Fast Computations on Ordered Nominal Sets*

David Venhoek[a], Joshua Moerman[a], Jurriaan Rot[b,a]

[a] *Institute for Computing and Information Sciences,*
Radboud Universiteit, Nijmegen, The Netherlands
[b] *University College London, London, UK*

Abstract

We show how to compute efficiently with nominal sets over the total order symmetry, by developing a direct representation of such nominal sets and basic constructions thereon. In contrast to previous approaches, we work directly at the level of orbits, which allows for an accurate complexity analysis. The approach is implemented as the library ONS (Ordered Nominal Sets).

Our main motivation is nominal automata, which are models for recognising languages over infinite alphabets. We evaluate ONS in two applications: minimisation of automata and active automata learning. In both cases, ONS is competitive compared to existing implementations and outperforms them for certain classes of inputs.

Keywords: nominal sets, automata theory, minimisation, automata learning

1. Introduction

Automata over infinite alphabets are natural models for programs with unbounded data domains. Such automata, often formalised as *register automata*, are applied in modelling and analysis of communication protocols, hardware, and software systems (see [2, 3, 4, 5, 6, 7] and references therein). Typical infinite alphabets include sequence numbers, timestamps, and identifiers. This means one can model data flow in such automata beside the basic control flow provided by ordinary automata. Recently, it has been shown in a series of papers that such models are amenable to learning [8, 9, 10, 11, 12, 13] with the verification of (closed source) TCP implementations as a prominent example [14].

A foundational approach to infinite alphabets is provided by the notion of *nominal set*, originally introduced in computer science as an elegant formalism

[*] This is a revised and extended version of a paper which appeared in the proceedings of ICTAC 2018 [1]. The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 795119.

Email addresses: `david@venhoek.nl` (David Venhoek), `joshua.moerman@cs.ru.nl` (Joshua Moerman), `jrot@cs.ru.nl` (Jurriaan Rot)

for name binding [15, 16]. Nominal sets have been used in a variety of applications in semantics, computation, and concurrency theory (see [17] for an overview). Bojańczyk et al. introduce *nominal automata*, which allow one to model languages over infinite alphabets with different symmetries [2]. Their results are parametric in the structure of the data values. Important examples of data domains are ordered data values (e.g., timestamps) and data values that can only be compared for equality (e.g., identifiers). In both data domains, nominal automata and register automata are equally expressive [2].

Important for applications of nominal sets and automata are implementations. A couple of tools exist to compute with nominal sets. Notably, Nλ [18] and LOIS [19, 20] provide a general purpose programming language to manipulate infinite sets.[1] Both tools are based on SMT solvers and use logical formulas to represent the infinite sets. These implementations are very flexible, and the SMT solver does most of the heavy lifting, which makes the implementations themselves relatively straightforward. Unfortunately, this comes at a cost as SMT solving is in general PSPACE-hard. Since the formulas used to describe sets tend to grow as more calculations are done, running times can become unpredictable.

In the current paper, we use a direct representation, based on symmetries and orbits, to represent nominal sets. We focus on the *total order symmetry*, where data values are rational numbers and can be compared for their order. Nominal automata over the total order symmetry are more expressive than automata over the equality symmetry (i.e., traditional register automata [5]). A key insight is that the representation of nominal sets from [2] becomes rather simple in the total order symmetry; each orbit is presented solely by a natural number, intuitively representing the number of variables or registers.

Our main contributions include the following.

- We develop the *representation theory* of nominal sets over the total order symmetry. We give concrete representations of nominal sets, their products, and equivariant maps.

- We provide *time complexity bounds* for operations on nominal sets such as intersections and membership. Using those results we give the time complexity of Moore's minimisation algorithm (generalised to nominal automata) and prove that it is polynomial in the number of orbits.

- Using the representation theory, we are able to *implement nominal sets in a C++ library* ONS. The library includes all the results from the representation theory (sets, products, and maps). We also developed a Haskell implementation, called ONS-HS.

- We *evaluate the performance* of ONS(-HS), and compare it to Nλ and

[1] Other implementations of nominal techniques that are less directly related to our setting (Mihda, Fresh OCaml, and Nominal Isabelle) are discussed in Section 10.

LOIS, using two algorithms on nominal automata: minimisation [21] and automata learning [12]. We use randomly generated automata as well as concrete, logically structured models such as FIFO queues. For random automata, our methods are considerably faster in most cases than the other tools. On the other hand, LOIS and Nλ are faster in minimising the structured automata as they exploit their logical structure. In automata learning, the logical structure is not available a-priori, and ONS(-HS) is faster in most cases.

The structure of the paper is as follows. The first three sections contain background material: Section 2 on nominal sets, Section 3 on nominal automata, and Section 4 on representation of nominal sets. Next, Section 5 describes the concrete representation of nominal sets, equivariant maps and products in the total order symmetry. The implementation in C++ and Haskell are presented in Sections 6 and 7 respectively. Complexity results are presented in Section 8. Section 9 reports on the evaluation of ONS on algorithms for nominal automata. Related work is discussed in Section 10, and future work in Section 11.

The current paper extends the conference version (ICTAC 2018 [1]) with proofs of all results, new experiments for evaluating ONS based on randomly generated formulas, and an implementation in Haskell, ONS-HS.

2. Nominal sets

Nominal sets are infinite sets that carry certain symmetries, allowing a finite representation in many interesting cases. We recall their formalisation in terms of group actions, following [2, 17], to which we refer for an extensive introduction.

2.1. Group actions.

Let G be a group, with the multiplication denoted by juxtaposition and the unit by 1. Given a set X, a *(right) G-action* is a function $\cdot \colon X \times G \to X$ satisfying $x \cdot 1 = x$ and $(x \cdot g) \cdot h = x \cdot (gh)$ for all $x \in X$ and $g, h \in G$. A set X with a G-action is called a *G-set* and we often write xg instead of $x \cdot g$. The *orbit* of an element $x \in X$ is the set $\{xg \mid g \in G\}$. A G-set is always a disjoint union of its orbits (in other words, the orbits partition the set). We say that X is *orbit-finite* if it has finitely many orbits, and we denote the number of orbits by $\mathrm{N}(X)$.

A map $f \colon X \to Y$ between G-sets is called *equivariant* if it preserves the group action, i.e., for all $x \in X$ and $g \in G$ we have $f(x)g = f(xg)$. If an equivariant map f is bijective, then f is an *isomorphism* and we write $X \cong Y$. A subset $Y \subseteq X$ is called equivariant if for all $y \in Y$ and $g \in G$, we have $yg \in Y$. The *product* of two G-sets X and Y is given by the Cartesian product $X \times Y$ with the pointwise group action on it, i.e., $(x, y)g = (xg, yg)$. Union and intersection of X and Y are well-defined if the two actions agree on their common elements.

2.2. Nominal sets.

A *data symmetry* is a pair $(\mathcal{D}, G)$ where $\mathcal{D}$ is a set and G is a subgroup of $\mathrm{Sym}(\mathcal{D})$, the group of bijections on $\mathcal{D}$. Note that the group G naturally acts on $\mathcal{D}$ by defining $xg = g(x)$.[2] In the most studied instance, called the *equality symmetry*, $\mathcal{D}$ is a countably infinite set and $G = \mathrm{Sym}(\mathcal{D})$. In this paper, we will mostly focus on the *total order symmetry* given by $\mathcal{D} = \mathbb{Q}$ and $G = \{\pi \mid \pi \in \mathrm{Sym}(\mathbb{Q}), \pi \text{ is monotone}\}$.

Let $(\mathcal{D}, G)$ be a data symmetry and X be a G-set. A finite set of data values $S \subseteq \mathcal{D}$ is called a *support* of an element $x \in X$ if for all $g \in G$ with $\forall s \in S \colon sg = s$ we have $xg = x$. A G-set X is called *nominal* if every element $x \in X$ has a (necessarily finite) support.

Example 2.1. We list several examples for the total order symmetry. The set $\mathbb{Q}^2$ is nominal as each element $(q_1, q_2) \in \mathbb{Q}^2$ has the finite set $\{q_1, q_2\}$ as its support. The set has the following three orbits:

$$\{(q_1, q_2) \mid q_1 < q_2\}\,,\ \{(q_1, q_2) \mid q_1 > q_2\}\,,\ \{(q_1, q_2) \mid q_1 = q_2\}\,.$$

For a set X, the set of all subsets of size $n \in \mathbb{N}$ is denoted by

$$\mathcal{P}_n(X) = \{Y \subseteq X \mid \#Y = n\}\,.$$

The set $\mathcal{P}_n(\mathbb{Q})$ is a single-orbit nominal set for each n, with the action defined by direct image: $Yg = \{yg \mid y \in Y\}$.

The group of monotone bijections also acts by direct image on the full power set $\mathcal{P}(\mathbb{Q})$, but this is *not* a nominal set. For instance, the set $\mathbb{Z} \in \mathcal{P}(\mathbb{Q})$ of integers has no finite support.

If $S \subseteq \mathcal{D}$ is a support of an element $x \in X$, then any finite set $S' \subseteq \mathcal{D}$ such that $S \subseteq S'$ is also a support of x. A set $S \subseteq \mathcal{D}$ is a *least support* of $x \in X$ if it is a support of x and $S \subseteq S'$ for any support S' of x. The existence of least supports is crucial for representing orbits. Unfortunately, even when elements have a finite support, in general they do not always have a least support. A data symmetry $(\mathcal{D}, G)$ is said to *admit least supports* if every element of every nominal set has a least support. Both the equality and the total order symmetry admit least supports. For other (counter)examples of data symmetries admitting least supports, see [2]. Having least supports is useful for a finite representation.

Given a nominal set X, the size of the least support of an element $x \in X$ is denoted by $\dim(x)$, the *dimension* of x. We note that all elements in the orbit of x have the same dimension. For an orbit-finite nominal set X, we define

$$\dim(X) = \max\{\dim(x) \mid x \in X\}\,.$$

For a single-orbit nominal set O, observe that $\dim(O) = \dim(x)$ where x is any element $x \in O$.

[2] This is a well-defined action if we use the group multiplication $f \cdot g = g \circ f$.

3. Automata over Nominal Sets

Nominal sets are used to formalise languages over infinite alphabets. These languages naturally arise as the semantics of register automata. Register automata were introduced by Kaminski and Francez [5] and the connection to nominal automata is well exposed by Bojańczyk [22]. The definition of register automata is not as simple as that of ordinary finite automata. Consequently, transferring results from automata theory to this setting often requires non-trivial proofs. Nominal automata, instead, are defined as ordinary automata by replacing finite sets with orbit-finite nominal sets. The theory of nominal automata is developed in [2] and it is shown that many algorithms from automata theory transfer to nominal automata. For instance, emptiness and equivalence of deterministic automata can be decided with a slight adaptation of the classical algorithms. Nonetheless, not all algorithm generalise: equivalence of non-deterministic automata is undecidable in the nominal setting.

Example 3.1. Consider the following language on rational numbers:

$$\mathcal{L}_{\text{int}} = \{a_1 b_1 \cdots a_n b_n \mid a_i, b_i \in \mathbb{Q}, a_i < a_{i+1} < b_{i+1} < b_i \text{ for all } i\}.$$

We call this language the *interval language* as a word $w \in \mathbb{Q}^*$ is in the language when it denotes a sequence of nested intervals. This language contains arbitrarily long words. For this language it is crucial to work with an infinite alphabet as for each finite set $C \subset \mathbb{Q}$, the restriction $\mathcal{L}_{\text{int}} \cap C^*$ is just a finite language. Note that the language is equivariant: $w \in \mathcal{L}_{\text{int}} \iff wg \in \mathcal{L}_{\text{int}}$ for any monotone bijection g, because nested intervals are preserved by monotone maps.[3] Indeed, $\mathcal{L}_{\text{int}}$ is a nominal set, although it is not orbit-finite.

Informally, the language $\mathcal{L}_{\text{int}}$ can be accepted by the automaton depicted in Figure 1. Here we allow the automaton to store rational numbers and compare them to new symbols. For example, the transition from q_2 to q_3 is taken if any value c between a and b is read and then the currently stored value a is replaced by c. For any other value read at state q_2 the automaton transitions to the sink state q_4. Such a transition structure is made precise by the notion of nominal automaton.

Definition 3.2. A *nominal language* is an equivariant subset $L \subseteq A^*$ where A is an orbit-finite nominal set.

Definition 3.3. A *nominal deterministic finite automaton* is a tuple (S, A, F, δ), where S is an orbit-finite nominal set of states, A is an orbit-finite nominal set of symbols, $F \subseteq S$ is an equivariant subset of final states, and $\delta \colon S \times A \to S$ is the equivariant transition function.

Given a state $s \in S$, we define the usual acceptance condition: a word $w \in A^*$ is *accepted* if w denotes a path from s to a final state.

[3] The G-action on words is defined point-wise: $(w_1 \dots w_n)g = (w_1 g) \dots (w_n g)$.

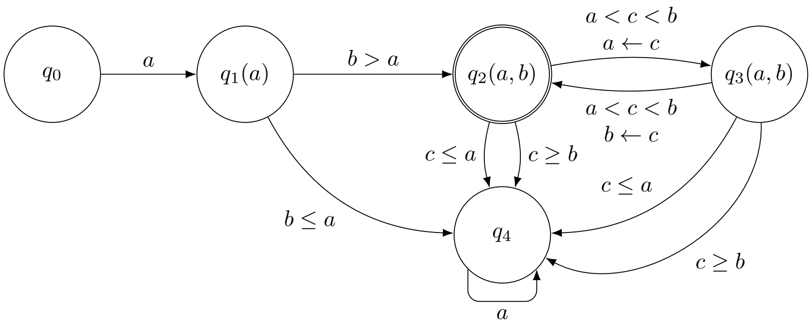


Figure 1: Example automaton that accepts the language $\mathcal{L}_{\text{int}}$.

The automaton in Figure 1 can be formalised as a nominal deterministic finite automaton as follows. Let

$$S = \{q_0, q_4\} \cup \{q_1(a) \mid a \in \mathbb{Q}\} \cup \{q_2(a,b) \mid a < b \in \mathbb{Q}\} \cup \{q_3(a,b) \mid a < b \in \mathbb{Q}\}$$

be the set of states, where the group action is defined as one would expect. The transition we described earlier can now be formally defined as

$$\delta(q_2(a,b), c) = q_3(c,b) \quad \text{for all } a < c < b \in \mathbb{Q}\,.$$

By defining δ on all states accordingly and defining the final states as

$$F = \{q_2(a,b) \mid a < b \in \mathbb{Q}\}\,,$$

we obtain a nominal deterministic automaton $(S, \mathbb{Q}, F, \delta)$. The state q_0 accepts the language $\mathcal{L}_{\text{int}}$.

3.1. Minimisation of Nominal Automata

For languages recognised by nominal DFAs, a Myhill-Nerode theorem holds which relates states to right congruence classes [2]. This guarantees the existence of unique minimal automata. We say an automaton is *minimal* if its set of states has the least number of orbits and each orbit has the smallest dimension possible.[4]

Example 3.4. Consider the language

$$\mathcal{L}_{\max} = \{wa \in \mathbb{Q}^* \mid a = \max(w_1, \ldots, w_n)\}$$

consisting of those words where the last symbol is the maximum of previous symbols. Figure 2 depicts a nominal automaton accepting $\mathcal{L}_{\max}$, which is however not minimal. Figure 3 is the minimal nominal automaton accepting $\mathcal{L}_{\max}$.

[4] Abstractly, an automaton is minimal if it has no proper quotients. Minimal deterministic automata are unique up to isomorphism.

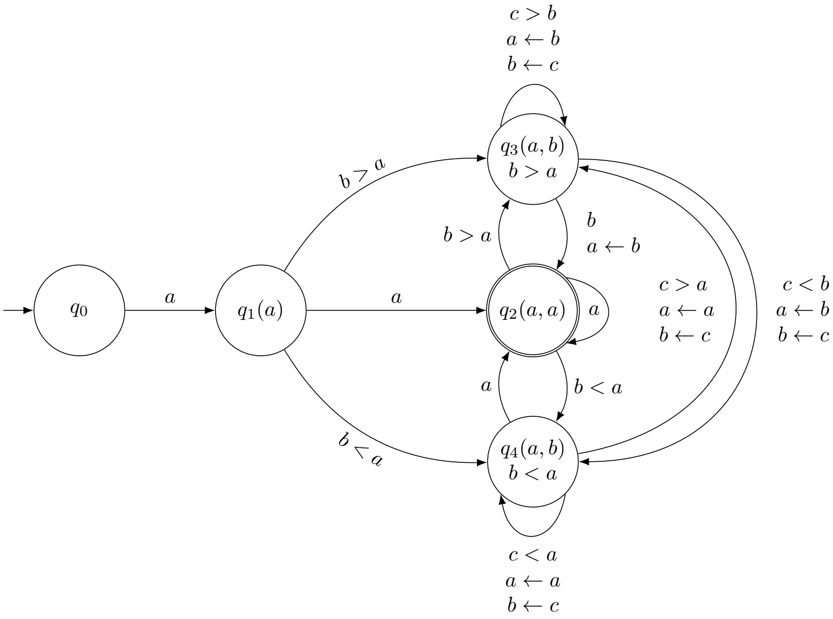


Figure 2: Example automaton that accepts the language $\mathcal{L}_{\max}$.

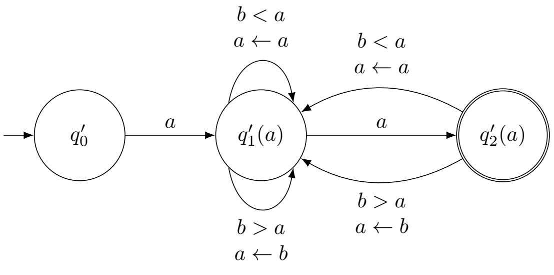


Figure 3: The automaton from Figure 2, minimised.

There exist algorithms in order to minimise automata. In this paper we focus on Moore's minimisation algorithm. It generalises to nominal DFAs since it uses set operations which work just as well on nominal sets (see Algorithm 1). We will perform a complexity analysis in Section 8 and later use this algorithm for testing our library.

Algorithm 1 Moore's minimisation algorithm for nominal DFAs

```
Require: Nominal automaton $(S, A, F, \delta)$.
$i \leftarrow 0$, $\equiv_{-1} \leftarrow S \times S$, $\equiv_0 \leftarrow F \times F \cup (S \backslash F) \times (S \backslash F)$
while $\equiv_i \neq \equiv_{i-1}$ do
    $\equiv_{i+1} = \{(q_1, q_2) \mid (q_1, q_2) \in \equiv_i \wedge \forall a \in A, (\delta(q_1, a), \delta(q_2, a)) \in \equiv_i\}$
    $i \leftarrow i + 1$
end while
$E \leftarrow S/_{\equiv_i}$
$F_E \leftarrow \{e \in E \mid \forall s \in e, s \in F\}$
Let $\delta_E$ be the map such that, if $s \in e$ and $\delta(s, a) \in e'$, then $\delta_E(e, a) = e'$.
return $(E, A, F_E, \delta_E)$.
```

3.2. Learning nominal automata

Another interesting application is *automata learning*. The aim of automata learning is to infer an unknown regular language $\mathcal{L}$. We use the framework of active learning as set up by Dana Angluin [23] where a learning algorithm can query an oracle to gather information about $\mathcal{L}$. Formally, the oracle can answer two types of queries:

1. *membership queries*, where a query consists of a word $w \in A^*$ and the oracle replies whether $w \in \mathcal{L}$, and
2. *equivalence queries*, where a query consists of an automaton $\mathcal{H}$ and the oracle replies positively if $\mathcal{L}(\mathcal{H}) = \mathcal{L}$ or provides a counterexample if $\mathcal{L}(\mathcal{H}) \neq \mathcal{L}$.

With these queries, the $\mathrm{L}^\star$ algorithm can learn regular languages efficiently [23]. In particular, it learns the unique minimal automaton for $\mathcal{L}$ using only finitely many queries. The $\mathrm{L}^\star$ algorithm has been generalised to $\nu\mathrm{L}^\star$ in order to learn *nominal* regular languages [12]. In particular, it learns a nominal DFA (over an infinite alphabet) using only finitely many queries. The algorithm is not polynomial, unlike the minimisation algorithm described above. However, the authors conjecture that there is a polynomial algorithm.[5] For the correctness, termination, and comparison with other learning algorithms see [12].

Learning register automata is an active research area with applications such as bug-finding in internet protocols [24]; see [13] for other applications. We will implement $\nu\mathrm{L}^\star$ to test our library in Section 9.4.

[5] See `joshuamoerman.nl/papers/2017/17popl-learning-nominal-automata.html` for a sketch of the polynomial algorithm.

4. Representing nominal orbits

In this section we recall the representation of nominal sets according to [2]. We represent nominal sets as collections of single orbits. The finite representation of single orbits is makes use of the technical notions of *restriction* and *extension.* We only briefly report their definitions here. However, the reader can safely move to the concrete representation theory in Section 5 with only a superficial understanding of Theorem 4.1 below.

The *restriction* of an element $\pi \in G$ to a subset $C \subseteq \mathcal{D}$, written as $\pi|_C$, is the restriction of the function $\pi \colon \mathcal{D} \to \mathcal{D}$ to the domain C. The restriction of a group G to a subset $C \subseteq \mathcal{D}$ is defined as

$$G|_C = \{\pi|_C \mid \pi \in G,\, C\pi = C\}\,.$$

The *extension* of a subgroup $S \leq G|_C$ is defined as

$$\mathrm{ext}_G(S) = \{\pi \in G \mid \pi|_C \in S\}\,.$$

For $C \subseteq \mathcal{D}$ and $S \leq G|_C$, define

$$[C, S]^{ec} = \{\{sg \mid s \in \mathrm{ext}_G(S)\} \mid g \in G\}\,,$$

i.e., the set of right cosets of $\mathrm{ext}_G(S)$ in G. Then $[C, S]^{ec}$ is a single-orbit nominal set.

Using the above, we can formulate the representation theory from [2] that we will use in the current paper. This gives a finite description for all single-orbit nominal sets X, namely a finite set C together with some of its symmetries.

Theorem 4.1. *Let X be a single-orbit nominal set for a data symmetry $(\mathcal{D}, G)$ that admits least supports and let $C \subseteq \mathcal{D}$ be the least support of some element $x \in X$. Then there exists a subgroup $S \leq G|_C$ such that $X \cong [C, S]^{ec}$.*

Proof sketch. We restrict to a sketch of the main ideas of the proof, and refer the reader to [2] for a fully worked out version of the proof.

First, we note that an element $x \in X$ can be fully described by the subgroup H of G of all group elements that leave x invariant. Properties such as whether a set C is a support of x can be determined by just examining the structure of the subgroup H. Most importantly here, it can be shown that, similarly to how one can reconstruct the single-orbit set X from x as the latter's orbit $\{xg \mid g \in G\}$, the orbit X is isomorphic to the set of right cosets of H in G.

This approach reduces the problem to describing subgroups of the group G. This is not possible for general subgroups, as there can be uncountably many such subgroups. However, the subgroups from the previous step are special in that they have a least support.

To represent such a subgroup we use the least support to split the elements of this subgroup into three categories. First, there are group elements that are "trivially" part of H as they have the elements of the least support C as fixed points. Second, there are elements that act as the identity outside of C, but permute the elements of C. Finally, there are the elements of H that both permute the elements of C, and that do not act as the identity outside of C.

This last category of elements can be shown to be generated by elements from the first and second category. But the first two categories have relatively straightforward, finite, representations:

- The group elements that have the elements of C as a fixed points can simply be described by the set C.

- The group elements that only permute C can be restricted to C, and then form (with addition of the identity) a subgroup of the permutation group of C, which is a finite group.

These elements combined form the representation of the above theorem: A finite subset C of $\mathcal{D}$, and a finite group S acting on C. □

5. Representation in the total order symmetry

This section develops a concrete representation of nominal sets over the total order symmetry, as well as equivariant maps and products. It is based on the abstract representation theory from Section 4. From now on, by *nominal set* we always refer to a nominal set over the total order symmetry. Hence, our data domain is $\mathbb{Q}$ and we take G to be the group of monotone bijections.

5.1. Orbits and nominal sets

From the representation in Section 4, we find that any single-orbit set X can be represented as a tuple (C, S). Our first observation is that the finite group of 'local symmetries', S, in this representation is always trivial, i.e., $S = I$, where $I = \{1\}$ is the trivial group. This follows from the following lemma and the fact that $S \leq G|_C$.

Lemma 5.1. *For every finite subset $C \subset \mathbb{Q}$, we have $G|_C = I$.*

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 $G|_C = I$. □

Immediately, we see that $(C, S) = (C, I)$, and hence that the orbit is fully represented by the set C. Together with Theorem 4.1 this leads to a complete characterisation of $[C, I]^{ec}$ in Lemma 5.3. In its proof, we also need the following.

Lemma 5.2 (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 $C\pi = C'$.*

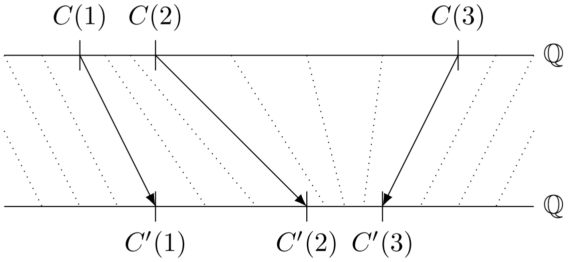


Figure 4: Visualisation of π from Lemma 5.2

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 4 for a visualisation):

$$\begin{aligned} x < C(1) &\implies x\pi = x - C(1) + C'(1) \\ x \geq C(1) \wedge x < C(2) &\implies x\pi = (x - C(1))\tfrac{C'(2)-C'(1)}{C(2)-C(1)} + C'(1) \\ x \geq C(2) \wedge x < C(3) &\implies x\pi = (x - C(2))\tfrac{C'(3)-C'(2)}{C(3)-C(2)} + C'(2) \\ &\cdots \\ x \geq C(n) &\implies x\pi = x - C(n) + C'(n) \end{aligned}$$

Note that since $\frac{C'(i)-C'(i-1)}{C(i)-C(i-1)} > 0$ for any $1 < i \leq n$, π is monotone. Furthermore, its inverse is given by:

$$\begin{aligned} x < C'(1) &\implies x\pi^{-1} = x - C'(1) + C(1) \\ x \geq C'(1) \wedge x < C'(2) &\implies x\pi^{-1} = (x - C'(1))\tfrac{C(2)-C(1)}{C'(2)-C'(1)} + C(1) \\ x \geq C'(2) \wedge x < C'(3) &\implies x\pi^{-1} = (x - C'(2))\tfrac{C(3)-C(2)}{C'(3)-C'(2)} + C(2) \\ &\cdots \\ x \geq C'(n) &\implies x\pi^{-1} = x - C'(n) + C(n) \end{aligned}$$

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

Lemma 5.3. *Given a finite subset $C \subset \mathbb{Q}$, we have $[C, I]^{ec} \cong \mathcal{P}_{\#C}(\mathbb{Q})$.*

Proof. From Lemma 5.2 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 4.1 gives a subgroup $S \leq G|_C$ such that $\mathcal{P}_{\#C}(\mathbb{Q}) \cong [C, S]^{ec}$. Since $S \leq G|_C$, Lemma 5.1 implies $S = I$. This proves that $[C, I]^{ec} \cong \mathcal{P}_{\#C}(\mathbb{Q})$. □

By Theorem 4.1 and the above lemmas, we can represent an orbit by a single integer n, the size of the least support of its elements.

Corollary 5.4. *Let X be an orbit-finite nominal set. Then $X \cong \mathcal{P}_{\dim(X)}(\mathbb{Q})$.*

Proof. By Theorem 4.1, we get C and $S \leq G|_C$ such that $X \cong [C, S]^{ec}$. By Lemma 5.1, $S = I$, and by Lemma 5.3 we get

$$X \cong [C, I]^{ec} \cong \mathcal{P}_{\#C}(\mathbb{Q})\,.$$

But $\#C = \dim(X)$, since C is the least support of some element $x \in X$. □

This naturally extends to (orbit-finite) nominal sets with multiple orbits by using a multiset of natural numbers, representing the size of the least support of each of the orbits. These multisets are formalised here as functions $f \colon \mathbb{N} \to \mathbb{N}$.

Definition 5.5. Given a function $f \colon \mathbb{N} \to \mathbb{N}$, we define a nominal set $[f]^o$ by

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

Proposition 5.6. *For every orbit-finite nominal set X, there is a unique 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.*

Proof. We start by proving the existence. 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 Corollary 5.4, we have $O_{i,j} \cong \{j\} \times \mathcal{P}_i(\mathbb{Q})$ for each orbit $O_{i,j}$.

Define $f \colon \mathbb{N} \to \mathbb{N}$ such that $f(i) = k_i$. Then $\{n \mid f(n) \neq 0\}$ is finite, since X is orbit-finite. 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 need to show that f is unique. Suppose $g \colon \mathbb{N} \to \mathbb{N}$ also represents X, e.g. $X \cong [g]^o$. Then it follows that $[f]^o \cong [g]^o$. Let $h \colon [f]^o \to [g]^o$ 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 $f = g$, proving that f is unique. □

Example 5.7. Consider the set $\mathbb{Q} \times \mathbb{Q}$. The elements (a, b) split in three orbits, one for $a < b$, one for $a = b$ and one for $a > b$. These have dimension 2, 1 and 2 respectively, so the set $\mathbb{Q} \times \mathbb{Q}$ is represented by the multiset $\{1, 2, 2\}$.

Remark 5.8. The presentation in terms of a function $f\colon \mathbb{N} \to \mathbb{N}$ enforces that there are only finitely many orbits of any given dimension. The first part of the above proposition generalises to arbitrary nominal sets by replacing the codomain of f by the class of all sets and adapting Definition 5.5 accordingly. However, the resulting correspondence will no longer be one-to-one.

5.2. Equivariant maps

We show how to represent equivariant maps, using two basic properties. Let $f\colon X \to Y$ be an equivariant map. The first property is that the direct image of an orbit (in X) is again an orbit (in Y), that is to say, f is defined 'orbit-wise'. Second, equivariant maps cannot introduce new elements in the support (but they can drop them). More precisely:

Lemma 5.9. *Let $f\colon X \to Y$ be an equivariant map, and $O \subseteq X$ a single orbit. The direct image $f(O) = \{f(x) \mid x \in O\}$ 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 5.10. *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 a 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. □

Hence, equivariant maps are fully determined by associating two pieces of information for each orbit in the domain: the orbit on which it is mapped and a string denoting which elements of the least support of the input are preserved. These ingredients are formalised in the first part of the following definition. The second part describes how these ingredients define an equivariant function. Proposition 5.12 then states that every equivariant function can be described in this way.

Definition 5.11. 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 $f_H\colon \bigcup I_i \to \bigcup O_i$ as the unique equivariant function such that, given $x \in I_i$ with least support C, $f_H(x)$ is the unique 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 $C(j)$ is the j-th smallest element of C.

Proposition 5.12. *For every equivariant map $f\colon X \to Y$ between orbit-finite nominal sets X and Y there exists a unique set H as in Definition 5.11 such that $f = f_H$.*

Proof. We start with showing existence 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 5.9, $f(I_i) = O_i$. 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 support $C'_i \cap C_i$. By Lemma 5.10, $C'_i \cap C_i = C'_i$, 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 $f = f_H$.

To show that H is unique, consider an H' such that $f = f_{H'}$. As a consequence, we have $f_H = f_{H'}$. From the definition of orbits it follows immediately that the split of X into I_i is essentially unique, and that we can label the tuples (I'_i, F'_i, O'_i) in H' such that $I'_i = I_i$. It then follows that $O_i = f_H(I_i) = f_{H'}(I'_i) = O'_i$. To show $F_i = F'_i$, consider an $x \in I_i$. Let C denote the least support of x, and C_f the least support of $f(x)$. By definition of f_H and $f_{H'}$, it follows that $\{C(j) \mid F_i(j) = 1\} = C_x = \{C(j) \mid F'_i(j)$. But this is only possible if $F_i = F'_i$, and hence $H = H'$. □

Example 5.13. Consider the function $\min\colon \mathcal{P}_3(\mathbb{Q}) \to \mathbb{Q}$ which returns the smallest element of a 3-element set. Note that both $\mathcal{P}_3(\mathbb{Q})$ and $\mathbb{Q}$ are single orbits. Since for the orbit $\mathcal{P}_3(\mathbb{Q})$ we only keep the smallest element of the support, we can thus represent the function min with $\{(\mathcal{P}_3(\mathbb{Q}), 100, \mathbb{Q})\}$.

Example 5.14. Consider the (right) projection $\pi_2\colon \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$. Recall from Example 5.7 that the set $\mathbb{Q} \times \mathbb{Q}$ has three orbits $Q_1 = \{(a, b) \mid a < b\}$, $Q_2 = \{(a, b) \mid a = b\}$ and $Q_3 = \{(a, b) \mid a > b\}$. The function π_2 is represented by $\{(Q_1, 01, \mathbb{Q}), (Q_2, 1, \mathbb{Q}), (Q_3, 10, \mathbb{Q})$.

5.3. Products

The product $X \times Y$ of two nominal sets is again a nominal set and hence, it can be represented itself in terms of the dimension of each of its orbits as shown in Section 5.1. However, this approach has some disadvantages.

Example 5.15. We start by showing that the orbit structure of products can be non-trivial. Consider the product of $X = \mathbb{Q}$ and the set $Y = \{(a, b) \in \mathbb{Q}^2 \mid a < b\}$. This product consists of five orbits, more than one might naively expect from the fact that both sets are single-orbit:

$$\begin{array}{ll}
\{(a,(b,c)) \mid a,b,c \in \mathbb{Q}, a < b < c\}, & \{(a,(a,b)) \mid a,b \in \mathbb{Q}, a < b\}, \\
\{(b,(a,c)) \mid a,b,c \in \mathbb{Q}, a < b < c\}, & \{(b,(a,b)) \mid a,b \in \mathbb{Q}, a < b\}, \\
\{(c,(a,b)) \mid a,b,c \in \mathbb{Q}, a < b < c\}. &
\end{array}$$

We find that this product is represented by the multiset $\{2, 2, 3, 3, 3\}$. Unfortunately, this is not sufficient to accurately describe the product as it abstracts away from the relation between its elements with those in X and Y. In particular, it is not possible to reconstruct the projection maps from such a representation.

The essence of our representation of products is that each orbit O in the product $X \times Y$ is described entirely by the dimension of O together with the two (equivariant) projections $\pi_1 \colon O \to X$ and $\pi_2 \colon O \to Y$. This combination of the orbit and the two projection maps can already be represented using Propositions 5.6 and 5.12. However, as we will see, a combined representation for this has several advantages. For discussing such a representation, let us first introduce what it means for tuples of a set and two functions to be isomorphic:

Definition 5.16. Given nominal sets X, Y, Z_1 and Z_2, and equivariant functions $l_1 \colon Z_1 \to X$, $r_1 \colon Z_1 \to Y$, $l_2 \colon Z_2 \to X$ and $r_2 \colon Z_2 \to Y$, we define $(Z_1, l_1, r_1) \cong (Z_2, l_2, r_2)$ if there exists an isomorphism $h \colon Z_1 \to Z_2$ such that $l_1 = l_2 \circ h$ and $r_1 = r_2 \circ h$.

Our goal is to have a representation that, for each orbit O, produces a tuple (A, f_1, f_2) isomorphic to the tuple (O, π_1, π_2). The next lemma gives a characterisation that can be used to simplify such a representation.

Lemma 5.17. *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 $\forall c \in C_x \cup C_y, c\pi = c$. Then $(x, y)\pi = (x\pi, y\pi) = (x, y)$ since C_x and C_y are supports of x and y respectively. Hence $C_x \cup C_y$ is a support of x, and since C is the least support of x, $C \subseteq C_x \cup C_y$.

Now suppose that C is strictly smaller than $C_x \cup C_y$. Then there is an element $c \in C_x \cup C_y$ with $c \notin C$. Without loss of generality we can assume $c \in C_x$. The set $(C_x \cup C_y) \setminus \{c\}$ is *not* a support of x, since the least support C_x of x is not contained in $(C_x \cup C_y) \setminus \{c\}$. Hence, there is some $\pi \in G$ such that $\forall c' \in (C_x \cup C_y) \setminus \{c\}, c'\pi = c'$, but $x\pi \neq x$. For this π, we have $(x, y)\pi = (x\pi, y\pi) \neq (x, y)$. However, $(C_x \cup C_y) \setminus \{c\}$ is a support of (x, y), since $C \subseteq (C_x \cup C_y) \setminus \{c\}$. Hence $(x, y)\pi = (x, y)$, yielding a contradiction. □

With Proposition 5.12 we represent the maps π_1 and π_2 by tuples (O, F_1, O_1) and (O, F_2, O_2) respectively. Using Lemma 5.17 and the definitions of F_1 and F_2, we see that at least one of $F_1(i)$ and $F_2(i)$ equals 1 for each i.

We can thus combine the strings F_1 and F_2 into a single string $P \in \{L, R, B\}^*$ as follows. We set $P(i) = L$ when only $F_1(i)$ is 1, $P(i) = R$ when only $F_2(i)$ is 1, and $P(i) = B$ when both are 1. The string P fully describes the strings F_1 and F_2. This process for constructing the string P gives it two useful properties:

- The number of Ls and Bs in the string P equals the dimension of O_1.
- The number of Rs and Bs in the string P equals the dimension of O_2.

We will call strings P with the above two properties *valid* (with respect to O_1, O_2).

Thus, to describe a single orbit of the product $X \times Y$, a valid string P together with the images of π_1 and π_2 is sufficient. This is stated more precisely in Proposition 5.20.

Definition 5.18. Let $O_1 \subseteq X$, $O_2 \subseteq Y$ be single-orbit sets, and let $P \in \{L, R, B\}^*$ be a valid string with respect to O_1, O_2. Define

$$[(P, O_1, O_2)]^t = (\mathcal{P}_{|P|}(\mathbb{Q}), f_{H_1}, f_{H_2}),$$

where $H_i = \{(\mathcal{P}_{|P|}(\mathbb{Q}), F_i, O_i)\}$ and the string F_1 is defined as the string P with Ls and Bs replaced by 1s and Rs by 0s. The string F_2 is similarly defined with the roles of L and R swapped.

This construction generates orbits of $X \times Y$:

Lemma 5.19. *Let (P, O_1, O_2) be a tuple as in Definition 5.18. Then we have $[(P, O_1, O_2)]^t \cong (O, \pi_1, \pi_2)$ for some orbit $O \subseteq X \times Y$.*

Proof. Let $(O', f, g) = [(P, O_1, O_2)]^t$. By construction, we find $f(O') \subseteq X$ and $g(O') \subseteq Y$. Denote by $\langle f, g\rangle \colon O' \to X \times Y$ the pairing, i.e., $\langle f, g\rangle(x) = (f(x), g(x))$. By Lemma 5.9, since O' is single-orbit, so is $\langle f, g\rangle(O')$. The latter is an orbit of $X \times Y$.

We now show that $\langle f, g\rangle$ is an isomorphism. First, 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 5.17, 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 $\langle f, g\rangle(O')$ have the same support size. Since both O' and $\langle f, g\rangle(O')$ are single-orbit, this makes $\langle f, g\rangle$ a bijection. Hence,

$$[(P, O_1, O_2)]^t = (O', f, g) \cong (\langle f, g\rangle(O'), \pi_1|_{\langle f,g\rangle(O')}, \pi_2|_{\langle f,g\rangle(O')}) .$$

□

The following result shows that every orbit of $X \times Y$ arises in this way, up to isomorphism.

Proposition 5.20. *For every orbit $O \subseteq X \times Y$ there is a unique tuple (P, O_1, O_2) such that $O_1 \subseteq X$, $O_2 \subseteq Y$ are orbits, P is a valid string and*

$$[(P, O_1, O_2)]^t \cong (O, \pi_1|_O, \pi_2|_O) .$$

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 5.12 provides two tuples (O, F_1, O_1) and (O, F_2, O_2) with O_1 an orbit of X and O_2 an orbit of Y such that $\pi_1|_O = f_{\{(O,F_1,O_1)\}}$ and $\pi_2|_O = f_{\{(O,F_2,O_2)\}}$. Now construct P as the sequence of length $\dim(O)$ as follows:

$$P(i) = \begin{cases} L & \text{if } F_1(i) = 1 \text{ and } F_2(i) = 0 \\ R & \text{if } F_1(i) = 0 \text{ and } F_2(i) = 1 \\ B & \text{if } F_1(i) = F_2(i) = 1 \end{cases}$$

This covers all cases, since Lemma 5.17 guarantees that it will never be the case that the i-th letters of F_1 and F_2 are both 0.

We first show that P is valid. To see this, observe that by Definition 5.11, each 1 in the string F_1 corresponds to a unique element in the least support of an element of O_1. Hence $\dim(O_1) = |F_1|$. By definition of F_1 we find $\dim(O_1) = |F_1| = |P|_L + |P|_B$. A similar line of reasoning, replacing L with R, shows $\dim(O_2) = |F_2| = |P|_R + |P|_B$.

To show that the correspondence is bijective, we first show that, given an orbit $O \subseteq X \times Y$, if (P, O_1, O_2) is the corresponding triple then

$$(O, \pi_1|_O, \pi_2|_O) \cong [(P, O_1, O_2)]^t\,.$$

Let $n = \dim(O)$. By Corollary 5.4, we have $\mathcal{P}_n(\mathbb{Q}) \cong O$. Let $g \colon O \overset{\cong}{\to} \mathcal{P}_n(\mathbb{Q})$ be the isomorphism between them. Then $f_{\{(O,F_1,O_1)\}} = f_{\{(\mathcal{P}_{|F_1|}(\mathbb{Q}),F_1,O_1)\}} \circ g$ and $f_{\{(O,F_2,O_2)\}} = f_{\{(\mathcal{P}_{|F_2|}(\mathbb{Q}),F_2,O_2)\}} \circ g$. As a consequence, we have

$$(O, \pi_1|_O, \pi_2|_O) \cong (\mathcal{P}_{|F_1|}(\mathbb{Q}), f_{\{(\mathcal{P}_{|F_1|}(\mathbb{Q}),F_1,O_1)\}}, f_{\{(\mathcal{P}_{|F_2|}(\mathbb{Q}),F_2,O_2)\}})\,.$$

By Definition 5.18 and the above construction of P, the right-hand side of the above equation equals $[(P, O_1, O_2)]^t$. Hence, $(O, \pi_1|_O, \pi_2|_O) \cong [(P, O_1, O_2)]^t$.

Finally, for uniqueness, consider two tuples (P, O_1, O_2) and (P', O'_1, O'_2), such that $[(P, O_1, O_2)]^t \cong [(P', O'_1, O'_2)]^t$. We let F_1 and F_2 denote the strings from Definition 5.18 for $[(P, O_1, O_2)]^t$, and similarly F'_1 and F'_2 the strings for $[(P', O'_1, O'_2)]^t$.

Since $[(P, O_1, O_2)]^t \cong [(P', O'_1, O'_2)]^t$ we have $\mathcal{P}_{|P|}(\mathbb{Q}) \cong \mathcal{P}_{|P'|}(\mathbb{Q})$, hence $|P| = |P'|$. Furthermore, by the isomorphism, for any $x \in \mathcal{P}_{|P|}(\mathbb{Q})$, there exists an $x' \in \mathcal{P}_{|P'|}(\mathbb{Q})$ such that

$$\begin{aligned} f_{\{(\mathcal{P}_{|P|}(\mathbb{Q}),F_1,O_1)\}}(x) &= f_{\{(\mathcal{P}_{|P'|}(\mathbb{Q}),F'_1,O'_1)\}}(x')\,, \text{ and} \\ f_{\{(\mathcal{P}_{|P|}(\mathbb{Q}),F_2,O_2)\}}(x) &= f_{\{(\mathcal{P}_{|P'|}(\mathbb{Q}),F'_2,O'_2)\}}(x')\,. \end{aligned} \tag{1}$$

Since O_1, O_2, O'_1 and O'_2 single-orbit, this implies $O_1 = O'_1$ and $O_2 = O'_2$.

Moreover, by Lemma 5.17, the least support of any element $x \in \mathcal{P}_{|P|}(\mathbb{Q})$ equals the least support of $(f_{\{(\mathcal{P}_{|P|}(\mathbb{Q}),F_1,O_1)\}}(x), f_{\{(\mathcal{P}_{|P|}(\mathbb{Q}),F_2,O_2)\}}(x))$. But if we choose x' corresponding to x as in the previous paragraph, then by (1) we obtain that the least support of x equals the least support of x'. Hence $x = x'$. But this implies that $f_{\{(\mathcal{P}_{|P|}(\mathbb{Q}),F_1,O_1)\}} = f_{\{(\mathcal{P}_{|P'|}(\mathbb{Q}),F'_1,O_1)\}}$ and $f_{\{(\mathcal{P}_{|P|}(\mathbb{Q}),F_2,O_2)\}} = f_{\{(\mathcal{P}_{|P'|}(\mathbb{Q}),F'_2,O_2)\}}$, hence also $F_1 = F'_1$ and $F_2 = F'_2$. But $F_1 = F'_1$ and $F_2 = F'_2$ can hold only if P and P' are equal. From this, we conclude that $(P, O_1, O_2) = (P', O'_1, O'_2)$. □

From the above proposition it follows that we can generate the product $X \times Y$ simply by enumerating all valid strings P for all pairs of orbits (O_1, O_2) of X and Y. Given this, we can calculate the multiset representation of a product from the multiset representations of both factors.

Theorem 5.21. *For $X \cong [f]^o$ and $Y \cong [g]^o$ we have $X \times Y \cong [h]^o$, where*

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

Proof. Every string $P \in \{L, R, B\}^*$ of length n with $|P|_L = n - j$, $|P|_R = n - i$ and $|P|_B = i + j - n$ satisfies the requirements of Lemma 5.20, 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 $i \geq 0$, $j \geq 0$ 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. □

Example 5.22. To illustrate some aspects of the above representation, let us use it to calculate the product of Example 5.15. First, we observe that both $\mathbb{Q}$ and $S = \{(a, b) \in \mathbb{Q}^2 \mid a < b\}$ consist of a single orbit. Hence any orbit of the product corresponds to a triple $(P, \mathbb{Q}, S)$, where the string P satisfies $|P|_L + |P|_B = \dim(\mathbb{Q}) = 1$ and $|P|_R + |P|_B = \dim(S) = 2$. We can now find the orbits of the product $\mathbb{Q} \times S$ by enumerating all strings satisfying these equations. This yields:

- LRR, corresponding to the orbit $\{(a, (b, c)) \mid a, b, c \in \mathbb{Q}, a < b < c\}$,
- RLR, corresponding to the orbit $\{(b, (a, c)) \mid a, b, c \in \mathbb{Q}, a < b < c\}$,
- RRL, corresponding to the orbit $\{(c, (a, b)) \mid a, b, c \in \mathbb{Q}, a < b < c\}$,
- RB, corresponding to the orbit $\{(b, (a, b)) \mid a, b \in \mathbb{Q}, a < b\}$, and
- BR, corresponding to the orbit $\{(a, (a, b)) \mid a, b \in \mathbb{Q}, a < b\}$.

Each product string fully describes the corresponding orbit. To illustrate, consider the string BR. The corresponding bit strings for the projection functions are $F_1 = 10$ and $F_2 = 11$. From the lengths of the string we conclude that the dimension of the orbit is 2. The string F_1 further tells us that the left element of the tuple consists only of the smallest element of the support. The string F_2 indicates that the right element of the tuple is constructed from both elements of the support. Combining this, we find that the orbit is $\{(a, (a, b)) \mid a, b \in \mathbb{Q}, a < b\}$.

5.4. Summary

We summarise our concrete representation in the following table. Propositions 5.6, 5.12 and 5.20 correspond to the three rows in the table.

Notice that in the case of maps and products, the orbits are inductively represented using the concrete representation. As a base case we can represent single orbits by their dimension.

6. C++ Implementation of Ons

The ideas outlined above have been implemented in the C++ library Ons.[6] The library can represent orbit-finite nominal sets and their products, (disjoint)

[6] Ons can be found at `https://github.com/davidv1992/ONS`

Object	*Representation*
Single orbit O	Natural number $n = \dim(O)$
Nominal set $X = \bigcup_i O_i$	Multiset of these numbers
Map from single orbit $f\colon O \to Y$	The orbit $f(O)$ and a bit string F
Equivariant map $f\colon X \to Y$	Set of tuples $(O, F, f(O))$, one for each orbit
Orbit in a product $O \subseteq X \times Y$	The corresponding orbits of X and Y, and a string P relating their supports
Product $X \times Y$	Set of tuples (P, O_X, O_Y), one for each orbit

Table 1: Overview of representation.

unions, and maps. A full technical description of what it can do, and how to use it, is given in the documentation included with Ons.

Let us start here by showing an example program to calculate the product of the sets $\{(a, b) \mid a < b\}$ and $\mathbb{Q}$ (see Example 5.22). The program below calculates this product, and then prints one element for each orbit of 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() << "␣";
```

In the first line, we create a nominal set A, and initialize it with the built-in set $\mathbb{Q}$. The type of such a nominal set variable is `nomset<T>`, where T is the type of the elements. In case of A this is the `rational` type.

In the second line, we create the set B containing the elements of $\{(a, b) \mid a < b\}$. To do this, we instruct the constructor of B to create the minimal nominal set containing the element $(1, 2)$. This creates the nominal set with the orbit of $(1, 2)$, which is exactly the set $\{(a, b) \mid a < b\}$.

Having created these sets, it then computes the product using the function `nomset_product`. This returns the product, of type `nomset<pair<A,B>>`, where `A` and `B` are the types of the elements of A and B respectively. In our case, this means that the result is a nominal set containing elements of type `pair<rational,pair<rational,rational>>`.

Finally, we loop over the orbits of the result, stored in the variable `AtimesB`, with `for (auto orbit : AtimesB)`. This returns an `orbit` object for each orbit in the nominal set `AtimesB`. These objects describe the properties of the individual orbits of a nominal set. We use it here to get an element (with `.getElement()`), which is printed through standard out.

Running this code gives the following output (‘`/1`’ signifies the denominator):

```
(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))
```

We see here a list of five elements, each corresponding to a single orbit of the product $\mathbb{Q} \times \{(a, b) \mid a < b\}$.

6.1. Core functionality

The main implementation of nominal sets and equivariant functions in the Ons library is split up in three main concepts:

1. `orbit<T>`, representing orbits;
2. `nomset<T>`, representing nominal sets, containing a number of `orbit<T>` objects;
3. `eqimap`, equivariant functions.

The `orbit<T>` objects contain a complete description of a single orbit of elements of type T. They allow querying of basic properties such as the size of the least support (with `.supportSize`), checking whether an element is a member of the orbit (with `.isElement`), and extraction of sample elements (with `.getElement`).

Next, `nomset<T>` is used to represent entire sets. Functionality includes checking whether an element or other set is contained in the set (with `.contains`), iterating over the orbits (see above), and querying for the size of the set (with `.size`).

For working with nominal sets, Ons also provides implementations of common set operations. Examples of these are set union (with `nomset_union`), intersection (with `nomset_intersect`), and set products (with `nomset_product`).

The Ons library also implements support for filtering (with `nomset_filter`) and mapping (with `nomset_map`) of nominal sets. These take an (equivariant) function as argument, which can either be given as an equivariant function object `eqimap` or as a `C++` function or function object, as long as the resulting behaviour when invoked is equivariant.

Finally, objects of type `eqimap` can be used to represent dynamically generated equivariant functions. They implement an evaluation, allowing the application of the function to concrete argument values. They also contain several functions for querying properties of the function represented (such as whether elements are in its domain, with `.inDomain`), and manipulating the function represented (such as extending the mapping, with `.add`).

6.2. Another example

Let us now consider a slightly more complicated piece of code. It refines a relation R (called `previousPartition`) on the states Q of an automaton, using a transition function $f\colon Q \times A \to A$ (called `transitionFunction`), over an alphabet A (called `alphabet`). It returns as a result the set

$$\{(q, q') \in R \mid \forall a \in A\colon (f(q, a), f(q', a)) \in R\}\,.$$

The code is as follows.

```
template <typename Q, typename A>
nomset<pair<Q,Q>> refineRelation(
    nomset<A> alphabet,
    nomset<pair<Q,Q>> previousRelation,
    eqimap<pair<Q,A>, Q> transitionFunction) {

    // calculate R x A
    nomset<pair<pair<Q,Q>,A>> transitions =
        nomset_product(previousRelation, alphabet);

    // Find those where (f(q,a), f(q',a)) not in R
    nomset<pair<pair<Q,Q>,A>> invalid =
        nomset_filter(transitions, [&](pair<pair<Q,Q>,A> input) {
            Q state1 = input.first.first;
            Q state2 = input.first.second;
            A letter = input.second;

            Q result1 = transitionFunction({state1, letter});
            Q result2 = transitionFunction({state2, letter});

            return !previousRelation.contains({result1, result2});
        });

    // Strip away alphabet
    nomset<pair<Q,Q>> toRemove =
        nomset_map(invalid, [](pair<pair<Q,Q>,A> input) {
            return input.first;
        });

    // Calculate result
    return nomset_minus(previousRelation, toRemove);
}
```

This result is computed in four steps. First, it calculates the set of all transition pairs it still needs to consider (transitions), using `nomset_product` to calculate the product $R \times A$.

Next, it uses `nomset_filter` to select those elements $((q, q'), a)$ for which $(f(q, a), f(q', a)) \notin R$. Note that the function which calculates this is specified as a plain C++ lambda, whose behaviour is equivariant.

For modifying the relation, only the first part of $((q, q'), a)$ is relevant, so in the third step we use `nomset_map` to project out the alphabet letters of the witnesses in the set invalid.

This leaves the code with a set of pairs (q, q') which need to be removed from R to produce the final result. This is calculated using the set minus operation `nomset_minus`, resulting in the refined partition.

7. Haskell Implementation of Ons

We have implemented a similar library in Haskell, called Ons-hs.[7] This showcases the generality of the theoretical characterisation in Section 5.

At the core, there is the type class *Nominal*.

```
class Nominal a where
  type Orbit a :: *
  toOrbit :: a -> Orbit a
  getElement :: Orbit a -> Support -> a
  support :: a -> Support
  index :: Proxy a -> Orbit a -> Int
```

It provides to basic functionality, toOrbit and getElement, to convert between elements (of type a) and orbits of elements (of type Orbit a). The functions support and index are utility functions, returning the (least) support of an element, and the dimension of an orbit.

Instances are defined for basic data types such as the type of rational numbers, Atom. Other instances can be derived for any algebraic data structure (following Table 1). For example, the data type for the states of the automaton in Figure 1 can be defined as:

```
data State = Q0 | Q1 Atom | Q2 Atom Atom | Q3 Atom Atom | Q4
  deriving (Eq, ...)
  deriving Nominal via Generic State
```

Deriving instances with generics make it easy for the user to use the library. One can also easily define trivial instances, where the group action is defined as the identity function. This is used for the EquivariantSet data structure. This data type provides an interface to (infinite) nominal sets and the usual set constructions are defined. Some of these functions are shown below (we have omitted the type class context from the type signatures).

```
data EquivariantSet a = ...
  deriving Nominal via Trivial (EquivariantSet a)

map :: (a -> b) -> EquivariantSet a -> EquivariantSet b
filter :: (a -> Bool) -> EquivariantSet a -> EquivariantSet a
product :: EquivariantSet a -> EquivariantSet b -> EquivariantSet (a, b)
rationals :: EquivariantSet Atom
```

Function arguments (in, e.g., map and filter) are required to be equivariant. Finally, a data type for equivariant maps, EquivariantMap is provided with the expected functions for a map data structure.

With these functions, we can define all the states of the automaton in Figure 1 and the accepting states can easily be filtered out.

[7] Available at `https://gitlab.science.ru.nl/moerman/ons-hs`

```
states = fromList [Q0, Q4] <> map Q1 rationals
  <> map (uncurry Q2) (product rationals rationals)
  <> map (uncurry Q3) (product rationals rationals)
acceptingStates = filter accept states where
  accept (Q2 a b) = a < b
  accept _ = False
```

8. Complexity of set operations

Since our implementations are directly based on representing orbits, it is possible to derive concrete complexities for the set operations. To simplify such an analysis, we make the following assumptions on operations on orbits:

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

These assumptions are justified as each of these operations takes time proportional to the size of the representation of an individual orbit, which in practice is small and approximately constant. For instance, the orbit $\mathcal{P}_n(\mathbb{Q})$ is represented by just the integer n and its type.

Furthermore, two of the operations considered make use of an external function. Since these can be implemented in a variety of ways, and the time complexity of actually invoking these functions is highly dependent both on what it calculates, and the specific way it is implemented in the program, we will here simply consider invocations of these functions to take $O(1)$ time.

For the notation in the following statement, recall that $\mathrm{N}(X)$ denotes the number of orbits of X, and $\dim(X)$ the maximal size of the least support of its elements.

Theorem 8.1. *If nominal sets are implemented with a tree-based set structure (as in* Ons*), the complexity of the following set operations 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 $X \times Y$	$O(\mathrm{N}(X \times Y)) \subseteq O(3^{\dim(X)+\dim(Y)}\,\mathrm{N}(X)\,\mathrm{N}(Y))$

The functions $p\colon X \to 2$ and $f\colon X \to Y$ are user defined, and assumed to take $O(1)$ time per invocation.

Proof. Since most parts are proven similarly, we only include proofs for the first and last item.

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 $O(\log(\mathrm{N}(X)))$.

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 generating these orbits 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. Recall that orbits in a product are represented by strings of length at most $\dim(X) + \dim(Y)$. (If the string is shorter, we pad it with one of the symbols.) Since there are three symbols (L, R and B), the product of X and Y will have at most $3^{\dim(X)+\dim(Y)}\,\mathrm{N}(X)\,\mathrm{N}(Y)$ orbits. It follows that taking products has time complexity of $O(3^{\dim(X)+\dim(Y)}\,\mathrm{N}(X)\,\mathrm{N}(Y))$. □

Using the above complexity results on individual operations, we can derive the complexity of algorithms using nominal sets. We will demonstrate this here using Moore's algorithm. Recall from Section 3.1:

Algorithm 2 Moore's minimisation algorithm for nominal DFAs

```
Require: Nominal automaton $(S, A, F, \delta)$.
1: $i \leftarrow 0$, $\equiv_{-1} \leftarrow S \times S$, $\equiv_0 \leftarrow F \times F \cup (S \backslash F) \times (S \backslash F)$
2: while $\equiv_i \neq \equiv_{i-1}$ do
3:     $\equiv_{i+1} = \{(q_1, q_2) \mid (q_1, q_2) \in \equiv_i \wedge \forall a \in A, (\delta(q_1, a), \delta(q_2, a)) \in \equiv_i\}$
4:     $i \leftarrow i + 1$
5: end while
6: $E \leftarrow S/_{\equiv_i}$
7: $F_E \leftarrow \{e \in E \mid \forall s \in e, s \in F\}$
8: Let $\delta_E$ be the map such that, if $s \in e$ and $\delta(s, a) \in e'$, then $\delta_E(e, a) = e'$.
9: return $(E, A, F_E, \delta_E)$.
```

Theorem 8.2. *The runtime complexity of Moore's algorithm on nominal deterministic automata is* $O(3^{5k} k\,\mathrm{N}(S)^3\,\mathrm{N}(A))$*, where* $k = \dim(S \cup A)$.

Proof. This is shown by counting operations, using the complexity results of set operations stated in Theorem 8.1. We first focus on the while loop on lines 2 through 5. The runtime of an iteration of the loop is determined by line 3, as this is the most expensive step. Since the dimensions of S and A are at most k, computing $S \times S \times A$ takes $O(\mathrm{N}(S)^2\,\mathrm{N}(A)3^{5k})$. Filtering $S \times S$ using that then takes $O(\mathrm{N}(S)^2 3^{2k})$. The time to compute $S \times S \times A$ dominates, hence each iteration of the loop takes $O(\mathrm{N}(S)^2\,\mathrm{N}(A)3^{5k})$.

Next, we need to count the number of iterations of the loop. 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 S 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 $\mathrm{N}(S) - 1$ 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 S 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 S. It follows that there are at most $O(k\,\mathrm{N}(S))$ iterations of the loop, giving the loop a complexity of $O(k\,\mathrm{N}(S)^3\,\mathrm{N}(A)3^{5k})$

The remaining operations outside the loop have a lower complexity than that of the loop, hence the complexity of Moore's minimisation algorithm for a nominal automaton is $O(k\,\mathrm{N}(S)^3\,\mathrm{N}(A)3^{5k})$. □

The above theorem shows in particular that minimisation of nominal automata is fixed-parameter tractable (FPT) with the dimension as fixed parameter. The complexity of Algorithm 1 for nominal automata is very similar to the $O((\#S)^3\#A)$ bound given by a naive implementation of Moore's algorithm for ordinary DFAs. This suggest that it is possible to further optimise an implementation with similar techniques used for ordinary automata.

9. Evaluation

This section presents an experimental evaluation of Ons and Ons-hs, comparing them against existing tools in two tasks related to nominal automata: learning and minimisation.

9.1. The Tools: Ons, Ons-hs, Nλ, and Lois

In order to evaluate our library Ons(-hs), we compare it against two existing libraries for computing with nominal sets, Nλ [18] and Lois [19, 20]. We briefly describe how these tools work and what the differences are with Ons.

Both Nλ and Lois work symbolically. Nominal sets are represented with set-builder expressions: values with variables and a first-order formula describing those values. For example, the orbit $\{\{a, b, c\} \mid a, b, c \in \mathbb{Q}, a < b < c\}$ is represented simply as:

$$\{\mathsf{fromList}\ [x_0, x_1, x_2] \mid x_0 < x_1 \wedge x_1 < x_2\}.$$

(Here fromList takes a list and constructs a set.) In Ons, orbits are represented more compactly: in this case, the integer 3 suffices. On the other hand, a set such as $\mathbb{Q}^3$ has a compact representation in Nλ and Lois:

$$\{(x_0, x_1, x_2) \mid \top\}$$

and a larger representation in Ons, consisting of 13 strings (see Section 5.3).

Since Nλ and Lois use formulas, many set operations are expressed by manipulating these formulas. One of the crucial operations is determining whether a set is empty, which can be resolved by checking satisfiability of the formula. To this end, these libraries use an SMT solver (by default, both libraries use Z3 [25]). Consequently, the runtime will depend on the size of these formulas, and both libraries have routines to simplify formulas.

9.2. Benchmarks

We evaluate the scalability of each library by implementing the automata minimisation algorithm and learning algorithm discussed in Section 3. These are then tested on the following three sets of automata.

Structured automata. We define the following automata.

FIFO(n) Automata accepting valid traces of a finite FIFO data structure of size n. The alphabet is defined by two orbits: $\{\text{Put}(a) \mid a \in \mathbb{Q}\}$ and $\{\text{Get}(a) \mid a \in \mathbb{Q}\}$.

$ww(n)$ Automata accepting the language of words of the form ww, where $w \in \mathbb{Q}^n$.

$\mathcal{L}_{\text{max}}$ The language $\mathcal{L}_{\text{max}}$ where the last symbol is the maximum of previous symbols (Example 3.4).

$\mathcal{L}_{\text{int}}$ The language accepting a series of nested intervals (Example 3.1).

The first two classes of structured automata are used as test cases in [12]. These two classes are also equivariant w.r.t. the equality symmetry. The structured automata can be encoded directly in symbolic form in Nλ or Lois. In Ons, this structure is lost and the algorithms operate purely on orbits. Where applicable, the automata listed above were generated using the same code as used in [12], ported to the other libraries as needed.

The automaton accepting the FIFO language is not minimal. It is based on the purely functional queue which has two lists of data values: one for pushing, one for popping. If the list for popping is empty, the list for pushing is reversed and moved to the list for popping [26]. The use of two lists is redundant and hence the automaton is not minimal.

Orbit-wise random automata. Besides structured automata, we generate orbit-wise random automata as follows. The input alphabet is always $\mathbb{Q}$ and the number of orbits and dimension k of the state space S are fixed. For each orbit in the set of states, its dimension is chosen uniformly at random between 0 and k, inclusive. Each orbit has a probability $\frac{1}{2}$ of consisting of accepting states.

To generate the transition function δ, we enumerate the orbits of $S \times \mathbb{Q}$ and choose a target state uniformly from the orbits S 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 $\text{rand}_{\text{N}(S),k}$. The choices made here are arbitrary and only provide basic automata. We note that the automata are generated orbit-wise and this may favour our tool.

Random automata with formulae. The two classes above (orbit-wise random and structured) are very different in nature. The random ones are defined orbit-wise (which is an advantage for Ons), whereas the structured ones hardly use the values in an interesting way. To provide a middle-ground, we also generate random automata which use formulas on transitions.

For these automata, the state space is constructed from multiple copies of $\mathbb{Q}^n$. We will refer to these copies as locations and we will refer to the number of locations in the state space as the size of the automaton. We fix the size of the automaton, and then for each location we sample its dimension n uniformly from $[0, k]$, where k is a chosen constant. This creates the set of states. Every location has a probability $\frac{1}{2}$ of being accepting. Note that $\mathbb{Q}^n$ consists of more than one orbit if $n > 1$, and consequently, the number of orbits in the state of the resulting automata can vary. The alphabet used is always the set $\mathbb{Q}$.

To generate the transition function, we generate a formula for each of the locations. This is done by creating a tree, where every node is one operation. The tree starts with an empty root, and is then expanded by repeatedly selecting one of the empty nodes and replacing it with either a logical operation ('and' or 'or') with two new empty nodes, or by a literal, i.e., a comparison between two variables (either $<$, $=$, or $>$). Both options occur with equal chance. In the latter case, the variables are drawn from either the state values or the input value. This process is repeated until either there are no more empty nodes, or the limit on the number of logical operators is reached, at which point the remaining empty nodes are filled with literals. Finally, for each node in the tree we randomly invert the output or not.

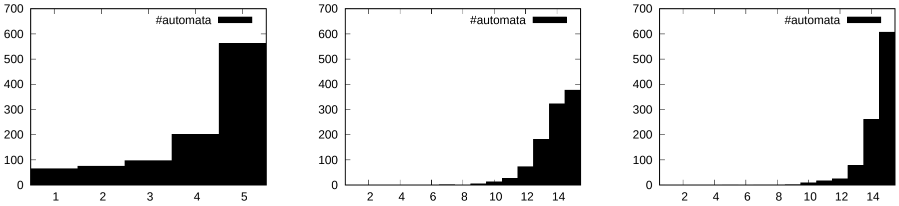
These formulas are then used to create two edges: one for when the formula is true, and one for when the formula is false. The target state is specified by randomly choosing a location, and randomly selecting which elements of the original state and input are kept in the target location (we allow for duplicate values).

Properties of the random automata. Our main motivation for using the above described random automata is the unavailability of a good source of nominal automata used in practical applications. However, this makes it difficult to judge whether or not our random automata are representative for actual performance in real test cases. We will show some properties of the generated automata so the reader can make their own judgement. In particular, we will focus primarily on the degree to which the size of the automata is reduced during minimisation.

For the orbit-wise generated automata, this is shown in Figure 5. It can be clearly seen that the vast majority of the generated automata is either minimal, or very close to being minimal.

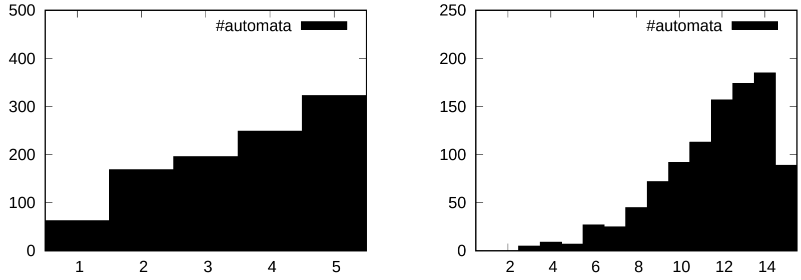
In contrast, the data for the formula automata (see Figures 6 and 7) shows a much broader distribution, generating automata that use significantly more orbits than needed for the recognised languages.

Given this, we think that our test cases are varied enough to give a decent representation of the performance of the various libraries.



(a) $N(S) = 5, k = 1$ (b) $N(S) = 15, k = 1$ (c) $N(S) = 15, k = 3$

Figure 5: Histogram of the number of orbits after minimisation for orbit-wise random automata. Each figure shows 1000 automata. The number of orbits before minimisation is $N(S)$ and the dimension is k.



(a) 5 locations, $k = 1$ (b) 15 locations, $k = 1$

Figure 6: Number of state orbits after minimisation for formula automata of size 5 and 15, of dimension 1. The figure shows 1000 automata. Note that because the dimension is 1, the number of orbits before minimisation is always 5 or 15 respectively.

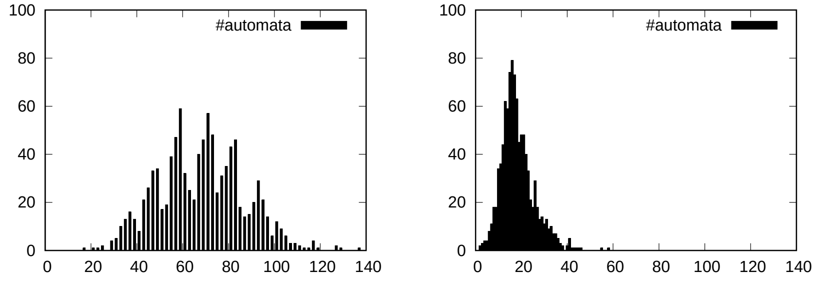


Figure 7: Number of state orbits before (left) and after (right) minimisation for formula automata of size 15 and dimension 3. Each figure shows 1000 automata. The striped pattern on the left is a consequence of the fact that the used settings only generate automata with an odd number of orbits.

Model	N	L	dim	Ons (s)	Ons-hs (s)	Nλ (s)	Lois (s)
Random	5		≤ 1	0.00	0.00	0.05	0.71
Random	10		≤ 1	0.00	0.00	0.93	12.31
Random	10		≤ 2	0.01	0.00	26.60	> 2h
Random	15		≤ 1	0.00	0.00	5.01	41.31
Random	15		≤ 2	0.01	0.00	61.98	> 2h
Random	15		≤ 3	0.04	0.02	418.22	> 2h
Formula		5	≤ 2	0.00	0.01	0.25	
Formula		10	≤ 2	0.01	0.02	1.52	
Formula		10	≤ 3	0.62	0.57	1.52	
Formula		25	≤ 2	0.06	0.05	48.81	
Formula		25	≤ 3	2.89	1.58	108.46	
Formula		25	≤ 5	> 2h	2176.83	255.39	
FIFO(1)	4	4	1	0.02	0.00	0.01	0.03
FIFO(2)	13	7	2	0.01	0.01	1.03	0.24
FIFO(3)	65	11	3	0.35	0.70	9.32	2.44
FIFO(4)	440	16	4	37.60	39.77	68.44	15.33
FIFO(5)	3686	22	5	> 2h	3027.54	382.33	71.59
ww_1	4	4	1	0.02	0.00	0.01	0.03
ww_2	8	6	2	0.00	0.00	0.12	0.03
ww_3	24	8	3	0.16	0.17	0.75	0.16
ww_4	112	10	4	23.71	30.51	2.85	0.60
ww_5	728	12	5	5880.04	> 2h	9.27	1.83
$\mathcal{L}_{\text{max}}$	5		2	0.02	0.00	1.08	0.06
$\mathcal{L}_{\text{int}}$	5		2	0.01	0.00	1.71	0.04

Table 2: Running times for Algorithm 1. N (dim) is the size (resp. dimension) of the input. The column L denotes the number of locations, if the automaton can be expressed symbolically. The first two sets of automata ('Random' and 'Formula') are the randomly generated automata. Each rows consists of 10 automata and we report the average runtime. If one of the runs times out, the cell indicates > 2h.

9.3. Minimisation Results

For Nλ and Lois we used the implementations of Moore's minimisation algorithm from the original papers [18, 19, 20]. For each of the libraries, we wrote routines to read in an automaton from a file and, for the structured test cases, to generate the requested automaton. For Ons, all automata were read from file. The output of these programs was manually checked to see if the minimisation was performed correctly.

The results in Table 2 show a clear advantage for Ons for random automata. The library is capable of running all supplied test cases in less than one second. This in contrast to both Lois and Nλ, which take more than 2 hours on the largest random automata.

The results for structured automata show a clear effect of the extra structure. Both Nλ and Lois remain capable of minimising 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.

The random automata with formulae show a mixed bag (as expected). (We have not implemented this for LOIS.) We note that when the number of locations grow, Nλ becomes rather slow. Nonetheless, Nλ catches up when increasing the dimension. All the results show that Ons(-hs) is faster in lower dimensions, even with a high number of orbits, but that Nλ and Lois can handle higher dimensions.

The libraries Ons and Ons-hs are very comparable in terms of scalability. However, we note that Ons-hs is sometimes faster, we expect this is due to the lazy nature of Haskell: When iterating through a set (especially those formed by products), breaking early means that the remainder of the set does not need to be constructed.

9.4. Learning Results

Both implementations in Nλ and Ons are direct implementations of the pseudocode for νL* with no further optimisations. The authors of Lois implemented νL* in their library as well.[8] They reported similar performance as the implementation in Nλ (private communication). Hence we focus our comparison on Nλ and Ons. We use the variant of νL* where counterexamples are added as columns instead of prefixes.

The implementation in Nλ has the benefit that it can work with different symmetries. Indeed, the structured examples, FIFO and *ww*, are equivariant w.r.t. the equality symmetry as well as the total order symmetry. For that reason, we run the Nλ implementation using both the equality symmetry and the total order symmetry on those languages. For the languages $\mathcal{L}_{\text{max}}$, $\mathcal{L}_{\text{int}}$ and the random automata, we can only use the total order symmetry.

To run the νL* algorithm, we implement an external oracle for the membership queries. This is akin to the application of learning black box systems [13]. For equivalence queries, we constructed counterexamples by hand. All implementations receive the same counterexamples. We measure CPU time instead of real time, so that we do not account for the external oracle.

The results in Table 3 show an advantage for Ons for random automata. Additionally, we report the number of membership queries, which can vary for each implementation as some steps in the algorithm depend on the internal ordering of set data structures.

We have not benchmarked the learning algorithm with the random automata with formulae. The reason is that the logical information, the formulae, are not even given to the learning algorithm, since the learning algorithm can only query the language.

In contrast to the case of minimisation, the results suggest that Nλ cannot exploit the logical structure of FIFO(n), $\mathcal{L}_{\text{max}}$ and $\mathcal{L}_{\text{int}}$ as it is not provided

[8] Can be found on `github.com/eryxcc/lois/blob/master/tests/learning.cpp`

			ONS		ONS-HS		Nλ^{ord}		Nλ^{eq}	
Model	N	dim	time (s)	MQs	time (s)	MQs	time (s)	MQs	time (s)	MQs
Random	4	1	127.47	2321	6.94	1915	2391.08	1243		
Random	5	1	0.12	404	0.08	404	2433.77	435		
Random	3	0	0.86	499	0.14	470	1818.97	422		
Random	5	1	> 1h		192.18	6870	> 1h			
Random	4	1	0.08	387	0.06	387	2097.43	387		
FIFO(1)	3	1	0.04	119	0.01	119	3.17	119	1.76	51
FIFO(2)	6	2	1.73	2655	0.55	2655	391.89	3818	40.00	434
FIFO(3)	19	3	2793.93	298400	451.67	302868	> 1h		2047.32	8151
ww_1	4	1	0.42	134	0.04	111	2.49	77	1.47	30
ww_2	8	2	265.79	3671	14.30	2317	227.66	2140	30.58	237
ww_3	24	3	> 1h		> 1h		> 1h		> 1h	
$\mathcal{L}_{\max}$	3	1	0.01	54	0.01	54	3.58	54		
$\mathcal{L}_{\text{int}}$	5	2	0.59	478	0.17	478	83.26	478		

Table 3: Running times and number of membership queries for the νL* algorithm. For Nλ we used two version: Nλ^{ord} uses the total order symmetry Nλ^{eq} uses the equality symmetry.

a priori. For $ww(2)$ we inspected the output on Nλ and saw that it learned some logical structure. For example, it outputs $\{(a, b) \mid a \neq b\}$ as a single object instead of two orbits $\{(a, b) \mid a < b\}$ and $\{(a, b) \mid b < a\}$. This may explain why Nλ is still competitive. For languages which are equivariant for the equality symmetry, the Nλ implementation using the equality symmetry can learn with much fewer queries. This is expected as the automata themselves have fewer orbits. It is interesting to see that these languages can be learned more efficiently by choosing the right symmetry.

10. Related work

As stated in the introduction, Nλ [18] and LOIS [19] use first-order formulas to represent nominal sets and use SMT solvers to manipulate them. This makes both libraries very flexible and they indeed implement the equality symmetry as well as the total order symmetry. As their representation is not unique, the efficiency depends on how the logical formulas are constructed. As such, they do not provide complexity results. In contrast, our direct representation allows for complexity results (Section 6) and leads to different performance characteristics (Section 9).

A second big difference is that both Nλ and LOIS implement a "programming paradigm" instead of just a library. This means that they overload natural programming constructs in their host languages (Haskell and C++ respectively). For programmers this means they can think of infinite sets without having to know about nominal sets.

It is worth mentioning that an older (unreleased) version of Nλ implemented nominal sets with orbits instead of SMT solvers [27]. However, instead of characterising orbits (e.g., by its dimension), they represent orbits by a representative element. The authors of Nλ have reported that the current version is faster [18].

The theoretical foundation of our work is the main representation theorem in [2]. We add to that by instantiating it to the total order symmetry and distil a concrete representation of nominal sets. As far as we know, we provide the first implementation of the representation theory in [2].

Another tool using nominal sets is Mihda [28]. Here, only the equality symmetry is implemented. This tool implements a translation from π-calculus to history-dependent automata (HD-automata), with the aim of minimisation and checking bisimilarity. The implementation in OCaml is based on *named sets*, which are finite representations for nominal sets. The theory of named sets is well-studied and has been used to model various behavioural models with local names. For those results, the categorical equivalences between named sets, nominal sets and a certain (pre)sheaf category have been exploited [29, 30]. The total order symmetry is not mentioned in their work. We do, however, believe that similar equivalences between categories can be stated. Interestingly, the product of named sets is similar to our representation of products of nominal sets: pairs of elements together with data which denotes the relation between data values.

Fresh OCaml [31] and Nominal Isabelle [32] are both specialised in name-binding and α-conversion used in proof systems. They only use the equality symmetry and do not provide a library for manipulating nominal sets. Hence they are not suited for our applications.

On the theoretical side, there are many complexity results for register automata [4, 33]. In particular, we note that problems such as emptiness and equivalence are NP-hard depending on the type of register automaton. This does not easily compare to our complexity results for minimisation. One difference is that we use the total order symmetry, where the local symmetries are always trivial (Lemma 5.1). As a consequence, all the complexity required to deal with groups vanishes. Rather, the complexity is transferred to the input of our algorithms, because automata over the equality symmetry require more orbits when expressed over the total order symmetry. Another difference is that register automata allow for duplicate values in the registers. In nominal automata, such configurations will be encoded in different orbits. An interesting open problem is whether equivalence of unique-valued register automata is in Ptime [33].

Orthogonal to nominal automata, there is the notion of symbolic automata [3, 34]. These automata are also defined over infinite alphabets but they use predicates on transitions, instead of relying on symmetries. Symbolic automata are finite state (as opposed to infinite state nominal automata) and do not allow for storing values. However, they do allow for general predicates over an infinite alphabet, including comparison to constants.

11. Conclusion and Future Work

We presented a concrete finite representation for nominal sets over the total order symmetry. This allowed us to implement a library, Ons, and provide complexity bounds for common operations. The experimental comparison of Ons against existing solutions for automata minimisation and learning show that our implementation is much faster in many instances. As such, we believe Ons is a promising implementation of nominal techniques.

A natural direction for future work is to consider other symmetries, such as the equality symmetry. Here, we may take inspiration from existing tools such as Mihda (see Section 10). Another interesting question is whether it is possible to translate a nominal automaton over the total order symmetry which accepts an equality language to an automaton over the equality symmetry. This would allow one to efficiently move between symmetries. Finally, our techniques can potentially be applied to timed automata by exploiting the intriguing connection between the nominal automata that we consider and timed automata [21].

Acknowledgements

We would like to thank Szymon Toruńczyk and Eryk Kopczyński for their prompt help when using the Lois library. For general comments and suggestions we would like to thank Ugo Montanari and Niels van der Weide. At last, we want to thank the anonymous reviewers of ICTAC 2018 for their comments.

References

[1] D. Venhoek, J. Moerman, J. Rot, Fast computations on ordered nominal sets, in: B. Fischer, T. Uustalu (Eds.), Theoretical Aspects of Computing - ICTAC 2018 - 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings, Vol. 11187 of Lecture Notes in Computer Science, Springer, 2018, pp. 493–512 (2018). `doi:10.1007/978-3-030-02508-3_26`.
URL `https://doi.org/10.1007/978-3-030-02508-3_26`

[2] M. Bojańczyk, B. Klin, S. Lasota, Automata theory in nominal sets, Logical Methods in Computer Science 10 (3) (2014). `doi:10.2168/LMCS-10(3:4)2014`.

[3] L. D'Antoni, M. Veanes, The power of symbolic automata and transducers, in: R. Majumdar, V. Kuncak (Eds.), Computer Aided Verification, CAV 2017, Part I, Vol. 10426 of Lecture Notes in Computer Science, Springer, 2017, pp. 47–67 (2017). `doi:10.1007/978-3-319-63387-9_3`.

[4] R. Grigore, N. Tzevelekos, History-register automata, Logical Methods in Computer Science 12 (1) (2016). `doi:10.2168/LMCS-12(1:7)2016`.

[5] M. Kaminski, N. Francez, Finite-memory automata, Theor. Comput. Sci. 134 (2) (1994) 329–363 (1994). `doi:10.1016/0304-3975(94)90242-9`.

[6] U. Montanari, M. Pistore, An introduction to history dependent automata, Electr. Notes Theor. Comput. Sci. 10 (1998) 170–188 (1998). `doi:10.1016/S1571-0661(05)80696-6`.

[7] L. Segoufin, Automata and logics for words and trees over an infinite alphabet, in: Z. Ésik (Ed.), Computer Science Logic, CSL 2006, Vol. 4207 of Lecture Notes in Computer Science, Springer, 2006, pp. 41–57 (2006). doi:10.1007/11874683_3.

[8] F. Aarts, P. Fiterău-Broştean, H. Kuppens, F. W. Vaandrager, Learning register automata with fresh value generation, in: M. Leucker, C. Rueda, F. D. Valencia (Eds.), Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings, Vol. 9399 of Lecture Notes in Computer Science, Springer, 2015, pp. 165–183 (2015). doi:10.1007/978-3-319-25150-9_11.
URL https://doi.org/10.1007/978-3-319-25150-9_11

[9] B. Bollig, P. Habermehl, M. Leucker, B. Monmege, A fresh approach to learning register automata, in: M. Béal, O. Carton (Eds.), Developments in Language Theory, DLT 2013, Vol. 7907 of Lecture Notes in Computer Science, Springer, 2013, pp. 118–130 (2013). doi:10.1007/978-3-642-38771-5_12.

[10] S. Cassel, F. Howar, B. Jonsson, B. Steffen, Active learning for extended finite state machines, Formal Asp. Comput. 28 (2) (2016) 233–263 (2016). doi:10.1007/s00165-016-0355-5.

[11] S. Drews, L. D'Antoni, Learning symbolic automata, in: A. Legay, T. Margaria (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2017, Part I, Vol. 10205 of Lecture Notes in Computer Science, 2017, pp. 173–189 (2017). doi:10.1007/978-3-662-54577-5_10.

[12] J. Moerman, M. Sammartino, A. Silva, B. Klin, M. Szynwelski, Learning nominal automata, in: Castagna and Gordon [35], pp. 613–625 (2017). doi:10.1145/3009837.
URL http://dl.acm.org/citation.cfm?id=3009879

[13] F. W. Vaandrager, Model learning, Commun. ACM 60 (2) (2017) 86–95 (2017). doi:10.1145/2967606.

[14] P. Fiterău-Broştean, R. Janssen, F. W. Vaandrager, Combining model learning and model checking to analyze TCP implementations, in: S. Chaudhuri, A. Farzan (Eds.), Computer Aided Verification, CAV 2016, Part II, Vol. 9780 of Lecture Notes in Computer Science, Springer, 2016, pp. 454–471 (2016). doi:10.1007/978-3-319-41540-6_25.

[15] M. Gabbay, A. M. Pitts, A new approach to abstract syntax with variable binding, Formal Asp. Comput. 13 (3-5) (2002) 341–363 (2002). doi:10.1007/s001650200016.

[16] A. M. Pitts, Nominal techniques, SIGLOG News 3 (1) (2016) 57–72 (2016). doi:10.1145/2893582.2893594.

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

[18] B. Klin, M. Szynwelski, SMT solving for functional programming over infinite structures, in: R. Atkey, N. R. Krishnaswami (Eds.), Proceedings 6th Workshop on Mathematically Structured Functional Programming, MSFP 2016, Vol. 207 of EPTCS, 2016, pp. 57–75 (2016). doi:10.4204/EPTCS.207.3.

[19] E. Kopczynski, S. Toruńczyk, LOIS: an application of SMT solvers, in: T. King, R. Piskac (Eds.), Proceedings of the 14th International Workshop on Satisfiability Modulo Theories, SMT 2016, Vol. 1617 of CEUR Workshop Proceedings, CEUR-WS.org, 2016, pp. 51–60 (2016).
URL http://ceur-ws.org/Vol-1617/paper5.pdf

[20] E. Kopczynski, S. Toruńczyk, LOIS: syntax and semantics, in: Castagna and Gordon [35], pp. 586–598 (2017). doi:10.1145/3009837.
URL http://dl.acm.org/citation.cfm?id=3009876

[21] M. Bojańczyk, S. Lasota, A machine-independent characterization of timed languages, in: A. Czumaj, K. Mehlhorn, A. M. Pitts, R. Wattenhofer (Eds.), Automata, Languages, and Programming, ICALP 2012, Part II, Vol. 7392 of Lecture Notes in Computer Science, Springer, 2012, pp. 92–103 (2012). doi:10.1007/978-3-642-31585-5_12.

[22] M. Bojańczyk, Slightly Infinite Sets, Draft December 4, 2018, 2018 (2018).
URL https://www.mimuw.edu.pl/~bojan/upload/main-6.pdf

[23] D. Angluin, Learning regular sets from queries and counterexamples, Inf. Comput. 75 (2) (1987) 87–106 (1987). doi:10.1016/0890-5401(87)90052-6.

[24] P. Fiterău-Broştean, Active model learning for the analysis of network protocols, Ph.D. thesis, Radboud University, Nijmegen, The Netherlands (2018).
URL http://hdl.handle.net/2066/187331

[25] L. M. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: C. R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, Vol. 4963 of Lecture Notes in Computer Science, Springer, 2008, pp. 337–340 (2008). doi:10.1007/978-3-540-78800-3_24.
URL https://doi.org/10.1007/978-3-540-78800-3_24

[26] C. Okasaki, Purely functional data structures, Cambridge University Press, 1999 (1999).

[27] M. Bojańczyk, L. Braud, B. Klin, S. Lasota, Towards nominal computation, in: J. Field, M. Hicks (Eds.), Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012, ACM, 2012, pp. 401–412 (2012). doi:10.1145/2103656.2103704.
URL https://doi.org/10.1145/2103656.2103704

[28] G. L. Ferrari, U. Montanari, E. Tuosto, Coalgebraic minimization of hd-automata for the pi-calculus using polymorphic types, Theor. Comput. Sci. 331 (2-3) (2005) 325–365 (2005). doi:10.1016/j.tcs.2004.09.021.

[29] V. Ciancia, A. Kurz, U. Montanari, Families of symmetries as efficient models of resource binding, Electr. Notes Theor. Comput. Sci. 264 (2) (2010) 63–81 (2010). doi:10.1016/j.entcs.2010.07.014.

[30] V. Ciancia, U. Montanari, Symmetries, local names and dynamic (de)-allocation of names, Inf. Comput. 208 (12) (2010) 1349–1367 (2010). doi:10.1016/j.ic.2009.10.007.

[31] M. R. Shinwell, A. M. Pitts, Fresh objective Caml user manual, Tech. rep., University of Cambridge, Computer Laboratory (2005).

[32] C. Urban, C. Tasson, Nominal techniques in isabelle/hol, in: R. Nieuwenhuis (Ed.), Automated Deduction - CADE-20, Vol. 3632 of Lecture Notes in Computer Science, Springer, 2005, pp. 38–53 (2005). doi:10.1007/11532231_4.

[33] A. S. Murawski, S. J. Ramsay, N. Tzevelekos, Bisimilarity in fresh-register automata, in: 30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, IEEE Computer Society, 2015, pp. 156–167 (2015). doi:10.1109/LICS.2015.24.
URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7174833

[34] O. Maler, I. Mens, A generic algorithm for learning symbolic automata from membership queries, in: Models, Algorithms, Logics and Tools, 2017, pp. 146–169 (2017). doi:10.1007/978-3-319-63121-9_8.

[35] G. Castagna, A. D. Gordon (Eds.), Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, ACM, 2017. doi:10.1145/3009837.

Appendix A. Auxiliary results and omitted proofs

Theorem Appendix A.1 (Theorem 8.1)**.** *If nominal sets are implemented with a tree-based set structure (as in* Ons*), the complexity of the following set operations 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 $X \times Y$	$O(\mathrm{N}(X \times Y)) \subseteq O(3^{\dim(X)+\dim(Y)} \mathrm{N}(X) \mathrm{N}(Y))$

The functions $p \colon X \to 2$ *and* $f \colon X \to Y$ *are user defined, and assumed to take* $O(1)$ *time per invocation.*

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 (L, R and B), 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 $O(3^{\dim(X)+\dim(Y)}\,\mathrm{N}(X)\,\mathrm{N}(Y))$. $\square$