

SCHOOL OF COMPUTATION,  
INFORMATION AND TECHNOLOGY —  
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

**Verified Enumeration of Trees**

Nils Cremer

SCHOOL OF COMPUTATION,  
INFORMATION AND TECHNOLOGY —  
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

**Verified Enumeration of Trees**

**Verifiziertes Aufzählen von Bäumen**

Author:	Nils Cremer
Supervisor:	Prof. Tobias Nipkov
Advisor:	Emin Karayel
Submission Date:	May 15, 2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, May 15, 2023

Nils Cremer

## **Acknowledgments**

# Abstract

# Contents

<b>Acknowledgments</b>	<b>iii</b>
<b>Abstract</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Graphs and Trees in Isabelle</b>	<b>2</b>
<b>3 Enumeration of Labeled Trees</b>	<b>3</b>
<b>4 Enumeration of Unlabeled Rooted Trees</b>	<b>4</b>
4.1 Algorithm . . . . .	5
4.2 Proof of correctness . . . . .	9
4.2.1 Totality . . . . .	9
4.2.2 Distinctness modulo Isomorphism . . . . .	10
<b>Abbreviations</b>	<b>12</b>
<b>Bibliography</b>	<b>13</b>

# 1 Introduction

## 2 Graphs and Trees in Isabelle

Graph library [Edm22]



### **3 Enumeration of Labeled Trees**

## 4 Enumeration of Unlabeled Rooted Trees

This Chapter describes the verification of an algorithm enumerating all unlabeled rooted trees with a specific number of nodes. Let  $G = (V, E, r)$  denote the rooted graph with vertex set  $V$ , edge set  $E$  and a distinguished root  $r \in V$ .

```
locale rgraph = graph_system +  
  fixes r  
  assumes root_wf: " $r \in V$ "
```

Similarly, a rooted tree is a tree with a fixed root.

```
locale rtree = tree + rgraph
```

As can be seen in the definition the nodes of rooted graphs are still labeled. To get a notion of unlabeled trees we define isomorphism on these graphs. A graph isomorphism for (unrooted) graphs is a bijection between the vertex sets of two graph which preserves edge connectivity.

```
locale graph_isomorphism =  
  G: graph_system V_G E_G for V_G E_G +  
  fixes V_H E_H f  
  assumes bij_f: " $\text{bij\_betw } f \text{ } V_G \text{ } V_H$ "  
  and edge_preserving: " $((') f) ' E_G = E_H$ "
```

Similarly, graph isomorphism is defined on rooted graphs with the additional requirement that the isomorphism also preserves the root node.

```
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: " $f \text{ } r_G = r_H$ "
```

The algorithm described here produces a list of all rooted trees with  $n$  vertices modulo isomorphism. That is, every rooted tree is isomorphic to exactly one rooted tree in the resulting list.

## 4.1 Algorithm

The idea of the algorithm is based on one described by Beyer and Hedetniemi [BH80]. It works on ordered, rooted trees. An ordered, rooted tree is a rooted tree with a linear order with which subtrees can be ordered. We will represent such a tree graphically as in Figure 4.1 by drawing the root node at the top and the subtrees in the corresponding order beneath it. Since we are interested in unlabeled trees node labeled will generally be omitted. The two trees shown differ in the ordering of their subtrees but represent the same underlying rooted tree. Of the possibly many ordered trees corresponding to a rooted tree, one canonical tree will be defined. For this, a linear order will be defined on ordered rooted trees. First we define the type of ordered rooted trees as the root node with a list of ordered subtrees.

```
datatype tree = Node "tree list"
```

This order defined on these trees will be the reversed lexicographical ordering. No tree is smaller than the singleton tree (a tree only containing a root node). The singleton tree is smaller than every other tree besides itself. Otherwise, the subtrees are compared element wise from the back. To simplify the definition in a functional setting, we first define the regular lexicographical order on trees and then define the desired ordering by mirroring its arguments.

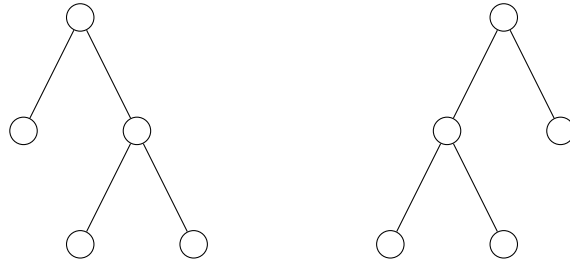


Figure 4.1: Example rooted trees

```

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))"

definition
  tree_less_def: "(t::tree) < r  $\longleftrightarrow$  lexord_tree (mirror t) (mirror r)"

```

For a rooted tree  $T$ , we now define the canonical ordered tree as the greatest ordered tree corresponding to  $T$ . In Figure 4.1 the left tree is the canonical ordered tree.

The algorithm will produce all canonical ordered trees. Instead of working with rooted trees directly the algorithm of Beyer and Hedetniemi uses an alternative representation of rooted trees, namely level sequences. The algorithm described here enumerates rooted trees in the same order but is defined recursively on a recursive tree datatype instead of on the level sequences.

To traverse all rooted trees a successor function is defined giving the next smallest canonical tree. To accomplish this, some auxiliary function are needed. We define a function that trims a tree,  $trim\_tree\ n\ t$  trims  $t$  to  $n$  nodes by removing nodes in postorder until the tree has at most  $n$  nodes. See Figure 4.2 for an example of how  $trim\_tree$  works.

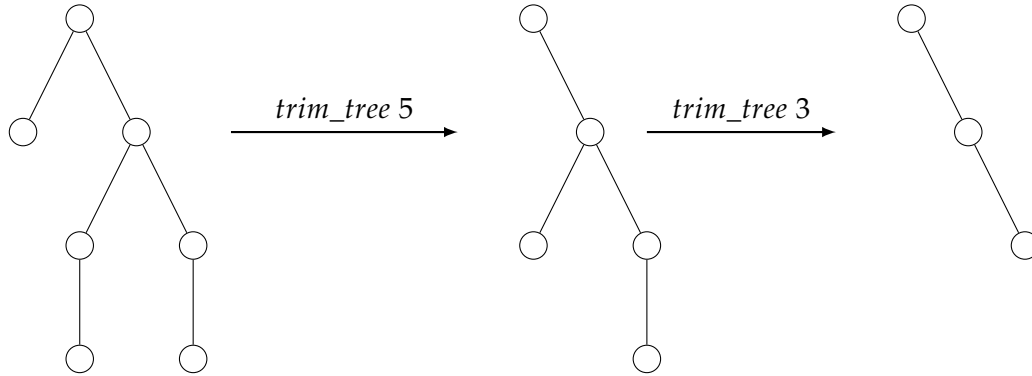


Figure 4.2: Example of  $trim\_tree$

```

fun trim_tree :: "nat  $\Rightarrow$  tree  $\Rightarrow$  nat  $\times$  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')  $\Rightarrow$  (0, t') |
    (n1, Node ts')  $\Rightarrow$ 
      let (n2, t') = trim_tree n1 t
      in (n2, Node (t'#ts')))"

```

With the help of this a function *fill\_tree* is defined, *fill\_tree n t* produces a list of trees with a total number of  $n$  nodes. Each tree in the list will be  $t$  except the first one which might be trimmed to a suitable size in order for the total number of nodes to be  $n$ .

```

fun fill_tree :: "nat  $\Rightarrow$  tree  $\Rightarrow$  tree list" where
  "fill_tree 0 _ = []"
| "fill_tree n t =
  (let (n', t') = trim_tree n t
   in fill_tree n' t' @ [t'])"

```

Now we define the successor function which produces, given a tree, the next smallest canonical tree. Generating the next smallest tree follows these 4 steps:

1. Remove all direct subtrees of the root node which are singleton trees
2. Find the first node  $v$  (in preorder) which has a singleton tree as the first subtree.  
Let the subtree of  $v$  be  $T_v$
3. Remove the first singleton subtree of  $T_v$  which result in a tree  $T'_v$
4. Let  $p$  be the parent of  $v$ . Fill the subtree list of  $p$  with  $T'_v$  until the entire tree consists of  $n$  nodes again

See Figure 4.3 for examples of how the function operates.

This function can be directly defined recursively on the tree datatype.

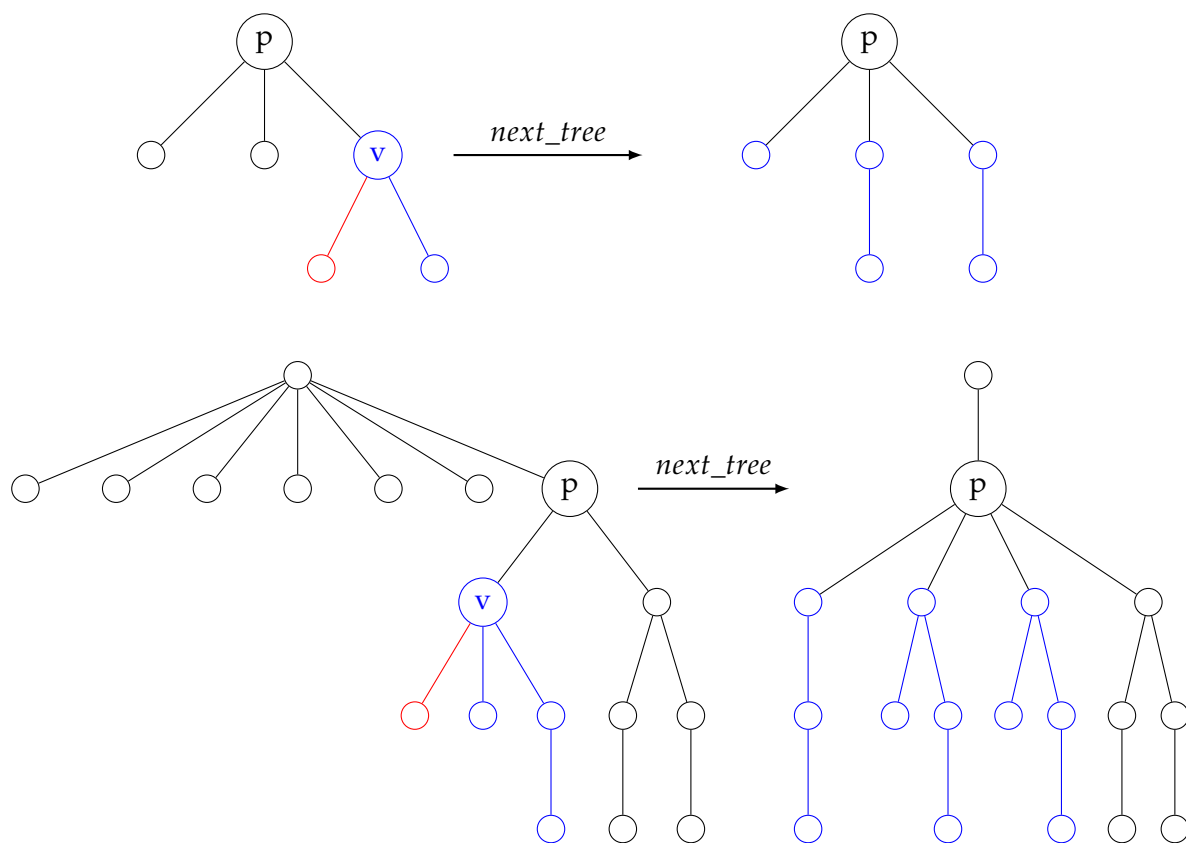


Figure 4.3: Example of *next\_tree*

```

fun next_tree_aux :: "nat  $\Rightarrow$  tree  $\Rightarrow$  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  $\Rightarrow$  tree option" where
  "next_tree t = next_tree_aux 0 t"

```

None will be returned if the argument is already the smallest tree and thus there is no next smaller tree.

To now enumerate all rooted trees with  $n$  nodes, one starts with the greatest tree on  $n$  nodes and repeatedly calls *next\_tree* until the smallest tree is reached.

```

fun greatest_tree :: "nat  $\Rightarrow$  tree" where
  "greatest_tree (Suc 0) = Node []"
| "greatest_tree (Suc n) = Node [greatest_tree n]"

function n_tree_enum_aux :: "tree  $\Rightarrow$  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')"
by pat_completeness auto

fun n_tree_enum :: "nat  $\Rightarrow$  tree list" where
  "n_tree_enum 0 = []"
| "n_tree_enum n = n_tree_enum_aux (greatest_tree n)"

```

To prove termination of the enumeration we show that the output of *next\_tree* (if it is not *None*) is strictly smaller than its input and that there are only finitely many rooted trees with  $n$  nodes.

## 4.2 Proof of correctness

### 4.2.1 Totality

Instead of working with canonical rooted tree, we work with an alternative (equivalent) definition of regularity. A rooted ordered tree is regular iff the subtrees are ordered with respect to the linear order previously defined and all subtrees are regular themselves.

To prove correctness we first show that the algorithm enumerates all regular ordered rooted trees and then that the set of regular ordered rooted trees is equivalent to the set of rooted trees modulo isomorphism.

By simple induction all defined functions can be shown to preserve regularity. In addition to that *next\_tree* preserves the size of the tree, again by induction and simple auxiliary lemmas on the size of the output of the defined function. Since the *greatest\_tree*  $n$  is regular and in of size  $n$  the algorithm produces only regular trees of size  $n$ . Due to the strict monotonicity of *next\_tree* proven earlier it immediately follows that no tree is generated twice. In addition, we prove that there is a unique minimal tree of size  $n$ , namely the tree consisting of a root node and  $n - 1$  singleton subtrees. The enumeration stops only if this minimal tree is reached. What is left to prove is that the algorithm does not skip any regular rooted tree. For this some auxiliary lemmas are needed. First, observe that due to the order in which *trim\_tree* removes nodes, *trim\_tree*  $n$   $t$  is the greatest tree with at most  $n$  nodes that is smaller or equal to  $t$ . This also means that a tree which has as subtrees the list of trees generated by *fill\_tree*  $n$   $t$  is the greatest tree of size  $n + 1$  (observe the additional root node) where every subtree is smaller or equal to  $t$ . Now to get an intuition of why *next\_tree* generates the next smallest tree consider counting down numbers as analogy. When counting down from 14200 one tries to decrease the rightmost digit that can be decreased, i.e. the 2. Then set the digits to the right of the decreased number to their maximum resulting in 14199.

Since subtrees are compared in reverse order, the leftmost subtree that can be decreased (i.e. is not a singleton) will be decreased by removing its leftmost leaf. Then using *fill\_tree* the subtree left to the decreased tree is maximized. This argument can be made formal by computation induction on *next\_tree* and with some technicalities we omit here. Thus, we have proven that all regular trees of size  $n$  are enumerated by the algorithm.

#### 4.2.2 Distinctness modulo Isomorphism

In order for the algorithm to meet its specification we need to show that the regular ordered rooted trees are equivalent to all rooted trees. We define a function to map an ordered unlabeled rooted tree to a rooted labeled tree in graph form. This is done via two additional intermediate steps which simplify the proofs. First we define ordered rooted labeled trees which are like ordered rooted trees except that each node has a label associated with it. A function is defined to label the nodes of an unlabeled tree. This can be done arbitrarily as long as the nodes have different labels since any labeling will produce an isomorphic tree. Next we define unordered labeled trees where the subtrees of a node are represented as a finite set instead of an ordered list. As the last step one can transform an unordered labeled tree  $t$  into its graph representation simply



by the graph  $G = (V, E)$  where  $V$  is the set of nodes in  $t$  and  $E$  contains an edge for every node to the roots of its subtrees. Let  $t$  be an ordered unlabeled rooted tree, then  $G_t$  denotes the graph representation of  $t$ . It can be shown that  $G_t$  is indeed a tree, i.e. connected and acyclic.

We now want to show for every tree graph  $T$  there exists exactly one  $t$  such that  $T$  and  $G_t$  are isomorphic. Let  $r$  be the root of  $T$ . Moreover, let  $C$  be the set of connected components of  $T - \{r\}$  (the tree without its root). Then one can generate unordered unlabeled trees for each connected component recursively by taking the unique vertex in it that is connected to  $r$  as its root. The sorted list of these trees will be the subtrees of  $t$ . This construction results in a regular tree  $t$  such that  $G_t$  is isomorphic to  $T$ . Denote this tree by  $t(T)$ .

The remaining step to show uniqueness is to show that  $G_{t_1}$  and  $G_{t_2}$  are not isomorphic for two distinct regular trees  $t_1$  and  $t_2$ . One can prove that  $t(G_t) = t$  for all regular trees  $t$ . Also, since the construction of  $t(T)$  does not depend on the labels of the nodes,  $t(G_{t_1}) = t(G_{t_2})$ . This means

$$t_1 = t(G_{t_1}) = t(G_{t_2}) = t_2$$

which is a contradiction to the assumption that  $t_1$  and  $t_2$  are distinct.

This concludes the entire proof and shows that the described algorithm generates each tree exactly once modulo isomorphic, that is it generates all unlabeled rooted trees.

## Abbreviations

# Bibliography

- [BH80] T. Beyer and S. M. Hedetniemi. “Constant Time Generation of Rooted Trees.” In: *SIAM Journal on Computing* 9.4 (1980), pp. 706–712. DOI: 10.1137/0209055. eprint: <https://doi.org/10.1137/0209055>.
- [Edm22] C. Edmonds. “Undirected Graph Theory.” In: *Archive of Formal Proofs* (Sept. 2022). [https://isa-afp.org/entries/Undirected\\_Graph\\_Theory.html](https://isa-afp.org/entries/Undirected_Graph_Theory.html), Formal proof development. ISSN: 2150-914x.