Saarland University
Faculty of Mathematics and Computer Science

Bachelor's Thesis

# Nu-Trees in Coq

## Their Language and Automata

**Author**
Leonhard Staut

**Advisor**
Dominik Kirst

**Supervisor**
Prof. Dr. Gert Smolka

**Reviewers**
Prof. Dr. Gert Smolka
Prof. Bernd Finkbeiner, Ph.D.

Submitted : 28th September 2017, Saarbrücken

**Eidesstattliche Erklärung**

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

**Statement in Lieu of an Oath**

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

**Einverständniserklärung**

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

**Declaration of Consent**

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 28th September, 2017

# Abstract

We study tree languages obtained by systematically permuting names over an infinite alphabet. For this purpose we define $\nu$-trees which contain $\nu$-nodes that bind names. We use nominal sets embedded in type theory to formalize types with a permutation action on their elements and use this as a framework for our development.
The language of a $\nu$-tree is a class of pure trees. We show that this class is decidable and show equivalence laws which correspond to the nominal axioms proposed by Gabbay. This leads towards a decision procedure for $\nu$-tree language equivalence.
Furthermore we introduce an automaton model for $\nu$-trees which follows the definition of dependency tree automata by Stirling. We show that acceptance is decidable for our model.

# Acknowledgements

I dearly thank Dominik Kirst not only for providing the topic but also for his valuable guidance in the process of developing and writing this thesis. Working with him was fun and motivating. Without his continuous input and helpful feedback this thesis would not have been possible.
I thank Prof. Gert Smolka for introducing me to the field of computational logic and also for the Coq Base library which I use in my development. Yannik Forster created this thesis template which I use here, and I thank him for that.

# Contents


**Abstract** **v**

**1 Introduction** **1**
1.1 Contributions . . . 2
1.2 Outline . . . 3

**2 Nominal Types** **4**

**3 Pure Trees** **7**
3.1 Definition . . . 7
3.2 Nominality . . . 8

**4 $\nu$-Trees and their languages** **10**
4.1 Definition and Nominality . . . 10
4.2 $\nu$-Tree Language . . . 12
4.2.1 Definition . . . 13
4.2.2 Equivariance . . . 13

**5 Equivalence Laws** **16**
5.1 $\rho$-Renaming . . . 17
5.2 Swapping bindings . . . 19
5.3 Weakening and Strengthening . . . 20
5.4 Empty bindings . . . 23
5.5 Pushing bindings . . . 24
5.6 Towards decidability of $\nu$-tree equivalence . . . 28

**6 Decidability of $\nu$-tree languages** **30**
6.1 Decision Procedure . . . 30
6.2 Correctness . . . 31

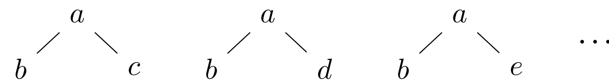
**7 $\nu$-Tree Automata** **34**
7.1 The NTA model . . . 34
7.2 Decidability of the Word Problem . . . 37

**8 Discussion** **40**
8.1 Related Work . . . . . 40
8.2 Implementation . . . . . 41
8.2.1 Names and permutations . . . . . 41
8.2.2 Well-ranked trees . . . . . 41
8.2.3 Similar trees . . . . . 42
8.2.4 Exhaustive mappings between two lists . . . . . 43
8.3 Future work . . . . . 45
8.4 Development details . . . . . 45

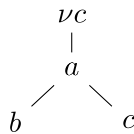
# Chapter 1

# Introduction

Tree languages have applications in software verification [2, 3], and XML parsing [4]. The alphabet in these systems is infinite, and reasoning is therefore undecidable in general. The approach is to focus on decidable subclasses of these languages. The intention is to find a subclass that can be represented using a finite structure but is still rich enough to contain significant languages. This thesis is about the formal development of $\nu$-trees which define a class of tree languages over an infinite alphabet $\mathbb{A}$. We begin with some examples of trees.



These trees follow a pattern. They share the same structure, but the lower right node has a new name in each example which is different from the other names which stay fixed. Since $\mathbb{A}$ is infinite, there are infinitely many more similar trees. The goal is to find a finite representation of this class of trees. Adding bindings to the tree is our chosen method to express this pattern.



The idea is that a $\nu$-binding captures all permutations of the respective bound name, $c$ in the example, with the constraint that the bound name does not clash with unbound constants in the tree. Furthermore, all $\nu$-bindings that lie on the same path in a tree are required to be instantiated differently, meaning a $\nu$-binding always introduces a new name. These two constraints are called the freshness conditions and pose some restrictions on which names in $\mathbb{A}$ may be used to instantiate a binding. However, since $\mathbb{A}$ is infinite, and the constraints are finite, there are infinitely many fresh names to instantiate a binding, and we achieve a finite representation of infinitely many different trees.
The language of a single $\nu$-tree is a collection of trees where the structure is the

same and the $\nu$-bindings are instantiated with fresh names while the unbound names stay fixed.
Two different $\nu$-trees can have the same language. For instance, two $\nu$-trees that differ only in the concrete names of their $\nu$-bindings but are otherwise exactly identical have the same language. The $\nu$-bindings are instantiated and therefore their concrete name is irrelevant. This is one of the equivalence laws that we establish in this thesis under which $\nu$-tree languages are invariant.
The theory of nominal sets as developed by Pitts [13] provides a framework for structures with names and a permutation action. Its aim is to express symmetries of variable names in structures that have variable binding, abstracting from the concrete names of bound variables. A structure is nominal if it contains only finitely many names. The idea is that we only need finite resources to fix such a structure even in the context of an infinite alphabet of names.
The semantics for $\nu$-bindings and $\nu$-trees are given by Kirst [11] as a fragment of the $\nu$-calculus [12, 15]. In his work, the language of a $\nu$-tree is defined as a possibly infinite set of pure trees.
We use the constructive type theory underlying Coq as the basis for our formalization. Therefore we translate the concept of nominal sets to nominal types. We develop an inductive type of $\nu$-trees and formalize the language of a $\nu$-tree as a class of pure trees.
From there we define an automaton model which takes $\nu$-trees as input. The model is an instance of Stirling's dependency tree automata [16] and has also been studied by Kirst in [11]. The idea of this automaton model is to use a state assignment to remember if a name is bound or not. In this model there are different kinds of transitions depending on whether the current name is bound. The accepted language of this automaton is a class of $\nu$-trees. Since the language of a $\nu$-tree is a class of pure trees, the language of the automaton brings forth a language of pure trees.

## 1.1 Contributions

We give a fully constructive development of $\nu$-trees extending Kirst's work on them. We show new strengthening and weakening results for $\nu$-tree languages. We extend Gabbay's work on $\nu$-words by lifting the nominal axioms from words to equivalence laws for $\nu$-tree languages. With these equivalence laws we have taken steps towards a decision procedure for the equivalence of $\nu$-tree languages. To show the decidability of acceptance for $\nu$-tree languages we define a decision procedure and give a correctness proof using our equivalence laws. Finally, we formalize the $\nu$-tree automaton model and its language. We implement a generalized decision procedure for acceptance by $\nu$-tree automata which was outlined by Kirst. The full source of the formalization is available at:

`https://www.ps.uni-saarland.de/~staut/bachelor.php`

## 1.2 Outline

We start by introducing nominal types in Chapter 2 as a framework for types with a permutation action. We give a short treatment of pure trees equipped with a permutation action in Chapter 3 and we show that the type of pure trees is nominal. Afterwards we introduce $\nu$-trees in Chapter 4. Similarly to pure trees, we first show that the type of $\nu$-trees is nominal. We then go on to define the language of $\nu$-trees, a class of pure trees. There are certain equivalence laws that hold for $\nu$-tree languages which we establish in Chapter 5. We study how these equivalence laws correspond to the nominal axioms for $\nu$-words by Gabbay. In Chapter 6 we show that $\nu$-tree languages are decidable, that is, acceptance by a $\nu$-tree is decidable. For this purpose we define a decision procedure and prove it correct. In Chapter 7 we study an automaton model for $\nu$-trees. We define its language and show that it is decidable. In Chapter 8 we summarize previous work related to our topic. We also discuss possible future work and provide some implementation details of our formalization.

# Chapter 2

# Nominal Types

The theory of nominal sets is a formalization of sets with a group action on them. The group of permutations is of particular interest to us since we consider $\nu$-bindings in a tree which are instantiated by a permutation action.
Nominal sets themselves were originally introduced by Fraenkel in [5] to show the independence of the choice axiom from the ZF set theory without foundation. In computer science Gabbay and Pitts rediscovered them in [6] as an elegant way to formalize names and binding in languages. Pitts later summarized the theory of nominal sets in [13]. Since we work in type theory we translate nominal sets to nominal types in this chapter.
We assume a countably infinite alphabet of names $\mathbb{A}$.

**Definition 2.1** A permutation on a type $X$ is a function $\pi : X \to X$ which has an inverse permutation $\pi'$, such that

$$\forall\, x : X.\ \pi(\pi'(x)) = x = \pi'(\pi(x)).$$

We write $\mathtt{Sym}(\mathbb{A})$ for the type of permutations on $\mathbb{A}$.

Now we consider types $X$ that provide some structure with respect to $\mathtt{Sym}(\mathbb{A})$. We define **actions** that apply permutations $\pi \in \mathtt{Sym}(\mathbb{A})$ on the elements $x : X$.

**Definition 2.2** A function $(\cdot) : \mathtt{Sym}(\mathbb{A}) \to X \to X$ is called a **sym action** if

$$\mathrm{id} \cdot x = x \qquad \pi \cdot (\pi' \cdot x) = (\pi \circ \pi') \cdot x, \forall \pi, \pi' \in \mathtt{Sym}(\mathbb{A})$$

$X$ is called a **sym type** if such a function exists.

**Remark** The standard theory of nominal sets only considers the set of finite permutations $\mathtt{Perm}(\mathbb{A})$, where $\pi(a) \neq a$ for only finitely many $a$. We do not need that distinction and consider arbitrary permutations instead. The most general definition even considers actions of arbitrary groups.

**Definition 2.3** A unary predicate $P : T \to \mathtt{Prop}$ on any type $T$ is called a **class**. We write $t \in P$ if $P\ t$ holds. Two classes $P, P'$ are equivalent if $\forall\, t.\ P\ t \leftrightarrow P'\ t$ holds and we write $P \equiv P'$.

We now define the **orbit** of an element of a sym type which is the class obtained by abstracting away from all names appearing in that element.

**Definition 2.4** Let $X$ be a sym type, and $x, y : X$.
We say $x$ and $y$ are **orbit equivalent** if $x$ can be obtained by applying a permutation on $y$.

$$x \approx_{\text{orb}} y \coloneqq \exists\, \pi.\ \pi \cdot y = x$$

The **orbit** of $x$ is defined as the equivalence class of all elements to which $x$ is orbit equivalent.

$$y \in \texttt{orb}_x \coloneqq x \approx_{\text{orb}} y$$

If $X$ allows only finitely many intensionally different equivalence classes, we call $X$ **orbit-finite**.

**Remark** The fact that $\approx_{\text{orb}}$ is indeed an equivalence relation follows from the group properties of $\texttt{Sym}(\mathbb{A})$.

**Example 2.5** The alphabet $\mathbb{A}$ itself is an example of an orbit-finite type. The elements of $\mathbb{A}$ are names. We define $(\cdot)$ by $\pi \cdot a = \pi(a)$. It is easy to see that this is indeed a sym action, and $\mathbb{A}$ is therefore a sym type.
We have for any two names $a$ and $b$ that $a \approx_{\text{orb}} b$, since $(ab) \cdot a = b$. This means any name $a$ is orbit equivalent to any other name $b$. Therefore there is only a single equivalence class.

We now go on to define what it means for a sym type $X$ to be nominal.

**Definition 2.6** Let $A : \mathscr{L}(\mathbb{A})$ be a list of names.
$\texttt{fix}(A) : \texttt{Sym}(\mathbb{A}) \to \texttt{Prop}$ is the class of all permutations that fix all $a_k \in A$.

$$\pi \in \texttt{fix}(A) \coloneqq \forall\, a \in A.\ \pi(a) = a$$

We say $\pi$ **fixes** $A$ if $\pi \in \texttt{fix}(A)$.

**Definition 2.7** Let $A : \mathscr{L}(\mathbb{A}), x : X$.
$A$ is a **finite support** of $x$ if

$$\pi \cdot x = x \text{ holds for all } \pi \text{ that fix } A.$$

$X$ is called **nominal**, if every $x : X$ has a finite support.

**Remark** In the standard theory of nominal sets a support is defined as a set of names, and a finite support as a finite set of names. In nominal sets the finiteness of the support is required explicitly. In nominal types we only formalize finite supports.

The concept of nominality captures the idea that the elements of a type depend only on finitely many names. Even though $\mathbb{A}$ is infinite, only finitely many names have to be chosen to fix an element of the type. Intuitively, any type where the elements have a finite structure is nominal, since any element depends only on finitely many names. This includes for example the usual context free grammars of programming languages.
A significant concept for nominal types is **equivariance**.

**Definition 2.8** Let $X$ be a nominal type.
A class $P : X \to \texttt{Prop}$ is called **equivariant** if

$$\forall\, \pi.\ \pi \cdot P \equiv P,$$

where $\pi \cdot P$ denotes the pointwise lifting of the action, $x \in \pi \cdot P \coloneqq \pi \cdot x \in P$.

A class is equivariant if it is closed under all permutations. The orbit is a straightforward example of an equivariant class. If $y \in \texttt{orb}_x$, then there is a permutation $\pi_y$, such that $\pi_y \cdot y = x$. For all permutations $\pi$ there exists an inverse permutation $\pi'$. We have $\pi_y \cdot \pi' \cdot \pi \cdot y = \pi_y \cdot y = x$, and therefore $\pi \cdot y \in \texttt{orb}_x$.
Next we also define equivariance for functions.

**Definition 2.9** Let $f : X \to Y$ be a function for nominal types $X, Y$.
$f$ is called **equivariant** if

$$\forall \pi.\ \pi \cdot f(x) = f(\pi \cdot x).$$

Intuitively a function is equivariant, if its behavior does not depend on any concrete names, meaning there is no case distinction on them. Instead it is defined only in terms of the structure of the arguments.
There are a few differences between nominal types and nominal sets. We defined a support as a list of names which is finite by construction. A sym type $X$ is therefore nominal if every element has a support. There is no need to additionally require the support to be finite. This becomes necessary in nominal sets where a support is defined as a possibly infinite set. In nominal types we do not consider infinite supports.
Another difference is the definition of equivariance. We defined it for classes and for functions on nominal types. In nominal sets it is defined for subsets $Y \subseteq X$ of nominal sets $X$ instead of classes. Since functions in set theory are sets of pairs $f \subseteq X \times Y$, the definition of equivariance for functions is a special case of the definition for subsets. In type theory this is not the case and we need both definitions.

# Chapter 3

# Pure Trees

In this chapter we study pure trees over the infinite alphabet $\mathbb{A}$. We assume $\mathbb{A}$ to be a ranked alphabet, meaning there is a rank function $\mathtt{rk} : \mathbb{A} \to \mathbb{N}$ which assigns a rank $\mathtt{rk}(a)$ to each name $a : \mathbb{A}$. If $\mathtt{rk}(a) = k$, we also write $a_k$ to indicate the rank of the name. We furthermore assume $\mathbb{A}$ to be countably infinite in every rank. This setup is such that we always have a fresh name for any rank, relative to a list of blocked names. In fact, we can compute a name which is fresh and has the rank $k$.

**Lemma 3.1** $\forall L\ k.\ \exists a.\ a \notin L \wedge rk(a) = k$

**Proof** We first define $\mathbb{A}_k := \{a \in \mathbb{A} \mid \mathtt{rk}(a) = k\}$ as all names of rank $k$, and $L_k := \{a \in L \mid \mathtt{rk}(a) = k\}$ as all names in $L$ of rank $k$. Since $\mathbb{A}$ is countably infinite in every rank, $\mathbb{A}_k$ can be ordered by bijection with $\mathbb{N}$. Therefore there exists a largest element $m$ in the list $L_k$ such that $\forall\ a \in L.\ a \leq m$. Clearly $S\ m > m$, and therefore $S\ m \notin L \wedge \mathtt{rk}(S\ m) = k$. □

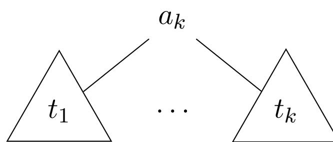
## 3.1 Definition

We begin by defining the type of trees where every node is labeled with a name from $\mathbb{A}$.

**Definition 3.2** The type of pure trees $\mathbb{A}$-**Tree** is defined inductively by

$$t ::= a_k t_1 \dots t_k$$

This corresponds to the tree




**Remark** We only consider well-ranked trees where a node labeled with $a_k$ has exactly $k$ children.
This definition includes leafs with names of rank zero but excludes the empty tree.

**Definition 3.3** The list of names in a tree $\texttt{Name}(t)$ is defined by

$$\texttt{Name}(a_k t_1 \dots t_k) = a_k :: (\texttt{Name}(t_1) +\!\!+ \dots +\!\!+ \texttt{Name}(t_k)),$$

where $(::)$ denotes the cons operation, and $(+\!\!+)$ denotes list concatenation.

## 3.2 Nominality

Next we equip $\mathbb{A}$-Tree with an action which defines how a permutation of names acts on a tree.

**Definition 3.4** The action $(\cdot)$ on $\nu$-Tree is defined by

$$\pi \cdot (a_k t_1 \dots t_k) = \pi(a_k)(\pi \cdot t_1) \dots (\pi \cdot t_k).$$

**Remark** Since we work with a ranked alphabet we only consider permutations where $\texttt{rk}(a) = \texttt{rk}(\pi(a))$. We call such permutations **rank preserving**. The reasoning behind it is that rank preserving permutations preserve well-rankedness when applied to a tree with the action $(\cdot)$.

It is straight forward to verify that $(\cdot)$ is indeed a sym action and hence $\mathbb{A}$-Tree is a sym type.

**Fact 3.5** $(\cdot)$ is a sym action for $\mathbb{A}$-Tree.

**Proof** We prove this claim by structural induction on the tree $t$.

$$\begin{aligned} & \text{id} \cdot (a_k t_1 \dots t_k) \\ = {} & \text{id}(a_k)(\text{id} \cdot t_1) \dots (\text{id} \cdot t_k) \\ \overset{\text{IH}}{=} {} & (a_k t_1 \dots t_k) \end{aligned}$$

$$\begin{aligned} & (\pi \circ \pi') \cdot (a_k t_1 \dots t_k) \\ = {} & (\pi \circ \pi')(a_k)((\pi \circ \pi') \cdot t_1 \dots (\pi \circ \pi') \cdot t_k) \\ \overset{\text{IH}}{=} {} & (\pi(\pi'(a_k))(\pi \cdot (\pi' \cdot t_1) \dots (\pi \cdot (\pi' \cdot t_k)) \\ = {} & \pi \cdot (\pi' \cdot (a_k t_1 \dots t_k)) \end{aligned}$$

□

One basic fact is the following. Two permutations $\pi, \pi'$ that they behave the same on $\texttt{Name}(t)$, result in the same tree when applied to $t$. This enables us to swap $\pi$ and $\pi'$.

**Lemma 3.6** $(\forall a \in \texttt{Name}(t).\ \pi(a) = \pi'(a)) \to \pi \cdot t = \pi' \cdot t$.

**Proof** We prove this claim by structural induction on the tree $t$.

$$\begin{aligned} & \pi \cdot (a_k t_1 \dots t_k) \\ = {} & \pi(a_k)(\pi \cdot t_1) \dots (\pi \cdot t_k) \\ \overset{\text{IH}}{=} {} & \pi'(a_k)(\pi' \cdot t_1) \dots (\pi' \cdot t_k) \\ = {} & \pi' \cdot (a_k t_1 \dots t_k). \end{aligned}$$

□

This lemma is significant, because it allows swapping permutations on a tree that behave the same on finitely many names. They do not need to behave the same on the entire alphabet $\mathbb{A}$. Furthermore, with this lemma we do not need to assume functional extensionality. In the context of trees, it allows us to swap two different permutations if they behave the same but they do not need to be exactly identical.
We now show that $\mathbb{A}$-Tree is a nominal type. Specifically, we prove that for all trees $t \in \mathbb{A}$-Tree the list of names $\texttt{Name}(t)$ is a finite support for $t$.

**Fact 3.7** $\mathbb{A}$-Tree with $(\cdot)$ is a nominal type.

**Proof** Let $\pi \in \texttt{fix}(\texttt{Name}(t))$. On $\texttt{Name}(t)$ the permutation $\pi$ agrees with the identity permutation. By Lemma 3.6 and the property of the identity permutation in a sym type (Fact 3.5):

$$\pi \cdot t = \text{id} \cdot t = t.$$

□

**Remark** It is worth mentioning that $\texttt{Name}(t)$ is the least support for $t$. When removing any name $a_k$ in $\texttt{Name}(t)$, it is easy to construct a permutation from it, that does not fix $t$ by not fixing $a_k$.

# Chapter 4

# $\nu$-Trees and their languages

We build on the previous section by defining $\nu$-trees which contain $\nu$-bindings in addition to trees. These $\nu$-bindings capture all permutations of the bound names with fresh names. If we instantiate all bindings we obtain a pure tree without $\nu$-bindings that has the same structure as the $\nu$-tree. We consider the class of all pure trees where the structure and the names match after instantiating the bindings in a $\nu$-tree and call this the $\nu$-tree language $[\![-]\!] : \nu\text{-Tree} \to \text{Tree} \to \texttt{Prop}$. Since there are infinitely many fresh names, there are infinitely many different instantiations for any $\nu$-binding. If a $\nu$-tree has therefore at least one $\nu$-binding we capture infinitely many instantiations and the $\nu$-tree language is an infinite class of pure trees.

## 4.1 Definition and Nominality

We now give a definition of $\nu$-trees which follows the definition given by Kirst in [11].

**Definition 4.1** The type **$\nu$-Tree** is defined inductively by

$$n ::= a_k n_1 \dots n_k \mid \nu a_k.n$$

We say $\nu a_k.n$ binds the ranked name $a_k$ in the tree $n$. In particular, $\nu a_k$ and $\nu a_l$ with $k \neq l$ bind different names. If a name is not bound in the tree, we call it free.

**Definition 4.2** The list of names in a $\nu$-tree $\texttt{Name}(n)$ is defined by

$$\begin{aligned} \texttt{Name}(a_k n_1 \dots n_k) &= a_k :: (\texttt{Name}(n_1) +\!\!\!+ \dots +\!\!\!+ \texttt{Name}(n_k)) \\ \texttt{Name}(\nu a_k.n) &= a_k :: \texttt{Name}(n) \end{aligned}$$

**Definition 4.3** The list of free names in a $\nu$-tree $\texttt{FN}(n)$ is the list of those names that are not bound. It is defined by

$$\begin{aligned} \texttt{FN}(a_k n_1 \dots n_k) &= a_k :: (\texttt{FN}(n_1) +\!\!\!+ \dots +\!\!\!+ \texttt{FN}(n_k)) \\ \texttt{FN}(\nu a_k.n) &= \texttt{FN}(n) \setminus \{a_k\}, \end{aligned}$$

where $(\setminus)$ denotes element removal.

We equip the type $\nu$-Tree with an action by lifting the action of $\mathbb{A}$-Tree to $\nu$-Tree.

**Definition 4.4** The action $(\cdot)$ on $\nu$-Tree is defined by

$$\begin{aligned} \pi \cdot (a_k n_1 \dots n_k) &= (\pi a_k)(\pi \cdot n_1) \dots (\pi \cdot n_k) \\ \pi \cdot (\nu a_k.n) &= \nu(\pi a_k).(\pi \cdot n) \end{aligned}$$

**Remark** Recall that we require permutations to be rank preserving. Well-rankedness is therefore preserved.

Now we show that $(\cdot)$ on $\nu$-Tree does indeed satisfy the properties of a sym action, and hence $\nu$-Tree with $(\cdot)$ is a sym type.

**Fact 4.5** $(\cdot)$ is a sym action for $\nu$-Tree.

**Proof** The only difference between $\mathbb{A}$-Tree and $\nu$-Tree is one additional case. The first case when using structural induction on a $\nu$-tree is analogous to pure trees. Therefore we only consider the $\nu$ case.

$$\begin{aligned} &\mathrm{id} \cdot (\nu a_k.n) \\ =\ &\nu\ \mathrm{id}(a_k).(\mathrm{id} \cdot n) \\ \overset{\mathrm{IH}}{=}\ &(\nu a_k.n) \end{aligned} \qquad \begin{aligned} &(\pi \circ \pi') \cdot (\nu a_k.n) \\ =\ &\nu(\pi \circ \pi')(a_k).((\pi \circ \pi') \cdot n) \\ \overset{\mathrm{IH}}{=}\ &\nu(\pi(\pi'(a_k)).(\pi \cdot (\pi' \cdot n)) \\ =\ &\pi \cdot (\pi' \cdot (\nu a_k.n)) \end{aligned}$$

□

We now show for any $\nu$-tree $n$, that two permutations agreeing on $\texttt{Name}(n)$ result in the same tree when applied to $n$. This result is analogous to Lemma 3.6.

**Lemma 4.6** $(\forall a \in \texttt{Name}(n).\ \pi(a) = \pi'(a)) \to \pi \cdot n = \pi' \cdot n.$

**Proof** We prove this claim by structural induction on the $\nu$-tree $n$. The first case is identical to pure trees, therefore we only consider the case of a $\nu$-node.

$$\begin{aligned} &\pi \cdot (\nu a_k.n) \\ =\ &\nu(\pi(a_k).\pi \cdot n) \\ \overset{\mathrm{IH}}{=}\ &\nu(\pi'(a_k).\pi' \cdot n) \\ =\ &\pi' \cdot (\nu a_k.n) \end{aligned}$$

□

With this lemma we now show that the sym type $\nu$-Tree with $(\cdot)$ is a nominal type. We use the same argument that we made for $\mathbb{A}$-Tree by choosing $\texttt{Name}(n)$ as the finite support.

**Fact 4.7** $\nu$-Tree with the action $(\cdot)$ is a nominal type.

**Proof** Let $\pi$ be any permutation that fixes $\mathtt{Name}(n)$. On $\mathtt{Name}(n)$ the permutation $\pi$ agrees with the identity permutation. By Lemma 4.6 and the property of the identity permutation in a sym type (Fact 4.5):

$$\pi \cdot n = id \cdot n = n \qquad \square$$

## 4.2 $\nu$-Tree Language

The language of a $\nu$-tree is a class of pure trees. The idea is that a $\nu$-tree represents all pure trees of the same structure where the bound names are instantiated with fresh names. For freshness we consider the path above and the subtree below a node. A name is fresh if it is different from all the free names and also different from the instantiations of all the other bindings in that part of the $\nu$-tree. Any name satisfying those conditions may be used to instantiate the binding. It is worth mentioning that it is a design choice not to consider names that appear in different paths of the $\nu$-tree at all. It is familiar when considering programming languages where variables on different branches are allowed to be declared with the same name. It is also a very natural choice since the language of a $\nu$-tree depends on its children and this way the children do not depend on each other. In the following example $c$ is a free name, $a$ and $b$ are bound. Therefore $a$ and $b$ are instantiated with any name from the alphabet $\mathbb{A}$ except $c$. They are bound on the same path, so their instantiation has to be different. The rank is left implicit in the examples to improve readability.

$$\begin{array}{ccc} & \nu a & \\ & | & \\ & \nu b & \\ & | & \\ & c & \\ / & & \backslash \\ a & & b \end{array} \qquad \rightsquigarrow \qquad \left\{ \begin{array}{ccc} & c & \\ a & & b \end{array} \;\middle|\; a, b \in \mathbb{A} \setminus \{c\}; a \neq b \right\}$$

In the next example $a$ and $b$ are bound in different paths. Therefore they may be instantiated with the same name which results in a different language.

$$\begin{array}{ccc} & c & \\ / & & \backslash \\ \nu a & & \nu b \\ | & & | \\ a & & b \end{array} \qquad \rightsquigarrow \qquad \left\{ \begin{array}{ccc} & c & \\ a & & b \end{array} \;\middle|\; a, b \in \mathbb{A} \setminus \{c\} \right\}$$

Specifically, trees of the form $caa$, where the names of the two child nodes is identical, are only in the language of the second $\nu$-tree but not in the language of the first. It can be seen from these examples that the location of a binding matters since it affects the freshness conditions.

### 4.2.1 Definition

We now formalize the language as a class of pure trees $[\![-]\!]$. The definition follows Kirst's definition in [11]. It is compositional and uses the structure provided by $\nu$-Tree. The advantage of this definition is that it enables inductive proofs. However, we need an additional argument, a list of blocked names $A$ carrying all the names that have been used already in the path above.

**Definition 4.8** The **language** of a $\nu$-tree $[\![-]\!] : \mathscr{L}(\mathbb{A}) \to \nu\text{-Tree} \to \mathbb{A}\text{-Tree} \to$ `Prop` is defined inductively by

$$\frac{\forall\, 1 \leq i \leq k.\ t_i \in [\![n_i]\!]_{a_k::A}}{a_k t_1 \dots t_k \in [\![a_k n_1 \dots n_k]\!]_A}$$

$$\frac{t \in [\![(a_k b_k) \cdot n]\!]_{b_k::A} \qquad b_k \notin A \quad b_k \notin \mathtt{FN}(\nu a_k.n)}{t \in [\![\nu a_k.n]\!]_A}$$

**Remark** $(a_k b_k)$ denotes the transposition of $a_k$ and $b_k$ which is the permutation that swaps $a_k$ and $b_k$ and leaves all other names untouched.
We write $[\![n]\!]$ for the language under the empty list of names $[\![n]\!]_{[]}$.

The definition is set up such that the first inductive rule matches exactly the tree structure. Specifically, the name $a_k$ at the top has to match, and all children $t_i$ have to be in the language of $n_i$, while remembering $a_k$ as a used name in that path.
In the rule for the $\nu$ case we instantiate the name in the binding $a_k$ with some name $b_k$ by removing the $\nu$ and applying the transposition $(a_k b_k)$. The name $b_k$ has to satisfy the freshness conditions, that is, $b_k$ has not been used already on the path above and it is also not a free name in the $\nu$-tree below.
The list $A$ carries exactly the names that have been used in the path above, both the free names and the instantiations for bindings. Names from different paths are not added to $A$.
Note that the language of a $\nu$-tree is never empty. For any $\nu$-tree there is a pure tree that has the same structure. Furthermore $\mathbb{A}$ is infinite. The freshness conditions however impose only finitely many restrictions on a $\nu$-binding. Therefore there are always infinitely many valid instantiations for any binding in $n$.

### 4.2.2 Equivariance

Now we study how the language of a $\nu$-tree $[\![n]\!]$ behaves in the context of permutations on $n$. The first result is equivariance. Intuitively, this result makes sense, because the definition of the language does not depend on any specific name in the $\nu$-tree, it only depends on the structure.
In order to prove equivariance of $[\![-]\!]$, we first need equivariance of $\mathtt{FN}(-)$.

**Lemma 4.9** $\pi \cdot \texttt{FN}(n) = \texttt{FN}(\pi \cdot n)$

**Proof** We prove this claim by structural induction on the $\nu$-tree $n$.

$$\begin{aligned}
&\pi \cdot \texttt{FN}(a_k n_1 \dots n_k) \\
&= \pi \cdot (a_k :: (\texttt{FN}(n_1) +\!\!+ \dots +\!\!+ \texttt{FN}(n_k))) \\
&= \pi(a_k) :: (\pi \cdot \texttt{FN}(n_1) +\!\!+ \dots +\!\!+ \pi \cdot \texttt{FN}(n_k))) \\
&\overset{\text{IH}}{=} \pi(a_k) :: (\texttt{FN}(\pi \cdot n_1) +\!\!+ \dots +\!\!+ \texttt{FN}(\pi \cdot n_k))) \\
&= \texttt{FN}(\pi \cdot (a_k n_1 \dots n_k))
\end{aligned}$$

$$\begin{aligned}
&\pi \cdot \texttt{FN}(\nu a_k.n) \\
&= \pi \cdot (\texttt{FN}(n) \setminus \{a_k\}) \\
&= (\pi \cdot \texttt{FN}(n)) \setminus \{\pi(a_k)\} \\
&\overset{\text{IH}}{=} \texttt{FN}(\pi \cdot n) \setminus \{\pi(a_k)\} \\
&= \texttt{FN}(\pi \cdot (\nu a_k.n))
\end{aligned}$$

□

**Theorem 4.10** $\pi \cdot [\![n]\!]_A \equiv [\![\pi \cdot n]\!]_{\pi \cdot A}$

**Proof** We prove this claim by structural induction on the $\nu$-tree $n$.

$$\begin{aligned}
&\pi \cdot (a_k t_1 \dots t_k) \in \pi \cdot [\![a_k n_1 \dots n_k]\!]_A \\
&\Leftrightarrow (a_k t_1 \dots t_k) \in [\![a_k n_1 \dots n_k]\!]_A \\
&\overset{\text{def}}{\Leftrightarrow} t_i \in [\![n_i]\!]_{a_k :: A} \\
&\overset{\text{IH}}{\Leftrightarrow} \pi \cdot t_i \in [\![\pi \cdot n_i]\!]_{\pi \cdot (a_k :: A)} \\
&\overset{\text{def}}{\Leftrightarrow} \pi(a_k)(\pi \cdot t_1 \dots \pi \cdot t_k) \in [\![\pi(a_k)(\pi \cdot n_1 \dots \pi \cdot n_k)]\!]_{\pi \cdot A} \\
&\Leftrightarrow \pi \cdot (a_k t_1 \dots t_k) \in [\![\pi \cdot (a_k n_1 \dots n_k)]\!]_{\pi \cdot A}
\end{aligned}$$

Therefore: $\pi \cdot [\![a_k n_1 \dots n_k]\!]_A \equiv [\![\pi \cdot (a_k n_1 \dots n_k)]\!]_{\pi \cdot A}$

$$\begin{aligned}
&\pi \cdot t \in \pi \cdot [\![\nu a_k.n]\!]_A \\
&\Leftrightarrow t \in [\![\nu a_k.n]\!]_A \\
&\overset{\text{def}}{\Leftrightarrow} t \in [\![(a_k b_k) \cdot n]\!]_{b_k :: A} \wedge b_k \notin A \wedge b_k \notin \texttt{FN}(\nu a_k.n) \\
&\overset{\text{IH}}{\Leftrightarrow} \pi \cdot t \in [\![\pi \cdot ((a_k b_k) \cdot n)]\!]_{\pi \cdot (b_k :: A)} \wedge b_k \notin A \wedge b_k \notin \texttt{FN}(\nu a_k.n)
\end{aligned}$$

Now we use that FN is equivariant (Lemma 4.9).

$$\begin{aligned}
&b_k \notin \texttt{FN}(\nu a_k.n) \\
&\Leftrightarrow \pi(b_k) \notin \pi \cdot (\texttt{FN}(\nu a_k.n)) \\
&\Leftrightarrow \pi(b_k) \notin \texttt{FN}(\pi \cdot (\nu a_k.n))
\end{aligned}$$

Therefore we have

$$\begin{aligned}
&\pi \cdot t \in [\![\pi \cdot ((a_k b_k) \cdot n)]\!]_{\pi(b_k) :: (\pi \cdot A)} \\
&\qquad \wedge \pi(b_k) \notin \pi \cdot A \wedge \pi(b_k) \notin \texttt{FN}(\pi \cdot (\nu a_k.n))
\end{aligned}$$

We also swap the permutations applied to $n$, specifically from $\pi \cdot ((a_k b_k) \cdot n)$ to $(\pi(a_k)\pi(b_k)) \cdot (\pi \cdot n)$. It is easy to see that both permutations behave the same, therefore swapping them is justified by Lemma 4.6.

$$\begin{aligned} &\pi \cdot t \in [\![(\pi(a_k)\pi(b_k)) \cdot (\pi \cdot n)]\!]_{\pi(b_k)::(\pi \cdot A)} \\ &\qquad \wedge \pi(b_k) \notin \pi \cdot A \wedge \pi(b_k) \notin \mathtt{FN}(\pi \cdot (\nu a_k.n)) \\ \overset{\text{def}}{\Leftrightarrow}\ &\pi \cdot t \in [\![\pi \cdot (\nu a_k.n)]\!]_{\pi \cdot A} \end{aligned}$$

Therefore: $\pi \cdot [\![\nu a_k.n]\!]_A \equiv [\![\pi \cdot (\nu a_k.n)]\!]_{\pi \cdot A}$ □

# Chapter 5

# Equivalence Laws

Since the language of a $\nu$-tree is a class of pure trees, it is natural to ask when those classes are equivalent, that is, when two different $\nu$-trees have the same language. In this chapter we establish laws of the form $[\![n]\!]_A \equiv [\![n']\!]_A$. For simplicity, we also write $n \equiv n' := \forall\, A.\; [\![n]\!]_A \equiv [\![n']\!]_A$.
First we show a general result that the language is invariant under a certain class of permutations. Names under a binding in a $\nu$-tree and also names that do not appear anywhere in the $\nu$-tree do not really matter. Only free names are fixed constants. Permutations which fix free names and only permute names of the other kinds have no impact on the language of a $\nu$-tree.

$$\pi \in \texttt{fix}(\texttt{FN}(n)) \to n \equiv \pi \cdot n$$

There are four additional equivalence laws satisfied by $\nu$-tree languages which are very similar to the nominal axioms for $\nu$-words by Gabbay [7]. These nominal axioms have been proposed in the context of the nominal Kleene algebra. We only consider the four axioms for $\nu$-words without Kleene operators:

$$\begin{aligned} b \notin \texttt{FN}(x) &\to \nu a.x = \nu b.\ (ab) \cdot x \\ \nu a.\nu b.x &= \nu b.\nu a.x \\ a \notin \texttt{FN}(x) &\to \nu a.x = x \\ a \notin \texttt{FN}(x) &\to x(\nu a.y) = \nu a.xy \end{aligned}$$

We can generalize these axioms from $\nu$-words to $\nu$-trees. The first axiom states that the names of $\nu$-bindings are irrelevant.

$$b \notin \texttt{FN}(n) \to \nu a_k.n \equiv \nu b.(a_k b_k) \cdot n$$

This means we can use any other name for the binding and its bound names provided it does not clash with the free names in the $\nu$-tree. As it turns out, this axiom is just an special case of the equivalence law above.
The second axiom states that the order of two successive $\nu$-bindings does not matter. They can be swapped without changing the language.

$$\nu a_k.\nu b_l.n \equiv \nu b_l.\nu a_k.n$$

The idea behind this equivalence is that the freshness conditions stay the same when instantiating successive bindings in different order. Since the freshness conditions are the same, the bindings in the left tree can be instantiated in exactly the same way as the bindings in the right tree. Therefore the $\nu$-tree language is the same.
The third axiom states that $\nu$-bindings which do not bind any name can be ignored.

$$a \notin \mathtt{FN}(n) \rightarrow \nu a_k.n \equiv n$$

The idea behind this equivalence is simple. If a $\nu$-binding does not bind a name, then instantiating the binding does not change the $\nu$-tree.
The three axioms above are straightforward generalizations from $\nu$-words to $\nu$-trees. Their formulation for words and trees is in fact identical, because we did not use the branching tree structure. This is different for the fourth and last nominal axiom. It states that the language stays the same when moving a $\nu$-binding inside a word provided we do not change which names are in the scope of the binding. Moving a $\nu$-binding inside a tree is not a straightforward generalization from words. It requires some additional considerations which we leave to Section 5.5.
With these equivalence laws for $\nu$-tree languages we have taken significant steps towards a decision procedure for $[\![n]\!]_A \equiv [\![n']\!]_A$. This decision procedure would use these equivalence laws to compute a normal form for $\nu$-trees as outlined in Section 5.6.

## 5.1 ρ-Renaming

First we study whether the language $[\![n]\!]_A$ is closed under some permutations $\pi$, that is, whether $[\![\pi \cdot n]\!]_A \equiv [\![n]\!]_A$. This does not hold for all permutations, since for instance the $\nu$-tree with a single node $(a_0)$ has the same language as $\pi \cdot (a_0)$ only if $\pi$ fixes $a_0$. More generally, the language of any $\nu$-tree $n$ is closed under $\pi \in \mathtt{fix}(\mathtt{FN}(n))$. We call such a permutation a **ρ-renaming** of $n$. We say two $\nu$-trees $n, n'$ are **ρ-equivalent** if there exists a $\rho$-renaming $\pi$, such that $\pi \cdot n = n'$. In that case we write $n \approx_\rho n'$.

**Theorem 5.1** $\pi \in \mathtt{fix}(\mathtt{FN}(n)) \rightarrow [\![\pi \cdot n]\!]_A \equiv [\![n]\!]_A$

**Proof** We prove this claim by structural induction on the $\nu$-tree $n$.

$$\begin{aligned} & (a_k t_1 \dots t_k) \in [\![\pi \cdot (a_k n_1 \dots n_k)]\!]_A \\ \Leftrightarrow\ & (a_k t_1 \dots t_k) \in [\![\pi(a_k)(\pi \cdot n_1) \dots (\pi \cdot n_k)]\!]_A \end{aligned}$$

Since $a_k$ is at the top of the $\nu$-tree, it is a free name. Therefore it is fixed by $\pi$, and we have $\pi(a_k) = a_k$.

$$\begin{aligned} & (a_k t_1 \dots t_k) \in [\![a_k(\pi \cdot n_1) \dots (\pi \cdot n_k)]\!]_A \\ \overset{\text{def}}{\Leftrightarrow}\ & t_i \in [\![\pi \cdot n_i]\!]_{a_k::A} \\ \overset{\text{IH}}{\Leftrightarrow}\ & t_i \in [\![n_i]\!]_{a_k::A} \\ \overset{\text{def}}{\Leftrightarrow}\ & (a_k t_1 \dots t_k) \in [\![(a_k n_1 \dots n_k)]\!]_A \end{aligned}$$

$$\text{Therefore: } [\![\pi \cdot (a_k n_1 \dots n_k)]\!]_A \equiv [\![a_k n_1 \dots n_k]\!]_A$$

$$\begin{aligned} & t \in [\![\pi \cdot (\nu a_k.n)]\!]_A \\ \Leftrightarrow\ & t \in [\![\nu\pi(a_k).(\pi \cdot n)]\!]_A \\ \overset{\text{def}}{\Leftrightarrow}\ & t \in [\![(\pi(a_k) b_k) \cdot (\pi \cdot n)]\!]_{b_k::A} \wedge b_k \notin A \wedge b_k \notin \mathtt{FN}(\nu\pi(a_k).(\pi \cdot n)) \end{aligned}$$

Now we use Lemma 4.9 that $\mathtt{FN}$ is equivariant.

$$t \in [\![(\pi(a_k) b_k) \cdot (\pi \cdot n)]\!]_{b_k::A} \wedge b_k \notin A \wedge b_k \notin \pi \cdot \mathtt{FN}(\nu a_k.n)$$

In order to use the inductive hypothesis, we now need to swap the first permutation applied to $n$, from $\pi \cdot n$ to $((\pi(a_k)\pi(b_k)) \cdot (\pi \cdot ((a_k b_k) \cdot n)))$. This chain of permutations transposes $a_k$ and $b_k$, applies $\pi$, and finally transposes the image of $a_k$ and $b_k$ under $\pi$ back again. So both permutations behave the same and swapping them is justified by Lemma 4.6.

$$\begin{aligned} & t \in [\![(\pi(a_k) b_k) \cdot ((\pi(a_k)\pi(b_k)) \cdot (\pi \cdot ((a_k b_k) \cdot n)))]\!]_{b_k::A} \wedge b_k \notin A \wedge b_k \notin \pi \cdot \mathtt{FN}(\nu a_k.n) \\ \overset{\text{IH}}{\Leftrightarrow}\ & t \in [\![(a_k b_k) \cdot n]\!]_{b_k::A} \wedge b_k \notin A \wedge b_k \notin \pi \cdot \mathtt{FN}(\nu a_k.n) \end{aligned}$$

Now we use that $\pi$ fixes $\mathtt{FN}(\nu a_k.n)$.

$$\begin{aligned} & t \in [\![(a_k b_k) \cdot n]\!]_{b_k::A} \wedge b_k \notin A \wedge b_k \notin \mathtt{FN}(\nu a_k.n) \\ \overset{\text{def}}{\Leftrightarrow}\ & t \in [\![(\nu a_k.n)]\!]_A \end{aligned}$$

$$\text{Therefore: } [\![\pi \cdot (\nu a_k.n)]\!]_A \equiv [\![\nu a_k.n]\!]_A$$

□

**Remark** If a $\nu$-tree $n$ does not contain any free names, then we call $n$ **closed** and every permutation $\pi$ is a $\rho$-renaming of $n$. In that case the class $[\![n]\!]$ is equivariant, that is, $[\![n]\!] \equiv \pi \cdot [\![n]\!]$.

An immediate consequence of Theorem 5.1 is the irrelevance of the concrete names in $\nu$-bindings which is one of the nominal axioms in [7]. Consider $[\![\nu a_k.n]\!]$. Since $a_k$ is instantiated, we could indeed use any other name $b_k$ for the binding and the corresponding bound names, unless $b_k \in \mathtt{FN}(\nu a_k.n)$, since free names act like constants and must stay fixed.

**Corollary 5.2** $b_k \notin \mathtt{FN}(\nu a_k.n) \rightarrow [\![\nu a_k.n]\!]_A \equiv [\![\nu b_k.\ (a_k b_k) \cdot n]\!]_A$

**Proof** We have $b_k \notin \mathtt{FN}(\nu a_k.n)$ by assumption and $a_k \notin \mathtt{FN}(\nu a_k.n)$ since it is bound at the top. The transposition $(a_k b_k)$ is therefore a $\rho$-renaming for $\nu a_k.n$. Using Theorem 5.1 we get

$$[\![\nu a_k.n]\!]_A \equiv [\![(a_k b_k) \cdot \nu a_k.n]\!]_A \equiv [\![\nu b_k.\ (a_k b_k) \cdot n]\!]_A \qquad \square$$

## 5.2 Swapping bindings

Next we show that the language of a $\nu$-tree does not change when swapping the position of two successive $\nu$-bindings, that is, there are no conflicts when instantiating in a different order. This equivalence is founded with a simple intuition. The freshness conditions for the $\nu$-bindings ensure that both instantiations are different from free names in subtree and also that both instantiations are different from each other. One can see that the freshness conditions stay the same when swapping the bindings.

**Theorem 5.3** $[\![\nu a_k.\nu b_l.n]\!]_A \equiv [\![\nu b_l.\nu a_k.n]\!]_A$

**Proof** The formal proof however requires considerable effort since we need to show that the possible instantiations for both bindings stay the same. For that we essentially need to transform the freshness conditions for the bindings into each other.
We only consider the non trivial case where $a \neq b$. We also assume the bindings to be of the same rank, $\text{rk}(a) = k = \text{rk}(b)$, since there are no conflicts between instantiations of different rank.

$$\begin{aligned} & t \in [\![\nu a_k.\nu b_k.n]\!]_A \\ \overset{\text{def}}{\Leftrightarrow}\ & t \in [\![(a_k c_k) \cdot (\nu b_k.n)]\!]_{c_k::A} \wedge c_k \notin A \wedge c_k \notin \mathtt{FN}(\nu a_k.\nu b_k.n) \\ \Leftrightarrow\ & t \in [\![\nu (a_k c_k)(b_k).(a_k c_k) \cdot n]\!]_{c_k::A} \wedge c_k \notin A \wedge c_k \notin \mathtt{FN}(\nu a_k.\nu b_k.n) \end{aligned}$$

Let $b'_k := (a_k c_k)(b_k)$.

$$t \in [\![\nu b'_k.(a_k c_k) \cdot n]\!]_{c_k::A} \wedge c_k \notin A \wedge c_k \notin \mathtt{FN}(\nu a_k.\nu b_k.n)$$

$$\overset{\text{def}}{\Leftrightarrow} t \in [\![(b'_k d_k) \cdot (a_k c_k) \cdot n]\!]_{d_k::c_k::A} \wedge c_k \notin A \wedge d_k \notin (c_k :: A)$$
$$\wedge\, c_k \notin \mathtt{FN}(\nu a_k.\nu b_k.n) \wedge d_k \notin \mathtt{FN}(\nu b'_k.(a_k c_k) \cdot n)$$

Since $d_k \notin (c_k :: A)$ we have $d_k \neq c_k$ and therefore $c_k \notin (d_k :: A)$.
Let $a'_k := (b_k d_k)(a_k)$.

$$t \in [\![(a'_k c_k) \cdot (b_k d_k) \cdot n)]\!]_{c_k::d_k::A} \wedge c_k \notin (d_k :: A) \wedge d_k \notin A$$
$$\wedge\, c_k \notin \mathtt{FN}(\nu a'_k.(b_k d_k) \cdot n) \wedge d_k \notin \mathtt{FN}(\nu b'_k.(a_k c_k) \cdot n)$$

$$\overset{\text{def}}{\Leftrightarrow} t \in [\![\nu a'_k.(b_k d_k) \cdot n)]\!]_{d_k::A} \wedge d_k \notin A \wedge d_k \notin \mathtt{FN}(\nu b'_k.(a_k c_k) \cdot n)$$
$$\Leftrightarrow t \in [\![\nu (b_k d_k)(a_k).(b_k d_k) \cdot n)]\!]_{d_k::A} \wedge d_k \notin A \wedge d_k \notin \mathtt{FN}(\nu b'_k.(a_k c_k) \cdot n)$$
$$\Leftrightarrow t \in [\![(b_k d_k) \cdot (\nu a_k.n)]\!]_{d_k::A} \wedge d_k \notin A \wedge d_k \notin \mathtt{FN}((a_k c_k) \cdot \nu b_k.n)$$

We have $d_k \notin \mathtt{FN}((a_k c_k) \cdot \nu b_k.n)$. Using equivariance of FN (Lemma 4.9) we get $(a_k c_k)(d_k) \notin (\mathtt{FN}(n) \setminus \{b_k\})$.
Therefore we have also $(a_k c_k)(d_k) \notin (\mathtt{FN}(n) \setminus \{a_k, b_k\})$, and we want to show $d_k \notin (\mathtt{FN}(n) \setminus \{a_k, b_k\})$. There are three cases for the transposition $(a_k c_k)(d_k)$. First, if $d_k = a_k$, then we clearly have $d_k \notin (\mathtt{FN}(n) \setminus \{a_k, b_k\})$. The second case, $d_k = c_k$, is a contradiction since we have $d_k \neq c_k$. And otherwise we simply get $d_k \notin (\mathtt{FN}(n) \setminus \{a_k, b_k\})$.
Therefore we have in all cases that $d_k \notin (\mathtt{FN}(n) \setminus \{a_k, b_k\})$ which is by definition $d_k \notin \mathtt{FN}(\nu b_k.\nu a_k.n)$.

$$t \in [\![(b_k d_k) \cdot (\nu a_k.n)]\!]_{d_k::A} \wedge d_k \notin A \wedge d_k \notin \mathtt{FN}(\nu b_k.\nu a_k.n)$$
$$\overset{\text{def}}{\Leftrightarrow} t \in [\![\nu b_k.\nu a_k.n]\!]_A \qquad \square$$

## 5.3 Weakening and Strengthening

The language of $n$ is constrained by the list of blocked names $A$ which are not used to instantiate any bindings in $n$. Therefore the language becomes larger when removing a name $c$ from $A$ since this removes a constraint on the bindings. We call this **weakening**.

**Lemma 5.4** $t \in [\![n]\!]_{c_k::A} \to t \in [\![n]\!]_A$

**Proof** We prove this claim by induction on $[\![-]\!]$.

$$\begin{aligned}
& a_k t_1 \dots t_k \in [\![a_k n_1 \dots n_k]\!]_{c_k::A} \\
\overset{\text{def}}{\Leftrightarrow}\ & t_i \in [\![n_i]\!]_{a_k::c_k::A} \\
\Leftrightarrow\ & t_i \in [\![n_i]\!]_{c_k::a_k::A} \\
\overset{\text{IH}}{\Rightarrow}\ & t_i \in [\![n_i]\!]_{a_k::A} \\
\overset{\text{def}}{\Leftrightarrow}\ & a_k t_1 \dots t_k \in [\![a_k n_1 \dots n_k]\!]_A
\end{aligned}$$

$$\begin{aligned}
& t \in [\![\nu a_k.n]\!]_{c_k::A} \\
\overset{\text{def}}{\Leftrightarrow}\ & t \in [\![(a_k b_k) \cdot n]\!]_{b_k::c_k::A} \wedge b_k \notin (c_k :: A) \wedge b_k \notin \mathtt{FN}(\nu a_k.n) \\
\Rightarrow\ & t \in [\![(a_k b_k) \cdot n]\!]_{c_k::b_k::A} \wedge b_k \notin A \wedge b_k \notin \mathtt{FN}(\nu a_k.n) \\
\overset{\text{IH}}{\Leftrightarrow}\ & t \in [\![(a_k b_k) \cdot n]\!]_{b_k::A} \wedge b_k \notin A \wedge b_k \notin \mathtt{FN}(\nu a_k.n) \\
\overset{\text{def}}{\Leftrightarrow}\ & t \in [\![\nu a_k.n]\!]_A
\end{aligned}$$ □

Next we now show a lemma which states that free names $a_k \in \mathtt{FN}(n)$ are not instantiated or permuted, that is, free names in $n$ have to appear in any tree $t \in [\![n]\!]$. In other words, free names behave like constants.

**Lemma 5.5** $c_k \in \mathtt{FN}(n) \to t \in [\![n]\!]_A \to c_k \in \mathtt{Name}(t)$

**Proof** We prove this claim by induction on $[\![-]\!]$.
In the first case we have $c_k \in \mathtt{FN}(a_k n_1 \dots n_k)$. If $a_k = c_k$, then we have immediately $c_k \in \mathtt{Name}(a_k t_1 \dots t_k)$. Otherwise there exists a $j \leq k$, such that

$$\begin{aligned}
& c_k \in \mathtt{FN}(n_j) \\
\overset{\text{IH}}{\Leftrightarrow}\ & c_k \in \mathtt{Name}(t_j) \\
\Leftrightarrow\ & c_k \in \mathtt{Name}(a_k t_1 \dots t_k)
\end{aligned}$$

In the second case we have $c_k \in \mathtt{FN}(\nu a_k n)$, that is, $c_k \in \mathtt{FN}(n) \wedge c_k \neq a_k$. There also exists an instantiation for the binding $b_k \notin \mathtt{FN}(\nu a_k.n)$ by induction. Therefore $c_k \neq b_k$ and $(a_k b_k)(c_k) = c_k$.

$$\begin{aligned}
& c_k \in \mathtt{FN}(n) \\
\Leftrightarrow\ & (a_k b_k)(c_k) \in \mathtt{FN}(n) \\
\Leftrightarrow\ & c_k \in \mathtt{FN}((a_k b_k) \cdot n) \\
\overset{\text{IH}}{\Leftrightarrow}\ & c_k \in \mathtt{Name}(t)
\end{aligned}$$ □

The language of a $\nu$-tree $n$ becomes smaller when adding a name $c$ to $A$ since this adds a constraint on the instantiation of $\nu$-bindings. For a specific tree $t \in [\![n]\!]_A$ this means $t \in [\![n]\!]_{c_k::A}$ only if all bindings can still be instantiated in the right way when adding $c$. Consider for example the $\nu$-tree $\nu a_0.a_0$. Then $b_0 \in [\![\nu a_0.a_0]\!]_{[]}$. However $b_0 \notin [\![\nu a_0.a_0]\!]_{[b_0]}$.
It can be seen from the example that we cannot add a name $c$ to $A$ if $c \in \texttt{Name}(t)$, since we might need this $c$ to instantiate a binding. We now show that if $c \notin \texttt{Name}(t)$, then $c$ can be added to $A$ and all instantiations are still possible. We call this **strengthening**.

**Lemma 5.6** $t \in [\![n]\!]_A \to c \notin \texttt{Name}(t) \to t \in [\![n]\!]_{c_k::A}$

**Proof** We prove this claim by induction on $[\![-]\!]$.

$$\begin{aligned}
& a_k t_1 \dots t_k \in [\![a_k n_1 \dots n_k]\!]_A \\
\overset{\text{def}}{\Leftrightarrow}\ & t_i \in [\![n_i]\!]_{a_k::A} \\
\overset{\text{IH}}{\Rightarrow}\ & t_i \in [\![n_i]\!]_{c_k::a_k::A} \\
\Leftrightarrow\ & t_i \in [\![n_i]\!]_{a_k::c_k::A} \\
\overset{\text{def}}{\Leftrightarrow}\ & a_k t_1 \dots t_k \in [\![a_k n_1 \dots n_k]\!]_{c_k::A}
\end{aligned}$$

This concludes the tree case. In the $\nu$ case we have

$$\begin{aligned}
& t \in [\![\nu a_k.n]\!]_A \\
\overset{\text{def}}{\Leftrightarrow}\ & t \in [\![(a_k b_k) \cdot n]\!]_{b_k::A} \land b_k \notin (A) \land b_k \notin \texttt{FN}(\nu a_k.n)
\end{aligned}$$

For the binding $\nu a_k$ we have some instantiation $b_k$. Now we first consider the case (1) that $b_k \neq c_k$. This is the case where we add a name to the $A$ which is different from the instantiation.

$$\begin{aligned}
& t \in [\![(a_k b_k) \cdot n]\!]_{b_k::A} \land b_k \notin A \land b_k \neq c_k \land b_k \notin \texttt{FN}(\nu a_k.n) \\
\Leftrightarrow\ & t \in [\![(a_k b_k) \cdot n]\!]_{b_k::A} \land b_k \notin (c_k :: A) \land b_k \notin \texttt{FN}(\nu a_k.n) \\
\overset{\text{IH}}{\Rightarrow}\ & t \in [\![(a_k b_k) \cdot n]\!]_{b_k::c_k::A} \land b_k \notin (c_k :: A) \land b_k \notin \texttt{FN}(\nu a_k.n) \\
\overset{\text{def}}{\Leftrightarrow}\ & t \in [\![\nu a_k.n]\!]_{c_k::A}
\end{aligned}$$

If $b_k = c_k$ we further consider case (2) that $b_k \in \texttt{FN}(n)$. Then $b_k = a_k$ since we also have $b_k \notin \texttt{FN}(\nu a_k.n)$. Since $b_k = a_k$, we have $a_k \in \texttt{FN}(n)$ and $a_k \notin \texttt{Name}(t)$. However, we have $t \in [\![(a_k b_k) \cdot n]\!]_{b_k::A}$ by induction, that is, $t \in [\![n]\!]_{a_k::A}$. We get a contradiction by Lemma 5.5 which concludes this case.
Otherwise if $b_k \notin \texttt{FN}(n)$, we further distinguish $a_k$. We now consider case (3) that $a_k \in \texttt{FN}(n)$. then $(a_k b_k)(a_k) \in \texttt{FN}(n)$. With $t \in [\![(a_k b_k) \cdot n]\!]_{b_k::A}$ and Lemma 5.5 we get $b_k \in \texttt{Name}(t)$, and since $c_k \notin \texttt{Name}(t)$, we get $b_k \neq c_k$ which is a contradiction.

Finally we have the case (4) where $b_k \notin \mathtt{FN}(n)$ and $a_k \notin \mathtt{FN}(n)$. In this case we show that we could have chosen a different instantiation than $b_k$. We choose a fresh name $d_k$ constructed by Lemma 3.1, which is fresh relative to the list $(b_k :: A) ++ \mathtt{Name}(\nu a.n) ++ \mathtt{Name}(t)$. Now that $b_k, a_k,$ and $d_k \notin \mathtt{FN}(n)$, the transpositions $(a_k b_k)$ and $(a_k d_k)$ are both $\rho$-renamings for $n$.

$$\begin{aligned} & t \in [\![(a_k b_k) \cdot n]\!]_{b_k::A} \wedge d_k \notin (b_k :: A) \wedge d_k \notin \mathtt{Name}(t) \wedge d_k \notin \mathtt{FN}(\nu a_k.n) \\ \overset{\text{IH}}{\Leftrightarrow} & t \in [\![(a_k b_k) \cdot n]\!]_{d_k::b_k::A} \wedge d_k \notin (b_k :: A) \wedge d_k \notin \mathtt{Name}(t) \wedge d_k \notin \mathtt{FN}(\nu a_k.n) \end{aligned}$$

Note that we use the inductive hypothesis for the name $d_k$, which is justified since $d_k \notin \mathtt{Name}(t)$. With the invariance of $\nu$-tree languages under $\rho$-renamings (Theorem 5.1) we get

$$\begin{aligned} & t \in [\![(a_k b_k) \cdot n]\!]_{d_k::b_k::A} \\ \Leftrightarrow\ & t \in [\![n]\!]_{d_k::b_k::A} \\ \Leftrightarrow\ & t \in [\![(a_k d_k) \cdot n]\!]_{d_k::b_k::A} \end{aligned}$$

And therefore

$$\begin{aligned} & t \in [\![(a_k d_k) \cdot n]\!]_{d_k::b_k::A} \wedge d_k \notin (b_k :: A) \wedge d_k \notin \mathtt{Name}(t) \wedge d_k \notin \mathtt{FN}(\nu a_k.n) \\ \overset{\text{def}}{\Leftrightarrow} & t \in [\![\nu a_k.n]\!]_{d_k::A} \end{aligned}$$

□

## 5.4 Empty bindings

**Definition 5.7** For a $\nu$-tree $\nu a.n$ we call the binding **proper** if $a \in \mathtt{FN}(n)$. Then the binding $\nu a$ binds a name in $n$.
We call the binding **empty** if $a \notin \mathtt{FN}(n)$. Then the binding $\nu a$ does not bind any name in $n$.

We now show that the $[\![n]\!]$ is invariant under empty $\nu$-bindings. This equivalence makes sense intuitively. In the case that the binding is empty, $\nu a$ does not bind any name in $n$, and therefore no name will be instantiated effectively. When instantiating an empty binding, it simply vanishes without affecting the remaining $\nu$-tree.

**Theorem 5.8** $a_k \notin \mathtt{FN}(n) \to [\![\nu a_k.n]\!]_A \equiv [\![n]\!]_A$.

**Proof** We need to show $t \in [\![n]\!]_A \leftrightarrow t \in [\![\nu a_k.n]\!]_A$

- " $\to$ ": We have $t \in [\![n]\!]_A$. Let $c_k$ be a fresh name constructed by Lemma 3.1. Then we have in particular $f \notin \mathtt{FN}(n)$. Then the transposition $(a_k c_k)$ is a $\rho$-renaming for $n$. Using Theorem 5.1 and strengthening (Lemma 5.6) we get
$$t \in [\![n]\!]_A \Leftrightarrow t \in [\![(a_k c_k) \cdot n]\!]_A \Rightarrow t \in [\![(a_k c_k) \cdot n]\!]_{c_k::A}$$

Using the freshness of $c_k$ we get

$$t \in [\![(a_k c_k) \cdot n]\!]_{c_k::A} \wedge c_k \notin A \wedge c_k \notin \mathtt{FN}(\nu a_k.n) \overset{\text{def}}{\Leftrightarrow} t \in [\![\nu a_k.n]\!]_A$$

- " $\leftarrow$ ": We have

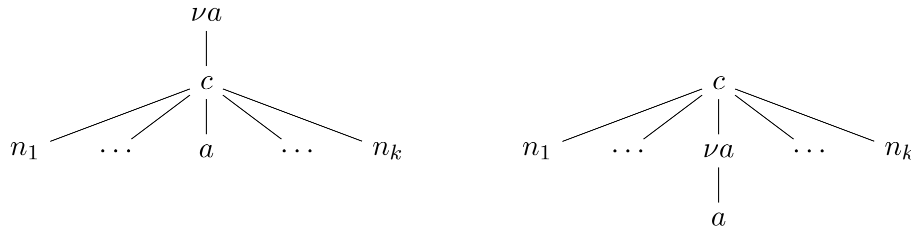
$$t \in [\![\nu a_k.n]\!]_A \overset{\text{def}}{\Leftrightarrow} t \in [\![(a_k b_k) \cdot n]\!]_{b_k::A} \wedge b_k \notin A \wedge b_k \notin \mathtt{FN}(\nu a_k.n)$$

If $a_k = b_k$ then $b_k \notin \mathtt{FN}(n)$ by assumption. If $a_k \neq b_k$, then also $b_k \notin \mathtt{FN}(n)$. The transposition $(a_k b_k)$ is therefore a $\rho$-renaming for $n$. Using Theorem 5.1 and weakening Theorem 5.4 we get

$$t \in [\![(a_k b_k) \cdot n]\!]_{b_k::A} \Leftrightarrow t \in [\![n]\!]_{b_k::A} \Rightarrow t \in [\![n]\!]_A \qquad \square$$

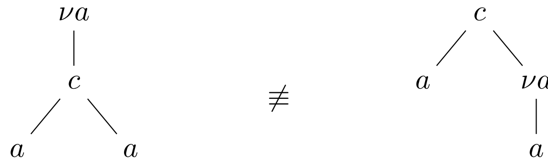
## 5.5 Pushing bindings

The last equivalence establishes how the position of one $\nu$-binding in a tree node affects the language of the $\nu$-tree. We study under which conditions a $\nu$-binding can be moved up or down inside a $\nu$-tree. Consider the following setup.

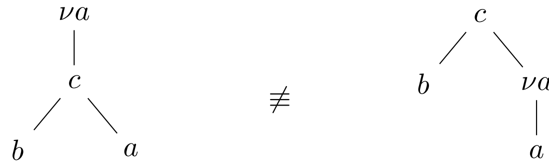



Reading the example as a transformation from left to right, we push the $\nu$-binding downwards into exactly one subtree. From right to left, we pull the $\nu$-binding to the top.

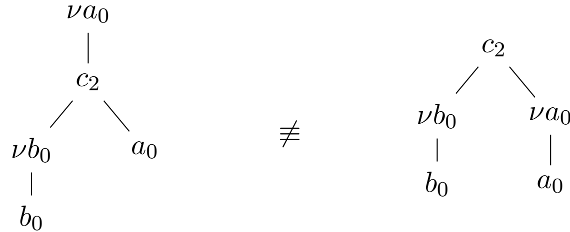
To state when these $\nu$-trees are equivalent we first investigate how the position of $\nu$-bindings has an impact on the language of a $\nu$-tree. There are three different scenarios where the position of a binding matters. The first observation is that the names in scope of a binding depend on the position of the binder. Consider the following example.



If we push the binding into either subtree the binding is lost in the other. We therefore cannot move the binding if the scope of the binding is changed. Since we can push the $\nu$-binding only into one subtree, this subtree has to contain all names that are bound by this binding, and all other subtrees must not contain the bound name.
The second scenario is a result of the freshness conditions imposed on the binding by the free names in the $\nu$-tree.




In the left $\nu$-tree there is the freshness condition that $\nu a$ must not be instantiated with $b$. In the right $\nu$-tree there is no such freshness condition because the name $b$ is not visible. Since the freshness conditions for the binding is different in both $\nu$-trees, they are not equivalent.
The third scenario is a result from the freshness conditions imposed by other $\nu$-bindings. Consider the example.




In the left $\nu$-tree there is the freshness condition that $\nu a_0$ and $\nu b_0$ must be instantiated differently. In the right $\nu$-tree there is no such freshness condition because the bindings do not lie on the same path. Therefore the $\nu$-trees are not equivalent. In this scenario the rank is significant. A binding $\nu a_k$ binds the ranked name $a_k$ and is instantiated with names of rank $k$. Bindings of different rank are therefore always instantiated differently independent of their position. Therefore, the position of bindings only matters for bindings of the same rank.
In conclusion, a $\nu$-binding may only be moved if the scope and the freshness conditions stay the same. With these considerations we now state the equivalence law which needs three assumptions to ensure that none of the three scenarios above can occur.

**Theorem 5.9** *Let $j$ be the index of the subtree into which we want to push the binding.*

$$
\begin{aligned}
&(\forall\, l \neq j.\ a_k \notin \mathtt{FN}(n_l)) \\
\to\ &\mathtt{FN}(\nu a_k.b_k(n_1 \ldots n_j \ldots n_k)) \setminus A \subseteq \mathtt{FN}(\nu a_k.n_j) \\
\to\ &(\forall\, l \neq j.\ \nexists\, (\nu d_k.n') \in n_l) \\
\to\ &[\![\nu a_k.b_k(n_1 \ldots n_j \ldots n_k)]\!]_A \equiv [\![b_k(n_1 \ldots (\nu a_k.n_j) \ldots n_k)]\!]_A
\end{aligned}
$$

**Proof** In both directions the proof is structured in a similar way. We first use the definition of $[\![-]\!]$. Then we treat the subtree $n_j$ separately from all others $n_l$. In the direction from right to left we remove the $\nu$-binding from $n_j$ and put in on top of the entire $\nu$-tree while showing that the language of all others $n_l$ stays fixed. In the other direction we push the $\nu$-binding from the top of the $\nu$-tree to $n_j$ and show again that the language of the other subtrees $n_l$ stays fixed.

$" \leftarrow "$:

$$b_k t_1 \ldots t_k \in [\![b_k\ n_1 \ldots (\nu a_k.n_j) \ldots n_k]\!]_A$$

By definition we have $t_i \in [\![n_i]\!]_{b_k::A}$ and in particular $t_j \in [\![\nu a_k.n_j]\!]_{b_k::A}$. By definition we have a name $c_k$, such that

$$t_j \in [\![(a_k c_k).n_j]\!]_{c_k::b_k::A} \wedge c_k \notin (b_k :: A) \wedge c_k \notin \mathtt{FN}(\nu a_k.n_j)$$

Next we consider all $l \neq j$. We now show that the permutation that we used to instantiate the $\nu$-binding in $\nu a_k.n_j$ essentially does not matter for all other subtrees $n_l$. We have

$$t_l \in [\![n_l]\!]_{b_k::A}$$

By assumption we have $a_k \notin \mathtt{FN}(n_l)$. With $c_k \notin \mathtt{FN}(\nu a_k.n_j)$ and the assumption $\mathtt{FN}(\nu a_k.b_k\ n_1 \ldots n_k) \subseteq \mathtt{FN}(\nu a_k.n_j)$ we get $c_k \notin \mathtt{FN}(\nu a_k.b_k\ n_1 \ldots n_k)$, and therefore $c_k \notin \mathtt{FN}(n_l)$.
Since $a_k \notin \mathtt{FN}(n_l)$ and $c_k \notin \mathtt{FN}(n_l)$ the transposition $(a_k c_k)$ is a $\rho$-renaming for $n_l$. With Theorem 5.1 we get

$$t_l \in [\![(a_k c_k) \cdot n_l]\!]_{b_k::A}$$

The transposition $(a_k c_k)$ used to instantiate $\nu a_k.n_j$ does therefore not conflict with free names in other subtrees $n_l$. Now we only need to show that it does not conflict with any bindings in $n_l$.
When instantiating a binding of rank $k$, we only ever instantiate it with names of the same rank to preserve well-rankedness. Since $n_l$ contains no $\nu$-bindings of rank $k$ by assumption, we can add $c_k$ to the list of blocked names because it will never be used to instantiate any binding in that subtree.

$$t_l \in [\![(a_k c_k) \cdot n_l]\!]_{c_k::b_k::A}$$

We therefore have for all subtrees $1 \leq i \leq k$

$$\begin{aligned} & t_i \in [\![(a_k c_k) \cdot n_i]\!]_{c_k::b_k::A} \\ \overset{\text{def}}{\Leftrightarrow} & b_k t_1 \dots t_k \in [\![(b_k\ (a_k c_k) \cdot n_1 \dots (a_k c_k) \cdot n_j \dots (a_k c_k) \cdot n_k)]\!]_{c_k::A} \end{aligned}$$

Finally note that $b_k \neq a_k$ and $b_k \neq c_k$, and therefore $(a_k c_k)(b_k) = b_k$.

$$\begin{aligned} & b_k t_1 \dots t_k \in [\![(a_k c_k) \cdot (b_k\ n_1 \dots n_j \dots n_k)]\!]_{c_k::A} \wedge c_k \notin A \wedge c_k \notin \texttt{FN}(\nu a_k.n) \\ \overset{\text{def}}{\Leftrightarrow} & b_k t_1 \dots t_k \in [\![\nu a_k.b_k n_1 \dots n_j \dots n_k]\!] \end{aligned}$$

" $\rightarrow$ ":

$$\begin{aligned} & b_k t_1 \dots t_k \in [\![\nu a_k.b_k n_1 \dots n_j \dots n_k]\!]_A \\ \overset{\text{def}}{\Leftrightarrow} & b_k t_1 \dots t_k \in [\![(a_k c_k) \cdot (b_k\ n_1 \dots n_j \dots n_k)]\!]_{c_k::A} \wedge c_k \notin A \wedge c_k \notin \texttt{FN}(\nu a_k.b_k n_1 \dots n_k) \end{aligned}$$

Similar to the other proof direction we have $b_k \neq a_k$ and $b_k \neq c_k$, and therefore $(a_k c_k)(b_k) = b_k$.

$$\begin{aligned} & b_k t_1 \dots t_k \in [\![b_k\ (a_k c_k) \cdot n_1 \dots (a_k c_k) \cdot n_j \dots (a_k c_k) \cdot n_k]\!]_{c_k::A} \\ \overset{\text{def}}{\Leftrightarrow} & t_i \in [\![(a_k c_k) \cdot n_i]\!]_{b_k::c_k::A} \end{aligned}$$

In particular we have for $t_j$

$$t_j \in [\![(a_k c_k) \cdot n_j]\!]_{b_k::c_k::A}$$

We now show that the instantiation $c_k$ when instantiating $\nu a_k$ at the top is still valid when moving the $\nu a_k$ down and put it on top of $n_j$. We have $c_k \neq b_k$ and therefore $c_k \notin b_k :: A$.
Furthermore we have $c_k \notin \texttt{FN}(\nu a_k.b_k n_1 \dots n_k)$ and in particular $c_k \notin \texttt{FN}(\nu a_k.n_j)$. This can be seen intuitively since this is the direction where we push a $\nu$-binding downwards into a subtree. From the new position there are less free names visible to the $\nu$-binding. An instantiation that is fresh relative to the free names visible at the top is also fresh relative to the free names visible in a subtree.

$$\begin{aligned} & t_j \in [\![(a_k c_k) \cdot n_j]\!]_{b_k::c_k::A} \wedge c_k \notin b_k :: A \wedge c_k \notin \texttt{FN}(\nu a_k.n_j) \\ \overset{\text{def}}{\Leftrightarrow} & t_j \in [\![\nu a_k.n_j]\!]_{b_k::A} \end{aligned}$$

For all other $l \neq j$ we have

$$t_l \in [\![(a_k c_k) \cdot n_l]\!]_{b_k::c_k::A}$$

We have $a \notin \texttt{FN}(n_l)$ by assumption. Furthermore we have $c_k \notin \texttt{FN}(\nu a_k.b_k n_1 \dots n_k)$ and therefore in particular $c_k \notin \texttt{FN}(n_l)$. The transposition $(a_k c_k)$ is therefore a $\rho$-renaming for $n_l$. Using the invariance of $\nu$-tree languages under $\rho$-renamings (Theorem 5.1) and weakening (Lemma 5.4) we get

$$t_l \in [\![n_l]\!]_{b_k::A}$$

In conclusion we have

$$\begin{aligned} & t_1 \dots t_j \dots t_k \in [\![n_1 \dots (\nu a_k.n_j) \dots n_k]\!]_{b_k::A} \\ \overset{\text{def}}{\Leftrightarrow}\ & b_k t_1 \dots t_j \dots t_k \in [\![b_k\ n_1 \dots (\nu a_k.n_j) \dots n_k]\!]_A \end{aligned}$$ □

**Remark** Note that the proof of the direction " $\rightarrow$ " needs neither the second nor the third assumption. They are necessary to ensure that the freshness conditions are the same. However the freshness conditions if the binding is at the top, as in $\nu a_k.b_k(n_1 \dots n_j \dots n_k)$, are always a superset of the freshness conditions if the binding is in a subtree, as in $b_k(n_1 \dots (\nu a_k.n_j) \dots n_k)$. Intuitively, more is visible if the binding is at the top of the $\nu$-tree. The second and third assumption are only necessary for the equivalence to hold in the direction " $\leftarrow$ ".

Admittedly, changing the position of a $\nu$-binding is a very technical equivalence. This is largely due to our choice of freshness constraints. Freshness depends on the position of a binding. We never consider parallel paths in a tree, only the subtree and the path above. While these freshness conditions are a natural choice and enable a nice inductive definition, they make this equivalence statement more complicated. After moving a $\nu$-binding the subtree is obviously different. Therefore there might be other free names and $\nu$-bindings in it than before which would impact the freshness conditions. Here we need to put high requirements on the $\nu$-tree, such that the freshness conditions do not change effectively. Only if all the requirements are satisfied the binding may be moved. In the general case the position of a binding makes a difference for the language of a $\nu$-tree as demonstrated in the examples.

## 5.6 Towards decidability of $\nu$-tree equivalence

We now elaborate on a possible decision procedure for $[\![n]\!]_A \equiv [\![n']\!]_A$ by using the equivalence laws to normalize $\nu$-trees.

**Definition 5.10** Two $\nu$-trees $n, n'$ are called **$\alpha$-equivalent** if they only differ in bound names. We write $n \approx_\alpha n'$.

With the equivalence laws we can only normalize $\nu$-tree up to $\alpha$-equivalence. Consider the following example.

These two $\nu$-trees are $\alpha$-equivalent since they only differ in bound names, but they are not $\rho$-equivalent since no permutation applied to the whole tree can rename one $\nu$-tree into the other. Their language is the same, but there is no applicable equivalence law. This phenomenon occurs because we did not assume bindings to be distinct.

With the setup so far, a decision procedure would normalize any two $\nu$-trees in the following way. First, empty bindings are removed using Theorem 5.8. All remaining bindings are then pushed down in an ordered way as far as possible using Theorem 5.9 and Theorem 5.3. This would normalize two $\nu$-trees to two $\alpha$-equivalent $\nu$-trees if they have the same language. Then the idea is to decide $\alpha$-equivalence of the normalized $\nu$-trees to decide language equivalence.

Alternatively, one could use a procedure to make the bindings in a $\nu$-tree distinct as a final normalization step. If every binding occurs at most once then $n \approx_\rho n' \leftrightarrow n \approx_\alpha n'$. Two equivalent $\nu$-trees are therefore normalized to two $\rho$-equivalent $\nu$-trees and one would use Theorem 5.1 to rename all bindings such that they match up. Two equivalent $\nu$-trees are then exactly equal.

Showing that equivalence of $\nu$-tree languages is decidable was an optional goal of this thesis which we did not accomplish completely. The sound equivalence laws serve as a first step towards a decision procedure. It remains future work to show that the equivalence laws are complete and that the equivalence of $\nu$-tree languages is decidable.

# Chapter 6

# Decidability of $\nu$-tree languages

If a $\nu$-tree $n$ has at least one non-empty $\nu$-binding, then the class $[\![n]\!]$ is infinite, that is, there are infinitely many pure trees $t$, such that $t \in [\![n]\!]$, since there are infinitely many instantiations for the binding. In the fields software verification or XML parsing, trees over an infinite alphabet are a commonly used model and it is therefore important to analyze whether the language of that model is decidable.

## 6.1 Decision Procedure

We define by defining a decision procedure $\mathcal{D} : \mathscr{L}(\mathbb{A}) \to \nu\text{-Tree} \to \mathbb{A}\text{-Tree} \to \mathbb{B}$ that returns true iff a given pure tree is in the language of a $\nu$-tree. Just like the inductive definition of $[\![-]\!]$, the procedure $\mathcal{D}$ uses an additional parameter list to carry blocked names.

**Definition 6.1** The function $\boldsymbol{\mathcal{D}} : \mathscr{L}(\mathbb{A}) \to \nu\text{-Tree} \to \mathbb{A}\text{-Tree} \to \mathbb{B}$ is defined recursively on the $\nu$-tree.

$$\mathcal{D}\ A\ (a_k n_1 \dots n_k)\ (a_k t_1 \dots t_k) := \bigwedge_{i=1,\dots,k} \mathcal{D}\ (a_k :: A)\ n_i\ t_i$$

$$\mathcal{D}\ A\ (\nu a_k.n)\ t := \begin{cases} \displaystyle\bigvee_{b_k \in\ (\mathcal{C}\ a_k\ A\ n)} \mathcal{D}\ (b_k :: A)\ ((a_k b_k) \cdot n)\ t & \text{, if } a_k \in \mathtt{FN}(n) \\ \mathcal{D}\ A\ n\ t & \text{, if } a_k \notin \mathtt{FN}(n) \end{cases}$$

where $\mathcal{C}$ is the list of possible instantiations for $a_k$:

$$\mathcal{C}\ a_k\ A\ n\ := \mathtt{Name}_k(t) \setminus (A \cup \mathtt{FN}(\nu a_k.n))$$

The first case mirrors the inductive definition of the language and recurses on the pairs of children in a straight forward manner if the names at the top are identical.

In the second case we distinguish further whether the binding is empty. If it is empty, then we do not instantiate it at all and simply ignore it. Otherwise, it is a

proper binding that binds a name somewhere in the subtree. Then we try all possible instantiations to find the right one among them. The right instantiation is a name in $\mathtt{Name}_k(t)$ that satisfies the freshness conditions, since the bound name in the $\nu$-tree has to match the corresponding name in $t$ after instantiating while complying with the freshness conditions. Therefore, if the the correct instantiation exists, then it is in the list $\mathcal{C}$ and it is also unique. If no such instantiation is found, the function returns false.
The following example justifies why the case distinction between empty and proper bindings is necessary. Consider the $\nu$-tree $\nu a_0.b_0$, which has an empty binding. Then $b_0 \in [\![\nu a_0.b_0]\!]$. However $\mathcal{C}\ a_k\ A\ n = []$, meaning the right instantiation for this empty binding is not in $\mathcal{C}$. For proper bindings there is always exactly one right instantiation. But for empty bindings there is no single right instantiation that we could find. Instead, there are infinitely many instantiations which all work out. Any arbitrary fresh name can be used to instantiate an empty binding, as demonstrated in the proof of Theorem 5.8.

## 6.2 Correctness

**Theorem 6.2** $t \in [\![n]\!] \leftrightarrow \mathcal{D}\ A\ n\ t = \mathsf{true}$

**Proof** We prove this claim by induction on $n$ and $t$.

- " $\rightarrow$ ":

$$\begin{aligned} & a_k t_1 \dots t_k \in [\![a_k n_1 \dots n_k]\!]_A \\ \overset{\text{def}}{\Leftrightarrow}\ & t_i \in [\![n_i]\!]_{a_k::A} \\ \overset{\text{IH}}{\Leftrightarrow}\ & \mathcal{D}\ (a_k :: A)\ n_i\ t_i\ = \mathsf{true} \\ \Leftrightarrow\ & \bigwedge_{i=1,\dots,k} \mathcal{D}\ (a_k :: A)\ n_i\ t_i = \mathsf{true} \\ \overset{\text{def}}{\Leftrightarrow}\ & \mathcal{D}\ A\ (a_k n_1 \dots n_k)\ (a_k t_1 \dots t_k) = \mathsf{true} \end{aligned}$$

  This concludes the tree case. In the $\nu$ case we first consider the case that we have a proper binding, that is, $a_k \in \mathtt{FN}(n)$.

$$\begin{aligned} & t \in [\![\nu a_k.n]\!]_A \wedge a_k \in \mathtt{FN}(n) \\ \overset{\text{def}}{\Leftrightarrow}\ & t \in [\![(a_k b_k) \cdot n]\!]_{b_k::A} \wedge b_k \notin A \wedge b_k \notin \mathtt{FN}(\nu a_k.n) \wedge a_k \in \mathtt{FN}(n) \end{aligned}$$

  We now need to show that $b_k$ is in the list of names $\mathcal{C}\ a_k\ A\ n$ which we proposed as the list of possible instantiations.
  We already have $t \in [\![(a_k b_k).n]\!]_{b_k::A}$. Now we use the equivariance of $\mathtt{FN}$ (Lemma 4.9) and the fact that the free names stay fixed (Lemma 5.5), and

we get

$$\begin{aligned} & a_k \in \mathtt{FN}(n) \\ \Leftrightarrow\ & (a_k b_k)(b_k) \in \mathtt{FN}(n) \\ \Leftrightarrow\ & b_k \in \mathtt{FN}((a_k b_k) \cdot n) \\ \Rightarrow\ & b_k \in \mathtt{Name}(t) \end{aligned}$$

Therefore:

$$\begin{aligned} & t \in [\![(a_k b_k).n]\!]_{b_k::A} \wedge b_k \notin A \wedge b_k \notin \mathtt{FN}(\nu a_k.n) \wedge a_k \in \mathtt{FN}(n) \wedge b_k \in \mathtt{Name}(t) \\ \overset{\text{def}}{\Leftrightarrow}\ & t \in [\![(a_k b_k).n]\!]_{b_k::A} \wedge a_k \in \mathtt{FN}(n) \wedge b_k \in \mathcal{C}\ a_k\ A\ n \end{aligned}$$

Having verified that the instantiation $b_k$ is indeed in the list $\mathcal{C}$, we can apply induction to finish the first case.

$$\begin{aligned} \overset{\text{IH}}{\Leftrightarrow}\ & \mathcal{D}\ (b_k :: A)((a_k b_k) \cdot n)\ t = \mathsf{true} \wedge a_k \in \mathtt{FN}(n) \wedge b_k \in \mathcal{C}\ a_k\ A\ n \\ \Leftrightarrow\ & \bigvee_{b_k \in\ (\mathcal{C}\ a_k\ A\ n)} \mathcal{D}\ (b_k :: A)\ ((a_k b_k) \cdot n)\ t \wedge a_k \in \mathtt{FN}(n) \\ \overset{\text{def}}{\Leftrightarrow}\ & \mathcal{D}\ A\ (\nu a_k.n)\ t = \mathsf{true} \end{aligned}$$

Now we consider the case of an empty binding, that is $a_k \notin \mathtt{FN}(n)$. Then we use Theorem 5.8 to finish the proof.

$$\begin{aligned} & t \in [\![\nu a_k.n]\!]_A \wedge a_k \notin \mathtt{FN}(n) \\ \Leftrightarrow\ & t \in [\![n]\!]_A \wedge a_k \notin \mathtt{FN}(n) \\ \overset{\text{IH}}{\Leftrightarrow}\ & \mathcal{D}\ A\ n\ t = \mathsf{true} \wedge a_k \notin \mathtt{FN}(n) \\ \overset{\text{def}}{\Leftrightarrow}\ & \mathcal{D}\ A\ (\nu a_k.n)\ t = \mathsf{true} \end{aligned}$$

- $" \leftarrow "$:

$$\begin{aligned} & \mathcal{D}\ A\ (a_k n_1 \dots n_k)(a_k t_1 \dots t_k) = \mathsf{true} \\ \overset{\text{def}}{\Leftrightarrow}\ & \mathcal{D}\ (a_k :: A)\ n_i\ t_i\ \ = \mathsf{true} \\ \overset{\text{IH}}{\Leftrightarrow}\ & t_i \in [\![n_i]\!]_{a_k::A} \\ \overset{\text{def}}{\Leftrightarrow}\ & (a_k t_1 \dots t_k) \in [\![a_k n_1 \dots n_k]\!]_A \end{aligned}$$

This concludes the tree case. In the $\nu$ case we first consider the case that we have a proper binding, that is, $a_k \in \mathtt{FN}(n)$.

$$\begin{aligned} & \mathcal{D}\ A\ (\nu a_k.n)\ t = \mathsf{true} \wedge a_k \in \mathtt{FN}(n) \\ \overset{\text{def}}{\Leftrightarrow}\ & \bigvee_{b_k \in\ (\mathcal{C}\ a_k\ A\ n)} \mathcal{D}\ (b_k :: A)\ ((a_k b_k) \cdot n)\ t = \mathsf{true} \end{aligned}$$

Since the disjunction $\bigvee$ is true, there exists a name $b_k \in \mathcal{C}\ a_k\ A\ n$, such that $\mathcal{D}\ (b_k :: A)\ ((a_k b_k) \cdot n)\ t = \mathsf{true}$. By the definition of $\mathcal{C}$ the instantiation $b_k$ satisfies the freshness conditions and is therefore a valid instantiation.

$$
\begin{aligned}
&\mathcal{D}\ (b_k :: A)\ ((a_k b_k) \cdot n)\ t = \mathsf{true} \wedge b_k \in \mathcal{C}\ a_k\ A\ n \\
\overset{\text{IH}}{\Leftrightarrow}\ & t \in [\![(a_k b_k) \cdot n]\!]_{b_k :: A} \wedge b_k \in \mathcal{C}\ a_k\ A\ n \\
\overset{\text{def}}{\Leftrightarrow}\ & t \in [\![(a_k b_k) \cdot n]\!]_{b_k :: A} \wedge b_k \in \mathtt{Name}(t) \wedge b_k \notin A \wedge b_k \notin \mathtt{FN}(\nu a_k.n) \\
\overset{\text{def}}{\Leftrightarrow}\ & t \in [\![\nu a_k.n]\!]_A
\end{aligned}
$$

Now we consider the case $a_k \notin \mathtt{FN}(n)$. The binding is empty and we use Theorem 5.8

$$
\begin{aligned}
&\mathcal{D}\ A\ (\nu a_k.n)\ t = \mathsf{true} \wedge a_k \notin \mathtt{FN}(n) \\
\overset{\text{def}}{\Leftrightarrow}\ & \mathcal{D}\ A\ nt = \mathsf{true} \wedge a_k \notin \mathtt{FN}(n) \\
\overset{\text{IH}}{\Leftrightarrow}\ & t \in [\![n]\!]_A \wedge a_k \notin \mathtt{FN}(n) \\
\Leftrightarrow\ & t \in [\![\nu a_k.n]\!]_A
\end{aligned}
$$

□

We admit that this is not the most efficient algorithm since it works in a brute force way. When instantiating $\nu$-bindings, we compute a list of possible names and eliminate those which are blocked by the freshness constraints. The algorithmic complexity of the $\nu$ case therefore depends on the number of different names in $t$ since a recursive call on the instantiated $\nu$-tree is performed for each of them. This is inefficient because for a given tree $t$ there is at most one correct instantiation for the binding since the name in the $\nu$-tree has to match name in $t$ which is fixed.
A smarter algorithm could therefore first analyze the binding some more to find the unique correct instantiation. One approach for this would be to compute the positions in the tree where the bound name appears and then look up the names in the pure tree at the same positions. If the pure tree is in fact in the language of the $\nu$-tree, then the algorithm will find exactly one name which is the correct instantiation for this binding. If more than one name is found, the smarter algorithm would even be able to return false immediately.
We chose the simpler algorithm because it did not require the additional formalization of positions in trees which would also further complicate the proof.

# Chapter 7

# $\nu$-Tree Automata

The language of a $\nu$-tree is a class of pure trees, and we have shown that this class is decidable. We are now interested in classes of $\nu$-trees. For this purpose we define a $\nu$-tree automaton (NTA) that takes $\nu$-trees as input. We define the language of an NTA as the class of all accepted $\nu$-trees and show that the word problem is decidable.

## 7.1 The NTA model

In this section we define an automaton model suitable to decide properties of $\nu$-trees. It is a stream-lined instance of Stirling's dependency tree automata [16].

**Definition 7.1** A ranked **$\nu$-tree automaton (NTA)** $\mathcal{A}$ is a pair $(Q, \Delta)$, where

- $Q$ is a finite type of states
- $\Delta$ is a triple of lists of transitions:
  - $\Delta_1 : \mathscr{L}(Q \times \mathbb{A} \times \mathscr{L}(Q)) \ni q \stackrel{a_k}{\Rightarrow} [q_1, \ldots, q_k]$
  - $\Delta_2 : \mathscr{L}(Q \times Q \times \mathbb{A} \times \mathscr{L}(Q)) \ni (q, q') \stackrel{a_k}{\Rightarrow} [q_1, \ldots, q_k]$
  - $\Delta_3 : \mathscr{L}(Q \times \mathbb{A} \times Q) \ni q \stackrel{a_k}{\Rightarrow} q'$

**Remark** An NTA is not required to be deterministic. The left hand side of a rule is not necessarily unique.

The definition of the NTA benefits greatly from the use of a ranked alphabet because it simplifies the transitions. When reading a name it is determined by its rank how many successor states follow. We globally use a single countably infinite ranked alphabet $\mathbb{A}$, since it is the most natural choice coming from our treatment of $\nu$-trees in the previous sections. This is in contrast to the dependency tree automaton by Stirling in [16] which is defined on a finite ranked alphabet $\Sigma$ consisting of three disjoint sets, $\Sigma_1$ which are the binders, $\Sigma_2$ which are the bound variables, and $\Sigma_3$ which are the remaining names. A further significant difference between the dependency tree automaton and the NTA is that we do not need to restrict our definition to a finite alphabet since every NTA has an internal alphabet

of those names thames that appear in some transition. This internal alphabet is only finite since $\Delta$ is finite.

**Definition 7.2** The list of names in an automaton $\texttt{Name}(\mathcal{A})$ is defined by a projection on the transitions which yields exactly the names that appear in some transition.

$$\begin{aligned}\texttt{Name}(Q, \Delta) &:= \mathsf{map}\ \mathsf{proj}_2\ \Delta_1 \\ &\ \mathbin{+\!\!+}\ \mathsf{map}\ \mathsf{proj}_3\ \Delta_2 \\ &\ \mathbin{+\!\!+}\ \mathsf{map}\ \mathsf{proj}_2\ \Delta_3\end{aligned}$$

The language of an NTA $\mathcal{L}(\mathcal{A}, q, \varphi)$ is a class of $\nu$-trees. It is defined inductively which enables inductive proofs. The inductive characterization is taken from Kirst [11].
The language depends on a state $q \in Q$ and additionally on a partial state assignment $\varphi : \mathbb{A} \to Q_\perp$, where $Q_\perp$ denotes the option type of $Q$. This assignment is used to distinguish between bound and free names by remembering the state in which any name was bound. If $\varphi(a_k) = q$, then the name $a_k$ was bound in the state $q$. Otherwise, if $\varphi(a_k) = \perp$, it is a free name.

**Definition 7.3**

$$\frac{\varphi(a_k) = \perp \quad (q \overset{a_k}{\Rightarrow} [q_1, \dots, q_k]) \in \Delta_1 \quad n_i \in \mathcal{L}(\mathcal{A}, q_i, \varphi)}{a_k n_1 \dots n_k \in \mathcal{L}(\mathcal{A}, q, \varphi)}\ \text{(LF)}$$

$$\frac{\varphi(a_k) = q' \quad ((q, q') \overset{a_k}{\Rightarrow} [q_1, \dots, q_k]) \in \Delta_2 \quad n_i \in \mathcal{L}(\mathcal{A}, q_i, \varphi)}{a_k n_1 \dots n_k \in \mathcal{L}(\mathcal{A}, q, \varphi)}\ \text{(LB)}$$

$$\frac{(q \overset{a_k}{\Rightarrow} q') \in \Delta_3 \quad n \in \mathcal{L}(\mathcal{A}, q', \varphi[a_k := q])}{\nu a_k.n \in \mathcal{L}(\mathcal{A}, q, \varphi)}\ \text{(LN)}$$

**Remark** $\varphi[a_k := q]$ denotes the function that behaves like $\varphi$ but maps $a_k$ to $q$.

The first inductive rule (LF) lets the automaton read free names, and the second (LB) lets it read bound names. The third rule (LN) introduces a $\nu$-binding for a name and updates $\varphi$ accordingly to remember the state in which the name was bound.
Our definition of the language of an NTA differs from the definition of the language of a dependency tree automaton in [16]. A dependency tree automaton is defined with a fixed initial state $q_0$, whereas the language of our NTA model depends on a given state $q$. A further difference is that Stirling defines acceptance using a run on a tree, which is a procedural approach that walks through the tree and assigns labels to each node, whereas we use an inductive definition. In [11]

both language characterizations are shown equivalent by Kirst.
At this point it is also worth mentioning that an NTA is never closed under equivalences of $\nu$-tree languages. As demonstrated in the previous chapter, for any $\nu$-tree $n$ there are infinitely many different $\nu$-trees that have the same $\nu$-tree language. Since any NTA $\mathcal{A}$ has a finite internal alphabet we can use Lemma 3.1 to construct a fresh name $d$ which is fresh relative to the names in the tree $\nu$-tree $\texttt{Name}(n)$ and to the names in the automaton $\texttt{Name}(\mathcal{A})$. When adding a binding to this name $\nu d.n$, this binding is empty. With Theorem 5.8 the language stays the same, that is, $[\![n]\!] \equiv [\![\nu d.n]\!]$, but $\mathcal{A}$ has no applicable transition for this new binding since we chose a name that does not appear in any transition.
Now we study the binding function $\varphi$. It has an infinite domain, $\mathbb{A}$. However, we show that the list $\texttt{Name}(\mathcal{A})$ is the relevant domain for the language $\mathcal{L}$, since those are the names that appear in some transition which is where $\varphi$ has an impact on $\mathcal{L}$.

**Lemma 7.4** $(\forall a \in \texttt{Name}(\mathcal{A}).\ \varphi(a) = \varphi'(a)) \to \mathcal{L}(\mathcal{A}, q, \varphi) = \mathcal{L}(\mathcal{A}, q, \varphi')$.

**Proof** We show $n \in \mathcal{L}(\mathcal{A}, q, \varphi) \Leftrightarrow n \in \mathcal{L}(\mathcal{A}, q, \varphi')$ by induction on $\mathcal{L}$.

- We have $\varphi(a_k) = \bot = \varphi'(a_k)$ and $(q \overset{a_k}{\Rightarrow} [q_1, \dots, q_k]) \in \Delta_1$, such that

$$\begin{aligned} &(a_k n_1 \dots n_k) \in \mathcal{L}(\mathcal{A}, q, \varphi) \\ \Leftrightarrow\ & n_i \in \mathcal{L}(\mathcal{A}, q_i, \varphi) \\ \overset{\text{IH}}{\Leftrightarrow}\ & n_i \in \mathcal{L}(\mathcal{A}, q_i, \varphi') \\ \Leftrightarrow\ & (a_k n_1 \dots n_k) \in \mathcal{L}(\mathcal{A}, q, \varphi') \end{aligned}$$

- We have $\varphi(a_k) = q' = \varphi'(a_k)$ and $((q, q') \overset{a_k}{\Rightarrow} [q_1, \dots, q_k]) \in \Delta_2$, such that

$$\begin{aligned} &(a_k n_1 \dots n_k) \in \mathcal{L}(\mathcal{A}, q, \varphi) \\ \Leftrightarrow\ & n_i \in \mathcal{L}(\mathcal{A}, q_i, \varphi) \\ \overset{\text{IH}}{\Leftrightarrow}\ & n_i \in \mathcal{L}(\mathcal{A}, q_i, \varphi') \\ \Leftrightarrow\ & (a_k n_1 \dots n_k) \in \mathcal{L}(\mathcal{A}, q, \varphi') \end{aligned}$$

- We have $(q \overset{a_k}{\Rightarrow} q') \in \Delta_3$, such that

$$\begin{aligned} &(\nu a_k.n) \in \mathcal{L}(\mathcal{A}, q, \varphi) \\ \Leftrightarrow\ & n \in \mathcal{L}(\mathcal{A}, q', \varphi[a_k := q]) \\ \overset{\text{IH}}{\Leftrightarrow}\ & n \in \mathcal{L}(\mathcal{A}, q', \varphi'[a_k := q]) \\ \Leftrightarrow\ & (\nu a_k.n) \in \mathcal{L}(\mathcal{A}, q, \varphi') \end{aligned}$$

  Note that $\forall\ b \in \texttt{Name}(\mathcal{A}).\ \varphi(b) = \varphi'(b) \to \varphi[a_k := q](b) = \varphi'[a_k := q](b)$. Therefore applying the inductive hypothesis is justified. □

This lemma is significant, because it reduces the infinite alphabet $\mathbb{A}$ to a list of names $\texttt{Name}(\mathcal{A})$ that are relevant for the function $\varphi$. The language $\mathcal{L}(\mathcal{A}, q, \varphi)$ depends on the state mapping $\varphi$ but in the context of the automaton $\mathcal{A}$, $\varphi$ only depends on finitely many names.

## 7.2 Decidability of the Word Problem

We now show that the word problem of $\mathcal{L}(\mathcal{A}, q, \varphi)$ is decidable by defining a decision procedure $\mathcal{M}$. Given an initial state $q$ and an initial partial state assignment $\varphi$, the function $\mathcal{M}$ simply tries all applicable transitions of which there are finitely many. If at least one of the possible transitions is accepting, $\mathcal{M}$ returns true and otherwise false.

**Definition 7.5** The function $\boldsymbol{\mathcal{M}}$ : NTA $\to$ $\nu$-Tree $\to Q \to (\mathbb{A} \to Q_\perp) \to \mathbb{B}$ is defined recursively on the $\nu$-tree.

$$\mathcal{M}\ \mathcal{A}\ \ (a_k n_1 \dots n_k)\ q\ \varphi \quad := \begin{cases} \bigvee\limits_{(q \overset{a_k}{\Rightarrow} [q_1,\dots,q_k]) \in \Delta_1} \left( \bigwedge\limits_{i=1\dots k} (\mathcal{M}\ \mathcal{A}\ n_i\ q_i\ \varphi) \right) & \text{, if } \varphi(a_k) = \perp \\ \bigvee\limits_{((q,q') \overset{a_k}{\Rightarrow} [q_1,\dots,q_k]) \in \Delta_2} \left( \bigwedge\limits_{i=1\dots k} (\mathcal{M}\ \mathcal{A}\ n_i\ q_i\ \varphi) \right) & \text{, if } \varphi(a_k) = q' \end{cases}$$

$$\mathcal{M}\ \mathcal{A}\ \ (\nu a_k.n)\ q\ \varphi \quad := \bigvee_{(q \overset{a_k}{\Rightarrow} q') \in \Delta_3} (\mathcal{M}\ \mathcal{A}\ n\ q'\ \varphi[a_k := q])$$

Note that in any state there are possibly several applicable transitions because of non-determinism. However, there are finitely many because the list of transitions is finite.

**Theorem 7.6** $\forall \mathcal{A}\ n\ q\ \varphi.\ \mathcal{M}\ \mathcal{A}\ n\ q\ \varphi = \mathsf{true} \leftrightarrow n \in \mathcal{L}((A), q, \varphi)$.

**Proof** In both directions we distinguish three inductive cases which all result in a straight forward proof. In the direction $\to$ we get the existence of an accepting transition through $\bigvee$, and in the direction $\leftarrow$ we get it by induction.

- " $\to$ ": We prove this claim by structural induction on the $\nu$-tree $n$.

  In the $(a_k n_1 \dots n_k)$ case we have $\mathcal{M}\ \mathcal{A}\ (a_k n_1 \dots n_k)\ q\ \varphi = \mathsf{true}$. We perform a case analysis of $\varphi(a_k)$.
  If $\varphi(a_k) = \perp$, then there exists a transition $(q \overset{a_k}{\Rightarrow} [q_1, \dots, q_k]) \in \Delta_1$, such

that

$$\begin{aligned} &\mathcal{M}\ \mathcal{A}\ n_i\ q_i\ \varphi = \mathsf{true} \\ \overset{\text{IH}}{\Rightarrow}\ & n_i \in \mathcal{L}((A), q_i, \varphi) \\ \overset{\text{def}}{\Leftrightarrow}\ & (a_k n_1 \dots n_k) \in \mathcal{L}((A), q, \varphi). \end{aligned}$$

If $\varphi(a_k) = q'$, then there exists a transition $((q, q') \overset{a_k}{\Rightarrow} [q_1, \dots, q_k]) \in \Delta_2$, such that

$$\begin{aligned} &\mathcal{M}\ \mathcal{A}\ n_i\ q_i\ \varphi = \mathsf{true} \\ \overset{\text{IH}}{\Rightarrow}\ & n_i \in \mathcal{L}((A), q_i, \varphi) \\ \overset{\text{def}}{\Leftrightarrow}\ & (a_k n_1 \dots n_k) \in \mathcal{L}((A), q, \varphi). \end{aligned}$$

In the $(\nu a_k.n)$ case we have $\mathcal{M}\ \mathcal{A}\ (\nu a_k.n)\ q\ \varphi = \mathsf{true}$. Then there exists a transition $(q \overset{a_k}{\Rightarrow} q') \in \Delta_3$, such that

$$\begin{aligned} &\mathcal{M}\ \mathcal{A}\ n\ q'\ (\varphi[a_k \coloneqq q]) = \mathsf{true} \\ \overset{\text{IH}}{\Rightarrow}\ & n \in \mathcal{L}((A), q', \varphi[a_k \coloneqq q]) \\ \overset{\text{def}}{\Leftrightarrow}\ & (\nu a_k.n) \in \mathcal{L}((A), q, \varphi). \end{aligned}$$

- " $\leftarrow$ ": We prove this claim by induction on $\mathcal{L}(\mathcal{A}, q, \varphi)$, which results in three cases.

  In the first case we have $\varphi(a_k) = \bot$ and $(q \overset{a_k}{\Rightarrow} [q_1, \dots, q_k]) \in \Delta_1$, such that

$$\begin{aligned} & n_i \in \mathcal{L}(\mathcal{A}, q_i, \varphi) \\ \overset{\text{IH}}{\Rightarrow}\ & \mathcal{M}\ \mathcal{A}\ n_i\ q_i\ \varphi = \mathsf{true} \\ \Rightarrow\ & \mathcal{M}\ \mathcal{A}\ (a_k n_1 \dots n_k)\ q\ \varphi = \mathsf{true} \end{aligned}$$

  In the second case we have $\varphi(a_k) = q'$ and $((q, q') \overset{a_k}{\Rightarrow} [q_1, \dots, q_k]) \in \Delta_2$, such that

$$\begin{aligned} & n_i \in \mathcal{L}(\mathcal{A}, q_i, \varphi) \\ \overset{\text{IH}}{\Rightarrow}\ & \mathcal{M}\ \mathcal{A}\ n_i\ q_i\ \varphi = \mathsf{true} \\ \Rightarrow\ & \mathcal{M}\ \mathcal{A}\ (a_k n_1 \dots n_k)\ q\ \varphi = \mathsf{true} \end{aligned}$$

  In the third case we have $(q \overset{a_k}{\Rightarrow} q') \in \Delta_3$, such that

$$\begin{aligned} & n \in \mathcal{L}(\mathcal{A}, q', \varphi[a_k \coloneqq q]) \\ \overset{\text{IH}}{\Rightarrow}\ & \mathcal{M}\ \mathcal{A}\ n\ q'\ (\varphi[a_k \coloneqq q]) = \mathsf{true} \\ \Rightarrow\ & \mathcal{M}\ \mathcal{A}\ (\nu a_k.n)\ q\ \varphi = \mathsf{true} \end{aligned}$$

□

We can therefore decide the word problem for a given NTA $\mathcal{A}$, $\nu$-tree $n$, state $q$, and state assignment $\varphi$.

**Corollary 7.7** It is decidable for an NTA $\mathcal{A}$, a $\nu$-tree $n$, and a function $\varphi$. whether there exists a state $q$, such that $n \in \mathcal{L}(\mathcal{A}, q, \varphi)$.

**Proof** $Q$ is a finite type. There exists a list $l$, that contains all elements, that is, $\forall\, q.\ q \in l$. We define the decision procedure

$$\mathcal{M}'\ \mathcal{A}\ n\ \varphi := \bigvee \mathsf{map}\ (\lambda q.\ \mathcal{M}\ \mathcal{A}\ n\ q\ \varphi)\ l$$

Correctness follows with Theorem 7.6. □

**Corollary 7.8** It is decidable for an NTA $\mathcal{A}$ and a $\nu$-tree $n$ whether there exists a state $q$ and a function $\varphi$, such that $n \in \mathcal{L}(\mathcal{A}, q, \varphi)$.

**Proof** $Q_\perp$ is finite, since $Q$ is. Furthermore we only need to consider names in the list $\texttt{Name}(\mathcal{A})$. Therefore there are finitely many different state assignments from $\texttt{Name}(\mathcal{A})$ to $Q_\perp$, and there exists a list $l$ that contains all those. We define the decision procedure

$$\mathcal{M}''\ \mathcal{A}\ n\ := \bigvee \mathsf{map}\ (\lambda f.\ \mathcal{M}'\ \mathcal{A}\ n\ f)\ l$$

Decidability follows with Corollary 7.7. □

**Remark** Since we work in constructive type theory, we do not get the list $l$ containing all possible state assignments for free. In fact, we cannot construct all functions since there are infinitely many. Instead we use lists of pairs which can be interpreted as functions. Generating all functions then reduces to the combinatorial exercise of generating all possible lists of pairs between two finite domains. A possible way to construct them is explained in Section 8.2.4.

# Chapter 8

# Discussion

## 8.1 Related Work

Languages over infinite alphabets have been studied thoroughly and formal methods based on automata have been applied in automated reasoning systems. When the systems are infinite, reasoning is undecidable in general. Then the goal is to identify decidable special cases. Parsing XML documents is one scenario where the data is the source of the infinity and one of the motivations of studying languages over an infinite alphabet. XML documents are often modeled as labeled trees where the labels are Unicode strings that appear as attribute values, and the set of Unicode strings is obviously infinitely large.

In [4] Figueira describes data trees, where every label has symbol from a finite alphabet and a data element from an infinite alphabet. The design is such that it models the typical structure of an XML document. The automaton model used is a top-down alternating tree register automaton with one register to store and test data, which has also been described in [9].

Variable automata are a different model. They have been studied by Grumberg in [8]. This automaton model operates on pure words from an infinite alphabet, without the additional symbol that was used in the data tree and without registers to remember a specific datum. Instead, a set of bound variables is used that associates each variable with a letter when it has been read.

Finite memory automata have been studied by Kaminski in [10]. This is also an automaton model for word languages over an infinite alphabet. When reading an input symbol, finite memory automata can remember it in a register. For that reason, they are also called register automata. They are constrained to only using those symbols in registers that are read from the input. Because the symbols cannot be modified, finite memory automata cannot perform computations in their registers.

In [1], Bojanczyk uses nominal sets as a framework for his work and defines a different automaton model closer to regular automata where finiteness of the components is relaxed to orbit-finiteness. Alphabets $A$ are orbit finite $G$-sets, a $G$-language over such an alphabet is a set of strings $A^*$. By allowing the components to be orbit-finite, the nominal automaton can use an infinite alphabet and

express properties that are more abstract and independent of concrete names. It is then shown that the class of languages recognized by $G$-automata is equivalent to the class of languages recognized by Kaminski's finite memory automata.
Stirling defines dependency tree automata in [16]. This automaton model resembles our model very closely. We have pointed out differences in our treatment. Apart form the differences mentioned, our NTA model is an instance of Stirling's model.
Languages with $\nu$-bindings have been studied by Gabbay in [7]. In his work he considers the language of $\nu$-words, which are finite strings over an infinite alphabet that contain $\nu$-bindings. He defines a set of axioms for equivalences in a nominal Kleene algebra and show that this axiomatization is sound and complete.
Previously, the concept of $\nu$-bindings has also been used in different context [12, 14] as binders in the $\lambda\nu$-calculus which is an extension to the $\lambda$-calculus. Our $\nu$-trees do not contain $\lambda$-terms and we do not have the computational component.
In his thesis [11] Kirst introduces the definition of $\nu$-trees and their denoted language which we also use in our development. The goal of his thesis was to relate intersection type systems to various sorts of nominal automata, including our NTA model.

## 8.2 Implementation

### 8.2.1 Names and permutations

The alphabet $\mathbb{A}$ is implemented as $(\mathbb{N} \times \mathbb{N})$, meaning names are pairs of natural numbers where the first component is the actual symbol and the second component is the rank. Therefore, $\mathbb{A}$ is countably infinite in every rank by construction.
In the proof of Lemma 3.1 we use an order on the names which is achieved by projecting out the first component and using the order of natural numbers which Coq has already implemented. We order names only by their first component to simplify the construction of a fresh name.
Permutations are bijective functions $\mathbb{A} \to \mathbb{A}$. We implement them equivalently as functions which have an inverse since the inverse is more useful for our purpose. We furthermore add the requirement that the rank is preserved.

### 8.2.2 Well-ranked trees

Trees are implemented in a straightforward inductive way as a name and a list of subtrees.
First note that induction in Coq on this type does not generate a helpful inductive hypothesis. The most reasonable induction on a tree would use the element predicate on lists and generate an inductive hypothesis for all subtrees, that is, all

trees that are elements of the list.
Secondly we only consider well-ranked trees where the rank of the name agrees with the length of the list of subtrees. However, with this type for trees it is easy to construct trees which are not well-ranked.
To address both issues we introduce a well-rankedness predicate `WR` on $\mathbb{A}$-Tree.

$$\frac{\forall\, 1 \leq i \leq k.\ \texttt{WR}\ t_i}{\texttt{WR}\ (a_k t_1 \dots t_k)}$$

We use this predicate in our proofs to ensure well-rankedness. When using induction on it, we also get the expected inductive hypothesis for all subtrees. This predicate is also lifted to $\nu$-Tree to express well-ranked $\nu$-trees.

$$\frac{\forall\, 1 \leq i \leq k.\ \texttt{NWR}\ n_i}{\texttt{NWR}\ (a_k n_1 \dots n_k)} \qquad \frac{\texttt{NWR}\ n}{\texttt{NWR}\ (\nu a_k.n)}$$

### 8.2.3 Similar trees

In Definition 6.1 we have given an algorithm to decide the language for a given tree $t$ and $\nu$-tree n which is recursive in both $t$ and $n$. In the correctness proof of Theorem 6.2 we use induction on $n$ and $t$ at the same time. To make this more convenient in Coq we introduce a similarity predicate which expresses structural similarity between a $\nu$-tree and a pure tree, $\approx\ :\ \nu\text{-Tree} \rightarrow \mathbb{A}\text{-Tree} \rightarrow \texttt{Prop}$. For structural similarity we abstract away from all names in the trees and ignore $\nu$-bindings since they are instantiated.

**Definition 8.1** Structural similarity $\approx$ is defined inductively.

$$\frac{\forall\, 0 < i \leq k.\ n_i \approx t_i}{a_k n_1 \dots n_k \approx b_k t_1 \dots t_k} \qquad \frac{n \approx t}{\nu a_k.n \approx t}$$

We further enhance $\approx$ by considering the following. If a $\nu$-tree $n$ is similar to a pure tree $t$, then $\pi \cdot n$ is also similar to $\pi' \cdot t$ for all permutations $\pi, \pi'$. This is true since permutations do not alter the structure, only names are permuted. This motivates the alternative predicate $\approx_p$.

**Definition 8.2** The stronger similarity condition for $\approx_p$ is obtained by quantifying over all permutations.

$$\frac{\forall\, \pi\ \pi'.\ \forall\, 0 < i \leq k.\ \pi \cdot n_i \approx_p \pi' \cdot t_i}{a_k n_1 \dots n_k \approx_p b_k t_1 \dots t_k} \qquad \frac{\forall\, \pi\ \pi'.\ \pi \cdot n \approx_p \pi' \cdot t}{\nu a_k.n \approx_p t}$$

Using induction on $\approx_p$ results in an even more general inductive hypothesis which is what we use in the proof of Theorem 6.2. Note that the predicates $\approx$ and $\approx_p$

are in fact equivalent since structurally similar trees are already closed under permutations, but the inductive hypothesis by induction on $\approx_p$ is more useful.
The reason why induction on structural similarity is especially useful in the proof of Theorem 6.2 is the following. If $t \in [\![n]\!]$, then this means exactly that $t$ and $n$ are of similar structure and the names match up after instantiating all $\nu$-bindings in $n$. This means that $\approx_p$ contains already half of $[\![-]\!]$. When using induction on $\approx_p$, we already have the structural part and only need to show that the names match up.

### 8.2.4 Exhaustive mappings between two lists

In type theory, the space of functions between two finite domains is not finite without the assumption of functional extensionality, even though there are only finitely many different mappings. However, in the proof of Corollary 7.8 we use that there are finitely many state mappings $\varphi$. To achieve this we use lists of pairs that are interpreted as functions. We generate a list of all possible combinations of lists of pairs and show that every different function can be modeled as a list of pairs. First we show how a list of pairs is interpreted as a function.

$$\begin{aligned} \texttt{to_fun}\ [\,] &\coloneqq \mathsf{const}_y \\ \texttt{to_fun}\ (x,y) :: r &\coloneqq \texttt{to_fun}(r)[x \coloneqq y] \end{aligned}$$

An empty list is interpreted as an arbitrary constant function which maps all inputs to a fixed value, and a pair updates the function.
In the following let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be finite types, We are interested in all possible different mappings $\boldsymbol{X} \to \boldsymbol{Y}$. Since the types are finite there exist lists that contain all elements of that type, which we call $X : \mathscr{L}(\boldsymbol{X})$ and $Y : \mathscr{L}(\boldsymbol{Y})$. To get all different functions we therefore only need to generate a list of all different lists of pairs $l : \mathscr{L}(\boldsymbol{X} \times \boldsymbol{Y})$. This is done using the following functions:

$$\texttt{all_assignments}\ x \coloneqq \mathsf{map}\ (\lambda y.\ (x,y))\ Y$$

The function `all_assignments` $x$ generates all single pairs $(x,y)$ where the first component is fixed. Those are all possible assignments that any function could choose for this $x$.

$$\texttt{all_extensions}\ x\,l \coloneqq \mathsf{map}\ (\lambda p.\ p :: l)\ (\texttt{all_assignments}\ x)$$

The function `all_extensions` $x\ l$ extends a given list of pairs $l$ by all possible assignments of $x$. This results in all possible ways that the list $l$ could be updated.

$$\texttt{step_var}\ x\ L \coloneqq \mathsf{flatten}\ (\mathsf{map}\ (\texttt{all_extensions}\ x)\ L)$$

The function $\texttt{step_var}\ x\ g$ lifts $\texttt{all_extensions}$. Given a name $x$ and a list of list of pairs $L$ it computes all possible extensions for each list of pairs in $L$.

$$\begin{aligned} \texttt{all_mappings'}\ [] &:= [\,[]\,] \\ \texttt{all_mappings'}\ (x :: r) &:= \texttt{step_var}\ x\ (\texttt{all_mappings'}\ r) \\ \texttt{all_mappings} &:= \texttt{all_mappings'}\ X \end{aligned}$$

The function $\texttt{all_mappings}$ folds $\texttt{step_var}$ over $X$.
The following lemma establishes that we do indeed get all combinations.

**Lemma 8.3** $l \in \texttt{all_mappings'}\ A \rightarrow \forall\, x\, y.\ (x, y) :: l \in \texttt{all_mappings'}\ (x :: l)$

**Proof** Since $y \in Y$, we have

$$\begin{aligned} & (x, y) \in \mathsf{map}\ (\lambda y.\ (x, y))\ Y \\ \Leftrightarrow\ & (x, y) \in (\texttt{all_assignments}\ x) \\ \Leftrightarrow\ & (x, y) :: l \in \mathsf{map}\ \ (\lambda p.\ p :: l)(\texttt{all_assignments}\ x) \\ \Leftrightarrow\ & (x, y) :: l \in (\texttt{all_extensions}\ x\ l) \end{aligned}$$

Since $l \in \texttt{all_mappings'}\ A$, we have

$$(\texttt{all_extensions}\ x\ l) \in (\mathsf{map}\ (\texttt{all_extensions}\ x)\ (\texttt{all_mappings'}\ A))$$

Therefore

$$\begin{aligned} & (x, y) :: l \in (\texttt{all_extensions}\ x\ l) \wedge \\ & \qquad (\texttt{all_extensions}\ x\ l) \in (\mathsf{map}\ (\texttt{all_extensions}\ x)\ (\texttt{all_mappings'}\ A)) \\ \Leftrightarrow\ & (x, y) :: l \in \mathsf{flatten}\ (\mathsf{map}\ (\texttt{all_extensions}\ x)\ (\texttt{all_mappings'}\ A)) \\ \Leftrightarrow\ & (x, y) :: l \in \texttt{step_var}\ x\ (\texttt{all_mappings'}\ A) \\ \Leftrightarrow\ & (x, y) :: l \in \texttt{all_mappings'}\ (x :: A) \end{aligned}$$

□

Now we show that for all functions $f$ there is a list of pairs $l$ that behaves exactly like $f$ when interpreted as a function.

**Fact 8.4** $\forall\, f.\ \exists\, l \in \texttt{all_mappings'}\ A \wedge \forall\, x \in A.\ (\texttt{to_fun}\ l)(x) = f(x).$

**Proof** We prove this claim by induction on $A$.

- We have $[] \in \texttt{all_mappings'}\ []$, and $\forall\, x \in [].\ (\texttt{to_fun}\ [])(x) = f(x)$.
- By induction we have

  $$l \in \texttt{all_mappings'}\ A\ \wedge \forall\, x \in A.\ (\texttt{to_fun}\ l)(x) = f(x).$$

  Using Lemma 8.3 we get

  $$(x, f(x)) :: l) \in \texttt{all_mappings'}\ (x :: A),$$

  and clearly

  $$\forall\, x' \in (x :: A).\ (\texttt{to_fun}\ \ (x, f(x)) :: l)(x') = f(x').$$

□

## 8.3 Future work

We have adapted the nominal axioms by Gabbay [7] to equivalences laws for $\nu$-tree languages. Using these equivalences, one goal is to define an algorithm that computes a normal form for $\nu$-trees and is capable of deciding if two $\nu$-trees have the same language.
For the NTA we have shown decidability of the word problem. Emptiness and closure under complement are two language properties which we have not shown for NTA languages.
Furthermore we have shown that the NTA languages are not closed under the equivalence laws for $\nu$-tree languages, that is, for any $n \in \mathcal{L}(\mathcal{A}, q, \varphi)$, there is a $\nu$-tree $n'$ with $[\![n]\!] \equiv [\![n']\!]$ and $n' \notin \mathcal{L}(\mathcal{A}, q, \varphi)$. It would be interesting to study whether we can decide the other direction, that is, if it is decidable for a given $n' \notin \mathcal{L}(\mathcal{A}, q, \varphi)$ whether there exists an equivalent $\nu$-tree n with $[\![n]\!] \equiv [\![n']\!]$ and $\nu$-tree $n' \in \mathcal{L}(\mathcal{A}, q, \varphi)$. Intuitively it would decide the question whether the NTA rejected the $\nu$-tree only because it was constructed in an unexpected way.

## 8.4 Development details

This thesis has been formalized using the Coq proof assistant.

| spec | proof | |
|---|---|---|
| 228 | 248 | Base |
| 103 | 99 | Name permutations |
| 188 | 293 | Lists |
| 38 | 39 | Pure trees |
| 212 | 274 | $\nu$-trees |
| 135 | 284 | Equivalence laws |
| 89 | 221 | Decidability of $t \in [\![n]\!]$ |
| 173 | 173 | NTA |
| 1166 | 1631 | total |

We use parts of the Base library which has been used in the course Introduction to Computational Logic by Prof. Gert Smolka to implement basic list functions and decision procedures.
Our development starts with the implementation of names and permutations. Since a transposition permutes two specific names, we always need two decisions for equality. This results in an exponential growth in case distinctions when transpositions are involved. Because we often deal with more than one transposition in our proofs, we implemented a tactic to solve frequent trivial cases which performs these decisions and makes use of Coq's builtin `congruence` tactic.
We add our own list module which includes basic functionality, such as permu-

tations for lists of names needed for trees and lists of booleans needed for our decision procedures. They are implemented in a generic way and are used a lot over the course of the development.
In the tree modules we use well-rankedness to enable inductive proofs. This often generates proof obligations that certain kinds of subtrees are well-ranked. We implemented a tactic that solves common cases of well-rankedness which was less work than solving each specific case by a lemma.
The equivalence laws for $\nu$-tree languages make heavy use of the inductive definition. Even more induction is used with the inductive similarity predicate which eases the proof process for the decision procedure.
The language of the NTA is also formulated inductively which carries the proofs of most facts.

# Bibliography

[1] Mikołaj Bojańczyk, Bartek Klin, and Slawomir Lasota. Automata theory in nominal sets. *Logical Methods in Computer Science*, 2014.

[2] Ahmed Bouajjani, Peter Habermehl, Yan Jurski, and Mihaela Sighireanu. *Rewriting Systems with Data*. Springer Berlin Heidelberg, 2007.

[3] Ahmed Bouajjani, Peter Habermehl, and Richard Mayr. Automatic verification of resursive procedures with one integer parateter. *Theoretical Computer Science*, 2003. Mathematical Foundations of Computer Science.

[4] Diego Figueira. Forward-xpath and extended register automata on data-trees. In *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings*, 2010.

[5] A. A. Fraenkel. Über die unabhängigkeit des auswahlaxioms und einiger seiner folgerungen. *Journal of Symbolic Logic*, 4, 1939.

[6] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 2002.

[7] Murdoch K. Gabbay and Vincenzo Ciancia. Freshness and name-restriction in sets of traces with names. FOSSACS'11/ETAPS'11, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] Orna Grumberg, Orna Kupferman, and Sarai Sheinvald. *Variable Automata over Infinite Alphabets*. Springer Berlin Heidelberg, 2010.

[9] Marcin Jurdzinski and Ranko Lazic. Alternating automata on data trees and xpath satisfiability. *CoRR*, 2008.

[10] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, November 1994.

[11] Dominik Kirst. Intersection type systems corresponding to nominal automata. Master's thesis, Oxford University, 2016.

[12] Martin Odersky. A functional theory of local names. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1994.

[13] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, 2013.

[14] Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or what's new? In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science*. Springer-Verlag, 1993.

[15] Ian David Bede Stark. *Names and hihger-order functions*. PhD thesis, University of Cambridge, UK, 1994.

[16] Colin Stirling. *Dependency Tree Automata*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.