Nominal sets and automata: Representation theory and computations

David Venhoek

Supervisors:
dr. J.C. Rot
MSc. J.S. Moerman

Second reader:
dhr. dr. W. Bosma

Institute for Computing and Information Sciences
Radboud University Nijmegen

Contents

Introduction **v**

1 Languages and Automata Theory **1**
1.1 Languages . . . 1
1.2 Computing languages . . . 2
1.2.1 Formalizing deterministic finite automata . . . 3
1.3 Minimizing deterministic finite automata . . . 4
1.3.1 Intermezzo: partitions and equivalence relation . . . 5
1.3.2 Moore's algorithm . . . 7
1.3.3 Hopcroft's algorithm . . . 10

2 Theory of Nominal Sets **15**
2.1 G-sets . . . 15
2.1.1 Representation theory of G-sets . . . 17
2.2 Data symmetries and nominal sets . . . 18
2.2.1 Nominal automata . . . 21
2.2.2 Representation theory . . . 21

3 Exploiting the total order symmetry **27**
3.1 Specializing representation theory . . . 27
3.1.1 Orbits and Sets . . . 27
3.1.2 Equivariant maps . . . 30
3.1.3 Products . . . 32
3.2 Calculating with nominal sets . . . 35
3.2.1 Complexity analysis . . . 37

4 Nominal Automata and Applications **39**
4.1 Nominal automata theory . . . 40
4.1.1 Moore's algorithm for nominal automata . . . 44
4.2 Performance testing of Ons . . . 47
4.2.1 Performance on minimization . . . 49
4.2.2 Performance on learning . . . 50

5 Conclusion and Outlook **53**

A Admittance of least supports and infinite products **55**

Bibliography **59**

Introduction

Software verification, the checking of correct behaviour of programs, is one of the big open problems in computer science. Deterministic finite automata (DFAs), models of programs running with finite input and output alphabets, using a finite number of program states, can be used for verification. They can model software and hardware, and these models can be used to prove properties on the behaviour of the software or hardware. Furthermore, there exist automated techniques to learn these models. This has recently been successfully applied, for instance to several implementations of the TCP protocol [10].

For many programs, the input and output domains are infinite. When using DFAs these input and output alphabets need to be reduced to some finite set. In [10] this is done through domain-specific reductions of the input and output alphabets. Another approach is to use a more general type of automaton able to directly use the infinite input alphabet, but which has limited access to the values of that alphabet. Such automata have recently been proven amenable to automated learning [1, 5, 6, 8, 17, 23].

The theory of nominal sets provides a mathematically elegant way to deal with such infinite sets, by introducing restrictions on how programs can distinguish elements. It was first introduced as a formalism for name binding, in a form allowing only equality checking between elements [11, 19]. Nominal sets have since been applied to various other problems in computer science, an overview of which is given in [18]. Bojańczyk et al. [3] introduced nominal automata, and parameterize the structure of nominal sets over data symmetries, allowing more flexibility in the type of relations between elements. This provides a powerful framework for working with automata using infinite input and output alphabets. Most importantly, they show that important concepts from classical automata theory such as minimization carry over to nominal automata.

Furthermore, Bojańczyk et al. [3] derive a representation theory for nominal sets. This representation theory provides a finite notation for nominal sets that directly expresses their nominal structure. However, current general purpose implementations of nominal sets, Nλ [14] and Lois [16, 15], are not based on representation theory. Instead, they use logical formula to represent nominal sets, using an SMT solver to implement calculations. Although more straightforward to implement, this approach does not allow one to derive time complexity results for algorithms common in the classical setting, such as a variant of Moore's algorithm for minimizing nominal automata. A framework for

calculating with nominal sets based directly on the representation theory offers the promise of allowing such complexity results to be derived.

In this thesis, we will explore the representation theory of nominal sets. As a side result, we show that there exist data symmetries for which products do not preserve finiteness, but which are otherwise well-behaved. We then use the representation theory to build a concrete formalism for working with nominal sets of the total order symmetry. For this, we develop methods for manipulating nominal sets beyond describing their structure, which is necessary to work with functions and products on nominal sets. We then use the representation theory to build a concrete representation of nominal sets over the total order symmetry. This also requires the development of methods for manipulating nominal sets, beyond just describing their nominal structure. Based on this, we constructed a library for calculating with nominal sets: Ons. Ons is the first general purpose library known to the author using representation theory to implement nominal sets.

The use of representation theory allows us to derive concrete complexity results for a nominal version of Moore's algorithm. Furthermore, comparing running time of implementations of algorithms for minimization and learning between Ons and the existing libraries Nλ and Lois shows that Ons can be a significant improvement. Particularly in learning nominal automata, Ons performs significantly better than existing solutions. These results were also presented by the author in collaboration with J. Rot and J. Moerman in [17], which contains a more succinct presentation of the ideas here.

In short, the following new results are presented:

- The Ons library for calculating with nominal sets, which uses representation theory directly, together with complexity results for basic set operations. The author considers this the main contribution.
- A concrete complexity analysis of Moore's algorithm for minimizing nominal automata.
- An example of a data symmetry that admits least supports (a technical property allowing a finite representation of nominal sets), but for which products do not preserve finiteness.

Structure of this thesis

Chapter 1 gives an introduction to automata theory in the context of normal (finite) sets. We then switch to nominal sets, introducing the concept and general representation theory in Chapter 2.

In Chapter 3, we specialize the representation theory to the total order data symmetry. During this, we also construct representations of functions and products of nominal sets. This leads us to the introduction of the library Ons, for which we derive complexity results on the basic operations.

Finally, in Chapter 4, we discuss the theory of nominal deterministic automata. We derive a variant of Moore's algorithm for nominal automata,

proving its correctness and time complexity bounds. We then use two algorithms on nominal automata to benchmark the performance of Ons: automaton minimization and learning.

Related work

Although Ons is the first general purpose library known to the author using the representation theory of nominal sets, it has been previously applied in specific problem domains. Mihda [9] implements a translation between π-calculus and history dependent automata using a finite representation of nominal sets over the equality symmetry. Fresh OCaml [21] and Nominal Isabelle [22] focus on name binding and α-conversion. None of these provide a general purpose library for manipulating nominal sets, and we will not provide further details here.

The libraries Nλ [14] and Lois [16, 15], which will be compared to Ons in this text, use a different approach for working with nominal sets. Both use SMT solvers to implement calculations on logical formula representing nominal sets.

Acknowledgements

This thesis, and the research for it, would not have been possible without the support of many great people around me, whom I would like to thank for their help here.

First of all I would like to thank my parents, for supporting me, both financially and mentally, in my studies, and for encouraging me to pursue knowledge in the topics of interest to me. I would like to thank my family, for supporting and encouraging my curiosity, and for always being interested in what I do.

I would like to thank my friends for making life fun, pulling me along to things I might not otherwise do and making life about more than just physics and mathematics.

I would also like to thank Tom van Bussel and Rick Erkens for providing feedback on some of the drafts of this text.

Finally, I would like to thank my supervisors, Jurriaan Rot and Joshua Moerman, for providing an exciting subject to work on, and for guiding me through the process of writing this thesis, especially when things got though.

Chapter 1

Languages and Automata Theory

We will start by considering deterministic finite automata. These are a model of computation that produces answers to membership queries on a set. Whilst not as powerful as things like Turing machines, this is compensated for by having a rich theory for manipulating them.

This chapter intends to give an introduction to the theory of deterministic finite automata. A more in-depth introduction to the subject can be found in [13], which was the basis for most of this chapter.

1.1 Languages

As we will see soon, deterministic finite automata are a method for deciding whether some object is a member of a set. To formalize the computations for this, some method is needed for describing the objects.

For example, when considering the question whether or not a given natural number is even, some way is needed to give the natural number. In written text, it is typical to do this as a sequence of symbols, in this case the digits 0 through 9. We interpret these sequences of digits as natural numbers in the usual way. Extending this idea, we can describe the objects using series of symbols, of which we have only a finite number. The sets to do membership tests on are then simply sets containing such sequences.

This notion of sets of sequences can be captured in the notion of a *language*. We let A denote a finite set of symbols, the *alphabet*. Finite sequences of those symbols are called *words*. The set of all words over a given alphabet is denoted by A^*. A *language* is then a subset $\mathcal{L} \subseteq A^*$. We will denote the length of a word w as $|w|$. Extending this notation, for an $a \in A$, we use $|w|_a$ to denote the number of times the letter a occurs in w.

For notation, throughout the rest of this text, we will use w, u and v for words. The empty word will be denoted by ϵ. Individual letters will be denoted

with a, b and c. It is sometimes useful to denote the result of concatenating two words, or of letters and words. This will be done by simply placing them adjacent, so uav is the word formed by starting with the letters from u, followed by the letter a, and then the letters of v at the end.

As an informal example, consider the question of whether a given natural number is even. For the alphabet, take $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Each word in A^* is now a sequence of digits. Each of these sequences can be interpreted as a natural number in the usual way. We will just ignore leading zeros. The even numbers now form the language

$$\mathcal{L}_{\text{even}} = \{w \in A^* \mid \text{the last letter of } w \text{ is in } \{0, 2, 4, 6, 8\}\}.$$

1.2 Computing languages

For our purposes, we can now think of a computation as some set of instructions that, using perhaps some extra symbols, manipulates the sequence of input symbols and eventually produces a decision of whether or not its input is in a language.

We will focus here on a restricted model for computations: deterministic finite automata. There are languages that cannot be computed using deterministic finite automata even though they can by other methods of computation. However, the simpler model for deterministic finite automata gives more tools for analyzing and optimizing their operation.

Deterministic finite automata can be thought of as restricted, formalized flowcharts. As an example, consider again the case of even numbers. Drawing a simple flowchart for determining membership would give something along the lines of Figure 1.1. We will call the boxes *states*, and the arrows *edges*.

For deterministic finite automata, there are two rules:

1. The word can only be processed one character at a time.

2. Only after the entire word is processed is the decision made whether the word is in the language or not.

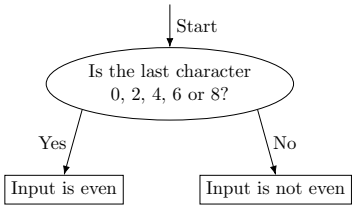

Figure 1.1: Flowchart for deciding whether or not a number is even.

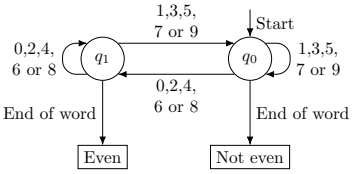

Figure 1.2: Deciding whether a number is even by only considering the current character.

These rules come down to the following: Start at the beginning of the word. For each letter, look at what that letter is, and follow the corresponding edge. Once every letter has been looked at, check whether the current state is one that says the word should be accepted. Writing the example of recognizing even numbers in this way gives Figure 1.2.

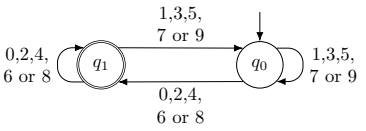

Figure 1.3: Automaton for the language of even numbers.

The notation in Figure 1.2 is still verbose. There are two simplifications: Start is just denoted by an arrow from nowhere. Second, instead of adding end-of-word arrows, a number is accepted if following the edges for a word ends up in a doubly circled state. The result of these transformations is shown in Figure 1.3.

1.2.1 Formalizing deterministic finite automata

The above idea of a deterministic finite automaton can be formalized in the language of sets and functions:

Definition 1. A *deterministic finite automaton* over an alphabet A is a tuple (Q, F, δ, q_0), where Q is a finite set of *states*, $F \subseteq Q$ is the set of *accepting states*, $q_0 \in Q$ is the *initial state*, and $\delta : Q \times A \to Q$ is the *transition function*.

Translating back to the images from before, each element of Q corresponds to a state in the diagram. The elements of F are the states that accept words finishing there. The initial state is q_0. Finally, all the internal arrows in the diagram are given by the transition function.

Translating the automaton for recognizing even numbers gives the following: The alphabet is given by $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. As there are two states, $Q = \{q_0, q_1\}$. Of this, q_1 corresponds to the accepting state in the diagram, so $F = \{q_1\}$. The transition function is then given by:

$$\delta(q, n) = \begin{cases} q_1 & n \in \{0, 2, 4, 6, 8\} \\ q_0 & \text{otherwise} \end{cases}$$

The entire automaton is then (Q, F, δ, q_0).

To formalize the language recognized by an automaton, let us first extend δ to words:

$$\delta^*(q, \epsilon) = q$$
$$\delta^*(q, wa) = \delta(\delta^*(q, w), a)$$

The language recognized by the automaton is then the set of words w with the property that $\delta^*(q_0, w) \in F$.

More generally, it is possible to associate a language with each of the states of the automata, representing the language computed if that state were the initial state. This is denoted as $\mathcal{L}_q = \{w \in A^* \mid \delta^*(q, w) \in F\}$.

1.3 Minimizing deterministic finite automata

For a given language, there can be multiple automata that recognize it. For example, the automaton in Figure 1.4 recognizes the same language as the automaton given previously in Figure 1.3.

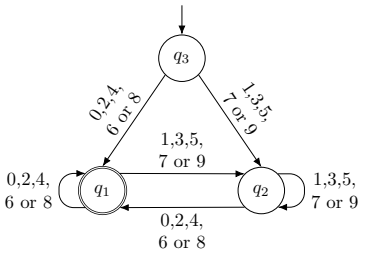

Figure 1.4: A second automaton for recognizing even numbers.

Since these automata recognize the same language, it would be interesting to see whether we could map the states of one onto the states of the other. For this to be reasonable, it is necessary that such maps interact well with the structure of the automata. For our applications, it will be easier to formalize these concepts on automata without initial states. Hence, we will omit the initial state q_0 from here on, giving an automaton as a tuple (Q, F, δ). We interpret the automaton as recognizing the collection of languages $\mathcal{L}_q$.

Definition 2. Given two automata (Q, F, δ) and (Q', F', δ'), a *morphism* of automata is a function $f : Q \to Q'$ such that $q \in F \Leftrightarrow f(q) \in F'$ and $f(\delta(q, a)) = \delta'(f(q), a)$. A morphism is an *isomorphism* if the function f is bijective.

For the example above, the function mapping q_1 and q_3 to q_1, and q_2 to q_2, is a morphism from the automaton of Figure 1.4 to the automaton of Figure 1.3.

Lemma 1. *A morphism of automata is* language preserving*, in the sense that, given a morphism f from (Q, F, δ) to (Q', F', δ'), it holds that $\mathcal{L}_q = \mathcal{L}_{f(q)}$.*

Proof. This can be shown using induction on words. Starting with the empty word ϵ:

$$\epsilon \in \mathcal{L}_q \Leftrightarrow q \in F \Leftrightarrow f(q) \in F' \Leftrightarrow \epsilon \in \mathcal{L}_{f(q)}.$$

Next, let us assume that for fixed w, for any state q', $w \in \mathcal{L}_{q'} \Leftrightarrow \mathcal{L}_{f(q')}$. Given a letter $a \in A$, it then holds that

$$aw \in \mathcal{L}_q \Leftrightarrow w \in \mathcal{L}_{\delta(q,a)} \Leftrightarrow w \in \mathcal{L}_{f(\delta(q,a))} \Leftrightarrow w \in \mathcal{L}_{\delta'(f(q),a)} \Leftrightarrow aw \in \mathcal{L}_{f(q)}.$$

By induction, it follows that for any word w, $w \in \mathcal{L}_q \Leftrightarrow w \in \mathcal{L}_{f(q)}$, and hence $\mathcal{L}_q = \mathcal{L}_{f(q)}$. □

Through morphisms, one also obtains an ordering on automata. If there is a surjective morphism from an automaton (Q, F, δ) to a second automaton (Q', F', δ'), we say $(Q, F, \delta) \geq (Q', F', \delta')$. For example, the morphism given above allows us to state that the automaton of Figure 1.4 is greater than or equal to the automaton of Figure 1.3.

One question is then whether there exists a *minimal* automaton. For this, let us first fix what we mean by minimal:

Definition 3. An automaton (Q, F, δ) is minimal iff for any automaton (Q', F', δ') that recognizes the same languages as (Q, F, δ), there is a surjective morphism f from (Q', F', δ') to (Q, F, δ).

This definition was chosen as it is easier to generalize later, but there is another useful characterization:

Theorem 1. *The following are equivalent:*

1. *The automaton (Q, F, δ) is minimal.*
2. *For any two states $q, q' \in Q$, with $q \neq q'$, $\mathcal{L}_q \neq \mathcal{L}_{q'}$.*

Proof. This is easiest to prove by constructing an explicit minimal automaton. Let $\bar{Q} = \{\mathcal{L}_q \mid q \in Q\}$, and $\bar{F} = \{\mathcal{L}_q \mid q \in F\}$. Construct $\bar{\delta}$ using

$$\bar{\delta}(\mathcal{L}_q, a) = \{w \mid aw \in \mathcal{L}_q\}.$$

To show that $\bar{\delta}(\mathcal{L}_q, a) \in \bar{Q}$, observe that $aw \in \mathcal{L}_q$ if and only if $w \in \mathcal{L}_{\delta(q,a)}$. Observe that, by construction, $(\bar{Q}, \bar{F}, \bar{\delta})$ recognizes the same collection of languages as (Q, F, δ).

Let us use this automaton to prove (1) $\Rightarrow$ (2): Since (Q, F, δ) is minimal, there is a surjective morphism $f : \bar{Q} \to Q$. Now let $q, q' \in Q$. Since f is a surjection, there exist $\bar{q}, \bar{q}' \in \bar{Q}$ such that $f(\bar{q}) = q$ and $f(\bar{q}') = q'$. Suppose $\mathcal{L}_q = \mathcal{L}_{q'}$, then $\mathcal{L}_{\bar{q}} = \mathcal{L}_{f(\bar{q})} = \mathcal{L}_q = \mathcal{L}_{q'} = \mathcal{L}_{f(\bar{q}')} = \mathcal{L}_{\bar{q}'}$. By construction of $\bar{Q}$, this implies $\bar{q} = \bar{q}'$, and hence $q = q'$, proving (2).

For (2) $\Rightarrow$ (1), consider the function $f : Q \to \bar{Q}$ given by $f(q) = \mathcal{L}_q$. By (2), this is a bijection, and by construction of $(\bar{Q}, \bar{F}, \bar{\delta})$ it is a morphism. It follows that f is an isomorphism between (Q, F, δ) and $(\bar{Q}, \bar{F}, \bar{\delta})$. It is thus enough to show that $(\bar{Q}, \bar{F}, \bar{\delta})$ is minimal.

For this, consider an automaton (Q', F', δ') recognizing the same set of languages. Construct $g : Q' \to \bar{Q}$ such that $g(q) = \mathcal{L}_q$. Since $aw \in \mathcal{L}_q$ if and only if $w \in \mathcal{L}_{\delta'(q,a)}$, this is a morphism. Since both (Q', F', δ') and $(\bar{Q}, \bar{F}, \bar{\delta})$ recognize the same set of languages, which by construction is precisely $\bar{Q}$, f is necessarily surjective. This proves that $(\bar{Q}, \bar{F}, \bar{\delta})$ is minimal. □

Corollary 1. *For each automaton, there exists a unique (up to isomorphism) minimal automaton, and this automaton has a minimal number of states.*

The theory on minimizing automata presented above can also be formulated in terms of coalgebras. The decision was made not to pursue this line because of the extra prerequisites needed. The general idea is to formulate minimization as factoring of the unique morphism to the final coalgebra. A good introduction to this subject for the interested reader is found in [20].

1.3.1 Intermezzo: partitions and equivalence relation

Given the existence of minimal automata, it would be useful to have a method for finding a minimal automaton given a (potentially) non-minimal automaton. As we will see in the next section, this involves grouping similar states in

the automaton together. Let us look here at two ideas for representing such groupings:

Definition 4. Given a set X, a *partition* $\mathcal{P}$ is a subset of the powerset $P(X)$ such that the following hold:

- $\emptyset \notin \mathcal{P}$.
- For any $U, V \in \mathcal{P}$, if $U \neq V$, then $U \cap V = \emptyset$.
- $\bigcup_{U \in \mathcal{P}} U = X$.

Definition 5. Given a set X, an *equivalence relation* is a subset $R \subseteq X \times X$ such that the following hold:

- For any $x \in X$, $(x, x) \in R$.
- Given $(x, y) \in R$, also $(y, x) \in R$.
- Let $x, y, z \in X$. If $(x, y) \in R$ and $(y, z) \in R$ then also $(x, z) \in R$.

Note that partitions and equivalence relations act as each others dual. Given a partition $\mathcal{P}$, one can consider the equivalence relation given by xRy iff there exists an $U \in P$ such that $x \in U$ and $y \in U$. We will denote this relation as $R_{\mathcal{P}}$. Conversely, given a relation $R \subseteq X \times X$, the *quotient* set $X/R = \{\{y \in X \mid xRy\} \mid x \in X\}$ is a partition (from the construction it might seem as though an element $x \in X$ is in many subsets, but all those subsets are in fact the same element of the partition). This duality can be used to translate descriptions of partitions to relations, and vice versa:

Definition 6. A relation R *refines* a second relation R' iff $R \subseteq R'$. It is a *strict refinement* if $R \neq R'$. Translated to partitions, a partition $\mathcal{Q}$ refines a partition $\mathcal{P}$ if for any $Q \in \mathcal{Q}$ there exists a $P \in \mathcal{P}$ such that $Q \subseteq P$.

Definition 7. A partition $\mathcal{P}$ *saturates* a set F when there is a subset $\mathcal{Q} \subseteq \mathcal{P}$ such that $\bigcup_{Q \in \mathcal{Q}} Q = F$. An equivalence relation $R \subseteq X \times X$ saturates F when xRy implies that x and y are either both in F, or neither.

When considering all partitions on a set, it is possible to define a join and meet operation such that the entire structure forms a lattice:

Definition 8. Given to partitions $\mathcal{P}, \mathcal{Q}$, we define:

$$\mathcal{P} \vee \mathcal{Q} = \{P \cap Q \mid P \in \mathcal{P}, Q \in \mathcal{Q}, P \cap Q \neq \emptyset\}$$
$$\mathcal{P} \wedge \mathcal{Q} = \{\{x \mid \exists n, z_1, \ldots, z_n : (xR_{\mathcal{P}}z_1 \vee xR_{\mathcal{Q}}z_1) \wedge (z_1R_{\mathcal{P}}z_2 \vee z_1R_{\mathcal{Q}}z_2) \wedge \ldots \wedge (z_nR_{\mathcal{P}}y \vee z_nR_{\mathcal{Q}}y)\} \mid y \in X\}$$

The join $\vee$ finds the coarsest partition refining both of $\mathcal{P}$ and $\mathcal{Q}$, and the meet $\wedge$ the finest partition coarsening $\mathcal{P}$ and $\mathcal{Q}$.

In this lattice, $\{X\}$ acts as the identity for $\vee$, and $\{\{x\} \mid x \in X\}$ acts as the identity for $\wedge$. Verification of the lattice axioms is left as an exercise to the reader.

1.3.2 Moore's algorithm

The tools from the previous chapter can now be used to define algorithms for constructing the minimal automaton. In the proof of Theorem 1 the minimal automaton was constructed explicitly using the languages recognized by each state. Unfortunately, those languages can be infinitely large.

However, Theorem 1 also shows that it is sufficient to construct an automaton where no two states represent the same language. This can be done by computing which states in the original automaton represent the same language, and then merging all such states together.

More formally, it is sufficient to compute for all states whether or not they compute the same language. This gives the relation $\equiv_L \subseteq Q \times Q$, defined through $q_1 \equiv_L q_2 \Leftrightarrow \mathcal{L}_{q_1} = \mathcal{L}_{q_2}$. The minimal automaton is then given by (Q', F', δ') with:

$$\begin{aligned} Q' &= Q/\equiv_L \\ F' &= F/\equiv_F \\ \delta'(S, a) &= \{\delta(q, a) \mid q \in S\} \end{aligned}$$

The map δ' is well defined since if $q \equiv_L q'$, then also $\delta(q, a) \equiv_L \delta(q', a)$. Note that the state space can be calculated similarly by taking the image of Q under $\mathcal{L}_{(\_)}$, since $Q/\equiv_L \cong \{\mathcal{L}_q \mid q \in Q\}$.

The question then is how to effectively calculate $\equiv_L$. In the approach taken by Moore's algorithm, we define inductively a set of equivalence relations that approximate $\equiv_L$:

Definition 9. Define the relations $\equiv_i \subseteq Q \times Q$, with $i \in \mathbb{N}$ as follows:

$$\begin{aligned} q \equiv_0 q' &\text{ iff } q \in F \Leftrightarrow q' \in F \\ q \equiv_{i+1} q' &\text{ iff } q \equiv_i q' \text{ and for all } a \in A : \delta(q, a) \equiv_i \delta(q', a) \end{aligned}$$

Lemma 2. *For any two states $q, q' \in Q$, we have $q \equiv_i q'$ if and only if, for all words w of length at most i, $\delta^*(q, w) \in F \Leftrightarrow \delta^*(q', w) \in F$. In other words, $q \equiv_i q'$ if and only if q and q' recognize the same words up to length i.*

Proof. This is proven by induction. First, for $i = 0$, the requirement $\delta^*(q, w) \in F \Leftrightarrow \delta^*(q, w)$ for all words of length at most 0 reduces to $\delta^*(q, \epsilon) \in F \Leftrightarrow \delta^*(q, \epsilon)$. As $\delta^*(q, \epsilon) = q$, this is reduces to $q \in F \Leftrightarrow q' \in F$, which is the definition of $\equiv_0$.

Assume the lemma holds for a fixed i. Consider two states $q, q' \in Q$. We need to prove two things:

$$q \equiv_{i+1} q' \Rightarrow (\forall w \in A^* : |w| \leq i + 1 \Rightarrow (\delta^*(q, w) \in F \Leftrightarrow \delta^*(q', w) \in F)) \quad (1.1)$$

$$(\forall w \in A^* : |w| \leq i + 1 \Rightarrow (\delta^*(q, w) \in F \Leftrightarrow \delta^*(q', w) \in F)) \Rightarrow q \equiv_{i+1} q' \quad (1.2)$$

Let us start with proving 1.1. Let w be a word of length at most $i+1$. If w is of length i or shorter, then $\delta^*(q, w) \in F \Leftrightarrow \delta^*(q', w)$ follows from the induction hypothesis and the fact that $q \equiv_{i+1} q' \Rightarrow q \equiv_i q'$ by definition.

If w is of length $i+1$, we can split it into a letter $a \in A$, and a word $v \in A^*$ such that $w = av$ and $|v| = i$. We then calculate:

$$\begin{aligned}
\delta^*(q,w) \in F &\Leftrightarrow \delta^*(q,av) \in F \\
&\Leftrightarrow \delta^*(\delta(q,a),v) \in F \\
&\Leftrightarrow \delta^*(\delta(q',a),v) \in F \qquad \text{Since } q \equiv_{i+1} q' \Rightarrow \delta(q,a) \equiv_i \delta(q',a) \\
&\Leftrightarrow \delta^*(q',av) \\
&\Leftrightarrow \delta^*(q',w)
\end{aligned}$$

Now for 1.2. Let us assume that the left hand side holds. We then need to prove $q \equiv_{i+1} q'$.

First, since the left hand side holds, for any word w of length at most i, $\delta^*(q,w) \in F \Leftrightarrow \delta^*(q',w)$. By induction, it follows that $q \equiv_i q'$.

Next, take any $a \in A$. Using the left hand side, we have that, for any word v of length at most i, $\delta^*(q,av) \in F \Leftrightarrow \delta^*(q',av) \in F$. Then, by definition of δ^*, we also have $\delta^*(\delta(q,a),v) \in F \Leftrightarrow \delta^*(\delta(q',a),v) \in F$. From the induction hypothesis, it follows that $\delta(q,a) \equiv_i \delta(q',a)$.

Since $q \equiv_i q'$ and $\delta(q,a) \equiv_i \delta(q',a)$ for all $a \in A$, it follows that $q \equiv_{i+1} q'$.

By induction, the lemma thus holds. □

Using this, we find that $q \equiv_L q'$ if and only if $q \equiv_i q'$ for all $i \in \mathbb{N}$. This still is an infinite number of checks. However, using the above observation that $\equiv_{i+1}$ follows directly from $\equiv_i$ and δ, it follows that if $\equiv_i = \equiv_{i+1}$, then $\equiv_i = \equiv_j$ for all $j \geq i$. Since Q is finite, so are all $\equiv_i$. But for a finite relation, there exists only a finite number of refinements. Hence there will always be a finite i with $\equiv_i = \equiv_{i+1}$. Thus, only a finite sequence is ever needed to calculate $\equiv_L$. These observations lead directly to Moore's algorithm:

Algorithm 1 Moore's minimization algorithm for DFAs

Require: Automaton (Q, F, δ).
1: $i \leftarrow 0$.
2: $\equiv_{-1} \leftarrow Q \times Q$.
3: $\equiv_0 \leftarrow F \times F \cup (Q \backslash F) \times (Q \backslash F)$.
4: **while** $\equiv_i \neq \equiv_{i-1}$ **do**
5: $\quad \equiv_{i+1} = \{(q_1,q_2) \mid (q_1,q_2) \in \equiv_i \wedge \forall a \in A, (\delta(q_1,a), \delta(q_2,a)) \in \equiv_i\}$.
6: $\quad i \leftarrow i+1$.
7: **end while**
8: $E \leftarrow Q/_{\equiv_i}$.
9: $F_E \leftarrow \{e \in E \mid \forall s \in e, s \in F\}$.
10: Let δ_E be the map with $\delta_E(e,a) = \{\delta(q,a) \mid q \in e\}$.
11: **return** (E, F_E, δ_E).

Theorem 2. *The time complexity of Algorithm 1 is $O(n^3k)$, where n is the number of states, and k the number of letters in the alphabet.*

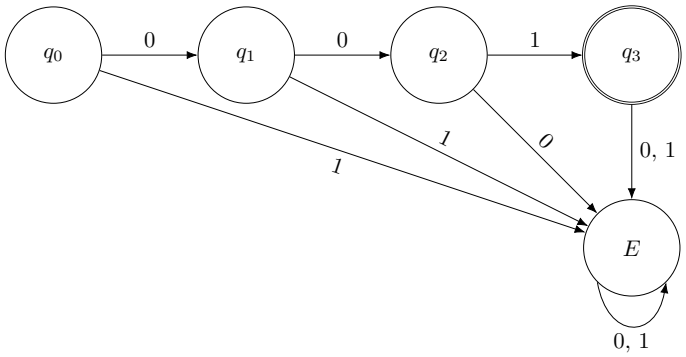

Figure 1.5: Automaton recognizing the language $\{001\}$.

Proof. There are two big operations in the algorithm. The construction of the equivalence relation in the loop on lines 4-7, and the construction of the result automaton on lines 8-10.

Starting with the creation of the result automaton, line 8 takes $O(n^2)$ time to compute the quotient. Line 9 then only needs $O(n)$ time to compute the set of final states. If the map δ_E is stored explicitly, by keeping track of which inputs map to which outputs, constructing that takes $O(nk)$ time.

In the loop on lines 4-7, line 5 takes the majority of the time. Given that the relation contains at most n^2 pairs, line 5 can be computed in at most $O(n^2k)$ time. Next, it is necessary to calculate how often loop runs. This is most easily done by considering the relations as partitions, via the duality described above.

Each loop iteration computes a finer partition, or stops the loop. Hence, there are at most as many loop iterations as there are partitions of Q that each refine the previous one. Since each refined partition splits at least two states from each other, there can only be n such partitions in a sequence. Hence, the entire loop is iterated through at most $O(n)$ times.

Using this, the time complexity of lines 4-7, and hence of the entire algorithm, is $O(n^3k)$. □

Note that the bound on the number of times the loop on lines 4-7 is run is strict. The minimal automaton recognizing the language $\{000\ldots01\}$, with n zeros before the single 1, provides an example of an automaton hitting this bound. See Figure 1.5 for an example of such an automaton for $n = 2$.

It is actually possible to implement a variant of Moore's algorithm in a way that gives a better complexity than the approach given above. For this, it is useful to make two changes: First, instead of computing the equivalence relation first, and calculating the corresponding partition only in the final step, it is possible to just compute the partition in each iteration of the loop.

Second, it is useful to consider each letter one at a time when splitting the partitions, instead of considering all of them at once. Doing this gives a different sequence of partitions, but after i iterations over all letters, the

resulting partition will be a refinement of the partition corresponding to $\equiv_i$, meaning that the algorithm will still be correct.

By storing the partition as a list allowing iteration over the states in order, refining the partition with a single letter can be done in $O(n)$ time. Using that, the complexity of the entire algorithm, as presented in listing 2, reduces to $O(n^2k)$.

Algorithm 2 Optimized Moore's minimization algorithm

Require: Automaton (Q, F, δ).
1: $\mathcal{P}_{-1} \leftarrow \{Q\}$.
2: $\mathcal{P}_0 \leftarrow \{F, Q \setminus F\}$.
3: $i \leftarrow 0$.
4: **while** $\mathcal{P}_{i-1} \neq \mathcal{P}_i$ **do**
5: $\quad \mathcal{Q} \leftarrow \mathcal{P}_i$.
6: $\quad$ **for** $a \in A$ **do**
7: $\quad\quad \mathcal{Q} \leftarrow \mathcal{Q} \vee \{\{q \mid \delta(q, a) \in P\} \mid P \in \mathcal{P}\}$.
8: $\quad$ **end for**
9: $\quad i \leftarrow i + 1$.
10: $\quad \mathcal{P}_i \leftarrow \mathcal{Q}$.
11: **end while**
12: $F_{\mathcal{P}} \leftarrow \{P \mid P \in \mathcal{P}_i \wedge \forall x \in P : x \in F\}$.
13: Let $\delta_{\mathcal{P}}$ be the map with $\delta_{\mathcal{P}}(P, a) = \{\delta(q, a) \mid q \in P\}$.
14: **return** $(\mathcal{P}_i, F_{\mathcal{P}}, \delta_{\mathcal{P}})$.

1.3.3 Hopcroft's algorithm

Algorithm 2 can be further optimized to give a running time of $O(nk \log(n))$. This algorithm is due to Hopcroft [12], and is shown in Algorithm 4. To derive this algorithm, we will follow the approach of [2].

First, let us split up the computation of $\{\{q \mid \delta(q, a) \in P\} \mid P \in \mathcal{P}\}$. We can split this partition into a number of smaller partitions of the form $\{\{q \mid \delta(q, a) \in P\}, \{q \mid \delta(q, a) \notin P\}\}$ for each of the $P \in \mathcal{P}$. Then

$$\{\{q \mid \delta(q, a) \in P\} \mid P \in \mathcal{P}\} = \bigvee_{P \in \mathcal{P}} \{\{q \mid \delta(q, a) \in P\}, \{q \mid \delta(q, a) \notin P\}\}$$

Let us call these simpler partitions *splitters*, and use the shorthand notation of (P, a) for $\{\{q \mid \delta(q, a) \in P\}, \{q \mid \delta(q, a) \notin P\}\}$.

Now instead of applying all splitters at once by computing the join with $\{\{q \mid \delta(q, a) \in P\} \mid P \in \mathcal{P}\}$, it is possible to compute the same by just computing the joins with each of the splitters separately.

The number of splitters that need to be applied can then be reduced by noting that applying the same splitter twice is exactly the same as applying it only once. We could exploit this by keeping track of which splitters we already applied when generating the splitters for the next stage. However, it is more

effective to let go of the entire idea of calculating $\equiv_L$ in stages, and instead generate new splitters when the current splitter splits an existing segment of the partition. All splitters that are generated but not yet applied we keep in a separate work queue. This gives Algorithm 3:

Algorithm 3 Splitter based minimization algorithm

Require: Automaton (Q, F, δ).
$\mathcal{P} \leftarrow \{F, Q \setminus F\}$.
$W \leftarrow \{(F, a) \mid a \in A\} \cup \{(Q \setminus F, a) \mid a \in A\}$.
while $W \neq \emptyset$ **do**
    Let (P, a) be any element of W.
    Remove (P, a) from W.
    $\mathcal{Q} \leftarrow \mathcal{P} \vee (P, a)$.
    for $P' \in \mathcal{Q} \setminus \mathcal{P}$, $b \in A$ **do**
        Add (P', b) to W.
    end for
    $\mathcal{P} \leftarrow \mathcal{Q}$.
end while
$F_{\mathcal{P}} \leftarrow \{P \mid P \in \mathcal{P} \wedge \forall x \in P : x \in F\}$.
Let $\delta_{\mathcal{P}}$ be the map with $\delta_{\mathcal{P}}(P, a) = \{\delta(q, a) \mid q \in P\}$.
return $(\mathcal{P}, F_{\mathcal{P}}, \delta_{\mathcal{P}})$.

In this implementation, no splitter will be applied twice. However, as seen in the next lemma, there is still some extra work done:

Lemma 3. *Given that Q is the set of all states, $P \subseteq Q$, $P' \subset P$, and $a \in A$, the following partitions are equal:*

$$(P, a) \vee (P', a) \vee (P \setminus P', a) \tag{1.3}$$
$$(P, a) \vee (P', a) \tag{1.4}$$
$$(P', a) \vee (P \setminus P', a) \tag{1.5}$$

Proof. This is seen by writing out each of the partitions and noting that they all equal

$$\{\{q \mid \delta(q, a) \in Q \setminus P\}, \{q \mid \delta(q, a) \in P'\}, \{q \mid \delta(q, a) \in P \setminus P'\}\}.$$

□

This lemma can be used to reduce the number of splitters that need to be applied. When a splitter (P, a) is used to split some equivalence P' of the partition into two parts P'' and $P''' = P' \setminus P''$. In Algorithm 3, both (P'', b) and (P''', b) are added to the work queue. Since at the creation of P', (P', b) was already added, all three of (P', b), (P'', b) and (P''', b) are in the queue. But by the above lemma, when two out of three of those are applied, the third has no further effect. Hence, only one of (P'', b) and (P''', b) needs to be added to the work queue.

Note that this still works even if (P', b) was eliminated in a previous step by similar considerations. This is because the splitters already in the work queue ensure that applying (P', b) has no additional effect, hence the situations with and without it in the queue are equivalent. A reduction of the starting work queue is found by a similar argument, using the fact that the splitter (Q, a) does nothing for any $a \in A$. This gives Hopcroft's algorithm:

Algorithm 4 Hopcroft's algorithm for minimizing deterministic finite automata.

Require: Automaton (Q, F, δ).
  $\mathcal{P} \leftarrow \{F, Q \backslash F\}$.
  $W \leftarrow \{(P, a) \mid P = min(F, Q \backslash F), a \in A\}$.
  while $W \neq \emptyset$ **do**
    Let (P, a) be an element of W.
    Remove (P, a) from W.
    for $P' \in \mathcal{P}$ split by (P, a) **do**
      Label the pieces (P, a) splits P' in as P'' and P'''.
      Update the partition with P'' and P''' in place of P'.
      for $a \in A$ **do**
        Add $(min(P'', P'''), a)$ to W.
      end for
    end for
  end while
  $F_{\mathcal{P}} \leftarrow \{P \mid P \in \mathcal{P} \wedge \forall x \in P : x \in F\}$.
  Let $\delta_{\mathcal{P}}$ be the map with $\delta_{\mathcal{P}}(P, a) = \{\delta(q, a) \mid q \in P\}$.
  return $(\mathcal{P}, F_{\mathcal{P}}, \delta_{\mathcal{P}})$.

Lemma 4. *Hopcroft's algorithm runs in $O(kn \log(n))$.*

Proof. To show this, it easiest to first give a visual representation of the partition generated by Hopcroft's algorithm, and how it came to be. We represent the partition as a forest, with the sets F and $Q \setminus F$ as roots. Every time some segment of the partition is split, we draw the new, smaller segments as children of the original segment. Next, we colour the smaller of each of the split sets (including the smaller of F and $Q \backslash F$). This gives a tree like the one in Figure 1.6.

We now note the following: each state of original automaton is in exactly one of the leaves in this forest. Furthermore, if we start walking up the tree from that leaf, every time we meet a blue segment, the next segment will be at least twice as big. Hence, we will only see at most $\log(n)$ blue nodes walking to the root.

This shows that every state q is used in at most $k \log(n)$ splitters. More importantly, for any state, there are at most $k \log(n)$ splitters (P, a) for which $\delta(q, a) \in P$.

This last fact matters because it is possible to apply a splitter (P, a) to the result partition in $O(\#\{q \mid \delta(q, a) \in P\})$. Counting from the states instead of from the splitters, the fact that each state is in the preimage set of

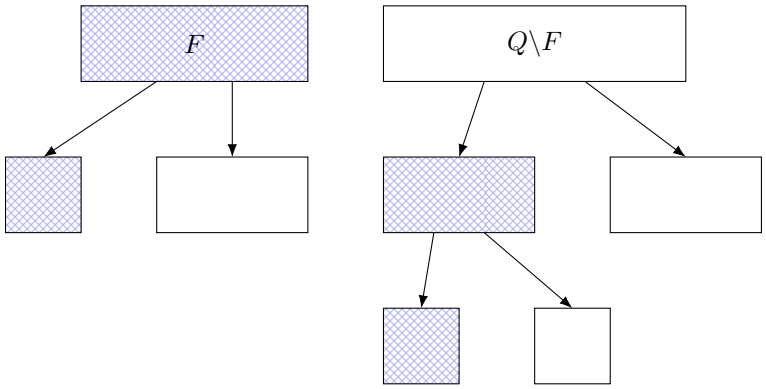

Figure 1.6: Forest representing a partition and it's history

at most $k \log(n)$ splitters implies that applying all splitters will take at most $O(kn \log(n))$.

Since the running time of the algorithm is dominated by the time needed to apply the splitters, the complexity of the entire algorithm is $O(kn \log(n))$. □

Chapter 2

Theory of Nominal Sets

The deterministic finite automata introduced in the previous chapter, while useful, are also limited in their expressivity. For example, consider the language where the first letter of each word matches the last letter. If we have a finite alphabet, such a language can be recognized by a deterministic finite automaton. However, for an infinite input alphabet, an automaton recognizing the same language would require an infinite number of states.

Example 1. Let A be a (potentially infinite) alphabet. Let A^* denote the set of finite sequences of elements of A, known as the words. We define $\mathcal{L}_\circ \subseteq A^*$ such that $w \in \mathcal{L}_\circ$ if and only if the first and last letters of w are the same.

The idea behind a nominal set is to provide a concept of sets that are infinite, but that still have enough structure to admit a finite representation. Using such sets, we can then extend deterministic finite automata to accept languages over infinite alphabets. We will use Example 1 to illustrate the concepts needed throughout this chapter, finally constructing a nominal automaton that recognizes the language $\mathcal{L}_\circ$.

In developing nominal sets and automata over them, we will follow the approach taken in [3]. The interested reader can also find more details there, including connections to model theory.

Here, the focus will be on obtaining a notion of infinite sets having finite representations. Such a representation will be the cornerstone of creating a toolkit for calculations with such sets, and by extension on automata defined over such sets. As we will see, G-sets, sets with a group action on them, fail in this regard. This is solved by introducing extra restrictions, resulting in nominal sets. These are shown to have a finite representation.

2.1 G-sets

The language of Example 1 has an intriguing property. When we take a permutation of the alphabet, and apply it to each of the individual letters of a word,

this does not change whether the first and last letters match. In essence, the language itself has symmetry.

Furthermore, these symmetries form a group: we can apply one permutation after another to get a new permutation. Also, since permutations are bijections, they can also be undone.

We formalize this concept of sets with symmetry, with subsets and functions on those sets that respect that symmetry, through the notion of G-sets:

Definition 10. Given a fixed group G, a set X with a right G-action is called a *G-set*. Recall that a right action of a group on a set is a function $\phi : X \times G \to X$ such that $\phi(\phi(x, \pi), \sigma) = \phi(x, \pi\sigma)$ for all $\pi, \sigma \in G$, $x \in X$, and for the identity $1 \in G$, we have $\phi(x, 1) = x$ for all $x \in X$. We will use the shorthand notation $x\pi$ for the right action of π on x.

Definition 11. Given two G-sets X and Y, the cartesian product $X \times Y$ forms a G-set whose action satisfies $(x, y)\pi = (x\pi, y\pi)$ for all $\pi \in G$.

Definition 12. Given a G-set X, the power set $\mathcal{P}(X)$ is a G-set with the action $C\pi = \{c\pi \mid c \in C\}$ for all $\pi \in G$, $C \subseteq X$.

Definition 13. Given two G-sets X and Y, a function $f : X \to Y$ is *equivariant* if for all $\pi \in G, x \in X$ we have $f(x\pi) = f(x)\pi$. The G-sets X and Y are isomorphic if there exists a bijective equivariant function from X to Y.

Definition 14. A subset $Y \subset X$ of a G-set is an *equivariant subset* if, for all $x \in X$ and $\pi \in G$, if $x \in Y$ then also $x\pi \in Y$. That is, if Y is preserved under the group action.

Before looking at an example, let us make two observations: First, any set X can be turned into a G-set with the trivial action. Applying this to the set $\{0, 1\}$, we see that for a subset $Y \subseteq X$, the characteristic function $\chi_Y : X \to \{0, 1\}$ is an equivariant function if and only if Y is an equivariant subset.

Furthermore, when $Y \subseteq X$ is an equivariant subset, the action of G on X gives a natural action on Y through restriction. Unless otherwise noted, when considering equivariant subsets, the action of G on the elements of Y will be the action of X.

G-sets allow us to formalize the intuitive notion of symmetry from Example 1. Given the alphabet A, the group $G = \mathrm{Sym}(A)$ is the permutation group of A. The right action of G on A can then be used to inductively define a right action on A^*: For all $\pi \in G$, let $\epsilon\pi = \epsilon$, and for $a \in A$, $w \in A^*$, $(aw)\pi = (a\pi)(w\pi)$, applying the group action to each letter individually. This makes A^* a G-set.

Let us consider the subset $A^+ = A^* \setminus \{\epsilon\}$. This is an equivariant subset of A^*. The function $h : A^+ \to A$ be defined by $h(aw) = a$ for all $a \in A$, $w \in A^*$ is then equivariant. To see this, take a $\pi \in G$, and consider $h((aw)\pi)$. Calculating, we find $h((a\pi)(w\pi)) = a\pi = h(aw)\pi$.

Now consider the language $\mathcal{L}_\circ$. Since, for all $\pi \in G$, a word $w \in \mathcal{L}_\circ$ if and only if $w\pi \in \mathcal{L}_\circ$, we find that $\mathcal{L}_\circ$ is an equivariant subset of A^*.

Let us now attempt to create finite representation of a subclass of G-sets. We start by considering what the building blocks of G-sets are. In traditional

set theory, this would be individual elements, but removing an element from a G-set can cause the resulting set to be no longer closed under it's action.

However, the group action divides a G-set X into orbits, subsets of elements that can be transformed into each other by the group elements. Each such an orbit is closed under the group action, which means that its removal is compatible with the group action. These orbits are the natural building blocks of the G-set.

Definition 15. The *size* of a G-set X, denoted $\mathrm{N}(X)$, is the number of orbits of which X consists. If $\mathrm{N}(X)$ is finite, we call the set X orbit-finite, or finite for short. If $\mathrm{N}(X) = 1$, we will call X a single-orbit nominal set, or single orbit for short.

In the example above, the alphabet A consists of a single orbit. In contrast, the language $\mathcal{L}_\circ$ consists of an infinite number of orbits.

2.1.1 Representation theory of G-sets

In the representation theory, the focus will be on orbit-finite G-sets. These can be written as a finite union of orbits. This reduces the problem to finding a finite description of each orbit.

Definition 16. For a subgroup $H \leq G$, the set $[H]^c = G/H^r = \{H\pi \mid \pi \in G\}$, the right quotient of G with respect to H, is given a G-set structure through the action $(H\pi) \cdot \sigma = H(\pi\sigma)$.

Theorem 3. *For each single orbit G-set X there is a subgroup $H \leq G$ such that $[H]^c \cong X$.*

Proof. Let x be an element of X. Then any element of X can be written as $x \cdot \pi$, with $\pi \in G$. Let $H = \{\pi \in G \mid x \cdot \pi = x\}$. H is a subgroup of H since if π, σ are elements of H, then $x \cdot (\pi\sigma) = (x \cdot \pi) \cdot \sigma = x \cdot \sigma = x$, implying $\pi\sigma \in H$, and $x = x \cdot (\pi\pi^{-1}) = (x \cdot \pi) \cdot \pi^{-1} = x \cdot \pi^{-1}$, hence $\pi^{-1} \in H$.

Claim: $[H]^c \cong X$.

To see this, consider the function $f : X \to [H]^c$ with $f(x \cdot \pi) = H\pi$. First up, we need to show that this is a well-defined function. Suppose $x \cdot \pi = x \cdot \sigma$. Then $x \cdot (\pi\pi^{-1}) = x \cdot (\sigma\pi^{-1})$, hence $x = x \cdot (\sigma\pi^{-1})$. This shows that $\sigma\pi^{-1}$ is in H. But then $H\pi = H(\sigma\pi^{-1})\pi = H\sigma(\pi\pi^{-1}) = H\sigma$, hence f is well-defined.

It remains to show that f is injective, surjective and equivariant. Equivariance follow trivially from $f(x\pi \cdot \sigma) = H\pi\sigma = (H\pi)\sigma = f(x\pi)$. Next, suppose $f(x\pi) = f(x\sigma)$. Then $H\pi = H\sigma$, and thus $H\pi\pi^{-1} = H\sigma\pi^{-}1$, implying $\sigma\pi^{-1} \in H$. But then $x\sigma\pi^{-1} = x$, and hence $x\pi = x\sigma\pi^{-1}\pi = x\sigma$, thus f is injective. Since $f(x\pi) = H\pi$, f is also trivially surjective.

We have thus shown that $[H]^c \cong X$, proving the theorem. □

From the above lemma, it follows that a single-orbit G-set can be represented by a subgroup $H \leq G$. A G-set consisting of an arbitrary finite number of orbits

can then be represented as the union of its orbits, each orbit in turn represented as a subgroup $H \leq G$.

For the representation suggested by the above theorem to be finite, there need to be only a countable number of subgroups $H \leq G$ that we need to be able to describe. Unfortunately, there are groups G, including the group $\mathrm{Sym}(\mathbb{N})$, that have an uncountable number of subgroups. This needn't be a problem on its own, as long as a countable subset of them is enough to represent all orbits.

Unfortunately, this isn't the case. To demonstrate this, let us first consider when two subgroups represent the same orbit:

Theorem 4. *Given subgroups H and K of G, the single-orbit sets $[H]^c$ and $[K]^c$ are isomorphic if and only if the groups H and K are conjugate. That is, if there exists a $\pi \in G$ such that $K = \pi H \pi^{-1}$.*

Proof. Suppose H and K conjugate. Then there exists a $\pi \in G$ such that $K = \pi H \pi^{-1}$. Define $f : [H]^c \to [K]^c$ such that $f(H\sigma) = K\pi\sigma$. To show that this is well-defined consider the case where $H\sigma = H\sigma'$. Then $\sigma\sigma'^{-1} \in H$. It follows that $\pi\sigma\sigma'^{-1}\pi^{-1} \in K$, and hence $K\pi\sigma' = K\pi\sigma\sigma'^{-1}\pi^{-1}\pi\sigma = K\pi\sigma$.

By construction, f is an equivariant surjection. To show that it is also injective, suppose $f(H\sigma) = f(H\sigma')$. Then $K\pi\sigma = K\pi\sigma'$, and hence $\pi\sigma(\pi\sigma')^{-1} = \pi\sigma\sigma'^{-1}\pi^{-1} \in K$. But since $K = \pi H \pi^{-1}$, it follows that $\sigma\sigma'^{-1} \in H$, hence $H\sigma = H\sigma'$.

We conclude that if H and K are conjugate, then $[H]^c \cong [K]^c$.

To prove the converse, let $f : [H]^c \to [K]^c$. Let $\pi \in G$ be such that $f(H) = K\pi$. For all $\sigma \in H$, it then holds that $K\pi = f(H) = f(H\sigma) = f(H)\sigma = K\pi\sigma$, implying $\pi\sigma\pi^{-1} \in K$. Hence $\pi H \pi^{-1} \leq K$.

Similarly, if $\sigma \in K$, then $H\pi^{-1} = f^{-1}(K) = f^{-1}(K\sigma) = H\pi^{-1}\sigma$, giving $\pi^{-1}\sigma\pi \in H$. Hence $\pi^{-1}K\pi \leq H$, and thus $K \leq \pi H \pi^{-1}$. Combining with the previously shown $\pi H \pi^{-1} \leq K$ gives $K = \pi H \pi^{-1}$, hence H and K conjugate. □

However, as shown by proposition 8.7 in [3], the group $\mathrm{Sym}(\mathbb{N})$ has uncountably many non-conjugate subgroups. As these each represent a different orbit, there exists uncountably many possible orbits for a G-set, implying that there is no finite representation.

2.2 Data symmetries and nominal sets

To work around the problem of there being uncountably many orbits, a further restriction is needed on the sets considered. For this, we first extend the information carried with the symmetry group:

Definition 17. A pair $(\mathcal{D}, G)$, with $G \subseteq \mathrm{Sym}(\mathcal{D})$ is called a *data symmetry*.

The set $\mathcal{D}$, also called the *data set*, is the natural set G acts on.

Definition 18. The *equality symmetry* is the data symmetry $(\mathbb{N}, \mathrm{Sym}(\mathbb{N}))$.

Assuming that the alphabet from Example 1 is countably infinite, it is isomorph to the set $\mathbb{N}$. The equality symmetry can then be used in lieu of $\mathrm{Sym}(A)$, defining the action of $\mathrm{Sym}(\mathbb{N})$ on A using a bijection between $\mathbb{N}$ and A.

The equality symmetry is not the only data symmetry one could use, other examples are:

Definition 19. The *total order symmetry* is the data symmetry given by

$$(\mathbb{Q}, \{g \in \mathrm{Sym}(\mathbb{Q}) \mid \forall x, y \in \mathbb{Q}, x < y \Rightarrow xg < yg\}).$$

Definition 20. The *integer symmetry* is the data symmetry given by

$$(\mathbb{Z}, \{g \in \mathrm{Sym}(\mathbb{Z}) \mid \forall x, y \in \mathbb{Z}, x - y = xg - yg\}).$$

Examples of more exotic symmetries, and their applications can be found in [4].

In basing the group on data symmetries, the elements of the data set can be used to characterize which group elements affect an element of a G-set, and which don't.

Definition 21. For an element x in a G-set X, $C \subseteq \mathcal{D}$ is a *support* of x iff for all $g \in G$ such that $(\forall d \in C, dg = d)$ we have $xg = x$.

Definition 22. A *nominal set* X is a G-set with the property that, for all $x \in X$, there exists a finite set $C \subseteq \mathcal{D}$ that is a support of x.

Note that, for a given data symmetry $(\mathcal{D}, G)$, the set $\mathcal{D}$ is always a nominal set, as for any element $d \in \mathcal{D}$, the set $\{d\}$ forms a finite support. As any nominal set is a G-set, the definitions of equivariant maps, equivariant subsets and products are inherited from those.

The following are also examples of nominal sets:

Example 2. The set $\mathcal{P}_n(\mathcal{D}) = \{C \subseteq \mathcal{D} \mid \#C = n\}$, the set of all subsets of size n, together with the action $C\pi = \{c\pi \mid c \in C\}$, is a nominal set.

Example 3. Given any data symmetry $(\mathcal{D}, G)$, the set $\{\mathcal{D} \setminus \{d\} \mid d \in \mathcal{D}\}$ is a nominal set. For the element $\mathcal{D} \setminus \{d\}$, $\{d\}$ is a support.

In contrast, we find that the following, while a G-set, is not a nominal set:

Example 4. Let $\mathbb{N}_e = \{n \in \mathbb{N} \mid n \text{ is even}\}$. Using the equality symmetry, the G-set $\{\mathbb{N}_e\pi \mid \pi \in \mathrm{Sym}(\mathbb{N})\}$ is not nominal.

For nominal sets X, when a set C supports an element x of X, we would like to think of x as being constructed from some subset of the elements of C. However, this view isn't always completely accurate. For example, consider the nominal set $\mathbb{Z}$ using the integer symmetry. If $y \in \mathbb{Z}$, then y is supported by the set $\{y\}$. But, y is also supported by the set $\{n\}$, for all $n \in \mathbb{Z}$. Unless $n = y$, this last set does not contain y itself.

The integer symmetry has a further problem. Ideally, when applying basic set operations on an orbit finite nominal set, we would like to end up with another orbit finite nominal set. Unfortunately, in the case of the integer symmetry, while the set $\mathbb{Z}$ is single orbit, the product $\mathbb{Z} \times \mathbb{Z}$ has infinitely many orbits. This can be seen by considering that for any pair (x, y), the difference between x and y is fixed under the group action. As there are infinitely many possible differences, this implies that $\mathbb{Z} \times \mathbb{Z}$ consists of an infinite number of orbits.

To deal with the first problem, we introduce a notion of least support:

Definition 23. For an element x in a G-set X, a finite set $C \subseteq \mathcal{D}$ is a least support of x if C is a support of x and for any finite set $C' \subseteq \mathcal{D}$ that is a support of x, $C \subseteq C'$.

Definition 24. A data symmetry $(\mathcal{D}, G)$ admits least supports if, for any element $x \in X$, where X is a nominal set over $(\mathcal{D}, G)$, there exists a least support S of x.

As seen above, the integer symmetry does not admit least support. Both $\{1\}$ and $\{2\}$ are supports of $1 \in \mathbb{Z}$, but their only common subset $\emptyset$ is not. In contrast, the equality symmetry and total order symmetry do admit least supports [3].

Note that although admittance of least supports is formulated as a property over nominal sets for a given data symmetry, it is purely a property of the data symmetry itself. This can be more readily seen once we develop some further representation theory.

One can hope that by requiring admittance of least support, basic set operations also become well behaved, in the sense that the product, union, intersection and difference of orbit finite sets are again orbit finite. Although the equality and total order symmetries are well behaved in this way, this does not hold in general. An example of a data symmetry admitting least support, but having products of orbit finite sets that are not themselves orbit finite is given in Appendix A

As we will see when discussing the representation theory of the orbits of nominal sets, the existence of least supports plays a pivotal role in providing a finite representation. For that reason, we will only use data symmetry admitting least supports from here on.

Definition 25. The *dimension* of an element $x \in X$ of a nominal set X, denoted $\dim(x)$, is the size of the least support of x. This is extended to orbit-finite nominal sets X by $\dim(X) = \max_{x \in X} \dim(x)$.

The notion of dimension can be seen as a description of how "complex" a nominal set is. This will become more clear in the next chapter, when we develop a framework for calculating with nominal sets over the total order symmetry.

2.2.1 Nominal automata

The notion of nominal set allows us to formalize the notion of an infinite automaton that has a finite description posited in Example 1 at the start of this chapter.

Definition 26. A *deterministic nominal automaton* over a finite nominal alphabet A is a tuple (Q, F, δ), where Q is a finite nominal set of states, $F \subseteq Q$ an equivariant subset of accepted states, and $\delta : Q \times A \to Q$ an equivariant function acting as the transition function.

Given this definition, the extension of δ to words and the concept of the language accepted by a state translate without changes.

As an example, let us work this out for the automaton accepting words that begin and end with the same letter. Given that we are only interested in whether letters are equal or not, we will work with the equality symmetry $(\mathbb{N}, \mathrm{Sym}(\mathbb{N}))$. For ease of reference, we will ignore any alternative labeling for the alphabet, and take $A = \mathbb{N}$. Our automaton then needs 3 groups of states:

- An initial state, where no letter has yet been read.
- States where we have read some number of letters, remembering the first, and where the last letter read was not equal to the first.
- States where we have read some number of letters, remembering the first, and where the last letter read was equal to the first.

Combining these, we will use $Q = \{()\} \cup \{(0, i) \mid i \in \mathbb{N}\} \cup \{(1, i) \mid i \in \mathbb{N}\}$. The group action on Q is given by $()\pi = ()$, $(0, i)\pi = (0, i\pi)$ and $(1, i)\pi = (1, i\pi)$, where $\pi \in G$ and $i \in \mathbb{N}$. As a subset of this, $F = \{(1, i) \mid i \in \mathbb{N}\}$ will be the states where the last letter read was equal to the first.

The transition function δ is then defined as:

$$\delta((), a) = (1, a)$$

$$\delta((0, i), a) = \begin{cases} (0, i) & i \neq a \\ (1, i) & i = a \end{cases}$$

$$\delta((1, i), a) = \begin{cases} (0, i) & i \neq a \\ (1, i) & i = a \end{cases}$$

Given these definitions $\mathcal{L}_{()}$ is the language of words whose first and last letters are equal. A visual representation of this automaton is shown in Figure 2.1. A further discussion of nominal automata will be postponed until after we have developed a full framework for calculating with nominal sets.

2.2.2 Representation theory

Again, let us now turn to finding a useful representation theory for nominal sets. Since nominal sets are a special kind of G-sets, we can follow a similar

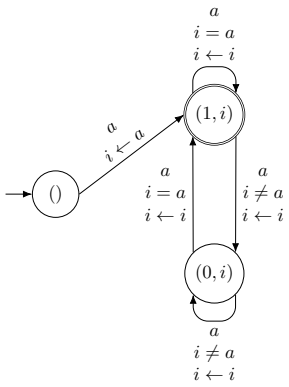

Figure 2.1: Nominal automaton for recognizing the language of words starting and ending with the same letter.

approach. A nominal set can still be seen as a collection of orbits, hence the size of a nominal set will refer to the number of orbits it contains.

For representing nominal sets, a set can be written down as a collection of orbits. Each orbit can then still be represented by a subgroup $H \leq G$. To get a finite representation however, we will need to use some of the extra properties of nominal sets to get a finite description of H.

As seen in the proof of Theorem 3, given a fixed element $x \in X$, the group H representing the orbit of x can be chosen such that is precisely the group elements having x as a fixed point. Given this correspondence, it is possible to translate the concept of support from elements to subgroups. For this, it is useful to first introduce some notation for special subgroups, and for extending and restricting the domain of subgroups.

Definition 27. For an element $d \in \mathcal{D}$, let $G_d = \{\pi \in G \mid d\pi = d\}$.

Definition 28. For a subset $C \subseteq \mathcal{D}$, let $G_C = \{\pi \in G \mid \forall d \in C, d\pi = d\}$.

Note that, given two sets $C \subseteq \mathcal{D}$ and $C' \subseteq \mathcal{D}$, if $C \subseteq C'$, then $G_{C'} \leq G_C$.
Using these, let us define the notion of support of subgroups:

Definition 29. Given a subgroup $H \leq G$, $C \subseteq \mathcal{D}$ is a support of H iff $G_C \leq H$.

Lemma 5. *Let $x \in X$ be an element of a nominal set, and $C \subseteq \mathcal{D}$. Then the following are equivalent:*

1. *C is a support of x.*
2. *C is a support of the subgroup $H = \{\pi \in G \mid x\pi = x\}$.*

Proof. To show (1) $\Rightarrow$ (2), let C be a support of x. If $\pi \in G_C$, then for all $d \in C$, $d\pi = d$. Since C is a support of x, this implies $x\pi = x$. Hence $\pi \in H$, and since both H and G_C are groups, $G_C \leq H$.

To prove the converse, let C be a support of H. Let $\pi \in G$ be such that $\forall d \in C, d\pi = d$. Then $\pi \in G_C$, and since $G_C \leq H$, also $\pi \in H$. But by definition of H, $\pi \in H$ implies that $x\pi = x$. Hence, C is a support of x. □

Using the above definitions, the concept of least support translates immediately, and also gives a more contained definition of when a data symmetry admits least supports:

Definition 30. A subset $C \subseteq \mathcal{D}$ is a least support of a group H if C is a support of H and, for any support C' of H, $C \subseteq C'$.

Lemma 6. *Given a data symmetry $(\mathcal{D}, G)$, the following are equivalent:*

1. *$(\mathcal{D}, G)$ admits least supports.*
2. *Given a nominal set X, and an element $x \in X$, if $C \subseteq \mathcal{D}$ and $C' \subseteq \mathcal{D}$ are finite supports of x, then $C \cap C'$ is a support of x.*
3. *Given finite subsets $C \subseteq \mathcal{D}$ and $C' \subseteq \mathcal{D}$, $G_{C\cap C'} = \langle G_C, G_{C'} \rangle$, the group generated by the elements of G_C and $G_{C'}$.*

Proof. (1) ⇒ (2): Consider a nominal set X, an element $x \in X$, and two finite supports $C \subseteq \mathcal{D}$ and $C' \subseteq \mathcal{D}$ of x. From (1), it follows that x has a least support $C'' \subseteq \mathcal{D}$. Since C and C' are also supports of x, it follows that $C \subseteq C''$ and $C \subseteq C'$. Hence, also $C'' \subseteq C \cap C'$. Since C'' is a support of x, and $C'' \subseteq C \cap C'$, it follows that $C \cap C'$ is also a support of x, proving (2).

(2) ⇒ (3): Let $C \subseteq \mathcal{D}$ and $C' \subseteq \mathcal{D}$ be finite sets. By construction, we have that $G_C \leq \langle G_C, G_{C'} \rangle$ and $G_{C'} \leq \langle G_C, G_{C'} \rangle$. From this, it follows that C and C' support the element $\langle G_C, G_{C'} \rangle$ of $[\langle G_C, G_{C'} \rangle]^c$. From (2), it the follows that $C \cap C'$ also supports $\langle G_C, G_{C'} \rangle$, and hence $G_{C\cap C'} \leq \langle G_C, GC' \rangle$. From the definition of $G_{C\cap C'}$ it also follows that $G_C \leq G_{C\cap C'}$ and $G_{C'} \leq G_{C\cap C'}$. This means that also $\langle G_C, C_{C'} \rangle \leq G_{C\cap C'}$. Since both $G_{C\cap C'} \leq \langle G_C, G_{C'} \rangle$ and $\langle G_C, C_{C'} \rangle \leq G_{C\cap C'}$, it follows that $\langle G_C, C_{C'} \rangle = G_{C\cap C'}$, proving (3).

(3) ⇒ (1): Let X be a nominal set, and $x \in X$. Let $H \leq G$ be the subgroup with the property that $\pi \in H$ implies $x\pi = x$. Now construct the set $I = \{C' \mid C' \subseteq \mathcal{D} \wedge C' \text{ finite support of } x\}$. Let C be a smallest element of I in terms of element count. This exists since X is nominal, and hence there is some finite support C of x. By construction of I we have $G_C \leq H$. Let C' be any support of x. Then $G_{C'} \leq H$, and hence by construction $\langle G_C, G_{C'} \rangle \leq H$. From (3), we obtain $\langle G_C, G_{C'} \rangle = G_{C\cap C'}$, and hence $C \cap C'$ is also a finite support of x, implying $C \cap C' \in I$. Since C is the smallest element of C, it follows that $C \cap C'$ is equal in size to C. Since C is finite, we find $C = C \cap C'$, or equivalently $C \subseteq C'$. Using this construction we can find a least support for any element of a nominal set, proving that $(\mathcal{D}, G)$ admits least supports. □

Intuitively, the least support can be thought of as specifying the large scale structure of a subgroup. They state that, so long as a group element doesn't touch a finite subset of elements, it will be a member of the subgroup. However, a subgroup can still contain group elements that, for example, swap two of

the elements in the least support. To work with this type of structure, let us introduce the notions of restrictions and extensions.

Definition 31. The restriction of an element $\pi \in G$ to a subset $C \subseteq \mathcal{D}$, written as $\pi|_C$, is the restriction of the corresponding function $\pi : \mathcal{D} \to \mathcal{D}$ to the domain C.

Definition 32. The restriction of a subgroup $H \leq G$ to a subset $C \subseteq \mathcal{D}$ is $H|_C = \{\pi|_C \mid \pi \in H \wedge C\pi = C\}$.

Note that while $\pi|_C$ is defined for any combination of $\pi \in G$, it is only an element of $H|_C$ if $C\pi = C$. Because of this, the elements of $H|_C$ can be interpreted as functions from C to C, whose function composition makes $H|_C$ a group.

Definition 33. The extension of a subgroup $S \leq G|_C$ is defined as $\mathrm{ext}_G(S) = \{\pi \in G \mid \pi|_C \in S\}$.

Using this, we can define subgroups using their support, and the structure of the group on that support:

Definition 34. For $C \subseteq \mathcal{D}$, and $S \leq G|_C$, define $[C, S]^e = \mathrm{ext}_G(S)$.

It turns out that any subgroup $H \leq G$ having a least support there is a pair (C, S) such that $H = [C, S]^e$. Since both C and S are finite, this gives a finite representation of orbits of nominal sets whose data symmetry admits least supports.

Lemma 7. *If $C \subseteq \mathcal{D}$ is a support of $H \leq G$, and $\pi \in H$, then $C\pi$ is a support of H.*

Proof. To show this, we rewrite $G_{C\pi}$:

$$\begin{aligned} G_{C\pi} &= \{\sigma \in G \mid (C\pi)\sigma = C\pi\} \\ &= \{\sigma \in G \mid C\pi\sigma\pi^{-1} = C\pi\pi^{-1}\} \\ &= \{\sigma \in G \mid C\pi\sigma\pi^{-1} = C\} \\ &= \{\sigma \in G \mid \pi\sigma\pi^{-1} \in C\} \end{aligned}$$

Thus, $\sigma \in G_{C\pi}$ iff $\pi\sigma\pi^{-1} \in G_C$. But then, since C supports H and hence $G_C \leq H$ also $\pi\sigma\pi^{-1} \in H$. Since $\pi \in H$, this gives us that $\sigma \in H$. Hence $G_{C\pi} \leq H$, which implies that $C\pi$ supports H. □

Theorem 5. *For any $H \leq G$ with finite support, there is a pair (C, S), $C \subseteq \mathcal{D}$, $S \leq G|_C$ such that $H = [C, S]^e$.*

Proof. This is proven by constructing such a pair, and then proving it satisfies the requirements.

Let C be the least support of H. This exists since H has a finite support, and we assumed that the data symmetry admits least supports. Construct $S = H|_C$. Then we can calculate $[C, S]^e$:

$$\begin{aligned}[C, S]^e &= \mathrm{ext}_G(S) \\ &= \{\pi \in G \mid \pi|_C \in S\} \\ &= \{\pi \in G \mid \exists \sigma \in H : \pi|_C = \sigma|_C \wedge C\sigma = C\}\end{aligned}$$

Since C supports H, $G_C \leq H$. In particular, this means that if $\pi \neq \sigma$ but $\pi|_C = \sigma|_C$, then there exists a $\tau \in H$ such that $\pi\tau = \sigma$. Hence, if $\sigma \in H$, then also $\pi \in H$. Using this we simplify:

$$[C, S]^e = \{\pi \in H \mid C\pi = C\}$$

This leaves one condition to remove: $C\pi = C$. However, by Lemma 7, we know that for any $\pi \in H$, $G_{C\pi} \leq H$. Hence, for any $\pi \in H$, $C\pi$ is also a support for H. Using the fact that C is a least support, this implies $C\pi \subseteq C$. Since C is of finite size, and π a bijection on $\mathcal{D}$, we get $C\pi = C$. Hence $[C, S]^e = H$. □

Corollary 2. *A finite nominal set X can be finitely represented as a union of its orbits, each individual orbit represented by a pair (C_i, S_i).*

Chapter 3

Exploiting the total order symmetry

Although the representation from Theorem 5 gives a finite representation of orbits that could in theory be used for calculations, this is rather tricky in practice. The primary difficulty is in the manipulation of the finite groups S used in the representation.

As we will see in this section, by restricting ourselves to the total order symmetry, we can significantly simplify the representation theory. In particular, the representation theory can be formulated without need for explicit calculations on groups.

The goal of this chapter is to construct an explicit representation of nominal sets over the total order symmetry, together with tools for manipulating products and equivariant functions. This is then used in implementing a library for calculations on nominal sets, and analyzing the time complexity of various operations in this library.

The work on this by the author is also in [24], in collaboration with Joshua Moerman and Jurriaan Rot, submitted (at the time of writing) to ICALP 2018.

3.1 Specializing representation theory

3.1.1 Orbits and Sets

The restriction to the total order allows a significant simplification of the representation of orbits. This simplification is based on two properties of the total order symmetry:

Lemma 8 (group triviality)**.** *For every finite subset $C \subset \mathbb{Q}$, we have $G|_C = I$, where I is the trivial group. Recall that G is the group of monotonic increasing bijections on $\mathbb{Q}$.*

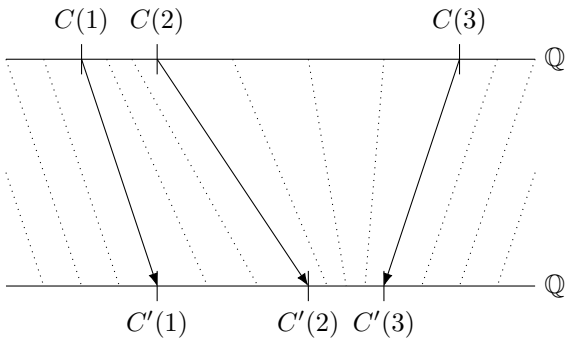

Figure 3.1: Visualization of π from Lemma 9

Proof. Let $\pi \in G|_C$ be any element of $G|_C$. If π is not the identity, then since C is finite, there exists a smallest element $c \in C$ with $c\pi \neq c$. Since π is a bijection mapping C to C, we find $c\pi\pi \neq c\pi$ and $c\pi \in C$, hence $c < c\pi$. Furthermore, there exists some $c' \in C$ with $c'\pi = c$. Since by assumption $c' \neq c$, also $c' < c$. But then both $c < c'$ and $c\pi > c = c'\pi$, contradicting monotonicity of π. Hence π is the identity element, and $G|_C = I$. □

Corollary 3. *For a single orbit set X, if $x \in X$ and $y \in X$ have the same least support, then $x = y$.*

Lemma 9 (homogeneity)**.** *For any two finite $C \subseteq \mathbb{Q}$, $C' \subseteq \mathbb{Q}$, if $\#C = \#C'$, then there is a $\pi \in G$ such that $C\pi = C'$.*

Proof. This is shown through construction of π. Number the elements of C from smallest to largest, such that $C(1)$ is the smallest element and $C(n)$ the largest. Do the same for C'. We define π such that $C(i)\pi = C'(i)$, interpolating in between (see Figure 3.1 for a visualization):

$$x < C(1) \to x\pi = x - C(1) + C'(1).$$
$$x \geq C(1) \wedge x < C(2) \to x\pi = (x - C(1))\frac{C'(2) - C'(1)}{C(2) - C(1)} + C'(1).$$
$$x \geq C(2) \wedge x < C(3) \to x\pi = (x - C(2))\frac{C'(3) - C'(2)}{C(3) - C(2)} + C'(2).$$
$$\dots$$
$$x \geq C(n) \to x\pi = x - C(n) + C'(n).$$

Note that since $\frac{C'(i)-C'(i-1)}{C(i)-C(i-1)} > 0$ for any $1 < i \leq n$, π is monotone. Further-

more, its inverse is given by:

$$\begin{aligned}
x < C'(1) &\to x\pi^{-1} = x - C'(1) + C(1).\\
x \geq C'(1) \wedge x < C'(2) &\to x\pi^{-1} = (x - C'(1))\frac{C(2) - C(1)}{C'(2) - C'(1)} + C(1).\\
x \geq C'(2) \wedge x < C'(3) &\to x\pi^{-1} = (x - C'(2))\frac{C(3) - C(2)}{C'(3) - C'(2)} + C(2).\\
&\dots\\
x \geq C'(n) &\to x\pi^{-1} = x - C'(n) + C(n).
\end{aligned}$$

Hence π is a monotones bijection, and we conclude $\pi \in G$. □

As proven in the next lemma, these properties allow for the reduction of the representation of orbits to a single integer: the size of the least support of its elements.

Lemma 10. *Given a finite subset $C \subset \mathbb{Q}$, we have $[C, I]^{ec} \cong \mathcal{P}_{\#C}(\mathbb{Q})$.*

Proof. From Lemma 9 it follows that $\mathcal{P}_{\#C}(\mathbb{Q})$ consists of a single orbit. Given this, in combination with the fact that $C \in \mathcal{P}_{\#C}(\mathbb{Q})$, Theorem 5 gives a subgroup $S \leq G|_C$ such that $\mathcal{P}_{\#C}(\mathbb{Q}) \cong [C, S]^{ec}$. Since $S \leq G|_C$, Lemma 8 implies $S = I$. This proves that $[C, I]^{ec} \cong \mathcal{P}_{\#C}(\mathbb{Q})$. □

This reduced representation can be used to formulate a direct representation of nominal sets in terms of functions from the natural numbers to themselves. Such functions can equivalently be thought of as multisets.

Definition 35. Given a function $f\colon \mathbb{N} \to \mathbb{N}$, we define a nominal set $[f]^o$ by

$$[f]^o = \bigcup_{\substack{n\in\mathbb{N}\\ 1\leq i\leq f(n)}} \{i\} \times \mathcal{P}_n(\mathbb{Q}).$$

Theorem 6. *For every orbit-finite nominal set X, there is a function $f\colon \mathbb{N} \to \mathbb{N}$ such that $X \cong [f]^o$ and the set $\{n \mid f(n) \neq 0\}$ is finite. Furthermore, the mapping between X and f is one-to-one up to isomorphism of X when limiting to $f\colon \mathbb{N} \to \mathbb{N}$ for which the set $\{n \mid f(n) \neq 0\}$ is finite.*

Proof. We start by proving the first part. For this, grade X by the dimension of its elements, defining $X_i = \{x \in X \mid \dim(x) = i\}$. Now split each X_i up into its k_i orbits $O_{i,j}$, such that $X_i = \bigcup_{1\leq j\leq k_i} O_{i,j}$.

By Theorem 5, for each orbit $O_{i,j}$, there exists a set $C_{i,j}$ and a subgroup $S_{i,j} \leq G|_{C_{i,j}}$ such that $O_{i,j} \cong [C_{i,j}, S_{i,j}]^{ec}$. By Lemma 8, $S_{i,j} = I$, and since $\#C_{i,j} = i$, Lemma 10 implies $O_{i,j} \cong \{j\} \times \mathcal{P}_i(\mathbb{Q})$.

Define $f : \mathbb{N} \to \mathbb{N}$ such that $f(i) = k_i$. Then writing out gives

$$[f]^o = \bigcup_{\substack{i\in\mathbb{N}\\ 1\leq j\leq f(i)}} \{j\} \times \mathcal{P}_i(\mathbb{Q}) \cong \bigcup_{\substack{i\in\mathbb{N}\\ 1\leq j\leq k_i}} O_{i,j} = X.$$

Next, we will show that the mapping is one-to-one up to isomorphisms. By Lemma 10 we find $\mathrm{N}(\mathcal{P}_n(\mathbb{Q})) = 1$ and $\dim(\mathcal{P}_n(\mathbb{Q})) = n$. It follows that $\mathrm{N}([f]^o) = \sum_{n \in \mathbb{N}} f(n)$. Hence any f with finite support corresponds to an orbit-finite nominal set.

Now suppose that $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$ are functions with the property $[f]^o \cong [g]^o$. Let $h : [f]^o \to [g]^o$ be the isomorphism. Grade $[f]^o$ and $[g]^o$, letting $[f]^o_i = \{x \in [f]^o \mid \dim(x) = i\}$, and similarly for $[g]^o_i$. Since h is an isomorphism, we have for any $x \in [f]^o$ that $\dim(h(x)) = \dim(x)$, implying $h([f]^o_i) = [g]^o_i$. Furthermore, the fact that h is an isomorphism gives $\mathrm{N}(h([f]^o_i)) = \mathrm{N}([f]^o_i)$. Using $\mathrm{N}([f]^o_i) = f(i)$, we find that $f(i) = \mathrm{N}([f]^o_i) = \mathrm{N}(h([f]^o_i)) = \mathrm{N}([g]^o_i) = g(i)$. Hence $f = g$.

Combining these two arguments, we find that the correspondence is one-to-one up to isomorphism. □

Note that the requirement that X is orbit finite is needed to ensure there are only a finite number of orbits of each size. As such this establishes a normal form for orbit finite nominal sets over the total order symmetry. A similar theorem for general orbit finite nominal sets (though without the property of being one-to-one) can be formulated by taking $f : \mathbb{N} \to \mathrm{Sets}$.

As an example, consider the nominal set $\mathbb{Q}^2$. This set consists of three orbits:

- The elements (a, b) with $a < b$, having dimension 2.
- The elements (a, b) with $a > b$, having dimension 2.
- The elements (a, b) with $a = b$, having dimension 1.

Hence the nominal set $\mathbb{Q}^2$ can be represented using the function $f : \mathbb{N} \to \mathbb{N}$ with $f(1) = 1$, $f(2) = 2$, and f equal to zero everywhere else. This corresponds to the multiset $\{1, 2, 2\}$.

3.1.2 Equivariant maps

Beyond just representing nominal sets, it is also useful to be able to work with combinations of nominal sets, in the form of functions and products. Both fundamentally rely on representing equivariant maps, which we will focus on first.

The representation of equivariant maps is based on two observations:

Lemma 11. *Let $f : X \to Y$ be an equivariant map, and $O \subseteq X$ a single orbit. Then $f(O) = \{f(x) \mid x \in O\}$ is a single orbit nominal set.*

Proof. Let y and y' both be elements of $f(O)$. To show that $f(O)$ is single orbit, we need to construct a $\pi \in G$ such that $y\pi = y'$. By definition of $f(O)$, there exist $x \in O$, $x' \in O$ such that $f(x) = y$ and $f(x') = y'$. Since O is single orbit, there exists a $\pi \in G$ such that $x\pi = x'$. As f is an equivariant function, we find $y\pi = f(x)\pi = f(x\pi) = f(x') = y'$. This proves that $f(O)$ is single orbit. □

Lemma 12. *Let $f\colon X \to Y$ be an equivariant map between two nominal sets X and Y. Let $x \in X$ and let C be the least support of x. Then C supports $f(x)$.*

Proof. Let $\pi \in G$ be such that $\forall c \in C, c\pi = c$. Then since C is the support of x, $x\pi = x$. But then also $f(x)\pi = f(x\pi) = f(x) = f(x)$. Hence C is a support of $f(x)$. Then, by definition, the least support of $f(x)$ is contained in C. □

These two lemmas state that an equivariant function can be thought of as mapping single orbits onto single orbits, potentially forgetting part of the support of the elements in the process. Most importantly, an equivariant map can never introduce new elements to the least support of an image compared to that of its input.

This allows us to build up a representation of equivariant functions on a per orbit basis. For each orbit in the domain, the representation contains the orbit it is mapped on, and how the least support is affected.

Definition 36. Let $H = \{(I_1, F_1, O_1), \ldots, (I_n, F_n, O_n)\}$ be a finite set of tuples where the I_i's are disjoint single orbit nominal sets, the O_i's are single orbit nominal sets with $\dim(O_i) \leq \dim(I_i)$, and the F_i's are bit strings of length $\dim(I_i)$ with exactly $\dim(O_i)$ ones.

Given a set H as above, we define $f_H : \bigcup I_i \to \bigcup O_i$ as the unique equivariant function such that, given $x \in I_i$ with least support C, $f_H(x)$ is the unique (by Corollary 3) element of O_i with support $\{C(j) \mid F_i(j) = 1\}$, where $F_i(j)$ is the j-th bit of F_i, and $C(j)$ the j-th smallest element of C.

Lemma 13. *For every equivariant map $f\colon X \to Y$ between orbit finite nominal sets X and Y there is a set H as in Definition 36 such that $f = f_H$.*

Proof. This is shown by construction. Split X into its constituent orbits, call them I_1 through I_n. For each of these, select an element $e_i \in I_i$. Let O_i be the orbit of $f(e_i)$. By Lemma 11, $f(I_i) = O_i$. For each e_i, let C_i be the least support of e_i and C_i' the least support of $f(e_i)$. Let F_i be the string with $F_i(j) = 1$ if $C_i(j) \in C_i'$, and $F_i(j) = 0$ otherwise. Let $H = \{(I_i, F_i, O_i) \mid i \in \{1, \ldots, n\}\}$. By construction, $f_H(e_i)$ is the unique element of O_i with as support $C_i' \cap C_i$. By Lemma 12, $C_i' \cap C_i = C_i'$, implying $f_H(e_i) = f(e_i)$. Since both are equivariant functions with the same domain, we have $f(x) = f_H(x)$ for all $x \in X$. Hence $f = f_H$. □

As an example, consider the projection function $\pi_1 : \mathbb{Q}^2 \to \mathbb{Q}$. As noted before, $\mathbb{Q}^2$ consists of three orbits: $O_1 = \{(a, b) \mid a, b \in \mathbb{Q}, a < b\}$, $O_2 = \{(a, b) \mid a, b \in \mathbb{Q}, a > b\}$ and $O_3 = \{(a, a) \mid a \in \mathbb{Q}\}$. For O_1, the first element is smaller than the second, so the smallest element of the least support needs to be kept. For O_2 it is the other way around, so the largest element needs to be kept. O_3 has dimension 1, the same as $\mathbb{Q}$, so the entire support is kept. This gives the following representation of π_1: $H = \{(O_1, 10, \mathbb{Q}), (O_2, 01, \mathbb{Q}), (O_3, 1, \mathbb{Q})\}$.

3.1.3 Products

The product of two orbit-finite nominal sets X and Y is again a nominal set. As we are working with the total order symmetry, $X \times Y$ will also be orbit finite. Using this, Theorem 6 gives a representation of the set $X \times Y$.

In practice, this is not sufficient to fully describe the product in applications, as it only provides a set isomorphic to $X \times Y$. Generally, one also wants to have the projection maps describing how the elements of $X \times Y$ map onto the elements of X and Y.

Thus, our representation of products consists of a representation of the set $X \times Y$, together with the projection maps $\pi_1 : X \times Y \to X$ and $\pi_2 : X \times Y \to Y$. We could represent each of these directly, using the representation of nominal sets and equivariant maps presented above.

To do this, we use the fact that the representations of $X \times Y$, π_1 and π_2 are each constructed on a per-orbit basis, all based on the orbits of $X \times Y$. By combining the representations of all three, the following observation can be used to both simplify the representation, and provide an explicit construction of the product given representations of the sets X and Y.

Lemma 14. *Let X and Y be nominal sets, and $(x, y) \in X \times Y$. If C, C_x and C_y are the least supports of (x, y), x and y respectively, then $C = C_x \cup C_y$.*

Proof. Let $\pi \in G$ be a group element such that $\forall c \in C_a \cup C_b, c\pi = c$. By definition $x = (\pi_a(x), \pi_b(x))$, and hence $x\pi = (\pi_a(x), \pi_b(x))\pi = (\pi_a(x)\pi, \pi_b(x)\pi) = (\pi_a(x), \pi_b(x))$. It follows that $C_a \cup C_b$ is a support of x, and since C is the least support of x, $C \subseteq C_a \cup C_b$.

Now suppose that C is strictly smaller than $C_a \cup C_b$. Then there is an element $c \in C_a \cup C_b$ with $c \notin C$. Without loss of generality we can assume $c \in C_a$. Since C_a is the least support of $\pi_a(x)$, there is some $\pi \in G$ such that $\forall c' \in (C_a \cup C_b) \setminus \{c\}, c'\pi = c'$, but $\pi_a(x)\pi \neq \pi_a(x)$, since C_a is not contained in $(C_a \cup C_b) \setminus \{c\}$, implying that the latter cannot be a support of $\pi_a(x)$. For this π, we have $x\pi = (\pi_a(x), \pi_b(x))\pi = (\pi_a(x)\pi, \pi_b(x)\pi) \neq (\pi_a(x), \pi_b(x)) = x$. But $(C_a \cup C_b) \setminus \{c\}$ is a support of x since C is a subset of it, leading to a contradiction.

We conclude that $C = C_a \cup C_b$. □

Note that the lemma above holds for arbitrary symmetries.

This knowledge can be used to merge the representation of the two projection maps, together with the representation of the orbit, into a single tuple:

Definition 37. Let $P \in \{A, B, C\}^*$, and $O_1 \subseteq X$, $O_2 \subseteq Y$ single orbit sets. Given a tuple (P, O_1, O_2), where the string P satisfies $|P|_A + |P|_C = \dim(O_1)$ and $|P|_B + |P|_C = \dim(O_2)$, define

$$[(P, O_1, O_2)]^t = (\mathcal{P}_{|P|}(\mathbb{Q}), f_{\{(\mathcal{P}_{|P|}(\mathbb{Q}), s_1(P), O_1)\}}, f_{\{(\mathcal{P}_{|P|}(\mathbb{Q}), s_2(P), O_1)\}}),$$

where $s_1 : \{A, B, C\}^* \to \{0, 1\}$ and $s_2 : \{A, B, C\}^* \to \{0, 1\}$ are defined with

$$\begin{aligned} s_1(\epsilon) &= \epsilon & s_2(\epsilon) &= \epsilon \\ s_1(Aw) &= 1s_1(w) & s_2(Aw) &= 0s_2(w) \\ s_1(Bw) &= 0s_1(w) & s_2(Bw) &= 1s_2(w) \\ s_1(Cw) &= 1s_1(w) & s_2(Cw) &= 1s_2(w). \end{aligned}$$

Before proving that the above is a representation, let us define some notation:

Definition 38. Given nominal sets A, B, P and Q, functions $f_1 : P \to A$, $f_2 : P \to B$, $g_1 : Q \to A$ and $g_2 : Q \to B$, we say $(P, f_1, f_2) \cong (Q, g_1, g_2)$ if and only if there exists an isomorphism $h : P \to Q$ such that $f_1 = g_1 \circ h$ and $f_2 = g_2 \circ h$.

Lemma 15. *There exists a correspondence between orbits $O \subseteq X \times Y$, and tuples (P, O_1, O_2) satisfying $O_1 \subseteq X$, $O_2 \subseteq Y$, $|P|_A + |P|_C = \dim(O_1)$ and $|P|_B + |P|_C = \dim(O_2)$, such that $[(P, O_1, O_2)]^t \cong (O, \pi_1|O, \pi_2|O)$. Furthermore, this correspondence is one-to-one.*

Proof. Let us start by constructing such a tuple (P, O_1, O_2) for a given orbit $O \subseteq X \times Y$. Since O is an orbit, Lemma 13 provides two tuples (O, F_1, O_1) and (O, F_2, O_2) such that $\pi_1|_O = f_{\{(O,F_1,O_1)\}}$ and $\pi_2|_O = f_{\{(O,F_2,O_2)\}}$.

By Lemma 10, there exists an $n \in \mathbb{N}$ such that $\mathcal{P}_n \cong O$. Let $g : O \to \mathcal{P}$ be the isomorphism between them. Since n is the dimension of O, and so are the lengths of F_1 and F_2, we find $n = |F_1| = |F_2|$. This shows that $f_{\{(O,F_1,O_1)\}} = f_{\{(\mathcal{P}_{|F_1|},F_1,O_1)\}} \circ g$ and $f_{\{(O,F_2,O_2)\}} = f_{\{(\mathcal{P}_{|F_2|},F_2,O_2)\}} \circ g$. This shows $(O, \pi_1|_O, \pi_2|_O) \cong (\mathcal{P}_{|F_1|}, f_{\{(\mathcal{P}_{|F_1|},F_1,O_1)\}}, f_{\{(\mathcal{P}_{|F_2|},F_2,O_2)\}})$.

Now construct P as follows: If the i-th letter of F_1 is 1, and the i-th letter of F_2 is 0, the i-th letter of P is A. Similarly, if the i-th letter of F_2 is 1, and the i-th letter of F_1 is 0, then the i-th letter of P is B. If the i-th letters of both F_1 and F_2 are 1, let the i-th letter of P be C. Lemma 14 guarantees that it will never be the case that the i-th letters of F_1 and F_2 are both 0. Using this construction, we find $s_1(P) = F_1$ and $s_2(P) = F_2$. Hence $(\mathcal{P}_{|F_1|}, f_{\{(\mathcal{P}_{|F_1|},F_1,O_1)\}}, f_{\{(\mathcal{P}_{|F_2|},F_2,O_2)\}}) = (\mathcal{P}_{|P|}, f_{\{(\mathcal{P}_{|P|},s_1(P),O_1)\}}, f_{\{(\mathcal{P}_{|P|},s_2(P),O_2)\}}) = [(P, O_1, O_2)]^t$.

It remains necessary to show that P satisfies $|P|_A + |P|_C = \dim(O_1)$ and $|P|_B + |P|_C = \dim(O_2)$. To show this, consider F_1. By Definition 36, each 1 in the string F_1 corresponds to a unique element in the least support of an element of O_1. Hence $\dim(O_1) = |F_1|_1$. Since $F_1 = s_1(P)$, and $s_1(P)$ replaces each A and C with 1s, and every B with a 0, we find $\dim(O_1) = |F_1|_1 = |P|_A + |P|_C$. A similar line of reasoning, replacing A with B, shows $\dim(O_2) = |F_2|_1 = |P|_B + |P|_C$.

Having shown that every orbit corresponds to a tuple with the given properties, let us now prove the converse. Take a tuple (P, O_1, O_2) satisfying the conditions of this lemma. Use this to construct $(O', f, g) = [(P, O_1, O_2)]^t$. By construction, we find $f(O') \subseteq X$ and $g(O') \subseteq Y$. Hence, given an $x \in O'$, $(f(x), g(x)) \in X \times Y$. More generally, $f \times g$ is a map from O to $X \times Y$. By Lemma 11, since O is single orbit, so is $f \times g(O)$.

Furthermore, by construction of f and g, we find that if C is the least support of $x \in O'$, then c is also the least support of $(f(x), g(x))$, since every element in the support of x is in at least one of the least supports of $f(x)$ and $g(x)$, and by Lemma 14, the least support of $(f(x), g(x))$ is the union of the least supports of $f(x)$ and $g(x)$. This implies that the elements of both O' and $f \times g(O')$ have the same support size, and as both O' and $f \times g(O')$ are single orbit, this makes $f \times g$ a bijection, showing that $[(P, O_1, O_2)]^t = (O', f, g) \cong (f \times g(O'), \pi_1|_{f\times g(O')}, \pi_2|_{f\times g(O')})$.

Finally, to show that the correspondence is one-to-one, consider two tuples (P, O_1, O_2) and (P', O_1', O_2'). For $[(P, O_1, O_2)]^t \cong [(P', O_1', O_2')]^t$ to hold, we need $\mathcal{P}_{|P|} \cong \mathcal{P}_{|P'|}$, hence $|P| = |P'|$. Furthermore, for any $x \in \mathcal{P}$, there needs to exists an $x' \in \mathcal{P}'$ such that $f_{\{(\mathcal{P}_{|P|}, s_1(P), O_1)\}}(x) = f_{\{(\mathcal{P}_{|P'|}, s_1(P'), O_1')\}}(x')$ and $f_{\{(\mathcal{P}_{|P|}, s_2(P), O_2)\}}(x) = f_{\{(\mathcal{P}_{|P'|}, s_2(P'), O_2')\}}(x')$. Since O_1, O_2, O_1' and O_2' single orbit, this implies $O_1 = O_1'$ and $O_2 = O_2'$. Furthermore, since by the reasoning above, the least support of $(f_{\{(\mathcal{P}_{|P|}, s_1(P), O_1)\}}(x), f_{\{(\mathcal{P}_{|P|}, s_2(P), O_2)\}}(x)) = (f_{\{(\mathcal{P}_{|P'|}, s_1(P'), O_1)\}}(x'), f_{\{(\mathcal{P}_{|P'|}, s_2(P'), O_2)\}}(x'))$ equals the least support of x and x', we have $x = x'$. But this implies that $f_{\{(\mathcal{P}_{|P|}, s_1(P), O_1)\}} = f_{\{(\mathcal{P}_{|P'|}, s_1(P'), O_1)\}}$, and $f_{\{(\mathcal{P}_{|P|}, s_2(P), O_2)\}} = f_{\{(\mathcal{P}_{|P'|}, s_2(P'), O_2)\}}$, meaning that $s_1(P) = s_1(P')$ and $s_2(P) = s_2(P')$. But P and P' can have equal images under both s_1 and s_2 only if P and P' are equal. From this, we conclude that $[(P, O_1, O_2)]^t \cong [(P', O_1', O_2')]^t$ if and only if $(P, O_1, O_2) = (P', O_1', O_2')$. □

From the lemma above it follows that we can generate the product $X \times Y$ simply by generating every valid string P for any pair of orbits (O_1, O_2) of X and Y. We use the representation of Theorem 6 to give an explicit description of $X \times Y$.

Theorem 7. *For $X \cong [f]^o$ and $Y \cong [g]^o$ we have $X \times Y \cong [h]^o$, where*

$$h(n) = \sum_{\substack{0 \le i,j \le n \\ i+j \ge n}} f(i)g(j)\binom{n}{j}\binom{j}{n-i}.$$

Proof. Every string $P \in \{A, B, C\}^*$ of length n with $|P|_A = n - j$, $|P|_B = n - i$ and $|P|_C = i + j - n$ satisfies the requirements of Lemma 15, and hence describes a unique orbit for every pair of orbits O_1 and O_2 where the least support of the elements of O_1 has size i, and the least support of elements of O_2 have size j. Combinatorics tells us that there are $\binom{n}{j}\binom{j}{n-i}$ such strings. Summing over all $i \ge 0$, $j \ge 0$ such that $i + j - n$, $n - j$ and $n - i$ are positive, and multiplying with the number of orbits of the required size gives the result. □

Reexamining the set $\mathbb{Q}^2$ from before, enumerating the product strings give the following:

- $(AB, \mathbb{Q}, \mathbb{Q})$, for the orbit $O_1 = \{(a, b) \mid a \in \mathbb{Q} \wedge b \in \mathbb{Q} \wedge a < b\}$.
- $(BA, \mathbb{Q}, \mathbb{Q})$, for the orbit $O_2 = \{(a, b) \mid a \in \mathbb{Q} \wedge b \in \mathbb{Q}, a > b\}$.

- $(C, \mathbb{Q}, \mathbb{Q})$, for the orbit $O_3 = \{(a, a) \mid a \in \mathbb{Q}\}$.

For a more complex example, let us consider the product $\mathbb{Q} \times O_1 = \mathbb{Q} \times \{(a, b) \in \mathbb{Q}^2 \mid a < b\}$. Both sets are single orbit, and for those orbit, we find the following:

- $(ABB, \mathbb{Q}, O_1)$, for the orbit $\{(a, (b, c)) \mid a, b, c \in \mathbb{Q}, a < b < c\}$.
- $(BAB, \mathbb{Q}, O_1)$, for the orbit $\{(b, (a, c)) \mid a, b, c \in \mathbb{Q}, a < b < c\}$.
- $(BBA, \mathbb{Q}, O_1)$, for the orbit $\{(c, (a, b)) \mid a, b, c \in \mathbb{Q}, a < b < c\}$.
- $(CB, \mathbb{Q}, O_1)$, for the orbit $\{(a, (a, b)) \mid a, b \in \mathbb{Q}, a < b\}$.
- $(BC, \mathbb{Q}, O_1)$, for the orbit $\{(b, (a, b)) \mid a, b \in \mathbb{Q}, a < b\}$.

To illustrate how the projection maps are constructed, let us work this out for $(BC, \mathbb{Q}, O_1)$, corresponding to the orbit $\{(b, (a, b)) \mid a, b \in \mathbb{Q}, a < b\}$. Calculating, we find $s_1(BC) = 01$ and $s_2(BC) = 11$. Hence the first projection map $\pi_1 : \mathcal{P}_2(\mathbb{Q}) \to \mathbb{Q}$ is defined by $\pi_1(\{a, b\}) = \max(a, b)$, which indeed corresponds to the first element of the tuple $(b, (a, b))$, as $a < b$. For $\pi_2 : \mathcal{P}_2(\mathbb{Q}) \to \mathbb{Q}$, we find that $\pi_2(\{a, b\})$ is the element of O_1 with $\{a, b\}$ as support. Assuming that $a < b$, this is the element (a, b), matching the second element of the tuple.

3.2 Calculating with nominal sets

The ideas presented above were used to implement the library Ons for calculating with nominal sets in C++, which can be found at `https://github.com/davidv1992/ONS`. The goal of the library was to provide all the basic set operations (union, intersection, products, membership and subset checking). Filtering with equivariant functions, and calculating the image under an equivariant function are also included. Ons also allows programs to directly access the orbit structure of it's individual set for more complicated manipulations.

The following is a simple example of a program written using Ons. It calculates the product $\mathbb{Q} \times \{(a, b) \in \mathbb{Q}^2 \mid a < b\}$ from before, and prints a representative element for each of the orbits in the result:

```
nomset<rational> A = nomset_rationals();
nomset<pair<rational, rational>> B({rational(1),rational(2)});

auto AtimesB = nomset_product(A, B);   // compute the product
for (auto orbit : AtimesB)
    cout << orbit.getElement() << endl;
```

Running this gives the following output:

```
(1/1,(2/1,3/1))
(1/1,(1/1,2/1))
(2/1,(1/1,3/1))
(2/1,(1/1,2/1))
(3/1,(1/1,2/1))
```

This example illustrates the three main methods for constructing a nominal set:

- By using a predefined set, in this case the rationals through the use of `nomset_rationals`.
- By specifying elements it should contain, in this case the smallest set containing the pair $(1, 2)$.
- As the result of a set calculation, in this case products.

The code above prints one element from each of the orbits in the result. The bare representation from Theorem 6 only allows the reconstruction of a set that is isomorphic to the original. On its own, this is not enough information to construct elements of a nominal set. To remedy this, Ons stores a description of how the elements of an orbit can be constructed from the least support alongside the integer representing the orbit. The exact details of how this is done are type-dependent, but for product types, such as pair, the representation of products described above is used.

The combination of orbit size and how to construct elements is stored in the `orbit` class. The `nomset` class then stores a collection of these objects, in a tree-based set datastructure. This allows a fast lookup of whether an orbit is contained in a set, as well as fast manipulation.

Another example program is given below: It calculates the set of elements $(a, b) \in \mathbb{Q}^2$ that satisfy both $a \geq b$ and $a \leq b$:

```
// Create the universe of pairs
nomset<pair<rational,rational> > U =
    nomset_product(nomset_rationals(), nomset_rationals());

// Filter out those pairs where a<= b, and those where a>= b
nomset<pair<rational,rational> > A =
    nomset_filter(U, [](pair<rational,rational> p){
            return p.first <= p.second;
        });
nomset<pair<rational,rational> > B =
    nomset_filter(U, [](pair<rational,rational> p){
            return p.first >= p.second;
        });

// And calculate the intersection
nomset<pair<rational,rational> > I = nomset_intersect(A,B);

for (auto orbit : I)
    cout << orbit.getElement() << endl;
```

Running this gives the following output:

```
(1/1,1/1)
```

The `eqimap` class implements the representation of equivariant functions presented above. It allows the manipulation of equivariant functions, and the

construction of completely new functions at runtime. Such functions can be defined in three ways:

- By providing an implementation of the desired function in C++, and converting it to an `eqimap`.
- By specifying its value on a number of inputs, such that at least one input is specified for each of the orbits in the domain.
- By directly manipulating the representation presented above.

An example of an application of `eqimap` is to represent the transition function of nominal automata. It will allow us to read automata from text files, which is used in Chapter 4. The code for this is rather large, so we refer the interested reader to the Ons repositories.

3.2.1 Complexity analysis

As Ons is implemented using an explicit representation of orbits, it becomes possible to determine the complexity of the basic set operations. To simplify this analysis, let us make the following assumptions:

- The comparison of two orbits take $O(1)$.
- Constructing an orbit from an element takes $O(1)$.
- Checking whether an element is in an orbit takes $O(1)$.

In practice, each of these operations needs to look at the entirety of the data stored for the orbit, which is dependent on the size of the orbit itself. However, these are typically small and approximately constant.

Theorem 8. *The complexity of the following set operations as implemented in* Ons *is as follows:*

Operation	*Complexity*
Test $x \in X$	$O(\log \mathrm{N}(X))$
Test $X \subseteq Y$	$O(\min(\mathrm{N}(X) + \mathrm{N}(Y), \mathrm{N}(X) \log \mathrm{N}(Y)))$
Calculate $X \cup Y$	$O(\mathrm{N}(X) + \mathrm{N}(Y))$
Calculate $X \cap Y$	$O(\mathrm{N}(X) + \mathrm{N}(Y))$
Calculate $\{x \in X \mid p(x)\}$	$O(\mathrm{N}(X))$
Calculate $\{f(x) \mid x \in X\}$	$O(\mathrm{N}(X) \log \mathrm{N}(X))$
Calculate $X \times Y$	$O(\mathrm{N}(X \times Y)) \subseteq O(3^{\dim(X)+\dim(Y)} \mathrm{N}(X) \mathrm{N}(Y))$

Proof. Each of the statements will be proven individually:

Membership. To decide $x \in X$, we first construct the orbit containing x, which is done in constant time. Then we use a logarithmic lookup to decide whether this orbit is in our set data structure. Hence, membership checking is $O(\log(\mathrm{N}(X)))$.

Inclusion. Similarly, checking whether a nominal set X is a subset of a nominal set Y can be done in $O(N(X) \log(N(Y)))$ time. However, it is also possible

to do a simultaneous in-order walk of both sets, which takes $O(N(X) + N(Y))$ time. The implementation uses a cutoff on the size of X relative to Y to deal with this, giving a time complexity of $O(min(N(X)+N(Y), N(X)\log(N(Y))))$.

Union and Intersection. This idea of a simultaneous walk through both sets X and Y is also useful for computing their union and intersection. This gives a complexity of $O(N(X) + N(Y))$ for intersections and unions.

Filtering. Filtering a nominal set X using some equivariant function f mapping it to the (trivially) nominal set $\{true, false\}$ can be done in linear time, as the results are obtained in order, giving a complexity of $O(N(X))$, assuming the time complexity of the function to be constant.

Mapping. Mapping is a bit different. The original set can still be processed in order, but the results will, in general, be out of order. Hence, for a tree-based implementation of sets, a sorting step is needed (or equivalently, iterated insertion needs to be done), which brings the complexity of mapping to $O(N(X)\log(N(X)))$, again assuming the time complexity of the function to be constant.

Products. Calculating the product of two nominal sets is the most complicated construction. For each pair of orbits in the original sets X and Y, all product orbits need to be generated. Each product orbit itself is constructed in constant time. By ordering the generation of these orbits such that they are generated in-order, the resulting set takes $O(\mathrm{N}(X \times Y))$ time to construct.

We can also give an explicit upper bound for the number of orbits in terms of the input. For this we recall that orbits in a product are represented by strings of length at most $\dim(X) + \dim(Y)$. (If the string is shorter, we can pad it with one of the symbols.) Since there are three symbols ('A', 'B' and 'C'), the product of X and Y will have at most $3^{\dim(X)+\dim(Y)}\,\mathrm{N}(X)\,\mathrm{N}(Y)$ orbits. It then follows that taking products has time complexity of $O(3^{\dim(X)+\dim(Y)}\,\mathrm{N}(X)\,\mathrm{N}(Y))$. □

Chapter 4

Nominal Automata and Applications

In the previous chapter we explored the representation theory of nominal sets over the total order symmetry. This representation theory was used to build Ons, an implementation of a library for calculating with nominal sets over the total order symmetry. As mentioned in the introduction, Ons is not the only library for working with nominal sets. Both Lois and Nλ provide implementations for calculating with nominal sets over the total order symmetry.

Unlike Ons, Lois and Nλ are not directly based on representation theory, instead using logical formulas to represent nominal sets, combined with an SMT solver to compute with these formulas. This approach allows them to make use of symmetries and patterns in these sets beyond the nominal structure. Furthermore, the approach is very flexible, and allows for a relatively straightforward implementation effort. A result of this is that Nλ and Lois support several technical features not yet available in Ons, such as sets of sets. However, the use of SMT solvers makes it hard to derive complexity results for operations, and an analog of Theorem 8 for Lois or Nλ is not known to the author.

Given the existence of this alternative approach for computing with nominal sets over the total order symmetry, an interesting question is how the performance of Ons compares. In this chapter, the theory of nominal automata, and how to minimize them, is worked out. The minimization algorithm found is then used to compare the performance of Ons to Lois and Nλ.

A second comparison, using the problem of learning nominal automata (first addressed in [17]) is also presented. The theory behind learning is beyond the scope of this text, but the results provide an interesting example of a practical application of Ons.

4.1 Nominal automata theory

Let us start by recapping the definition of a nominal automaton. In Chapter 2 we defined:

Definition 39. A *deterministic nominal automaton* over a finite nominal alphabet A is a tuple (Q, F, δ), where Q is a finite nominal set of states, $F \subseteq Q$ an equivariant subset of accepted states, and $\delta : Q \times A \to Q$ an equivariant function acting as the transition function.

In Figure 2.1 we saw an example of a nominal automaton over the equality symmetry. Let us consider another example over the total order symmetry:

Example 5. The automaton in Figure 4.1 accepts the language $\mathcal{L}_{\max} = \{wa \mid w \in \mathbb{Q}^*, a \in \mathbb{Q}, \max(w) = a\} \subset \mathbb{Q}^*$, where $\max(w)$ is the largest letter in the word w.

It can be formalized as a tuple (Q, F, δ) by taking

$$\begin{aligned} Q &= \{q_0\} \cup \{q_1(a) \mid a \in \mathbb{Q}\} \cup \{q_2(a,a) \mid a \in \mathbb{Q}\} \\ &\quad \cup \{q_3(a,b) \mid a \in \mathbb{Q} \wedge b \in \mathbb{Q} \wedge a < b\} \cup \{q_4(a,b) \mid a \in \mathbb{Q} \wedge b \in \mathbb{Q} \wedge a > b\} \\ F &= \{q_2(a,a) \mid a \in \mathbb{Q}\} \end{aligned}$$

and defining $\delta : Q \times \mathbb{Q} \to Q$ as follows:

$$\delta(q_0, a) = q_1(a)$$

$$\delta(q_1(a), b) = \begin{cases} q_3(a,b) & a < b \\ q_2(a,a) & a = b \\ q_4(a,b) & a > b \end{cases}$$

$$\delta(q_2(a,a), b) = \begin{cases} q_3(a,b) & a < b \\ q_2(a,a) & a = b \\ q_4(a,b) & a > b \end{cases}$$

$$\delta(q_3(a,b), c) = \begin{cases} q_3(b,c) & b < c \\ q_2(b,b) & b = c \\ q_4(b,c) & b > c \end{cases}$$

$$\delta(q_4(a,b), c) = \begin{cases} q_3(a,c) & a < c \\ q_2(a,a) & a = c \\ q_4(a,c) & a > c \end{cases}$$

Like in the case of classical deterministic finite automata, the transition function δ can be extended to a function $\delta^* : Q \times \mathbb{Q}^*$:

$$\begin{aligned} \delta^*(q, \epsilon) &= q \\ \delta^*(q, aw) &= \delta^*(\delta(q, a), w). \end{aligned}$$

This extends the definition of $\mathcal{L}_q = \{w \in A^* \mid \delta^*(q, w) \in F\}$ to nominal automata. In the example above, we find $\mathcal{L}_{q_0} = \mathcal{L}_{\max}$.

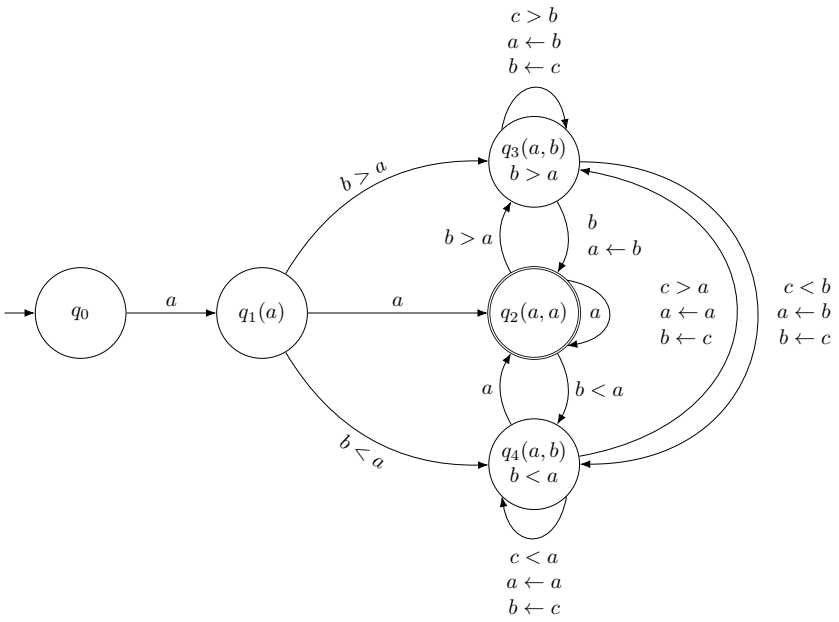

Figure 4.1: Example automaton that accepts the language $\mathcal{L}_{\max}$ when starting at q_0.

Lemma 16. *The function $\delta^* : Q \times A^* \to Q$ is equivariant, and so is the function $\mathcal{L}_{(\_)} : Q \to \mathcal{P}(A^*)$.*

Proof. This is shown by writing out the definitions of both. Start by considering δ^*. Let $\pi \in G$, then for any $q \in Q$, $a \in A$ and $w \in A^*$ it follows using induction that:

$$\begin{aligned}\delta^*(q,\epsilon)\pi &= q\pi = \delta(q\pi,\epsilon)\\ \delta^*(q,aw)\pi &= \delta^*(\delta(q,a),w)\pi = \delta^*(\delta(q,a)\pi, w\pi)\\ &= \delta^*(\delta(q\pi,a\pi),w\pi) = \delta^*(q\pi,(a\pi)(w\pi)) = \delta^*(q\pi,(aw)\pi).\end{aligned}$$

Hence, δ^* is equivariant.

Using equivariance of δ^*, proving the second part becomes straightforward: Let $q \in Q$ and $\pi \in G$. Then

$$\begin{aligned}\mathcal{L}_{q\pi} &= \{w \mid w \in A^* \wedge \delta^*(q\pi, w) \in F\} = \{w\pi \mid w \in A^* \wedge \delta^*(q\pi, w\pi) \in F\}\\ &= \{w\pi \mid w \in A^* \wedge \delta^*(q,w)\pi \in F\} = \{w\pi \mid w \in A^* \wedge \delta^*(q,w) \in F\}\\ &= \{w \mid w \in A^* \wedge \delta^*(q,w) \in F\}\pi = \mathcal{L}_q\pi.\end{aligned}$$

□

We now once again examine the question of minimal automata. The definition here will parallel that given in Chapter 1 for classical automata, and the

proofs are very similar. A different approach, based on formulating a Myhill-Nerode theorem, can be found in [3]. Both approaches lead to equivalent notions of minimal automata, and the proofs use similar ideas.

The parallels between the proofs here and those in Chapter 1 can be made more explicit using notions from category theory and coalgebra. As mentioned in Chapter 1, the choice was made not to use this to reduce the number of prerequisites needed.

Definition 40. Given two nominal automata (Q, F, δ) and (Q', F', δ'), a *morphism* is an equivariant function $f : Q \to Q'$ such that $q \in F \Leftrightarrow f(q) \in F'$ and $f(\delta(q, a)) = \delta'(f(q), a)$ for any $q \in Q$, $a \in A$. A morphism is an *isomorphism* if it is bijective.

Lemma 17. *A morphism of nominal automata is* language preserving*, in the sense that given a morphism f from (Q, F, δ) to (Q', F', δ'), we have $\mathcal{L}_q = \mathcal{L}_{f(q)}$.*

Proof. This can be shown using induction on words. Starting with the empty word ϵ:

$$\epsilon \in \mathcal{L}_q \Leftrightarrow q \in F \Leftrightarrow f(q) \in F' \Leftrightarrow \epsilon \in \mathcal{L}_{f(q)}.$$

Next, let us assume that for fixed w, for any state q', $w \in \mathcal{L}_{q'} \Leftrightarrow \mathcal{L}_{f(q')}$. Given a letter $a \in A$, it then holds that

$$aw \in \mathcal{L}_q \Leftrightarrow w \in \mathcal{L}_{\delta(q,a)} \Leftrightarrow w \in \mathcal{L}_{f(\delta(q,a))} \Leftrightarrow w \in \mathcal{L}_{\delta'(f(q),a)} \Leftrightarrow aw \in \mathcal{L}_{f(q)}.$$

By induction, it follows that for any word w, $w \in \mathcal{L}_q \Leftrightarrow w \in \mathcal{L}_{f(q)}$, and hence $\mathcal{L}_q = \mathcal{L}_{f(q)}$. □

Theorem 9. *The following are equivalent:*

1. *The nominal automaton (Q, F, δ) is minimal.*
2. *For any two states $q, q' \in Q$, with $q \neq q'$, $\mathcal{L}_q \neq \mathcal{L}_{q'}$.*

Proof. This is easiest to prove by constructing an explicit minimal automaton. Let $\bar{Q} = \{\mathcal{L}_q \mid q \in Q\}$, and $\bar{F} = \{\mathcal{L}_q \mid q \in F\}$. Construct $\bar{\delta}$ using

$$\bar{\delta}(\mathcal{L}_q, a) = \{w \mid aw \in \mathcal{L}_q\}.$$

To show that $\bar{\delta}(\mathcal{L}_q, a) \in \bar{Q}$, observe that $aw \in \mathcal{L}_q$ if and only if $w \in \mathcal{L}_{\delta(q,a)}$. Observe that, by construction, $(\bar{Q}, \bar{F}, \bar{\delta})$ recognizes the same collection of languages as (Q, F, δ). Furthermore, since the function $q \mapsto \mathcal{L}_q$ is equivariant, $\bar{Q}$ is orbit finite, and $\bar{F}$ is an equivariant subset. A straightforward calculation, similar to that used to prove Lemma 16 shows that $\bar{\delta}$ is an equivariant map.

Let us use this automaton to prove (1) ⇒ (2): Since (Q, F, δ) is minimal, there is a surjective morphism $f : \bar{Q} \to Q$. Now let $q, q' \in Q$. Since f is a surjection, there exist $\bar{q}, \bar{q}' \in \bar{Q}$ such that $f(\bar{q}) = q$ and $f(\bar{q}') = q'$. Suppose $\mathcal{L}_q = \mathcal{L}_{q'}$, then $\mathcal{L}_{\bar{q}} = \mathcal{L}_{f(\bar{q})} = \mathcal{L}_q = \mathcal{L}_{q'} = \mathcal{L}_{f(\bar{q}')} = \mathcal{L}_{\bar{q}'}$. By construction of $\bar{Q}$, this implies $\bar{q} = \bar{q}'$, and hence $q = q'$, proving (2).

For (2) $\Rightarrow$ (1), consider the function $f : Q \to \bar{Q}$ given by $f(q) = \mathcal{L}_q$. By (2), this is a bijection, and by construction of $(\bar{Q}, \bar{F}, \bar{\delta})$ it is a morphism. It follows that f is an isomorphism between (Q, F, δ) and $(\bar{Q}, \bar{F}, \bar{\delta})$. It is thus enough to show that $(\bar{Q}, \bar{F}, \bar{\delta})$ is minimal.

For this, consider an automaton (Q', F', δ') recognizing the same set of languages. Construct $g : Q' \to \bar{Q}$ such that $g(q) = \mathcal{L}_q$. Since $aw \in \mathcal{L}_q$ if and only if $w \in \mathcal{L}_{\delta'(q,a)}$, this is a morphism. Since both (Q', F', δ') and $(\bar{Q}, \bar{F}, \bar{\delta})$ recognize the same set of languages, which by construction is precisely $\bar{Q}$, f is necessarily surjective. This proves that $(\bar{Q}, \bar{F}, \bar{\delta})$ is minimal. □

Corollary 4. *For each nominal automaton, there exists a unique (up to isomorphism) minimal nominal automaton.*

As an illustration of applications of the previous theorem, let us take another look at the automaton of Example 5. Calculating the language associated with each of the states we find:

$$\begin{aligned}
\mathcal{L}_{q_0} &= \{w \mid w \text{ starts and ends with the same letter}\} \\
\mathcal{L}_{q_1(a)} &= \{w \mid w \text{ ends with an } a\} \\
\mathcal{L}_{q_2(a,a)} &= \{w \mid w \text{ ends with an } a\} \cup \{\epsilon\} \\
\mathcal{L}_{q_3(a,b)} &= \{w \mid w \text{ ends with an } a\} \\
\mathcal{L}_{q_4(a,b)} &= \{w \mid w \text{ ends with an } a\}
\end{aligned}$$

Since the languages of the states $q_2(a, a)$, $q_3(a, b)$ and $q_4(a, b)$ are the same for any choice of $a, b \in \mathbb{Q}$, this automaton is not minimal. Compare this to the automaton in Figure 4.2. Computing the language of that automaton we find:

$$\begin{aligned}
\mathcal{L}_{q'_0} &= \{w \mid w \text{ starts and ends with the same letter}\} \\
\mathcal{L}_{q'_1(a)} &= \{w \mid w \text{ ends with an } a\} \\
\mathcal{L}_{q'_2(a,a)} &= \{w \mid w \text{ ends with an } a\} \cup \{\epsilon\}
\end{aligned}$$

All the languages are distinct, hence this is a minimal automaton. Furthermore it recognizes the same set of languages as the automaton from Example 5. This implies that there exists a surjective morphism f from the automaton in Example 5 to this one. Indeed, the function f given by

$$\begin{aligned}
f(q_0) &= q'_0 \\
f(q_1(a)) &= q'_1(a) \\
f(q_2(a, a)) &= q'_2(a) \\
f(q_3(a, b)) &= q'_1(a) \\
f(q_4(a, b)) &= q'_1(a)
\end{aligned}$$

is a surjective morphism.

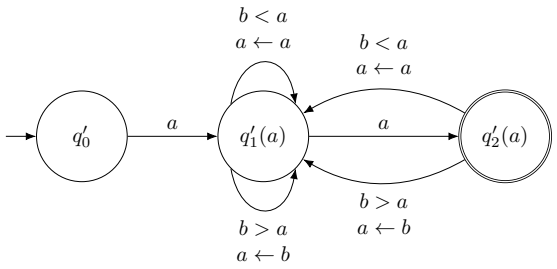

Figure 4.2: A minimal automaton for recognizing $\mathcal{L}_{\max}$.

4.1.1 Moore's algorithm for nominal automata

The characterization of minimal nominal automata in terms of automata where all states recognize distinct languages allows us to directly generalize Moore's algorithm as formulated in Chapter 1. Similar to the classical case, given a nominal automaton (Q, F, δ), the proof of Theorem 9 states that the minimal automaton can be found by determining the language equivalence $\equiv_L$, using it to compute the quotient automaton (Q', F', δ') with:

$$\begin{aligned} Q' &= Q/\equiv_L \\ F' &= F/\equiv_F \\ \delta'(S, a) &= \{\delta(q, a) \mid q \in S\} \end{aligned}$$

The equivalence relation $\equiv_L$ can once again be found by approximating it:

Definition 41. Define the relations $\equiv_i \subseteq Q \times Q$, with $i \in \mathbb{N}$ as follows:

$$\begin{aligned} q \equiv_0 q' &\Leftrightarrow (q \in F \Leftrightarrow q' \in F) \\ q \equiv_{i+1} q' &\Leftrightarrow (q \equiv_i q' \wedge \forall a \in A : \delta(q, a) \equiv_i \delta(q', a)) \end{aligned}$$

Lemma 18. *For any two states $q, q' \in Q$, we have $q \equiv_i q'$ if and only if, for all words w of length at most i, $\delta^*(q, w) \in F \Leftrightarrow \delta^*(q', w) \in F$. In other words, $q \equiv_i q'$ if and only if q and q' recognize the same words up to length i.*

The proof of the above lemma is the same as that of Lemma 2, and won't be repeated here. Using $\equiv_i$, we formulate Moore's algorithm for nominal automata:

Theorem 10. *Algorithm 5 produces a minimal automaton recognizing the same languages as its input. Furthermore, its time complexity, when implemented using the* Ons *library, is $O(3^{5k} k \,\mathrm{N}(Q)^3 \,\mathrm{N}(A))$, where $k = \dim(Q)$, and A is the alphabet.*

Proof. We will start by showing that if Algorithm 5 produces a result, that result is correct. Suppose the algorithm terminates. Then i is such that $\equiv_i = \equiv_{i-1}$. By definition of $\equiv_i$ it then follows that, for any $k \in \mathbb{N}$, $\equiv_{i+k} = \equiv_i$. This

Algorithm 5 Moore's minimization algorithm for nominal DFAs

Require: Automaton (Q, F, δ).
1: $i \leftarrow 0$, $\equiv_{-1} \leftarrow Q \times Q$.
2: $\equiv_0 \leftarrow F \times F \cup (Q \backslash F) \times (Q \backslash F)$.
3: **while** $\equiv_i \neq \equiv_{i-1}$ **do**
4: $\quad \equiv_{i+1} = \{(q_1, q_2) \mid (q_1, q_2) \in \equiv_i \wedge \forall a \in A, (\delta(q_1, a), \delta(q_2, a)) \in \equiv_i\}$.
5: $\quad i \leftarrow i + 1$.
6: **end while**
7: $E \leftarrow Q/_{\equiv_i}$.
8: $F_E \leftarrow \{e \in E \mid \forall s \in e, s \in F\}$.
9: Let δ_E be the map with $\delta_E(e, a) = \{\delta(q, a) \mid q \in e\}$.
10: **return** (E, F_E, δ_E).

implies that $\equiv_i = \equiv_L$. But then the output is the quotient automaton of the input automaton with respect to $\equiv_L$, which is minimal and recognizes the same languages.

The complexity is proven using the complexity of set operations derived in Theorem 8. The most expensive step is the calculation of $\equiv_{i+1}$ on line 4. Assuming that the dimension of Q and A are at most k, computing $Q \times Q \times A$ takes $O(\mathrm{N}(Q)^2\,\mathrm{N}(A)3^{5k}$. Filtering $Q \times Q$ using that then takes $O(\mathrm{N}(Q)^2 3^{2k})$. The time to compute $Q \times Q \times A$ dominates, hence line 4 takes $O(\mathrm{N}(Q)^2\,\mathrm{N}(A)3^{5k})$.

This leaves the question as to how many times the loop runs. Each iteration of the loop gives rise to a new partition, which is a refinement of the previous partition. Furthermore, every partition generated is equivariant. Note that this implies that each refinement of the partition does at least one of two things: Distinguish between two orbits of Q previously in the same element(s) of the partition, or distinguish between two members of the same orbit previously in the same element of the partition. The first can happen only $\mathrm{N}(Q) - 1$ times, as after that there are no more orbits lumped together. The second can only happen $\dim(S)$ times per orbit, because each such a distinction between elements is based on splitting on the value of one of the elements of the support. Hence, after $\dim(S)$ times on a single orbit, all elements of the support are used up. Combining this, the longest chain of partitions of Q has length at most $O(k\,\mathrm{N}(S))$.

Since each partition generated in the loop is unique, the loop cannot run for more iterations than the length of the longest chain of partitions on Q. It follows that the complexity of Moore's algorithm on a nominal automaton is $O(k\,\mathrm{N}(Q)^3\,\mathrm{N}(A)3^{5k})$. □

Theorem 10 gives an upper bound to the running time of Moore's algorithm, but unlike the classical case, it is not immediately clear how tight this bound is. In particular, it is unclear whether there are cases where the loop on lines 3-6 will iterate over all partitions in a chain. It turns out that automata showing such behaviour do indeed exist:

Theorem 11. *Let Q be a orbit finite nominal set, $F \subsetneq Q$ a proper equivariant subset, and let $\mathcal{P}_n \to \ldots \to \mathcal{P}_0$ be a chain of partitions with $\mathcal{P}_0 = \{F, Q \backslash F\}$, where $\mathcal{P}_i$ refines $\mathcal{P}_{i-1}$ for all $1 \leq i \leq n$, and with each $\mathcal{P}_i$ distinct. Then there exists an alphabet A, and an automaton (Q, F, δ) over A such that the partitions produced by running Moore's algorithm on that automaton are precisely those of the chain.*

Proof. This is shown by construction. The main idea is to construct an alphabet based on the sequence of partitions desired. This alphabet and the transition function are set up such that each set of letters is responsible for precisely one of the partition refinements.

First, let us introduce some notation. We define $\mathcal{P}_{-1} = \{Q\}$. Furthermore, let $\mathcal{P}_i(q)$ be the element of the partition containing q. In other words, $\mathcal{P}_i(q)$ is the unique $P \in \mathcal{P}_i$ such that $q \in P$ (non uniqueness is in contradiction with the fact that the elements of the partition are pairwise disjoint).

We construct the alphabet in parts. Let $A_i = \{(i, P, q, q') | (P, q, q') \in \mathcal{P}_i \times Q \times Q \wedge \mathcal{P}_{i-2}(q) = \mathcal{P}_{i-2}(q')\}$, for $1 \leq i \leq n$. Since both Q and $\mathcal{P}_i$ are orbit finite, so is A_i. Let the alphabet $A = \bigcup_{1 \leq i \leq n} A_i$. Since A is the union of finitely many orbit finite sets, A is also orbit finite.

Using this alphabet, let $\delta : Q \times A \to Q$ be defined such that

$$\delta(q'', (i, P, q, q')) = \begin{cases} q & \mathcal{P}_i(q'') = P \\ q' & \mathcal{P}_i(q'') \neq P \end{cases}.$$

We can now make two observations on the behaviour of δ: Let $i, j \in \mathbb{N}$, and $q, q' \in Q$ be such that $\mathcal{P}_{j-2}(q) = \mathcal{P}_{j-2}(q')$, and $q'', q''' \in Q$. Then when

1. $0 \leq i \leq j - 2 \leq n$, $\mathcal{P}_i(\delta(q'', (j, P, q, q'))) = \mathcal{P}_i(\delta(q''', (j, P, q, q')))$.
2. $0 \leq j \leq i \leq n$, $\mathcal{P}_i(q'') = \mathcal{P}_i(q''') \Rightarrow \delta(q'', (j, P, q, q')) = \delta(q''', (j, P, q, q'))$.

To show (1), let $q, q' \in Q$ be such that $\mathcal{P}_{j-2}(q) = \mathcal{P}_{j-2}(q')$. By definition of δ and A_j, we have for any $P \in \mathcal{P}_j$ and pair of states $q'', q''' \in Q$ that $\mathcal{P}_{j-2}(\delta(q'', (j, P, q, q'))) = \mathcal{P}_{j-2}(\delta(q''', (j, P, q, q')))$. Since $i \leq j - 2$, $\mathcal{P}_{j-2}$ refines $\mathcal{P}_i$, and hence also $\mathcal{P}_i(\delta(q'', (j, P, q, q'))) = \mathcal{P}_i(\delta(q''', (j, P, q, q')))$.

To show (2), note that $\mathcal{P}_i$ refines $\mathcal{P}_j$. Hence, for any two states $q'', q''' \in Q$ such that $\mathcal{P}_i(q'') = \mathcal{P}_i(q''')$ we find $\mathcal{P}_j(q'') = \mathcal{P}_j(q''')$. By definition of δ, this implies that for any $(j, P, q, q') \in A_j$, $\delta(q'', (j, P, q, q')) = \delta(q''', (j, P, q, q'))$.

Given these observations, we can prove that the partitions computed by Moore's algorithm are precisely the partitions $\mathcal{P}_i$. By definition of $\mathcal{P}_0$, this holds for $i = 0$.

Assume now that the i-th partition computed by Moore's algorithm is $\mathcal{P}_i$, where $i < n$ (when $i = n$, we are done). Then by the first observation, the letters from A_j with $j \geq i + 2$ won't cause splits. Furthermore, the second observation shows that letters from A_j with $j \leq i$ won't cause splits. That leaves the elements of A_{i+1}. Observe that, for two states $q'', q''' \in Q$ with $\mathcal{P}_i(q'') = \mathcal{P}_i(q''')$, we have $\mathcal{P}_i(\delta(q'', (i + 1, P, q, q'))) = \mathcal{P}_i(\delta(q'', (i + 1, P, q, q')))$

for all $(i+1, P, q, q') \in A_{i+1}$ if and only if $\mathcal{P}_{i+1}(q'') = \mathcal{P}_{i+1}(q''')$. Hence, the $i+1$-th partition calculated by Moore's algorithm is $\mathcal{P}_{i+1}$.

Induction on i now proves that the i-th partition computed by Moore's algorithm is $\mathcal{P}_i$, for any $0 \leq i \leq n$, completing the proof. □

Another interesting question is if, like in the classical case, we can find optimizations of Moore's algorithm reducing its time complexity. The factor $\mathrm{N}(Q)^3\,\mathrm{N}(A)$ suggest that the current formulation is equivalent to a naive implementation, suggesting there might be room for improvement. When working with classical automata, the running time of Moore's algorithm is improved by sorting the set of states according to the partition. Unfortunately, it is unclear to the author how such an approach would generalize to nominal sets and eliminate a factor $\mathrm{N}(Q)$.

Similarly, a generalization of Hopcroft's algorithm to the nominal case is difficult, as it is unclear how to implement a fast method for applying splitters to partitions of nominal sets.

4.2 Performance testing of Ons

We now use algorithms working with automata to measure the performance of Ons. For this, a number of choices need to be made. Most importantly, a choice needs to be made as to which set of automata to use for testing, and how to measure running time.

For testcases, the decision was made to use two classes of automata: random automata and structured automata. The random automata were chosen since we did not have a good source of practical automata. Structured automata were added to test how effectively the formula based approach of Nλ and Lois is able to exploit the availability of extra symmetries in automata.

Random Automata

We generate random automata as follows. The input alphabet is always $\mathbb{Q}$ and the number of orbits $\mathrm{N}(Q)$ and dimension k of Q are fixed. For each orbit in the set of states, the dimension is chosen uniformly at random between 0 and k, inclusive. Each orbit has probability $\frac{1}{2}$ of being an accepting state.

To generate the transition function δ, we enumerate the orbits of $Q \times \mathbb{Q}$ and choose a target state uniformly from the orbits Q with small enough dimension. The bit string indicating which part of the support is preserved is then sampled uniformly from all valid strings. We will denote these automata as $\mathrm{rand}_{N(Q),k}$.

Structured Automata

For structured automata, two sets were used. The set of automata FIFO(n) modeling a finite first-in first-out queue of length n, and the family of automata $ww(n)$ accepting the language of repeated words of length n were taken from [17]. The automaton for FIFO(2) is shown in Figure 4.3. To this were

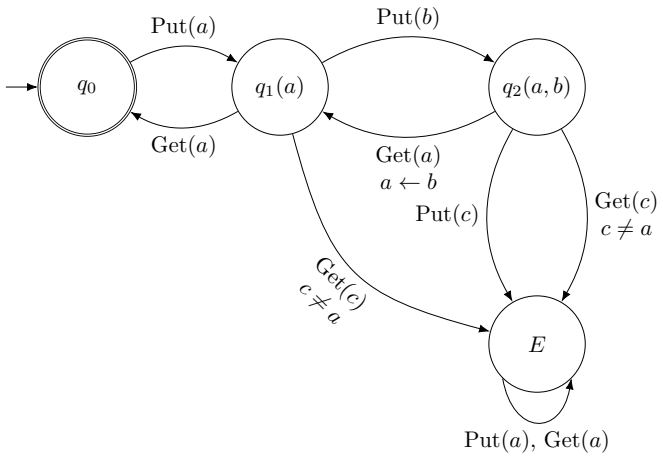

Figure 4.3: The FIFO(2) automaton. Note that unless explicitly noted, there are no restrictions on the values of a and b

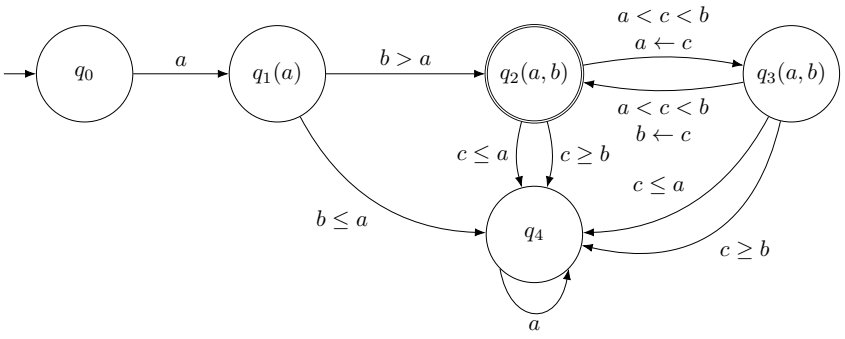

Figure 4.4: Example automaton that accepts the language $\mathcal{L}_{\text{int}}$.

added the $\mathcal{L}_{\max}$ automaton from Example 5, and the automaton recognizing $\mathcal{L}_{\text{int}}$ described below.

Example 6. Let $\mathcal{L}_{\text{int}} \subset \mathbb{Q}^*$ be the interval language, containing words of the form $w = x_1y_1x_2y_2 \dots x_ny_n$ with $n \geq 1$, $x_i < y_i$ for all $1 \leq i \leq n$, and $x_i < x_{i+1}$, $y_i > y_{i+1}$ for all $1 \leq i < n$. This is called the interval language as the pairs (x_i, y_i) form a sequence of narrowing intervals. An automaton recognizing $\mathcal{L}_{\text{int}}$ is given in Figure 4.4.

Time measurement

All time measurements were done on a dedicated machine, running only the operating system and the code under test. Running time was measured using

Type	N(S)	N($S^{\min}$)	Ons		Nλ	Lois
				Gen.		
$\text{rand}_{5,1}$ (x10)	5	n/a	0.02s	n/a	0.82s	3.14s
$\text{rand}_{10,1}$ (x10)	10	n/a	0.03s	n/a	17.03s	1m 32s
$\text{rand}_{10,2}$ (x10)	10	n/a	0.09s	n/a	35m 14s	> 60m
$\text{rand}_{15,1}$ (x10)	15	n/a	0.04s	n/a	1m 27s	10m 20s
$\text{rand}_{15,2}$ (x10)	15	n/a	0.11s	n/a	55m 46s	> 60m
$\text{rand}_{15,3}$ (x10)	15	n/a	0.46s	n/a	> 60m	> 60m
FIFO(2)	13	6	0.01s	0.01s	1.37s	0.24s
FIFO(3)	65	19	0.38s	0.09s	11.59s	2.4s
FIFO(4)	440	94	39.11s	1.60s	1m 16s	14.95s
FIFO(5)	3686	635	> 60m	39.78s	6m 42s	1m 11s
$ww(2)$	8	8	0.00s	0.00s	0.14s	0.03s
$ww(3)$	24	24	0.19s	0.02s	0.88s	0.16s
$ww(4)$	112	112	26.44s	0.25s	3.41s	0.61s
$ww(5)$	728	728	> 60m	6.37s	10.54s	1.80s
$\mathcal{L}_{\max}$	5	3	0.00s	0.00s	2.06s	0.03s
$\mathcal{L}_{\text{int}}$	5	5	0.00s	0.00s	1.55s	0.03s

Table 4.1: Running times for Algorithm 5 as implemented in the three libraries. $N(S)$ is the size of the input and $N(S^{\min})$ the size of the minimal automaton. For Ons, the time used to generate the automaton is reported separately.

the `time` command. The testing machine was a desktop with the following specifications and software versions:

Processor	i5-3470 @ 3.20GHz
OS	Ubuntu 17.10
GCC	version 7.2.0
Stack	version lts-9.8 (includes GHC 8.0.2)
Z3	version 4.4.1
nlambda	commit 744b4d3
lois	commit 9fe3d85
nominal-lstar	commit 96ae5ba

4.2.1 Performance on minimization

We implemented the minimization algorithm in Ons. For Nλ and Lois we used their implementations [14, 15, 16]. For each of the libraries, we wrote routines to read an automaton from a file and, for the structured testcases, to generate the requested automaton. For Ons, all automata were read from file. The output of these programs was checked manually to see if the minimization was performed correctly.

The results (shown in Table 4.1) for random automata show a clear advantage for Ons, which is capable of running all supplied testcases in less than one second. This in contrast to both Lois and Nλ, which take more than 2 hours

Model	$\mathrm{N}(S)$	$\dim(S)$	Ons time	Ons MQs	Nλ^{ord} time	Nλ^{ord} MQs	Nλ^{eq} time	Nλ^{eq} MQs
$\mathrm{rand}_{5,1}$	4	1	2m 7s	2321	39m 51s	1243		
$\mathrm{rand}_{5,1}$	5	1	0.12s	404	40m 34s	435		
$\mathrm{rand}_{5,1}$	3	0	0.86s	499	30m 19s	422		
$\mathrm{rand}_{5,1}$	5	1	$>$ 60m	n/a	$>$ 60m	n/a		
$\mathrm{rand}_{5,1}$	4	1	0.08s	387	34m 57s	387		
FIFO(1)	3	1	0.04s	119	3.17s	119	1.76s	51
FIFO(2)	6	2	1.73s	2655	6m 32s	3818	40.00s	434
FIFO(3)	19	3	46m 34s	298400	$>$ 60m	n/a	34m 7s	8151
$ww(1)$	4	1	0.42s	134	2.49s	77	1.47s	30
$ww(2)$	8	2	4m 26s	3671	3m 48s	2140	30.58s	237
$ww(3)$	24	3	$>$ 60m	n/a	$>$ 60m	n/a	$>$ 60m	n/a
$\mathcal{L}_{\max}$	3	1	0.01s	54	3.58s	54		
$\mathcal{L}_{\mathrm{int}}$	5	2	0.59s	478	1m 23s	478		

Table 4.2: Running times and number of membership queries for the $\nu\mathrm{L}^\star$ algorithm. For Nλ we used two version: Nλ^{ord} uses the total order symmetry Nλ^{eq} uses the equality symmetry (Originally from [24]).

on the largest automata.

The results for structured automata show a clear effect of the extra structure. Both Nλ and Lois remain capable of minimizing the automata in reasonable amounts of time for larger sizes. In contrast, Ons benefits little from the extra structure. Despite this, it remains viable: even for the larger cases it falls behind significantly only for the largest FIFO automaton and the two largest ww automata.

4.2.2 Performance on learning

In [24], automata learning is also presented as a testbed for Ons. In automata learning, the goal is to deduce the structure of an automaton based on its external behaviour. The software tries to reconstruct an automaton by asking two types of queries: "Is this word accepted", and "is this the correct automaton". Theoretical background on how this is achieved in the $\nu\mathrm{L}^\star$ algorithm can be found in [17].

J. Moerman implemented $\nu\mathrm{L}^\star$ in Ons. This is compared to the existing implementation in Nλ from [17]. No direct comparison with Lois was done, as the authors of Lois reported in private communication that an implementation of learning in Lois showed similar performance as that in Nλ.

The Nλ implementation of $\nu\mathrm{L}^\star$ also works for the equality symmetry. A number of the languages from the structured automata are also languages under the equality symmetry. For these languages, the Nλ implementation was run using both symmetries. This allows for an evaluation of the advantage of using a different symmetry.

In measuring running time, care was taken to make sure that compute time used to evaluate membership queries was not included in the measurement, as this time will not cause significant differences when using $\nu L^\star$ for black box learning. Furthermore, counterexamples for the equivalence queries were computed by hand, ensuring that each implementation got the same counterexamples.

The results in Table 4.2 show the running time of each of the implementations of the learning algorithm, as well as the number of queries made. On random automata, ONS shows a clear advantage. This advantage remains for the structured examples when Nλ is using the total order symmetry, with only the $ww(2)$ automaton showing a slight edge for the Nλ implementation.

The results for Nλ with the equivalence symmetry however show that the reduction in the number of queries needed has a significant impact. This allows the Nλ implementation to beat the ONS implementation significantly on all but the smallest examples $ww(1)$ and FIFO(1).

Chapter 5

Conclusion and Outlook

We used the representation theory from [3] and derived an explicit representation for nominal sets over the total order symmetry. This representation was then implemented in the Ons library, the first general purpose library to calculate on nominal sets using representation theory explicitly.

The explicit use of representation theory allowed us to derive time complexity bounds for basic set operations such as unions, intersections and products, among others. This in turn allowed the derivation of a time complexity bound on Moore's algorithm. These results do not exist for the Nλ and Lois libraries.

Furthermore, the time measurements comparing Ons to Lois and Nλ show that Ons is an improvement for learning automata over the total order symmetry. For minimization, the results were more split, but still showed a significant improvement on random automata.

Outlook

The results above already show some potential for the use of Ons in learning automata over the total order symmetry, an application that could be useful for software verification. However, it is currently limited to using the total order symmetry. This is an expressive symmetry, but also results in more queries and larger result automata when compared to the equality symmetry. This disadvantage is particularly evident in those testcases where the underlying symmetry was really an equality symmetry, as Nλ showed both reductions in the number of queries as well as compute time required. An interesting extension in the future would be generalizing Ons to the equality symmetry, and potentially other data symmetries. The primary obstacle for this is finding a suitable method for calculating with the finite groups in the representation of Theorem 5.

Furthermore, Ons is currently limited in the type of sets it can work with. Sets of sets in particular, such as $\{\{(a, b) \mid b \in Q \wedge b > a\} \mid a \in Q\}$, cannot currently be represented explicitly in Ons. Implementing these would require some way to work with sets that are non-nominal, but still exhibit some form of symmetry, such as $\{(\frac{1}{2}, b) \mid b \in Q \wedge b > \frac{1}{2}\}$. The current idea is that this should be possible by representing these as nominal sets, but with some of the support of the elements of each orbit fixed.

Appendix A

Admittance of least supports and infinite products

As alluded to in Chapter 2, there exist data symmetries that admit least support, but for which products do not preserve finiteness. This appendix contains a worked out example of one such data symmetry.

First, some notation. Given an element $a \in A^{\mathbb{N}}$, a is an infinite sequence of elements of A. Let $a(i)$ denote the i-th element of that sequence. We will use 1 to refer to the identity elements of groups.

Definition 42. The *infinite equality symmetry* is the tuple $(\mathcal{D}, G)$ with:

$$\mathcal{D} = \{d \in \mathbb{N}^{\mathbb{N}} \mid \exists N, \forall i > N, d(i) = 0\}$$
$$G = \{\pi \in \mathrm{Sym}(\mathbb{N})^{\mathbb{N}} \mid \exists N, \forall i > N, \pi(i) = 1\}$$

Unless explicitly mentioned, throughout the rest of this appendix we will work with the infinite equality symmetry.

Before we start proving the core results, let us first introduce some useful notation, and make some simplifying observations. Given a subset $C \subseteq \mathcal{D}$, let $C(i) = \{d(i) | d \in C\}$. Similarly, given a subgroup $H \leq G$, we can define $H(i) = \{\pi(i) \mid \pi \in H\} \leq \mathrm{Sym}(\mathbb{N})$. Note that given an index $i \in \mathbb{N}$, we now have two subgroups of $\mathrm{Sym}(\mathbb{N})$: $\mathrm{Sym}(\mathbb{N})_{C(i)}$, and $G_C(i)$. Although at first sight these look different, they are not:

Lemma 19. *The subgroups* $\mathrm{Sym}(\mathbb{N})_{C(i)}$ *and* $G_C(i)$ *are equal.*

Proof. Consider any $\pi \in G_C$. By definition of G_C, for any $d \in C$ we have $d\pi = d$. In particular this means $d(i)\pi(i) = d(i)$. Since for any $n \in C(i)$ there exists a $d \in C$ with $d(i) = n$, it follows that $\pi(i) \in G_{C(i)}$. And since for any $\pi' \in G_C(i)$ there exists a $\pi \in G_C$ such $\pi(i) = \pi'$, this shows that $G_C(i) \leq \mathrm{Sym}(\mathbb{N})_{C(i)}$.

Conversely, let $\pi' \in \mathrm{Sym}(\mathbb{N})_{C(i)}$. Construct $\pi \in G$ such that

$$\pi(j) = \begin{cases} \pi' & i = j \\ 1 & i \neq j \end{cases}$$

holds. Consider any $d \in C$. Since $d(i) \in C(i)$, $d(i)\pi(i) = d(i)\pi' = d(i)$. For $j \neq i$, $d(j)\pi(j) = d(j)1 = d(j)$. This implies $d\pi = d$, and hence $\pi \in G_C$. Since $\pi(i) = \pi'$, $\pi \in G_C$ implies $\pi' \in G_C(i)$. This shows $G_{C(i)} \leq G_C(i)$.

Combining these proves that $\mathrm{Sym}(\mathbb{N})_{C(i)} = G_C(i)$. □

From here on out, we will not distinguish between $\mathrm{Sym}(\mathbb{N})_{C(i)}$ and $G_C(i)$, and always write $G_C(i)$. Using the above result, we now prove a construction for elements of G_C.

Lemma 20. *Given a finite subset $C \subseteq \mathcal{D}$, and a finite sequence of group elements $\pi_0 \in G_C(0), \pi_2 \in G_C(1), \ldots \pi_k \in G_C(k)$, the group element $\pi \in G$ defined with*

$$\pi(i) = \begin{cases} \pi_i & i \leq k \\ 1 & i > k \end{cases}$$

is an element of G_C.

Proof. Let $d \in C$. Then for $0 \leq i \leq k$, we find $d(i) \in C(i)$, and since $\pi_i \in G_C(i)$, it follows that $d(i)\pi(i) = d(i)\pi_i = d(i)$. For $i > k$, $\pi(i) = 1$, and hence $d(i)\pi(i) = d(i)1 = d(i)$. Combining this, $d\pi = d$, hence $\pi \in G_C$. □

The above lemmas and proofs allow us to more easily reason about the infinite equality symmetry. We will now show three things:

- The infinite equality symmetry admits least supports. (Lemma 21)
- The set $\mathcal{D}$ consists of a single orbit. (Lemma 22)
- The set $\mathcal{D}^2$ consists of infinitely many orbits. (Lemma 23)

Together, these three facts prove that the infinite equality symmetry is an example of a data symmetry that preserves least supports, but for which products don't preserve finiteness.

Lemma 21. *The infinite equality symmetry admits least supports.*

Proof. Using Lemma 6 it is sufficient to show that, given two finite subsets $C \subseteq \mathcal{D}$ and $C' \subseteq \mathcal{D}$, the identity $\langle G_C, G_{C'} \rangle = G_{C \cap C'}$ holds.

By construction, we immediately find $\langle G_C, G_{C'} \rangle \leq G_{C \cap C'}$. To show the converse, consider an element $\pi \in G_{C \cap C'}$. By definition of G, there exists an $N \in \mathbb{N}$ such that $\forall i > N, \pi(i) = 1$.

From the definitions it follows that, for $0 \leq i \leq N$, $\pi(i) \in G_{C \cap C'}(i)$. Since the regular equality symmetry $(\mathbb{N}, \mathrm{Sym}(\mathbb{N})$ admits least support, we have

$G_{C\cap C'}(i) = \langle G_C(i), G_{C'}(i)\rangle$. Using this we can find an $l \in \mathbb{N}$, and $\sigma_{i,j} \in G_C(i)$, $\tau_{i,j} \in G_{C'}(i)$ with $0 \leq j \leq l$ such that

$$\pi(i) = \sigma_{i,1}\tau_{i,1}\sigma_{i,2}\tau_{i,2}\dots\sigma_{i,l}\tau_{i,l}.$$

Using this define group elements $\sigma_i \in G$ and $\tau_i \in G$:

$$\sigma_i(j) = \begin{cases} \sigma_{i,j} & 0 \leq j \leq k \\ 1 & \text{otherwise} \end{cases},$$
$$\tau_i(j) = \begin{cases} \tau_{i,j} & 0 \leq j \leq k \\ 1 & \text{otherwise} \end{cases}.$$

By Lemma 20 we have $\sigma_i \in G_C$ and $\tau_i \in G_{C'}$. Furthermore, by construction

$$\sigma_0\tau_0\sigma_1\tau_1\dots\sigma_l\tau_l(i) = \begin{cases} \pi(i) & 0 \leq i \leq N \\ 1 & \text{otherwise} \end{cases}.$$

Since also $\pi(i) = 1$ for $i > N$, this implies $\sigma_0\tau_0\sigma_1\tau_1\dots\sigma_l\tau_l = \pi$. It follows that $\pi \in \langle G_C, G_{C'}\rangle$, and as this holds for any $\pi \in G_{C\cap C'}$, $G_{C\cap C'} \leq \langle G_C, G_{C'}\rangle$.

Combining this with the previous conclusion that $\langle G_C, G_{C'}\rangle \leq G_{C\cap C'}$, we find that $G_{C\cap C'} = \langle G_C, G_{C'}\rangle$. Hence, the infinite equality symmetry admits least supports. □

Lemma 22. *The nominal set $\mathcal{D}$ consists of a single orbit.*

Proof. Let $d_1 \in \mathcal{D}$ and $d_2 \in \mathcal{D}$ be arbitrary elements. By definition of $\mathcal{D}$, there exists an $N_1 \in \mathbb{N}$ such that $d_1(i) = 0$ for all $i > N_1$, and an $N_2 \in \mathbb{N}$ such that $d_2(i) = 0$. Let $N = \max(N_1, N_2)$.

Now define the sequence $\pi_0, \pi_1, \dots \pi_N$, $\pi_i \in \mathrm{Sym}(\mathbb{N})$, such that

$$k\pi_i = \begin{cases} d_2(i) & k = d_1(i) \\ d_1(i) & k = d_2(i) \\ k & \text{otherwise} \end{cases}.$$

Then there exists a $\pi \in G$ such that

$$\pi(i) = \begin{cases} \pi_i & 0 \leq i \leq N \\ 1 & \text{otherwise} \end{cases}.$$

For this π, we find $d_1\pi = d_2$, since for $0 \leq i \leq N$, $d_1(i)\pi(i) = d_2(i)$ holds by construction, and for $i > N$ $d_1(i) = 0 = d_2(i)$ we immediately have $d_1(i)\pi(i) = 0 \cdot 1 = 0 = d_2(i)$. Hence d_1 and d_2 are in the same orbit. Since they were arbitrary elements of $\mathcal{D}$, it follows that $\mathcal{D}$ consists of a single orbit. □

Lemma 23. *The nominal set $\mathcal{D}^2$ contains an infinite number of orbits.*

Proof. We will show this by explicit construction. Define $f : \mathbb{N} \to \mathbb{N}$ and $g_i : \mathbb{N} \to \mathbb{N}$ as follows:

$$f(j) = 0$$
$$g_i(j) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

Since $f(j) = 0$ for all $j \in \mathbb{N}$, and $g_i(j) = 0$ for all $j > i$, it follows that both $f \in \mathcal{D}$ and $g_i \in \mathcal{D}$.

Now consider any two pairs (f, g_i) and (f, g_j). Suppose there exists a $\pi \in G$ such that $(f, g_i)\pi = (f, g_j)$. Then by definition of the action on $\mathcal{D}^2$, we have $f\pi = f$ and $g_i\pi = g_j$. Since $f\pi = f$, we find that for any $k, l \in \mathbb{N}$, $k\pi(l) = 0$ if and only if $k = 0$. However, if $i \neq j$, $g_i\pi = g_j$ requires that $1\pi(i) = 0$. Since this is a contradiction, π can only exist if $i = j$.

From this, we find that each pair (f, g_i) is contained in a distinct orbit. Hence $\mathcal{D}^2$ consists of an infinite number of orbits. □

Corollary 5. *There exists a data symmetry admitting least supports, for which products do not preserve finiteness.*

Bibliography

[1] F. Aarts, P. Fiterau-Brostean, H. Kuppens, and F. W. Vaandrager. Learning register automata with fresh value generation. In M. Leucker, C. Rueda, and F. D. Valencia, editors, *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, volume 9399 of *Lecture Notes in Computer Science*, pages 165–183. Springer, 2015.

[2] J. Berstel, L. Boasson, O. Carton, and I. Fagnot. Minimization of automata. *CoRR*, abs/1010.5318, 2010.

[3] M. Bojańczyk, B. Klin, and S. Lasota. Automata theory in nominal sets. *Logical Methods in Computer Science*, 10(3), 2014.

[4] M. Bojańczyk, B. Klin, S. Lasota, and S. Toruńczyk. Turing machines with atoms. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 183–192. IEEE Computer Society, 2013.

[5] B. Bollig, P. Habermehl, M. Leucker, and B. Monmege. A fresh approach to learning register automata. In M. Béal and O. Carton, editors, *Developments in Language Theory - 17th International Conference, DLT 2013, Marne-la-Vallée, France, June 18-21, 2013. Proceedings*, volume 7907 of *Lecture Notes in Computer Science*, pages 118–130. Springer, 2013.

[6] S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Asp. Comput.*, 28(2):233–263, 2016.

[7] G. Castagna and A. D. Gordon, editors. *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, 2017.

[8] S. Drews and L. D'Antoni. Learning symbolic automata. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 173–189, 2017.

[9] G. L. Ferrari, U. Montanari, R. Raggi, and E. Tuosto. From co-algebraic specifications to implementation: The mihda toolkit. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*, pages 319–338. Springer, 2002.

[10] P. Fiterau-Brostean, R. Janssen, and F. W. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 454–471. Springer, 2016.

[11] M. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002.

[12] J. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189 – 196. Academic Press, 1971.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[14] B. Klin and M. Szynwelski. SMT solving for functional programming over infinite structures. In R. Atkey and N. R. Krishnaswami, editors, *Proceedings 6th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2016, Eindhoven, Netherlands, 8th April 2016.*, volume 207 of *EPTCS*, pages 57–75, 2016.

[15] E. Kopczynski and S. Toruńczyk. LOIS: an application of SMT solvers. In T. King and R. Piskac, editors, *Proceedings of the 14th International Workshop on Satisfiability Modulo Theories affiliated with the International Joint Conference on Automated Reasoning, SMT@IJCAR 2016, Coimbra, Portugal, July 1-2, 2016.*, volume 1617 of *CEUR Workshop Proceedings*, pages 51–60. CEUR-WS.org, 2016.

[16] E. Kopczynski and S. Toruńczyk. LOIS: syntax and semantics. In Castagna and Gordon [7], pages 586–598.

[17] J. Moerman, M. Sammartino, A. Silva, B. Klin, and M. Szynwelski. Learning nominal automata. In Castagna and Gordon [7], pages 613–625.

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

[19] A. M. Pitts. Nominal techniques. *SIGLOG News*, 3(1):57–72, 2016.

[20] J. J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In D. Sangiorgi and R. de Simone, editors, *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*, volume 1466 of *Lecture Notes in Computer Science*, pages 194–218. Springer, 1998.

[21] M. R. Shinwell and A. M. Pitts. Fresh objective Caml user manual. Technical report, University of Cambridge, Computer Laboratory, 2005.

[22] C. Urban and C. Tasson. Nominal techniques in isabelle/hol. In R. Nieuwenhuis, editor, *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings*, volume 3632 of *Lecture Notes in Computer Science*, pages 38–53. Springer, 2005.

[23] F. W. Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, 2017.

[24] D. Venhoek, J. Moerman, and J. Rot. Using representation theory for fast computations on ordered nominal sets. In preparation.