Data Monoids[*]

M. Bojańczyk[1]

1 **University of Warsaw**

Abstract

We develop an algebraic theory for languages of data words. We prove that, under certain conditions, a language of data words is definable in first-order logic if and only if its syntactic monoid is aperiodic.

1998 ACM Subject Classification F.4.3 Formal Languages

Keywords and phrases Monoid, Data Words, Nominal Set, First-Order Logic

1 Introduction

This paper is an attempt to combine two fields.

The first field is the algebraic theory of regular languages. In this theory, a regular language is represented by its syntactic monoid, which is a finite monoid. It turns out that many important properties of the language are reflected in the structure of its syntactic monoid. One particularly beautiful result is the Schützenberger-McNaughton-Papert theorem, which describes the expressive power of first-order logic.

> Let $L \subseteq A^*$ be a regular language. Then L is definable in first-order logic if and only if its syntactic monoid M_L is aperiodic.

For instance, the language "words where there exists a position with label a" is defined by the first-order logic formula (this example does not even use the order on positions $<$, which is also allowed in general)

$$\exists x.\ a(x).$$

The syntactic monoid of this language is isomorphic to $\{0, 1\}$ with multiplication, where 0 corresponds to the words that satisfy the formula, and 1 to the words that do not. Clearly, this monoid does not contain any non-trivial group. There are many results similar to theorem above, each one providing a connection between seemingly unrelated concepts of logic and algebra, see e.g. the book [8].

The second field is the study of languages over infinite alphabets, commonly called languages of data words. Regular languages are usually considered for finite alphabets. Many current motivations, including XML and verification, require the use of infinite alphabets. In this paper, we use the following definition of data words. We fix for the rest of the paper a single infinite alphabet D, and study words and languages over D. A typical language, which we use as our running example, is "some two consecutive positions have the same letter", i.e.

$$L_{dd} = \bigcup_{d \in D} D^* dd D^* = \{d_1 \cdots d_n \in D^* : d_i = d_{i+1} \text{ for some } i \in \{1, \ldots, n-1\}\}.$$

* Author supported by ERC Starting Grant "Sosna".

A number of automata models have been developed for such languages. The general theme is that there is a tradeoff between the following three properties: expressivity, good closure properties, and decidable (or even efficiently decidable) emptiness. The existing models strike different balances in this tradeoff, and it is not clear which balance, if any, should deserve the name of “regular language”. Also, logics have been developed to express properties of data words. For more information on automata and logics for data words, see the survey [7].

The motivation of this paper is to combine the two fields, and prove a theorem that is analogous to the Schützenberger-McNaughton-Papert characterization theorem, but talks about languages of data words. If we want an analogue, we need to choose the notion of regular language for data words, the definition of first-order logic for data words, and then find the property corresponding to aperiodicity.

For first-order logic over data words we use a notion that has been established in the literature. Formulas are evaluated in data words. The quantifiers quantify over positions of the data word, we allow two binary predicates: $x < y$ for order on positions, and $x \sim y$ for having the same data value. An example formula of a formula of first-order logic is the formula

$$\exists x \exists y \qquad x < y \quad \wedge \quad x \sim y \quad \wedge \quad \neg(\exists z\; x < z \wedge z < y),$$

which defines the language L_{dd} in the running example.

As for the notion of regular language, we use the monoid approach. For languages of data words, we use the syntactic monoid, defined in the same way as it is for words over finite alphabets. That is, elements of the syntactic monoid are equivalence classes of the two-sided Myhill-Nerode congruence. When the alphabet is infinite, the syntactic monoid is infinite for every language of data words, except those that depend only on the length of the word. For instance, in the case of the running example language L_{dd}, two words $w, w' \in D^*$ are equivalent if and only if: either both belong to L_{dd}, or both have the same first and last letters. Since there are infinitely many letters, there are infinitely many equivalence classes.

Instead of studying finite monoids, we study something called *orbit finite* monoids. Intuitively speaking, two elements of a syntactic monoid are considered to be in the same orbit if there is a renaming of data values that maps one element to the other[1]. For instance, in the running example L_{dd}, the elements of the syntactic monoid that correspond to the words $1 \cdot 7$ and $2 \cdot 3 \cdot 4 \cdot 8$ are not equal, but they are in the same orbit, because the renaming $i \mapsto i + 1$ maps $1 \cdot 7$ to $2 \cdot 8$, which corresponds to the same element of the syntactic monoid as $2 \cdot 3 \cdot 4 \cdot 8$. It is not difficult to see that the syntactic monoid of L_{dd} has four orbits: one for the empty word, one for words inside L_{dd}, one for words outside L_{dd} where the first and last letters are equal, and one for words outside L_{dd} where the first and last letters are different.

The contribution of this paper is a study of orbit finite data monoids. We develop the algebraic theory of orbit finite data monoids, and show that it resembles the theory of finite monoids. The main result of the paper is Theorem 11, which shows that the Schützenberger-McNaughton-Papert characterization also holds for languages of data words with orbit finite syntactic data monoids:

> Let $L \subseteq D^*$ be a language whose syntactic data monoid M_L is orbit finite. Then L is definable in first-order logic if and only if M_L is aperiodic.

[1] This is related to Proposition 3 in [6]

Related work. The idea to present effective characterizations of logics on data words was proposed by Benedikt, Ley and Puppis. In [1], they show that definability in first-order logic is undecidable if the input is a nondeterministic register automaton. Also, they provide some decidable characterizations, including a characterization of first-order logic with local data comparisons within the class of languages recognized by deterministic register automata. (This result is incomparable the one in this paper.)

The key property of data monoids, namely that elements of the monoid are acted on by permutations of data values, appears in the nominal sets of Gabbay and Pitts [5]. The connections with the literature on nominal sets are still relatively unexplored.

There are two papers on algebra for languages over infinite alphabets. One approach is described by Bouyer, Petit and Thérien [3]. Their monoids are different from ours in the following ways: the definition of a monoid explicitly talks about registers, there is no syntactic monoid, monoids have undecidable emptiness (not mention questions such as aperiodicity or first-order definability). Our setting is closer to the approach of Francez and Kaminski from [4], but the latter talks about automata and not monoids, and does not study the connection with logics.

Acknowledgments. I would like to thank Clemens Ley for introducing me to the subject, Sławomir Lasota for suggesting the key idea of orbits, and Bartek Klin for pointing out the connection to nominal sets. I would also like to thank Michael Kaminski and the anonymous referees for their many helpful comments.

2 Myhill-Nerode equivalence

We start out with an investigation of the Myhill-Nerode syntactic congruence, in the case of data words.

We fix for the rest of this paper an infinite set of data values D. One can think of D as being the natural numbers, but all the structure of natural numbers (such as order) is not going to play a role. Data words are words in D^*. Inside D, we will distinguish a finite subset $C \subseteq D$ of constants. Although D is fixed, the constants will vary. The constants are introduced for two purposes. First, interpreting a finite alphabet as a set of constants, one recovers the traditional theory of regular languages in this setting. Second, constants are useful in the proofs. All definitions will use the set of constant as a parameter.

We use the term *data renaming (with constants C)* for a bijection on the alphabet that is the identity function on C.[2] To simplify proofs, we require a data renaming to be the identity on all but finitely many data values[3]. We write Γ_C for the set of all data renamings with constants C. This set has a group structure. We write π, τ, σ for data renamings. When we apply a data renaming π to a data value d, we skip the brackets and write πd instead of $\pi(d)$.

A data renaming π is naturally extended to a homomorphism $[\pi] : D^* \to D^*$ of data words. For a fixed set of constants C, we study only languages that are invariant under data renaming, i.e. languages such that $[\pi]L = L$ for any data renaming $\pi \in \Gamma_C$.[4]

Suppose that $L \subseteq D^*$ is a language that is invariant under data renamings. We define two equivalence relations on D^*.

[2] This is called C-preservation in Definition 3 of [4].
[3] The same results hold when all bijections are allowed, including the Memory Theorem.
[4] These languages are called co-C-invariant languages in Definition 8 of [4].

The first equivalence relation is the two-sided Myhill-Nerode equivalence relation:

$$w \equiv_L v \qquad \text{if for all } u_1, u_2 \in D^*,\ u_1 w u_2 \in L \Leftrightarrow u_1 v u_2 \in L.$$

We also use the name *syntactic congruence* for this relation. If L uses only constants, i.e. $L \subseteq C^*$, then this equivalence relation has finitely many equivalence classes. However, for any language that uses non-constants in a nontrivial way, this equivalence relation will have infinitely many equivalence classes. The problem is that the relation really cares about specific data values.

This problem is fixed by the second equivalence relation:

$$w \simeq_L v \qquad \text{if for some data renaming } \tau \in \Gamma_C,\ [\tau]w \equiv_L v.$$

As discussed in the introduction, in the running example L_{dd} this equivalence relation has four equivalence classes. On the other hand, some reasonable languages have infinitely many equivalence classes in $\simeq_L$. An example of such an L is "the first letter also appears one some other position". This language is recognized by a left-to-right register automaton with three states. Nevertheless, an equivalence class of a word w under $\equiv_L$ is determined by the first letter of w and the set of distinct letters in w. Consequently, any two words with a different number of distinct letters are not equivalent under $\simeq_L$. (This problem disappears when one uses the one-sided Myhill-Nerode equivalence.)

Despite the limitations, we will be studying languages that have finitely many equivalence classes for $\simeq_L$, because they yield a monoid structure.

3 Data monoids

Fix a finite set of constants $C \subseteq D$. We want to define something like a monoid for data languages. We want there to be a "free object", which should describe D^*. We want each language to have its syntactic object, which should be finite in many interesting cases, and be somehow optimal for all other objects capturing the language. These goals are achieved by the notion of a data monoid, as defined below.

Definition of a data monoid. A *data monoid* (with constants C) is a monoid M, which is acted upon by data renamings (with constants C). That is, for every data renaming $\tau \in \Gamma_C$, there is an associated monoid automorphism $[\tau] : M \to M$. The mapping $\tau \mapsto [\tau]$ should be a group action, namely $[\tau \circ \sigma] = [\tau] \circ [\sigma]$, and the identity renaming should be associated to the identity automorphism. There are two key examples:

- The free data monoid. The monoid is D^*, while the action is defined by

$$[\tau](d_1 \cdots f_n) = \tau(d_1) \cdots \tau(d_n) \qquad \text{for } d_1, \ldots, d_n \in D.$$

- The syntactic data monoid of a language $L \subseteq D^*$, which we denote M_L. The monoid is the quotient of D^* under the Myhill-Nerode equivalence $\equiv_L$. This is a well defined monoid, since $\equiv_L$ is a monoid congruence (unlike $\simeq_L$). The action is defined by

$$[\tau]([w]_{/\equiv_L}) = [[\tau](w)]_{/\equiv_L}.$$

  In Lemma 1, we show that this action is well defined and does not depend on w.

In the sequel, we will drop the brackets $[\tau]$ and simply write τ, when the context indicates if τ is treated as function on data values, or on elements of the monoid. Note that for two renamings τ, σ and an element m of the monoid, the notation $\tau\sigma m$ is unambiguous, since we are dealing with an action.

Finite support axiom. We say that a set $X \subseteq D$ of data values *supports* an element m of a data monoid, if $\tau m = m$ holds for every data renaming in $\tau \in \Gamma_X$. We require an additional property from a data monoid, called the *finite support axiom*: for any element m of the data monoid, there is a finite support.

Both examples above satisfy the finite support axiom. The axiom is violated by some data monoids, even finite ones, as shown in the following example, due to Szymon Toruńczyk. Let $\mathbb{Z}_2$ be the two element group, with additive notation. Let $h : \Gamma_\emptyset \to \mathbb{Z}_2$ assign 0 to even permutations and 1 to odd permutations. The monoid M is also $\mathbb{Z}_2$; the action is defined by $\tau m = m + h(\tau)$. The finite support axiom fails, since odd permutations can involve any pair of data values.

Nevertheless, all data monoids in this paper are homomorphic images (see below) of the free data monoid, which guarantees the finite support axiom.

Congruences of data monoids. A congruence of a data monoid is an equivalence relation $\simeq$ on the underlying monoid that is compatible with concatenation:

$$m_1 \simeq n_1 \text{ and } m_2 \simeq n_2 \quad \text{implies} \quad m_1 n_1 \simeq m_2 n_2 \tag{1}$$

and which is also compatible with every data renaming $\tau \in \Gamma_C$:

$$m \simeq n \quad \text{implies} \quad \tau m \simeq \tau n \tag{2}$$

Note that the notion of congruence depends implicitly on the set C of constants, since the set of constants indicates which functions τ are data renamings. It might be the case that a relation is a congruence for a set of constants C, but it is no longer a congruence for a smaller set of constants $E \subsetneq C$.

▶ **Lemma 1.** *For a data language $L \subseteq D^*$ that is closed under data renamings, the syntactic equivalence $\equiv_L$ is a congruence of the free data monoid D^*.*

Proof. The syntactic equivalence satisfies (1), as it does for any language $L \subseteq D^*$, not only those closed under data renamings. We use the closure under data renamings to prove (2). Let w and w' be $\equiv_L$-equivalent data words. We need to show that for any data renaming τ, also τw and $\tau w'$ are $\equiv_L$-equivalent. In other words, we need to show that

$$u(\tau w)v \in L \qquad \Leftrightarrow \qquad u(\tau w')v \in L \qquad \text{for any } u, v \in D^*. \tag{3}$$

Since τ^{-1} induces an bijection on data words, we have

$$u(\tau w)v \in L \qquad \Leftrightarrow \qquad \tau^{-1}(u(\tau w)v) \in \tau^{-1}L.$$

Because concatenating words commutes with data renamings,

$$\tau^{-1}(u(\tau w)v) = (\tau^{-1}u)w(\tau^{-1}v).$$

By the assumption on L being closed under data renamings, $L = \tau^{-1}L$. It follows that the left side of the equivalence (3) is equivalent to

$$(\tau^{-1}u)w(\tau^{-1}v) \in L.$$

Likewise, the right side of the equivalence (3) becomes

$$(\tau^{-1}u)w'(\tau^{-1}v) \in L.$$

Both sides are equivalent, because $w \equiv_L w'$. ◀

Homomorphisms of data monoids. Suppose that M and N are data monoids, with constants C. A *data monoid homomorphism* $h : M \to N$ is defined to be a monoid homomorphism of the underlying monoids that commutes with the action of data renamings

$$h(\tau s) = \tau(h(s)) \qquad \text{for all } \tau \in \Gamma_C.$$

In the literature on nominal sets, a monoid homomorphism that satisfies the commuting condition above is called *equivariant*.

A congruence gives rise to a homomorphism, in the usual way. It follows that the function that maps a word w to its equivalence class under $\equiv_L$, is a data monoid homomorphism. This function is called the *syntactic morphism*. The target of the syntactic morphism is called the *syntactic data monoid* of L, and denoted M_L.

Note that data monoid homomorphisms preserve the finite support axiom. Therefore, all the syntactic monoids we will study, as images of the free data monoid, will satisfy the finite support axiom.

We say that a data monoid homomorphism $h : D^* \to M$ *recognizes* a data language L if membership $w \in L$ is uniquely determined by the image $h(w) \in M$. In other words, $L = h^{-1}(h(L))$. Clearly, the syntactic morphism of L recognizes L. The following lemma shows that the syntactic morphism is the "best" morphism recognizing L.

▶ **Lemma 2.** *Consider a data language $L \subseteq D^*$. Any surjective data monoid homomorphism that recognizes L can be uniquely extended to the syntactic data monoid homomorphism.*

Proof. Let $\alpha : D^* \to M$ be a surjective data monoid homomorphism that recognizes L.

We claim that $\alpha(w) = \alpha(w')$ implies $w \equiv_L w'$. If the contrary were true, we would have two words w, w' with the same image under α, and some words v, u such that exactly one of the two words $vwu, vw'u$ would belong to L. This would contradict the assumption on α recognizing L, since $vwu, vw'u$ would have the same image under α.

From the claim, it follows that for every element $m \in M$, all words in $\alpha^{-1}(m)$ are mapped by the syntactic monoid homomorphism to the same equivalence class, call it W_m. Therefore, the function $m \mapsto W_m$ extends α to the syntactic data monoid homomorphism. ◀

The following lemma justifies the use of the name "free data monoid".

▶ **Lemma 3.** *Let M be a data monoid, and let $h : D \to M$ be a function that commutes with data renamings (i.e. an equivariant function). This function can be uniquely extended to a data monoid homomorphism $[h] : D^* \to M$.*

Orbits. A syntactic data monoid usually has an infinite carrier. This is not a problem, because what really interests us in a data monoid is not the elements, but their orbits. Formally, the *orbit* of an element m of a data monoid M is the set

$$\Gamma_C \cdot m = \{\tau m : \tau \in \Gamma_C\}.$$

Note how the set of constants influences the notion of orbit.

Consider the syntactic data monoid and the syntactic data monoid homomorphism of a data language $L \subseteq D^*$. As elements of the monoid are to equivalence classes of $\equiv_L$, orbits are to equivalence classes of $\simeq_L$. More formally, for two words $w, v \in D^*$, the orbits of $h_L(w)$ and $h_L(v)$ are equal if and only if $w \simeq_L v$. Suppose that a language $L \subseteq D^*$ is closed under data renamings. If this language is recognized by a data monoid homomorphism h, then the image $h(L)$ is necessarily a union of orbits.

A data monoid is called *orbit finite* if it has finitely many distinct orbits. We are interested in languages whose syntactic data monoids are orbit finite. As far as this paper is concerned, these are the "regular" languages of data words.

This completes the definition of data monoids and their basic properties.

4 Memory

As usual, the definition of data monoids attempts to use the least amount of primitive concepts. We now show how other natural concepts can be derived from these primitives.

Consider the running example language L_{dd}. Let M_{dd} be the syntactic monoid of this language. Formally speaking, elements of M_{dd} are equivalence classes of data words. The equivalence class of a data word $d_1 \cdots d_n \not\in L_{dd}$ can be identified with the ordered first/last letter pair (d_1, d_n). Intuitively speaking, the data values d_1 and d_n are "important" for this equivalence class (monoid element), while the other data values $d_2, \ldots, d_{n-1}$ are not "important". Below we present a definition, called *memory*, which automatically extracts the "important" data values from an element of a data monoid, and uses only the primitive concepts of a data monoid. This definition is inspired by the notion of memory from [2].

Let m be an element of a data monoid. We write stab_m for the subgroup of data renamings τ that satisfy $\tau m = m$. In other words, $\mathrm{stab}_m = \Gamma_{C \cup \{m\}}$. In the running example, when m is the equivalence class of a word $d_1 \ldots d_n \not\in L_{dd}$, then stab_m is the set of data renamings that have both d_1 and d_n as fixpoints. (When m is the equivalence class of a word $w \in L_{dd}$, or m is the identity, then stab_m contains all data renamings.) For each data value $d \in D$, we define its *data orbit with respect to* m as the following set of data values

$$\{\tau d : \tau \in \mathrm{stab}_m\} = \mathrm{stab}_m(d).$$

In the running example, when m is the equivalence class of $d_1 \ldots d_n \not\in L_{dd}$, the data orbit of d_1 with respect to m is $\{d_1\}$, for d_n the data orbit is $\{d_n\}$, and for any other data value it is $D - \{d_1, d_n\}$.

The following theorem gives a more precise interpretation of the finite support axiom. It says that an element does not care about what happens when data values in its unique infinite data orbit are swapped around.

▶ **Theorem 4** (Memory Theorem). *Every element m of an orbit finite data monoid has finitely many data orbits, of which exactly one is infinite. Furthermore, if $mem(m)$ is defined as the union of the finite data orbits, then stab_m contains all data renamings that are the identity on $mem(m)$, i.e. $\Gamma_{mem(m)} \subseteq \mathrm{stab}_m$.*

Note that $mem(m)$ includes all constants, since they have one element data orbits.

The result above only refers to the set structure of a data monoid, i.e. the group action on its elements. It does not refer to the monoid structure, i.e. the multiplication operation and the neutral element. Therefore, it is a result about nominal sets, as in [5]. The Memory Theorem is a corollary of Proposition 3.4 from [5], which says that an element m of a nominal set M is supported by the intersection of all sets of data values that support it.

▶ **Corollary 5.** *For m, n in a data monoid, $mem(mn) \subseteq mem(m) \cup mem(n)$.*

Proof. Let d, e be elements outside $mem(m) \cup mem(n)$. Let τ be the data renaming that swaps d, e. By the Memory Theorem, $\tau \in \mathrm{stab}_m \cap \mathrm{stab}_n$. It follows that $\tau(mn) = \tau m \cdot \tau n = mn$, and therefore $\tau \in \mathrm{stab}_{mn}$. We have shown that any two elements outside $mem(m) \cup mem(n)$ are in the same data orbit for mn. Since there is only one infinite data orbit, it follows that they are not in $mem(mn)$. ◀

5 Algebraic properties of data monoids

In this section we develop the algebraic theory of data monoids.

Local finiteness. Recall that a monoid is called *locally finite* if all of its finitely generated submonoids are finite. Much of Green theory works for locally finite monoids, which makes the following theorem important.

▶ **Theorem 6.** *Every orbit finite data monoid is locally finite.*

Preliminary work indicates that local finiteness holds under much weaker conditions than being orbit finite. Namely, for every language of data words that is defined in monadic second-order logic with order and data equality, the syntactic monoid is locally finite. This covers, for instance, languages recognized by nondeterministic register automata.

Green's relations. Suppose that M is a data monoid. Using the underlying monoid, one can define Green's relations on elements $m, n \in M$.

- $m \leq_{\mathcal{L}} n$ if $Mm \subseteq Mn$
- $m \leq_{\mathcal{R}} n$ if $mM \subseteq nM$
- $m \leq_{\mathcal{J}} n$ if $MmM \subseteq MnM$

We sometimes say that n is a *suffix* of m instead of writing $m \leq_{\mathcal{L}} n$. Likewise for *prefix* and *infix*, and the relations $\leq_{\mathcal{R}}$ and $\leq_{\mathcal{J}}$. Finally, we use the name *extension* for the converse of infix. Like in any monoid, these relations are transitive and reflexive, so one can talk about their induced equivalence relations $\sim_{\mathcal{L}}$, $\sim_{\mathcal{R}}$ and $\sim_{\mathcal{J}}$. Equivalence classes of these relations are called $\mathcal{L}$-classes, $\mathcal{R}$-classes and $\mathcal{J}$-classes. An $\mathcal{H}$-class is defined to be the intersection of an $\mathcal{L}$-class and an $\mathcal{R}$-class.

In data monoids, we add a new relation, by lifting the order $\leq_{\mathcal{J}}$ to orbits. One could do the same for $\mathcal{R}$ and $\mathcal{L}$, but in our application of data monoids to characterize of first-order logic, we only use the case for $\mathcal{J}$. We write $m \leq_{\mathcal{O}} n$ if there is some data renaming τ such that $\tau m \leq_{\mathcal{J}} n$. The letter $\mathcal{O}$ stands for orbit. In terms of ideals, $\tau(MmM) \subseteq MnM$, or equivalently $M \cdot \tau m \cdot M \subseteq MnM$, or also equivalently $m \in M \cdot \tau n \cdot M$.

▶ **Lemma 7.** *In any data monoid, the relation $\leq_{\mathcal{O}}$ is transitive.*

Proof. Suppose that $x \geq_{\mathcal{O}} y \geq_{\mathcal{O}} z$ holds. This means that z can be written as $\tau(pyq)$ and y can be written as $\sigma(mxn)$, for some data renamings τ, σ and data monoid elements p, q, m, n. Combining the two, we get

$$z = \tau(p \cdot \sigma(mxn) \cdot q) = \tau(p \cdot \sigma m \cdot x \cdot \sigma n \cdot q) \in M \cdot \tau x \cdot M.$$

◀

Of course, in an orbit finite data monoid there are finitely many $\mathcal{O}$-classes, since each is a union of orbits.

$\mathcal{J}$-classes. We present a version of the Memory Theorem for $\mathcal{J}$-classes. For a $\mathcal{J}$-class J, we define its memory as

$$mem(J) = \bigcap_{m \in J} mem(m).$$

Clearly this set is finite, as an intersection of finite sets.

▶ **Theorem 8** (Memory Theorem for $\mathcal{J}$-classes.). *Let J be a $\mathcal{J}$-class. Any data renaming τ that is the identity on $mem(J)$ satisfies $\tau J = J$.*

Quotient under a $\mathcal{J}$-class. Suppose that J is a $\mathcal{J}$-class in a data monoid. Let $\simeq_J$ be the equivalence relation on the data monoid defined by:

$$m \simeq_{/J} n \qquad \text{if } m = n \text{ or neither of } m, n \text{ is an infix of some element of } J.$$

▶ **Lemma 9.** *The relation $\simeq_{/J}$ is a congruence of data monoids, assuming the constants are $mem(J)$. Consequently, the quotient function denoted $h_{/J}$, is a data monoid homomorphism, with constants $mem(J)$.*

Orbit-equivalent $\mathcal{J}$-classes. We say that two $\mathcal{J}$-classes $J, K \subseteq M$ of a data monoid are *orbit-equivalent* if there is some data renaming τ such that $\tau J = K$.

▶ **Lemma 10.** *Orbit-equivalent $\mathcal{J}$-classes form antichains in the order $\leq_{\mathcal{J}}$.*

6 First-order logic

We test the algebra on first-order logic, yielding the main result of the paper, Theorem 11. We extend first-order logic, as described in the introduction, by a unary predicate $d(x)$ for every data value d, which selects positions with data value d. A predicate $d(x)$ is called a *data predicate*. Any formula uses a finite number of data predicates, and its language is invariant under data renamings that preserve the data values of the data predicates it uses. The formulas produced below will use data predicates only for the constants.

▶ **Theorem 11.** *Let L be a data language whose syntactic data monoid M_L is orbit finite. Then L is definable in first-order logic if and only if M_L is aperiodic.*

The implication from first-order definable to aperiodic is proved in the same way as usual, e.g. using an Ehrenfeucht-Fraïssé game. We focus on the more difficult implication. The proof is an induction, which needs a more detailed statement, as presented below.

▶ Proposition 12. Let L be a data language recognized by a data monoid homomorphism $h : D^* \to M$ into a data monoid that is orbit finite and aperiodic. For every $m \in M$, the language $h^{-1}(m)$ can be defined by a sentence φ_m of FO that uses data predicates for data values in $mem(m)$.

Note that in Theorem 11 and Proposition 12, there is an implicit set of constants $C \subseteq D$, which influences the definitions of data language, data monoid, and homomorphism.

Before we prove Proposition 12, we show how it implies Theorem 11. The language L is a union of orbits. Let $m_1, \ldots, m_n$ be chosen representatives of these orbits. We know that L is the same as the union

$$\Gamma_C \cdot m_1 \quad \cup \quad \cdots \quad \cup \quad \Gamma_C \cdot m_n.$$

Each of the languages in the finite union above can be defined in first-order logic, thanks to Proposition 12 and the following lemma.

▶ **Lemma 13.** *Suppose that L is a language defined by a formula of first-order logic with data predicates A. For any $B \subseteq A$, the language*

$$\Gamma_B L = \{\tau w : \tau \in \Gamma_B, w \in L\}$$

is defined by a formula of first-order logic with data predicates B.

We now proceed with the proof of Proposition 12. The proof is by induction on two parameters, ordered lexicographically. The first, and more important, parameter is the number of orbits in the data monoid M. The second parameter is the position of the $\mathcal{O}$-class of m in the (inverse of the) order $\leq_{\mathcal{O}}$. We begin with the $\mathcal{O}$-class of the empty word, and proceed to bigger words, the idea being that it is easier to write formulas for infixes than extensions.

The induction base is done the same way as the induction step, so we omit it, and only do the induction step. Let O be the $\mathcal{O}$-class of m. By induction assumption, we can use formulas for elements with strictly bigger $\mathcal{O}$-classes, which correspond to infixes of m. There are two cases two consider, depending on whether O is a $\mathcal{J}$-class, or not. The case when O is a $\mathcal{J}$-class is more similar to the proof for finite alphabets.

O is not a $\mathcal{J}$-class. Let $J \subsetneq O$ be the $\mathcal{J}$-class that contains m. Let us expand the set of constants from C to $mem(J)$. Consider the data monoid homomorphism

$$h_{/J} : M \to M_{/J}$$

which squashes all elements that are not infixes of J into a single element, as in Lemma 9. Recall the data monoid homomorphism h that recognizes L. It is easy to see that the composition $g = h_J \circ h$ recognizes $h^{-1}(m)$, since g maps the same elements to m as h.

▶ **Lemma 14.** *$M_{/J}$ has fewer orbits than M.*

Thanks to the above lemma, we can use the induction assumption to produce the formula for m. This finishes the case when O is not a $\mathcal{J}$-class.

O is a $\mathcal{J}$-class. For $n \in M$, define K_n to be the set of data words that have image n under h, but all of their proper infixes have image in a strictly bigger $\mathcal{O}$-class than O. Using the induction assumption, we prove the following lemma.

▶ **Lemma 15.** *For each $n \in M$, the language K_n can be defined by a formula of first-order logic with data predicates $mem(n)$.*

▶ **Lemma 16.** *There are finitely many elements $m_1, \ldots, m_k$ and a subset $B \subseteq mem(m)$ such that for every $n \in M$,*

$$n \sim_{\mathcal{R}} m \qquad \textit{iff} \qquad n \in \Gamma_B\{m_1, \ldots, m_k\}.$$

By combining Lemmas 15, 13 and 16, we get the following lemma.

▶ **Lemma 17.** *The infinite union of languages K_n, ranging over $n \sim_{\mathcal{R}} m$, can be defined by a formula of first-order logic with data predicates $mem(m)$.*

Using Lemma 17 and its symmetric variant for $\mathcal{L}$-classes, we can write a formula of first-order logic, with data predicates $mem(m)$, which describes the set, call it L_m, of elements that have a prefix in the $\mathcal{R}$-class of m, and a suffix in the $\mathcal{L}$-class of m:

$$L_m = \bigcup_{\substack{n \\ n \sim_{\mathcal{R}} m}} K_n D^* \quad \cap \quad \bigcup_{\substack{n \\ n \sim_{\mathcal{L}} m}} D^* K_n.$$

If a word w belongs to L_m, then its image $n = h(w)$ satisfies $n \leq_{\mathcal{R}} m$, likewise for $\leq_{\mathcal{L}}$. By the following lemma, if we additionally assume that $n \sim_{\mathcal{J}} m$, then also $n \sim_{\mathcal{L}} m$ and $n \sim_{\mathcal{R}} m$, and hence $n \sim_{\mathcal{H}} m$.

▶ **Lemma 18.** *In a locally finite monoid, $m \sim_{\mathcal{J}} n$ and $m \leq_{\mathcal{R}} n$ imply $m \sim_{\mathcal{R}} n$. Likewise for $\mathcal{L}$.*

In other words, if a word belongs to L_m and has the same $\mathcal{J}$-class as m, then its image is in the $\mathcal{H}$-class of m. In a locally finite aperiodic monoid, this means that its image is simply m:

▶ **Lemma 19.** *In a locally finite aperiodic monoid, all $\mathcal{H}$-classes are singletons.*

Above we have shown that a word in L_m has image m, using the assumption that its image is in the same $\mathcal{J}$-class as m. Therefore, a word has image m if and only if it belongs to language L_m, which is definable in first-order logic, and its $\mathcal{J}$-class is O. Let K be the set of data words that have an infix in O, but are not in O. As argued before, a word has image m if and only if it belongs to the difference $L_m - K$. Therefore, to conclude we only need the following lemma.

▶ **Lemma 20.** *The language K can be defined in first-order logic.*

Proof. When does a word w belong to K? Consider the smallest infix of w that belongs to K. Cutting off the first or last letter of that infix leads to a word in O. Therefore:

$$K = D^* \cdot \Big(\bigcup_{\substack{d \in D, n \in O \\ h(d) \cdot n \notin O}} d \cdot h^{-1}(n) \quad \cup \bigcup_{\substack{d \in D, n \in O \\ n \cdot h(d) \notin O}} h^{-1}(n) \cdot d \Big) \cdot D^*$$

The important observation is that it does no harm to replace $h^{-1}(n)$ by the language L_n defined above. This is because L_n contains $h^{-1}(n)$ and is contained in $h^{-1}(n) \cup K$. Therefore,

$$K = D^* \cdot \Big(\bigcup_{\substack{d \in D, n \in O \\ h(d) \cdot n \notin O}} d \cdot L_n \quad \cup \bigcup_{\substack{d \in D, n \in O \\ n \cdot h(d) \notin O}} L_n \cdot d \Big) \cdot D^*$$

We deal with the infinite union using Lemma 13. ◀

7 Further work

Characterize other logics. It is natural to extend the characterization of first-order logic to other logics. Candidates that come to mind include first-order logic with two variables, or various logics inspired by XPath, or piecewise testable languages. Also, it would be interesting to see the expressive power of languages recognized by orbit-finite data monoids. This class of languages is incomparable in expressive power to first-order logic, e.g. the first-order definable language "some data value appears twice" is not recognized by an orbit-finite data monoid. It would be nice to see a logic, maybe a variant of monadic second-order logic, with the same expressive power as orbit-finite data monoids.

More structure on the data values. In this paper, we study languages that are closed under arbitrary renamings of data values. One could be interested in weaker requirements, that give more general language classes. For instance, consider languages $L \subseteq \mathbb{R}^*$ that are closed under order-preserving renamings. Another, very interesting example, concerns languages of timed automata.

Use mechanisms more powerful than monoids. Even if we are interested in languages that are preserved under arbitrary data renamings, orbit finite data monoids have weak expressive power. Contrary to the case of finite alphabets, over infinite alphabets, (orbit finite) data monoids are strictly less expressive than the natural automaton counterpart[5]. If we only require the syntactic left-to-right automaton to be orbit finite, we get a larger class of languages. This larger class includes the language "the first letter in the word appears also on some other position", which has a syntactic monoid with infinitely many orbits. Therefore, one can ask: is it decidable if an orbit finite automaton recognizes a language that can be defined in first-order logic? We conjecture that this problem is decidable, and even that a necessary and sufficient condition is aperiodicity of the syntactic monoid (which need not be orbit finite). Aperiodicity is not, in general, the right condition for first-order logic, as witnessed by the language "words with an even number of distinct letters", which has an aperiodic syntactic monoid (sic!), but is not definable in first-order logic.

References

1 Michael Benedikt, Clemens Ley, and Gabriele Puppis. Automata vs. logics on data words. In *CSL*, pages 110–124, 2010.

2 Michael Benedikt, Clemens Ley, and Gabriele Puppis. What you must remember when processing data words. In *AMW*, 2010.

3 Patricia Bouyer, Antoine Petit, and Denis Thérien. An algebraic approach to data languages and timed languages. *Inf. Comput.*, 182(2):137–162, 2003.

4 Nissim Francez and Michael Kaminski. An algebraic characterization of deterministic regular languages over infinite alphabets. *Theor. Comput. Sci.*, 306(1-3):155–175, 2003.

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

6 Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

7 Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, pages 41–57, 2006.

8 Howard Straubing. *Finite Automata, Formal Languages, and Circuit Complexity.* Birkhäuser, Boston, 1994.

[5] This counterpart has the same expressive power as deterministic memory automata of Francez and Kaminski [6]

A Finite monoid that violates the finite support axiom

This example is due to Szymon Toruńczyk. Let $\mathbb{Z}_2$ be the two element group, with additive notation. Let $h : \Gamma_\emptyset \to \mathbb{Z}_2$ assign 0 to even permutations and 1 to odd permutations. The monoid M is also $\mathbb{Z}_2$; the action is defined by $\tau m = m + h(\tau)$. The finite support axiom fails, since odd permutations can involve any pair of data values.

B Proof of Lemma 3

Proof. If the extension $[h]$ is to be a data monoid homomorphism, it needs to satisfy

$$h(d_1 \cdots d_n) = h(d_1) \cdots h(d_n).$$

Since every word $w \in D^*$ has a unique decomposition $w = d_1 \cdots d_n$, the above can be seen as the definition of $[h]$. It remains to check that $[h]$ commutes with data renamings, i.e.

$$\tau([h](d_1 \cdots d_n)) = [h](\tau(d_1 \cdots d_n)).$$

By definition of $[h]$, the left side of the above equality is the same as

$$\tau(h(d_1) \cdots h(d_n)).$$

By the definition of the action of data renamings in D^*, the above is the same as

$$\tau(h(d_1)) \cdots \tau(h(d_n)).$$

By the assumption that h commutes with data renamings, the above becomes

$$h(\tau d_1) \cdots h(\tau d_n).$$

Finally, by using the definition of $[h]$ again, the above becomes

$$[h](\tau(d_1 \cdots d_n)).$$

◀

C Proof of Theorem 4

By the finite support axiom, there is some set X of data values such that m is a fixpoint of every data renaming that is the identity on X. Fix this set X for the proof.

All elements outside X are in the same data orbit with respect to m, so there are finitely many finite data orbits. Let $mem(m)$ be the union of these finitely many finite data orbits. Note that $mem(m) \subseteq X$.

We now show that if a data renaming τ is the identity on $mem(m)$, then $\tau m = m$.

For two data values d, e we write τ_{de} for the data renaming that swaps d with e, and leaves all other data values intact. In other words, this is a transposition. We will prove that for every $d \notin mem(m)$ and every $e \notin X$, the transposition τ_{de} has m as a fixpoint. It follows that any data renaming that is the identity on $mem(m)$ has m as a fixpoint, since such data renamings are generated by transpositions of elements not in $mem(m)$ with elements not in X.

▶ **Lemma 21.** *For every $d \notin D$ there is some $e \notin X$ such that m is a fixpoint of τ_{de}.*

Proof. As usual, a permutation can be decomposed into cycles. For a data value f and a data renaming τ, we use the name cycle generated by f under τ for the sequence

$$f, \tau f, \tau^2 f, \ldots, \tau^n f$$

where n is the smallest number with $\tau^{n+1} f = f$. Since d is not in $mem(m)$, it can be mapped to any data value outside X by a data renaming with fixpoint m. Let d_2 be such a data value, and τ such a renaming. Consider the cycle $d = d_1, d_2, \ldots, d_n$ generated by d under the data renaming τ. This cycle does not contain any values from $mem(m)$, because otherwise d would in the same data orbit as some element of $mem(m)$, and therefore also inside $mem(m)$. Consider now some data value $e_1 \notin X \cup \{d_1, \ldots, d_n\}$. Because $d_2, e_1 \notin X$, it follows that m is a fixpoint of the transposition $\tau_{d_2 e_1}$. Let σ_1 be the data renaming obtained by composing τ with the transposition $\tau_{d_2 e_1}$. Clearly, m is a fixpoint of σ_1. The cycle, under σ_1, generated by d is now $d_0, e_1, d_1, \ldots, d_{n-1}$. We can iterate this construction any number $k \in \mathbb{N}$ of times, yielding data values $e_1, \ldots, e_k \notin X$ and a data renaming σ_k with fixpoint m such that the cycle of d under σ_k is

$$d_1, e_1, \ldots, e_k, d_2, \ldots, d_n. \tag{4}$$

The other cycles of the permutation σ_k are the same as for τ, regardless of k. Let q be the least common multiple of the lengths of the other cycles. This number does not depend on k. If i is a multiple of q, then $(\sigma_k)^i$ is the identity on all elements that do not participate in the cycle (4). If i is half of the length of the cycle (4), assuming that length is even, then the cycle generated by d_1 under $(\sigma_k)^i$ has length 2. Finally, if k is large enough, then that cycle maps d_1 to one of the elements $e_1, \ldots, e_k \notin X$, namely the one in the middle of the cycle. By combining these observations, we can find i and k so that $(\sigma_k)^i$ is the transposition that swaps $d = d_1$ with one of the elements $e_1, \ldots, e_k \notin X$. ◀

We now complete the proof of the Memory Theorem. As remarked above, it suffices to show that for every $e' \in X$, the transposition $\tau_{de'}$ has fixpoint m. We have

$$\tau_{de'} = \tau_{ee'} \cdot \tau_{de} \cdot \tau_{ee'}.$$

The transposition $\tau_{ee'}$ has m as a fixpoint, because $e, e' \notin X$. Therefore, also $\tau_{de'}$ has m as a fixpoint.

This concludes the proof of the Memory Theorem. The proof uses the assumption that data renamings can change only a finite number of data values. The Memory Theorem does hold, however, in the more general case where data renamings are allowed to change infinitely many data values. In the proof of the more general case, one needs to consider infinite cycles, which are more similar to copies of the integers. One can then show that finite cycles can be "cut out" from such infinite cycles.

D Proof of Theorem 6

Let X be a set of data values. For a data monoid M, define $M[X]$ to be

$$M[X] = \{m \in M : mem(m) \subseteq X\}.$$

By Lemma 5, $M[X]$ is a submonoid of M, i.e. it is closed under composition. Note that the subset $M[X]$ is not preserved under data renamings, so it does not have the structure of a data monoid, at least not in the sense inherited from M.

▶ **Lemma 22.** *Let τ be a data renaming. For any m in a data monoid,*

$$\mathrm{stab}_{\tau m}(\tau d) = \tau(\mathrm{stab}_m(d)).$$

Theorem 6 follows from the following lemma, since any finitely generated submonoid of M is included in $M[X]$, where X is the union of memories of the generators.

▶ **Lemma 23.** *In an orbit finite data monoid, $M[X]$ is finite for any finite set $X \subseteq D$.*

Proof. Suppose that $M[X]$ is infinite. Then it contains an infinite set S elements in the same orbit which have the same memory $Y \subseteq X$. Let $m \in S$ be some chosen element. Since all elements in S are in the same orbit, for any $n \in S$ there is some data renaming τ_n with $\tau_n n = m$. By Lemma 22, τ_n maps Y to Y. By infinity of S, there must be $n \neq k \in S$ such that τ_n and τ_k induce the same permutation on Y. It follows that $\tau_n^{-1} \cdot \tau_k$ is the identity on Y. Furthermore, this data renaming maps k to n. By the Memory Theorem, $k = n$, a contradiction. ◀

Preliminary work indicates that local finiteness holds under much weaker conditions than being orbit finite. Namely, for every language of data words that is defined in monadic second-order logic with order and data equality, the syntactic monoid is locally finite. This covers, for instance, languages recognized by nondeterministic register automata.

E Proof of the Memory Theorem for $\mathcal{J}$-classes

Proof. Let d, e be any two data values outside $mem(J)$. We will show that the transposition τ_{de} satisfies $\tau_{de} J = J$. This is enough to prove the theorem, since these transpositions generate all data renamings that are the identity on $mem(J)$. (Recall the assumption that data renamings change only a finite number of data values.)

By assumption on d, e, there must be some $m, n \in J$ with $d \notin mem(m)$ and $e \notin mem(n)$. Let f be some data value outside $mem(m) \cup mem(n)$. By the Memory Theorem, the transposition τ_{df} preserves m and the transposition τ_{ef} preserves n. Any automorphism τ that preserves some element of J must preserve J, i.e. satisfy $\tau J = J$. It follows that the transpositions τ_{df} and τ_{ef} preserve J. Therefore also τ_{de} preserves J, since it is the composition $\tau_{de} = \tau_{df}\tau_{ef}\tau_{df}$. ◀

F Proof of Lemma 10

Proof. Suppose that τJ is comparable to J, say $\tau J \subseteq MJM$. By iterating the above, we see that $\tau^i J \subseteq MJM$ holds for any number $i \in \mathbb{N}$. Since τ permutes only a finite number of data values, there is some i such that τ^i is the identity function. Therefore,

$$J = \tau(\tau^{i-1} J) \subseteq \tau(MJM) = M \cdot \tau J \cdot M,$$

which shows that J and τJ are the same $\mathcal{J}$-class. ◀

G Proof of Lemma 13

Proof. For every $E \subseteq A - B$, we will write a formula φ_E of first-order logic such that φ_E is satisfied by a word w if and only if $w = \tau v$ for some $v \in L$ and $\tau \in \Gamma_B$ such that E are exactly the data values from $A - B$ that appear in v. Observe that $\Gamma_B L$ is the union, over all sets $E \subseteq A - B$ of the languages defined by φ_E.

Let φ be a formula defining L. Let $E = \{a_1, \dots, a_k\}$. Let $x_1, \dots, x_k$ be variables that do not appear in φ. We define φ_E to be the formula

$$\exists x_1 \cdots \exists x_k \qquad \bigwedge_{i \neq j \in \{1,\dots,k\}} x_i \not\sim x_j \quad \wedge \quad \bigwedge_{\substack{i \in \{1,\dots,k\} \\ b \in B}} \neg b(x_i) \quad \wedge \quad \varphi'(x_1, \dots, x_n)$$

where $\varphi'(x_1, \dots, x_n)$ is obtained from φ by doing the following subformula replacements for every variable y and every $a \in A - B$.

- If $a = a_i$ for some $i \in \{1, \dots, k\}$, replace every occurrence of $a(y)$ by $x_i \sim y$.
- Otherwise, replace every occurrence of $a(y)$ by a false formula.

◀

H Proof of Lemma 15

Decompose the set K_n as the union $K_n = K_{1n} \cup K_{2n}$, where

$$K_{1n} = \{wd : h(w) >_{\mathcal{J}} n, d \in D, h(wd) = n\} \text{ and}$$
$$K_{2n} = \{dw : h(w) >_{\mathcal{J}} n, d \in D, h(dw) = n\}.$$

We will show a first-order formula for the set K_{1n} that uses data predicates from $mem(n)$. The formula for K_{2n} is symmetric. For a data value d, define

$$X_{dn} = \{wd : h(w) >_{\mathcal{J}} n, h(wd) = n\}.$$

Of course, we have

$$K_{1n} = \bigcup_{d \in mem(n)} X_{dn} \quad \cup \quad \bigcup_{d \notin mem(n)} X_{dn}. \tag{5}$$

▶ **Lemma 24.** *For every $e \notin mem(n)$, we have*

$$\bigcup_{d \notin mem(n)} X_{dn} \quad = \quad \Gamma_{mem(n)} \cdot X_{en}.$$

By applying the above lemma to (5), we see that for every $e \notin mem(n)$,

$$K_{1n} = \bigcup_{d \in mem(n)} X_{dn} \quad \cup \quad \Gamma_{mem(n)} \cdot X_{en}.$$

Thanks to Lemma 13, the language on the right hand side is defined in first-order logic with data predicates from $mem(n)$, as long as X_{en} is defined by some first-order formula, possibly using data predicates outside $mem(n)$. By closure of first-order logic under finite union, it remains to show the following lemma.

▶ **Lemma 25.** *For every $d \in D$, the language X_{dn} is definable in first-order logic, with constants $mem(n) \cup \{d\}$.*

Proof. Consider the set

$$Y_{dn} = \{w : h(w) >_{\mathcal{J}} n, h(wd) = n\}.$$

By induction assumption, this set is definable by a first-order formula with constants $mem(n) \cup \{d\}$. The set X_{dn} is the concatenation $Y_{dn} \cdot d$, so it is also definable by a first-order formula with constants $mem(n) \cup \{d\}$. ◀

I Proof of Lemma 16

Let $R \subseteq M$ be the $\mathcal{R}$-class of m. Define

$$B = \bigcap_{n \in R} mem(n).$$

By the same reasoning as in the Memory Theorem for $\mathcal{J}$-classes, any data renaming $\tau \in \Gamma_B$ satisfies $\tau R = R$. Two elements $x, y \in M$ are called B-dependent if $x = \tau y$ for some $\tau \in \Gamma_B$. This is an equivalence relation.

▶ **Lemma 26.** *The set R has finitely many equivalence classes of B-dependence.*

Proof. Suppose that S is an infinite subset of R where every two distinct elements are not B-dependent. By orbit finiteness, we may assume that all elements of S are in the same orbit. Fix some element $x \in S$. Since all of S is in the same orbit as x, for every $y \in S$ there is some τ_y with $x = \tau_y y$. For each y, consider the set $\tau_y(B)$. Since $B \subseteq mem(y)$, it follows that for every $y \in S$,

$$\tau_y(B) \subseteq mem(x).$$

Therefore, there must be some $y, z \in S$ such that τ_y and τ_z are the same function when restricted to B. Note that $y = \tau_y^{-1}\tau_z z$. Furthermore, the mapping $\tau_y^{-1}\tau_z$ is the identity on B. It follows that y and z are B-dependent. ◀

From the above lemma it follows that there is a subset $\{m_1, \ldots, m_k\} \subseteq R$ such that

$$R \subseteq \Gamma_B \cdot \{m_1, \ldots, m_k\}.$$

The converse inclusion follows from $R = \Gamma_B \cdot R$.