

Verified Enumeration of Trees

Nils Cremer

September 13, 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 | 2 |
| 1.1 | Miscellaneous | 2 |
| 1.2 | Degree | 3 |
| 1.3 | Walks | 4 |
| 1.4 | Paths | 4 |
| 1.5 | Cycles | 5 |
| 1.6 | Subgraphs | 5 |
| 1.7 | Connectivity | 6 |
| 1.8 | Connected components | 7 |
| 1.9 | Trees | 10 |
| 1.10 | Graph Isomorphism | 12 |
| 2 | Enumeration of Labeled Trees | 13 |
| 2.1 | Algorithm | 13 |
| 2.2 | Correctness | 14 |
| 2.3 | Totality | 15 |
| 2.4 | Distinction | 16 |

| | | |
|----------|---|-----------|
| 3 | Rooted Trees | 17 |
| 3.1 | Rooted Graphs | 22 |
| 3.2 | Rooted Graph Isomorphism | 22 |
| 3.3 | Conversion between unlabeled, ordered, rooted trees and tree graphs | 23 |
| 3.4 | Injectivity with respect to isomorphism | 31 |
| 4 | Enumeration of Rooted Trees | 35 |
| 4.1 | Enumeration is monotonically decreasing | 36 |
| 4.2 | Size preservation | 36 |
| 4.3 | Setup for termination proof | 36 |
| 4.4 | Algorithms for enumeration | 37 |
| 4.5 | Regularity | 37 |
| 4.6 | Totality | 38 |
| 4.7 | Distinctness | 40 |

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*: $V_1 \subseteq V_2 \implies \text{induced-edges } V_1 \subseteq \text{induced-edges } V_2$
 ⟨proof⟩

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

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

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

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

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

1.2 Degree

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

lemma (in *fin-ulgraph*) *handshaking*: $(\sum v \in V. \text{degree } v) = 2 * \text{card } E$
 ⟨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*: $\neg \text{vert-adj } u v$
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*: $is\text{-walk } p \implies set (walk\text{-edges } p) \subseteq induced\text{-edges } (set\ p)$
(*proof*)

lemma (in *ulgraph*) *walk-edges-in-verts*: $e \in set (walk\text{-edges } xs) \implies e \subseteq set\ xs$
(*proof*)

lemma (in *ulgraph*) *is-walk-prefix*: $is\text{-walk } (xs@ys) \implies xs \neq [] \implies is\text{-walk } xs$
(*proof*)

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

1.4 Paths

lemma (in *ulgraph*) *is-gen-path-wf*: $is\text{-gen-path } p \implies set\ p \subseteq V$
(*proof*)

lemma (in *ulgraph*) *path-wf*: $is\text{-path } p \implies set\ p \subseteq V$
(*proof*)

lemma (in *fin-ulgraph*) *length-gen-path-card-V*: $is\text{-gen-path } p \implies walk\text{-length } p \leq card\ V$
(*proof*)

lemma (in *fin-ulgraph*) *length-path-card-V*: $is\text{-path } p \implies length\ p \leq card\ V$
(*proof*)

lemma (in *ulgraph*) *is-gen-path-prefix*: $is\text{-gen-path } (xs@ys) \implies xs \neq [] \implies is\text{-gen-path } (xs)$
(*proof*)

lemma (in *ulgraph*) *connecting-path-append*: $connecting\text{-path } u\ w\ (xs@ys) \implies xs \neq [] \implies connecting\text{-path } u\ (last\ xs)\ xs$
(*proof*)

lemma (in *ulgraph*) *connecting-path-tl*: $connecting\text{-path } u\ v\ (u \# w \# xs) \implies connecting\text{-path } w\ v\ (w \# xs)$
(*proof*)

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

1.5 Cycles

context *ulgraph*
begin

definition *is-cycle2* :: 'a list \Rightarrow bool **where**
is-cycle2 *xs* \longleftrightarrow *is-cycle* *xs* \wedge *distinct* (*walk-edges* *xs*)

lemma *loop-is-cycle2*: $\{v\} \in E \Longrightarrow$ *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* \Longrightarrow *walk-length* *c* \leq *Suc*
(*card* *V*)
<proof>

lemma (**in** *ulgraph*) *is-cycle-connecting-path*: *is-cycle* (*u#v#xs*) \Longrightarrow *connecting-path*
v u (v#xs)
<proof>

lemma (**in** *ulgraph*) *cycle-edges-notin-tl*: *is-cycle2* (*u#v#xs*) \Longrightarrow $\{u,v\} \notin$ *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* \Longrightarrow *G.is-walk* *xs*
<proof>

lemma *is-closed-walk*: *H.is-closed-walk* *xs* \Longrightarrow *G.is-closed-walk* *xs*
<proof>

lemma *is-gen-path*: *H.is-gen-path* *p* \Longrightarrow *G.is-gen-path* *p*
<proof>

lemma *connecting-path*: *H.connecting-path* *u v p* \Longrightarrow *G.connecting-path* *u v p*
<proof>

lemma *is-cycle*: $H.is-cycle\ c \implies G.is-cycle\ c$
<proof>

lemma *is-cycle2*: $H.is-cycle2\ c \implies G.is-cycle2\ c$
<proof>

lemma *vert-connected*: $H.vert-connected\ u\ v \implies G.vert-connected\ u\ v$
<proof>

lemma *is-connected-set*: $H.is-connected-set\ V' \implies G.is-connected-set\ V'$
<proof>

end

lemma (*in graph-system*) *subgraph-remove-vertex*: $remove-vertex\ v = (V', E') \implies$
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 $\neg\ vert-connected\ v\ u$
<proof>

lemma (*in fin-connected-ulgraph*) *degree-not-0*:
assumes *card* $V \geq 2$
and *inV*: $v \in V$
shows *degree\ v* $\neq 0$
<proof>

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

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

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

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

lemma (in *connected-ulgraph*) *connected-remove-leaf*:
assumes *degree*: *degree* $l = 1$
and *remove-vertex*: *remove-vertex* $l = (V', E')$
shows $\text{ulgraph.is-connected-set } V' E' V'$
 ⟨*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* $\equiv \{(u,v). \text{vert-connected } u v\}$

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

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

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

lemma *vert-connected-rel-refl*: *refl-on* $V \text{ vert-connected-rel}$
 ⟨*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*: $V' \in \text{connected-components} \implies V' \neq \{\}$
 ⟨*proof*⟩

lemma *connected-component-connected*: $V' \in \text{connected-components} \implies \text{is-connected-set } V'$
 ⟨*proof*⟩

lemma *connected-component-wf*: $V' \in \text{connected-components} \implies V' \subseteq V$
 ⟨*proof*⟩

lemma *connected-component-of-self*: $v \in V \implies v \in \text{connected-component-of } v$
 ⟨*proof*⟩

lemma *conn-comp-of-conn-comps*: $v \in V \implies \text{connected-component-of } v \in \text{connected-components}$
 ⟨*proof*⟩

lemma *Un-connected-components*: $\text{connected-components} = \text{connected-component-of } 'V$
 ⟨*proof*⟩

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

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

lemma *vert-connected-connected-component*: $C \in \text{connected-components} \implies u \in C \implies \text{vert-connected } u v \implies v \in C$
 ⟨*proof*⟩

lemma *connected-components-connected-ulgraphs*:
assumes *conn-comp*: $V' \in \text{connected-components}$
shows *connected-ulgraph* $V' (\text{induced-edges } V')$
 ⟨*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 \longleftrightarrow 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
 \implies 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 $\bigwedge u v. \{u,v\} \in E \implies u \in C \implies v \in C$
 shows *C ∈ connected-components*

<proof>

lemma (*in ulgraph*) *subset-conn-comps-if-Union*:
 assumes *A-subset-conn-comps*: $A \subseteq \text{connected-components}$
 and *Un-A*: $\bigcup A = V$
 shows $A = \text{connected-components}$
<proof>

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

1.9 Trees

locale *tree* = *fin-connected-ulgraph* +
 assumes *no-cycles*: $\neg \text{is-cycle2 } c$
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*: $\exists T. \text{spanning-tree } V E T$
<proof>

context *tree*
begin

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

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

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

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*: $leaf\ v \implies v \in V$
<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 singleton insert, induct set: tree*]:
 assumes *tree*: *tree* $V E$
 and *trivial*: $\bigwedge v. tree\ \{v\}\ \{\}\implies P\ \{v\}\ \{\}$
 and *insert*: $\bigwedge l\ v\ V\ E. tree\ V\ E \implies P\ V\ E \implies l \notin V \implies v \in V \implies \{l, v\} \notin E \implies tree.leaf\ (insert\ \{l, v\}\ E)\ l \implies P\ (insert\ l\ V)\ (insert\ \{l, v\}\ E)$
 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 $\exists! u \in C. \text{vert-adj } v \ u$
 ⟨*proof*⟩

lemma *non-trivial-card-E*: *non-trivial* $\implies \text{card } E \geq 1$
 ⟨*proof*⟩

lemma *V-Union-E*: *non-trivial* $\implies V = \bigcup E$
 ⟨*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*: $((\cdot) f) \cdot E_G = E_H$
begin

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

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

definition *inv-iso* \equiv *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** \simeq 50) **where**
 $(V_G, E_G) \simeq (V_H, E_H) \longleftrightarrow (\exists f. \text{graph-isomorphism } V_G E_G V_H E_H f)$

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

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

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

lemma *graph-isomorphism-trans*: *graph-isomorphism* $V_G E_G V_H E_H f \implies \text{graph-isomorphism}$
 $V_H E_H V_F E_F g \implies \text{graph-isomorphism } V_G E_G V_F E_F (g \circ f)$
 ⟨*proof*⟩

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

end

2 Enumeration of Labeled Trees

theory *Labeled-Tree-Enumeration*

imports *Tree-Graph*

begin

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

2.1 Algorithm

Prüfer sequence to tree

definition *prufer-sequences* :: 'a *list* ⇒ 'a *list set* **where**
 $\text{prufer-sequences } \text{verts} = \{xs. \text{length } xs = \text{length } \text{verts} - 2 \wedge \text{set } xs \subseteq \text{set } \text{verts}\}$

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

Some $a \Rightarrow \text{insert } \{a,b\} (\text{tree-edges-of-prufer-seq } (\text{remove1 } a \text{ verts}) \text{ seq}))$

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

definition *labeled-tree-enum* :: 'a list \Rightarrow '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 \Rightarrow 'a edge set \Rightarrow 'a edge set **where**
remove-vertex-edges *v* *E* = { $e \in E. \neg \text{graph-system.vincident } v \ e$ }

lemma *find-in-list[termination-simp]*: find *P* *verts* = Some *v* $\Longrightarrow v \in \text{set } \text{verts}$
 <proof>

lemma [*termination-simp*]: find *P* *verts* = Some *v* $\Longrightarrow \text{length } \text{verts} - \text{Suc } 0 < \text{length } \text{verts}$
 <proof>

fun *prufer-seq-of-tree* :: 'a list \Rightarrow 'a edge set \Rightarrow 'a list **where**
prufer-seq-of-tree *verts* *E* =
 (if length *verts* \leq 2 then []
 else (case find (tree.leaf *E*) *verts* of
 Some leaf \Rightarrow (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* \geq 2
and *distinct-verts*: distinct *verts*

locale *tree-of-prufer-seq-ctx* = *valid-verts* +
fixes *seq*
assumes *prufer-seq*: *seq* \in *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 $\exists x. \text{find } (\lambda x. x \notin \text{set } \text{ys}) \text{ } xs = \text{Some } x$
 <proof>

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

and $\bigwedge \text{verts } b \text{ seq } a.$
 $\text{find } (\lambda x. x \notin \text{set } (b \# \text{seq})) \text{ verts} = \text{Some } a$
 $\implies a \in \text{set } \text{verts} \implies a \notin \text{set } (b \# \text{seq}) \implies \text{seq} \in \text{prufer-sequences}$
 $(\text{remove1 } a \text{ verts})$
 $\implies \text{tree-of-prufer-seq-ctx } (\text{remove1 } a \text{ verts}) \text{ seq} \implies P (\text{remove1 } a \text{ verts})$
 $\text{seq} \implies P \text{ verts } (b \# \text{seq})$
shows $P \text{ verts seq}$
 $\langle \text{proof} \rangle$

lemma (in *tree-of-prufer-seq-ctx*) *tree-edges-of-prufer-seq-tree*:
shows $\text{tree } (\text{set } \text{verts}) (\text{tree-edges-of-prufer-seq } \text{verts } \text{seq})$
 $\langle \text{proof} \rangle$

lemma (in *tree-of-prufer-seq-ctx*) *tree-of-prufer-seq-tree*: $(V, E) = \text{tree-of-prufer-seq}$
 $\text{verts } \text{seq} \implies \text{tree } V E$
 $\langle \text{proof} \rangle$

lemma (in *valid-verts*) *labeled-tree-enum-trees*:
assumes *VE-in-labeled-tree-enum*: $(V, E) \in \text{set } (\text{labeled-tree-enum } \text{verts})$
shows $\text{tree } V E$
 $\langle \text{proof} \rangle$

2.3 Totality

locale *prufer-seq-of-tree-context* =
 $\text{valid-verts } \text{verts} + \text{tree } \text{set } \text{verts } E \text{ for } \text{verts } E$
begin

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

lemma *prufer-seq-of-tree-wf*: $\text{set } (\text{prufer-seq-of-tree } \text{verts } E) \subseteq \text{set } \text{verts}$
 $\langle \text{proof} \rangle$

lemma *length-prufer-seq-of-tree*: $\text{length } (\text{prufer-seq-of-tree } \text{verts } E) = \text{length } \text{verts}$
 $- 2$
 $\langle \text{proof} \rangle$

lemma *prufer-seq-of-tree-prufer-seq*: $\text{prufer-seq-of-tree } \text{verts } E \in \text{prufer-sequences}$
 verts
 $\langle \text{proof} \rangle$

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

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

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

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

end

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

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

2.4 Distinction

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

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

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

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

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

end

3 Rooted Trees

theory *Rooted-Tree*

imports *Tree-Graph HOL-Library.FSet*

begin

datatype *tree* = *Node tree list*

fun *tree-size* :: *tree* \Rightarrow *nat* **where**

tree-size (*Node ts*) = *Suc* ($\sum t \leftarrow ts.$ *tree-size* *t*)

fun *height* :: *tree* \Rightarrow *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*]:

$(t = \text{Node } [] \implies P) \implies (\bigwedge r ts. t = \text{Node } (r \# ts) \implies P) \implies P$
<proof>

lemma *tree-rev-exhaust*[*case-names Nil Snoc*]:

$(t = \text{Node } [] \implies P) \implies (\bigwedge ts r. t = \text{Node } (ts @ [r]) \implies P) \implies P$
<proof>

lemma *tree-cons-induct*[*case-names Nil Cons*]:

assumes *P* (*Node []*)

and $\bigwedge t ts. P t \implies P (\text{Node } ts) \implies P (\text{Node } (t \# ts))$

shows *P t*

<proof>

fun *lexord-tree* **where**

lexord-tree *t* (*Node []*) \longleftrightarrow *False*

| *lexord-tree* (*Node []*) *r* \longleftrightarrow *True*

| *lexord-tree* (*Node (t#ts)*) (*Node (r#rs)*) \longleftrightarrow *lexord-tree* *t r* \vee (*t = r* \wedge *lexord-tree* (*Node ts*) (*Node rs*))

fun *mirror* :: *tree* \Rightarrow *tree* **where**

mirror (*Node ts*) = *Node* (*map mirror* (*rev ts*))

instantiation *tree* :: *linorder*

begin

definition

tree-less-def: $(t :: \text{tree}) < r \longleftrightarrow \text{lexord-tree } (\text{mirror } t) (\text{mirror } r)$

definition

tree-le-def: $(t :: \text{tree}) \leq r \longleftrightarrow t < r \vee t = r$

lemma *lexord-tree-empty2[simp]*: $\text{lexord-tree } (\text{Node } []) \ r \longleftrightarrow r \neq \text{Node } []$
(proof)

lemma *mirror-empty[simp]*: $\text{mirror } t = \text{Node } [] \longleftrightarrow t = \text{Node } []$
(proof)

lemma *mirror-not-empty[simp]*: $\text{mirror } t \neq \text{Node } [] \longleftrightarrow t \neq \text{Node } []$
(proof)

lemma *tree-le-empty[simp]*: $\text{Node } [] \leq t$
(proof)

lemma *tree-less-empty-iff*: $\text{Node } [] < t \longleftrightarrow t \neq \text{Node } []$
(proof)

lemma *not-tree-less-empty[simp]*: $\neg t < \text{Node } []$
(proof)

lemma *tree-le-empty2-iff[simp]*: $t \leq \text{Node } [] \longleftrightarrow t = \text{Node } []$
(proof)

lemma *lexord-tree-antisym*: $\text{lexord-tree } t \ r \implies \neg \text{lexord-tree } r \ t$
(proof)

lemma *tree-less-antisym*: $(t::\text{tree}) < r \implies \neg r < t$
(proof)

lemma *lexord-tree-not-eq*: $\text{lexord-tree } t \ r \implies t \neq r$
(proof)

lemma *tree-less-not-eq*: $(t::\text{tree}) < r \implies t \neq r$
(proof)

lemma *lexord-tree-irrefl*: $\neg \text{lexord-tree } t \ t$
(proof)

lemma *tree-less-irrefl*: $\neg (t::\text{tree}) < t$
(proof)

lemma *lexord-tree-eq-iff*: $\neg \text{lexord-tree } t \ r \wedge \neg \text{lexord-tree } r \ t \longleftrightarrow t = r$
(proof)

lemma *mirror-mirror*: $\text{mirror } (\text{mirror } t) = t$
(proof)

lemma *mirror-inj*: $\text{mirror } t = \text{mirror } r \implies t = r$
(proof)

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

lemma *lexord-tree-trans*: $lexord-tree\ t\ r \implies lexord-tree\ r\ s \implies lexord-tree\ t\ s$
<proof>

instance
<proof>

end

lemma *tree-size-children*: $tree-size\ (Node\ ts) = Suc\ n \implies t \in set\ ts \implies tree-size\ t \leq n$
<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*: $tree-size\ (Node\ ts) = Suc\ n \implies length\ ts \leq n$
<proof>

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

lemma *height-0-iff*: $height\ t = 0 \implies t = Node\ []$
<proof>

lemma *height-children*: $height\ (Node\ ts) = Suc\ n \implies t \in set\ ts \implies height\ t \leq n$
<proof>

lemma *height-children-le-height*: $\forall t \in set\ ts. height\ t \leq n \implies height\ (Node\ ts) \leq Suc\ n$
<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*: $\text{lexord-tree } (\text{Node } ts) (\text{Node } (ts@[t]))$
 ⟨proof⟩

lemma *tree-less-cons*: $\text{Node } ts < \text{Node } (t\#ts)$
 ⟨proof⟩

lemma *tree-le-cons*: $\text{Node } ts \leq \text{Node } (t\#ts)$
 ⟨proof⟩

lemma *tree-less-cons'*: $t \leq \text{Node } rs \implies t < \text{Node } (r\#rs)$
 ⟨proof⟩

lemma *tree-less-snoc2-iff[simp]*: $\text{Node } (ts@[t]) < \text{Node } (rs@[r]) \longleftrightarrow t < r \vee (t = r \wedge \text{Node } ts < \text{Node } rs)$
 ⟨proof⟩

lemma *tree-le-snoc2-iff[simp]*: $\text{Node } (ts@[t]) \leq \text{Node } (rs@[r]) \longleftrightarrow t < r \vee (t = r \wedge \text{Node } ts \leq \text{Node } rs)$
 ⟨proof⟩

lemma *lexord-tree-cons2[simp]*: $\text{lexord-tree } (\text{Node } (ts@[t])) (\text{Node } (ts@[r])) \longleftrightarrow \text{lexord-tree } t r$
 ⟨proof⟩

lemma *tree-less-cons2[simp]*: $\text{Node } (t\#ts) < \text{Node } (r\#ts) \longleftrightarrow t < r$
 ⟨proof⟩

lemma *tree-le-cons2[simp]*: $\text{Node } (t\#ts) \leq \text{Node } (r\#ts) \longleftrightarrow t \leq r$
 ⟨proof⟩

lemma *tree-less-sorted-snoc*: $\text{sorted } (ts@[r]) \implies \text{Node } ts < \text{Node } (ts@[r])$
 ⟨proof⟩

lemma *lexord-tree-comm-prefix[simp]*: $\text{lexord-tree } (\text{Node } (ss@ts)) (\text{Node } (ss@rs)) \longleftrightarrow \text{lexord-tree } (\text{Node } ts) (\text{Node } rs)$
 ⟨proof⟩

lemma *less-tree-comm-suffix[simp]*: $\text{Node } (ts@ss) < \text{Node } (rs@ss) \longleftrightarrow \text{Node } ts < \text{Node } rs$
 ⟨proof⟩

lemma *tree-le-comm-suffix[simp]*: $\text{Node } (ts@ss) \leq \text{Node } (rs@ss) \longleftrightarrow \text{Node } ts \leq \text{Node } rs$
 ⟨proof⟩

lemma *tree-less-comm-suffix2*: $t < r \implies \text{Node } (ts@t\#ss) < \text{Node } (r\#ss)$
 ⟨proof⟩

lemma *lexord-tree-append[simp]*: $\text{lexord-tree } (\text{Node } ts) (\text{Node } (ts@rs)) \longleftrightarrow rs \neq []$
 ⟨proof⟩

lemma *tree-less-append[simp]*: $\text{Node } ts < \text{Node } (rs@ts) \longleftrightarrow rs \neq []$
 ⟨proof⟩

lemma *tree-le-append*: $\text{Node } ts \leq \text{Node } (ss@ts)$
 ⟨proof⟩

lemma *tree-less-singleton-iff[simp]*: $\text{Node } (ts@[t]) < \text{Node } [r] \longleftrightarrow t < r$
 ⟨proof⟩

lemma *tree-le-singleton-iff[simp]*: $\text{Node } (ts@[t]) \leq \text{Node } [r] \longleftrightarrow t < r \vee (t = r \wedge ts = [])$
 ⟨proof⟩

lemma *lexord-tree-nested*: $\text{lexord-tree } t (\text{Node } [t])$
 ⟨proof⟩

lemma *tree-less-nested*: $t < \text{Node } [t]$
 ⟨proof⟩

lemma *tree-le-nested*: $t \leq \text{Node } [t]$
 ⟨proof⟩

lemma *lexord-tree-iff*:
 $\text{lexord-tree } t r \longleftrightarrow (\exists ts t' ss rs r'. t = \text{Node } (ss @ t' \# ts) \wedge r = \text{Node } (ss @ r' \# rs) \wedge \text{lexord-tree } t' r') \vee (\exists ts rs. rs \neq [] \wedge t = \text{Node } ts \wedge r = \text{Node } (ts @ rs))$
 (is ?l \longleftrightarrow ?r)
 ⟨proof⟩

lemma *tree-less-iff*: $t < r \longleftrightarrow (\exists ts t' ss rs r'. t = \text{Node } (ts @ t' \# ss) \wedge r = \text{Node } (rs @ r' \# ss) \wedge t' < r') \vee (\exists ts rs. rs \neq [] \wedge t = \text{Node } ts \wedge r = \text{Node } (rs @ ts))$ (is ?l \longleftrightarrow ?r)
 ⟨proof⟩

lemma *tree-empty-cons-lt-le*: $r < \text{Node } (\text{Node } [] \# ts) \implies r \leq \text{Node } ts$
 ⟨proof⟩

fun *regular* :: $\text{tree} \Rightarrow \text{bool}$ **where**
 $\text{regular } (\text{Node } ts) \longleftrightarrow \text{sorted } ts \wedge (\forall t \in \text{set } ts. \text{regular } t)$

definition *n-trees* :: $\text{nat} \Rightarrow \text{tree set}$ **where**
 $n\text{-trees } n = \{t. \text{tree-size } t = n\}$

definition *regular-n-trees* :: $\text{nat} \Rightarrow \text{tree set}$ **where**
 $\text{regular-n-trees } n = \{t. \text{tree-size } t = n \wedge \text{regular } t\}$

3.1 Rooted Graphs

type-synonym $'a$ rpregraph = ($'a$ set) \times ($'a$ edge set) \times $'a$

locale rgraph = graph-system +
fixes r
assumes root-wf: $r \in V$

locale rtree = tree + rgraph
begin

definition subtrees :: $'a$ rpregraph set **where**

subtrees =
 (let (V', E') = remove-vertex r
 in ($\lambda C. (C, \text{graph-system.induced-edges } E' C, \text{THE } r'. r' \in C \wedge \text{vert-adj } r r')$)
 'ulgraph.connected-components $V' E'$)

lemma rtree-subtree:

assumes subtree: $(S, E_S, r_S) \in \text{subtrees}$
shows rtree $S E_S r_S$
 $\langle \text{proof} \rangle$

lemma finite-subtrees: finite subtrees

$\langle \text{proof} \rangle$

lemma remove-root-subtrees:

assumes remove-vertex: remove-vertex $r = (V', E')$
and conn-component: $C \in \text{ulgraph.connected-components } V' E'$
shows rtree $C (\text{graph-system.induced-edges } E' C) (\text{THE } r'. r' \in C \wedge \text{vert-adj } r r')$
 $\langle \text{proof} \rangle$

end

3.2 Rooted Graph Isomorphism

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

locale rgraph-isomorphism =

$G: \text{rgraph } V_G E_G r_G + \text{graph-isomorphism } V_G E_G V_H E_H f$ **for** $V_G E_G r_G$
 $V_H E_H r_H f +$
assumes root-preserving: $f r_G = r_H$

begin

interpretation $H: \text{graph-system } V_H E_H \langle \text{proof} \rangle$

lemma rgraph-H: rgraph $V_H E_H r_H$

$\langle \text{proof} \rangle$

interpretation H : $rgraph\ V_H\ E_H\ r_H\ \langle proof \rangle$

lemma $rgraph\text{-}isomorphism\text{-}inv$: $rgraph\text{-}isomorphism\ V_H\ E_H\ r_H\ V_G\ E_G\ r_G\ inv\text{-}iso\ \langle proof \rangle$

end

fun $rgraph\text{-}isomorph$:: $'a\ rpregraph \Rightarrow 'b\ rpregraph \Rightarrow bool$ (**infix** \simeq_r 50) **where**
 $(V_G, E_G, r_G) \simeq_r (V_H, E_H, r_H) \iff (\exists f. rgraph\text{-}isomorphism\ V_G\ E_G\ r_G\ V_H\ E_H\ r_H\ f)$

lemma (**in** $rgraph$) $rgraph\text{-}isomorphism\text{-}id$: $rgraph\text{-}isomorphism\ V\ E\ r\ V\ E\ r\ id\ \langle proof \rangle$

lemma (**in** $rgraph$) $rgraph\text{-}isomorph\text{-}refl$: $(V, E, r) \simeq_r (V, E, r)\ \langle proof \rangle$

lemma $rgraph\text{-}isomorph\text{-}sym$: $G \simeq_r H \implies H \simeq_r G\ \langle proof \rangle$

lemma $rgraph\text{-}isomorphism\text{-}trans$: $rgraph\text{-}isomorphism\ V_G\ E_G\ r_G\ V_H\ E_H\ r_H\ f \implies rgraph\text{-}isomorphism\ V_H\ E_H\ r_H\ V_F\ E_F\ r_F\ g \implies rgraph\text{-}isomorphism\ V_G\ E_G\ r_G\ V_F\ E_F\ r_F\ (g\ o\ f)\ \langle proof \rangle$

lemma $rgraph\text{-}isomorph\text{-}trans$: $transp\ (\simeq_r)\ \langle proof \rangle$

lemma (**in** $rtree$) $rgraph\text{-}isomorph\text{-}app\text{-}iso$: $inj\text{-}on\ f\ V \implies app\text{-}rgraph\text{-}isomorphism\ f\ (V, E, r) = (V', E', r') \implies rgraph\text{-}isomorphism\ V\ E\ r\ V'\ E'\ r'\ f\ \langle proof \rangle$

lemma (**in** $rtree$) $rgraph\text{-}isomorph\text{-}app\text{-}iso$: $inj\text{-}on\ f\ V \implies (V, E, r) \simeq_r app\text{-}rgraph\text{-}isomorphism\ f\ (V, E, r)\ \langle proof \rangle$

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

datatype $'a\ ltree = LNode\ 'a\ 'a\ ltree\ list$

fun $ltree\text{-}size$:: $'a\ ltree \Rightarrow nat$ **where**
 $ltree\text{-}size\ (LNode\ r\ ts) = Suc\ (\sum\ t \leftarrow ts. ltree\text{-}size\ t)$

fun $root\text{-}ltree$:: $'a\ ltree \Rightarrow 'a$ **where**
 $root\text{-}ltree\ (LNode\ r\ ts) = r$

fun $nodes\text{-}ltree$:: $'a\ ltree \Rightarrow 'a\ set$ **where**

$nodes\text{-}ltree (LNode\ r\ ts) = \{r\} \cup (\bigcup_{t \in set\ ts} nodes\text{-}ltree\ t)$

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

fun *distinct-ltree-nodes* :: 'a ltree \Rightarrow bool **where**
distinct-ltree-nodes (LNode a ts) \longleftrightarrow ($\forall t \in set\ ts. a \notin nodes\text{-}ltree\ t$) \wedge *distinct* ts
 \wedge *disjoint-family-on* nodes-ltree (set ts) \wedge ($\forall t \in set\ ts. distinct\text{-}ltree\text{-}nodes\ t$)

fun *postorder-label-aux* :: nat \Rightarrow tree \Rightarrow nat \times 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') \Rightarrow (n'', LNode r (t'#ts')))

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

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

fun *regular-ltree* :: 'a ltree \Rightarrow bool **where**
regular-ltree (LNode r ts) \longleftrightarrow *sorted-wrt* ($\lambda t\ s. tree\text{-}ltree\ t \leq tree\text{-}ltree\ s$) ts \wedge
($\forall t \in set\ ts. regular\text{-}ltree\ t$)

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

lemma *stree-size-child-lt*[*termination-simp*]: t \in ts $\implies size\ t < Suc\ (\sum_{s \in fset\ ts} size\ s)$
⟨*proof*⟩

lemma *stree-size-child-lt'*[*termination-simp*]: t $\in fset\ ts \implies size\ t < Suc\ (\sum_{s \in fset\ ts} size\ s)$
⟨*proof*⟩

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

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

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

fun *nodes-stree* :: 'a stree \Rightarrow 'a set **where**
nodes-stree (SNode a ts) = {a} \cup ($\bigcup_{t \in fset\ ts} nodes\text{-}stree\ t$)

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

$tree-graph-edges (SNode a ts) = ((\lambda t. \{a, root-stree t\}) \text{ ' fset } ts) \cup (\bigcup t \in \text{fset } ts. tree-graph-edges t)$

fun *distinct-stree-nodes* :: 'a stree \Rightarrow bool **where**

distinct-stree-nodes (SNode a ts) $\longleftrightarrow (\forall t \in \text{fset } ts. a \notin \text{nodes-stree } t) \wedge \text{dis-joint-family-on nodes-stree (fset } ts) \wedge (\forall t \in \text{fset } ts. \text{distinct-stree-nodes } t)$

fun *ltree-stree* :: 'a stree \Rightarrow 'a ltree **where**

ltree-stree (SNode r ts) = LNode r (SOME xs. *fset-of-list* xs = *ltree-stree* |¹ ts \wedge *distinct* xs \wedge *sorted-wrt* ($\lambda t s. \text{tree-ltree } t \leq \text{tree-ltree } s$) xs)

fun *stree-ltree* :: 'a ltree \Rightarrow 'a stree **where**

stree-ltree (LNode r ts) = SNode r (*fset-of-list* (map *stree-ltree* ts))

definition *tree-graph-stree* :: 'a stree \Rightarrow 'a rpregraph **where**

tree-graph-stree t = (nodes-stree t, tree-graph-edges t, root-stree t)

function *stree-of-graph* :: 'a rpregraph \Rightarrow 'a stree **where**

stree-of-graph (V,E,r) =
 (if $\neg \text{rtree } V E r$ then undefined else
 SNode r (Abs-fset (*stree-of-graph* ' rtree.subtrees V E r)))
 <proof>

termination

<proof>

definition *tree-graph* :: tree \Rightarrow nat rpregraph **where**

tree-graph t = *tree-graph-stree* (*stree-ltree* (*postorder-label* t))

fun *relabel-stree* :: ('a \Rightarrow 'b) \Rightarrow 'a stree \Rightarrow 'b stree **where**

relabel-stree f (SNode r ts) = SNode (f r) ((*relabel-stree* f) |¹ ts)

lemma *root-ltree-wf*: root-ltree t \in 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 \in nodes-stree t

<proof>

lemma *tree-graph-edges-wf*: e \in tree-graph-edges t \implies e \subseteq nodes-stree t

<proof>

lemma *card-tree-graph-edges-distinct*: $\text{distinct-stree-nodes } t \implies e \in \text{tree-graph-edges } t \implies \text{card } e = 2$

<proof>

lemma *nodes-stree-non-empty*: $\text{nodes-stree } t \neq \{\}$

<proof>

lemma *finite-nodes-stree*: $\text{finite } (\text{nodes-stree } t)$

<proof>

lemma *finite-tree-graph-edges*: $\text{finite } (\text{tree-graph-edges } t)$

<proof>

lemma *root-relabel-stree[simp]*: $\text{root-stree } (\text{relabel-stree } f \ t) = f \ (\text{root-stree } t)$

<proof>

lemma *nodes-stree-relabel-stree[simp]*: $\text{nodes-stree } (\text{relabel-stree } f \ t) = f \ \text{'nodes-stree } t$

<proof>

lemma *tree-graph-edges-relabel-stree[simp]*: $\text{tree-graph-edges } (\text{relabel-stree } f \ t) = ((\cdot) \ f) \ \text{'tree-graph-edges } t$

<proof>

lemma *nodes-stree-ltree[simp]*: $\text{nodes-stree } (\text{stree-ltree } t) = \text{nodes-ltree } t$

<proof>

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

<proof>

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

lemma *fset-of-list-ltree-stree-subtrees[simp]*: $\text{fset-of-list } (\text{ltree-stree-subtrees } ts) = \text{ltree-stree } |\cdot| \ ts$

<proof>

lemma *set-ltree-stree-subtrees[simp]*: $\text{set } (\text{ltree-stree-subtrees } ts) = \text{ltree-stree } \text{'fset } ts$

<proof>

lemma *distinct-ltree-stree-subtrees*: $\text{distinct } (\text{ltree-stree-subtrees } ts)$

<proof>

lemma *sorted-wrt-ltree-stree-subtrees*: $\text{sorted-wrt } (\lambda \ t \ s. \ \text{tree-ltree } t \leq \ \text{tree-ltree } s) \ (\text{ltree-stree-subtrees } ts)$

<proof>

lemma *nodes-ltree-stree[simp]*: $\text{nodes-ltree} (\text{ltree-stree } t) = \text{nodes-stree } t$
<proof>

lemma *stree-ltree-stree[simp]*: $\text{stree-ltree} (\text{ltree-stree } t) = t$
<proof>

lemma *nodes-tree-graph-stree*: $\text{tree-graph-stree } t = (V, E, r) \implies V = \text{nodes-stree } t$
<proof>

lemma *relabel-stree-stree-ltree*: $\text{relabel-stree } f (\text{stree-ltree } t) = \text{stree-ltree} (\text{relabel-ltree } f t)$
<proof>

lemma *relabel-stree-relabel-ltree*: $\text{relabel-ltree } f t1 = t2 \implies \text{relabel-stree } f (\text{stree-ltree } t1) = \text{stree-ltree } t2$
<proof>

lemma *app-rgraph-iso-tree-graph-stree*: $\text{app-rgraph-isomorphism } f (\text{tree-graph-stree } t) = \text{tree-graph-stree} (\text{relabel-stree } f t)$
<proof>

lemma (**in** *rtree*) *root-stree-of-graph[simp]*: $\text{root-stree} (\text{stree-of-graph } (V, E, r)) = r$
<proof>

lemma (**in** *rtree*) *nodes-stree-stree-of-graph[simp]*: $\text{nodes-stree} (\text{stree-of-graph } (V, E, r)) = V$
<proof>

lemma (**in** *rtree*) *tree-graph-edges-stree-of-graph[simp]*: $\text{tree-graph-edges} (\text{stree-of-graph } (V, E, r)) = E$
<proof>

lemma (**in** *rtree*) *tree-graph-stree-of-graph[simp]*: $\text{tree-graph-stree} (\text{stree-of-graph } (V, E, r)) = (V, E, r)$
<proof>

lemma *postorder-label-aux-mono*: $\text{fst} (\text{postorder-label-aux } n t) \geq n$
<proof>

lemma *nodes-postorder-label-aux-ge*: $\text{postorder-label-aux } n t = (n', t') \implies v \in \text{nodes-ltree } t' \implies v \geq n$
<proof>

lemma *nodes-postorder-label-aux-le*: $\text{postorder-label-aux } n t = (n', t') \implies v \in \text{nodes-ltree } t' \implies v \leq n'$

<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 \implies distinct-stree-nodes (stree-ltree t)*
<proof>

fun *distinct-edges :: 'a stree \implies bool where*
distinct-edges (SNode a ts) \longleftrightarrow inj-on ($\lambda t. \{a, \text{root-stree } t\}$) (fset ts)
 \wedge ($\forall t \in \text{fset } ts. \text{disjnt } ((\lambda t. \{a, \text{root-stree } t\}) ' \text{fset } ts) (\text{tree-graph-edges } t)$)
 \wedge disjoint-family-on tree-graph-edges (fset ts)
 \wedge ($\forall t \in \text{fset } ts. \text{distinct-edges } t$)

lemma *distinct-nodes-inj-on-root-stree: distinct-stree-nodes (SNode r ts) \implies 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 \implies card (nodes-stree t) = Suc (card (tree-graph-edges t))*
<proof>

lemma *tree-tree-graph-edges: distinct-stree-nodes t \implies 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 \implies tree-graph-stree t = (V,E,r) \implies rtree V E r*
<proof>

lemma *rtree-tree-graph: tree-graph t = (V,E,r) \implies rtree V E r*
<proof>

Cardinality of the resulting rooted tree is correct

lemma *ltree-size-postorder-label-aux*: $ltree\text{-}size\ (snd\ (postorder\text{-}label\text{-}aux\ n\ t)) = tree\text{-}size\ t$
 ⟨proof⟩

lemma *ltree-size-postorder-label*: $ltree\text{-}size\ (postorder\text{-}label\ t) = tree\text{-}size\ t$
 ⟨proof⟩

lemma *distinct-nodes-ltree-size-card-nodes*: $distinct\text{-}ltree\text{-}nodes\ t \implies ltree\text{-}size\ t = card\ (nodes\text{-}ltree\ t)$
 ⟨proof⟩

lemma *distinct-nodes-stree-size-card-nodes*: $distinct\text{-}stree\text{-}nodes\ t \implies stree\text{-}size\ t = card\ (nodes\text{-}stree\ t)$
 ⟨proof⟩

lemma *stree-size-stree-ltree*: $distinct\text{-}ltree\text{-}nodes\ t \implies stree\text{-}size\ (stree\text{-}ltree\ t) = ltree\text{-}size\ t$
 ⟨proof⟩

lemma *card-tree-graph-stree*: $distinct\text{-}stree\text{-}nodes\ t \implies tree\text{-}graph\text{-}stree\ t = (V, E, r) \implies card\ V = stree\text{-}size\ t$
 ⟨proof⟩

lemma *card-tree-graph*: $tree\text{-}graph\ t = (V, E, r) \implies card\ V = tree\text{-}size\ t$
 ⟨proof⟩

lemma *[termination-simp]*: $(t, s) \in set\ (zip\ ts\ ss) \implies size\ t < Suc\ (size\text{-}list\ size\ ts)$
 ⟨proof⟩

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

fun *postorder-relabel-aux* :: $nat \Rightarrow 'a\ ltree \Rightarrow nat \times (nat \rightarrow 'a)$ **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* :: $'a\ ltree \Rightarrow (nat \rightarrow 'a)$ **where**
postorder-relabel t = snd (*postorder-relabel-aux* 0 t)

lemma *fst-postorder-label-aux-tree-ltree*: $fst\ (postorder\text{-}label\text{-}aux\ n\ (tree\text{-}ltree\ t)) = fst\ (postorder\text{-}relabel\text{-}aux\ n\ t)$
 ⟨proof⟩

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

lemma *ran-postorder-relabel-aux*: $\text{ran} (\text{snd} (\text{postorder-relabel-aux } n \ t)) = \text{nodes-ltree } t$
 ⟨proof⟩

lemma *relabel-ltree-eq*: $\forall v \in \text{nodes-ltree } t. f \ v = g \ v \implies \text{relabel-ltree } f \ t = \text{relabel-ltree } g \ t$
 ⟨proof⟩

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

lemma *relabel-postorder-relabel*: $\text{relabel-ltree} (\text{the } o \ \text{postorder-relabel } t) (\text{postorder-label } (\text{tree-ltree } t)) = t$
 ⟨proof⟩

lemma *relabel-postorder-aux-inj*: $\text{distinct-ltree-nodes } t \implies \text{inj-on} (\text{the } o \ \text{snd} (\text{postorder-relabel-aux } n \ t)) (\text{nodes-ltree} (\text{snd} (\text{postorder-label-aux } n \ (\text{tree-ltree } t))))$
 ⟨proof⟩

lemma *relabel-postorder-inj*: $\text{distinct-ltree-nodes } t \implies \text{inj-on} (\text{the } o \ \text{postorder-relabel } t) (\text{nodes-ltree} (\text{postorder-label } (\text{tree-ltree } t)))$
 ⟨proof⟩

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

lemma *disintct-nodes-ltree-stree*: $\text{distinct-stree-nodes } t \implies \text{distinct-ltree-nodes} (\text{ltree-stree } t)$
 ⟨proof⟩

lemma (*in rtree*) *tree-graph-tree-of-graph*: $\text{tree-graph} (\text{tree-ltree} (\text{ltree-stree} (\text{stree-of-graph } (V, E, r)))) \simeq_r (V, E, r)$
 ⟨proof⟩

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

lemma *inj-ltree-stree*: $\text{inj } \text{ltree-stree}$
 ⟨proof⟩

lemma *ltree-size-ltree-stree[simp]*: $\text{ltree-size} (\text{ltree-stree } t) = \text{stree-size } t$
 ⟨proof⟩

lemma *tree-size-tree-ltree[simp]*: $\text{tree-size } (\text{tree-ltree } t) = \text{ltree-size } t$
 ⟨proof⟩

lemma *regular-ltree-stree*: $\text{regular-ltree } (\text{ltree-stree } t)$
 ⟨proof⟩

lemma *regular-tree-ltree*: $\text{regular-ltree } t \implies \text{regular } (\text{tree-ltree } t)$
 ⟨proof⟩

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

lemma (in *rtree*) *ex-regular-n-tree*: $\exists t \in \text{regular-n-trees } (\text{card } V). \text{tree-graph } t \simeq_r (V, E, r)$
 ⟨proof⟩

3.4 Injectivity with respect to isomorphism

lemma *app-rgraph-isomorphism-relabel-stree*: $\text{app-rgraph-isomorphism } f (\text{tree-graph-stree } t) = \text{tree-graph-stree } (\text{relabel-stree } f t)$
 ⟨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 $t: t = \text{SNode } r ts$

assumes *distinct-nodes*: $\text{distinct-stree-nodes } t$

and *remove-vertex*: $\text{graph-system.remove-vertex } (\text{nodes-stree } t) (\text{tree-graph-edges } t) r = (V', E')$

begin

interpretation $t: \text{rtree nodes-stree } t \text{ tree-graph-edges } t r$ ⟨proof⟩

interpretation *subg*: $\text{ulsubgraph } V' E' \text{ nodes-stree } t \text{ tree-graph-edges } t$ ⟨proof⟩

interpretation g' : $\text{ulgraph } V' E'$ ⟨proof⟩

lemma *neighborhood-root*: $t.\text{neighborhood } r = \text{root-stree } \text{' fset } ts$
 ⟨proof⟩

lemma V' : $V' = \text{nodes-stree } t - \{r\}$
 ⟨proof⟩

lemma E' : $E' = \bigcup (\text{tree-graph-edges } \text{' fset } ts)$
 ⟨proof⟩

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

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

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

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

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

lemma *subtrees-tree-subtrees*: $t.\text{subtrees} = \text{tree-graph-stree } \text{' fset } ts$
 $\langle \text{proof} \rangle$

end

lemma *stree-of-graph-tree-graph-stree[simp]*: $\text{distinct-stree-nodes } t \implies \text{stree-of-graph } (\text{tree-graph-stree } t) = t$
 $\langle \text{proof} \rangle$

lemma *distinct-nodes-relabel*: $\text{distinct-stree-nodes } t \implies \text{inj-on } f (\text{nodes-stree } t) \implies \text{distinct-stree-nodes } (\text{relabel-stree } f t)$
 $\langle \text{proof} \rangle$

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

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

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 $\exists f. \text{inj-on } f \text{ (nodes-stree } t1) \wedge \text{relabel-stree } f \text{ } t1 = t2$
 $\langle \text{proof} \rangle$

Skip the ltree representation as it introduces complications with the proofs

fun *tree-stree* :: 'a stree \Rightarrow 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 \Rightarrow tree \Rightarrow nat \times 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') \Rightarrow (n'' , SNode r (finsert t' ts')))

definition *postorder-label-stree* :: tree \Rightarrow 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)
 $\langle \text{proof} \rangle$

lemma *postorder-label-stree-aux-eq*: snd (*postorder-label-stree-aux* n t) = *stree-ltree* (snd (*postorder-label-aux* n t))
 $\langle \text{proof} \rangle$

lemma *postorder-label-stree-eq*: *postorder-label-stree* t = *stree-ltree* (*postorder-label* t)
 $\langle \text{proof} \rangle$

lemma *postorder-label-stree-aux-mono*: *fst* (*postorder-label-stree-aux* n t) $\geq n$
 $\langle \text{proof} \rangle$

lemma *nodes-postorder-label-stree-aux-ge*: *postorder-label-stree-aux* n t = (n' , t')
 $\Longrightarrow v \in \text{nodes-stree } t' \Longrightarrow v \geq n$
 $\langle \text{proof} \rangle$

lemma *nodes-postorder-label-stree-aux-le*: *postorder-label-stree-aux* n t = (n' , t')
 $\Longrightarrow v \in \text{nodes-stree } t' \Longrightarrow v \leq n'$
 $\langle \text{proof} \rangle$

lemma *distinct-nodes-postorder-label-stree-aux*: *distinct-stree-nodes* (snd (*postorder-label-stree-aux* n t))
 $\langle \text{proof} \rangle$

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

lemma *tree-stree-postorder-label-stree-aux*: *regular* $t \implies \text{tree-stree} (\text{snd} (\text{postorder-label-stree-aux}$ $n\ t)) = t$
 ⟨*proof*⟩

lemma *tree-ltree-postorder-label-stree[simp]*: *regular* $t \implies \text{tree-stree} (\text{postorder-label-stree}$ $t) = t$
 ⟨*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* $r\ ts$)) $\implies t \in \text{fset } ts \implies \text{inj-on}$ f (*nodes-stree* t)
 ⟨*proof*⟩

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

lemma *tree-ltree-relabel-ltree-postorder-label-stree*: *regular* $t \implies \text{inj-on } f$ (*nodes-stree* (*postorder-label-stree* t)) $\implies \text{tree-stree} (\text{relabel-stree } f$ (*postorder-label-stree* t)) = t
 ⟨*proof*⟩

lemma *postorder-label-stree-inj*: *regular* $t1 \implies \text{regular } t2 \implies \text{inj-on } f$ (*nodes-stree* (*postorder-label-stree* $t1$)) $\implies \text{relabel-stree } f$ (*postorder-label-stree* $t1$) = *postorder-label-stree* $t2 \implies t1 = t2$
 ⟨*proof*⟩

lemma *tree-graph-inj-iso*: *regular* $t1 \implies \text{regular } t2 \implies \text{tree-graph } t1 \simeq_r \text{tree-graph}$ $t2 \implies t1 = t2$
 ⟨*proof*⟩

lemma *tree-graph-inj*:
assumes *regular-t1*: *regular* $t1$
and *regular-t2*: *regular* $t2$
and *tree-graph-eq*: *tree-graph* $t1 = \text{tree-graph } t2$
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*: $\text{trim-tree } n \ t = (\text{Suc } n', t') \implies t = t'$
(proof)

lemma *trim-tree-le*: $(n', t') = \text{trim-tree } n \ t \implies t' \leq t$
(proof)

lemma *fill-tree-le*: $r \in \text{set } (\text{fill-tree } n \ t) \implies r \leq t$
(proof)

lemma *next-tree-aux-lt*: $\text{height } t \geq 2 \implies \text{the } (\text{next-tree-aux } n \ t) < t$
(proof)

lemma *next-tree-lt*: $\text{height } t \geq 2 \implies \text{the } (\text{next-tree } t) < t$
(proof)

lemma *next-tree-lt'*: $\text{next-tree } t = \text{Some } t' \implies t' < t$
(proof)

4.2 Size preservation

lemma *size-trim-tree*: $n \neq 0 \implies \text{trim-tree } n \ t = (n', t') \implies n' + \text{tree-size } t' = n$
(proof)

lemma *size-fill-tree*: $\text{sum-list } (\text{map } \text{tree-size } (\text{fill-tree } n \ t)) = n$
(proof)

lemma *size-next-tree-aux*: $\text{height } t \geq 2 \implies \text{tree-size } (\text{the } (\text{next-tree-aux } n \ t)) = \text{tree-size } t + n$
(proof)

lemma *size-next-tree*: $\text{height } t \geq 2 \implies \text{tree-size } (\text{the } (\text{next-tree } t)) = \text{tree-size } t$
(proof)

lemma *size-next-tree'*: $\text{next-tree } t = \text{Some } t' \implies \text{tree-size } t' = \text{tree-size } t$
(proof)

4.3 Setup for termination proof

definition *lt-n-trees* $n \equiv \{t. \text{tree-size } t \leq n\}$

lemma *n-trees-eq*: $n\text{-trees } n = \text{Node } \{ts. \text{tree-size } (\text{Node } ts) = n\}$
(proof)

lemma *lt-n-trees-eq*: $lt\text{-n-trees } (\text{Suc } n) = \text{Node } \{ts. \text{tree-size } (\text{Node } ts) \leq \text{Suc } n\}$
(proof)

lemma *finite-lt-n-trees*: $\text{finite } (lt\text{-n-trees } n)$
(proof)

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

lemma *finite-n-trees*: $\text{finite } (n\text{-trees } n)$
 ⟨proof⟩

4.4 Algorithms for enumeration

fun *greatest-tree* :: $\text{nat} \Rightarrow \text{tree}$ **where**
greatest-tree (Suc 0) = Node []
 | *greatest-tree* (Suc n) = Node [*greatest-tree* n]

function *n-tree-enum-aux* :: $\text{tree} \Rightarrow \text{tree list}$ **where**
n-tree-enum-aux t =
 (case *next-tree* t of None \Rightarrow [t] | Some t' \Rightarrow t # *n-tree-enum-aux* t')
 ⟨proof⟩

fun *n-tree-enum* :: $\text{nat} \Rightarrow \text{tree list}$ **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* :: $\text{nat} \Rightarrow \text{nat rpregraph list}$ **where**
n-rtree-graph-enum n = map *tree-graph* (*n-tree-enum* n)

4.5 Regularity

lemma *regular-trim-tree*: $\text{regular } t \Longrightarrow \text{regular } (\text{snd } (\text{trim-tree } n t))$
 ⟨proof⟩

lemma *regular-trim-tree'*: $\text{regular } t \Longrightarrow (n', t') = \text{trim-tree } n t \Longrightarrow \text{regular } t'$
 ⟨proof⟩

lemma *sorted-fill-tree*: $\text{sorted } (\text{fill-tree } n t)$
 ⟨proof⟩

lemma *regular-fill-tree*: $\text{regular } t \Longrightarrow r \in \text{set } (\text{fill-tree } n t) \Longrightarrow \text{regular } r$
 ⟨proof⟩

lemma *regular-next-tree-aux*: $\text{regular } t \Longrightarrow \text{height } t \geq 2 \Longrightarrow \text{regular } (\text{the } (\text{next-tree-aux } n t))$
 ⟨proof⟩

lemma *regular-next-tree*: $\text{regular } t \Longrightarrow \text{height } t \geq 2 \Longrightarrow \text{regular } (\text{the } (\text{next-tree } t))$
 ⟨proof⟩

lemma *regular-next-tree'*: $\text{regular } t \Longrightarrow \text{next-tree } t = \text{Some } t' \Longrightarrow \text{regular } t'$

<proof>

lemma *regular-n-tree-enum-aux*: $\text{regular } t \implies r \in \text{set } (n\text{-tree-enum-aux } t) \implies \text{regular } r$
<proof>

lemma *regular-n-tree-greatest-tree*: $n \neq 0 \implies \text{greatest-tree } n \in \text{regular-n-trees } n$
<proof>

lemma *regular-n-tree-enum*: $t \in \text{set } (n\text{-tree-enum } n) \implies \text{regular } t$
<proof>

lemma *size-n-tree-enum-aux*: $n \neq 0 \implies r \in \text{set } (n\text{-tree-enum-aux } t) \implies \text{tree-size } r = \text{tree-size } t$
<proof>

lemma *size-greatest-tree[simp]*: $n \neq 0 \implies \text{tree-size } (\text{greatest-tree } n) = n$
<proof>

lemma *size-n-tree-enum*: $t \in \text{set } (n\text{-tree-enum } n) \implies \text{tree-size } t = n$
<proof>

4.6 Totality

lemma *set (n-tree-enum n) ⊆ regular-n-trees n*
<proof>

lemma *greatest-tree-lt-Suc*: $n \neq 0 \implies \text{greatest-tree } n < \text{greatest-tree } (\text{Suc } n)$
<proof>

lemma *greatest-tree-ge*: $\text{tree-size } t \leq n \implies t \leq \text{greatest-tree } n$
<proof>

fun *least-tree* :: $\text{nat} \Rightarrow \text{tree}$ **where**
least-tree (Suc n) = Node (replicate n (Node []))

lemma *regular-n-tree-least-tree*: $n \neq 0 \implies \text{least-tree } n \in \text{regular-n-trees } n$
<proof>

lemma *height-lt-2-least-tree*: $t \in \text{regular-n-trees } n \implies \text{height } t < 2 \implies t = \text{least-tree } n$
<proof>

lemma *least-tree-le*: $n \neq 0 \implies \text{tree-size } t \geq n \implies \text{least-tree } n \leq t$
<proof>

lemma *trim-id'*: $n \geq \text{tree-size } t \implies \text{trim-tree } n t = (n', t') \implies t' = t$
<proof>

lemma *tree-ge-ll-suffix*: $\text{Node } ts \leq r \implies r < \text{Node } (t\#ts) \implies \exists ss. r = \text{Node } (ss @ ts)$
 ⟨proof⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

<proof>

theorem *n-rtree-graph-enum-surj*:

assumes *n-rtree-graph*: $G \in n\text{-rtree-graphs } n$

shows $\exists G' \in \text{set } (n\text{-rtree-graph-enum } n). G' \simeq_r G$

<proof>

4.7 Distinctness

lemma *n-tree-enum-aux-le*: $r \in \text{set } (n\text{-tree-enum-aux } t) \implies r \leq t$

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

and *H-in-n-rtree-graph-enum*: $H \in \text{set } (n\text{-rtree-graph-enum } n)$

and $G \simeq_r H$

shows $G = H$

<proof>

theorem *ex1-iso-n-rtree-graph-enum*: $G \in n\text{-rtree-graphs } n \implies \exists! G' \in \text{set } (n\text{-rtree-graph-enum } n). G' \simeq_r G$

<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.