Regularity as seen by Alice and Bob

February 18, 2026

Abstract

The goal of this paper is to propose a machine-independent characterisation of functions computable by finite-state automata. Inspired by communication complexity, we introduce a model of computation that defines functions of type $\Sigma^* \to \mathbb{D}$, where Σ is a finite alphabet and $\mathbb{D}$ is some domain. Domains of interest include: the Booleans, strings, or a field. In the model, the input string w is split into two parts $w = w_1 w_2$, which are sent to two parties, Alice and Bob. The two parties cooperate, exchanging a constant number of messages to compute the output of the function. The messages can be either bits, or elements of the output domain. They must produce the correct output regardless of the split $w = w_1 w_2$. For some domains, the model coincides with known finite state models: in the case of Boolean outputs it defines exactly the regular languages, and in the case of fields, it defines exactly the functions computable by weighted automata. We also conjecture that a similar situation arises for other domains, and present ample supporting evidence.

1 Introduction

This paper is motivated by a desire to understand the notion of regularity in formal language theory. We take the functional perspective, in which we consider functions

$$f : \Sigma^* \to \mathbb{D}$$

that input strings, and output values from some domain $\mathbb{D}$. If the output domain is the Booleans, then such functions are languages, and there is no question about which languages should be considered *regular*. There are tens – if not hundreds – of equivalent definitions, including regular expressions, finite automata in numerous forms, monoids, monadic second-order logic, and variants of λ-calculus. But what about other outputs? Let us review three examples where the nature of regularity is a topic of genuine discussion.

1. **String outputs.** Consider string-to-string functions

   $$f : \Sigma^* \to \Gamma^*.$$

   Similarly to languages, the literature on automata theory offers countless models. This time, however, not all of them are equivalent, but there is at least some semblance of order. Let us mention three classes of functions of particular interest: the *rational*, *regular*, and *polyregular* functions. These classes are described in Fig. 1 in Section 5 together with the appropriate references, with each one having at least five different characterisations, using models of varied origins, including logic, algebra and programming language theory. Which of these classes, if any, should be considered "the" regular string-to-string functions? We could simply go with the middle one, because the word "regular" is traditionally used for it, but a more principled approach would be preferable.

2. **Number outputs.** Consider string-to-number functions, say functions

   $$f : \Sigma^* \to \mathbb{Q}$$

   that output rational numbers (more generally, the outputs could be in some field). Here, the literature offers two natural candidates, namely *weighted automata* [Sch61], and *polynomial automata* [BDSW17, Section IV]. In both cases, there is an automaton that reads the input string in one pass, and stores in its state a vector $\mathbb{Q}^d$ of some fixed dimension. In weighted automata, this vector is updated using linear maps, while polynomial automata can use polynomial maps. These models are not equivalent – polynomial automata are strictly more powerful – but both have a good mathematical theory, one based on linear algebra, and the other based on algebraic geometry. Again, we might be tempted to ask: which is the right one?

3. **Infinite input alphabets.** Our final example is of a different kind than the previous ones. We return to functions with Boolean outputs

   $$f : \Sigma^* \to \{\text{true}, \text{false}\},$$

   i.e. languages, but this time the input alphabet is no longer required to be finite. The infinity needs to be somehow tamed, and the standard approach to do this is to use an infinite alphabet where letters can only be compared for equality [KF94]. This allows for languages such as "all

letters are different" or "the first letter is equal to the last one", but not for languages such as "the letters are in increasing order", since the letters are not equipped with a linear order, or any other kind of structure. The literature is rife with automata models for such languages, with seventeen examples listed in Fig. 2, all describing pairwise non-equivalent models. Again, we might be tempted to ask: which is the right one?

This type of question can be asked in other settings, with outputs such as trees, graphs or elements of some abstract semiring. One could also vary the inputs, and consider regularity for, say, graph-to-graph functions, but we refrain from considering general outputs in this paper, and we stay with string inputs. This paper attempts to provide a unified theory of regularity for such functions. We are guided by the following principle, which we believe to be essential for regularity:

Constant information flow. If the input is split into parts, then only a constant amount of information flows between them, as far as the output of the function is concerned.

This principle is only a vague guideline, since it does not identify what "information flow" means, or how to quantify its amount. For Boolean outputs, i.e. languages, constant information flow has a standard interpretation, which is the Myhill-Nerode Theorem, and it is known to correspond exactly to the regular languages. However, in the case of more complicated outputs, things are less clear. For example, if the outputs of the function are rational numbers, then it should be legitimate for the information flow to contain some rational numbers. How should this be formalised?

We choose to use a formalisation that is based on communication complexity [Yao79, Kus97]. In our model, the function is computed by two cooperating parties, called Alice and Bob. There are **no uniformity assumptions** and the two parties have unrestricted computational power; the goal is to measure information and not computation. The input string is split into two parts $w = w_1w_2$, and Alice has access to w_1 while Bob has access to w_2. The two parties exchange a **constant number of messages** in order to compute the function, where the constant depends only on the protocol and not on the input string. The output of the computation must be **split invariant**, which means that the output depends only on the input string w, and not on the split $w = w_1w_2$. In the communication, there are two kinds of messages: either bits in $\{\text{true}, \text{false}\}$ or elements of the output domain (as far as we can tell, the messages from the domain are the novelty of our approach). For example, if the outputs are rational numbers, then the messages can contain rational numbers. However, there is a restriction on the access to messages from the output domain, which is called the **black box restriction**. This restriction (which will be formally defined later in the paper) is intended to prevent tricks such as Alice sending her input string to Bob encoded as a rational number. Intuitively speaking, the black box restriction says that the parties cannot read the messages from the output domain, and instead they can only operate on them using predefined operations. For example, in the case of numbers, the operations are addition and multiplication.

The model, which we call *protocols*, can be applied to any output domain, and we study several examples in this paper. In the case of Boolean outputs, the model and its connection to finite automata have already been studied before [Hau89, Theorem 5], however the results on other output domains are new up to our best knowledge. As we discover, despite its non-uniformity, the model can only define well-behaved functions. In particular, in all cases that we have studied these functions are: (1) always computable, even in linear time; and (2) the output size is always at most linear (with the size of a rational number measured by the number of bits needed to represent it). This seems to indicate that the split invariance, together with a constant number of messages under the black box restriction, has unexpected computational consequences. In particular, in all cases

that we have studied, protocols coincide – or are conjectured by us to coincide – with existing automata models. Since automata are conceptually very different than protocols, we believe that those equivalences, summarized in the table below, justify the protocol-based approach to regularity.

Output	**Automata Model**
Booleans	Finite automata (Theorem 1.1)
Field	Weighted automata (Theorem 1.2)
Strings	Two-way automata with output (Conjecture 1.3)
Boolean, but infinite input alphabet	Unambiguous automata (Conjecture 6.5)

In the remainder of this introduction, we give a more detailed review of the four rows in the table, including substantial evidence for the conjectured equivalences.

1.1 Protocols for Boolean outputs

We begin by studying the protocol model for functions

$$f : \Sigma^* \to \{\text{true}, \text{false}\},$$

i.e. languages. This case is not new, and it has already been studied in [Hau89]. For Boolean outputs, the two parties only exchange bits, and they need to decide if the input string is in the language or not. Here is an example.

Example 1. [Parity] Suppose that the language is "the string has even length". In a protocol for this language, Alice sends the parity for her part of the input (one bit), and Bob uses this bit to return the final answer. □

As mentioned before, the parties have unbounded computational power, which means that their messages can contain answers to potentially undecidable questions about their parts of the input. Despite this, the protocols compute exactly the regular languages. This was first shown in [Hau89, Theorem 5], but we restate it here for completeness, and provide a self-contained proof later in the paper.

Theorem 1.1. *A language $L \subseteq \Sigma^*$ is computed by a protocol if and only if it is regular.*

One implication is immediate: every regular language can be computed by a protocol. Alice can send to Bob the state of a finite automaton that recognises the language. The other implication is proved in two steps, see Section 2 for more details. In the first step, the protocol is reduced to a one-round non-interactive version, where each party independently sends a message with a constant number of bits, and the decision is then made based on these two messages. In the second step, the Myhill-Nerode Theorem is used to show that the language must necessarily be regular.

Theorem 1.1 is simple technically, and its main value for us lies in its role as an inspiration for other results, which use other output domains such as fields or strings.

Before moving to the other output domains, let us comment on other related work that connects communication complexity with automata in the case of Boolean outputs. Much of this work is related to *state complexity*, where one studies the number of states needed for a given language in a given automaton model, and how this number is affected by operations on languages or changes in the model. See Wikipedia [sta] for a comprehensive summary with numerous references, or the recent paper [GKY22] which shows how to transfer lower bounds from communication complexity

to state complexity of unambiguous automata. The work on state complexity is mainly about the exact number of states, which is of secondary concern to us. For our purposes, a protocol that exchanges k bits is no different from a protocol that exchanges 2^k bits. We only care that this bound should be finite and independent of the input.

1.2 Field outputs

After the Booleans, we turn to functions with outputs in a field. This is the first original contribution of this paper. For the sake of concreteness, let us consider functions with outputs in the field of rationals

$$f : \Sigma^* \to (\mathbb{Q}, +, \times).$$

We adapt the protocol model to compute such functions. Similarly to the Boolean case, the parties exchange messages. However, this time there are two kinds of messages: bits (as in the case of Boolean outputs), and elements of the field. The elements of the field can be added and multiplied. Division is not allowed, and its role is discussed in Example 3. Using bit messages, we can still recognise all regular languages (more formally their characteristic functions). However, messages from the field allow computing new functions.

Example 2. [Length and exponential length] Consider the following two functions

$$\underbrace{w \mapsto |w|}_{\text{length}} \qquad \text{and} \qquad \underbrace{w \mapsto 2^{|w|}}_{\text{exponential length}}.$$

To compute the length of the input string, Alice sends the length of her part, and Bob adds this to his length, thus yielding the desired output. For the exponential length, we use a similar protocol, except that multiplication is used instead of addition. □

In the presence of an infinite message space, one needs to be careful about the design of the protocol. For example, Alice could try to send her part of the input string in a single message by encoding it as a rational. This would trivialise the model, enabling every function to be computed. To prevent such tricks, we use the *black box restriction* which was discussed before[1]: the messages which use the output domain (in this case, rational numbers) cannot be read directly, but can only be acted on by the operations in the output domain (in this case addition and multiplication). If the output domain is finite, e.g. it is a finite field, then the black box restriction is irrelevant (which is why it was not mentioned when talking about Boolean outputs). This is because one can use bits to sent elements of a finite output domain, and the bits are a preferrable communication channel, since they can be read directly and are not subject to the black box restriction.

Definition of the protocol model. Since there might be some ambiguity as to what exactly is allowed in the protocol, we give a more formal definition. There is a finite input alphabet Σ, and a constant number of rounds $k \in \{1, 2, \ldots\}$. Alice and Bob send messages in alternation, with Alice

[1]The black box postulate is related to polymorphic parametricity from the theory of programming languages [Rey83, Section 7], or to the recent algebraic group model in cryptography [FKL18, Section 1.2]. An important difference with the algebraic group model is that our model does not allow for equality tests, see Example 12 for a discussion.

sending the first message[2]. The messages are from

$$\underbrace{\{\text{true, false}\} + \mathbb{Q}}_{\text{disjoint union of bits and rational numbers}}.$$

When choosing their message in the i-th round, the corresponding party (Alice in odd rounds, Bob in even rounds) has access to their part of the input string, and the bits from previous messages. The numbers are rationals, and cannot influence the decision, as per the black box restriction. The information available in the i-th round is given by the set

$$\underbrace{\Sigma^*}_{\substack{\text{part of the input}\\ \text{string that is}\\ \text{known to the party}}} \times \underbrace{\{\text{true, false, unknown}\}^{i-1}}_{\substack{\text{messages received in previous rounds,}\\ \text{with numbers from } \mathbb{Q} \text{ replaced by ``unknown''}}}. \tag{1}$$

Based on this information, the corresponding party chooses a new message to send, which is either a bit, or a number. The number can be produced in two ways: either a fresh number is produced based on the available information, or otherwise two previously received numbers are combined using addition or multiplication. Therefore, the possibilities for the message sent in the i-th round are described by the set

$$\underbrace{\{\text{true, false}\}}_{\text{bits}} + \underbrace{\mathbb{Q}}_{\substack{\text{fresh}\\ \text{number}}} + \underbrace{\{(op, x, y) \mid \ op \in \{+, \times\} \text{ and } x, y \in \{1, \ldots, i-1\} \ \}}_{\text{addition or multiplication of previously received numbers}}. \tag{2}$$

If addition or multiplication is used, then the party sending the message is responsible for the operation to be well-defined, i.e. the messages sent in rounds x and y must have been numbers. Summing up, the strategy in round i is a function which inputs an element of the set from (1), and outputs an element of the set from (2). This function need not be computable. The protocol is then described by k such strategies, one for each round $i \in \{1, \ldots, k\}$. We assume that the last message, sent in the k-th round, is a number and not a bit – this number is defined to be the output of the protocol. Finally, the protocol must be split invariant, i.e. for every input string w, the same output must be produced regardless of the factorization $w = w_1 w_2$ into strings owned by Bob and Alice. This completes the definition of the protocol model, in the case of field outputs.

Equivalence with weighted automata. Our main result for field outputs is Theorem 1.2 below, which says that protocols are equivalent to weighted automata. The precise definition of weighted automata will be given later in Section 4, but the rough idea is that a weighted automaton maintains a vector $\mathbb{D}^d$ of field elements, with each input letter updating the vector via some fixed linear map.

Theorem 1.2. *Assume that the domain $\mathbb{D}$ is a field. Then a function*

$$f : \Sigma^* \to \mathbb{D}$$

is computed by a protocol if and only if it is computed by a weighted automaton over the same field.

[2]One could consider other patterns of communication. In fact, in Section 3 we use a more symmetric variant where both parties move in parallel in each round. These variants do not change the expressive power of the model, only the number of rounds.

This result might even seem surprising. Our protocol is designed to use polynomial operations on the output domain, and therefore one could expect the relevant automaton model to be also based on polynomials, such as the polynomial automata of [BDSW17], which are the extension of weighted automata that allow polynomial maps instead of linear ones. As it turns out, the split invariance in the protocols enforces linearity, and thus it excludes the general polynomials operations that are used in polynomial automata. The linearity phenomenon is true for outputs in a field – because weighted automata are based on linear maps – and it will also be true for other output domains, such as strings, see Theorem 1.4. We do not have a fully general understanding of this phenomenon.

The proof of Theorem 1.2 is given in Section 4.1, but here we present a rough outline. The proof is similar to the one used for the case of Boolean outputs, and has two steps:

1. We first show in Section 4.1.1 that any protocol with outputs in a field can be reduced to a special form, which we call a *scalar product protocol*. In this protocol, Alice and Bob apply in parallel two functions

$$\sigma_A, \sigma_B : \Sigma^* \to \mathbb{Q}^d$$

   to their parts of the input string, where d is some fixed finite dimension. Then, the output is obtained by combining these two d-dimensional vectors using scalar product. A scalar product protocol can be simulated by the general version of the protocol, but it is subject to certain restrictions: (a) there is no interaction; and (b) bits are not used, only field elements. Since, as we prove, every product can be reduced to this scalar form, it follows that interaction and bit messages are not needed in the protocol. In fact, the interaction can be removed for all output domains, but the removal of bits is specific to fields.

2. After reducing to the scalar product protocols, the next step (see Section 4.1.2) is to apply a version of the Myhill-Nerode Theorem for weighted automata. This is called the Fliess Theorem [Fli74], and it says that recognisability by a weighted automaton is equivalent to having finite rank for a certain matrix, which is called the Hankel matrix. Roughly speaking, the rows in the Hankel matrix, in the context of our protocols, describe strategies of Alice, and the columns describe strategies of Bob. Therefore, it is not hard to show that in a scalar product protocol that uses vectors of dimension d, the Hankel matrix has rank at most d. This, together with the Fliess Theorem, shows that protocols are equivalent to weighted automata, completing the proof of Theorem 1.2.

Example 3. [Division] What if we added division to the operations? Consider the function $w \mapsto 1/(|w|+1)$. This function can easily be computed using a protocol with division. We now argue that it cannot be computed using addition and multiplication only, thus proving that division gives extra power. Since the function depends only on the length of the input, it can be seen as having type $\mathbb{N} \to \mathbb{Q}$. In such a type, weighted automata are the same as linear recurrence sequences. The inverse function $1/(n+1)$ is not a linear recurrence sequence, which can be shown using the exponential polynomial form [BR08, Theorem 2.1]. Summing up, the choice of operations is important; we use a field, but we only allow the ring operations. We do not know what happens if division is allowed. □

Related work. Theorem 1.2 can be seen as a machine independent characterisation of weighted automata. This would not be the first such characterisation, e.g. the Fliess Theorem itself can be seen as a machine independent characterisation. Other research related to the Fliess Theorem is the categorical approach to minimisation of weighted automata from [CP17]. We think that the value of our approach is that it places weighted automata in a broader context, which is defined purely in terms of communication, and in a way that is applicable to other output domains, such as strings that will be considered next. As far as we know, the only work which takes such a broad view is the cost register automata of [ADD$^+$13, Section C], which are an automaton model that describes functions with outputs in an arbitrary output domain, similarly to our setting. However, unlike our model, cost register automata are defined in terms of a finite state machine model, and as such they lack the abstract machine independent flavour of our approach.

1.3 String outputs

Our third group of results concerns string-to-string functions

$$f : \Sigma^* \to \Gamma^*.$$

We use the same kind of protocol as in the previous section, except that instead of numbers, the black box messages contain strings from Γ^*, and instead of addition and multiplication, we have string concatenation. Here are some examples.

Example 4. [Reversal and duplication] Using the protocol, we can compute string reversal: Alice sends to Bob the reverse of her part of the input, and Bob concatenates this with his part of the reverse. Another string-to-string function that can be easily computed by a protocol is string duplication $w \mapsto ww$. □

The string-to-string case is of particular interest because, as we have mentioned earlier in the introduction, there is no consensus as to which string-to-string functions should be considered "regular". There are numerous automata models to choose from, some of which are summarized in Fig. 1, which contains twenty models, grouped by equivalence into three classes. We can exclude the weakest class (the rational functions), since it is too weak: it cannot compute the reverse or duplicate functions from Example 4. We can also exclude the strongest class (the polyregular functions), since it is too strong: polyregular functions can have superlinear output size, and as we will see in a moment, protocols can only have linear output size. By a process of elimination, we are left with the final class from Fig. 1, which is traditionally called the "regular functions". One of the definitions of the class is in terms of deterministic two-way automata with output [She59]; this model is formally defined in Section 5. We conjecture that this class is the correct answer (thus validating the traditional name):

Conjecture 1.3. *A string-to-string function is computed by a protocol iff it is computed by a deterministic two-way automaton with output.*

In Section 5 we discuss this conjecture in detail, and provide evidence in its favour, including a proof of the implication

$$\text{protocol} \quad \Longleftarrow \quad \text{two-way automaton},$$

This implication is rather easy, since a two-way automaton can be neatly simulated by the repeated interactions of the protocol. The content of the conjecture, and the subject of the more technical

results, is the $\implies$ implication. Most of Section 5 is devoted to evidence for this implication. Our first argument is the following result, which shows that functions computed by protocols have many properties which are known to hold for deterministic two-way automata with output.

Theorem 1.4. *If a string-to-string function is computed by a protocol, then:*

1. *outputs have at most linear size;*
2. *outputs can be computed in linear time (ignoring logarithmic factors);*
3. *preimages of regular languages are regular.*

One can invent functions which satisfy the three conditions in the above theorem, but which are not computed by deterministic two-way automata with output, see Example 15. However, all known examples of such functions are artificial, and none can be computed by protocols (or any known transducer models). The proof of Theorem 1.4, which is given in Section 5, uses a linear representation of strings as matrices, and then applies Theorem 1.2 about protocols with field outputs. In fact, this technique suggests an alternative approach to regularity, which connects string-to-string functions with the better understood case of string-to-field functions. This approach is discussed in Section 5.2.

Our second argument in favour of the Conjecture 5.8 is that we can prove it in the special case when the output alphabet is unary, i.e. it has only one letter. This is the content of Section 5.3. When the output alphabet is unary, the output strings are commutative, i.e. the order of letters is irrelevant; this commutativity is essential to our proof of the conjecture in the unary output case. The proof is based on well-quasi orders, plus some results on weighted automata.

Related work. One of the consequences of the Myhill-Nerode theorem for string-to-Boolean functions, or the Fliess Theorem for string-to-number functions, is the existence of canonical devices. There have been several attempts to generalise this to string-to-string functions, with a special emphasis the canonical devices. Before recalling this work, we observe that our approach seems to go in a different direction. Although we think of Conjecture 1.3 as being a machine independent characterisation, it does not necessarily yield canonical devices. In particular, our proof of the conjecture for unary alphabets does not yield a canonical device.

Here is a summary of results on canonical devices for string-to-string transducers: they have been proposed for subsequential functions [Cho77, Théorème 1.1], rational functions in [RS91, Theorem 1], and for rational functions on infinite words [FGLM18, Section 4]. A certain drawback of this line of work is that: (a) the canonical devices are relative to a given automaton model, which does not help in choosing one model over another; and (b) the "canonical" devices are not truly unique, since they depend on extra parameters, such as the output delay for subsequential functions or the lookahead for rational ones. Let us now move to the larger class of regular functions, which is the subject of our conjecture. Here, canonical model are unknown, and the only known way to recover them uses non-standard semantics, called origin semantics [Boj14, Theorem 1]. Another result of this kind, which is a machine independent characterisation of the regular functions that does not yield canonical devices, see [BN23, Theorem 3.2], is also implicitly based on origin semantics. Finally, for the polyregular functions, the situation is of course even harder, and the only known results concern a unary output alphabet [CDTL23, Section IV].

1.4 Infinite input alphabets

Our final group of results is about languages over infinite input alphabets. This is a departure from the previous setting, where the input alphabet was always finite. Following the standard approach in automata theory, we assume that letters can only be compared for equality. Formally, we only want to consider languages that are invariant under permutations of the alphabet, i.e.

$$w \in L \quad \iff \quad \pi(w) \in L \qquad \text{for every permutation } \pi \text{ of the alphabet.}$$

An example of such a language is "all letters are different", but not "the letters are in the increasing order". As mentioned earlier in the introduction, there is a rich literature on automata for such languages, see the surveys [NS03, Seg06, Boj17] or the lecture notes [Boj25b]. The relevant automata models typically use registers to store some letters from the input, so that they can be compared to later letters. Essentially any automaton model for finite alphabets can be lifted this way to infinite alphabets [NS03, Figure 1], and there is even a systematic way to do this, which is based on the theory of orbit-finite sets [Boj25b, Chapter 2]. Unfortunately, after this lifting, previously equivalent models become non-equivalent. This sad situation is illustrated in Fig. 2, which describes seventeen non-equivalent automata models for infinite alphabets; all of these models collapse to the regular languages when considered for finite alphabets. Equally sadly, there are almost no results in the literature on infinite alphabets that prove non-trivial equivalences of models. The only known cases of this kind are about the weakest of the available models, namely orbit-finite monoids [Boj13], which are known to be equivalent in expressive power to a certain variant of MSO [CLP15, Theorems 4.2 and 5.1], and also to single-use register automata [BS20, Theorem 6]. This situation desperately calls for some unifying principles.

Since protocols have successfully identified important models in the previous cases, we try to see what happens in the case of an infinite input alphabet. When extending protocols to infinite input alphabets, we adapt them as follows: (1) messages can contain input letters; (2) input letters can only be compared for equality. Condition (2) is formalised by saying that the execution of the protocol is invariant under permutations of the input alphabet, similarly to the languages that we consider. The protocols work their magic once again, and they point to (as we conjecture) one of the models in the literature. Before revealing this model, let us briefly compare protocols to the two most prominent models for infinite alphabets.

Example 5. [Deterministic too weak] We begin by considering a popular deterministic automaton model for infinite alphabets, namely i.e. *deterministic register automata* [KF94, Definition 3]. This model is defined formally in Section 6, but roughly speaking it is nondeterministic a finite automaton that is additionally equipped with a fixed number of registers, which can be used to store input letters and compare them for equality with later input letters. For example, the automaton can store the first input letter in a register, and then compare it with all later letters, thus recognising the language "the first letter appears elsewhere in the word".

A protocol can simulate any deterministic register automaton, using the same idea as for finite alphabets, i.e. Alice sends the intermediate state, including the register values, to Bob, who continues the run on his side. To see that the inclusion is strict, consider the language "the last letter appears elsewhere in the word", which is the reverse of the language described in the previous paragraph. It is computed by a protocol, in which Bob checks the condition on his side, and sends the last letter to Alice, so that she can verify that it does not appear on her side. On the other hand, it is a well-know fact that this language is not recognised by a deterministic register automata, as it proves that deterministic register automata are not closed under reverse [KF94, Examples 4 and 8]. □

Example 6. [Nondeterministic too strong] Consider the nondeterministic variant of the automata from the previous paragraph [KF94, Definition 1]. This model is already too strong for the protocols, as attested by the language "some letter appears twice", which can be computed by an automaton, see [KF94, Example 1], but not by a protocol. The intuitive reason for the negative result is that if neither Alice or Bob sees a repetition in their parts of the input string, then they should exchange all their letters to check for cross-part repetitions, which cannot be done in a constant number of messages – a detailed proof is given in Section 6. □

Which automaton model, if any, corresponds to protocols? As explained in the previous two examples, deterministic register automata are too weak, while nondeterministic register automata are too strong. We conjecture (in Conjecture 6.5), that the winner is a seemingly unexpected candidate, namely unambiguous register automata [Col15, Section 5]. This is the special case of nondeterministic register automata, in which for every input string there is at most one accepting run. We discuss this conjecture in Section 6, and provide evidence in its favour. We start with an actual proof of one implication, namely:

$$\text{protocol} \quad \Longleftarrow \quad \text{unambiguous automaton.}$$

Contrary to previous variants of this implication, the proof is non-trivial – the usual construction does not work, because the automaton is nondeterministic. One interesting phenomenon is that, in the case of infinite input alphabets, the interactive multi-round nature of the protocols becomes essential, and protocols cannot be reduced to one round, as was the case for finite input alphabets. In our proof of the implication $\Longleftarrow$, we design a protocol where the two parties progressively eliminate the uncertainty about letters used in the run of the automaton, until the unique accepting run is identified or its existence disproved. The proof uses a variant of the sunflower lemma.

Therefore, the content of the conjecture is – as in previous cases – the other implication, namely that protocols can be simulated by unambiguous register automata. We provide evidence for this implication, using the recently developed theory of orbit-finite vector spaces [BFKM24]. We show that every function computed by a protocol can be computed by a weighted automaton with registers. This is almost like an unmabiguous automaton, except that some runs might have negative weights, and the weights always cancel out to give a final result that is either 0 or 1. In particular, the functions computed by protocols are computable, which is not a priori clear from the model. Along the way, we develop some new theory, in particular an orbit-finite generalisation of the Fliess Theorem.

2 Boolean outputs

In this section, we formally describe our model of computation for the simplest output domain, namely the Booleans, and we prove that it defines exactly the regular languages. (As mentioned in the introduction, this result was already shown in [Hau89].) The definition that we describe, see Definition 2.1 below, has minor differences with respect to the informal description from the introduction. The messages are not necessarily bits, but they belong to some finite set, which is fixed in advance before the input is known. Also, the two parties send their messages in parallel in each round. These generalisations do not change the expressive power of the protocol (they might influence the number of rounds), but they will be useful in later sections, when we consider restrictions and generalisations.

Definition 2.1 (Boolean protocol)**.** *A* Boolean protocol *is given by the following ingredients:*

1. *a finite input alphabet* Σ*;*
2. *a number of rounds* $k \in \{1, 2, \ldots\}$*;*
3. *message spaces for Alice and Bob, which are finite sets* Q_A *and* Q_B*;*
4. *for each round* $i \in \{1, \ldots, k\}$*, two strategies*

$$\overbrace{\sigma_A^i : \underbrace{\Sigma^*}_{\substack{\text{Alice's}\\ \text{local}\\ \text{string}}} \times \underbrace{Q_B^{i-1}}_{\substack{\text{message}\\ \text{history}}} \to \underbrace{Q_A}_{\substack{\text{new}\\ \text{message}}}}^{\text{stategy for Alice in the } i\text{-th round}} \quad \text{and} \quad \overbrace{\sigma_B^i : \underbrace{\Sigma^*}_{\substack{\text{Bob's}\\ \text{local}\\ \text{string}}} \times \underbrace{Q_A^{i-1}}_{\substack{\text{message}\\ \text{history}}} \to \underbrace{Q_B}_{\substack{\text{new}\\ \text{message}}}}^{\text{stategy for Bob in the } i\text{-th round}};$$

5. *an output function of type* $(Q_A \times Q_B)^k \to \{\text{yes, no}\}$*.*

Given an input string $w \in \Sigma^*$ and a split $w = w_1 w_2$ into two strings, which are called the *local strings* of Alice and Bob, respectively, the protocol is run as follows. There are k rounds. In each round, both parties send messages, and therefore after i rounds are played, the communication history contains i messages sent by Alice and i messages sent by Bob. In round $i \in \{1, \ldots, k\}$, each of the two parties looks at their local string and the $i - 1$ messages sent by the other party in the previous rounds (only the messages sent by the other party are needed, since the party knows their own messages). Based on this information, each party uses their strategy to produce a new message. At the end of the protocol, the output function is used to determine the value of the function, based on all messages in the communication history. We say that the protocol computes a language $L \subseteq \Sigma^*$ if for every string w and every split $w = w_1 w_2$, the output of the protocol tells us if w belongs to the language. This corresponds to the split invariance condition that was discussed in the introduction. Also, the reader will recognise the restriction on the total number of bits (this is bounded by the number of rounds times the logarithm of the size of the message spaces), and the non-uniformity (there is no restriction on the strategies of Alice and Bob). By non-uniformity, the first message sent by Alice could contain an answer to some undecidable problem. However, as we will see, the split invariance restriction will make it impossible to use this information, since the protocol can only compute regular languages, as stated in the Theorem 1.1 from the introduction, which we now recall.

Theorem 1.1. *A language* $L \subseteq \Sigma^*$ *is computed by a protocol if and only if it is regular.*

Proof. We begin with the easier right-to-left implication, which says that the protocol can compute every regular language. If the language is regular, then it is recognised by a deterministic finite automaton, say with state space Q. To compute the language, we can use a *one-round protocol* (as we will see in a moment, this is not a coincidence, since all protocols can be reduced to one round). Alice sends the state in Q of the automaton after reading her local string, and Bob sends the dependency $Q \to \{\text{yes, no}\}$ which says how Alice's state determines acceptance. Once these two pieces of information are known, we can apply the function from Bob's message to the state in Alice's message to determine the output.

The rest of this proof is devoted to the left-to-right implication, i.e. to showing that every language computed by the protocol is regular. We begin by reducing to one round.

Lemma 2.2. *For every protocol, there is an equivalent one-round protocol.*

Proof. The general idea is that instead of engaging in interactive communication, each party sends the dependency of their message upon the unknown messages of the other party. Suppose that we are Alice. The sequence of messages that we send will depend on our local string, and the messages sent by Bob. Once the local string is fixed, this dependency is captured by a function of type

$$(Q_B)^k \to (Q_A)^k,$$

which satisfies the following *causality* constraint: the i-th coordinate of the output depends only on the first $i-1$ input coordinates. Instead of waiting for Bob's messages, Alice sends this function. At the same time, Bob sends an analogous function of type

$$(Q_A)^k \to (Q_B)^k,$$

which describes the dependency of his messages. Due to the causality constraints, the two functions combine to create a unique output in $(Q_A \times Q_B)^k$, which can be used to determine the output of the protocol. □

The proof of the above lemma incurs an exponential cost in the size of the message spaces. This is of little concern to us, since we only care about the protocol having a constant number of rounds. To complete the proof of the theorem, we use the Myhill-Nerode Theorem. Indeed, consider the strategy of Alice:

$$\sigma_A : \Sigma^* \to Q_A.$$

For all we know, this function could be non-computable. However, it classifies all local strings into finitely many categories, with one category for each message in Q_A. Furthermore, if two strings w_1 and w_1' are in the same category, then they are equivalent in the following sense:

$$w_1 w_2 \in L \Leftrightarrow w_1' w_2 \in L \qquad \text{for every } w_2 \in \Sigma^*. \tag{3}$$

The equivalence described above is the same equivalence as in the Myhill-Nerode Theorem. In particular, equivalence classes of this equivalence are the same as states of the minimal deterministic automaton. Since the number of possible messages in Q_A is finite, it follows that the minimal automaton is finite, and therefore the language is regular. (Observe that we have shown that the language has at most Q_A equivalence classes of Myhill-Nerode equivalence, which gives an upper bound on the size of the minimal automaton recognizing the language.) □

In Theorem 1.1, we have seen that protocols with a constant number of bits exchanged define exactly the regular languages. Before continuing, let us briefly discuss what happens if a small, but nonconstant, number of bits is allowed (the observations in this discussion were suggested by Katzper Michno). Already by sending $\log n$ bits, where n is the input length, one can compute any property of the length of the input string, including undecidable properties. This is because $\log n$ bits are enough to get the length of the input string. For one-round protocols, there is nothing between $\log n$ bits and a constant number. Indeed, as mentioned at the end of the proof of Theorem 1.1, the minimal number of bits that can be sent by Alice in a one-round protocol for a string of length n is

$$\log(\underbrace{\text{number of Myhill-Nerode equivalence classes for strings of length at most } n}_{\text{this is called the state complexity of the language}}).$$

For a regular language the state complexity is constant. For a non-regular language, the state complexity cannot be smaller than n. This can be proved similarly to the Morse-Hedlund, see [OP16, Theorem 1.3] for a generalisation which talks about multidmensional strings. We do not discuss this further, since our focus is on regular languages (and their generalisations to other output domains that will be dicussed in later sections), in which only a constant amount of communication is needed.

3 Beyond Boolean outputs

In the previous section, we considered Boolean outputs. In this section, we generalise protocols to account for an arbitrary output domain. The inputs remain unchanged – they will always be strings in this paper. For the output domain, we use a very general notion, namely a set with some operations.

Definition 3.1 (Output domain)**.** *An* output domain *consists of:*

1. *an underlying set $\mathbb{D}$; together with*
2. *a family of* operations*, each one having type $\mathbb{D}^n \to \mathbb{D}$ for some $n \in \{0, 1, \ldots\}$.*

An output domain is the same thing as a (non-indexed) algebra, in the sense of universal algebra [HM88, p.5]. The output domain will typically be infinite. In principle, the family of operations might be infinite as well, although any protocol will only use finitely many operations. By abuse of notation, we use the same symbol $\mathbb{D}$ to denote the output domain and its underlying set, with the operations being implicit. Here are the output domains that will be studied in this paper:

- *Boolean domain.* The set is $\{\text{true}, \text{false}\}$, and there are no basic operations. (We could add basic operations, such as the Boolean operations $\vee, \wedge$ and $\neg$, but this will not affect the expressive power of our protocol, so we choose to have no operations.)
- *Field domain.* The set is a field, such as the rationals or reals, and there are two operations for addition and multiplication. This is not one domain, but a family of domains, with one for each field. Division is not included as an operation.
- *String domain.* The set is Γ^* for some finite alphabet Γ, and the operation is string concatenation.

In the protocol, the output value will be constructed in a constant number of steps, by using operations from the output domain. We will not distinguish between the operations that are in the output domain, and other operations that are derived by composing them, as described in the following definition.

Definition 3.2 (Term operation)**.** *Consider an output domain $\mathbb{D}$. An operation*

$$t : \mathbb{D}^n \to \mathbb{D}$$

is called a term operation *if it can be obtained by applying the operations in the domain to variables $x_1, \ldots, x_n$. Each variable can be used multiple times, or not at all. We write*

$$\mathbb{D}^n \xrightarrow[term]{} \mathbb{D}$$

for the set of term operations with n arguments.

Example 7. Even if the output domain has no operations, the identity operation $\mathbb{D} \to \mathbb{D}$ is allowed, since it is defined by a term that consists of just a variable x_1. □

Example 8. If the operations are $+$ and $\times$, then after applying the usual distributivity laws, term operations are the same as polynomials with natural coefficients, such as

$$x_1 x_2^2 + x_1 x_2 x_3^3 + \underbrace{2x_1}_{\text{same as } x_1 + x_1}.$$

If the output domain is the Boolean domain, which has no operations, then the only kind of term operation is a single variable, e.g. x_2. In the case of a string domain, a term operation is some concatenation of the variables, such as $x_1 x_3 x_1 x_2$. □

We now generalise the protocol to cover functions

$$f : \Sigma^* \to \mathbb{D},$$

where $\mathbb{D}$ is some possibly infinite output domain. As in the Boolean version of the protocol, the output value is constructed by Alice and Bob, as a result of an exchange of a constant number of messages. Each message has two parts, which are called the *signal part* and the *output part*. The signal part consists of a finite amount of information, and is used to exchange information between the two parties as in the Boolean protocol. The output part is a list of elements from the output domain, and is meant to be part of the output value. As in the Boolean case, the above definition slightly deviates from the informal description in the introduction. In particular, a message can contain d elements of the output domain. This does not affect the expressive power of the protocol, but it might affect the number of rounds, and the above definition will be more convenient later one, where we consider one-round protocols. Another important difference is that the operations from the output domain are only applied once at the end of the protocol. This, again does affect the expressive power, since the two parties can wait with their operations until all signals have been exchanged.

Definition 3.3. *A* two-party protocol *is given by the following ingredients:*

1. *an output domain* $\mathbb{D}$*;*
2. *a finite input alphabet* Σ*;*
3. *a* number of rounds $k \in \{1, 2, \ldots\}$*;*
4. *signal spaces for Alice and Bob, which are finite sets* Q_A *and* Q_B*;*
5. *a* dimension $d \in \{0, 1, \ldots\}$*;*
6. *for each round* $i \in \{1, \ldots, k\}$*, a* strategy

$$\overbrace{\sigma_A^i : \underbrace{\Sigma^*}_{\substack{\text{Alice's}\\ \text{local}\\ \text{string}}} \times \underbrace{Q_B^{i-1}}_{\substack{\text{history of}\\ \text{signals}\\ \text{from Bob}}} \to \underbrace{Q_A \times \mathbb{D}^d}_{\substack{\text{new}\\ \text{message}}}}^{\text{strategy for Alice in the } i\text{-th round}} \qquad \overbrace{\sigma_B^i : \underbrace{\Sigma^*}_{\substack{\text{Bob's}\\ \text{local}\\ \text{string}}} \times \underbrace{Q_A^{i-1}}_{\substack{\text{history of}\\ \text{signals}\\ \text{from Alice}}} \to \underbrace{Q_B \times \mathbb{D}^d}_{\substack{\text{new}\\ \text{message}}}}^{\text{strategy for Bob in the } i\text{-th round}}$$

7. *an* output function *of type*

$$(Q_A \times Q_B)^k \to (\mathbb{D}^{2dk} \xrightarrow[term]{} \mathbb{D}).$$

The protocol is executed on a pair of strings (w_1, w_2). In each round, each of the two parties sends a message (which consists of a signal and some elements of the output domain) that is based on their local string and the history of signals coming from the other party. After all k rounds have been executed, the joint signal history of both players is used, by the output function, to determine a term operation. This operation is then applied to the output history, yielding the final result of the protocol. As in the Boolean case, we are interested in protocols that are *split invariant*, which means that for every string $w \in \Sigma^*$, the same output is produced for every possible decomposition $w = w_1 w_2$. Such protocols compute a function of type $\Sigma^* \to \mathbb{D}$.

Example 9. [Boolean domain] Consider the Boolean output domain. In this case, the distinction between output values and signals is irrelevant, since the output values can be sent as signals. To produce the output, one of the parties sends it in a message, and the output function is a single variable x_i that copies it, as in Example 7. Therefore, the general protocol coincides with the Boolean protocol from the previous section, and can only compute regular languages. The same remarks apply for a general but finite output domain – a function $f : \Sigma^* \to \mathbb{D}$ can be computed by a protocol if and only if for every $d \in \mathbb{D}$, the inverse image $f^{-1}(d)$ is a regular language. □

Other examples of output domains are fields and strings. These were mentioned in the introduction in Sections 1.2 and 1.3, and will be discussed at length in Sections 4 and 5.

We have little say to say about protocols in the case of a general output domain. The only result that we have at this level of generality is a reduction to one-round protocols, similarly to the Boolean case.

Lemma 3.4. *Every protocol is equivalent to a one-round protocol.*

Proof. The proof of Lemma 2.2 generalises to arbitrary output domains with minor modifications. Here we need to take care of elements of the output domain that are sent in messages. This kind of dependecy similarly captured by functions of type:

$$(Q_B)^k \to \mathbb{D}^{dk}$$

for Bob, and

$$(Q_A)^k \to \mathbb{D}^{dk}$$

for Alice. These functions can be included in the first message that each party sends, and therefore the same construction as in the proof of Lemma 2.2 applies. □

Slightly ahead of time, we mention that the reduction to one-round protocols will no longer be valid for infinite input alphabets, which will be studied in Section 6. As we will see, in that case the interactive nature of the protocols will be essential.

4 Field outputs

In this section, we discuss functions where the output domain is a field, equipped with addition and multiplication. We prove that protocols have exactly the same expressive power as weighted automata. We begin by recalling the notion of weighted automata.

Weighted automata. A weighted automaton is a device that is used to compute a function from strings to a field (more generally, a semiring, but we consider the case of fields here). This model was originally introduced by Schützenberger [Sch61]. A weighted automaton is defined like a nondeterministic automaton, except that the transitions have weights in the field, and instead of choosing subsets of initial and final states, we also have assignments of weights. A weighted automaton makes sense not only for fields, but also for semirings, so we define it that way.

Definition 4.1 (Weighted automaton)**.** *A* weighted automaton *over a semiring $\mathbb{D}$ consists of a finite input alphabet Σ, a finite set of states Q, and functions:*

$$\underbrace{I : Q \to \mathbb{D}}_{\textit{initial}} \quad \underbrace{F : Q \to \mathbb{D}}_{\textit{final}} \quad \underbrace{\Delta : Q \times \Sigma \times Q \to \mathbb{D}}_{\textit{transitions}}. \tag{4}$$

A run of this automaton is defined in the usual way: it is a sequence of transitions, one for each input letter, such that consecutive transitions agree on the connecting states. The weight of a run is the product of: (1) the initial weight of its source state; (2) the weights used by its transitions; and (3) the final weight of its target state. If the semiring is non-commutative, the order of multiplication is important, and transitions are multiplied in the order corresponding to the input string. For an input string, the output of the automaton is the sum of weights of all runs over this automaton (sum is always commutative).

The main result of this section is that our protocol is equivalent to weighted automata, as stated in the following theorem from the introduction, which we now recall:

Theorem 1.2. *Assume that the domain $\mathbb{D}$ is a field. Then a function*

$$f : \Sigma^* \to \mathbb{D}$$

is computed by a protocol if and only if it is computed by a weighted automaton over the same field.

As we will see in Example 11 below, the theorem is false for general semirings that are not fields. It is possible that the theorem extends to commutative semirings; there is some evidence supporting this conjecture (for example, the supports of the relevant functions appear to be regular), but it remains unproven. The problem with such generalisations is that our proof uses the Fliess Theorem, which is only known for fields. Before proving the theorem in Section 4.1, we return to the issue of division, which was already discussed in Example 3.

Example 10. [Division, continued] Because it is undefined for zero, division is not a total operation, and therefore technically speaking it does not fall into our framework. We could, however try to incorporate it, by making the two parties responsible for avoiding division by zero. Under this framework, we could use a protocol to compute the function $1/|w|$ (a better choice would be $1/(|w|+1)$, since it would avoid problems with the empty string). As we have discussed in Example 3, such a function cannot be computed by a protocol that uses only addition and multiplication. We do not know what functions can be computed if division is also allowed. □

Example 11. [Semiring outputs] In this example we show that for semirings which are not fields, the protocol need not be equivalent to weighted automata. The implication

$$\text{protocol} \quad \Longleftarrow \quad \text{weighted automaton}$$

in Theorem 1.2, as we will see in a moment, holds for any semiring, and therefore the problematic implication is the other one. Here is an example where it fails. Let $\mathbb{D}$ be the free (non-commutative) idempotent semiring generated by two letters a and b. Elements of this semiring are finite sets of words in $\{a, b\}^*$, such as

$$\{3ab, 5ba, 7aab\}$$

The addition operation is multiset union, and the multiplication operation is concatenation of words, extended to sets in the natural way, as illustrated on this example:

$$\{a, b\} \cdot \{a, b\} = \{aa, ab, ba, bb\}.$$

Weighted automata over this semiring are the same as the rational relations [Eil74, Chapter IX]. On the other hand, a protocol can define string-to-$\mathbb{D}$ functions that are not rational. This is witnessed already by functions that produce singleton sets (call these singleton functions), which can be seen as functions of type $\Sigma^* \to \{a, b\}^*$. For example, consider the singleton version of the reverse function, i.e.

$$w \mapsto \{\text{reverse of } w\} \in \mathbb{D}.$$

This function can be computed by a protocol, using the same idea as in Example 4. This function, however, is not a rational relation, and therefore it is not computed by a weighted automaton over $\mathbb{D}$.

For protocols over this semiring where the outputs are always singletons, we can connect with the results on string-to-string functions that will be discussed in Section 5. In this case, all messages sent during the protocol must be singletons (this is because once a non-singleton is produced, it can never be turned into a singleton). Therefore, the operation $+$ can never be used in a non-trivial way, and thus the protocol can only use multiplication. This means that it coincides with the protocols with outputs that are strings with concatenation, as discussed in Section 5. According to Conjecture 1.3, the singleton functions are therefore exactly the regular functions. It is worth mentioning that, in view of Theorem 1.2, two-way weighted automata over fields do not have greater expressive power than one-way weighted automata: every two-way weighted automaton over a field can be simulated by a protocol over the same field, and again by Theorem 1.2 that protocol can be simulated by a one-way weighted automaton. The example above shows that this equivalence can fail for general semirings. But similar to one direction of the proof of Theorem 1.2, protocols over a general semiring can simulate two-way weighted automata over the same semiring, but the converse is not clear for us. We conjecture that, for general semirings, protocols correspond to two-way weighted automata; however, we currently have no evidence for.

□

Example 12. [Equality tests] In this example, we discuss an extension of the protocol which allows for equality tests, similarly to the algebraic group model [FKL18]. Clearly, equality tests cannot be completely unrestricted. Otherwise, in the presence of a countable output domain (which is the case for all protocols studied in this paper), the receiver could compare the message with all possible values one by one, until the correct one would be identified. This would invalidate the black box discipline. A reasonable restriction is to allow a constant number of equality tests for each message; this constant can also be brought down to one, by possibly sending more copies of the same message.

The resulting protocol would be able of complementing a weighted automaton $\mathcal{A}$, in the following sense:

$$w \in \Sigma^* \quad \mapsto \quad \begin{cases} 1 & \text{if } \mathcal{A}(w) = 0 \\ 0 & \text{otherwise.} \end{cases}$$

This form of complementation is undesirable from the point of view of decidability. For example, language equivalence is undecidable for weighted automata that are complemented in this way [Boj25a, Theorem 4.9]. Since we strive for protocols that describe "regular" functions, and such functions should be decidable, we avoid equality tests. □

Example 13. [Wrong output domains] This discussion of equality tests from Example 12 also explains why we should not expect results about regularity that work for any output domain. For example, if we would extend the field domain with a unary complementation operation

$$x \quad \mapsto \quad \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise,} \end{cases}$$

then our protocols could recover the undecidable model discussed in the previous paragraph. Of course, one can come up with even more obviously wrong output domains, such as a domain that consists of Turing machines with certain evaluation operations. We do not know where the dividing line is between "right" and "wrong" output domains. □

4.1 Proof of Theorem 1.2

We now return to the proof of Theorem 1.2. The right-to-left implication says that every weighted automaton can be simulated by a protocol. This is proved essentially in the same way as in the Boolean case, and the proof is valid for all semirings and not just fields. Suppose that the function is computed by a weighted automaton, which has state space Q. To every input string $w \in \Sigma^*$, we can associate a $Q \times Q$ matrix over the semiring. This matrix is defined by

$$M[p, q] = \sum_{\rho} \text{product of weights of transitions used by } \rho,$$

where ρ ranges over runs of the weighte automaton that start in state p, read the input string w, and end in state q. This map is a homomorphism from strings to matrices, i.e. the matrix corresponding to a string $w_1 w_2$ is obtained by multiplying the matrices corresponding to w_1 and w_2. In the protocol, Alice sends the matrix which corresponds to her local string, and Bob sends the matrix which corresponds to his local string. These matrices are multiplied using the semiring operations, and then multiplied with vectors that represent the initial and final weights of states. This protocol has one round and is signal-free, i.e. no information is conveyed using signals.

The rest of this proof is devoted to the left-to-right implication, i.e. showing that every function computed by a protocol is computed by a weighted automaton. As in the Boolean case, we will do a sequence of reductions, such that the protocol becomes more and more restrictive. In particular, we will show that the protocol can be reduced to a version that has one-round and is signal-free.

4.1.1 Reduction to a scalar product protocol

In the first step, we show that each protocol can be constrained to have a special form, which has one round and is signal-free. This protocol uses only the scalar product, as explained in the following definition.

Definition 4.2 (Scalar product protocol)**.** *Assume that the output domain is a field. A* scalar product protocol *is defined as follows. First, each of the two parties uses their local string to produce a vector of field elements, of some fixed dimension d, as expressed by two functions:*

$$\sigma_A, \sigma_B : \Sigma^* \to \mathbb{D}^d.$$

Next, the output is defined to be the scalar product of the two vectors.

This protocol has the same power as general protocols. Furthermore, the proof works not only for fields, but also for the more general case of commutative semirings (these are semirings where both addition and muliplication are commutative, in contrast to general semirings, where only addition is assumed to be commutative). The assumption that the output domain is a field, and not just a commutative semiring, will be used at a later stage, when we apply the Fliess Theorem.

Lemma 4.3. *Assume that the output domain is a commutative semiring. If a function is computed by a protocol, then it is computed by a scalar product protocol.*

Proof. The proof is a sequence of reductions, where more and more conditions are imposed on the protocol.

Step 1. One-round protocol. The first step is to reduce the protocol to a one-round protocol. This is done using Lemma 3.4.

Step 2. Signal-free protocol. We say that a protocol is *signal-free* if both of the sets Q_A and Q_B have one element each. In other words, the signals do not convey any information, and the only messaging activity consists of sending elements of the output domain. In a signal-free protocol, the concept of rounds is irrelevant, since the behaviour of one party is not influenced by the communication from the other party. The following claim shows that every protocol can be made signal-free, even if the semiring is not necessarily commutative.

Claim 4.4. *Assume that the output domain is a semiring, not necessarily commutative. Then every one-round protocol is equivalent to a signal-free protocol.*

Proof. Consider a one-round protocol. Without loss of generality, we assume that both signal spaces Q_A and Q_B are the same space Q. (We can always use the union of two signal spaces for both parties.) Assume that each of the parties sends d semiring elements in the protocol. In other words, the protocol works as follows:

1. Based on her local string, Alice chooses a message $(q_A, \bar{x}) \in Q \times \mathbb{D}^d$;
2. Based on his local string, Bob chooses a message $(q_B, \bar{y}) \in Q \times \mathbb{D}^d$;
3. Based on the signals q_A and q_B, a term operation with $2d$ variables is chosen, call it t_{q_A,q_B}, and the output is obtained by applying this term operation to $(\bar{x}, \bar{y})$.

To prove the claim, we need to show that the protocol can be adapted so that always the same term operation is chosen, i.e. there is no dependence of this term operation on the signals q_A and q_B. This way the signals can be eliminated. To do this, we increase the protocol dimension from d to $d + |Q|$.

This means that for each possible signal $q \in Q$, each party sends a semiring element corresponding to this signal. This element will be either 0 or 1, which are elements that need to exist in every semiring (see [DKV09, p.7] for a formal definition of semirings). The idea is that instead of sending a signal $q \in Q$, each party will set the corresponding semiring element to 1, and the remaining semiring elements to 0. The corresponding term operation is then

$$\sum_{\substack{q_A \in Q \\ q_B \in Q}} \overbrace{x_{q_A} \cdot y_{q_B}}^{\substack{\text{variables corresponding} \\ \text{to the messages } q_A \text{ and } q_B,}} \cdot t_{q_A, q_B}(\bar{x}, \bar{y}).$$

When evaluating this term operation, the summands that do not correspond to the intended message (q_A, q_B) will be eliminated, since they will contain a variable that is set to 0. Only the summand corresponding to the intended message will be used, and thus the correct output will be produced. □

Step 3. Scalar product. In the previous step, we have reduced the protocol to a special case, where Alice and Bob send vectors, call them $\bar{x}, \bar{y} \in \mathbb{D}^d$, and then some fixed term operation t with $2d$ variables is applied to them. To complete the proof of the lemma, we show that the term operation can be turned into a scalar product. This term operation is a sum of monomials, with each monomial being a product of some variables. This part of the proof will work for commutative semirings (the previous step worked for all semirings).

Consider the monomials in the term operation t. Since muliplication is assumed to be commutative, for each monomial, its contribution to the output is obtained by multiplying two numbers: (a) the product of the variables in the term operation that are contributed by Alice; and (b) the product of the variables in the term operation that are contributed by Bob. We can redesign the protocol so that for each monomial, Alice sends the contribution (a), and Bob sends the contribution (b). In the new protocol, the dimension is the number of monomials from the original protocol, and the term operation is a scalar product. □

4.1.2 From a scalar product protocol to a weighted automaton

In this section, we complete the proof of Theorem 1.2, by showing that scalar product protocols can be simulated by weighted automata. Similarly to the Boolean case, the proof uses a Myhill-Nerode characterization. In the case of weighted automata, this characterization is called the Fliess Theorem, which characterizes functions computed by weighted automata in terms of a certain infinite matrix. It is here where we use the assumption that the output domain is a field, and not just a commutative semiring.

Definition 4.5 (Hankel Matrix)**.** *Let $\mathbb{D}$ be a field. The* Hankel matrix *of a function*

$$f : \Sigma^* \to \mathbb{D}$$

is the matrix where rows are words in Σ^*, columns are words in Σ^*, and the entry corresponding to a row u and a column v is $f(uv)$.

Another perspective on the Hankel matrix is that it describes the *derivatives* of the function f. Each row in the Hankel matrix can be seen as a function of type $\Sigma^* \to \mathbb{D}$, which inputs columns (i.e. strings) and outputs the corresponding entries in the Hankel matrix. If the row corresponds to a word w, then this function is

$$v \mapsto f(wv),$$

which is called the *left derivative* of f with respect to w. Similarly, the columns of the Hankel matrix describe *right derivatives* of f.

The Fliess Theorem [Fli74, Theorem 2.1.1] states that a function

$$f : \Sigma^* \to \mathbb{D}$$

is computed by a weighted automaton if and only if its Hankel matrix has finite rank, i.e. its rows (i.e. the left derivatives) are spanned by a finite subset. (This is equivalent to saying that the columns, or right derivatives, have a finite spanning subset.) Therefore, to complete the proof of Theorem 1.2, it is enough to show the following lemma.

Lemma 4.6. *If a function is computed by a scalar product protocol, then its Hankel matrix has finite rank.*

Proof. Essentially by definition, the Hankel matrix of a function computed by a scalar product protocol with dimension d can be obtained as a sub-matrix of the following matrix: rows and columns are vectors in $\mathbb{D}^d$, and the entries are obtained by taking scalar products. This matrix is easily seen to have finite rank, namely d, since the scalar product becomes a linear operation once one of the two arguments is fixed. □

5 String outputs

In this section, we consider the case where the output domain is strings over some finite alphabet. We use the name *string-to-string function* for any function of type $\Sigma^* \to \Gamma^*$, where both alphabets Σ and Γ are finite. For such functions, the protocols are assumed to use the output domain of strings Γ^* equipped with concatenation. In the case of string-to-string functions, we conjectured, see Conjecture 1.3, that protocols define exactly the so-called regular functions, which will be formally defined in Section 5.1 below.

The content of this section is an extended discussion of this conjecture. In Section 5.1 we formally define the regular functions and prove the implication

$$\text{protocol} \impliedby \text{regular}.$$

The content of the conjecture is therefore the implication

$$\text{protocol} \implies \text{regular}.$$

In Section 5.2, we present some evidence for this implication. We show that string-to-string functions computed by protocols share many good properties of the regular functions, such as linear output size and computability. In Section 5.3, we present further evidence for the implication, namely we prove it in the special case where the output alphabet has only one letter (the remaining case is two output letters, since more letters do not change the situation).

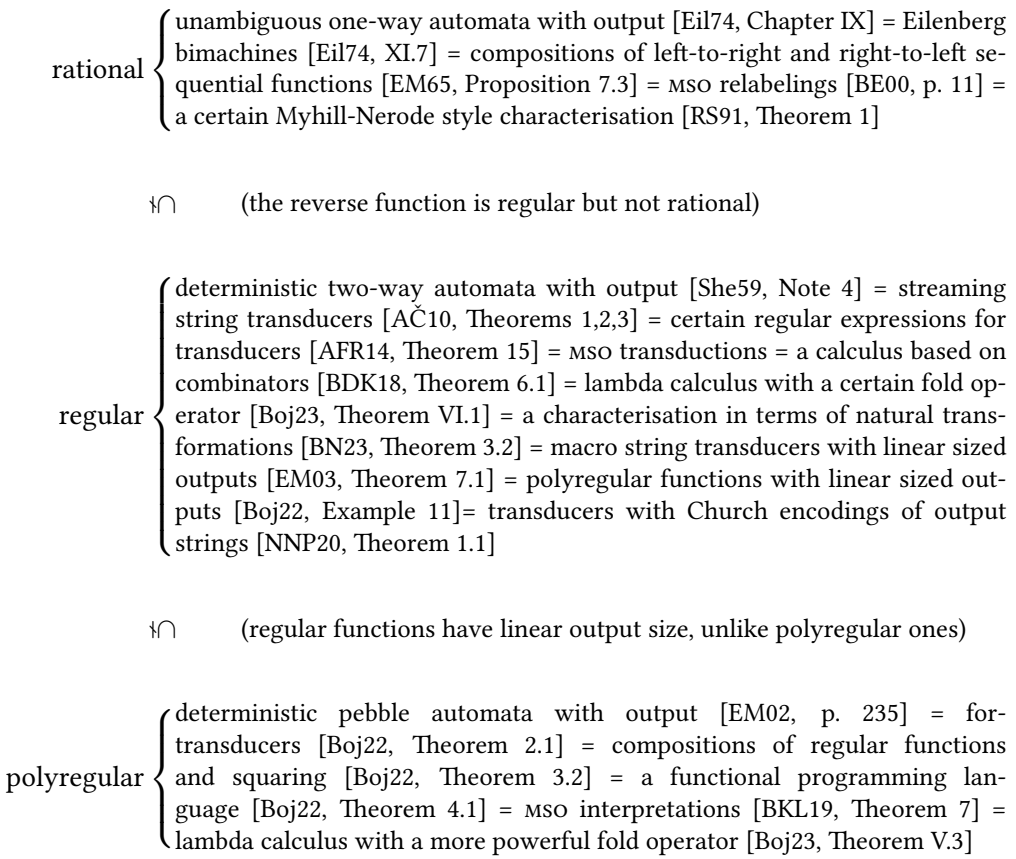


Figure 1: Three important classes of string-to-string functions.

5.1 Regular string-to-string functions

In this section, we define the class of regular string-to-string functions, and we prove the implication $\Longleftarrow$ in the conjecture. Historically, class of regular string-to-string functions was first defined in terms of deterministic two-way automata with output [She59, Note 4]. Although numerous equivalent definitions appeared later on, we will use the original definition.

Definition 5.1 (Two-way automaton)**.** *A* deterministic two-way automaton with output *is given by the following ingredients:*

1. *a finite input alphabet* Σ*;*
2. *a finite output alphabet* Γ*;*
3. *a finite set of states* Q*, with an initial state* $q_0 \in Q$*;*
4. *a transition function*

$$\delta : \underbrace{Q}_{\substack{\text{old}\\ \text{state}}} \times \underbrace{(\Sigma + \{\vdash, \dashv\})}_{\substack{\text{input letter}\\ \text{under the head}}} \to \{\mathit{halt}\} + (\underbrace{Q}_{\substack{\text{new}\\ \text{state}}} \times \underbrace{\{-1, 0, 1\}}_{\substack{\text{head}\\ \text{movement}}} \times \underbrace{\Gamma^*}_{\substack{\text{added}\\ \text{output}}}).$$

The automaton works as follows. The input string w is placed on a tape, with the left end marked by $\vdash$ and the right end marked by $\dashv$. The automaton starts in state q_0, with its head on the left end of the tape, which contains the marker $\vdash$. In each step, the automaton looks at its current state and the letter under its head, and based on this information, it uses the transition function to decide if it halts, or it continues its computation. In case it continues, it chooses a new state, the direction in which it moves its head, and a string over the output alphabet Γ, which is appended to the output tape. We assume that the automaton is always halting, which means that for every input string, the computation eventually halts. In particular, the computation must be well-defined, which means that the head never falls off the input by moving outside the endmarkers. The semantics of such an automaton is of type $\Sigma^* \to \Gamma^*$. (For automata which are not necessarily halting, the function would be partial, since it would be undefined for inputs where the automaton does not halt.)

Example 14. [Reverse] For each input alphabet Σ, the reverse function of type $\Sigma^* \to \Sigma^*$ is computed by a two-way automaton, which first moves its head to the end of the string, and then starts copying it to the output while moving in the left direction. $\square$

The class of functions computed by two-way automata has a remarkable number of equivalent descriptions, originating in different fields, including: monadic second-order transductions [EH01, Section 4], streaming string transducers [AČ10, Section 3], certain kinds of regular expressions [AFR14, Section 2], a calculus of functions based on combinators [BDK18, Theorem 6.1], a characterisation based on natural transformations [BN23, Theorem 3.2]. For this reason, some authors (starting with Engelfriet and Hoogeboom), use the name *regular* for this class of function, with the intended meaning being that these functions are the functional analogue of regular languages. Although this thesis can be questioned, see [Boj22], for the purposes of this paper we adopt the terminology of Engelfriet and Hoogeboom [EH01, p. 217], as expressed in the following definition.

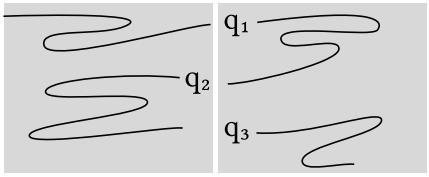
Definition 5.2 (Regular string-to-string function)**.** *A string-to-string function is called* function *if it is computed by a deterministic two-way automaton with output.*

One good property of the regular string-to-string functions is that they are closed under composition [CJ77, Theorem 2]. In particular, Conjecture 1.3 would imply that the same is true for functions computed by protocols. Without proving the conjecture, we do not see any direct way of proving composition for protocols. The regular string-to-string functions have other good properties, such as decidable equivalence [Gur82, Theorem 1], but we do not discuss these in more detail, since they seem to have little direct bearing on protocols.

The following lemma shows one of the implications in the conjecture.

Lemma 5.3. *If a string-to-string function is function, then it is computed by a protocol.*

Proof. The two parties can simulate a two-way automaton with output. The execution of the protocol describes the crossing sequence of the automaton, i.e. how it crosses the boundary between the two local strings of Alice and Bob. Here is a picture:



More formally, the crossing sequence is defined as follows, given a split of the input string into two parts $w_1 w_2$. We run the automaton until the first configuration which is in the word w_2. Then we run it until the first configuration which is in the word w_1. We continue this way, with odd-numbered steps describing runs inside w_1 that end in configurations from w_2, and even-numbered steps describing runs inside w_2 that end in a configuration from w_1. The last step is exceptional, since it ends with an accepting configuration. The number of steps in a crossing sequence is bounded by twice the number of states, since otherwise the automaton would enter an infinite loop. This bound is the number of rounds in the protocol. In each round, the state corresponding to this round is sent as a signal, and the output value in the message is the part of the output string that is produced in this step. At the end of the protocol, the pieces of the output string are concatenated. □

In view of the above lemma, the content of the conjecture is the opposite implication, namely that every protocol computes are regular function. The rest of Section 5 is devoted to evidence for the opposite implication.

5.2 Evidence for the conjecture

In this subsection, we show that the string-to-string functions computed by protocols share some good properties of regular functions, such as linear size outputs and computability. Even computability is not a priori obvious, due to the non-uniformity of the protocols. These results can be seen as evidence of the open implication

$$\text{protocol} \implies \text{regular}$$

in the conjecture. To prove these results, we will leverage the results on weighted automata from Section 4. The point of departure is the following lemma, which connects weighted automata and string-to-string functions that can be computed in our protocol.

Lemma 5.4. *Let $\mathbb{D}$ be a semiring, and consider two functions*

$$\Sigma^* \xrightarrow{f} \Gamma^* \xrightarrow{g} \mathbb{D},$$

such that f is computed by a protocol (with string outputs) and g is computed by a weighted automaton, then the composition $f;g$ is computed by a protocol (with outputs in $\mathbb{D}$).

Proof. For each string in Γ^*, the weighted automaton for g has an associated matrix over the semiring $\mathbb{D}$, as explained in the proof at the beginning of Section 4.1. Similarly to that proof, we can modify the protocol for f to a protocol for $f;g$ where the parties send matrices instead of strings. □

Corollary 5.5. *If the domain $\mathbb{D}$ is a field, then the conclusion of Lemma 5.4 can be strengthened to say that $f;g$ is computed by a weighted automaton.*

Proof. For field outputs, protocols are equivalent to weighted automata, thanks to Theorem 1.2. □

The above corollary establishes a property of f, namely that weighted automata (over a field) are closed under precomposition with f. We think that this is an important property, and therefore we give it a name.

Definition 5.6 (Field continuity)**.** *A string-to-string function $f : \Sigma^* \to \Gamma^*$ is called* field continuous *if functions computed by weighted automata over a field are closed under precomposition with f.*

The name "continuous" is inspired by a similar terminology that is used in automata theory for functions that preserve regularity under inverse images, see [PS05, Theorem 4.1] or [CCP20, Footnote 2]. For the latter notion, we use the name *Boolean continuity*.

Definition 5.7 (Boolean continuity)**.** *A string-to-string function $f : \Sigma^* \to \Gamma^*$ is called* Boolean continuous *if preimages of regular languages are regular.*

As we have shown in Corollary 5.5, all string-to-string functions computed by protocols are field continuous. In particular, since every regular string-to-string function is computed by a protocol, it follows that every regular string-to-string function is field continuous[3]. We conjecture that the converse is also true.

Conjecture 5.8. *A string-to-string function is field continuous if and only if it is regular.*

In Example 17 later in this section, we show that the conjecture becomes false if the left side is relaxed from field continuous to Boolean continuous.

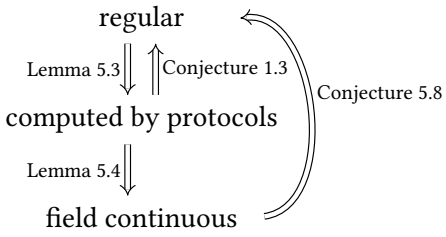
Conjecture 5.8 can be seen as a machine independent characterisation of the regular string-to-string functions. This would be a very valuable contribution. Almost all known characterisations of the regular string-to-string functions have somewhat lengthy definitions, based on specific computational models, and it is something of a miracle that all of these models are equivalent. A possible exception is the characterisation in [BN23], which does not use a machine model; however that characterisation uses the abstract language of category theory, and is less elementary than the one in Conjecture 5.8.

As in Conjecture 1.3, the content of Conjecture 5.8 is the left-to-right implication

$$\text{field continuous} \implies \text{regular}.$$

[3]To the best of our knowledge, this is a new result. It can also be proved directly, without passing through protocols.

Conjecture 5.8 is stronger than Conjecture 1.3, as explained in the following diagram, which shows the known relations between three kinds of string-to-string functions:



The following theorem gives some evidence for the stronger conjecture, and therefore also the weaker one, by showing that the field continuous functions share some well-known properties of the regular string-to-string functions.

Theorem 1.4. *If a string-to-string function is computed by a protocol, then:*

1. *outputs have at most linear size;*
2. *outputs can be computed in linear time (ignoring logarithmic factors);*
3. *preimages of regular languages are regular.*

Proof. For properties 1 and 2, we embed strings into numbers. An output string over alphabet Γ can be seen as a number in base $|\Gamma|$. To avoid the ambiguity that could result from leading zeros, we first prepend the string with the digit 1. Let

$$g : \Gamma^* \to \mathbb{N} \subseteq \mathbb{Q}$$

be the corresponding encoding. This encoding can be computed by a weighted automaton over the field $\mathbb{Q}$, see [Boj25a, Lemma 8.10]. By the assumption on field continuity, the composition $f; g$ can be computed by a weighted automaton. This is a weighted automaton that works in the field of rationals $\mathbb{Q}$, but only produces natural numbers on its output. By [BR08, p. 110] the automaton can be chosen so that it only uses integers $\mathbb{Z}$, possibly including negative integers. Summing up, we have a weighted automaton over $\mathbb{Z}$ that outputs the representation, in base $|\Gamma|$, of the output string produced by g. We claim that for such an automaton, the output number

1. has a linear number of digits;
2. can be computed in linear time.

These two claims yield the corresponding items in the statement of the theorem. The first item, about a linear number of digits, is true because it is true for every weighted automaton over $\mathbb{Z}$. This is because applying a fixed linear map can only add a constant number of digits, and therefore each input letter can increase the output by a constant number of digits. For the second item, use the fact that a weighted automaton can be evaluated in linear time. We did not find this result in the literature, so we give a proof sketch below.

Claim 5.9. *Fix a function $f : \Sigma^* \to \mathbb{Z}$ computed by a weighted automaton over $\mathbb{Z}$. Then f can be computed in linear time (ignoring logarithmic factors).*

Proof. We use the RAM model of computation. In this model, numbers with n digits can be multiplied in time linear in n (ignoring logarithmic factors), using Fast Fourier Transform [SS71, p.291]. If we would try to evaluate the weighted automaton naively, from left to right, we would be doing a linear number of multiplications, involving numbers with a linear number of bits. This would give a quadratic time algorithm. To get linear time, we use divide and conquer approach which was suggested to us by Marek Sokołowski, and which is inspired by similar algorithms for multipoint evaluation of polynomials [VZGG03, Section 10.1].

For each input letter, the weighted automaton has an associated matrix. Therefore, the essence of the problem is to compute the multiplication of n matrices $A_1 \cdots A_n$, which are taken from some fixed finite set of square matrices of same dimension. As we have already remarked before, if we multiply n matrices, then each entry in the resulting matrix has a $\mathcal{O}(n)$ bits. Instead of multiplying the matrices from left to right, we organise the multiplication in a balanced binary tree, and evaluate this tree bottom-up. At the bottom of this tree, we have $n/2$ multiplications of matrices from the finite set. More generally, in each subtree of height h, the resulting matrix has entries with $\mathcal{O}(2^h)$ bits, and there are $n/2^h$ such subtrees. Therefore, assuming that we have evaluted the value of each subtree of height $h-1$, then we can evaluate each subtree of height h, by using $\mathcal{O}(n/2^h)$ instances of matrix multiplication, with the ocrresponding matrices having entries that have $\mathcal{O}(2^h)$ bits. By using algorithms for addition and multiplication that are linear (ignoring logarithmic factors), the passage from height $h-1$ to height h can be done in linear time (ignoring logarithmic factors). Since the number of heights is logarithmic, we get the desired result. □

We are left with property 3, about Boolean continuity. This will follow from the special case of field continuity, where the field is the two-element field. This is because of the following folklore correspondence between regular languages and weighted automata over the two-element field.

Claim 5.10. *A language $L \subseteq \Gamma^*$ is regular iff its characteristic function $\Gamma^* \to \{0,1\}$ is computed by a weighted automaton over the two-element field*

Proof. For the left-to-right implication, we observe that a weighted automaton over a finite field can be simulated by a deterministic finite automaton. For the other direction, we observe that a weighted automaton can count the parity of the number of runs in a finite automaton, and if the automaton is deterministic then the number of runs is either zero or one, and thus the parity gives the right answer. □

In terms of the correspondence from the above claim, preimages of regular languages become precompositions of weighted automata over the two-element field. In particular, regularity is preserved. □

One could think that already the three properties in the above theorem are not only necessary for regularity, but also sufficient. This is not the case, as shown by the following example.

Example 15. [Factorials] Consider a string-to-string function

$$g : \Sigma^* \to \{a\}^*$$

where both the input and output alphabets are unary. A sufficient condition for Boolean continuity of such functions is given in [BN23, Example 2.12], using *factorials*, i.e. numbers in the set $\{n! \mid n \in \mathbb{N}\}$. This sufficient condition is that: (a) every output string arises from finitely many inputs; and (b) every output string has length that is a factorial.

It is not hard to come up with a non-regular function that has properties (a) and (b), thus ensuring Boolean continuity, and which has furthermore linear size outputs and is computable in linear time. For example, the function could map an input string w to the longest string of factorial length that is shorter than w. □

5.3 Unary output alphabet

In this section, we provide further evidence for Conjectures 1.3 and 5.8, by showing that they are true for output alphabets with only one letter. To prove the conjectures, we will pass through rational functions, which are a fragment of the regular functions. When the output alphabet has only one letter, rational functions and regular functions coincide, which means that the results on rational functions are relevant in this case.

5.3.1 Rational functions and monotone protocols

In this section, we use a variant of the protocols to characterise exactly the rational string-to-string functions. This will be done by applying a characterisation of the rational functions due to Schützenberger and Reutenauer [RS91], which happens to be ideally matched to our protocol model, subject to a monotonicity restriction. Let us begin by defining the relevant notions.

Rational functions. There are several ways of defining the rational string-to-string functions. For the results in this section, the particular choice of definition will not be important, since we will refer to an external result about rational functions. Nevertheless, for the reader's convenience we include a definition, which is based on weighted automata over the semiring of finite sets of words.

Consider a nondeterministic automaton with an input alphabet Σ, together with an output map

$$\text{output} : \underbrace{(I + F + \Delta)}_{\text{disjoint union of initial states, final states, and transitions}} \to \Gamma^*.$$

Using the output map, we can associate to each accepting run of the automaton an output string, which is obtained by concatenating the output strings associated with the initial state, the transitions used in the run, and the final state. This way, the automaton defines a relation between input and output strings, which maps an input string to the outputs of all possible accepting runs. Such a relation is called a *rational string-to-string relation*. A *rational string-to-string function* is the special case where each input string is mapped to exactly one output string.

Example 16. The identity function is rational, and so is the function which deletes all a's from the input string. Another example is the function which swaps the first and last letters, and leaves the rest of the input string unchanged. String reversal is not rational; also duplication is not rational. □

Monotone protocols. A protocol with string-to-string outputs is called *monotone* if the output string is produced by concatenating two strings, the left one produced by Alice, and the right one produced by Bob. One can consider monotone protocols with one or more rounds. The reduction to one-round protocols from Lemma 3.4 also works in the presence of the monotone restriction, and

therefore we will only talk about one-round monotone protocols. Such a protocol can be described more elementarily as follows. The two players have strategies

$$\sigma_A : \Sigma^* \to Q_A \times (\Gamma^*)^d,$$
$$\sigma_B : \Sigma^* \to Q_B \times (\Gamma^*)^d,$$

in which they send a signal as well as a tuple of d strings over the output alphabet. Based on the signals received from both players, the output is produced by concatenating two of the output strings, one from Alice and one from Bob. This is formalised by an output function of type

$$\text{output} : Q_A \times Q_B \to \{1, \ldots, d\}.$$

The output of the protocol is then defined as follows: we apply the output function to the signals produced by both players, which gives us an index $i \in \{1, \ldots, d\}$, and then the output string is obtained by concatenating the i-th string from Alice with the i-th string from Bob (thus ensuring that Alice's part of the output always comes before Bob's).

Equivalence of the models. We now show that the monotone protcols compute exactly the rational string-to-string functions. As it turns out, this equivalence was essentially already proved by Schützenberger and Reutenauer in [RS91], albeit in a different language. The corresponding characterisation is stated below.

Theorem 5.11. *For a string-to-string function $f : \Sigma^* \to \Gamma^*$, the following are equivalent:*

1. *f is computed by a monotone protocol;*
2. *there is a finite family of partial string-to-string functions, indexed by a finite set I*

   $$\{\alpha_i, \beta_i : \Sigma^* \to \Gamma^*\}_{i \in I}$$

   such that the (total) function

   $$(w_1, w_2) \quad \mapsto \quad f(w_1 w_2)$$

   is equal to the union of partial functions

   $$\bigcup_{i \in I} \Big((w_1, w_2) \mapsto \underbrace{\alpha_i(w_1) \cdot \beta_i(w_2)}_{\substack{\text{the concatenation is defined} \\ \text{only if both functions are defined}}} \Big).$$

3. *f is a rational string-to-string function.*

Proof. The equivalence of items 2 and 3, which is the essence of the theorem, was proved by Schützenberger and Reutenauer in [RS91, p. 674]. It remains to show the equivalence of items 1 and 2, which is a simple unfolding of the definitions.

- 2 $\implies$ 1. Suppose that f satisfies item 2. As her signal, Alice says which of the functions α_i are defined on her part of the input, and sends all the corresponding outputs. Bob does the same on his side. Based on the signals, they can determine an index i such that both α_i and β_i are defined, and then they can concatenate the corresponding outputs to produce the final output. This shows that item 1 holds.

- 1 $\implies$ 2. Suppose that f satisfies item 1. Based on the monotone protocol from the assumption, we define the indexing set I to be $Q_A \times Q_B$, i.e. all possible combinations of signals of Alice and Bob. For each such pair of signals $(q_A, q_B) \in Q_A \times Q_B$, we define two partial functions. The first one is defined on those strings w_1 for which Alice's signal is q_A, and in this case it produces the i-th string from Alice, where i is obtained by applying the output function to (q_A, q_B). The second one is defined similarly on Bob's side. It is easy to see that the conditions from item 2 are now satisfied.

□

Before continuing, let us remark that a similar approach could be taken to characterise the subsequential functions, which are the subclass of rational functions that corresponds to deterministic automata. In this case, the corresponding restriction on the protocols is monotonicity, plus the extra requirement is that the output function does not depend on the signal sent by Bob, i.e. the only signal information travels from left to right. Using a characterisation of subsequential functions from [Cho77], one can show that such protocols compute exactly the subsequential functions. The details are left to the reader.

5.3.2 Proofs of Conjectures 1.3 and 5.8 for a unary output alphabet

Using the above characterisation of rational functions, we can now prove Conjectures 1.3 and 5.8 for unary output alphabets. The first conjecture is an immediate corollary.

Theorem 5.12. *Conjecture 1.3 holds for string-to-string functions where the output alphabet has only one letter. In other words, for an output alphabet Γ with one letter, protocols compute exactly the regular string-to-string functions.*

Proof. When the output alphabet has only one letter, then: (a) rational string-to-string functions coincide with the regular string-to-string functions, which is a folklore result; and (b) monotone protocols coincide with general protocols, since there is no difference between concatenating two strings in different orders. Therefore, the equivalence from Theorem 5.11 implies the theorem. □

In the rest of this section, we use the above theorem to derive some further observations, and in particular to prove Conjecture 5.8 for unary output alphabets. The first observation is about protocols that use integers with addition and have non-negative outputs.

Theorem 5.13. *Consider a protocol with output domain $(\mathbb{Z}, +)$ where all outputs are non-negative. Then the same function can be computed by a protocol with output domain $(\mathbb{N}, +)$.*

Proof. Let us assume without loss of generality that f is computed by a one-round protocol. We use the name *configuration* for the message sent by alice, similarly to the proof of Theorem 5.12. The configuration consists of a signal, and several integers. We can improve the protocol so that: (a) for each input with a split $w = w_1 w_2$ exactly one of the numbers in Alice's configuration is part of the final output; and (b) if Alice's configuration for a string w_1 uses some integer, then there is some w_2 where this integer contributes to the final output. These assumptions can be ensured by a powerset construction on Alice's side.

Let us prove the following intermediate claim: the integers in Alice's configuration are uniformly bounded below. This is a simple corollary of the properties (a) and (b): if the integers would be

arbitrarily large negative numbers, then the contribution of Bob for some fixed string w_2 would not be enough to offset them.

Thanks to the above claim, we can further improve the protocol as follows. Alice sends her original configuration, with the following change: if some numbers were negative, then they are truncated to zero, and the signal is adjusted so that it stores the negative numbers. This can be done in a finite signal space, due to the claim on lower bounds. Once Bob receives the message, he can the produce the output of the original protocol, by possibly subtracting Alice's negative part encoded in the signal from his part of the output. □

This theorem allows us to derive a characterisation of functions computed by weighted automata over $\mathbb{N}$ that have linear growth.

Corollary 5.14. *Let $f : \Sigma^* \to \mathbb{Q}$ be a function computed by a weighted automaton over $\mathbb{Q}$, which has linear growth (i.e. the output number is at most linear in the input length), and such that all outputs are natural numbers. Then f is a regular function, when viewed as a string-to-string function with a unary output alphabet.*

Proof. It is well-known that weighted automata over $\mathbb{Q}$ computing integer values can be simulated by weighted automata over the semiring $\mathbb{Z}$: this is sometimes referred to as $\mathbb{Q}$ being a Fatou extension of $\mathbb{Z}$ [BR08, p. 110]. Therefore, we can assume without loss of generality that f is computed by a weighted automaton over $\mathbb{Z}$. Then, one can leverage a result from [CDTL23] stating that weighted automata over $\mathbb{Z}$ with linear growth can be computed as follows: first, a regular function $g : \Sigma^* \to \{a, b\}^*$ is computed, then each letter a is mapped to $+1$ and each letter b is mapped to -1, and finally all values are summed up. For a precise statement, see [CDTL23, Proposition II.13, Theorem III.3]. Now, it is clear from the above description that f can be computed by a protocol manipulating integer values, and combining them only using addition. By Theorem 5.13, f can also be computed by a protocol manipulating natural numbers. Finally, by Theorem 5.12, f is a regular function. □

From the above, we conclude that Conjecture 5.8 holds for unary output alphabets.

Corollary 5.15. *Conjecture 5.8 holds for string-to-string functions where the output alphabet has only one letter. In other words, a string-to-string function with unary output alphabet is field continuous if and only if it is regular.*

Proof. By Corollary 5.5, it suffices to show the only-if direction. Let $f : \Sigma^* \to \Gamma^*$ be a function with a unary output alphabet which is field continuous. Compose this function with the weighted automaton $\mathcal{A} : \Gamma^* \to \mathbb{Q}$ which computes the length of the output string. The result is a function that is subject to the assumptions of Corollary 5.14, and therefore it is a regular function. □

Let us conclude this section with an example showing that Corollary 5.14 cannot be extended to weighted automata over $\mathbb{Z}$ with arbitrary growth.

Example 17. [Quadratic counterexample] We show a function which: (a) is a linear combination of MSO counting functions of arity two, with negative coefficients; (b) has only non-negative outputs; and (c) cannot be presented as linear combination with positive coefficients. The idea, which is based on [BR08, Example 2.1], is to trivially ensure non-negativity by squaring. Take the function

$$w \in \{a, b\}^* \quad \mapsto \quad (\text{(number of } a\text{'s in } w) - (\text{number of } b\text{'s in } w))^2 .$$

This function clearly satisfies (a) and (b). As explained in [CDTL23, p.3], it also satisfies (c), since the inverse image of 0 is not a regular language, as would be the case if only positive coefficients were used. □

6 Infinite alphabets

In this section, we present a variant of our model which deals with an input alphabet. This direction is rooted in the tradition of language theory for infinite alphabets, which dates back to the work of Kaminski and Francez [KF94], and has been developed in many subsequent papers, see e.g. the survey [Boj17]. The general idea is that we have an infinite alphabet $\mathbb{A}$ (whose elements are called *atoms*), and the languages refer only to equality between letters, as in the following examples

$$\{w \in \mathbb{A}^* \mid \text{ the first letter is equal to the last letter } \} \tag{5}$$

$$\{w \in \mathbb{A}^* \mid \text{ some letter appears at least twice } \} \tag{6}$$

There are numerous models of automata for such languages, which typically involve some kind of finite memory, as well as registers that store letters from $\mathbb{A}$. For example, the language (5) is recognised by an automaton which loads the first letter into a register, and then toggles acceptance depending on comparison of the register with the current input letter. The language (6) is recognised by an automaton which nondeterministically guesses a position, puts its letter into a register, and then waits for this letter to appear again.

Numerous models for infinite alphabets have been proposed in the literature, see Figure 2 which contains a sample of seventeen models. Interestingly, all of those models are pairwise non-equivalent. This sharply contrasts with the finite-alphabet case, where virtually all models coincide and capture the regular languages.

In this section, we describe an infinite-alphabet version of our two-party protocols. The motivation for this study is twofold: (a) a search for a canonical model of regular languages for infinite alphabets; and (b) mathematical interest. Regarding the point (a), we hope that the adaptability of two-party protocols to various settings will help us find a canonical model of regular languages for infinite alphabets. This seems to be at least partially successful, since there is evidence — which we present in this section — that the protocols are equivalent to one of the existing automaton models[4], namely unambiguous register automata, see item 9 in Figure 2. If true, this equivalence would be unexpected, since there does not seem to be any syntactic connection between unambiguous register automata and protocols. Regarding point (b), one of the exciting features of our protocol model for infinite alphabets is that the interaction between the two parties becomes essential, and the protocol cannot be reduced to the one-round case as in Lemma 3.4.

6.1 Protocols for an infinite alphabet

We now give a more detailed description of our model. As explained in the introduction, we only care about languages that are closed under *permutations of the alphabet*, according to the following definition.

[4]Thus avoiding proliferation of standards, humanistically depicted in a famous XKCD comic.

1. deterministic register automata [KF94, Definition 3]
2. nondeterministic register automata [KF94, Definition 1]
3. nondeterministic register automata with guessing [Boj25b, Definition 2.7]
4. weighted register automata over the two-element field [BFKM24, Definition 3.1]
5. two-way deterministic register automata [KF94, Definition 5]
6. two-way nondeterministic register automata [NSV04, Definition 2.1]
7. alternating register automata [DL09, p. 16:8]
8. alternating register automata with one register [DL09, p. 16:19]
9. unambiguous register automata [Col15, Section 5]
10. register automata with pebbles [NSV04, Section 2.2]
11. single-use register automata [BS20, Definition 2]
12. data automata [BDM$^+$11, Section 4.2]
13. class automata [BL10, Section III]
14. regular expressions [KT04, Definition 2]
15. three other kinds of regular expressions [LTV15, Sections 4, 5, 6]
16. yet another kind of regular expressions [BS19, Section 5]
17. monadic second-order logic with equality [NSV04, Section 2.4]

Figure 2: A non-exhaustive list of models of automata for infinite alphabets. All models in the list are pairwise non-equivalent. In contrast, for finite alphabets, all models in this list are equivalent, and define exactly the regular languages.

Definition 6.1 (Equivariant language). *A language $L \subseteq \mathbb{A}^*$ is called* equivariant *if*

$$w \in L \quad \iff \quad \pi(w) \in L$$

holds for every permutation π of the alphabet $\mathbb{A}$.

Examples of equivariant languages include the languages in (5) and (6). On the other hand, the language "the first letter is a vowel" or "the letters are strictly increasing" are not equivariant, since there is no such thing as a vowel, or an ordering of the letters. The principle of equivariance will also be applied to protocols, as described below.

To define *(simple) equivariant protocols* for infinite alphabets, we use the same kind of protocols as in Definition 2.1, except that apart from bits, the parties can also send letters from the alphabet $\mathbb{A}$. It follows that the allowed set of messages is $\{\text{true}, \text{false}\} + \mathbb{A}$, i.e. the disjoint union of the Booleans and the input alphabet. Similarly, to Definition 2.1, the number of rounds is a fixed number k and in the i-th round each party chooses a new message according a strategy which is a function of the following type:

$$\underbrace{\mathbb{A}^*}_{\substack{\text{local}\\ \text{string}}} \times \underbrace{(\{\text{true}, \text{false}\} + \mathbb{A})^{i-1}}_{\substack{\text{messages received}\\ \text{in previous rounds}}} \to \underbrace{\{\text{true}, \text{false}\} + \mathbb{A}}_{\substack{\text{message sent}\\ \text{in this round}}}$$

In the last round, Bob must send a bit, and this bit is the output of the protocol. An important restriction in the protocol is that the strategies of both parties must be *equivariant* in the following sense: a strategy σ is equivariant if for every permutation π of $\mathbb{A}$ and for every i, it satisfies the following condition:

$$\sigma(w, m_1, \ldots, m_{i-1}) = m_i \quad \Rightarrow \quad \sigma(\pi(w), \pi(m_1), \ldots, \pi(m_{i-1})) = \pi(m_i).$$

In the above, π is applied to messages in the natural way – it modifies the atoms and leaves the Booleans unchanged.

Example 18. [Repetitions cannot be detected] Having formally defined the protocols for infinite alphabets, we can now revisit Example 6 from the introduction and prove that the language "some letter appears at least twice" cannot be computed by a protocol. Suppose, towards a contradiction, that there is a protocol with k rounds that computes this language. Consider an input string with $2k + 2$ pairwise different letters, split so that Alice and Bob get $k + 1$ letters each. In the execution of the protocol there are at most k atoms which are sent as messages. In particular, there must be some atom a that appears in Alice's part of the input string, but is not sent as a message, and similarly there must be some atom b that appears in Bob's part of the input string, but is not sent as a message. Consider an atom permutation π which swaps a with b. If we apply this atom permutation to Alice's part of the string (but not Bob's), then the communication history will remain unchanged. In particular, the output of the protocol will be the same on both inputs. However, after applying this permutation, the input string has a repetition, unlike the original one. □

6.1.1 Orbit-finite sets

In this section, we do a more systematic analysis of automata models for infinite alphabets, and their relationship to our protocols. As an organising principle, we use the approach of orbit-finite sets,

see [Boj17] for a longer survey, which is a generalisation of finite sets suitable for infinite alphabets sets. Using this notion, we can lift any model of computation that uses finite sets, to a model that uses orbit-finite sets, which allows us for a clean comparison of the two setting. For the purposes of this paper, we use a simplified definition of orbit-finite sets, which is sufficient for our purposes[5].

Definition 6.2 (Orbit-finite sets)**.** *An* orbit-finite set *is any set of the form*

$$\mathbb{A}^{d_1} + \cdots + \mathbb{A}^{d_n},$$

for some natural numbers $d_1, \ldots, d_n \in \{0, 1, \ldots\}$.

When a summand $\mathbb{A}^{d_i}$ uses $d_i = 0$, then it describes a set with exactly one element, namely the empty tuple. Therefore, orbit-finite sets generalise finite sets, since a finite set with n elements can be seen as the orbit-finite set which has n disjoint copies of $\mathbb{A}^0$. We will only be interested in subsets of orbit-finite sets and functions between them that are *equivariant*, i.e. invariant under permutations of the atoms, in the following sense:

$$\underbrace{x \in X \iff \pi(x) \in X}_{\substack{\text{equivariant subset } X \\ \text{of an orbit-finite set}}} \qquad \underbrace{f(x) = y \iff f(\pi(x)) = \pi(y)}_{\substack{\text{equivariant function } f \\ \text{between two orbit-finite sets}}}$$

We can now discuss various orbit-finite models of computation, by generalising finite sets to orbit-finite ones, and requiring all subsets and relations to be equivariant. As a first example of this approach, we can revisit the definition of simple equivariant protocols from Section 6.1, and define it in terms of orbit-finite sets:

Definition 6.3 (Orbit-finite protocol)**.** *An* orbit-finite Boolean two-party protocol *is defined in the same way as in Definition 2.1, except that:*

1. *the input alphabet* Σ*, and the message spaces* Q_A *and* Q_B *are orbit-finite; and*
2. *the strategies of both players and the output function are equivariant.*

Indeed, the simple equivariant protocols from Section 6.1 are the special case of the above definition where the input alphabet is $\mathbb{A}$, and the message spaces are both equal to

$$\underbrace{\mathbb{A}}_{\text{letter}} + \underbrace{\mathbb{A}^0}_{\text{bit 0}} + \underbrace{\mathbb{A}^0}_{\text{bit 1}}.$$

On the other hand, the special case is also equivalent to the general case, since an element of a general orbit-finite set can be transmitted using a constant number of bits and atoms (by first sending the index of the summand, and then sending the tuple of atoms). It follows that the protocols from

[5]Definition 6.2 is weaker than the usual notion of orbit-finite sets [Boj25b, Section 5]; in fact it is the special case of the usual notion that is called *polynomial orbit-finite sets* in [Boj25b, Section 1]. The stronger notion that is usually used allows for two extra features: (a) restricting to equivariant subsets (e.g. one could limit $\mathbb{A}^2$ to pairs which are non-repeating); and (b) symmetries (e.g. one could identify pairs in $\mathbb{A}^2$ if they agree up to swapping of coordinates, thus yielding unordered pairs). In some cases, the extra features are desirable, in particular they establish a connection with set theory [Bla16] and nominal sets [Pit13, Section 5]. However, those features do not play any role in the analysis of protocols and automata and, to avoid technicalities, we use the simpler polynomial version from Definition 6.2. This simplification is purely technical — all results continue to hold for the usual notion of orbit-finite sets.

Definition 6.3 and those from Section 6.1 have the same expressive power. From now on we will use the formalisation from Definition 6.3.

Orbit-finiteness can also be used to define automata. The following definition has the same expressive power as the standard (nondeterministic and deterministic) register automata for infinite alphabets from [KF94]; this equivalence was shown in [BKL14, Lemma 6.3] and is one of the original motivations for studying orbit-finiteness.

Definition 6.4 (Orbit-finite automata)**.** *A* nondeterministic orbit-finite automaton *is defined in the same way as a nondeterministic finite automaton, except that all sets are orbit-finite, and all subsets and functions are equivariant:*

$$\overbrace{\underbrace{Q}_{\text{states}} \quad \underbrace{\Sigma}_{\substack{\text{input}\\ \text{alphabet}}}}^{\text{orbit-finite}} \qquad \overbrace{\underbrace{I \subseteq Q}_{\substack{\text{initial}\\ \text{states}}} \quad \underbrace{F \subseteq Q}_{\substack{\text{final}\\ \text{states}}} \quad \underbrace{\Delta \subseteq Q \times \Sigma \times Q}_{\text{transitions}}}^{\text{equivariant}}.$$

A deterministic orbit-finite automaton *is the special case which has exactly one initial state, and where the transition relation is a function.*

As stated in Figure 2, deterministic and nondeterministic orbit-finite automata have different expressive power. Moreover, as stated in Examples 5 and 6 none of these models is equivalent to orbit-finite protocols: deterministic automata are too weak, and nondeterministic automata are too strong.

In Definition 6.4, we have defined one-way orbit-finite automata, which read the input string from left to right. A natural extension are the two-way automata, which can move their reading head in both directions according to their transition function. This extension is particularly natural in the context of protocols, with their two-way interaction between the communicating parties. However, in the orbit-finite setting, two-way automata are very strong:

Example 19. [Two-way too strong] The language "some letter appears at least twice" can also be recognised by a deterministic two-way orbit-finite automaton [Boj25b, Example 18]. Therefore, this automaton model cannot be simulated by protocols.

The reason why two-way orbit-finite automata are so strong is that they can make an unbounded number of visits to any given position – for example the automaton for the language "some letter appears at least twice" will visit the last position a linear number of times (where the length of its run is quadratic). One idea to tame this power is to consider the bounded-crossing variant of two-way automaton, which has a fixed bound k on the number of times that the automaton can visit a position [NS03, p. 92]. We believe that this model can actually be equivalent to the protocols. However, in our conjecture, we will focus on the better studied model presented in the following section. □

6.2 Unambiguous orbit-finite automata

As we have shown in Examples 6, 5 and 19, protocols are not equivalent to one-way deterministic or nondeterministic orbit-finite automata, or their two-way variants automata. So what is the right automaton model? We conjecture that the answer is *unambiguous orbit-finite automata*, i.e. the special case of nondeterministic orbit-finite automata that have zero or one accepting runs for every input string.

Conjecture 6.5. *A language over an orbit-finite alphabet is computed by an orbit-finite protocol if and only if it is recognised by an unambiguous orbit-finite automaton.*

One corollary of this conjecture would be that unambiguous orbit-finite automata are closed under complement, since protocols can be complemented by flipping the output bit. This corollary has been conjectured in [Col12, p.9], and to the best of our knowledge remains open, despite apparent claims to the contrary in [Col15, Footnote 5].

In this section, we prove the $\Longleftarrow$ implication of the conjecture, i.e. we show that orbit-finite protocols can simulate unambiguous orbit-finite automata. Unlike similar results earlier in this paper, this simulation is non-trivial. Also, despite the one-way nature of the automata, the simulation crucially depends on the interactive nature of protocols, i.e. it requires more than one round of communication. In particular multi-round protocols cannot be reduced to one round, as was the case for finite alphabets (i.e. Lemma 3.4 is no longer true for the orbit-finite case).

Theorem 6.6. *If a language L over an orbit-finite alphabet is recognised by an unambiguous orbit-finite automaton, then it is also computed by an orbit-finite protocol.*

Proof. For the rest of this proof fix an unambiguous orbit-finite automaton, whose state space is the orbit-finite set Q. Suppose that the input string is factorized as $w = w_1 w_2$. The general idea of the protocol is that Alice and Bob will jointly compute the intermediate state (if it exists), i.e. the state q which satisfies:

$$\underbrace{I \xrightarrow{w_1} q}_{\substack{\text{there is a run over } w_1 \\ \text{from an initial state to } q}} \quad \text{and} \quad \underbrace{q \xrightarrow{w_2} F}_{\substack{\text{there is a run over } w_2 \\ \text{from } q \text{ to a final state.}}}$$

By unambiguity, there is at most one intermediate state, and it exists if and only if the string is accepted.

Observe that Alice can compute the set of states that are reachable from an initial state by reading her string w_1, and Bob can compute the set of states from which a final state is reachable by reading his string w_2. So, the challenge is to compute their intersection, which is either a singleton with the intermediate state, or the empty set. Before we explain how to do this, let us first explain why this is non-trivial, i.e. why Alice cannot send her set of states to Bob, or vice versa. The problem is that the set of all reachable subsets of Q might not be orbit-finite, and therefore it cannot be sent in a constant number of messages. This issue is illustrated in the following example.

Example 20. Consider the following unambiguous orbit-finite automaton for the language "the last letter appears at least twice". The automaton guesses a position in the input string that is the penultimate appearance of the last letter, stores this letter in a register, and verifies that its next appearance is indeed the last letter in the input word. Formally, it has the following orbit-finite set of states:

$$Q \quad = \quad \underbrace{1}_{\substack{\text{waiting for the} \\ \text{penultimate} \\ \text{appearance} \\ \text{of the last letter}}} \quad + \quad \underbrace{\mathbb{A}}_{\substack{\text{verifying that} \\ \text{the next appearance is} \\ \text{indeed the last letter}}} \quad + \quad \underbrace{1}_{\substack{\text{next appearance of} \\ \text{the guessed letter;} \\ \text{accept if at} \\ \text{the end of input}}}.$$

Observe that the automaton is unambiguous, as it takes care to accept only if the guessed position is indeed the penultimate appearance of the last letter.

Suppose now, that Alice and Bob want to simulate this automaton, and Alice has read a string $w_1 = a_1 \dots a_n$ of pairwise distinct letters. After reading w_1, the automaton can be in any of the following $n + 1$ states: (a) waiting for the penultimate appearance of the last letter; or (b) verifying that the next appearance of a_i is the last letter, for $i \in \{1, \dots, n\}$. It follows that the size of the set of reachable states after reading w_1 is unbounded (as it grows with n), and therefore it cannot be transmitted to Bob in a bounded number of orbit-finite messages. □

To work around the issue explained above, Alice and Bob will engage in interactive communication, which will narrow down the set of possible candidates. At each stage, it will be represented using orbits, as defined below.

Definition 6.7 (Orbit). *For a finite set $S \subseteq \mathbb{A}$, the S-*orbit *of $q \in Q$ is the following set:*

$$\{\pi(q) \mid \pi \text{ is a permutation of } \mathbb{A} \text{ such that } \pi(a) = a \text{ for all } a \in S \ \}.$$

The set S is called the support *of the orbit.*

Example 21. Let $Q = \mathbb{A}^5$, and consider the $\{\text{John, Eve}\}$-orbit of the following tuple

(John, Tom, Mary, Tom, Eve)

An element of this orbit is any tuple of the form

(John, a, b, a, Eve)

where a and b are distinct atoms, which are not John or Eve. □

As the support increases, the orbit becomes smaller; in particular the biggest orbits are the ones with empty support, i.e. the $\emptyset$-orbits, which we call the *equivariant orbits*. It is not hard to see that every orbit-finite set has a finite number of equivariant orbits [Boj25b, Lemma 1.4]; in fact this is the reason for the name. Each orbit in an orbit-finite set is a subset of $\mathbb{A}^d$ for some d. In such an orbit, we partition the coordinates $\{1, \dots, d\}$ into two parts: the *fixed coordinates*, which use the atoms from the support, and the *free coordinates*, which do not use these atoms. In Example 21, the fixed coordinates are the first and last ones, while the free coordinates are the middle three. The *dimension* of an orbit is the number of distinct atoms in the free coordinates. In Example 21, the dimension is two, corresponding to the atoms a and b. An important special case is when the dimension is zero; in this case the orbit contains only one state.

In the protocol, Alice and Bob will jointly maintain a set $S \subseteq \mathbb{A}$ and list of S-orbits which may contain the intermediate state (starting with $S = \emptyset$ the list of all $\emptyset$-orbits). They will ensure that if the intermediate state exists, i.e. if the input string is accepted, then the intermediate state is contained in one of the orbits on the list. The goal is to decrease the dimension of the orbits on the list until they become zero-dimensional, by gradually computing the set S of atoms that appear in the intermediate state. Once the orbits become zero-dimensional, they will contain only a finite (and bounded) number of the candidates. At this point, Alice can compute which of these candidates are reachable from an initial state over w_1 and send this (bounded) information to Bob, who can then check if one of these candidates can reach a final state over w_2.

To decrease the dimension and increase S, we will use the following lemma.

Lemma 6.8. *Let $S \subset \mathbb{A}$ be a finite set, and let $\varphi \subseteq Q$ be an infinite S-orbit of dimension k. Consider an input string $w = w_1 w_2$, and the sets:*

$$\underbrace{X_1 = \{q \in \varphi \mid I \xrightarrow{w_1} q\}}_{\text{states reachable on Alice's side}} \qquad \underbrace{X_2 = \{q \in \varphi \mid q \xrightarrow{w_2} F\}}_{\text{states reachable on Bob's side}}$$

There is a set $T \subseteq \mathbb{A}$ of size at most k, such that either:

1. *every state from X_1 contains an atom from T on some free coordinate; or*
2. *every state from X_2 contains an atom from T on some free coordinate.*

Proof. In the proof of the lemma, we use an analysis of disjointness, which is inspired by the sunflower lemma. We say that two states p, q in an S-orbit are *S-disjoint* if

$$\forall a \in \mathbb{A} \qquad a \text{ appears in both } p \text{ and } q \quad \Rightarrow \quad a \in S.$$

In other words, the atoms from S can appear in both p and q (in fact, they must), but all other atoms must be disjoint in the following two states. For example, if we take the $\{\text{John}, \text{Eve}\}$-orbit from Example 21, then the two states

(John, Tom, Mary, Tom, Eve) (John, Ann, Timmy, Ann, Eve)

are $\{\text{John}, \text{Eve}\}$-disjoint, because the sets $\{\text{Tom}, \text{Mary}\}$ and $\{\text{Ann}, \text{Timmy}\}$ are disjoint.

The following claim characterises subsets of orbits that do not contain any pair of disjoint elements:

Claim 6.9. *Let Q be an S-orbit of dimension d, and let X be a subset of Q. If X does not contain two S-disjoint elements, then there is a set T of at most d atoms such that every element of X uses at least one atom from T on a free coordinate.*

Proof. Take some element $x \in X$. Either there is an element of X that is completely disjoint with x, or otherwise some atom from x must appear in every other element of X on a free coordinate. □

The claim leaves us with showing that at least one of X_1 or X_2 does not contain an S-disjoint pair of elements. Suppose, towards a contradiction that both X_1 and X_2 contain S-disjoint pairs of elements, say $p_1, p_2 \in X_1$ and $q_1, q_2 \in X_2$. It follows that the two pairs (p_1, p_2) and (q_1, q_2) are in the same equivariant orbit (of $Q \times Q$), i.e. there is some atom permutation π which sends p_1 to q_1 and p_2 to q_2. Applying π to Alices's part of the input string, we obtain a new input string $\pi(w_1)w_2$, in which both q_1 and q_2 are valid intermediate states. It follows that there are at least two accepting runs (one that passes through q_1 and one that passes through q_2), contradicting the unambiguity assumption. □

Using the above lemma, we will construct a protocol that simulates the automaton. As explained before, the idea is to narrow down orbit which contains the intermediate state. This idea is formalised in the following lemma.

Lemma 6.10. *Let S be a finite subset of atoms, and let $X \subseteq Q$ be an orbit. Alice and Bob can exchange a constant number of messages – which depends only on the dimension of X – to determine if the intermediate state belongs to X.*

Before proving the lemma, let us explain how to use it to complete the proof of Theorem 6.6. We know that the set of all states Q splits into constant number of equivariant orbits, so the two parties can run the protocol the lemma for each of these orbits with $S = \emptyset$. Each run of the protocol uses a constant number of rounds, so the total number of rounds is also constant. It remains to prove the lemma.

Proof of Lemma 6.10. The proof proceeds by induction on the dimension of the orbit X.

The induction basis is when the dimension is zero. In this case, the orbit has exactly one state, and Alice and Bob can simply check if the state is reachable on their side and exchange this bit of information.

Consider now the induction step. Apply Lemma 6.8, to the orbit. In the factorisation $w = w_1 w_2$, at least one of the two alternatives in the conclusion of Lemma 6.8 must hold. Alice can check if the first alternative holds, and Bob can check if the second alternative holds. At least one of the two parties must report success, which is witnessed by some finite set T of atoms. The successful party sends the set T to the other party. This is possible since the size of T is bounded by the dimension of X. The orbit X splits into finitely many orbits $X_1, \ldots, X_n$ with the larger support $S \cup T$, see [Boj25b, Lemma 10.9]. The number n depends only on the dimension of X (as the size of T is bounded by the dimension of X).

We know that the intermediate state contains at least one atom from T, so we are only interested in the orbits among $X_1, \ldots, X_n$ which use at least one atom from T on a coordinate that was free in X. These orbits have lower dimension, so the parties can sequentially apply the induction assumption to check if the intermediate state belongs to any of these orbits. This completes the proof of the lemma, and therefore also of Theorem 6.6. □

□

6.3 Weighted automata

In Theorem 6.6, we have proved one implication in Conjecture 6.5. This section is devoted to presenting some evidence for the other implication, i.e.

$$\text{protocol} \quad \Longrightarrow \quad \text{unambiguous automaton.} \tag{7}$$

We begin by explaining why the orbit-finite case cannot be handled using the techniques that were used to prove this implication in the finite case.

What goes wrong in the orbit-finite case? In the finite case, the proof had two parts: (a) a reduction to one-round protocols, and (b) the Myhill-Nerode Theorem. Part (b) does not seem to be problematic, as orbit-finite versions of the Myhill-Nerode Theorem are known in many variants, including monoids [Boj13, Lemma 3.3], automata [BKL14, Section 3.2], and – as we will prove later in this section – also for weighted automata. The problematic part is (a), in which the number of rounds is reduced to one. The key argument in this reduction was that the sets of strategies

$$(Q_B)^k \to (Q_A)^k \qquad \text{and} \qquad (Q_A)^k \to (Q_B)^k$$

are finite, and thus each party could simply send their strategy as a message. This argument fails to carry over from finite sets to orbit-finite sets. The reason is that orbit-finite sets are not closed

under taking function spaces $X \to Y$, see [BNS24] for an extended discussion of this phenomenon. The following example shows that the one-round reduction is indeed impossible in the orbit-finite case.

Example 22. [No reduction to one round] Consider a language L that is computed by an orbit-finite protocol with one round. Using the same argument as in Lemma 2.2, we can show that the Myhill-Nerode equivalence relation for the language, as defined in (3), has an orbit-finite set of equivalence classes. As mentioned above, it follows from [BKL14, Section 3.2] that L is also recognised by a deterministic orbit-finite automaton. As we have seen in Example 5, such automata are not strong enough to capture all protocols. □

In light of the above example, it is no longer surprising that the proof of Theorem 6.6 used multi-round protocols. In fact, we believe that the number of needed rounds can be arbitrarily large, as suggested by the following example.

Example 23. [Many rounds] We think of a string over the alphabet $\mathbb{A}^2$ as a representation of a partial function of type $\mathbb{A} \to \mathbb{A}$. The general idea is that we view the input string as consisting of a list of (input, output) pairs, and if an input appears several times, then the leftmost occurrence is binding. More formally, the function is defined as follows: if the input is a, then we search for the leftmost letter (a, b) that appears in the input string and has a as the first component. If there is such a letter, then the output is b, otherwise the ouptut is undefined. Using this representation, we define for each $k \in \{0, 1, \ldots\}$ a language as follows:

$$L_k = \{w \in (\mathbb{A}^2)^* \mid \begin{array}{l}\text{if } f \text{ is the partial function represented by } w, \text{ then} \\ f^k(a) = b \text{ where } (a, b) \text{ is the last letter of } w\end{array}\}.$$

This language can be computed by an orbit-finite protocol with k rounds, with each round corresponding to one iteration of the function. It seems unlikely that any bounded number of rounds would suffice for all L_k, but we do not prove this claim here. □

Weighted orbit-finite automata. In the reminder of this section, we present some evidence for the missing implication in Conjecture 6.5, using the orbit-finite version of weighted automata. Namely, we will prove Theorem 6.12, which states that every language computed by an orbit-finite protocol is also recognised by a weighted orbit-finite automaton over the two-element field. For finite alphabets, this would be enough to ensure regularity, see Claim 5.10. This claim is no longer true in the orbit-finite case, see Example 24, and therefore Theorem 6.12 can only be considered as evidence for the conjecture. However, at the very least it shows that languages computed by orbit-finite protocols are decidable, which was not a priori clear from the definition.

Let us begin by defining the orbit-finite version of weighted automata.

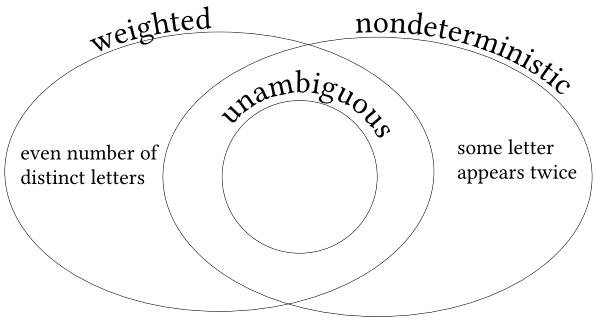
Definition 6.11 (Weighted orbit-finite automata)**.** *A* weighted orbit-finite automaton *over a semiring $\mathbb{D}$ is defined in the same way as in Definition 4.1, except that:*

1. *the input alphabet and state space are orbit-finite, instead of finite;*
2. *the functions in* (4) *are equivariant.*

We require that for every input string, there are only finitely many runs with non-zero weight.

For the purpose of this section, already the special case of the two-element field $\{0, 1\}$ is interesting. In this case, the automaton defines a function $\Sigma^* \to \{0, 1\}$, which can be seen as the characteristic function of a language. Therefore, we can compare weighted orbit-finite automata to other models, such as nondeterministic orbit-finite automata. The following example shows that these two models are incomparable.

Example 24. The language "some letter appears twice" is recognised by a nondeterministic orbit-finite automaton, but its characteristic function cannot be recognised by a weighted orbit-finite automaton over the two-element field. The non-expressivity can be proved using the orbit-finite version of the Fliess Theorem, see Theorem 6.19. On the other hand, the language "an even number of distinct letters" is not recognised by a nondeterministic orbit-finite automaton, while its characteristic function can be computed by a weighted orbit-finite automaton, see [BFKM24, Example 3.2]. Therefore, the two models – nondeterministic and weighted in the two-element field – are incomparable. Inside the intersection of these two classes we will find the unambiguous automata, since for unambiguous automata, counting the runs modulo two gives the same result as checking if a run exists. This discussion is summed up in the following picture:



□

The following theorem is the main result of Section 6.3.

Theorem 6.12. *Let Σ be an orbit-finite input alphabet, and let $\mathbb{D}$ be a field. If a language $L \subseteq \Sigma^*$ is computed by a protocol, then the corresponding characteristic function of type $\Sigma^* \to \{0, 1\} \subseteq \mathbb{D}$ is computed by a weighted orbit-finite automaton.*

In the proof of the theorem, we use the recently developed theory of orbit-finite vector spaces [BFKM24]. To streamline the presentation, we will use a special case of these spaces, namely those that have an orbit-finite basis. We begin by summarizing the necessary background: For an orbit-finite set Q, let us write $\mathrm{Lin}\, Q$ for the vector space which consists of finite formal linear combinations of elements of Q. In other words, an element of this space is a vector of the form

$$\alpha_1 q_1 + \cdots + \alpha_n q_n,$$

where the coefficients α_i are from the field, and the element q_i (which can be seen as basis vectors) are from Q. We use such spaces as the orbit-finite generalisation of finite dimension[6].

[6]Similarly to the case of orbit-finite sets, we use a simplified definition for this paper, as compared to the literature. The usual notion of vector spaces for orbit-finite sets, see [Boj25b, Definition 8.1] allows for more features; these features are irrelevant for our application and hence not discussed here.

Definition 6.13 (Vector space of orbit-finite dimension)**.** *A* vector space of orbit-finite dimension *is a vector space of the form* $\operatorname{Lin} Q$ *for some orbit-finite set* Q*.*

A space as in the above definition is equipped with two structures: as a vector space it is closed under linear combinations, and as a set with atoms it is closed under applying atom permutations. We will typically be interested in functions between such spaces that preserve both of those structures.

In the proof of Theorem 6.12, we will use an orbit-finite version of protocols with field outputs. Recall that there are two variants: the general version from Section 1.2 and the simpler scalar-product protocols from Section 4. We could begin with the general version and show that it is equivalent to the scalar-product one – this is indeed the case. However, to keep the exposition concise, we treat orbit-finite protocols with field outputs merely as a tool for proving Theorem 6.12, rather than as an object of independent interest. Therefore, we focus on the simplest form of protocol required for the proof, namely an orbit-finite version of the scalar product protocols from Section 4. In the orbit-finite case, instead of scalar products we will use the slightly more general notion of bilinear maps, i.e. maps that have two arguments and are linear in each argument separately[7].

Definition 6.14 (Orbit-finite bilinear protocol)**.** *An* orbit-finite bilinear protocol *consists of:*

1. *two vector spaces of orbit-finite dimension* V_A *and* V_B *and two strategies, which are equivariant functions*

$$\sigma_A : \Sigma^* \to V_A \quad \text{and} \quad \sigma_B : \Sigma^* \to V_B$$

2. *an output map, which is an equivariant bilinear map*

$$\mathit{out} : V_A \times V_B \to \mathbb{D}.$$

The output of the protocol is defined in the natural way: Alice and Bob apply their strategies to their local strings, yielding two vectors, and the output of the protocol is obtained using the output map. As usual, we require split invariance, i.e. the output of the protocol should depend only on the input string w and not on its factorisation $w = w_1 w_2$.

Let us go back to the proof of Theorem 6.12. The proof has two steps, as described in the following diagram.

orbit-finite protocol

$\Downarrow$ Lemma 6.15

orbit-finite bilinear protocol

$\Downarrow$ Claim 6.21

orbit-finite weighted automaton

We begin with the first step, which can be seen as form of reduction to one round. Recall that without vector spaces, a reduction to one round was not possible, see Example 22. This phenomenon is connected to closure under taking function spaces: orbit-finite sets are not closed under taking function spaces, but this closure is recovered once one moves to vector spaces, see [Boj25b, Section 8.3].

[7] It is possible to define orbit-finite scalar product protocols, but they are harder to work with. In particular, from our results it will follow that orbit-finite scalar product protocols (suitably defined) and orbit-finite bilinear protocols are equivalent, but we are not aware of a direct proof of this fact.

Lemma 6.15. *If $L \subseteq \Sigma^*$ is computed by an orbit-finite protocol, then its characteristic function is computed by an orbit-finite bilinear product protocol, over any field.*

Proof. We first introduce a common generalisation of the two models called *hybrid protocols*. In such a protocol, there are multiple rounds of messages, followed by a bilinear operation. We will show that the multiple rounds can be eliminated, yielding a plain orbit-finite bilinear protocol.

Here is the formal definition of the hybrid protocol: For each round $i \in \{1, \ldots, k-1\}$, the parties exchange messages just as in an orbit-finite protocol from Definition 6.3, using message spaces Q_A and Q_B and strategies of the following types:

$$\sigma_{A,i} : \Sigma^* \times (Q_B)^{i-1} \to Q_A$$
$$\sigma_{B,i} : \Sigma^* \times (Q_A)^{i-1} \to Q_B$$

Then, in the last k-th round, the message histories are used to produce vectors in two vector spaces V_A and V_B of orbit-finite dimension, as in Definition 6.13, using strategies of types

$$\sigma_{A,k} : \Sigma^* \times (Q_B)^{k-1} \to V_A$$
$$\sigma_{B,k} : \Sigma^* \times (Q_A)^{k-1} \to V_B.$$

Finally, from the two vectors, the output is computed using a bilinear map

$$\text{out} : V_A \times V_B \to \mathbb{D}.$$

The hybrid protocol generalises both orbit-finite protocols and orbit-finite bilinear protocols. For the latter, this is clear: we simply use $k = 1$ and there is no message exchange. For the former, we proceed as follows. We use trivial vector spaces, i.e. both V_A and V_B are the field. The bilinear map is multiplication. Once the two parties have agreed on a Boolean decision, they can both send 1 (in the case of a "yes" decision) or 0 (in the case of a "no" decision), and the bilinear map will give the correct output.

In order to complete the proof of the lemma, we will show that the number of rounds can always be reduced to one, thus yielding a bilinear protocol.

Claim 6.16. *For every $k > 1$, a hybrid protocol with k rounds can be simulated by a hybrid protocol with $k-1$ rounds.*

Proof. We will eliminate round $k-1$, where the last message is sent. Once Alice has received the first $k-2$ messages from Bob, her contribution to the rest of the protocol is described by an object of type

$$\underbrace{Q_A}_{\substack{\text{message}\\ \text{sent in}\\ \text{round } k-1}} \times \underbrace{(Q_B \xrightarrow[\text{fs}]{} V_A)}_{\substack{\text{message sent in}\\ \text{round } k, \text{ as a function}\\ \text{of the message sent}\\ \text{in round } k-1}} \tag{8}$$

Before we continue, let us now explain the "fs" annotation on the arrow above. It stands for the set of *finitely supported functions*, which is a relaxation of equivariant functions. The general

idea is that while equivariant functions can only refer to equality between letters, the finitely supported functions can also use finitely many constants. Before presenting the definition, we give two examples. The first example is the function of type $\mathbb{A} \to \mathbb{A}$ defined by

$$a \in \mathbb{A} \mapsto \begin{cases} \text{John} & \text{if } a = \text{Eve} \\ a & \text{otherwise.} \end{cases} \tag{9}$$

This function is finitely supported but not equivariant. Here is a second example of finitely supported functions, which is particularly relevant to our application.

Example 25. [Partial application not equivariant] A partial application of a function

$$f : X \times Y \to Z$$

to an argument $x \in X$ is a function $f(x, -) : Y \to Z$, defined as $f(x, -)(y) = f(x, y)$. Even if f is equivariant, then the partially applied function $f(x, -)$ might not be equivariant. For example, consider the equivariant function

$$f : \mathbb{A}^* \times \mathbb{A} \to \{\text{true}, \text{false}\},$$

such that $f(w, a) = \text{true}$ if and only if a appears in w. Now, consider the partial application

$$f' := f(\text{Celine John Anne}, -)$$

which is of type $\mathbb{A} \to \{\text{true}, \text{false}\}$. This function is no longer equivariant, because the input needs to be compared to the constants Celine, John, and Anne. However, the number of constants is finite, hence it is finitely supported. □

Definition 6.17 (finitely supported function)**.** *A function $f : X \to Y$ is* finitely supported *if there is a finite set S of atoms such that for every atom permutation π we have*

$$(\forall a \in S\ \pi(a) = a) \quad \Longrightarrow \quad f(\pi(x)) = \pi(f(x)).$$

An important property of finitely supported function spaces is that they do not preserve orbit-finiteness:

Example 26. If we take two orbit-finite sets, then the set of finitely supported functions between them might not be orbit-finite. For example, the set

$$\mathbb{A} \underset{\text{fs}}{\longrightarrow} \{\text{true}, \text{false}\}$$

of finitely supported functions from $\mathbb{A}$ to the Booleans is not orbit-finite, since such functions can refer to any number of constants, as demonstrated by the partial applications in Example 25. □

As mentioned before, one can recover closure under taking function spaces by moving to vector spaces. This is formalized in the following lemma, which stems from [BFKM24], but we use its formulation from [Boj25b], as it is more convenient for our purposes:

Lemma 6.18 ([Boj25b, Lemma 8.17])**.** *For all orbit-finite sets X and Y, the space of finitely supported functions $X \underset{fs}{\longrightarrow} \operatorname{Lin} Y$ is a vector space of orbit-finite dimension.*

Let us now close this digression about function spaces (for more discussion, see [Boj25b, Section 8.3] or [BNS24]) and go back to the proof of the claim. Observe, that the second coordinate in (8) is indeed of type

$$Q_B \xrightarrow[\text{fs}]{} V_A,$$

as it arises from a partial application of the equivariant function $\sigma_{A,k}$ to Alice's part of the input (i.e. $w_1 \in \Sigma^*$) and her message history from the first $k-2$ rounds (i.e. an element of $(Q_B)^{k-2}$). Therefore, in order to transmit it to Bob, we need to turn the type in (8) into a vector space. The second coordinate is already a vector space, since functions with outputs in a vector space can be added and scaled pointwise. What is more, by Lemma 6.18, it has an orbit-finite basis:

$$F_A \subseteq Q_B \xrightarrow[\text{fs}]{} V_A.$$

The first coordinate Q_A in (8) can be turned into a vector space by allowing linear combinations, i.e. $\operatorname{Lin} Q_A$. We combine these two using tensor product, yielding a vector space of orbit-finite dimension

$$W_A = (\operatorname{Lin} Q_A) \otimes (\operatorname{Lin} F_A).$$

We can do the same thing for Bob, obtaining a vector space

$$W_B = (\operatorname{Lin} Q_B) \otimes (\operatorname{Lin} F_B),$$

where F_B is an orbit-finite basis of the vector space

$$Q_A \xrightarrow[\text{fs}]{} V_B.$$

In order to define an equivariant bilinear map of type

$$\varphi : W_A \times W_B \to \mathbb{D}$$

it is enough to define an equivariant linear map of type

$$\varphi : W_A \otimes W_B \to \mathbb{D}$$

for which it is enough to define an equivariant function on its basis

$$Q_A \times F_B \times Q_B \times F_A \to \mathbb{D}$$

This definition is the only one that types, namely

$$(q_A, f_B, q_B, f_A) \quad \mapsto \quad \text{out}(f_A(q_B), f_B(q_A)).$$

Because the output map is bilinear, one can check that φ defined this way is consistent with the original protocol, i.e. if we take functions

$$f_A : Q_B \to V_A \qquad \text{and} \qquad f_B : Q_A \to V_B,$$

which are not necessarily basis vectors from F_A and F_B, then we have

$$\mathrm{out}(f_A(q_B), f_B(q_A)) = \varphi((q_A, f_B), (q_B, f_A)).$$

Therefore, we can implement the last two round of the original hybrid protocol using a single round. The message spaces and the strategies for the first $k-2$ rounds are unchanged. In the last round $k-1$, the new strategies

$$\sigma'_{A,k-1} : \Sigma^* \times (Q_B)^{k-2} \to W_A$$
$$\sigma'_{B,k-1} : \Sigma^* \times (Q_A)^{k-2} \to W_B$$

output the tensor pairs consisting of the contribution that was described in (8). Finally, the output map for the new protocol is φ. □

By repeatedly applying the above claim, we can reduce the number of rounds to one, in which case we get a bilinear protocol, as required in the statement of the lemma. □

6.3.1 Orbit-Finite Fliess Theorem

In this section, we prove an orbit-finite version of the Fliess Theorem, which characterises functions $\Sigma^* \to \mathbb{D}$ that are computed by weighted orbit-finite automata. This result will be used to complete the proof of Theorem 6.12.

As in the original Fliess Theorem, we will be interested in derivatives of the function, which live in the space

$$\Sigma^* \to \mathbb{D}.$$

This space has three kinds of structure, all of which will are used in the Fliess Theorem:

1. It is a vector space, since we can take linear combinations of functions.
2. It has a notion of left derivatives, i.e. for each function f and input string $w \in \Sigma^*$, we can consider the left derivative $v \mapsto f(wv)$, which is denoted by $f(w\_)$.
3. It has a notion of atom permutations: for each function f and atom permutation π, we can consider the function $\pi(f)$, which is the composition $\pi; f$.

We say that a subset $U \subseteq \Sigma^* \to \mathbb{D}$ is *orbit-finitely spanned* if there is some orbit-finite set Q, such that every element of U is a finite linear combination of elements from Q. We do not require the linear combination to be unique, i.e. we do not require Q to be a basis. (Choosing a basis can be problematic in the context of orbit-finite sets, see [Boj25b, Example 77].) We are now ready to state the orbit-finite version of the Fliess Theorem.

Theorem 6.19 (Orbit-Finite Fliess Theorem)**.** *The following two conditions are equivalent for every function $f : \Sigma^* \to \mathbb{D}$ where Σ is an orbit-finite alphabet and $\mathbb{D}$ is a field.*

1. *f is computed by a weighted orbit-finite automaton;*
2. *f is equivariant and its set of left derivatives is orbit-finitely spanned.*

Proof. Our proof follows the lines of the original theorem, without any significant changes.

We begin with the implication 1 $\implies$ 2. Consider a weighted orbit-finite automaton with state space Q. Define the *pre-weight* of a run in the same way as its weight, except that we do not use the final weight. In other words, this is the product of: (1) the initial weight of the first state; and (2) the weights of all transitions. Consider an input string w. Define the *configuration* of w to be the linear combination

$$\sum_{\rho} \alpha_\rho \cdot q_\rho, \tag{10}$$

where ρ ranges over runs that have input w and non-zero pre-weight, α_ρ is the pre-weight of the run ρ, and q_ρ is the last state in this run. By the assumption that each input string has finitely many runs with non-zero weight, the configuration is a finite sum, i.e. it belongs to the vector space $\operatorname{Lin} Q$. The left derivative which corresponds to the input string is uniquely determined by this configuration, and the space of configurations is orbit-finitely spanned. Hence, we get 2.

We now prove the other implication, 2 $\implies$ 1. Assume 1, which means that there is an orbit-finite set $Q \subseteq \Sigma^*$ such that every derivative of f can be decomposed as a finite linear combination of left derivatives

$$\sum_i \alpha_i f(w_i\_),$$

where each string w_i is in Q. The following claim shows that strings obtained by taking elements of Q and appending one letter (i.e. strings of the form $Q \cdot \Sigma$) can be decomposed in an equivariant way:

Claim 6.20. *There is an equivariant function*

$$\delta : Q \times \Sigma \to \operatorname{Lin} Q$$

such that the following conditions holds for every $w \in Q$ and $a \in \Sigma$:

$$\delta(w, a) = \sum_i \alpha_i w_i \qquad \Rightarrow \qquad f(wa\_) = \sum_i \alpha_i f(w_i\_).$$

Proof. Observe that Condition 2 in the theorem's statement already asserts that there exists such a function δ, possibly non-equivariant. In this proof we show how to modify it so that it becomes equivariant while still satisfying other requirements of the claim. We construct this modified function δ' as follows: for every orbit $Q \times \Sigma$ pick its representative (w, a) and then extend the result to the whole orbit by equivariance:

$$\delta'(\pi(w), \pi(a)) := \pi(\delta(w, a))$$

The new function δ' is equivariant by construction, and it keeps satisfying other requirements of the claim because f is an equivariant function, so its derivatives commute with atom permutation:

$$f(\pi(w)\_) = \pi(f(w\_)).$$

This gives us that:

$$f(wa\_) = \sum_i \alpha_i f(w_i\_) \qquad \iff \qquad f(\pi(wa)\_) = \sum_i \alpha_i f(\pi(w_i)\_),$$

which completes the proof of the claim. □

Using the function δ from the above claim, we define a weighted orbit-finite automaton. The state space is the set Q. (We assume without loss of generality that Q contains the empty string ε. This is not really necessary for the construction, but it makes it more intuitive.) The weights are defined as follows:

- **Initial weights.** The initial weight of ε is 1. All other states have initial weight 0.
- **Transition weights.** The weight of a transition

$$w \xrightarrow{a} v$$

  is the coefficient next v in the linear decomposition $\delta(w, a)$.

- **Final weights.** The final weight of a state $w \in Q$ is $f(w)$.

Remember that orbit-finite weighted automata can only admit finitely many runs for each input word. Our automaton satisfies this requirement, as all linear combinations returned by δ are finite.

Finally, we justify why this automaton computes the function f. A simple inductive proof shows that

$$\text{configuration of } w = \sum_\rho \alpha_\rho \cdot w_\rho \qquad \Longrightarrow \qquad f(w\_) = \sum_\rho \alpha_\rho \cdot f(w_\rho\_).$$

By choice of final weights, the output of the automaton is equal to $f(w)$. □

We are now ready to complete the proof of Theorem 6.12 by showing the missing link:

Claim 6.21. *Let $\mathbb{D}$ be a field. If a function $\Sigma^* \to \mathbb{D}$ is computed by an orbit-finite bilinear protocol, it can also be computed by an orbit-finite weighted automaton.*

Proof. Thanks to the Orbit-Finite Fliess Theorem, it is enough to show that if a function is computed by an orbit-finite bilinear protocol, then it has an orbit-finitely spanned vector space of left derivatives. This follows from the same argument as in Section 4: the vector produced by Alice in a bilinear protocol uniquely determines the left derivative of her part of the input. □

7 Conclusions

One could possibly consider other inputs, such as trees. It seems that at least some of our results could generalise to such inputs, as long as there would be a suitable theory of automata for the inputs, with theorems in the style of Myhill-Nerode (which is the case for trees). Nevertheless, we leave the the exploration of non-string inputs to future work.

References

[AČ10] Rajeev Alur and Pavol Černý. Expressiveness of streaming string transducers. In *Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, Chennai, India*, volume 8 of *LIPIcs*, pages 1–12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[ADD+13] Rajeev Alur, Loris D'Antoni, Jyotirmoy V. Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In *Logic in Computer Science, LICS, New Orleans, USA*, pages 13–22. IEEE Computer Society, 2013.

[AFR14] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In *Computer Science Logic and Logic in Computer Science, CSL-LICS 2014, Vienna, Austria,*, pages 1–10. ACM, 2014.

[BDK18] Mikołaj Bojańczyk, Laure Daviaud, and Shankara Narayanan Krishna. Regular and First-Order List Functions. In *Logic in Computer Science, LICS, Oxford, UK*, pages 125–134. ACM, 2018.

[BDM+11] Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27:1–27:26, 2011.

[BDSW17] Michael Benedikt, Timothy Duff, Aditya Sharad, and James Worrell. Polynomial automata: Zeroness and applications. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017.

[BE00] Roderick Bloem and Joost Engelfriet. A Comparison of Tree Transductions Defined by Monadic Second Order Logic and by Attribute Grammars. *Journal of Computer and System Sciences*, 61(1):1–50, 2000.

[BFKM24] Mikołaj Bojańczyk, Joanna Fijalkow, Bartek Klin, and Joshua Moerman. Orbit-Finite-Dimensional Vector Spaces and Weighted Register Automata. *TheoretiCS*, Volume 3, May 2024.

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

[BKL19] Mikolaj Bojanczyk, Sandra Kiefer, and Nathan Lhote. String-to-string interpretations with polynomial-size output. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, pages 106:1–106:14, 2019.

[BL10] Mikołaj Bojańczyk and Slawomir Lasota. An Extension of Data Automata that Captures XPath. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pages 243–252. IEEE Computer Society, 2010.

[Bla16] Andreas Blass. Power-dedekind finiteness. Unpublished manuscript at `www.math.lsa.umich.edu/~ablass/pd-finite.pdf`, 2016.

[BN23] Mikolaj Bojanczyk and Lê Thành Dung Nguyên. Algebraic recognition of regular functions. In Kousha Etessami, Uriel Feige, and Gabriele Puppis, editors, *50th International Colloquium on Automata, Languages, and Programming, ICALP 2023, July 10-14, 2023, Paderborn, Germany*, volume 261 of *LIPIcs*, pages 117:1–117:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[BNS24] Mikolaj Bojanczyk, Lê Thành Dung Nguyên, and Rafal Stefanski. Function spaces for orbit-finite sets. In Karl Bringmann, Martin Grohe, Gabriele Puppis, and Ola Svensson, editors, *ICALP*, volume 297 of *LIPIcs*, pages 130:1–130:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[Boj13] Mikołaj Bojańczyk. Nominal Monoids. *Theory Comput. Syst.*, 53(2):194–222, 2013.

[Boj14] Mikołaj Bojańczyk. Transducers with Origin Information. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 26–37. Springer, 2014.

[Boj17] Mikołaj Bojańczyk. Orbit-Finite Sets and Their Algorithms (Invited Talk). In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 1:1–1:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[Boj22] Mikołaj Bojańczyk. Transducers of polynomial growth. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '22, 2022.

[Boj23] Mikołaj Bojańczyk. Folding interpretations. In *Logic in Computer Science*, LICS '22. Association for Computing Machinery, 2023.

[Boj25a] Mikołaj Bojańczyk. An Automata Toolbox. Lecture notes available at `mimuw.edu.pl/~bojan/paper/automata-toolbox-book`, 2025.

[Boj25b] Mikołaj Bojańczyk. Slightly infinite sets. Lecture notes available at `mimuw.edu.pl/~bojan/paper/atom-book`. Accessed: July 3, 2025, 2025.

[BR08] Jean Berstel and Christophe Reutenauer. *Rational series and their languages*. EATCS Monographs, 2008.

[BS19] Paul Brunet and Alexandra Silva. A Kleene Theorem for Nominal Automata. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 107:1–107:13, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[BS20] Mikołaj Bojańczyk and Rafał Stefański. Single-Use Automata and Transducers for Infinite Alphabets. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *International Colloquium on Automata, Languages, and Programming (ICALP 2020)*, volume 168 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113:1–113:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[CCP20] Michaël Cadilhac, Olivier Carton, and Charles Paperman. Continuity of functional transducers: A profinite study of rational functions. *Logical Methods in Computer Science*, Volume 16, Issue 1, Feb 2020.

[CDTL23] Thomas Colcombet, Gaetan Doueneau-Tabot, and Aliaume Lopez. $\mathbb{Z}$-polyregular functions . In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, Los Alamitos, CA, USA, June 2023. IEEE Computer Society.

[Cho77] Christian Choffrut. Une caracterisation des fonctions sequentielles et des fonctions sous-sequentielles en tant que relations rationnelles. *Theoretical Computer Science*, 5(3):325–337, 1977.

[CJ77] Michal Chytil and Vojtech Jákl. Serial Composition of 2-Way Finite-State Transducers and Simple Programs on Strings. In *International Colloquium on Automata, Languages and Programming, ICALP, Turku, Finland*, volume 52 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 1977.

[CLP15] Thomas Colcombet, Clemens Ley, and Gabriele Puppis. Logics with rigidly guarded data tests. *Logical Methods in Computer Science*, 11(3), 2015.

[Col12] Thomas Colcombet. Forms of determinism for automata (invited talk). In *29th International Symposium on Theoretical Aspects of Computer Science (STACS 2012)*, pages 1–23. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2012.

[Col15] Thomas Colcombet. Unambiguity in automata theory. In *International Workshop on Descriptional Complexity of Formal Systems*, pages 3–18. Springer, 2015.

[CP17] Thomas Colcombet and Daniela Petrişan. Automata and minimization. *ACM SIGLOG News*, 4(2):4–27, May 2017.

[DKV09] Manfred Droste, Werner Kuich, and Heiko Vogler, editors. *Handbook of Weighted Automata*. Springer Berlin Heidelberg, 2009.

[DL09] Stéphane Demri and Ranko Lazic. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Log.*, 10(3):16:1–16:30, 2009.

[EH01] Joost Engelfriet and Hendrik Jan Hoogeboom. MSO Definable String Transductions and Two-way Finite-state Transducers. *ACM Trans. Comput. Logic*, 2(2):216–254, 2001.

[Eil74] Samuel Eilenberg. *Automata, languages, and machines. Vol. A*. Academic Press [A subsidiary of Harcourt Brace Jovanovich, Publishers], New York, 1974.

[EM65] C. C. Elgot and J. E. Mezei. On Relations Defined by Generalized Finite Automata. *IBM Journal of Research and Development*, 9(1):47–68, January 1965.

[EM02] Joost Engelfriet and Sebastian Maneth. Two-way finite state transducers with nested pebbles. In *International Symposium on Mathematical Foundations of Computer Science*, pages 234–244. Springer, 2002.

[EM03] Joost Engelfriet and Sebastian Maneth. Macro Tree Translations of Linear Size Increase are MSO Definable. *SIAM Journal on Computing*, 32(4):950–1006, January 2003.

[FGLM18] Emmanuel Filiot, Olivier Gauwin, Nathan Lhote, and Anca Muscholl. On Canonical Models for Rational Functions over Infinite Words. In Sumit Ganguly and Paritosh

Pandya, editors, *38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2018)*, volume 122 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In *Annual International Cryptology Conference*, pages 33–62. Springer, 2018.

[Fli74] Michel Fliess. Matrices de Hankel. *Journal de Mathématiques Pures et Appliquées*, 53:197–222, 1974.

[GKY22] Mika Göös, Stefan Kiefer, and Weiqiang Yuan. Lower Bounds for Unambiguous Automata via Communication Complexity. In Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, volume 229 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 126:1–126:13, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Gur82] Eitan M. Gurari. The Equivalence Problem for Deterministic Two-Way Sequential Transducers is Decidable. *SIAM J. Comput.*, 11(3):448–452, 1982.

[Hau89] George Hauser. *Communication Complexity*. PhD thesis, University of Rochester, 1989.

[HM88] David Charles Hobby and Ralph McKenzie. *The structure of finite algebras*, volume 76. American Mathematical Society Providence, 1988.

[KF94] Michael Kaminski and Nissim Francez. Finite-Memory Automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

[KT04] Michael Kaminski and Tony Tan. Regular expressions for languages over infinite alphabets. In Kyung-Yong Chwa and J. Ian J. Munro, editors, *Computing and Combinatorics*, pages 171–178, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Kus97] Eyal Kushilevitz. Communication complexity. In Marvin V. Zelkowitz, editor, *Advances in Computers*, volume 44 of *Advances in Computers*, pages 331–360. Elsevier, 1997.

[LTV15] Leonid Libkin, Tony Tan, and Dario Vrgoč. Regular expressions for data words. *Journal of Computer and System Sciences*, 81(7):1278–1297, 2015.

[NNP20] Lê Thành Dung Nguyên, Camille Noûs, and Pierre Pradic. Implicit automata in typed λ-calculi II: streaming transducers vs categorical semantics. *CoRR*, abs/2008.01050, 2020.

[NS03] Frank Neven and Thomas Schwentick. On the power of tree-walking automata. *Information and Computation*, 183(1):86–103, 2003.

[NSV04] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004.

[OP16] Nic Ormes and Ronnie Pavlov. Extender sets and multidimensional subshifts. *Ergodic Theory and Dynamical Systems*, 36(3):908–923, 2016.

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

[PS05] Jean-Éric Pin and Pedro V. Silva. A topological approach to transductions. *Theoretical Computer Science*, 340(2):443–456, 2005.

[Rey83] John C Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*, pages 513–523, 1983.

[RS91] Christophe Reutenauer and Marcel-Paul Schutzenberger. Minimization of rational word functions. *SIAM Journal on Computing*, 20(4):669–685, 1991.

[Sch61] Marcel Paul Schützenberger. On the definition of a family of automata. *Information and control*, 4(2–3):245–270, 1961.

[Seg06] Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In *International Workshop on Computer Science Logic*, pages 41–57. Springer, 2006.

[She59] J. C. Shepherdson. The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, 3(2):198–200, April 1959.

[SS71] Arnold Schönhage and Volker Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7(3):281–292, 1971.

[sta] Wikipedia page on state complexity. `en.wikipedia.org/wiki/State_complexity`.

[VZGG03] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge university press, 2003.

[Yao79] Andrew Chi-Chih Yao. Some complexity questions related to distributive computing(preliminary report). In *Proceedings of the eleventh annual ACM symposium on Theory of computing - STOC '79*, STOC '79, page 209–213. ACM Press, 1979.