proof*⟩

**lemma** *regular-n-tree-greatest-tree*: *n* ≠ *0* ⟹ *greatest-tree n* ∈ *regular-n-trees n*
⟨*proof*⟩

**lemma** *regular-n-tree-enum*: *t* ∈ *set* (*n-tree-enum n*) ⟹ *regular t*
⟨*proof*⟩

**lemma** *size-n-tree-enum-aux*: *n* ≠ *0* ⟹ *r* ∈ *set* (*n-tree-enum-aux t*) ⟹ *tree-size r* = *tree-size t*
⟨*proof*⟩

**lemma** *size-greatest-tree*[*simp*]: *n* ≠ *0* ⟹ *tree-size* (*greatest-tree n*) = *n*
⟨*proof*⟩

**lemma** *size-n-tree-enum*: *t* ∈ *set* (*n-tree-enum n*) ⟹ *tree-size t* = *n*
⟨*proof*⟩

## 4.6 Totality

**lemma** *set* (*n-tree-enum n*) ⊆ *regular-n-trees n*
⟨*proof*⟩

**lemma** *greatest-tree-lt-Suc*: *n* ≠ *0* ⟹ *greatest-tree n* < *greatest-tree* (*Suc n*)
⟨*proof*⟩

**lemma** *greatest-tree-ge*: *tree-size t* ≤ *n* ⟹ *t* ≤ *greatest-tree n*
⟨*proof*⟩

**fun** *least-tree* :: *nat* ⇒ *tree* **where**
*least-tree* (*Suc n*) = *Node* (*replicate n* (*Node* []))

**lemma** *regular-n-tree-least-tree*: *n* ≠ *0* ⟹ *least-tree n* ∈ *regular-n-trees n*
⟨*proof*⟩

**lemma** *height-lt-2-least-tree*: *t* ∈ *regular-n-trees n* ⟹ *height t* < *2* ⟹ *t* = *least-tree n*
⟨*proof*⟩

**lemma** *least-tree-le*: *n* ≠ *0* ⟹ *tree-size t* ≥ *n* ⟹ *least-tree n* ≤ *t*
⟨*proof*⟩

**lemma** *trim-id'*: *n* ≥ *tree-size t* ⟹ *trim-tree n t* = (*n'*, *t'*) ⟹ *t'* = *t*
⟨*proof*⟩

**lemma** *tree-ge-lt-suffix*: $Node\ ts \leq r \Longrightarrow r < Node\ (t\#ts) \Longrightarrow \exists\, ss.\ r = Node\ (ss\ @\ ts)$
⟨*proof*⟩

**lemma** *trim-tree-0-iff*: $fst\ (trim\text{-}tree\ n\ t) = 0 \longleftrightarrow n \leq tree\text{-}size\ t$
⟨*proof*⟩

**lemma** *trim-tree-greatest-le*: $tree\text{-}size\ r \leq n \Longrightarrow r \leq t \Longrightarrow r \leq snd\ (trim\text{-}tree\ n\ t)$
⟨*proof*⟩

**lemma** *fill-tree-next-smallest*: $tree\text{-}size\ (Node\ rs) \leq Suc\ n \Longrightarrow \forall\, r{\in}set\ rs.\ r \leq t \Longrightarrow Node\ rs \leq Node\ (fill\text{-}tree\ n\ t)$
⟨*proof*⟩

**fun** *fill-twos* :: $nat \Rightarrow tree \Rightarrow tree$ **where**
$fill\text{-}twos\ n\ (Node\ ts) = Node\ (replicate\ n\ (Node\ [])\ @\ ts)$

**lemma** *size-fill-twos*: $tree\text{-}size\ (fill\text{-}twos\ n\ t) = n + tree\text{-}size\ t$
⟨*proof*⟩

**lemma** *regular-fill-twos*: $regular\ t \Longrightarrow regular\ (fill\text{-}twos\ n\ t)$
⟨*proof*⟩

**lemma** *fill-twos-lt*: $n \neq 0 \Longrightarrow t < fill\text{-}twos\ n\ t$
⟨*proof*⟩

**lemma** *fill-twos-less*: $r < Node\ (t\#ts) \Longrightarrow t \neq Node\ [] \Longrightarrow fill\text{-}twos\ n\ r < Node\ (t\#ts)$
⟨*proof*⟩

**lemma** *next-tree-aux-successor*: $tree\text{-}size\ r = tree\text{-}size\ t + n \Longrightarrow regular\ r \Longrightarrow r < t \Longrightarrow height\ t \geq 2 \Longrightarrow r \leq the\ (next\text{-}tree\text{-}aux\ n\ t)$
⟨*proof*⟩

**lemma** *next-tree-successor*: $tree\text{-}size\ r = tree\text{-}size\ t \Longrightarrow regular\ r \Longrightarrow r < t \Longrightarrow next\text{-}tree\ t = Some\ t' \Longrightarrow r \leq t'$
⟨*proof*⟩

**lemma** *set-n-tree-enum-aux*: $t \in regular\text{-}n\text{-}trees\ n \Longrightarrow set\ (n\text{-}tree\text{-}enum\text{-}aux\ t) = \{r{\in}regular\text{-}n\text{-}trees\ n.\ r \leq t\}$
⟨*proof*⟩

**theorem** *set-n-tree-enum*: $set\ (n\text{-}tree\text{-}enum\ n) = regular\text{-}n\text{-}trees\ n$
⟨*proof*⟩

**theorem** *n-rtree-graph-enum-n-rtree-graphs*: $G \in set\ (n\text{-}rtree\text{-}graph\text{-}enum\ n) \Longrightarrow G \in n\text{-}rtree\text{-}graphs\ n$

⟨*proof*⟩

**theorem** *n-rtree-graph-enum-surj*:
**assumes** *n-rtree-graph*: $G \in$ *n-rtree-graphs* $n$
**shows** $\exists\, G' \in$ *set* (*n-rtree-graph-enum* $n$). $G' \simeq_r G$
⟨*proof*⟩

### 4.7 Distinctness

**lemma** *n-tree-enum-aux-le*: $r \in$ *set* (*n-tree-enum-aux* $t$) $\Longrightarrow r \leq t$
⟨*proof*⟩

**lemma** *sorted-n-tree-enum-aux*: *sorted-wrt* (>) (*n-tree-enum-aux* $t$)
⟨*proof*⟩

**lemma** *distinct-n-tree-enum-aux*: *distinct* (*n-tree-enum-aux* $t$)
⟨*proof*⟩

**theorem** *distinct-n-tree-enum*: *distinct* (*n-tree-enum* $n$)
⟨*proof*⟩

**theorem** *distinct-n-rtree-graph-enum*: *distinct* (*n-rtree-graph-enum* $n$)
⟨*proof*⟩

**theorem** *inj-iso-n-rtree-graph-enum*:
**assumes** *G-in-n-rtree-graph-enum*: $G \in$ *set* (*n-rtree-graph-enum* $n$)
**and** *H-in-n-rtree-graph-enum*: $H \in$ *set* (*n-rtree-graph-enum* $n$)
**and** $G \simeq_r H$
**shows** $G = H$
⟨*proof*⟩

**theorem** *ex1-iso-n-rtree-graph-enum*: $G \in$ *n-rtree-graphs* $n \Longrightarrow \exists!\, G' \in$ *set* (*n-rtree-graph-enum* $n$). $G' \simeq_r G$
⟨*proof*⟩

**end**

## References


[1] T. Beyer and S. M. Hedetniemi. Constant time generation of rooted trees. *SIAM Journal on Computing*, 9(4):706–712, 1980.