University of Oxford

Mathematical Institute
Department of Computer Science
Master of Science in Mathematics and Foundations of Computer Science

Intersection Type Systems

Corresponding to Nominal Automata

Candidate Number: 1002670

Trinity Term 2016

Acknowledgments

First of all, I owe my full gratitude to my advisors Steven Ramsay and Luke Ong. It was Steven who initially proposed this topic to me and introduced me to the involved literature. In our weekly meetings I could benefit from his constructive feedback and thoughtful teaching and it is due to him that I learned so much during this project.

Also, I am very grateful for the support I experienced at my wonderful college, Lady Margaret Hall. The academic community (not to mention the beautiful gardens) provided a great intellectual atmosphere which allowed for both concentrated work and inspiring discussions. In particular the college boat club and choir made the past year incredibly exciting.

At the same time my family and friends back in Germany never stopped supporting me and I am thankful for their always warm welcoming when I went home for holidays. I very much appreciate their uncompromising sympathy and patience and I was delighted by the vast number of guests I could show around at this fascinating place.

During this year at Oxford I received financial aid by the German National Academic Foundation (Studienstiftung des deutschen Volkes) and the German Academic Exchange Service (DAAD).

Abstract

In this master's dissertation we examine intersection type systems that correspond to nominal automata. That is, for a given automaton we explain how to define a type system such that its programs are typable (in a certain sense) exactly if they evaluate to an input which is accepted by the automaton. This topic has its origins in software verification where automata are used to analyse properties of program executions. By defining an equivalent type system one obtains an associated type checking algorithm that solves the automaton acceptance problem elegantly and efficiently.

Much of the correspondence is independent from the actual automaton model, hence we factor out a fully generic treatment and evaluate its instantiation with several examples. Those include standard finitary automaton models on words and trees and nominal automaton models on words over atomic names and, lastly, trees with fresh name binding. The latter is what we call ν-trees and as the customised automaton model we introduce ranked ν-tree automata and discuss some of their properties.

The contributions of this work lie firstly in presenting type system-automaton correspondences independent from their original application in software verification. In particular, we provide self-contained introductions to intersection types, nominal sets and (nominal) automata theory. Moreover, by studying new concrete instances we strengthen the connection between the involved topics and encounter some actual synergetic effects.

Contents

Chapter 1
Introduction

In this master's thesis we examine intersection type systems that correspond to nominal automata. We begin by providing the necessary background and by deriving the motivation for our project from a survey of related work.

Types are a widespread concept in mathematical logic and computer science. The essence is to assign certain labels (the types) to the mathematical objects of interest, in order to obtain a structural classification that excludes some paradoxical or at least nonsensical expressions. For instance, for mathematicians it is a natural intuition to distinguish "first-order" objects like natural numbers from "higher-order" objects like functions or relations operating on them. It makes sense to write down an expression like $f(4)$ if f is a function operating on the natural numbers but if g is another such function, writing $f(g)$ is meaningless since the types of f and g mismatch. This is exactly what led to type theories as foundations of mathematics beginning with the work of Russell [Rus08] and contemporarily being discussed in form of the homotopy type theory program [Uni13].

In modern computer science, the main applications of type systems are in typed programming languages, where the attached types rule out some very trivial programming flaws, in computational logic, exploiting a similarity between type derivations and logical deductions [How80] and in cyber security where typable protocols are guaranteed to meet some security policies (cf. [FKS11] for instance). The common pattern is to consider the expressions of some formal language (in our case this will be the so-called **lambda calculus**) and to study a rule-based system that assigns types to some of those terms in a meaningful way.

Intersection types are a slightly less common notion basically allowing terms to be assigned several types at once. This means that such terms can simultaneously act in several roles which ultimately admits more typable expressions. We use this style of systems to overcome some principal limitations of simpler types and to capture the functionality of automata more naturally.

Automata are a basic mathematical model of computation. Normally, they consist of collections of symbols, states and transitions. When started on some input symbols and an initial state, the machine processes the input as matching transitions from state to state apply. The main functionality is to accept or reject the input and hence to define a subset of the input space, the so-called **language**.

Apart from their origin in computability theory, automata have an important application in software verification. In this field it is a common pattern to translate some specification formula for program execution traces into an automaton. Then the accepted language expresses exactly the set of valid executions and a decision algorithm for acceptance is a tester for program executions.

Automata can be classified by the specification of their input, cardinality of components and properties of their transition relation. First, typical input shapes are words or trees over the alphabet of symbols; we will discuss several examples. Secondly, an important specification is whether or not the input alphabet and state space are finite. Treating infinite alphabets is one motivation for nominal automata. In standard automata theory, all incorporated collections are finite to enable decidability of problems such as **acceptance** of the input or **emptiness** of the language. Also, finite automata can be implemented as actual physical machines that process real (user) input. Finally, the transition relation can be either non-deterministic, deterministic or alternating, incorporating both non-determinism and the opposite universality of transitions. We will mostly discuss non-deterministic automata since that is arguably a simpler formulation.

The theory of **nominal sets** is an attempt to elegantly present the use and symmetries of variable names in computer science [Pit13]. The underlying idea actually dates back to the use of permutation models for independence proofs in axiomatic set theory [Fra39]. In modern formulation, a set equipped with a permutation group action is nominal if every element only depends on finitely many members of the permutation carrier. Moreover, if the group action admits only finitely many equivalence classes (the so-called **orbits**) it is called **orbit-finite**. We call all automaton models nominal if they involve nominal sets or at least a notion of atomic names. Nominal automata based on orbit-finite instead of just finite components allow possibly infinite alphabets and can express more abstract properties which can be independent from the concrete symbols.

Now a **type system-automaton correspondence** can be outlined as follows. For a fixed automaton, we define an intersection type system for programs that compute input words/trees. Then the correspondence is established in the way that a program can be assigned a certain type exactly if it terminates and computes an accepted word/tree. Hence typability in the system solves acceptance by the automaton and type inhabitance solves non-emptiness of the language. Moreover, typing programs that eventually normalise to accepted input can be seen as a strong generalisation of the acceptance condition of the automaton. This generalisation can even be extended to programs with gaps referring to assumed subroutines which reveals the compositional structure of acceptance.

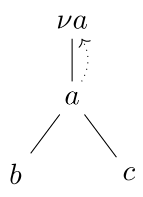
With this project we build on related work in mainly two fields. First, there is the general theory of nominal sets as developed by Pitts and Gabbay [Pit13] which provides the framework for nominal automaton models. In particular, the work of Pitts and Stark on the so-called **ν-calculus** [PS93, Sta98] and the similar **$\lambda\nu$-calculus** by Odersky [Ode94] will play a role. Secondly, the primal use of a type system-automaton correspondence was in the field of **higher-order model checking** [Kob09, KO09, Ram14] which is concerned with verifying properties of higher-order functional programs. Kobayashi was the first to exploit an example correspondence in order to develop simpler decision algorithms which led to the design of more efficient model checkers. Apart from these two fields we also refer to some non-standard automaton models such as the nominal word automata introduced by Bojanczyk [BKL14] and the dependency tree automata defined by Stirling [Sti09]. We will further provide more background references at the beginning of each chapter.

As initiated by Kobayashi, the main motivation for examining said correspondences is the pragmatic search for better algorithms. As we will see in Chapter 3, the acceptance conditions of some automaton models are rather complex and hence algorithms deciding whether or not a word is element of the automaton language can be rather obscure. Then simply translating the problem into the language of a type system allows for applying the arguably simpler type checking algorithms. Moreover, as usual when combining diverse perspectives, there are some mutual benefits. For instance, when put in relation to type systems, the compositional shape of automaton acceptance, that is acceptance of an input structure based on the acceptance of the substructures, is made explicit. This circumstance simplifies the reasoning a lot and is usually rather implicit in common definitions. Finally, there are the already hinted reasons to particularly study nominal automata since they are a well-defined attempt to incorporate infinite alphabets or other non-standard input spaces that for instance provide a notion of name binding (cf. Section 3.5).

With this thesis, we contribute to these research areas in the following ways:

- Our presentation of the type system-automaton correspondence is fully **self-contained**, making the topic accessible to a more general public than the verification community. In particular, we provide introductory accounts of intersection type systems (Section 2.3) and nominal sets (Section 3.3).
- We develop a **schematic correspondence proof** (Section 4.1) that allows for instantiations by very different kinds of concrete automaton models.
- We evaluate the scheme by discussing **correspondence proofs** for finite word automata (Section 4.2), unranked tree automata (Section 4.3), nominal word automata (Section 4.4) and unranked ν-tree automata (Section 4.6).
- The last mentioned **ν-tree automata** are a streamlined instance of Stirling's dependency tree automata. We study the denotation of their accepted languages in the context of nominal sets (Section 3.5) and prove the acceptance and emptiness problems decidable (Section 3.6).

A main challenge during the work on this project was to incorporate the separate fields of nominal sets, automata theory and type systems in one unified treatment. The way how these theories are meshed is best illustrated by an informal early example that foreshadows the type system correspondence for ν-tree automata. To this end we consider the simple ν-tree $\nu a.abc$ where the ν introduces a name a that is used in the body abc with b and c denoting some unbound constants. All these names range over a countably infinite set $\mathbb{A}$. The actual tree structure of $\nu a.abc$ can be depicted as follows:



Here, the dotted arrow denotes the binding of the free constant a to its binder νa. Now by embedding ν-trees into the framework of nominal sets, we can assign a set $[\![n]\!]$ of pure trees over $\mathbb{A}$ to every ν-tree n where the unbound names are fixed and the bound names are instantiated with all unused names from $\mathbb{A}$. In our example case that would be

$$[\![\nu a.abc]\!] = \{\, abc \mid a \in \mathbb{A} \wedge b \neq a \neq c \,\}$$

which is all one-layer binary trees with root different from the leaves b and c. In the usual formalism of nominal sets this collection contains exactly the trees $\pi \cdot abc := \pi(a)\pi(b)\pi(c) = \pi(a)bc$ for all finite permutations π over $\mathbb{A}$ that fix b and c. Next, we will see that it is easy to define a ν-tree automaton $\mathcal{A}$ that accepts exactly the ν-trees that contain a free b. Then in particular $\mathcal{A}$ accepts $\nu a.abc$, say by starting from an initial state q. Anticipating some lambda calculus, we can see that the program $(\lambda x.\nu a.axc)b$ computes our example ν-tree in one reduction step by interpreting $(\lambda x.\nu a.axc)$ as a function with variable x and b acting as input:

$$(\lambda x.\nu a.axc)b \to \nu a.abc$$

Now, once we established a correspondence of $\mathcal{A}$ to a customised type system, this type system assigns type q to all programs that compute an input accepted by $\mathcal{A}$ started in state q. So in particular the system types the example program and the ν-tree itself with q, which is usually denoted by $\vdash (\lambda x.\nu a.axc)b : q$ and $\vdash \nu a.abc : q$. This constitutes a generalisation of acceptance by $\mathcal{A}$ where we include all state-typable programs. In particular, because the system for $\mathcal{A}$ is based on a rich intersection type system, it generalises acceptance even to programs like $K((\lambda x.\nu a.ax)b)\Omega$ where K is a program that simply truncates its second input and Ω denotes the program that always diverges. In a simpler type systems such terms admitting non-terminating executions are not typable.

The remainder of this document is structured as follows. In Chapter 2 we review intersection types for the terms of the lambda calculus. We discuss the most important properties of the system and put it in contrast to the standard system of simple types. Then, in Chapter 3, we present some selected automaton models, starting from very basic classical instances and ending in our contributed ranked ν-tree automata. In parallel we outline the theory of nominal sets providing the necessary language for our purposes. The actual correspondence between intersection type systems and nominal automata is subject of Chapter 4. Here we prove the equivalence of respectively adjusted versions of the pure intersection type system to all the automaton models introduced in the chapter before. We conclude with some summarising remarks and an overview about possible further directions in Chapter 5.

We end this introduction by mentioning some of the notation used in the following text. Many sections will incorporate inductive definitions presented in the form of inference rules such as

$$\frac{H_1 \quad H_2 \quad H_3}{P}$$

which justifies a judgement P whenever H_1 through H_3 are given. In particular, we define grammars by giving a shorthand Backus-Naur Form such as

$$A, B ::= a \mid AB$$

which specifies a set by containing some atom a and for every two members A, B the concatenation AB. Furthermore, to make the statements of propositions as concise as possible, we will leave out universal quantification where possible and reasonable. We use a consistent assignment of variable names to specify the respective domains.

Chapter 2
Intersection Type Systems

In this chapter we develop the basic theory of lambda calculus up to a point that allows for understanding intersection type systems. We begin by introducing the untyped lambda calculus as a minimal Turing complete programming language (cf. [Bar84] for a standard reference). Then we move to the simply typed lambda calculus as introduced by Church [Chu40] and intersection type systems as presented by Hindley [Hin92]. A comprehensive text-book on several styles of typed lambda calculi is [BDS13].

2.1 Untyped Lambda Calculus

A good intuition is to think of the lambda calculus as a very simple programming language. From this perspective, the calculus consists of a set of terms, the programs, and a notion of term reduction, the execution of a program. There are many more subtle applications of lambda calculi in logic and foundations of mathematics but for our purposes this approach is perfectly fine. We first define the set of lambda terms:

Definition 2.1. *The set Λ of* ***lambda terms*** *is generated by the following grammar:*

$$s, t ::= x \mid \lambda x.s \mid st$$

Here, the variables x range over some countably infinite set $\mathsf{Var} = \{x, y, z, \ldots\}$.

Variables in a term can occur in two ways, either bound to a lambda or free. The set of **free variables** in a term s is usually denoted $\mathsf{FV}(s)$ and if $\mathsf{FV}(s) = \emptyset$ we call s **closed**. The three syntactical shapes of terms $s, t \in \Lambda$ can be understood as **variables** x denoting some already defined programs, **functions** $\lambda x.s$ taking an input x and computing the body s and **function applications** st where t is the input argument for the function s. The last case gives rise to an intuition of reduction as, if we apply an actual function $\lambda x.s$ to a term t, we expect to compute s with every occurrence of x bound to t.

We make this formal by first introducing the non-capturing **substitution** function:

Definition 2.2. *We define a function $s[t/x]$ by recursion on $s \in \Lambda$ (where $y \notin \mathsf{FV}(t)$):*

- $x[t/x] := t$
- $y[t/x] := y$
- $(\lambda x.s)[t/x] := \lambda x.s$
- $(\lambda y.s)[t/x] := \lambda y.s[t/x]$
- $(ss')[t/x] := s[t/x]s'[t/x]$

Now we define the actual reduction:

Definition 2.3. *We introduce the β-**reduction relation** $s \to t$ by the following rules:*

$$\frac{}{(\lambda x.s)t \to s[t/x]} \qquad \frac{s \to s'}{\lambda x.s \to \lambda x.s'} \qquad \frac{s \to s'}{st \to s't} \qquad \frac{t \to t'}{st \to st'}$$

*The reflexive-transitive closure of $s \to t$ is usually denoted $s \to^* t$ and the smallest equivalence relation closed under β-reduction is usually called β-**conversion**.*

Note that the first rule captures the actual reduction which replaces the variable name in a function body by the input argument. The other three rules are syntactic closure for allowing reductions at any place in terms. This makes β-reduction in particular non-deterministic, so we can obtain several distinct reduction paths starting from a single term. If there exists a reduction path $s \to^* n$ and n does not admit any further reductions we call n a **normal form** and write $s \Downarrow n$.

A first observation is that we can now distinguish three classes of terms. First, there are terms s that have a normal form n. We call them **weakly normalising** and write $s \in \mathsf{WN}$. Consider for example $Iy := (\lambda x.x)y \Downarrow y$, so $(\lambda x.x)y \in \mathsf{WN}$. Moreover, since the applied reduction is the only one applicable, this term comes with the even stronger property that all possible reduction paths end in a normal form. We call such terms **strongly normalising** and denote the corresponding set with SN. Finally, there exist terms that do not admit a normal form at all. For instance note that $\Omega := (\lambda x.xx)(\lambda x.xx)$ only reduces to itself in a single step and hence **diverges** on every reduction path. Of course it is $\mathsf{SN} \subseteq \mathsf{WN}$ and an example of a weakly but not strongly normalising term is $KI\Omega$ where $K := \lambda xy.x$. From the programming perspective the three kinds are simply programs that terminate if the operations are executed in a clever order, terminate always or diverge always. We will see that one motivation for intersection types is to characterise the sets WN and SN.

A more involved observation is that β-reduction is **Church-Rosser** [CR36]:

Theorem 2.1. *If $s \to^* t$ and $s \to^* t'$ then there is u with $t \to^* u$ and $t' \to^* u$.*

Sketch. This is a standard result that can be found in the cited literature. We give a brief outline of the proof to provide the main idea. For the slightly different **parallel reduction** $s \to_{||} t$ we can show the so-called **diamond property**, that is, there is u with $t \to_{||} u$ and $t' \to_{||} u$ if $s \to_{||} t$ and $s \to_{||} t'$ for $t \neq t'$. Then parallel reduction is established to be Church-Rosser by induction on the length of the reductions. Finally, we conclude the claim for β-reduction by observing that both reduction notions have the same reflexive-transitive closure.

This means that, although β-reduction is not really deterministic we can still make no irreversible mistakes during searching for normal forms. In particular, a direct consequence is that β-reduction admits **unique normal forms**:

Corollary 2.2. *If $s \Downarrow n$ and $s \Downarrow n'$ then $n = n'$.*

Proof. We have $s \to^* n$ and $s \to^* n'$ so by the Church-Rosser property there is a unifier u with $n \to^* u$ and $n' \to^* u$. Since n and n' are normal and so do not admit any further reduction we have $n = u = n'$. □

So considered as a programming language, the untyped lambda calculus comes with a very well-behaved program execution. Moreover, it provides the full computational power as any other Turing complete language. We will not study this in detail given that it is only conceptually interesting for our purposes. The main idea can be outlined as follows (cf. [Bar84]). First, one encodes numerals that capture the behaviour of the natural numbers as actual lambda terms. Then one encodes the booleans and pairs and hence obtains expressiveness of simple conditional calculations. In a last step, one encodes the concept of recursion by the means of so-called fixpoint combinators that compute fixpoints of arbitrary terms. Then one can prove a theorem along the lines of:

Theorem 2.3. *A numerical function $f : \mathbb{N}^n \to \mathbb{N}$ is definable as a lambda term exactly if it is general recursive (which we do not actually define here).*

Sketch. There is no further interest for us to study general recursive functions in detail. The only important message is that they are structured similarly as lambda terms in the sense that they consist of a certain set of base functions and provide a recursion operator. The base functions can directly be implemented using the encoded numbers, booleans and pairs, the recursion operator is modelled by fixpoint combinators.

Historically, this result is at least due to Church, Kleene and Rosser. After the same equivalence for Turing computable functions had been established by Turing, the computability community grew convinced that this captured the intuitive notion of "effectively" computable functions by pen-and-paper algorithms. Nowadays it is common to classify programming languages as Turing complete and in particular the modern functional languages are direct implementations of the idea of lambda calculus. These often provide a notion of typing that we will introduce in the next section. An important consequence of being Turing complete is that the calculus comes with the same limitations. For instance, there exist no computable functions that decide the sets WN or SN (cf. the **halting problem**).

2.2 Simply Typed Lambda Calculus

In Chapter 1 we gave some intuition for type systems as ruling out mathematically meaningless or contradictory statements. Specifically in computer science we can find the same concept in typed programming languages. Those come with a notion of data types (such as integers or strings) that are assigned to the operating variables. Then functions have an exact functional type from an input type to an output type. The advantage is that even before execution some very basic programming flaws like applying a function to an argument of the wrong type can be ruled out. Since we understand lambda calculus as a simple programming language, we can study a typing system implementing this idea. To start, we introduce the set of types:

Definition 2.4. *The set* SType *of simple types is generated by the following grammar:*

$$\sigma, \tau ::= a \mid \sigma \to \tau$$

Here the type variables a *range over some set* $\mathsf{Type} = \{a, b, c, \ldots\}$ *of base types.*

In an actual implementation the base types become the normal atomic types such as integers and strings. However, we treat the set abstractly to in particular allow for customised instantiations in Chapter 4. The arrow types $\sigma \to \tau$ are intended for functions taking arguments of type σ and producing results of type τ. Note that it is common to have the arrow associate to the right.

We can now examine how to assign these types to matching lambda terms by a rule-based syntactical definition. The system we introduce is known as **simply typed lambda calculus (STLC)**:

Definition 2.5. *Let* $\Gamma : \mathsf{Var} \rightharpoonup \mathsf{SType}$ *be a finite partial function, a so-called* ***context****. We define a typing relation of judgements* $\Gamma \vdash s : \tau$ *by the following rules:*

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ (Var) \qquad \frac{(\Gamma, x : \sigma) \vdash s : \tau}{\Gamma \vdash \lambda x.s : \sigma \to \tau}\ (Abs) \qquad \frac{\Gamma \vdash s : \sigma \to \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash st : \tau}\ (App)$$

Here $(\Gamma, x : \sigma)$ *denotes the function that behaves like* Γ *but also maps* x *to* σ*. As a shorthand, if we have* $\emptyset \vdash s : \tau$ *we simply write* $\vdash s : \tau$*.*

In words, variables are typed exactly following the assignments in the context. Abstractions are assigned a function type $\sigma \to \tau$ if the body can be assigned type τ whenever the input variable is assumed to be of type σ. Finally, applications are typed if the first operand has a function type and the second operand has matching input type.

We will now study some properties of this system. A first observation is that all judgements remain derivable in expanded contexts. We say a context Γ' expands Γ, denoted $\Gamma \subseteq \Gamma'$ if $\mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\Gamma')$ and we have $\Gamma(x) = \Gamma'(x)$ for all $x \in \mathsf{dom}(\Gamma)$. The mentioned property is usually called **weakening**:

Fact 2.4. *If* $\Gamma \vdash s : \tau$ *and* $\Gamma \subseteq \Gamma'$ *then* $\Gamma' \vdash s : \tau$.

Proof. This is by a straight-forward structural induction on $s \in \Lambda$. $\square$

The converse **strengthening** is true only if we erase typings of non-free variables:

Fact 2.5. *If* $(\Gamma, y : \tau') \vdash s : \tau$ *and* $y \notin \mathsf{FV}(s)$ *then* $\Gamma \vdash s : \tau$.

Proof. This is again by structural induction on $s \in \Lambda$. $\square$

In particular, this means that if s is closed and $\Gamma \vdash s : \tau$ we already know $\vdash s : \tau$. The next property is again best accessible from the programming perspective. If we have a program of a certain type we want to derive the same type after executing a computation step. This is normally named **subject reduction**:

Lemma 2.6. *If* $\Gamma \vdash s : \tau$ *and* $s \to t$ *then* $\Gamma \vdash t : \tau$.

Proof. We begin by an induction on the derivation of $s \to t$. In the three cases for syntactic closure the claim follows directly by induction. So we assume we are in the case $(\lambda x.s)t \to s[t/x]$ and have a judgement $\Gamma \vdash (\lambda x.s)t : \tau$. This can be inverted to $(\Gamma, x : \sigma) \vdash s : \tau$ and $\Gamma \vdash t : \sigma$ for some σ. Then a structural induction on $s \in \Lambda$ together with strengthening and weakening establishes $\Gamma \vdash s[t/x] : \tau$. $\square$

Now we could ask whether the converse **subject expansion** is true as well, that is, whether from $s \to t$ and $\Gamma \vdash t : \tau$ we can conclude $\Gamma \vdash s : \tau$. In fact, it is easy to show that this is not the case. Just consider the term $K := \lambda xy.x$. With the terms I and Ω from above we observe $KI\Omega \to I$. Now obviously we can type $\vdash I : \tau \to \tau$ for every $\tau \in \mathsf{SType}$. However, we cannot derive any typing for $KI\Omega$ since this would include a typing of the self-application xx appearing as a subterm in Ω. So in particular we cannot expect $\vdash KI\Omega : \tau \to \tau$.

The fact that terms containing self-applications are not typable in STLC has further consequences. Since, intuitively speaking, self-application is essential for every term yielding a reduction loop, we can prove the following strong normalisation theorem:

Theorem 2.7. *If* $\Gamma \vdash s : \tau$ *then* $s \in \mathsf{SN}$.

Sketch. This is an instance of a proof with "logical relations". In a first step, a class of reducible terms (so a unary logical relation) is defined by recursion on types. Then strong normalisation is established for every reducible term by induction on types and finally by induction on the syntax of terms every typable term is shown reducible.

In fact, this property marks one of the main differences between simple types and intersection types, as we will see in the next section. Since the typable terms turn out to be a very restricted class, many problems of STLC are algorithmically decidable. The three most common problems to consider are:

- **Type checking**: deciding whether a typing $\Gamma \vdash s : \tau$ is derivable
- **Typability**: deciding whether for a term s there is a typing $\Gamma \vdash s : \tau$
- **Inhabitance**: deciding whether for a type τ there is a closed s with $\vdash s : \tau$

All of the above can be proven decidable, we simply refer the reader to presentations as given by Hindley [Hin97]. These results make STLC a type system which is well-suited for algorithmic implementation. However, the cost is a very restricted typing system that lacks a lot of expressiveness. First, given strong normalisation, all functions definable in STLC are total which of course leaves the system far from Turing complete. Secondly, as we have discussed above, terms containing self-applications are generally not typable which does not only rule out the diverging or weakly normalising terms but also some strongly normalising terms such as $\omega := \lambda x.xx$. The latter was a main motivation for studying more generous systems such as the intersection type system we introduce in the next section.

2.3 Intersection Types

So how to assign a meaningful type to ω? Taking a closer look, the problem with the self-application xx is that the variable x appears in two roles, once as a function and once as an argument. So the first x must have a type $\sigma \rightarrow \tau$ which determines the second x to have type σ. Now we cannot derive a type accommodating both since there is no $\sigma, \tau \in \mathsf{SType}$ with $\sigma \rightarrow \tau = \sigma$. The solution is to simply assign both types to x by introducing a **type intersection** $(\sigma \rightarrow \tau) \wedge \sigma$. Hence, if we have $x : (\sigma \rightarrow \tau) \wedge \sigma$ we can derive $xx : \tau$. This idea is due to Coppo and Dezani [CDC80] and we refer to the presentation given by Hindley [Hin92]. We begin by first defining a richer type language incorporating the intersection operator $\wedge$:

Definition 2.6. *We define a set* IType *of intersection types* τ *by the following grammar:*

$$\tau ::= a \mid \sigma \rightarrow \tau$$
$$\sigma ::= \bigwedge_{i=1}^{k} \tau_i$$

Again, we have the type variables a *range over the set* $\mathsf{Type} = \{a, b, c, \ldots\}$ *of base types.*

The two-sorted grammar mainly has the purpose to only allow base types at the right-most position in a composite type. If $k > 1$ we will often just write $\bigwedge \tau_i$ to improve readability. For $k = 1$ we just write τ and if $k = 0$ we write $\bigwedge \emptyset$. Moreover, if k is small we write the intersection infix, as in the above $(\sigma \rightarrow \tau) \wedge \sigma$, where we have the intersection bind tighter than the arrow. The type system we are about to define is set up in a way that we do not have to care about associativity and commutativity of $\wedge$. Incorporating all this, examples for correct intersection types are $\bigwedge \emptyset \to a$ and $a \wedge b \to c$. Ill-formed are types like $a \to a \wedge b$ or $(a \to \bigwedge \tau_i) \to a$. This is no actual lack of expressibility since the idea of $a \to a \wedge b$ can be captured by the well-formed type $(a \to a) \wedge (a \to b)$. With this in mind, we can now introduce the **intersection typed lambda calculus (ITLC)**:

Definition 2.7. *Let Γ be a context mapping variables to type intersections of the syntactical sort σ. We introduce typing judgements $\Gamma \vdash s : \tau$ by the following rules:*

$$\frac{\Gamma(x) = \bigwedge \tau_i}{\Gamma \vdash x : \tau_i}\,(Var) \qquad \frac{(\Gamma, x : \sigma) \vdash s : \tau}{\Gamma \vdash \lambda x.s : \sigma \to \tau}\,(Abs) \qquad \frac{\Gamma \vdash s : \bigwedge \tau_i \to \tau \quad \Gamma \vdash t : \tau_i}{\Gamma \vdash st : \tau}\,(App)$$

The requirement $\Gamma \vdash t : \tau_i$ in (App) abbreviates a necessary typing for all $1 \leq i \leq k$. In particular, if $k = 0$ there is no typing condition on the argument t at all.

Note that this system always provides types of the sort τ which are not pure intersections. In words, we can type a variable with every type the context provides. Abstractions are typed as usual and applications are typed with τ if the argument can be typed with every type expected by the function. To first see an example, we give a full derivation of a typing for ω:

$$\cfrac{\cfrac{\cfrac{}{[x : \bigwedge \emptyset \to \tau] \vdash x : \bigwedge \emptyset \to \tau}\,(Var)}{[x : \bigwedge \emptyset \to \tau] \vdash xx : \tau}\,(App)}{\vdash \lambda x.xx : (\bigwedge \emptyset \to \tau) \to \tau}\,(Abs)$$

Note the trick how we incorporate $\bigwedge \emptyset$ to have no requirement for the argument in the (App) rule. This will become essential when we prove subject expansion. First note that the intersection type system satisfies weakening, strengthening and subject reduction in the same form as STLC did. The proofs are very similar so we omit them here and instead study subject expansion in full detail. This property is crucial for the correspondences in Chapter 4 and implies the completeness direction of Theorem 2.9.

Lemma 2.8. *If $\Gamma \vdash t : \tau$ and $s \to t$ then $\Gamma \vdash s : \tau$.*

Proof. This proof has the same structure as the above one for subject reduction. A first induction on $s \to t$ leaves as only interesting case $(\lambda x.s)t \to s[t/x]$. Then we assume $\Gamma \vdash s[t/x] : \tau$ and want to prove $\Gamma \vdash (\lambda x.s)t : \tau$. This is again by a second induction on $s \in \Lambda$ for arbitrary Γ and τ. Since the resulting cases are slightly more complex and use the properties of intersection types, we go through them explicitly:

- If $s = x$ the assumption is $\Gamma \vdash t : \tau$ and we derive the claim with:

$$\cfrac{\cfrac{(\Gamma, x : \tau) \vdash x : \tau}{\Gamma \vdash \lambda x.x : \tau \to \tau} \quad \Gamma \vdash t : \tau}{\Gamma \vdash (\lambda x.x)t : \tau}$$

- If $s = y \neq x$ the assumption is $\Gamma \vdash y : \tau$ and we establish the claim by:

$$\dfrac{\dfrac{(\Gamma, x : \bigwedge \emptyset) \vdash y : \tau}{\Gamma \vdash \lambda x.y : \bigwedge \emptyset \to \tau}}{\Gamma \vdash (\lambda x.y)t : \tau}$$

- If $s = \lambda x.s'$ the assumption is $\Gamma \vdash \lambda x.s' : \tau$. The claimed typing follows by:

$$\dfrac{\dfrac{(\Gamma, x : \bigwedge \emptyset) \vdash \lambda x.s' : \tau}{\Gamma \vdash \lambda xx.s' : \bigwedge \emptyset \to \tau}}{(\lambda xx.s')t : \tau}$$

- If $s = \lambda y.s'$ for $y \neq x$ the assumed judgement is $\Gamma \vdash \lambda y.s'[t/x] : \sigma \to \tau$. This inverts to $(\Gamma, y : \sigma) \vdash s'[t/x] : \tau$. By IH $(\Gamma, y : \sigma) \vdash (\lambda x.s')t : \tau$ which inverts to $(\Gamma, y : \sigma, x : \bigwedge \tau_i) \vdash s' : \tau$ and $(\Gamma, y : \sigma) \vdash t : \tau_i$ for each i. Then:

$$\dfrac{\dfrac{(\Gamma, x : \bigwedge \tau_i, y : \sigma) \vdash s' : \tau}{\Gamma \vdash \lambda xy.s' : \bigwedge \tau_i \to \sigma \to \tau} \quad \dfrac{y \notin t}{\Gamma \vdash t : \tau_i}}{\Gamma \vdash (\lambda xy.s')t : \sigma \to \tau}$$

- If $s = s_1 s_2$ the assumption is $\Gamma \vdash s_1[t/x]s_2[t/x]$. Inverting this yields $\Gamma \vdash s_1[t/x] : \bigwedge \tau_i \to \tau$ and $\Gamma \vdash s_2[t/x] : \tau_i$. Then the IH applies yielding $\Gamma \vdash (\lambda x.s_1)t : \bigwedge \tau_i \to \tau$ and $\Gamma \vdash (\lambda x.s_2)t : \tau_i$ which respectively invert to $(\Gamma, x : \bigwedge \tau'_k) \vdash s_1 : \bigwedge \tau_i \to \tau$ with $\Gamma \vdash t : \tau'_k$ and $(\Gamma, x : \bigwedge \tau''_l) \vdash s_2 : \tau_i$ with $\Gamma \vdash t : \tau''_l$. This altogether allows for the following derivation where τ_j denotes a combined indexing of both the τ'_k and τ''_l:

$$\dfrac{\dfrac{(\Gamma, x : \bigwedge \tau_j) \vdash s_1 : \bigwedge \tau_i \to \tau \quad (\Gamma, x : \bigwedge \tau_j) \vdash s_2 : \tau_i}{\Gamma \vdash \lambda x.s_1 s_2 : \bigwedge \tau_j \to \tau} \qquad \Gamma \vdash t : \tau_j}{\Gamma \vdash (\lambda x.s_1 s_2)t : \tau}$$

This finishes the proof as we have shown $\Gamma \vdash (\lambda x.s)t : \tau$ for all cases of $s \in \Lambda$. $\square$

In contrast to STLC, an intersection type system such as the one discussed here also provides types for only weakly normalising terms. For instance, consider the following derivation for the term $KI\Omega$ for some arbitrary $\tau \in \mathsf{IType}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[x : \tau \to \tau, y : \bigwedge \emptyset] \vdash x : \tau \to \tau}}{\vdash K : (\tau \to \tau) \to \bigwedge \emptyset \to (\tau \to \tau)} \quad \dfrac{\dfrac{}{[x : \tau] \vdash x : \tau}}{\vdash I : \tau \to \tau}}{\vdash KI : \bigwedge \emptyset \to (\tau \to \tau)}}{\vdash KI\Omega : \tau \to \tau}$$

As a consequence, we cannot expect the system to normalise strongly. However, it is still weakly normalising and in fact it types exactly the terms in WN:

Theorem 2.9. *Every $s \in \Lambda$ is ITLC-typable if and only if s is weakly normalising.*

Proof. The proof that ITLC-typable terms are weakly normalising is in [BCDC83] (Theorem 4.13). The converse direction is rather simple. First, we can establish by structural induction that all β-normal forms are typable. This is trivial for variables and directly by induction for lambdas. All β-normal applications have the form $xn_1 \dots n_k$ and from the inductive typings of the n_i we can derive a typing for the whole of $xn_1 \dots n_k$ where we exploit the properties of intersection types. Then all $s \in \mathsf{WN}$ are typable by subject expansion. □

Recall that the set WN is not decidable as it expresses a limitation of Turing complete languages. A way to justify this is via the Scott-Curry theorem which implies undecidability of every non-trivial subset of Λ that is closed under β-conversion. Thus and as a consequence of Theorem 2.9, typability is not decidable in ITLC. Moreover, the Scott-Curry theorem applies to the set $S := \{\, s \in \Lambda \mid \Gamma \vdash s : \tau \,\}$ for fixed Γ, τ given subject expansion and subject reduction. Hence also type checking is undecidable. Finally, Urzyzcyn has proven the inhabitance problem undecidable [Urz99], which altogether marks ITLC very different from STLC.

Note that there are also variants of intersection type systems that provide types exactly for the smaller set of strongly normalising terms, leading to slightly different settings. We use the generous system ITLC as a base for the systems capturing automata in order to achieve a correspondence for as many terms as possible.

Chapter 3
Nominal Automata

We begin this chapter by briefly discussing two standard types of automata, namely finite word and tree automata. In particular the former are common knowledge in computer science and we just provide the basic definitions to allow for a very simple example type system-automaton correspondence. The latter yield a suitable introduction into the treatment of trees as they arise from reduction of lambda-terms which will become important in Chapter 4. Much more background about tree automata is discussed in [CDG$^+$07]. Then, in Section 3.3, we give a brief introduction into the theory of nominal sets as developed by Pitts and Gabbay [Pit13]. A first instance of a nominal automaton will be examined in Section 3.4 which presents the work of Bojanczyk, Klin and Lasota [BKL14]. The remainder is our original work on automata on so-called ν-trees which are related to the ν-calculus of Pitts and Stark [PS93, Sta98]. The actual relationship will be discussed in more detail in Chapter 4. All automaton models will be defined in a non-deterministic way to keep some closure constructions as simple as possible. If not stated otherwise the definitions could equivalently be formulated deterministically.

3.1 Finite Word Automata

We begin by introducing finite word automata, probably the most simple instance of a finite-state machine. They belong to the basic education of every student in computer science and are normally studied as the most primitive class of computing machines. A standard text-book containing a profound presentation of finite automata is [Koz97]. In this text-book, one of the examined standard results is that finite automata express exactly the **regular languages**. Our discussion will be based on the following definition:

Definition 3.1. *A **non-deterministic finite automaton (NFA)** $\mathcal{A}$ consists of:*

- *Σ: a finite alphabet of symbols*
- *Q: a finite set of states*
- *$F \subseteq Q$: a finite subset of final states*
- *$\delta \subseteq Q \times \Sigma \times Q$: a finite transition relation*

We write $q \overset{a}{\to} q'$ if $(q, a, q') \in \delta$ and for $w = a_1 \dots a_n \in \Sigma^*$ we write $q \overset{w}{\to} q'$ if $q \overset{a_1}{\to} q_1 \overset{a_2}{\to} \dots \overset{a_n}{\to} q'$ for some $q_1, \dots, q_{n-1} \in Q$.

Note that in contrast to the standard presentation we do not fix a subset of initial states. This has the purpose that we can give a compositional definition of acceptance relative to a starting state, which is well-suited for reinterpretation in a type system. To illustrate this, we first give a traditional definition:

Definition 3.2. *Let $\mathcal{A}$ be an NFA. We say that $\mathcal{A}$ **accepts** a word $w \in \Sigma^*$ from state q if there is $q' \in F$ with $q \overset{w}{\to} q'$. All accepted w are collected in the set $\mathcal{L}(\mathcal{A}, q)$, the **$q$-language of** $\mathcal{A}$.*

Typical examples for NFA-definable languages are words that begin with a, contain at least two b and so on. The standard theory provides results for closure of NFA-languages under boolean operations, decidability of acceptance and emptiness and the equality of regular and NFA-languages. We will not discuss these properties here since we will see a more complex instance in Section 3.6. Instead, we study the mentioned compositional definition and prove both equivalent:

Definition 3.3. *For an NFA $\mathcal{A}$ we define the q-language $\mathcal{L}'(\mathcal{A}, q)$ by:*

$$\frac{q \in F}{\epsilon \in \mathcal{L}'(\mathcal{A}, q)}\ (LF) \qquad\qquad \frac{q \overset{a}{\to} q' \quad w \in \mathcal{L}'(\mathcal{A}, q')}{aw \in \mathcal{L}'(\mathcal{A}, q)}\ (LCst)$$

Here, $\epsilon \in \Sigma^*$ denotes the empty word of length 0 and aw is the concatenation of a and w.

The following lemma establishes the equality of both definitions:

Lemma 3.1. *For every NFA $\mathcal{A}$ and $q \in Q$ we have $\mathcal{L}(\mathcal{A}, q) = \mathcal{L}'(\mathcal{A}, q)$.*

Proof. We justify both inclusions by inductive reasoning:

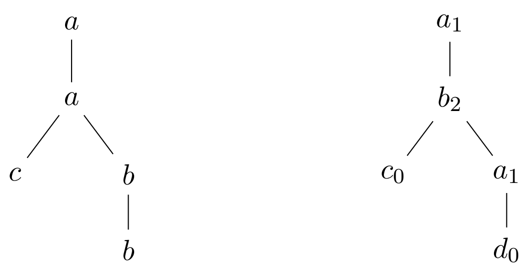
- Suppose $a_1 \dots a_n \in \mathcal{L}(\mathcal{A}, q)$, so there are $q_1, \dots, q_n \in Q$ with $q_n \in F$ and $q \overset{a_1}{\to} q_1 \overset{a_2}{\to} \dots \overset{a_n}{\to} q_n$. Then we establish $a_1 \dots a_n \in \mathcal{L}'(\mathcal{A}, q)$ by natural induction on $n \in \mathbb{N}$. If $n = 0$ we have $a_1 \dots a_n = \epsilon$ and $q \in F$. Hence $a_1 \dots a_n \in \mathcal{L}'(\mathcal{A}, q)$ by the first definitional rule. In the inductive case for $n > 0$ the IH yields $a_2 \dots a_n \in \mathcal{L}'(\mathcal{A}, q_1)$. Then $a_1 \dots a_n \in \mathcal{L}'(\mathcal{A}, q)$ follows from $q \overset{a_1}{\to} q_1$ and the second definitional rule.
- Conversely, if we assume $a_1 \dots a_n \in \mathcal{L}'(\mathcal{A}, q)$ we show $a_1 \dots a_n \in \mathcal{L}(\mathcal{A}, q)$ by induction on the derivation of the assumption. In the case of the first

rule we have $a_1 \dots a_n = \epsilon$ and $q \in F$. Then $a_1 \dots a_n \in \mathcal{L}(\mathcal{A}, q)$ since $q \xrightarrow{\epsilon} q$. In the case of the second rule we have $q \xrightarrow{a_1} q_1$ and $a_2 \dots a_n \in \mathcal{L}'(\mathcal{A}, q_1)$ for some $q_1 \in Q$. Then by IH we have a run $q_1 \xrightarrow{a_2 \dots a_n} q_n$ and by attaching $q \xrightarrow{a_1} q_1$ we obtain $a_1 \dots a_n \in \mathcal{L}(\mathcal{A}, q)$. $\square$

Henceforth and in all further instances, we will use the inductive reformulation as the working definition. The advantage is that the correspondence to type systems will be much easier to prove if we already start with a rule-based definition of acceptance. In particular for the following more complex automaton models reformulating the language definition is a major step in the correspondence proof.

3.2 Finite Tree Automata

In this section we discuss finite automata that operate on trees rather than pure words. Tree alphabets can come in two forms, namely **ranked** or **unranked**. In the case of a ranked alphabet Σ there is a function $\mathsf{rk} : \Sigma \to \mathbb{N}$ such that every node labelled with a symbol a has exactly $\mathsf{rk}(a)$ children. For ranked alphabets we simply write a_k to denote that $\mathsf{rk}(a) = k$. In the following example, the left tree is unranked given that the labels a and b appear with varying arity whereas the right tree is ranked with the arities attached as subscripts:



Since we will present ν-trees in a ranked form in Section 3.5 we will just introduce unranked tree automata here. For ranked tree automata we refer the reader to [CDG⁺07] and note that, since they can be seen as special unranked tree automata, the correspondence result shown in Section 4.3 applies to them as well. Therefore, consider the following formal definition of the set of finite unranked trees over a given alphabet:

Definition 3.4. *Let Σ denote an alphabet. Then the set Σ-*Tree *is generated by:*

$$n ::= a n_1 \dots n_k$$

Here $a \in \Sigma$ and $k = 0$ is included to allow for obtaining leaves but there is no empty tree.

Note that we exclude the empty tree in order to embed Σ-Tree into the non-empty terms of a lambda calculus with constant symbols in Chapter 4. Further, note that this definition is purely syntactic but still allows for a more common representation of trees as functions from an address space to the alphabet. Addresses w are simply words over $\mathbb{N}$ and we set $an_1 \dots n_k(\epsilon) := a$ and $an_1 \dots n_k(iw) := n_i(w)$ for $1 \leq i \leq k$. Then $\mathsf{dom}(n) \subseteq \mathbb{N}^*$ is the set of all valid addresses of the tree n which has the usual properties of being prefix closed and satisfying that, if $wi \in \mathsf{dom}(n)$ then $wj \in \mathsf{dom}(n)$ for all $1 \leq j \leq i$.

To understand these different representations, consider the tree on the left from the previous example. Syntactically, this is represented by $a(ac(bb)) \in \Sigma\text{-}\mathsf{Tree}$ for any $\Sigma \supseteq \{a, b, c\}$. Then we obtain the second representation as a function $a(ac(bb)) : \{\epsilon, 1, 11, 12, 121\} \to \Sigma$ by $a(ac(bb))(\epsilon) := a$ and $a(ac(bb))(12) := b$ etc. We now define automata on unranked trees following Cristau et al. [CLT05]:

Definition 3.5. *A **finite unranked tree automaton (UTA)** $\mathcal{A}$ consists of:*

- *Σ: a finite alphabet of symbols*
- *Q: a finite set of states*
- *$\delta \subseteq \mathsf{Reg}(Q) \times \Sigma \times Q$: a finite transition relation*

Here, $\mathsf{Reg}(Q)$ denotes the set of all regular languages over Q.

In the case of ranked tree automata, $\mathsf{Reg}(Q)$ would be restricted to the set of k-tuples over Q for symbols $a_k \in \Sigma$. Also note that this definition is **bottom-up** in the sense that the transitions are in the direction from children to parents. This does not matter as long as we allow non-determinism but we remark that for deterministic automata the top-down formulation is strictly less expressive. As we did before for finite word automata, we first define acceptance traditionally which incorporates the common notion of a **run** of an automaton on a tree:

Definition 3.6. *Let $\mathcal{A}$ be an UTA and $n \in \Sigma\text{-}\mathsf{Tree}$. A **run** of $\mathcal{A}$ on n is a function $\rho : \mathsf{dom}(n) \to Q$ such that for every $w \in \mathsf{dom}(n)$ with k successors $w_1, \dots, w_k$ there is a transition $(L, n(w), \rho(w)) \in \delta$ with $\rho(w_1) \dots \rho(w_k) \in L$. The **$q$-language** of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A}, q)$, is the set of all $n \in \Sigma\text{-}\mathsf{Tree}$ such that there is a run ρ of $\mathcal{A}$ on n such that $\rho(\epsilon) = q$.*

Note that, having acceptance relativised to a state, introducing a subset of final states is equally unnecessary as it was the case for initial states of NFAs. A run on a tree can be understood as a labelling that attaches a state to every node. Starting from the leaves, parents are labelled whenever there is an applicable transition taking the child labels into account. If this process does not get stuck, meaning it is possible to assign a label state to every node including the root, the automaton accepts the input tree from the root state.

As it was the case for finite word automata, the recognisable languages of both ranked and unranked tree automata are closed under boolean operations and all automaton models come with decidable acceptance and emptiness [CDG+07, CLT05]. We conclude this section by again bringing the definition of $\mathcal{L}(\mathcal{A}, q)$ in a compositional shape:

Definition 3.7. *We define the q-**language** $\mathcal{L}'(\mathcal{A}, q)$ of an UTA $\mathcal{A}$ inductively by:*

$$\frac{(L, a, q) \in \delta \quad q_1 \dots q_k \in L \quad n_i \in \mathcal{L}'(\mathcal{A}, q_i)}{an_1 \dots n_k \in \mathcal{L}'(\mathcal{A}, q)} \, (LCst)$$

As in Definition 2.7 the hypothesis $n_i \in \mathcal{L}'(\mathcal{A}, q_i)$ needs justification for all $1 \le i \le k$ and vanishes for $k = 0$.

We show both definitions equivalent by the following lemma:

Lemma 3.2. *For every UTA $\mathcal{A}$ and $q \in Q$ we have $\mathcal{L}(\mathcal{A}, q) = \mathcal{L}'(\mathcal{A}, q)$.*

Proof. We establish both inclusions by structural induction on $n \in \Sigma\text{-}\mathsf{Tree}$:

- Suppose $an_1 \dots n_k \in \mathcal{L}(\mathcal{A}, q)$, so there is a run ρ of $\mathcal{A}$ on $an_1 \dots n_k$ with $\rho(\epsilon) = q$. Since being a run, ρ witnesses a transition $(L, a, q) \in \delta$ with $\rho(1) \dots \rho(k) \in L$. Now set $\rho_i(w) := \rho(iw)$ and observe that ρ_i is a run on n_i with $\rho_i(\epsilon) = \rho(i)$. Hence we have $n_i \in \mathcal{L}(\mathcal{A}, \rho(i))$ and derive $n_i \in \mathcal{L}'(\mathcal{A}, \rho(i))$ inductively. This allows for applying the definitional rule of $\mathcal{L}'(\mathcal{A}, q)$ to conclude $an_1 \dots n_k \in \mathcal{L}'(\mathcal{A}, q)$.
- Now let $an_1 \dots n_k \in \mathcal{L}(\mathcal{A}, q)$ hence there is a transition $(L, a, q) \in \delta$ and $q_i \dots q_k \in L$ such that $n_i \in \mathcal{L}'(\mathcal{A}, q_i)$. Then by IH $n_i \in \mathcal{L}(\mathcal{A}, q_i)$ so there are runs ρ_i of $\mathcal{A}$ on n_i with $\rho_i(\epsilon) = q_i$. This time we set conversely $\rho(\epsilon) = q$ and $\rho(iw) := \rho_i(w)$ which yields a run ρ of $\mathcal{A}$ on $an_1 \dots n_k$ with $\rho(\epsilon) = q$. Thus $an_1 \dots n_k \in \mathcal{L}(\mathcal{A}, q)$. □

3.3 Basic Theory of Nominal Sets

Now that we have seen some classical automata we can move on to automata based on nominal sets. Therefore, we first summarise some theoretical background as presented in [Pit13]. Following the standard set-up, we first fix a countable set $\mathbb{A} = \{a, b, c, \dots\}$ of atomic names, that is black box objects providing no structure but their identity. Note that the set Var of variable names of lambda calculus is exactly such a set. In fact, syntax with binders is one original motivation for the more general theory of nominal sets.

Next we consider the group $\mathsf{Perm}(\mathbb{A})$ of finite permutations π on $\mathbb{A}$, that is, we have $\pi(a) \neq a$ for only finitely many $a \in \mathbb{A}$. Of course this defines a subgroup of the group $\mathsf{Sym}(\mathbb{A})$ of all permutations on $\mathbb{A}$. It is a standard result that every $\pi \in \mathsf{Perm}(\mathbb{A})$ can be written as a finite composition of transpositions of the form $(a\,b) \in \mathsf{Perm}(\mathbb{A})$ that swap $a, b \in \mathbb{A}$ and fix all other names. Then the underlying concept for nominal sets is the notion of an **action** of $\mathsf{Perm}(\mathbb{A})$ on a set:

Definition 3.8. *A **perm action** on a set X is a function $\cdot : \mathsf{Perm}(\mathbb{A}) \times X \to X$ with:*

$$\mathsf{id} \cdot x = x \qquad \pi \cdot (\pi' \cdot x) = (\pi \circ \pi') \cdot x$$

*We call a set X equipped with a perm action $\cdot$ a **perm set**.*

Note that the definition of a group action makes sense for arbitrary groups, in particular any other subgroup of $\mathsf{Perm}(\mathbb{A})$. This is examined to a certain extent in [BKL14] but we will only focus on actions of $\mathsf{Perm}(\mathbb{A})$ as those yield the standard theory of nominal sets from [Pit13].

We give some examples of such perm sets. First, note that every set X can be considered a perm set equipped with the trivial group action $\pi \cdot x := x$. Only slightly less trivial is the perm action $\pi \cdot a := \pi(a)$ on $\mathbb{A}$ itself. Furthermore, given two perm sets X, Y we inherit a perm action on $X \times Y$ by $\pi \cdot (x, y) := (\pi \cdot x, \pi \cdot y)$. Of course this works for tuples of arbitrary length as well as unordered subsets. The most important example for our purpose is syntax incorporating variables, for instance the terms of the lambda calculus over the set $\mathbb{A} := \mathsf{Var}$ from Chapter 2. The perm action on Λ is renaming of variables, defined recursively by:

- $\pi \cdot x := \pi(x)$
- $\pi \cdot (\lambda x.s) := \lambda \pi(x).\pi \cdot s$
- $\pi \cdot (st) := (\pi \cdot s)(\pi \cdot t)$

That this in fact yields a well defined perm action is established with a simple induction on $s \in \Lambda$. The treatment of ν-trees in Section 3.5 as perm set will be very similar. Every perm set comes with an induced equivalence relation as follows:

Definition 3.9. *Let X be a perm set and $x \in X$. The **orbit** of x is the set $\mathsf{Perm}(\mathbb{A}) \cdot x := \{\, \pi \cdot x \mid \pi \in \mathsf{Perm}(\mathbb{A}) \,\}$. We write $x \approx y$ to indicate that x and y have the same orbit or, equivalently, that there is π with $x = \pi \cdot y$. If X admits only finitely many equivalence classes of $\approx$ we call it **orbit-finite**.*

That $\approx$ defines an actual equivalence relation follows from the basic properties of $\mathsf{Perm}(\mathbb{A})$ being a group. We now introduce the concept of support and the derived notion of nominal sets:

Definition 3.10. *Let X be a perm set. We say that $A \subseteq \mathbb{A}$ **supports** $x \in X$ if every finite permutation $\pi \in \mathsf{Perm}(\mathbb{A})$ with $\pi(a) = a$ for all $a \in A$ already satisfies $\pi \cdot x = x$. Since the intersection of any two supports of x supports x again and we always have $\mathbb{A}$ supporting x there exists a minimal support denoted $\mathsf{supp}(x)$. If $\mathsf{supp}(x)$ is finite for all $x \in X$ we call X **nominal**.*

*Subsets $Y \subseteq X$ of a nominal set X are called **equivariant** if $\pi \cdot Y := \{\, \pi \cdot y \mid y \in Y \,\} = Y$ for all π. In particular, functions $f \subseteq X \times Y$ for nominal X, Y are equivariant if $\pi \cdot f(x) = f(\pi \cdot x)$.*

Intuitively, a set is nominal if every element only depends on finitely many names. Subsets are equivariant exactly if they are a union of orbits and functions are equivariant if they do not depend on concrete names.

It is easy to verify that nominal sets as objects and equivariant functions as morphisms form a well-defined category **Nom**. That this category provides all the defining structure of a Boolean topos is a much more involved result (cf. Theorem 2.23 in [Pit13]) and is beyond the scope of this dissertation.

Reconsidering the examples from above, we can state the following. First, arbitrary sets X equipped with the trivial group action $\pi \cdot x = x$ are of course nominal given that every $x \in X$ has already empty support. Since all orbits are singletons, this nominal set is orbit-finite if and only if it is finite.

The perm set $\mathbb{A}$ with $\pi \cdot a = \pi(a)$ is nominal given that every $a \in \mathbb{A}$ is supported by $\{a\}$. Moreover, since for any b we have $a = (a\,b) \cdot b$ the set comes with the single orbit $\mathbb{A}$.

Derived structure such as the product $X \times Y$ of nominal X, Y is nominal again. For $(x, y) \in X, Y$ we obtain a finite support by taking the union of the finite supports of x and y, respectively. The same works for finite subsets but infinite subsets, however, are not necessarily nominal again. For instance, only the finite and cofinite subsets of $\mathbb{A}$ are nominal.

If we consider syntax with $\mathbb{A}$ as variable names the generated set of terms is nominal as long as the terms are finite. As soon as the grammar admits infinite terms with infinitely many distinct variables the supports become infinite. On the other hand, already finite grammars may allow infinitely many distinct shapes of terms which results in infinitely many orbits. For our running example of lambda calculus it is the case that Λ is nominal but orbit-infinite.

The last concept we introduce here is called **name abstraction**. This captures the idea of treating elements of nominal sets independent from their concrete naming. For instance, it is common to identify the lambda terms Λ which are equal up to permutation of their bound variables, so called α-equivalence. Then it is natural to consider equivalence classes of $s \in \Lambda$ containing every permutation of a bound variable x. In a next step one can obtain all instantiations for fresh names $y \in \mathsf{Var}$ by defining a **concretion** operation $(<x> s) \mathbin{@} y$. We make this idea precise in the general case as follows:

Definition 3.11. *Let X be a nominal set. We define an equivalence relation on $\mathbb{A} \times X$ by writing $(a_1, x_1) \equiv (a_2, x_2)$ if there is some $a\#(a_1, x_1, a_2, x_2)$ such that $(a_1\, a) \cdot x_1 = (a_2\, a) \cdot x_2$. Here, the freshness condition $a\#(x_1, \dots, x_k)$ expresses that $a \notin \mathsf{supp}(x_i)$ for all $1 \leq i \leq k$. We denote the equivalence class of (a, x) by $<a> x$ and call it the a-**name abstraction of x**.*

For the sake of completeness we just mention some of the properties of this construction. First, the equivalence relation $\equiv$ itself is equivariant, that is we have $\pi \cdot (a_1, x_1) \equiv \pi \cdot (a_2, x_2)$ whenever we have $(a_1, x_1) \equiv (a_2, x_2)$. Secondly, $\pi\cdot <a> x :=<\pi(a)> (\pi \cdot x)$ defines a perm action on the quotient $[\mathbb{A}]X := X/_{\equiv}$. It follows that $[\mathbb{A}]X$ can be considered a nominal set. Finally, we can easily define the hinted concretion operation to either recover the element $x \in X$ we started with or to obtain all other meaningful instantiations $(a\,b) \cdot x$:

Definition 3.12. *For nominal X we define the partial* ***concretion*** $@ : [\mathbb{A}]X \times \mathbb{A} \rightharpoonup X$*:*

$$(<a> x) \mathbin{@} b := \begin{cases} x & \text{if } b = a \\ (a\,b) \cdot x & \text{if } b \neq a \text{ and } b \# x \end{cases}$$

In the case where $b \neq a$ but $b \in \mathsf{supp}(x)$ we leave $(<a> x) \mathbin{@} b$ undefined.

We will see examples of name abstraction and concretion in Section 3.5.

3.4 Nominal Word Automata

We now know enough about nominal sets to study the nominal word automata as defined by Bojanczyk et al. [BKL14]. In fact, they are exactly the same as finite automata where the required finite components are relaxed to be orbit-finite nominal sets. We discuss the strictly more expressive non-deterministic model:

Definition 3.13. *A* ***non-deterministic nominal automaton (NNA)*** *$\mathcal{A}$ consists of:*

- *Σ: an orbit-finite nominal alphabet of symbols*
- *Q: an orbit-finite nominal finite set of states*
- *$F \subseteq Q$: an equivariant subset of final states*
- *$\delta \subseteq Q \times \Sigma \times Q$: an equivariant transition relation*

As we did for NFA, we write $q \xrightarrow{a} q'$ if $(q, a, q') \in \delta$ and for $w = a_1 \dots a_n \in \Sigma^*$ we write $q \xrightarrow{w} q'$ if $q \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q'$ for some $q_1, \dots, q_{n-1} \in Q$.

Acceptance of words $w \in \Sigma$ and ultimately the q-language $\mathcal{L}(\mathcal{A}, q)$ are defined equally as for NFA. Now recall that the set $\mathbb{A}$ of atomic names is nominal with a single orbit. Hence it makes an example alphabet for NNA. Since the NNA-recognised languages with this choice of alphabet are equivariant subsets of $\mathbb{A}^*$ they express properties that must be independent from the concrete names. For instance, the regular languages of style "all words that contain at least two a" are not expressible for this alphabet. Instead, the acceptable languages express more general properties such as "all words that contain some letter at least two times".

To study an example, we give an automaton accepting the above language:

- $\Sigma := \mathbb{A}$, the countably infinite set of all atomic names.
  Recall that a trivial perm action can be defined by $\pi \cdot a := \pi(a)$ with $\mathsf{supp}(a) = \{a\}$ which makes it a nominal set. Moreover, the single orbit of Σ is of course Σ itself.

- $Q := \{\star\} \cup \mathbb{A} \cup \mathbb{A}^2$
  We define a perm action component-wise by $\pi(\star) := \star$, $\pi \cdot a := \pi(a)$ for $a \in \mathbb{A}$ and $\pi \cdot ab := \pi(a)\pi(b)$ for $ab \in \mathbb{A}^2$. This yields a nominal set given that $\mathsf{supp}(\star) = \emptyset$, $\mathsf{supp}(a) = \{a\}$ and $\mathsf{supp}(ab) = \mathsf{supp}(a) \cup \mathsf{supp}(b) = \{a, b\}$. The four orbits of Q are $\{\star\}$, $\mathbb{A}$, $\{\, aa \mid a \in \mathbb{A} \,\}$ and $\{\, ab \mid a \neq b \,\}$.

- $F := \mathbb{A}^2$
  This is equivariant since for any $\pi \in \mathsf{Perm}(\mathbb{A})$ and $ab \in \mathbb{A}^2$ we have $\pi \cdot ab = \pi(a)\pi(b) \in \mathbb{A}^2$. Equivalently, this can be justified by just mentioning that $\mathbb{A}^2$ is a union of two orbits of Q.
- $\delta := \left\{\star \xrightarrow{a} \star\right\} \cup \left\{\star \xrightarrow{a} a\right\} \cup \left\{a \xrightarrow{b} a\right\} \cup \left\{a \xrightarrow{a} aa\right\} \cup \left\{aa \xrightarrow{b} aa\right\}$ for all $a, b \in \mathbb{A}$
  Note that this in particular includes transitions $a \xrightarrow{a} a$ and $aa \xrightarrow{a} aa$ for all $a \in \mathbb{A}$. The relation is equivariant since each form of transitions is a union of orbits of $Q \times \mathbb{A} \times Q$.

Although this automaton contains infinite components, it works quite similar to a standard finite automaton accepting words that contain a concrete symbol at least twice. First, on the initial state $\star$ a loop for all names is admitted. Then at any point a consumed name $a \in \mathbb{A}$ can be used to transition to state a. Once in this state, the automaton will only accept if a second a triggering the transition to the state aa is encountered. It is straight-forward to establish formally that every $\mathbb{A}$-word with some name appearing at least twice can be processed by the automaton starting in $\star$ and ending in a final state and, conversely, that no word with all names distinct is accepted.

Note that we only consider NNA based on the group $\mathsf{Perm}(\mathbb{A})$. As shown in the article [BKL14], the languages of nominal automata of this form are closed under union and intersection. However, complementation of languages is not provided in general, the example used in the article is exactly the above language of words containing at least one letter twice since it can be shown that there is no NNA that accepts the complement language of words with all letters distinct.

Moreover, although the components of NNA may well be infinite, it is possible to derive decision procedures for emptiness and acceptance if one is given concrete representatives for all state, alphabet and transition orbits. This is due to the fact that the orbit equivalence $\approx$ can be shown decidable for every nominal set and hence particular states and symbols can be tested to be members of orbits of Q and Σ. Then, similarly as for NFA, an algorithm just has to test the effectively finitely many transitions and final states in order to determine a decision.

3.5 Ranked ν-Trees

The ν-calculus as developed by Stark and Pitts [PS93, Sta98] and, similarly, the $\lambda\nu$-calculus of Odersky [Ode94] are two extensions of the ordinary λ-calculus with binders for new atomic names and constants for booleans. That is, the usual grammar of λ-terms is extended with a binder $\nu a.s$ introducing a fresh atomic name $a \in \mathbb{A}$ that might occur in the body s. The main use of names in the original calculi is to compute boolean values depending on freshness constraints. Since we are interested in trees as input for automata, we first discuss lambda-free ν-trees and postpone studying our fragment of the full calculi to Section 4.6.

Said ν-trees give a finite representation of possibly infinite sets of trees over the infinite alphabet $\mathbb{A}$ via the binding of unused names. We intend to develop an automaton model in the style of Stirling's dependency tree automata [Sti09] which is capable of deciding properties of ν-trees and hence also properties of pure $\mathbb{A}$-trees. This use of ν-trees is very different from the use in ν-calculus and $\lambda\nu$-calculus which is why we develop our own semantics of this class of structures. In fact, our use of ν extends Kozen's work on ν-strings [KMS15] to tree structures, which to the best of our knowledge has not been done in this way.

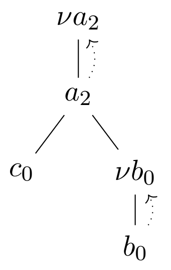
To simplify the automaton model, we assume $\mathbb{A}$ to be ranked for the remainder of this document. In fact, to obtain an infinite supply of fresh names of arbitrary rank, we assume $\mathbb{A}$ to be countably infinite on every rank. In particular, in this setting the group $\mathsf{Perm}(\mathbb{A})$ only contains finite **rank-preserving** permutations, that is for all $\pi \in \mathsf{Perm}(\mathbb{A})$ and $a_k \in \mathbb{A}$ we have $\mathsf{rk}(\pi(a_k)) = k$. Then we define the set of $\mathbb{A}$-trees with nus as follows:

Definition 3.14. *The set ν-*Tree *of **ranked ν-trees** is generated by:*

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

Here the names a_k range over $\mathbb{A}$ and we assume all binders in a tree to be distinct, that is, if n contains a nu νa_k then all other bound variables differ from a_k.
The set $\mathsf{Nme}(n)$ denotes all names occurring in n, whether in a nu, free or bound.
*We denote the **free names** of a ν-tree n by $\mathsf{FN}(n)$. If $\mathsf{FN}(n) = \emptyset$ we call n **closed**.*

Obviously, ν-trees do not contain lambdas and are well-ranked. Hence we can identify ν-trees $n \in \nu$-Tree with functions from the address domain $\mathsf{dom}(n)$ to the syntax nodes of n together with a partial binding function $\rightsquigarrow\colon \mathsf{dom}(n) \rightharpoonup \mathsf{dom}(n)$ that maps each bound variable a_k to its binder νa_k. For instance consider the following ν-tree with the binding function depicted by the dotted arrows:



From this perspective, ν-trees can be seen as a special instance or a streamlining of the binding trees used by Stirling [Sti09] whose definition allows an arbitrary alphabet of binders but puts some structural restrictions on the tree construction. In our formulation we only allow a single form of binders, namely the nodes νa_k for a single fresh name. On the other hand, our notion is a relaxation since we do not require all names to be bound and allow children of names to be names again (whereas Stirling uses dummy binders to achieve a regular alternation). Hence our definition naturally fits to the terms that arise from execution of programs and moreover allows for studying properties like closedness and alternation.

In this section we study how single ν-trees n can be used to denote subsets $[\![n]\!] \subseteq$ $\mathbb{A}$-Tree of pure $\mathbb{A}$-trees (cf. Definition 3.4). The aim is to obtain a denotational semantics in the sense that ν-trees equivalent in a certain way get assigned the same denotation. This semantics will employ the language of nominal sets and hence justifies classifying the soon to be defined ν-tree automata as nominal.

Before we give the definition of the denotation $[\![n]\!]$ we set up the framework to apply the formalism of nominal sets. First, we can equip pure $\mathbb{A}$-trees $m \in \mathbb{A}$-Tree with a group action by setting $\pi \cdot (a_k m_1 \dots m_k) \coloneqq \pi(a_k)(\pi \cdot m_1) \dots (\pi \cdot m_k)$ for finite rank-preserving permutations $\pi \in \mathsf{Perm}(\mathbb{A})$. Since these trees are finite and hence contain only finitely many names it follows that $\mathbb{A}$-Tree is a nominal set. Obviously, $\mathbb{A}$-Tree is far from being orbit-finite given that there exist already infinitely many strings of different length. Secondly, we can add the definitional rule $\pi \cdot (\nu a_k.n) \coloneqq \nu\pi(a_k).\pi \cdot n$ to obtain a group action on ν-trees $n \in \nu$-Tree. For the same reasons as above this makes ν-Tree nominal but not orbit-finite.

Now we move to the definition of $[\![n]\!]$. The idea is that we want $m_1, m_2 \in [\![n]\!]$ iff m_1 and m_2 are α-equivalent with respect to the bound names and all nus are instantiated with fresh names. As we will see later, this form of parametrised α-equivalence is related to name abstraction as defined in [Pit13] and in Section 3.3. The following examples should provide some intuition:

$$\begin{aligned}
[\![a]\!] &= \{a\} \\
[\![\nu a.a]\!] &= \{a \mid a \in \mathbb{A}\} = \mathbb{A} \\
[\![\nu a.ab]\!] &= \{ab \mid a \in \mathbb{A} \setminus \{b\}\} \\
[\![a(\nu b.bc)]\!] &= \{abc \mid b \in \mathbb{A} \setminus \{a, c\}\} \\
[\![a(\nu b.b)(\nu c.c)]\!] &= \{abc \mid b, c \in \mathbb{A} \setminus \{a\}\} \\
[\![a(\nu b.b(\nu c.c))]\!] &= \{a(bc) \mid b, c \in \mathbb{A} \setminus \{a\} \wedge c \neq b\}
\end{aligned}$$

To improve readability, we left out the rank indices at the names and will do so where possible and reasonable for the rest of the section. Note that it is a design choice in order to, for instance, have $abb \in [\![a(\nu b.b)(\nu b.b)]\!]$ and $aac \notin [\![a(\nu b.bc)]\!]$. The intuition we follow is the way how variables are treated in programming languages. In distinct branches of a conditional (such as in $a(\nu b.b)(\nu b.b)$) we may well introduce the same variable (meaning abb) but if there are globally or earlier defined variables (like a and c in $a(\nu b.bc)$) we want to avoid renaming them (which would be done by the instantiation aac). We will compare our decisions to the work by Stark, Pitts and Odersky in Section 4.5.

We now give the formal definition. To allow for this to be compositional and hence to enable inductive proofs, we parametrise to an additional argument that carries the already used names as follows:

Definition 3.15. *We define functions* $[\![-]\!]_A : \nu\text{-}\mathsf{Tree} \to \mathcal{P}(\mathbb{A}\text{-}\mathsf{Tree})$ *for* $A \in \mathcal{P}_{\mathrm{fin}}(\mathbb{A})$ *by*

$$\frac{m_i \in [\![n_i]\!]_{A\cup\{a_k\}}}{a_k m_1 \dots m_k \in [\![a_k n_1 \dots n_k]\!]_A} \qquad \frac{m \in [\![(a_k\, b_k)\cdot n]\!]_{A\cup\{b_k\}} \quad b_k \notin A \cup \mathsf{FN}(\nu a_k.n)}{m \in [\![\nu a_k.n]\!]_A}$$

We can now write $[\![n]\!]$ instead of $[\![n]\!]_\emptyset$. Then it easy to verify that all of the above examples are correct with respect to our formal definition. An important property of the just defined function is equivariance:

Lemma 3.3. *The function* $[\![-]\!]_-$ *is equivariant, that is, we have* $\pi \cdot [\![n]\!]_A = [\![\pi \cdot n]\!]_{\pi\cdot A}$ *for all finite rank-preserving permutations* π*,* ν*-trees* n *and finite subsets* $A \subseteq \mathbb{A}$*.*
In particular $[\![-]\!]$ *is equivariant, that is* $\pi \cdot [\![n]\!] = [\![\pi \cdot n]\!]$*.*

Proof. We prove the first claim by structural induction on $n \in \nu\text{-}\mathsf{Tree}$:

- We establish $\pi \cdot [\![a n_1 \dots n_k]\!]_A = [\![\pi \cdot (a n_1 \dots n_k)]\!]_{\pi\cdot A}$ by:

$$\begin{aligned}
& \pi \cdot (a m_1 \dots m_k) \in \pi \cdot [\![a n_1 \dots n_k]\!]_A \\
\Leftrightarrow\ & a m_1 \dots m_k \in [\![a n_1 \dots n_k]\!]_A \\
\overset{def}{\Leftrightarrow}\ & m_i \in [\![n_i]\!]_{A\cup\{a\}} \\
\overset{IH}{\Leftrightarrow}\ & \pi \cdot m_i \in [\![\pi \cdot n_i]\!]_{\pi\cdot(A\cup\{a\})} \\
\Leftrightarrow\ & \pi \cdot m_i \in [\![\pi \cdot n_i]\!]_{\pi\cdot A\cup\{\pi(a)\}} \\
\overset{def}{\Leftrightarrow}\ & \pi(a)(\pi\cdot m_1)\dots(\pi\cdot m_k) \in [\![\pi(a)(\pi\cdot n_1)\dots(\pi\cdot n_k)]\!]_{\pi\cdot A} \\
\Leftrightarrow\ & \pi \cdot (a m_1 \dots m_k) \in [\![\pi \cdot (a n_1 \dots n_k)]\!]_{\pi\cdot A}
\end{aligned}$$

- We establish $\pi \cdot [\![\nu a.n]\!]_A = [\![\pi \cdot (\nu a.n)]\!]_{\pi\cdot A}$ by:

$$\begin{aligned}
& \pi \cdot m \in \pi \cdot [\![\nu a.n]\!]_A \\
\Leftrightarrow\ & m \in [\![\nu a.n]\!]_A \\
\overset{def}{\Leftrightarrow}\ & m \in [\![(a\, b)\cdot n]\!]_{A\cup\{b\}} \wedge b \notin A \cup \mathsf{FN}(\nu a.n) \\
\overset{IH}{\Leftrightarrow}\ & \pi \cdot m \in [\![\pi \cdot ((a\, b)\cdot n)]\!]_{\pi\cdot(A\cup\{\beta\})} \wedge b \notin A \cup \mathsf{FN}(\nu a.n) \\
\overset{(*)}{\Leftrightarrow}\ & \pi \cdot m \in [\![(\pi(a)\,\pi(b))\cdot(\pi\cdot n)]\!]_{\pi\cdot A\cup\{\pi(b)\}} \wedge \pi(b) \notin \pi\cdot A \cup \mathsf{FN}(\pi\cdot(\nu a.n)) \\
\overset{def}{\Leftrightarrow}\ & \pi \cdot m \in [\![\nu\pi(a).\pi\cdot n]\!]_{\pi\cdot A} \\
\Leftrightarrow\ & \pi \cdot m \in [\![\pi \cdot (\nu a.n)]\!]_{\pi\cdot A}
\end{aligned}$$

Note that we used the simple fact that FN is an equivariant function at $(*)$. The second claim follows since $\pi \cdot \emptyset = \emptyset$ for all π. □

Since the $[\![n]\!]_A$ denote subsets of the nominal set $\mathbb{A}$-Tree it is reasonable to examine when those subsets are equivariant themselves, that is, when $\pi \cdot [\![n]\!]_A = [\![n]\!]_A$ holds for all $\pi \in \mathsf{Perm}(\mathbb{A})$. This is obviously not true if n contains free names, for instance $[\![a]\!] = \{a\}$ is not equivariant since $(a\, b)\cdot\{a\} = \{b\}$. Moreover, allowing non-empty A causes problems since $b \notin [\![\nu a.a]\!]_{\{b\}}$ but for $a \neq b$ we have $b \in (a\, b)\cdot[\![\nu a.a]\!]_{\{b\}}$. A general formulation of this insight can be stated as follows:

Lemma 3.4. *Let n be an arbitrary ν-tree and let π be a finite rank-preserving permutation that fixes all free names of n, that is, let $\pi(a_k) = a_k$ for all $a_k \in \mathsf{FN}(n)$.*
Then $[\![\pi \cdot n]\!]_A = [\![n]\!]_A$ holds.

Proof. We prove this claim by structural induction on $n \in \nu\text{-}\mathsf{Tree}$:

- The following shows $[\![\pi \cdot (an_1 \dots n_k)]\!]_A = [\![an_1 \dots n_k]\!]_A$:

$$\begin{aligned}
& am_1 \dots m_k \in [\![\pi \cdot (an_1 \dots n_k)]\!]_A \\
\Leftrightarrow\ & am_1 \dots m_k \in [\![a(\pi \cdot n_1) \dots (\pi \cdot n_k)]\!]_A \\
\Leftrightarrow\ & m_i \in [\![\pi \cdot n_i]\!]_{A \cup \{a\}} \\
\overset{IH}{\Leftrightarrow}\ & m_i \in [\![n_i]\!]_{A \cup \{a\}} \\
\Leftrightarrow\ & am_1 \dots m_k \in [\![an_1 \dots n_k]\!]_A
\end{aligned}$$

  The IH was applicable since, given that $\mathsf{FN}(n_i) \subseteq \mathsf{FN}(an_1 \dots n_k)$, if π fixes the free names of $an_1 \dots n_k$ it fixes the free names of each n_i.

- The following shows $[\![\pi \cdot (\nu a.n)]\!]_A = [\![\nu a.n]\!]_A$:

$$\begin{aligned}
& m \in [\![\pi \cdot (\nu a.n)]\!]_A \\
\Leftrightarrow\ & m \in [\![\nu \pi(a).\pi \cdot n)]\!]_A \\
\Leftrightarrow\ & m \in [\![(\pi(a)\, b) \cdot (\pi \cdot n)]\!]_{A \cup \{b\}} \wedge b \notin A \cup \mathsf{FN}(\nu \pi(a).\pi \cdot n) \\
\Leftrightarrow\ & m \in [\![((\pi(b)\, b) \circ \pi) \cdot ((a\, b) \cdot n)]\!]_{A \cup \{b\}} \wedge b \notin A \cup \pi \cdot \mathsf{FN}(\nu a.n) \\
\overset{IH}{\Leftrightarrow}\ & m \in [\![(a\, b) \cdot n]\!]_{A \cup \{b\}} \wedge b \notin A \cup \pi \cdot \mathsf{FN}(\nu a.n) \\
\overset{(*)}{\Leftrightarrow}\ & m \in [\![(a\, b) \cdot n]\!]_{A \cup \{b\}} \wedge b \notin A \cup \mathsf{FN}(\nu a.n) \\
\Leftrightarrow\ & m \in [\![\nu a.n]\!]_A
\end{aligned}$$

  The IH was applicable since $(\pi(b)\, b) \circ \pi$ fixes all free names of $(a\, b) \cdot n$. To see this, note that $\mathsf{FN}((a\, b) \cdot n) = \mathsf{FN}(\nu a.n) \cup \{b\}$ and recall that we assume π to be fixed on $\mathsf{FN}(\nu a.n)$. Then $(\pi(b)\, b) \circ \pi(b)$ concludes the justification. The step at $(*)$ is just the assumed property of π. $\square$

Then we can establish the following consequence:

Lemma 3.5. *If n is a closed ν-tree then $[\![n]\!] \subseteq \nu\text{-}\mathsf{Tree}$ is equivariant.*

Proof. Note that for closed n we have $\mathsf{FN}(n) = \emptyset$ so Lemma 3.4 is applicable for all π. Then we prove equivariance of $[\![n]\!]$ by $\pi \cdot [\![n]\!] = [\![\pi \cdot n]\!] = [\![n]\!]$. $\square$

Note that in the definition of $[\![-]\!]_-$ it suffices to restrict to finite sets A of used names since, when processing a finite tree, only finitely many names will be touched. Given that $\mathcal{P}_{\text{fin}}(\mathbb{A})$ is nominal since finite subsets of nominal sets are supported by the finite union of the supports of all their elements, the function $[\![-]\!]_-$ lives in a nominal argument space. Contrarily, the set $\mathcal{P}(\mathbb{A}\text{-}\mathsf{Tree})$ is not nominal in total given that it contains for instance sets $D \subseteq \mathbb{A}$ that are neither finite nor cofinite. However, we can show that $[\![-]\!]_-$ still lives in a nominal value space:

Lemma 3.6. *The set* $\mathsf{ran}([\![-]\!]_A)$ *is nominal for all* $A \in \mathcal{P}_{\text{fin}}(\mathbb{A})$.

Proof. We show that $[\![n]\!]_A \in$ has finite support $\mathsf{Nme}(n) \cup A$. First recall that this set is finite since we consider finite trees and have $A \in \mathcal{P}_{\text{fin}}(\mathbb{A})$. Now let π fix $\mathsf{Nme}(n) \cup A$, so $\pi(a_k) = a_k$ for all $a_k \in \mathsf{Nme}(n) \cup A$. Then in particular $\pi \cdot n = n$ and $\pi \cdot A = A$ so we have $\pi \cdot [\![n]\!]_A = [\![\pi \cdot n]\!]_{\pi \cdot A} = [\![n]\!]_A$. □

In other words, with Lemma 3.3 and 3.6 we conclude that $[\![-]\!]_-$ is a proper morphism in the category **Nom** of nominal sets and equivariant functions. Also note that Lemma 3.6 is an instance of the more general insight that equivariant functions with nominal domain have a nominal range (cf. Lemma 2.12 in [Pit13]).

By now we have developed enough machinery to give a concise formulation and proof of the correctness of our denotational semantics. As mentioned above, correctness relies on a notion of equivalence of terms. In the context of nominal sets, a natural choice would be the orbit-equivalence $n \approx n'$ from Section 3.3 expressing that there is a permutation π with $n = \pi \cdot n'$. However, since our denotation function distinguishes distinct free names, we will mainly use the slightly finer α-equivalence of bound names:

Definition 3.16. *Two ν-trees n, n' are α-**equivalent**, written $n \approx_\alpha n'$, if $n = \pi \cdot n'$ for a finite rank-respecting permutation π that fixes all free names of n.*

Note that if $n = \pi \cdot n'$ we have $\mathsf{FN}(n) = \pi \cdot \mathsf{FN}(n')$ so we could as well have required π to fix all free names of n' or simply both. The following fact makes the relationship between both equivalences precise:

Fact 3.7. *The following two statements hold for arbitrary $n, n' \in \nu$-*Tree*:*
(1) If $n \approx_\alpha n'$ then $n \approx n'$.
(2) If $n \approx n'$ and n, n' are closed then $n \approx_\alpha n'$.

Proof. Both are trivial, for (2) recall that $\mathsf{FN}(n) = \emptyset$ if n closed. □

We first prove the denotation function correct with respect to α-equivalence:

Theorem 3.8. *For arbitrary n, n' and A we have $n \approx_\alpha n'$ iff $[\![n]\!]_A = [\![n']\!]_A$.*

Proof. The soundness direction follows immediately from Lemma 3.4. Suppose $n \approx_\alpha n'$ so there is π with $n = \pi \cdot n'$ that fixes the free names of n and n'. Then $[\![n]\!]_A = [\![\pi \cdot n']\!]_A = [\![n']\!]_A$ where we applied Lemma 3.4 in the last step. We prove the completeness direction by mutual induction on $n, n' \in \nu$-Tree:

- Suppose $[\![an_1 \dots n_k]\!]_A = [\![an'_1 \dots n'_k]\!]_A$. By simply inverting the first definitional rule of $[\![-]\!]_-$ we observe $[\![n_i]\!]_{A \cup \{a\}} = [\![n'_i]\!]_{A \cup \{a\}}$. This is subject to the IH, so there are π_i witnessing $n_i \approx_\alpha n'_i$. Since we assume all binders within $an_1 \dots n_k$ to be distinct the π_i are mutually compatible. Thus we can define the permutation $\pi := \pi_1 \circ \cdots \circ \pi_k$ that still satisfies $\pi \cdot (an'_1 \dots n'_k) = \pi(a)(\pi \cdot n'_1) \dots (\pi \cdot n'_k) = an_1 \dots n_k$. Moreover, since π fixes a itself and all free names of the n_i we conclude $an_1 \dots n_k \approx_\alpha an'_1 \dots n'_k$.

- Now suppose $[\![\nu a.n]\!]_A = [\![\nu a'.n']\!]_A$. By a similar observation as above we obtain $[\![n]\!]_A = [\![(a\,a') \cdot n']\!]_A$. Then the IH yields some π witnessing $n \approx_\alpha (a\,a') \cdot n$. We easily establish $\nu a.n \approx_\alpha \nu a'.n'$ by considering the permutation $\pi \circ (a\,a')$. □

Then a weaker form for orbit-equivalence follows for closed trees:

Corollary 3.9. *For closed n, n' we have $n \approx n'$ iff $[\![n]\!]_A = [\![n']\!]_A$.*

Proof. This follows immediately from Theorem 3.8 and Lemma 3.7. □

We conclude this section by studying the similarity of our denotation and name abstraction from [Pit13] and Section 3.3. Intuitively, $\nu a.n$ and $<a>\,n$ have the same effect in denoting the equivalence class of n with all occurences of a replaced by names b that are not already mentioned in n. Of course we have no exact equality since $[\![\nu a.n]\!] \subseteq \mathbb{A}\text{-Tree}$ whereas $<a>\,n \subseteq \mathbb{A} \times \nu\text{-Tree}$ but we can state the similarity as follows:

Lemma 3.10. *If $n \in \mathbb{A}\text{-Tree}$ we have $[\![\nu a.n]\!] = \{<a>\,n \,@\, b \mid b \in \mathbb{A}\} =: S$.*

Proof. Let $m \in [\![\nu a.n]\!]$ so $m \in [\![(b\,a)\cdot n]\!]$ for some $b \in \mathbb{A}\backslash\mathsf{FN}(\nu a.n)$. By equivariance $(b\,a) \cdot m \in [\![n]\!]$ and since $n \in \mathbb{A}\text{-Tree}$ implies $[\![n]\!] = \{n\}$ we have $m = (b\,a) \cdot n$. Now since $(b\,a) \cdot n = <a>\,n \,@\, b$ we conclude $m \in S$. The second inclusion is similar. □

If we aim at generalising this result to any ν-trees we can only go as far as:

Lemma 3.11. $[\![\nu a.n]\!] \supseteq \{<a>\,m \,@\, b \mid b \in \mathbb{A} \wedge m \in [\![n]\!]\} =: S$

Proof. Let $<a>\,m \,@\, b \in S$ so $m \in [\![n]\!]$. If $a = b$ we have $<a>\,m \,@\, b = m$ and we clearly have $m \in [\![\nu a.n]\!]$ given that $m \in [\![n]\!]$. If $b\#m$ we have $<a>\,m \,@\, b = (a\,b)\cdot m$. By $m \in [\![n]\!]$ we have $(a\,b) \cdot m \in [\![(a\,b) \cdot n]\!]$ so $(a\,b) \cdot m \in [\![\nu a.n]\!]$. In any other case $<a>\,m \,@\, b$ is undefined. □

In fact, the converse direction is not necessarily true. For instance, we have $d(ac) \in [\![\nu a.a(\nu b.bc)]\!]$ since $d(ac) \in [\![(a\,d) \cdot (a(\nu b.bc))]\!]$. However, we fail at writing $d(ac)$ as $<a>\,a(bc) \,@\, d$ for $a \neq b \neq c$. This just shows that our denotation function is similar but not exactly the same as all possible conrections of name abstraction.

3.6 Ranked ν-Tree Automata

In this section we define an automaton model suitable to decide properties of ν-trees. Given the denotational semantics of ν-trees from the previous section, the accepted languages $\mathcal{L}$ can be lifted to languages of $\mathbb{A}$-trees by taking the union of all denotations $[\![n]\!]$ for $n \in \mathcal{L}$. We roughly follow Stirling's definition of nondeterministic dependency tree automata in [Sti09].

Definition 3.17. *A **ranked ν-tree automaton (NTA)** $\mathcal{A}$ consists of the following data:*

- $\Sigma \subset \mathbb{A}$*: a finite ranked alphabet*
- $Q = \{q_1, \dots q_m\}$*: a finite set of states*
- $L = \{l_1, \dots l_n\}$*: a finite set of labels*
- Δ*: a finite set of transitions of the following form:*
  - *(I)* $(q, a_k) \Rightarrow (q_1, \dots, q_k)$
  - *(II)* $((q, l), a_k) \Rightarrow (q_1, \dots, q_k)$
  - *(III)* $(q, \nu a_k) \Rightarrow (q', l)$

We do not require ranked ν-tree automata to be deterministic, that is, we allow distinct rules that agree on their left-hand sides. We abbreviate the statement $R \in \Delta$ for a rule R by just stating R.

First note that the direction of the transitions is reversed in comparison to the definition of UTA in Section 3.2. So contrarily, the defined ν-tree automata work in a **top-down** fashion. The idea behind that is to follow the binding structure more intuitively, meaning that whenever a bound name is encountered the next transition already depends on a unique label previously assigned to the corresponding nu. This will become more clear once we have defined runs in the upcoming Definition 3.18. Also note that, allowing non-determinism, we could still define the transitions bottom-up which would leave to guess the right label when processing bound names.

As we will see at the end of this section, employing only finite subsets $\Sigma \subset \mathbb{A}$ and hence only allowing finitely many transitions admits decidable acceptance and emptiness. This restriction is not too expensive since the binding of a single new name can express properties involving infinitely many names. For instance, if an NTA accepts the tree $\nu a.a$ then the lifted language on the level of $\mathbb{A}$-trees will include the whole of $\mathbb{A}$ given that $[\![\nu a.a]\!] = \mathbb{A}$. This is still way beyond the scope of finite tree automata on trees without name binding.

In comparison to our definition of ranked ν-tree automata, Stirling's dependency tree automata run on general binding trees as they were discussed in Section 3.5. Hence they admit an arbitrary alphabet of binders and using nus is a mere instance. However, we alter the definition in adding labels to express the non-local dependency of applicable transitions whereas Stirling reuses the states. We believe that separating these two control functionalities is a simplification of the definition of concrete automata where binding symbols can be coarsely grouped by states and then finely specified by labels. Also, as mentioned in the previous section, our automaton model is more general since we put less structural restrictions like dummy binders on the input trees, allowing for testing properties like closedness. Note that we could also give a generalised definition independent from the concrete use of ν-trees which would yield a strictly more expressive automaton model.

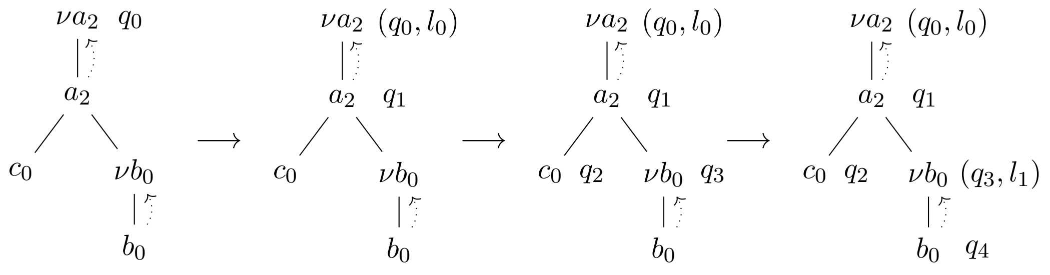
Definition 3.18. *A (q_0, φ)-**run** of an NTA $\mathcal{A}$ on a ν-tree n for a state $q_0 \in Q$ and a partial label assignment $\varphi : \Sigma \rightharpoonup L$ is a partial function $\rho : \mathsf{dom}(n) \rightharpoonup Q \cup (Q \times L)$*

defined as follows. First set $\rho(\epsilon) := q_0$. Then iteratively, for $w \in \mathsf{dom}(n)$ with $\rho(w) = q$ lastly defined we set:

- *If $n(w) = a_k$ and $w \notin \mathsf{dom}(\rightsquigarrow)$ and $a_k \notin \mathsf{dom}(\varphi)$ and there is a transition $(q, a_k) \Rightarrow (q_1, \ldots, q_k)$ we set $\rho(wi) := q_i$ for all $1 \leq i \leq k$.*
- *If $n(w) = a_k$ and $w \rightsquigarrow w'$ and $\rho(w') = (q', l)$ and there is a transition $((q, l), a_k) \Rightarrow (q_1, \ldots, q_k)$ we set $\rho(wi) := q_i$ for all $1 \leq i \leq k$.*
- *If $n(w) = a_k$ and $w \notin \mathsf{dom}(\rightsquigarrow)$ but $\varphi(a_k) = l$ and there is a transition $((q, l), a_k) \Rightarrow (q_1, \ldots, q_k)$ we set $\rho(wi) := q_i$ for all $1 \leq i \leq k$.*
- *If $n(w) = \nu a_k$ and there is a transition $(q, \nu a_k) \Rightarrow (q', l)$ we first update $\rho(w) := (q, l)$ and then set $\rho(w1) = q'$.*

*This procedure is iterated until no more transitions are applicable. Note that although the four cases above are exclusive, there may well be several distinct (q_0, φ)-runs on n given that the automaton itself is not necessarily deterministic. We say that a (q_0, φ)-run ρ of $\mathcal{A}$ on n is **accepting** and write $n \in \mathcal{L}(\mathcal{A}, q_0, \varphi)$ if $\mathsf{dom}(\rho) = \mathsf{dom}(n)$ and for all leaves a_0 of n there is an applicable transition of the form $(q, a_0) \Rightarrow ()$ or $((q, l), a_0) \Rightarrow ()$. If φ is empty we just write $n \in \mathcal{L}(\mathcal{A}, q_0)$.*

This is a procedural definition following the intuition of a computation of an actual physical machine. Given a tree, a start state and a label assignment as input, the machine first tags the root with the start state. Then it follows the structure downwards, always checking for a suitable rule to tag the children with successor states and where required consulting the label assignment. It accepts if it can tag all nodes, else it gets stuck halfway. The following illustrates a run of a trivial automaton with enumerated states q_i and labels l_i and some matching transitions on our example ν-tree from Section 3.5:



Initially, the root νa_2 is tagged with q_0 and we start with the empty partial label assignment $\emptyset$. Now we assume a type (III) transition $(q_0, \nu a_2) \Rightarrow (q_1, l_0)$. Hence in the next step the root is additionally assigned the label l_0 and the child a_2 is tagged with the next state q_1. Now we have a_2 bound by the root νa_2 which is expressed by the binding function $1 \rightsquigarrow \epsilon$. Hence only a type (II) transition depending on both q_1 and l_0 is applicable. So we assume a transition $((q_1, l_0), a_2) \Rightarrow (q_2, q_3)$ and tag the children with the next states q_2 and q_3, respectively. Next we assume a type (III) transition $(q_3, \nu b_0) \Rightarrow (q_4, l_1)$ and hence assign l_1 to νb_0 and q_4 to the child q_4. This defines a $(q_0, \emptyset)$-run on our example tree. It is accepting if we assume final transitions $(q_2, c_0) \Rightarrow ()$ and $((q_4, l_1), b_0) \Rightarrow ()$.

This example should provide some intuition for the mechanical interpretation of acceptance. However, having acceptance relativised to a start state and label assignment, we can give a much more elegant definition:

Definition 3.19. *We define the predicate $n \in \mathcal{L}'(\mathcal{A}, q, \varphi)$ inductively by:*

$$\frac{(q, a_k) \Rightarrow (q_1, \ldots, q_k) \quad n_i \in \mathcal{L}'(\mathcal{A}, q_i, \varphi) \quad a_k \notin \mathsf{dom}(\varphi)}{a_k n_1 \ldots n_k \in \mathcal{L}'(\mathcal{A}, q, \varphi)} \; (LFCst)$$

$$\frac{((q, l), a_k) \Rightarrow (q_1, \ldots, q_k) \quad n_i \in \mathcal{L}'(\mathcal{A}, q_i, \varphi) \quad \varphi(a_k) = l}{a_k n_1 \ldots n_k \in \mathcal{L}'(\mathcal{A}, q, \varphi)} \; (LBCst)$$

$$\frac{(q, \nu a_k) \Rightarrow (q', l) \quad n \in \mathcal{L}'(\mathcal{A}, q', \varphi[a_k := l])}{\nu a_k . n \in \mathcal{L}'(\mathcal{A}, q, \varphi)} \; (LNu)$$

Here, $\varphi[a_k := l]$ denotes the function that behaves like φ but maps a_k to l.

We let the reader verify that for the specified NTA $\mathcal{A}$ and ν-tree n from the example above we can derive $n \in \mathcal{L}'(\mathcal{A}, q_0, \emptyset)$. Indeed, the definitions turn out to be equivalent:

Lemma 3.12. $n \in \mathcal{L}(\mathcal{A}, q, \varphi) \iff n \in \mathcal{L}'(\mathcal{A}, q, \varphi)$

Proof. We first prove "$\Rightarrow$" by induction on $n \in \nu\text{-}\mathsf{Tree}$ for q, φ arbitrary:

- Suppose $a_k n_1 \ldots n_k \in \mathcal{L}(\mathcal{A}, q, \varphi)$. Then there exists an accepting (q, φ)-run ρ with $\rho(\epsilon) = q$. By being a run it holds either $(q_0, a_k) \Rightarrow (\rho(1), \ldots, \rho(k))$ and $a_k \notin \mathsf{dom}(\varphi)$ or $((q_0, l), a_k) \Rightarrow (\rho(1), \ldots, \rho(k))$ and $\varphi(a_k) = l$. In either case we can define functions ρ_i on $\mathsf{dom}(n_i)$ by $\rho_i(w) := \rho(i \cdot w)$. Then each ρ_i is an accepting $(\rho(i), \varphi)$-run on n_i so we have $n_i \in \mathcal{L}(\mathcal{A}, \rho(i), \varphi)$. By IH we obtain $n_i \in \mathcal{L}'(\mathcal{A}, \rho(i), \varphi)$ and hence $a_k n_1 \ldots n_k \in \mathcal{L}'(\mathcal{A}, q, \varphi)$ by either (LFCst) or (LBCst).
- Suppose $\nu a_k . n \in \mathcal{L}(\mathcal{A}, q, \varphi)$. Then there exists an accepting (q, φ)-run ρ on $\nu a_k . n$ with $\rho(\epsilon) = (q, l)$. By being a run this yields $(q, \nu a_k) \Rightarrow (\rho(1), l)$. We define a function ρ' on $\mathsf{dom}(n)$ by $\rho'(w) := \rho(1 \cdot w)$. It follows that ρ' is an accepting $(\rho(1), \varphi[a_k := l])$-run on n so we obtain $n \in \mathcal{L}(\mathcal{A}, \rho(1), \varphi[a_k := l])$. Then by IH we obtain $n \in \mathcal{L}'(\mathcal{A}, \rho(1), \varphi[a_k := l])$ and we conclude $\nu a_k . n \in \mathcal{L}'(\mathcal{A}, q, \varphi)$ by (LNu).

Now we show "$\Leftarrow$" by induction on $n \in \mathcal{L}'(\mathcal{A}, q, \varphi)$:

- If we have $a_k n_1 \ldots n_k \in \mathcal{L}'(\mathcal{A}, q, \varphi)$ by (LFCst) or (LBCst) we know that either $(q_0, a_k) \Rightarrow (q_1, \ldots, q_k)$ and $a_k \notin \mathsf{dom}(\varphi)$ or $((q_0, l), a_k) \Rightarrow (q_1, \ldots, q_k)$ and $\varphi(a_k) = l$. In either case IH yields $n_i \in \mathcal{L}(\mathcal{A}, q_i, \varphi)$ so there exist accepting (q_i, φ)-runs ρ_i on n_i. Then by setting $\rho(\epsilon) := q$ and $\rho(i \cdot w) := \rho_i(w)$ we obtain an accepting (q, φ)-run on $a_k n_1 \ldots n_k$ so we conclude $a_k n_1 \ldots n_k \in \mathcal{L}(\mathcal{A}, q, \varphi)$.

- If we have $\nu a_k.n \in \mathcal{L}'(\mathcal{A}, q, \varphi)$ by (LNu) we know that $(q, \nu a_k) \Rightarrow (q', l)$ and, already applying IH, that $n \in \mathcal{L}(\mathcal{A}, q', \varphi')$ where $\varphi' := \varphi[a_k := l]$. Hence there is an accepting (q', φ')-run ρ' on n. Then set $\rho(\epsilon) := q$ and $\rho(1 \cdot w) := \rho'(w)$. Now note that for every occurence of a_k at address w in the body of $\nu a_k.n$ we have $w \rightsquigarrow \epsilon$. It follows that ρ is an accepting (q, φ) run on $\nu a_k.n$ and thus $\nu a_k.n \in \mathcal{L}(\mathcal{A}, q, \varphi)$. □

We discuss some straight-forward examples of ν-tree automata. First, we define a trivial automaton $\mathcal{A}$ with a single state q and label l where Δ contains all possible transitions for a finite alphabet Σ. Then $\mathcal{A}$ clearly accepts all ν-trees with names from Σ. Secondly, if we drop all type (I) rules it accepts exactly the closed ν-trees whereas it accepts exactly the trees with all names unbound if we drop all type (II) rules instead. Finally, if we drop all (III) rules the resulting language will contain all pure trees with no nus.

More generally, our automaton model subsumes finite automata on words, nondeterministic ranked tree automata on pure ranked trees and, rephrased for arbitrary binders, nondeterministic dependency tree automata on alternating binding trees. The first two are clear since words and ranked trees are special forms of ranked ν-trees and the transitions of NFA and UTA are type (I) transitions of NTA. The latter holds since we effectively allow the same shapes of transitions but have a more general definition of the input trees.

We now study some algorithmic properties of ν-tree automata. First, we will describe procedures to obtain automata for union and intersection. Secondly, we will establish decidability of acceptance and emptiness. The first part consists of the following lemma:

Lemma 3.13. *Languages of ν-tree automata are closed under boolean operations, that is:*
(1) For all $\mathcal{A}_1, \mathcal{A}_2, q_1, q_2, \varphi_1, \varphi_2$ there exist $\mathcal{A}, q, \varphi$ with

$$\mathcal{L}(\mathcal{A}, q, \varphi) = \mathcal{L}(\mathcal{A}_1, q_1, \varphi_1) \cup \mathcal{L}(\mathcal{A}_2, q_2, \varphi_2)$$

(2) For all $\mathcal{A}_1, \mathcal{A}_2, q_1, q_2, \varphi_1, \varphi_2$ there exist $\mathcal{A}, q, \varphi$ with

$$\mathcal{L}(\mathcal{A}, q, \varphi) = \mathcal{L}(\mathcal{A}_1, q_1, \varphi_1) \cap \mathcal{L}(\mathcal{A}_2, q_2, \varphi_2)$$

Proof. We outline the constructions of the respective automata:

(1) This is a standard disjoint union construction. We first assume all components of $\mathcal{A}_1$ and $\mathcal{A}_2$ disjoint. Then we form $\mathcal{A}$ by the union of all respective components. This again defines a ν-tree automaton and it is easy to see that it accepts the union of both languages.

(2) This is a standard product construction. If we form the cartesian product of states and, respectively, labels we can define a set of transition rules simulating transitions in both automata at once. This defines a ν-tree automaton that accepts the intersection of the given languages. The only thing to consider is to make sure the automata are disjoint so the label assignments do not clash. □

To date we do not have a construction for complementation but the literature provides some confidence. First, it is a standard result that pure tree automata are closed under complementation [CDG+07]. Secondly, in [HORT16] it was shown that Stirling's alternating dependency tree automata [Sti09] have complements. This is a very similar model that only differs in the alternation. However, this difference might already be enough to find an NTA-language with non-acceptable complement.

Now considering our two standard decision problems, the restriction to finite Σ and Δ is crucial. Both algorithms basically try out all applicable transitions blindly. Hence we do not claim at all that there are no faster procedures possible. The proof for acceptance is simple as given a ν-tree n, a derivation must follow the finite structure of n:

Lemma 3.14. *Acceptance by ν-tree automata is decidable, that is, for any $\mathcal{A}$, q, φ and n there is an algorithm deciding $n \in \mathcal{L}(\mathcal{A}, q, \varphi)$.*

Proof. This is because in a derivation of $n \in \mathcal{L}(\mathcal{A}, q, \varphi)$ only finitely many choices of (LFCst), (LBCst) and (LNu) are applicable at any stage. Hence a decision procedure trying out all combinations until it either finds a correct derivation or runs out of options can be implemented. □

The decidability of emptiness is a little more involved since given some state q_0 and label assignment φ_0 there is principally no bound on the size of the ν-trees $n \in \mathcal{L}(\mathcal{A}, q_0, \varphi_0)$. However, we show that it is possible to define an algorithm that computes the smallest tree for every of the finitely many pairs (φ, q) that admit an accepted tree at all.

Lemma 3.15. *Emptiness of the language $\mathcal{L}(\mathcal{A}, q_0, \varphi_0)$ of a ν-tree automaton is decidable.*

Proof. Consider the following pseudo-code of a decision procedure:

```
trees($\varphi, q$):=$\bot$
changed:=True

while changed:
  changed := False

  for $\varphi \in \Sigma \rightharpoonup L$:

    for $(q, a_k) \Rightarrow (q_1, \dots q_k) \in \Delta$ with $q \notin \mathsf{dom}$(trees($\varphi$)):
      if $\{q_1, \dots q_k\} \subseteq \mathsf{dom}$(trees($\varphi$)) and $a_k \notin \mathsf{dom}(\varphi)$:
        trees($\varphi, q$):=$a_k$(trees($\varphi, q_1$))(trees($\varphi, q_k$))
        changed:=True

    for $((q, l), a_k) \Rightarrow (q_1, \dots q_k) \in \Delta$ with $q \notin \mathsf{dom}$(trees($\varphi$)):
      if $\{q_1, \dots q_k\} \subseteq \mathsf{dom}$(trees($\varphi$)) and $\varphi(a_k) = l$:
        trees($\varphi, q$):=$a_k$(trees($\varphi, q_1$))(trees($\varphi, q_k$))
        changed:=True
        for $(q, \nu a_k) \Rightarrow (q', l) \in \Delta$ with $q \notin \mathsf{dom}(\mathrm{trees}(\varphi))$:
          if $q' \in \mathsf{dom}(\mathrm{trees}(\varphi[a_k := l]))$:
            trees$(\varphi, q)$ := $\nu a_k$.(trees$(\varphi[a_k := l]), q'$)
            changed := True

    return trees$(\varphi_0, q_0)$ != $\bot$
```

In words, we first initialise a partial function trees: $(\Sigma \rightharpoonup L) \rightharpoonup Q \rightharpoonup \nu\text{-Tree}$ that is supposed to store the first encountered ν-tree accepted for every pair (φ, q). Then as long as new trees are generated, we run over all possible label assignments φ and test all transitions whether they can be used to construct a new tree from already existing ones. Once no new trees can be composed, the algorithm terminates and looks up trees to determine whether there is some $n \in \mathcal{L}(\mathcal{A}, q_0, \varphi_0)$.

If we forget about the label assignments for a moment, the idea is straight-forward. In the first iteration all trees consisting of a single leaf are constructed. Now say we have two of these leaves b_0 and c_0 accepted from q_1 and q_2, respectively. Then in the next iteration it is tested whether there is a transition of shape $(q, a_2) \Rightarrow (q_1, q_2)$. If so, the tree $a_2 b_0 c_0$ is stored as a tree accepted from q. Once a tree is obtained for a state q there is no need to find further ones since all trees accepted from q equally allow for constructing more complex trees for other states. Finally, considering label assignments again basically adds another loop and some jumps in the table trees whenever binding occurs.

This procedure always terminates since all loops are bounded, note that the finite domain of the partial function trees implicitly binds the while-loop. All trees constructed are accepted by the automaton via the conditionals and eventually all combinations are touched so no witnesses get missed. □

Thus ranked ν-tree automata as defined in this section yield well-defined decision procedures for properties of ν-trees and, using the assigned denotational semantics from Section 3.5 also for pure Σ-trees.

Chapter 4
Correspondence Proofs

In this chapter we bring together the two previous ones in presenting several instances of the type system-automaton correspondence, that is, for each of the discussed automaton models we show how to define intersection type systems which characterise the programs that compute accepted input. Since much of the correspondence is independent from the actual automaton model, we first describe the general procedure in Section 4.1. Then the remaining chapter can be seen as an evaluation of this schematic correspondence proof with respect to generality. In the following three sections, we show it strong enough to provide a unified treatment for all of finite word automata, unranked tree automata and nominal word automata. However, rather exotic automaton models such as our ranked ν-tree automata that rely on an extended term language constitute a limitation of the general scheme and hence we have to work out an independent proof. Before we study this final instance we fit in a brief comparison of ν-calculus, $\lambda\nu$-calculus and the smaller fragment we actually use in Section 4.5.

4.1 The General Scheme

We first discuss the properties of a slightly enriched lambda calculus with additional constants that will play the role of word or tree constructors. We show that all of subject reduction, subject expansion and weak normalisation are unaffected. Then we outline how to establish a correspondence between a particular automaton and a type system. We first define the extended term language:

Definition 4.1. *For a set Σ, the set Λ_Σ of terms with constants is generated by:*

$$s, t ::= x \mid \lambda x.s \mid st \mid a$$

The variables x still range over the countable set Var *and the constants a range over Σ.*

Then we can extend ITLC to incorporate arbitrary typings for constants:

Definition 4.2. *Let Σ be a set. A system Σ-ITLC of judgements $\Gamma \vdash_\Sigma s : \tau$ for terms $s \in \Lambda_\Sigma$ and types $\tau \in \mathsf{IType}$ given a binary relation $P \subseteq \Sigma \times \mathsf{IType}$ consists of ITLC together with the following constant rules induced by P:*

$$\frac{P(a, \tau)}{\Gamma \vdash_\Sigma a : \tau}\ (Cst)$$

Here $a \in \Sigma$ and we may leave out the subscript in $\vdash_\Sigma$ where possible and reasonable.

We prove every Σ-ITLC well-behaved by giving a translation of typings in Σ-ITLC to pure ITLC. To this end consider the following definition:

Definition 4.3. *Let $\Gamma \vdash_\Sigma s : \tau$ be a judgement in a Σ-ITLC. We denote s with all constants a replaced by corresponding variables x_a by s'. Moreover, for any $a \in s$ we denote by $\tau_1^a, \dots \tau_{k_a}^a$ all types that are derived from instances of (Cst) during deriving $\Gamma \vdash_\Sigma s : \tau$. Then we set $\Gamma' := \Gamma \cup \{\, x_a : \bigwedge \tau_i^a \mid a \in s \,\}$.*

This translation is easily shown correct in the following sense:

Fact 4.1. *If $\Gamma \vdash_\Sigma s : \tau$ then $\Gamma' \vdash s' : \tau$.*

Proof. This is by an induction on the derivation of $\Gamma \vdash_\Sigma s : \tau$, every occurrence of (Cst) can be replaced by an application of (Var) in the extended context Γ'. □

Now we can show the lemma that establishes the wished properties for any Σ:

Lemma 4.2. *Any Σ-ITLC satisfies subject reduction, expansion and weak normalisation:*
(1) if $\Gamma \vdash_\Sigma s : \tau$ and $s \to t$ then $\Gamma \vdash_\Sigma t : \tau$,
(2) if $\Gamma \vdash_\Sigma t : \tau$ and $s \to t$ then $\Gamma \vdash_\Sigma s : \tau$ and
(3) if $\Gamma \vdash_\Sigma s : \tau$ then $s \in \mathsf{WN}$.

Proof. We first show subject reduction, so suppose $\Gamma \vdash_\Sigma s : \tau$ and $s \to t$. By Fact 4.1 we know that $\Gamma' \vdash s' : \tau$ and it is trivial that we have a reduction $s' \to t'$ where t' denotes t after the same replacement of constants with variables. Now we can apply subject reduction of pure ITLC and hence obtain $\Gamma' \vdash t' : \tau$. Then by reversing the translation again we conclude $\Gamma \vdash_\Sigma t : \tau$.
The proof for subject expansion is similar. For weak normalisation just suppose $\Gamma \vdash_\Sigma s : \tau$. Then $\Gamma' \vdash s' : \tau$ and hence $s \in \mathsf{WN}$ by weak normalisation of pure ITLC (Theorem 2.9). □

We can now outline the structure of any particular correspondence proof. First, fix an automaton $\mathcal{A}$ of a certain model. This provides sets Σ and Q and a collection of transitions. The alphabet Σ induces the set Λ_Σ from Definition 4.1 and we employ Q as base types for IType, denoted IType_Q following Definition 2.6. Then we say that $s \in \Lambda_\Sigma$ has a **strong normal form** n if $s \to^* n$ and n is a well-formed input word or tree for $\mathcal{A}$. In this case we reuse the notation $s \Downarrow n$ and note that, since they contain no lambdas, strong normal forms are in particular β-normal. The main objective is to define a Σ-ITLC where the instances of (Cst) capture the transition relation of $\mathcal{A}$. Then for this system, denoted $\mathcal{A}$-ITLC, we prove the correspondence of type system and automaton for strong normal forms together with the property that base-typable normal forms must already be strong:

Lemma. *The following two hold for $\mathcal{A}$-ITLC:*
(1) For a strong normal form n we have $\vdash n : q$ iff $n \in \mathcal{L}(\mathcal{A}, q)$.
(2) If n is β-normal and $\vdash n : q$ then n is strongly normal.

Sketch. *All claims follow from induction on the derivation of the respective assumption.*

The exact proof of this correspondence lemma of course depends on the actual automaton model. By contrast, the generalisation to any term of Λ_Σ follows immediately with the properties of the system Σ-ITLC:

Theorem. *For any $s \in \Lambda_\Sigma$ we have $\vdash_{\mathcal{A}} s : q$ iff there is n with $s \Downarrow n$ and $n \in \mathcal{L}(\mathcal{A}, q)$.*

Proof. *Suppose $\vdash s : q$, then by normalisation there is a β-normal form n with $s \rightarrow^* n$. Then by subject reduction $\vdash n : q$ and the two parts of the correspondence lemma imply $s \Downarrow n$ and ultimately $n \in \mathcal{L}(\mathcal{A}, q)$.*
Conversely, if we assume $s \Downarrow n$ and $n \in \mathcal{L}(\mathcal{A}, q)$ we have $\vdash n : q$ by (1) of the correspondence lemma and conclude $\vdash s : q$ with subject expansion. □

In the next three sections we study direct instances of this procedure for finite word and tree automata as well as Bojanczyk's nominal automata. The proof for ranked ν-tree automata will be slightly more complex but it still follows a similar structure.

4.2 Type Systems for Finite Word Automata

This first instance is nearly trivial and we just mention it to illustrate a correspondence proof in the absence of any technical overhead. If we consider a particular NFA $\mathcal{A}$, we obtain sets Σ and Q that form the terms in Λ_Σ and types in IType_Q. Then the strong normal forms are exactly the unary trees of shape $a(b(\dots(c)\dots))$ for $a, b, \dots, c \in \Sigma$. We define the Σ-ITLC for $\mathcal{A}$, abbreviated $\mathcal{A}$-ITLC, by:

Definition 4.4. *Let $\mathcal{A}$ be an NFA. The system $\mathcal{A}$-ITLC of judgements $\Gamma \vdash_{\mathcal{A}} s : \tau$ for terms $s \in \Lambda_\Sigma$ and types $\tau \in \mathsf{IType}_Q$ consists of the rules of ITLC together with:*

$$\frac{q \xrightarrow{a} q' \quad q' \in F}{\Gamma \vdash_{\mathcal{A}} a : q}\ (F) \qquad\qquad \frac{q \xrightarrow{a} q'}{\Gamma \vdash_{\mathcal{A}} a : q' \to q}\ (Cst)$$

As usual, we may leave out the subscript in $\vdash_{\mathcal{A}}$ where possible and reasonable.

The intuition is that a word is assigned a type q iff it is accepted by $\mathcal{A}$ starting in state q. Hence, by rule (F) we can assign a type q to a single symbol a if there is a a-transition from q to a final state. Moreover, if there is any transition $q \xrightarrow{a} q'$ we can type a with $q' \to q$ since by prepending a we can transform a word being accepted from q' to a word being accepted from q.

By Lemma 4.2 we know that the Σ-ITLC for any NFA $\mathcal{A}$ is well-behaved, meaning that it satisfies subject reduction, subject expansion and weak normalisation. Hence, in order to establish the general correspondence of type system and automaton it suffices to show the correspondence lemma:

Lemma 4.3. *Let $\mathcal{A}$ be an NFA. Then the following two hold for $\mathcal{A}$-ITLC:*
(1) For a strong normal form n we have $\vdash n : q$ iff $n \in \mathcal{L}(\mathcal{A}, q)$.
(2) If n is β-normal and $\vdash n : q$ then n is strongly normal, that is a Σ-word.

Proof. We establish (1) by natural induction on the length of the word n:

- Since the empty word ϵ is no actual term of Λ_Σ, the base case is length 1. Hence we want to show that $\vdash a : q$ iff $a \in \mathcal{L}(\mathcal{A}, q)$. The only rule justifying $\vdash a : q$ is (F), so we know $q \xrightarrow{a} q'$ for some $q' \in F$. This already shows $a \in \mathcal{L}(\mathcal{A}, q)$. Conversely, assuming $a \in \mathcal{L}(\mathcal{A}, q)$ yields the necessary facts to witness $\vdash a : q$.
- In the inductive case we want to show $\vdash an : q$ iff $an \in \mathcal{L}(\mathcal{A}, q)$ and may assume by IH that $\vdash n : q'$ iff $n \in \mathcal{L}(\mathcal{A}, q')$ for any $q' \in Q$. So suppose $\vdash an : q$. This is a function application and hence inverts to $\vdash a : q' \to q$ and $\vdash n : q'$ for some $q' \in Q$. Then by IH we have $n \in \mathcal{L}(\mathcal{A}, q')$ so there is $q_F \in F$ with $q' \xrightarrow{n} q_F$. But since $\vdash a : q' \to q$ implies $q \xrightarrow{a} q'$ by inverting (Cst) we also have $q \xrightarrow{an} q_F$ and hence $an \in \mathcal{L}(\mathcal{A}, q)$. Conversely, if we assume $an \in \mathcal{L}(\mathcal{A}, q)$ we obtain $q' \in Q$ and $q_F \in F$ with $q \xrightarrow{a} q' \xrightarrow{n} q_F$ which implies $\vdash an : q$ employing the IH for $n \in \mathcal{L}(\mathcal{A}, q')$.

Now we show (2) by induction on the derivation of $\vdash n : q$:

- A derivation by (Var) is impossible since we consider the empty context.
- Likewise, a derivation by (Abs) is impossible since we consider a base type.
- So consider a derivation by (App), say we have $\vdash n_1 n_2 : q$ for both n_1, n_2 β-normal. By inversion we obtain $\vdash n_1 : \bigwedge \tau_i \to q$ and $\vdash n_2 : \tau_i$ for some $\tau_1, \ldots, \tau_k$. Since we assume $n_1 n_2$ to be β-normal, the judgement $\vdash n_1 : \bigwedge \tau_i \to q$ cannot rely (Abs). The only other rule deriving a function type is (Cst) so we have $n_1 = a$ and $\bigwedge \tau_i = q'$ for some $a \in \Sigma$ and $q' \in Q$. Then by the IH for $\vdash n_2 : q'$ we know that n_2 is a proper word and hence so is $n_1 n_2 = a n_2$.
- The rules (F) and (Cst) derive typings for trivial words. □

Having established the correspondence for normal forms, the generalisation to arbitrary terms is completely generic:

Corollary 4.4. *For any NFA $\mathcal{A}$ we have $\vdash_{\mathcal{A}} s : q$ iff there is $s \Downarrow n$ with $n \in \mathcal{L}(\mathcal{A}, q)$.*

Proof. This is exactly as outlined in Section 4.1. □

We end this section with a remark concerning decision problems of the system $\mathcal{A}$-ITLC. As we have observed in Section 2.3, general typability, type checking and type inhabitance are not decidable for ITLC and, given the mutual reduction outlined in the proof of Lemma 4.2, so they are for any Σ-ITLC. However, if we restrict to base types and strong normal forms typability and type checking become decidable. Moreover, we can decide type inhabitance for base types with arbitrary terms:

Lemma 4.5. *For any NFA $\mathcal{A}$, the system $\mathcal{A}$-ITLC has decidable normal typability, normal type checking and base type inhabitance, that is, there are algorithms that decide (1) for a given strong normal form $n \in \Lambda_\Sigma$ whether there is a state $q \in Q$ with $\vdash n : q$,*
(2) for given n, q whether $\vdash n : q$ and
(3) for given q whether there is s with $\vdash s : q$.

Proof. First note that, given only finitely many base types $q \in Q$ there is a straight-forward reduction of normal type checking to normal typability by just testing $\vdash n : q$ for all $q \in Q$. Now, normal type checking is decidable since by Lemma 4.3 it is equivalent to the decidable acceptance problem of the NFA $\mathcal{A}$. Finally, recall that for given q there is an algorithm deciding whether there is n with $n \in \mathcal{L}(\mathcal{A}, q)$. If so, then by the correspondence lemma $\vdash n : q$. If not, there is also no $s \in \Lambda_\Sigma$ with $\vdash s : q$ since, by Corollary 4.4 it would reduce to a strong normal form n witnessing $n \in \mathcal{L}(\mathcal{A}, q)$. □

4.3 Type Systems for Finite Tree Automata

We next study the correspondence for unranked tree automata. Note that all other common automaton models on finite trees, in particular on ranked trees, then can be considered special cases of this correspondence. The development follows exactly the structure from the previous chapter, only the proof of the correspondence lemma differs.

First consider some UTA $\mathcal{A}$. As above this gives rise to the set Λ_Σ of terms with constants from the alphabet Σ and IType with base types from the state space Q. The strong normal forms are the Σ-trees, note that these are exactly the lambda-free terms from Λ_Σ. Based on these components we can define a type system as follows:

Definition 4.5. *Let $\mathcal{A}$ be an UTA. The system $\mathcal{A}$-ITLC of judgements $\Gamma \vdash_{\mathcal{A}} s : \tau$ for terms $s \in \Lambda_\Sigma$ and types $\tau \in \mathsf{IType}_Q$ consists of the rules of ITLC together with:*

$$\frac{(L, a, q) \in \delta \quad q_1 \dots q_k \in L}{\Gamma \vdash_{\mathcal{A}} a : q_1 \to \dots \to q_k \to q} \ (Cst)$$

As usual, we may leave out the subscript in $\vdash_{\mathcal{A}}$ where possible and reasonable.

Considering a symbol $a \in \Sigma$ as a tree constructor we can provide intuition for the rule above. Suppose there are trees $n_1, \dots, n_k$ that are accepted from states

$q_1, \ldots, q_k$, respectively. Hence they can be assigned the respective states as types. Then, provided there is a matching transition, the tree $an_1 \ldots n_k$ is accepted from state q so it makes sense to assign a the type $q_1 \to \cdots \to q_k \to q$.

Again, the systems $\mathcal{A}$-ITLC are well-behaved by Lemma 4.2. Hence, as for NFA before, we only need to establish the correspondence lemma for UTA:

Lemma 4.6. *Let $\mathcal{A}$ be an UTA. Then the following two hold for $\mathcal{A}$-ITLC:*
(1) For a strong normal form n we have $\vdash n : q$ iff $n \in \mathcal{L}(\mathcal{A}, q)$.
(2) If n is β-normal and $\vdash n : q$ then n is strongly normal, that is a Σ-tree.

Proof. We show (1) by structural induction on $n \in \Sigma\text{-Tree}$:

- Suppose $\vdash an_1 \ldots n_k : q$. This is a k-fold function application and since (Cst) is the only rule that assigns a type to a the original judgement inverts to $\vdash a : q_1 \to \cdots \to q_k \to q$ and $\vdash n_i : q_i$ for some $q_1, \ldots q_k \in Q$. Further, the typing of a witnesses a transition $(L, a, q) \in \delta$ with $q_1 \ldots q_k \in L$. Moreover, the IH for $\vdash n_i : q_i$ yields $n_i \in \mathcal{L}(\mathcal{A}, q_i)$ hence we can altogether apply the rule (LCst) to establish $n \in \mathcal{L}(\mathcal{A}, q)$.
- Conversely, if we start with $n \in \mathcal{L}(\mathcal{A}, q)$ we can invert (LCst) and obtain a transition $(L, a, q) \in \delta$ with $q_1 \ldots q_k \in L$ together with the guarantee that $n_i \in \mathcal{L}(\mathcal{A}, q_i)$. The latter is subject to the IH, yielding $\vdash n_i : q_i$ and thus $\vdash an_1 \ldots n_k : q$ by applying (Cst) and (App).

Part (2) is again by induction on the derivation of $\vdash n : q$. As for NFA, the cases (Var), (Abs) and (Cst) are either impossible or trivial. Hence, suppose we are in the case (App), so assume $\vdash nn' : q$. As before, since nn' is β-normal n cannot be a lambda. Hence we actually must have $nn' = (an_1 \ldots n_{k-1})n_k$ for $a \in \Sigma$ and β-normal $n_1, \ldots n_k \in \Lambda_\Sigma$. Then the typing $\vdash an_1 \ldots n_k : q$ inverts to $\vdash a : q_1 \to \cdots \to q_k \to q$ and $\vdash n_i : q_i$ for some $q_1, \ldots q_k \in Q$. By IH all n_i are strongly normal and hence we can conclude that so is $an_1 \ldots n_k = nn'$. □

Note that the proof of the first claim becomes very concise using the inductive reformulation of $\mathcal{L}(\mathcal{A}, q)$, all technical arguing about accepting runs is hidden. The effect will be even more helpful in the proof for ranked ν-tree automata in Section 4.6. Again, the general correspondence follows immediately:

Corollary 4.7. *For any UTA $\mathcal{A}$ we have $\vdash_\mathcal{A} s : q$ iff there is $s \Downarrow n$ with $n \in \mathcal{L}(\mathcal{A}, q)$.*

Proof. This is again exactly as in Section 4.1. □

Finally, since we know that acceptance and emptiness are decidable for UTA, we have the same decidability results for the UTA systems as for the NFA systems:

Lemma 4.8. *For any UTA $\mathcal{A}$, the system $\mathcal{A}$-ITLC has decidable normal typability, normal type checking and base type inhabitance (cf. Lemma 4.5).*

Proof. This is by the same justification as Lemma 4.5. □

4.4 Type Systems for Nominal Word Automata

Since nominal automata as introduced by Bojanczyk do not differ from classical finite word automata from an operational perspective, the correspondence proof for NNA is exactly the same as for NFA. We just give the key definitions and statements for completeness, for the proofs we refer to Section 4.2. Note that in this section, strong normal forms are again words over an alphabet Σ.

Definition 4.6. *Let $\mathcal{A}$ be an NNA. The system $\mathcal{A}$-ITLC of judgements $\Gamma \vdash_{\mathcal{A}} s : \tau$ for terms $s \in \Lambda_\Sigma$ and types $\tau \in \mathsf{IType}_Q$ consists of the rules of ITLC together with:*

$$\frac{q \xrightarrow{a} q' \quad q' \in F}{\Gamma \vdash a : q}\,(F) \qquad\qquad \frac{q \xrightarrow{a} q'}{\Gamma \vdash a : q' \to q}\,(Cst)$$

As usual, we may leave out the subscript in $\vdash_{\mathcal{A}}$ where possible and reasonable.

Although this system looks the same as the one for NFA it differs substantially in size. While the finite components of NFA only induce finitely many rules of the form (F) and (Cst) there may be infinitely many such instances when considering an NNA. That the NNA system still comes with the same decidability results (Lemma 4.11) can be attributed to the fact that the typing relation is fully equivariant, that is, if we have $\Gamma \vdash_{\mathcal{A}} s : \tau$ then also $\pi \cdot \Gamma \vdash_{\mathcal{A}} \pi \cdot s : \pi \cdot \tau$ for all $\pi \in \mathsf{Perm}(\mathbb{A})$ for the self-explanatory perm actions on contexts, terms and types.

Once again we refer to Lemma 4.2 to justify that the NNA systems are well-behaved. Then we proceed as in the two instances before:

Lemma 4.9. *Let $\mathcal{A}$ be an NNA. Then the following two hold for $\mathcal{A}$-ITLC:*
(1) For a strong normal form n we have $\vdash n : q$ iff $n \in \mathcal{L}(\mathcal{A}, q)$.
(2) If n is β-normal and $\vdash n : q$ then n is strongly normal, that is a Σ-word.

Proof. This is the same as for NFA, so we just refer to Lemma 4.3. $\square$

The correspondence for arbitrary terms is established in the usual manner:

Corollary 4.10. *For any NNA $\mathcal{A}$ we have $\vdash_{\mathcal{A}} s : q$ iff there is $s \Downarrow n$ with $n \in \mathcal{L}(\mathcal{A}, q)$.*

Proof. This is once again exactly as in Section 4.1. $\square$

Finally, since we have observed decidability of emptiness and acceptance of NNA in Section 3.4, we can conclude the same results for $\mathcal{A}$-ITLC as for the finite automaton models from the previous sections:

Lemma 4.11. *For any NNA $\mathcal{A}$, the system $\mathcal{A}$-ITLC has decidable normal typability, normal type checking and base type inhabitance (cf. Lemma 4.5).*

Proof. This is by the same justification as Lemma 4.5. The reduction from type checking to typability still works given the decidability of orbit equivalence together with the orbit-finiteness of Q. $\square$

4.5 On ν-Calculus and $\lambda\nu$-Calculus

Before we move on to our final instance of a correspondence proof, we give a brief summary of Pitts' and Stark's ν-calculus [PS93] and the similar $\lambda\nu$-calculus [Ode94] by Odersky in order to introduce the class of programs that compute ν-trees as input for ν-tree automata. This class will be the set ν-Term from Definition 4.7 and the corresponding notion of reduction is defined in Definition 4.8. Both calculi address the theoretical treatment of bound names with local scopes in (functional) programming languages. After a comparison of the two calculi we clarify in which sense our work on ranked ν-trees is related.

Both ν-calculus and $\lambda\nu$-calculus are based on a simply typed lambda calculus with nus acting as binders for local names. Concretely, the term language is defined as follows:

Definition 4.7. *The set ν-Term is generated by the following grammar:*

$$s, t ::= x \mid \lambda x.s \mid st \mid a \mid \nu a.s$$

We have the variables x ranging over Var and the names a ranging over $\mathbb{A}$.

In particular, ν-trees are β-normal ν-terms, so it makes sense to consider ν-Term as programs computing ν-trees. Also note that ν-Term is actually just a fragment of the terms used in the original calculi since both also contain boolean constants, equality tests for names and pairs. We focus on this fragment since it will serve as the base for our type system in the next section but we may use some of the missing language features informally to point out differences between the two original calculi.

The typing rules for ν-calculus and $\lambda\nu$-calculus are the rules of STLC plus straightforward rules for the additional language components. In particular, for the introduced names and nus they are:

$$\frac{}{\Gamma \vdash a : \mathsf{Nme}} \qquad\qquad \frac{\Gamma \vdash s : \tau}{\Gamma \vdash \nu a.s : \tau}$$

Here Nme is a type constant in Type and we only consider simple types $\tau \in$ SType.

So the syntax of ν-calculus and $\lambda\nu$-calculus is the same and we have to move on to the respective semantics to unveil the actually quite different behaviour. On the one hand, the keywords for understanding ν-calculus are "dynamically allocated names" and "state-based evaluation". Dynamically allocated means that, when processing a ν-term as a program, every time a nu is encountered a fresh name is generated. In fact, Pitts and Stark concretely mention the type unitref of references to the unit object in ML as the domain of names. Then the evaluation becomes state-based given that during computation a state of currently active references is manipulated.

Formally, this boils down to defining an operational semantics of the form $(S, s) \Downarrow_\nu (S', v)$ where S is a collection of local names before execution of s and S' is the final state after evaluating s to the value v. We write $s \Downarrow_\nu (S', v)$ whenever the initial state S is empty. For instance, $\nu a.a \Downarrow_\nu (\{a\}, a)$ is such an evaluation where the final state $\{a\}$ simply displays that when processing $\nu a.a$ a new name a was allocated. Another example that underlines the dynamic behaviour is that $(\lambda x.x = x)(\nu a.a) \Downarrow (\{a\}, \mathsf{true})$ whereas $(\nu a.a) = (\nu a.a) \Downarrow (\{a, b\}, \mathsf{false})$. The former equality test evaluates to true since first the nu creates a new name a which is then tested equal with itself. The latter evaluates to false since the two nus create two distinct new names a and b. Thus ν-calculus does not satisfy normal β-reduction.

Odersky's $\lambda\nu$-calculus, on the other hand, is a direct extension of β-reduction with the "scope-intruding" rule $\nu a.\lambda x.s \to \lambda x.\nu a.s$ (and similar for pairs) together with $\nu a.b \to b$ for $b \neq a$. This yields a very different behaviour of the local names, for instance the term $\nu a.a$ fails to be reducible and in contrast to the example above we have $(\lambda x.x = x)(\nu a.a) \to (\nu a.a) = (\nu a.a)$ where the latter admits no further reductions. However, it was shown in [LP11] that dynamically allocated names can still be simulated within this system which admits a translation of terms from ν-calculus to $\lambda\nu$-calculus.

We end this rather conceptual discussion with a few remarks on our work on ranked ν-trees. Our motivation is that we consider ν-terms as a set of programs that compute ν-trees which in turn denote sets of $\mathbb{A}$-trees. This purpose has consequences for the applicable type system that actually marks our work very different from the two original calculi. First, if we were to include the first typing rule for names, most of the ν-trees would remain untypable since for trees like $a_2 b_0 b_0$ we need to assign a function type to a_2. Secondly, the rule for nus does not fit to automata that might change their state when processing a tree like $\nu a.a$. Here we would like to assign the subtree a a type q if it is accepted from state q but do not want to require the whole $\nu a.a$ to be accepted from the same state q. So even before mentioning intersection types, our type system will necessarily differ substantially from the simple types assigned in ν-calculus and $\lambda\nu$-calculus.

As we have seen in the previous sections, our type systems come with a well-behaved notion of reduction and it is natural to require the same for the type systems capturing NTA. Hence we employ an Odersky-style reduction that extends the normal β-reduction by a rule for nus. Formally, we define:

Definition 4.8. *We introduce the **ν-reduction relation** $s \to_\nu t$ by the rules*

$$\frac{}{(\lambda x.s)t \to_\nu s[t/x]} \qquad\qquad \frac{}{\nu a.\lambda x.s \to_\nu \lambda x.\nu a.s}$$

*and the usual contextual closure for one-sided reduction and reduction under lambda. Again, reflexive-transitive closure is usually denoted $s \to_\nu^* t$ and the smallest equivalence relation closed under ν-reduction is usually called **ν-conversion**.*
We will mostly leave out the subscript ν.

Since we can simply push nus through lambdas it is clear that ν-reduction still satisfies the Church-Rosser property stating that arbitrary reduction paths from the same root can be unified again. Thus the extended relation is equally well-behaved admitting in particular unique normal forms.

Note that we left out the second rule in $\lambda\nu$-calculus that stated $\nu a.b \to b$ for $a \neq b$. The reason is that, in order to match our use of names as tree constructors, it would require generalisation to arbitrary ν-trees along the lines $\nu a.n \to n$ if $a \notin \mathsf{FN}(n)$. However, this would be an unnecessary complication since we consider programs that compute ν-trees and hence do not actually want to have those trees reduce any further. In fact, employing such a rule would cause the familiar incompatibility with subject conversion since $\nu a.b$ and b are assigned different types if the corresponding automaton changes its state during the run.

Now, having formalised a set of terms and a notion of reduction, we can finally define type systems that capture the transitions of an NTA. As in the instances before, this will follow the compositional style of automaton acceptance, providing one rule for every shape of transitions. To obtain a type system that is closed under conversion as usual, this will require a push rule of the form

$$\frac{\Gamma \vdash_{\mathcal{A}} \lambda x.\nu a.s : \sigma \to \tau}{\Gamma \vdash_{\mathcal{A}} \nu a.\lambda x.s : \sigma \to \tau}$$

that allows for subject expansion. To see this, note that the base ITLC system enriched with rules for the constructors of ν-trees does not at all provide typings for terms of the form $\nu a.\lambda x.s$. However, since we might face a reduction sequence like $(\nu a.\lambda x.x)b \to (\lambda x.\nu a.x)b \to \nu a.b$ where the ν-tree $\nu a.b$ may well be accepted from a state q we must require the type system to assign type q to the ν-term $(\nu a.\lambda x.x)b$ as well. Hence the push rule. The formal definition of a type system for an NTA and the correspondence proof are subject of the next section.

We end this section with a remark on the denotational semantics of ν-trees from Section 3.5. As outlined in [Pit13], there are ways to define denotational semantics for both ν-calculus and $\lambda\nu$-calculus (cf. Sections 9.4 and 9.6). However, they differ fundamentally from our denotation of subsets of $\mathbb{A}$-trees in that they map types to nominal sets and typed terms to elements of those sets. This is a semantics that focuses on the meaning of programs whereas we are interested in the meaning of the resulting trees themselves.

4.6 Type Systems for Ranked ν-Tree Automata

In this section we present the final type system-automaton correspondence for NTA. So far we have seen that finite word and tree automata as well as the already rather special nominal word automata have correspondences that are direct instances of the general scheme from Section 4.1. However, given that they are based on a richer set of terms, NTA do not fit to this unified treatment.

Still, the structure of this section will be very similar to the previous instances but there will be some subtleties that require careful discussion. In order to incorporate the nus in ν-trees we base our type system on the terms in ν-Term from Definition 4.7 and rely on ν-reduction from Definition 4.8. In fact, since all NTA have their alphabet embedded into $\mathbb{A}$, the corresponding systems share their set of terms and only differ in the set of types and, obviously, instances of constant rules. The strong normal forms are the well-formed input terms of NTA, the ranked ν-trees. Then, for a given NTA we define the corresponding type system as follows:

Definition 4.9. *Let $\mathcal{A}$ be an NTA. The system $\mathcal{A}$-ITLC of judgements $\Gamma, \varphi \vdash_{\mathcal{A}} s : \tau$ for terms $s \in \nu$-Term and types $\tau \in \mathsf{IType}_Q$ consists of the rules of ITLC together with:*

$$\frac{\Gamma, \varphi \vdash_{\mathcal{A}} \lambda x.\nu a_k.s : \sigma \to \tau}{\Gamma, \varphi \vdash_{\mathcal{A}} \nu a_k.\lambda x.s : \sigma \to \tau} \ (Push)$$

$$\frac{(q, a_k) \Rightarrow (q_1, \ldots, q_k) \quad a_k \notin \mathsf{dom}(\varphi)}{\Gamma, \varphi \vdash_{\mathcal{A}} a_k : q_1 \to \cdots \to q_k \to q} \ (FCst)$$

$$\frac{((q, l), a_k) \Rightarrow (q_1, \ldots, q_k) \quad \varphi(a_k) = l}{\Gamma, \varphi \vdash_{\mathcal{A}} a_k : q_1 \to \cdots \to q_k \to q} \ (BCst)$$

$$\frac{(q, \nu a_k) \Rightarrow (q', l) \quad \Gamma, \varphi[a_k := l] \vdash s : q'}{\Gamma, \varphi \vdash_{\mathcal{A}} \nu a_k.s : q} \ (Nu)$$

In all other rules of ITLC, the label assignment φ is just passed on without effect. As usual, we may leave out the subscript in $\vdash_{\mathcal{A}}$ where possible and reasonable.

Note that, similar as in the inductive definition of acceptance from Section 3.6, we need to consider typing judgements relative to label assignments φ to allow for full compositionality. Then the rules are justified as follows. The (Push) rule is actually independent from the automaton $\mathcal{A}$ and makes the system satisfy subject expansion (cf. Section 4.5 for a discussion of this rule). The rules (FCst), (BCst) and (Nu) capture the transitions of type (I), (II) and (III), respectively. The former two work similar as the (CSt) rule for UTA, providing the correct function type for a tree constructor a_k. The latter captures the transition of an automaton from a binder to its body. Note that, as outlined in the previous session, these rules mark the type systems $\mathcal{A}$-ITLC remarkably different from the simple type system underlying ν-calculus and $\lambda\nu$-calculus.

Also, since the systems $\mathcal{A}$-ITLC for NTA are no pure instances of Σ-ITLC as defined in Section 4.1, we cannot apply Lemma 4.2 directly in order to establish the usual properties making the system well-behaved. However, given that the additional (Push) rule was introduced exactly for the purpose of achieving subject expansion as one of those properties, it remains easy to justify them for the extended system:

Lemma 4.12. *Let $\mathcal{A}$ be an NTA. Then $\mathcal{A}$-ITLC satisfies all of subject reduction, subject expansion and weak normalisation.*

Proof. We discuss the interesting cases for the three inductive proofs:

- For subject reduction, suppose there are $s, t \in \nu\text{-Term}$ with $s \to t$ and $\Gamma, \varphi \vdash s : \tau$. Then we show $\Gamma, \varphi \vdash t : \tau$ by induction on the derivation of $s \to t$. The only non-standard case is where $\nu a.\lambda x.s \to \lambda x.\nu a.s$ where the assumed judgement has the form $\Gamma, \varphi \vdash \nu a.\lambda x.s : \tau$. Since (Push) is the only rule that provides a type for such a term, we can invert the derivation and obtain the wished $\Gamma, \varphi \vdash \lambda x.\nu a.s : \tau$.
- Similarly, for subject expansion we assume $s \to t$ and $\Gamma, \varphi \vdash t : \tau$. We show the claim $\Gamma, \varphi \vdash s : \tau$ again by induction on $s \to t$. All cases are standard except for $\nu a.\lambda x.s \to \lambda x.\nu a.s$ with the assumption $\Gamma, \varphi \vdash \lambda x.\nu a.s : \tau$. Here we conclude the claim $\Gamma, \varphi \vdash \nu a.\lambda x.s : \tau$ immediately by citing (Push).
- For weak normalisation, suppose we have a judgement $\Gamma, \varphi \vdash s : \tau$. Then we want to find a ν-normal form for s, that is $n \in \nu\text{-Term}$ with $s \to_\nu^* n$ that admits no further reductions. Note that it suffices to find β-normal forms since every β-normal form reduces to a ν-normal form by pushing all remaining nus through the remaining lambdas. Now, in the cases where the typing $\Gamma, \varphi \vdash s : \tau$ was not derived by (Push) we can consider it a pure Σ-ITLC typing and hence, by Lemma 4.2 find a β-normal form $n \in \nu\text{-Term}$ for s. If the derivation was $\Gamma, \varphi \vdash \nu a_k.\lambda x.s$ by $\Gamma, \varphi \vdash \lambda x.\nu a.s : \tau$ and (Push) the IH yields a ν-normal form for $\lambda x.\nu a.s$. Then this is also a ν-normal form for $\nu a_k.\lambda x.s$ given that $\nu a_k.\lambda x.s \to_\nu \lambda x.\nu a.s$. $\square$

We proceed in the usual manner, so we first show the correspondence lemma:

Lemma 4.13. *Let $\mathcal{A}$ be an NTA. Then the following two hold for $\mathcal{A}$-ITLC:*
(1) For a strong normal form n we have $\varphi \vdash n : q$ iff $n \in \mathcal{L}(\mathcal{A}, q, \varphi)$.
(2) If n is ν-normal and $\varphi \vdash n : q$ then n is strongly normal, that is a ν-tree.

Proof. The first claim is by structural induction on $n \in \nu\text{-Tree}$:

- Consider the ν-tree $a_k n_1 \dots n_k$ and suppose $a_k n_1 \dots n_k \in \mathcal{L}(\mathcal{A}, q, \varphi)$. Depending on whether or not $a_k \in \mathsf{dom}(\varphi)$ this was derived by (LFCst) of (LBCst). In either case we obtain $n_i \in (\mathcal{A}, q_i, \varphi)$ together with a witnessing transition $(q, a_k) \Rightarrow (q_1, \dots, q_k)$ if $a_k \notin \mathsf{dom}(\varphi)$ and $((q, l), a_k) \Rightarrow (q_1, \dots, q_k)$ if $\varphi(a_k) = l$. Then by IH we have $\varphi \vdash n_i : q_i$ which is enough information to conclude $\varphi \vdash a_k n_1 \dots n_k : q$ with (FCst) or respectively (BCst) together with k applications of (App). The converse direction is similar.
- Now consider the tree $\nu a.n \in \nu\text{-Tree}$. This time, suppose we have a judgement $\varphi \vdash \nu a.n : q$. Note that this was not derived by (Push) since n contains no lambdas. Hence we can invert (Nu) and obtain a transition $(q, \nu a_k) \Rightarrow (q', l)$ and a judgement $\Gamma, \varphi[a_k := l] \vdash n : q'$. Then by IH we obtain $n \in \mathcal{L}(\mathcal{A}, q', \varphi[a_k := l])$ and thus directly conclude $\nu a.n \in \mathcal{L}(\mathcal{A}, q, \varphi)$ with (LNu). Again, the converse direction is similar.

The second claim follows as usual by an analysis of the judgement $\varphi \vdash n : q$. Directly excluded are the rules (Push), (Abs) and (Var) as the first two yield function types and the third requires a non-empty context. On the other hand, the nu case is immediate by induction and the constant cases for $k = 0$ are trivial. Hence only (App) is left and the justification works as before. First, since n is in particular β-normal the left-most operand of the application must be a constant a_k, so we have $\varphi \vdash a_k n_1 \ldots n_k : q$. Then this can be inverted which yields base typings $\varphi \vdash n_i : q_i$. By IH those n_i are ν-trees, hence so is $a_k n_1 \ldots n_k$. □

Then we can conclude the general correspondence by the generic justification:

Corollary 4.14. *For any NTA $\mathcal{A}$ it is $\varphi \vdash_{\mathcal{A}} s : q$ iff there is $s \Downarrow n$ with $n \in \mathcal{L}(\mathcal{A}, q, \varphi)$.*

Proof. This is similar as in Section 4.1, the assignment φ does not matter. □

Finally, having established the decidability of acceptance and emptiness for NTA in Section 3.6, we can conclude the same decidability results for the systems $\mathcal{A}$-ITLC as in the instances before:

Lemma 4.15. *For any NTA $\mathcal{A}$, the system $\mathcal{A}$-ITLC has decidable normal typability, normal type checking and base type inhabitance (cf. Lemma 4.5).*

Proof. This is by the same justification as Lemma 4.5. □

Chapter 5
Discussion

In the mathematical part of this thesis we presented intersection type systems, studied a selection of automaton models and instantiated a generic correspondence proof for type systems and automata. We will now conclude with some conceptual remarks on proof strategies, design decisions and future work.

First, we would like to emphasize the benefits of the **inductive (re)formulation** of automaton acceptance that we have studied for all discussed automaton models. This was a crucial simplification when proving the correspondence lemmas for unranked tree automata (Lemma 4.6) and ranked ν-tree automata (Lemma 4.13). Moreover, being purely structural, our reformulations are arguably more intuitive than run-based definitions and hence constitute a preferable alternative. We conjecture that compositional formulations that allow for elegant inductive reasoning are possible for most automaton models that run on finite structures. Also, the compositional nature of acceptance only became observable when studying automata in the context of type systems so this is the first of possibly further fruitful results of examining type system-automaton correspondences in their own right. Compositional definitions of acceptance for automata on infinite structures such as infinite trees can be obtained with coinduction or parity conditions. In [KO09] Kobayashi and Ong introduced typing games as the corresponding treatment on the side of type systems.

Next it is worth to spend a few words on the **use of intersection types**. Actually, intersection types are not at all necessary in order to capture the transitions of automata, all constant rules we have encountered in our instances construct simple (function) types. In fact, on strong normal forms (well-formed automaton input) STLC and ITLC agree. Thus, if embedding the constant rules into a simple type system, we can still show that base-typable programs evaluate to accepted input. Note that this uses subject reduction and normalisation of STLC. However, the converse relies on subject expansion and hence fails for STLC. That is, a program that has an accepted normal form is not guaranteed to be simply typable. So from this perspective, the reason for using intersection types is the higher amount of typable terms and hence the larger extent of the generalisation of acceptance from actual automaton input to input-computing programs.

Another design decision was to present NTA as customized automata for the acceptance of ν-trees. The alternative would be a more **abstract definition of binding tree automata** which would be a general streamlining of Stirling's non-deterministic dependency tree automata. This was mentioned in Section 3.6. We decided to only study the special case where the binders are nus since we were interested in deciding properties of ν-trees. In an abstract model, further languages with binders could be examined, in particular terms of λ-calculus themselves or logical expressions with quantifiers. Furthermore, it is clear that the non-determinism of NTA transitions can be extended to alternation, which would yield an automaton model similar to Stirling's alternating dependency tree automata.

Finally, it should be clarified that the required **distinct binders for ν-trees** can be relaxed in order to obtain a stronger automaton model. This was only included to enable the completeness direction of the correctness of our denotational semantics of ν-trees (Theorem 3.8). For instance, if we were to allow reuse of bound names it would be $[\![a(\nu b.b)(\nu c.c)]\!] = [\![a(\nu b.b)(\nu b.b)]\!]$ but $a(\nu b.b)(\nu c.c) \not\approx_\alpha a(\nu b.b)(\nu b.b)$. This is a consequence of our notion of α-equivalence being parallel by employing a simultaneous permutation of multiple names. Hence in this setting equality of denotations would be strictly coarser than α-equivalence of bound names. However, allowing rebinding increases the expressiveness of NTA in the following sense. Consider the set of ν-trees defined by $n_1 := a_0$ and $n_{k+1} := \nu b_1.b_1 n_k$ for all $k \in \mathbb{N}$. It is easy to define an NTA that accepts this language and hence, by including the semantics $[\![n_k]\!]$ for all n_k, denotes all linear $\mathbb{A}$-trees with leaf a_0 - a reasonable property. This is not expressible without rebinding since defining the ν-tree language would require infinitely many different names of rank 0 but processing those exceeds the capabilities of finite collections of transitions. Here both alternatives have their respective justifications and we just opted for the restriction to allow for studying a strong semantical correctness theorem.

We end by outlining three possible directions of future work:

- **Developing unranked ν-trees and unranked ν-tree automata.** This is a conservative generalisation since all ranked trees can obviously be considered unranked. In this framework we can study more natural result of reduction of programs. More general, a theory of unranked dependency automata can be imagined applicable to all types of binding languages. In particular, this would allow for examining "automaton application" where we are interested in finding an automaton that accepts trees of the form $s[t/x]$ if we have respective automata accepting $\lambda x.s$ and t. This would make the representation of binding inside the automata explicit and enables the notion of compositional properties.

- **Considering simply typed λY-terms as the base language.** In a typed lambda calculus with recursion operators at each type we can still express a lot of programs that compute trees. However, this puts certain bounds on finding types in derivations since the intersection types of an automaton system need to refine the simple types of the programs. This might suffice to make general type checking decidable. In general, it is worth exploring how decidability results for automata can be obtained from decidability results of the corresponding type systems. This would be the converse to our treatment as we started with automaton models already known to be decidable. Perhaps, exploiting the type system-automaton correspondences in the other direction could be a fruitful impulse for the development of automata theory. As a side effect, there is a notion of "order" considering λY-terms capturing the complexity of the types involved. It will be interesting to study the inherited notion of order for terms and automata and define a hierarchy of complexity levels with perhaps different properties.

- **Relating the work to the nominal type theory developed in [Che09].** Starting with a nominal set of type variables, this theory develops a system involving name abstraction and concretion both at type and term level. This might be useful in order to rephrase the automaton system we developed for nominal word automata. Exploiting all given symmetry along these lines, we might be able to capture decidability of automaton problems from the perspective of the type system.

Bibliography

[Bar84] H.P. Barendregt. *The lambda calculus: its syntax and semantics*. Studies in logic and the foundations of mathematics. North-Holland, 1984.

[BCDC83] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symbolic Logic*, 48(4):931–940, 12 1983.

[BDS13] H. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Lambda Calculus with Types. Cambridge University Press, 2013.

[BKL14] Mikolaj Bojańczyk, Bartek Klin, and Slawomir Lasota. Automata theory in nominal sets. *Logical Methods in Computer Science*, 10(3), 2014.

[CDC80] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame J. Formal Logic*, 21(4):685–693, 10 1980.

[CDG^{+}07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[Che09] James Cheney. A simple nominal type theory. *Electr. Notes Theor. Comput. Sci.*, 228:37–52, 2009.

[Chu40] A. Church. A formulation of a simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940. http://www.jstor.org/stable/2266866Electronic Edition.

[CLT05] Julien Cristau, Christof Löding, and Wolfgang Thomas. *Fundamentals of Computation Theory: 15th International Symposium, FCT 2005, Lübeck, Germany, August 17-20, 2005. Proceedings*, chapter Deterministic Automata on Unranked Trees, pages 68–79. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[CR36] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39:472–482, 1936. http://www.jstor.org/stable/2268573Electronic Edition.

[FKS11] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular Code-Based Cryptographic Verification. In *18th ACM Conference on Computer and Communications Security*, Chicago, United States, October 2011.

[Fra39] A. A. Fraenkel. Über die unabhangigkeit des auswahlaxioms und einiger seiner folgerungen. *Journal of Symbolic Logic*, 4(1):30–31, 1939.

[Hin92] J. Roger Hindley. Types with intersection: An introduction. *Formal Aspects of Computing*, 4(5):470–486, 1992.

[Hin97] J.R. Hindley. *Basic Simple Type Theory*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1997.

[HORT16] M. Hague, L. Ong, S. Ramsay, and T. Tsukada. Alternating dependency tree automata, higher-type mu-calculus and type-checking games. *Submitted for Review*, 2016.

[How80] William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

[KMS15] Dexter Kozen, Konstantinos Mamouras, and Alexandra Silva. Completeness and incompleteness in nominal kleene algebra. In Wolfram Kahl, Michael Winter, and José N. Oliveira, editors, *Relational and Algebraic Methods in Computer Science - 15th International Conference, RAMiCS 2015, Braga, Portugal, September 28 - October 1, 2015, Proceedings*, volume 9348 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2015.

[KO09] Naoki Kobayashi and C.-H. Luke Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, pages 179–188. IEEE Computer Society, 2009.

[Kob09] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 416–428. ACM, 2009.

[Koz97] Dexter C. Kozen. *Automata and Computability*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1997.

[LP11] S. Lösch and A. M. Pitts. Relating Two Semantics of Locally Scoped Names. In Marc Bezem, editor, *Computer Science Logic (CSL'11) - 25th*

International Workshop/20th Annual Conference of the EACSL, volume 12 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 396–411, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Ode94] Martin Odersky. A functional theory of local names. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 48–59, January 1994.

[Pit13] A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013.

[PS93] Andrew M. Pitts and Ian Stark. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In *Mathematical Foundations of Computer Science: Proceedings of the 18th International Symposium MFCS '93*, number 711 in Lecture Notes in Computer Science, pages 122–141. Springer-Verlag, 1993.

[Ram14] Steven J. Ramsay. Exact intersection type abstractions for safety checking of recursion schemes. In Olaf Chitil, Andy King, and Olivier Danvy, editors, *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, pages 175–186. ACM, 2014.

[Rus08] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):222–262, 1908.

[Sta98] Ian Stark. Names, equations, relations: Practical ways to reason about *new*. *Fundamenta Informaticae*, 33(4):369–396, April 1998.

[Sti09] Colin Stirling. *Foundations of Software Science and Computational Structures: 12th International Conference, FOSSACS 2009, York, UK, March 22-29. Proceedings*, chapter Dependency Tree Automata, pages 92–106. Springer, Berlin, Heidelberg, 2009.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[Urz99] Pawel Urzyczyn. The emptiness problem for intersection types. *The Journal of Symbolic Logic*, 64(3):1195–1215, 1999.