Folding interpretations

Mikołaj Bojańczyk (University of Warsaw)

Abstract

We study the polyregular string-to-string functions, whi ch are certain functions of polynomial output size that can be described using automata and logic. We describe a system of combinators that generates exactly these functions. Unlike previous systems, the present system includes an iteration mechanism, namely fold. Although unrestricted fold can define all primitive recursive functions, we identify a type system (inspired by linear logic) that restricts fold so that it defines exactly the polyregular functions. We also present related systems, for quantifier-free functions as well as for linear regular functions on both strings and trees.

ACM Reference Format:
Mikołaj Bojańczyk (University of Warsaw). 2023. Folding interpretations. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

This paper is about transducers that compute string-to-string functions. (We also have some results on trees, but trees will be discussed only at the end of the paper.) We are interested in two classes of functions: the linear regular functions[1], which have linear output size, and the polyregular functions, which have polynomial output size. Both classes can be described by many equivalent models, and have robust closure properties.

Let us begin with the more established class of linear regular functions. Two typical example functions from this class are:

$$\underbrace{[1,2,3] \mapsto [1,2,3,1,2,3]}_{\text{duplicate}} \qquad \underbrace{[1,2,3] \mapsto [3,2,1]}_{\text{reverse}}.$$

The linear regular functions can be described by many equivalent models, including: deterministic two-way automata with output [23, Note 4], MSO transductions [13, Section 4], streaming string transducers [1, Section 3], an extension of regular expressions [3, Section 2], and a calculus based on combinators [7, Theorem 6.1]. The many equivalent models, as well as the robustness and good decidability properties of the underlying class, are comparable to similar properties for the regular languages, which also have many equivalent descriptions, including automata, logic and regular expressions. For this reason, the linear regular functions have been intensively studied in the last decade.

The second class is the polyregular functions, which extended the linear regular functions by allowing polynomial growth, including functions such as the squaring operation

$$[1,2,3] \mapsto [1,2,3,1,2,3,1,2,3].$$

Similarly to the linear regular functions, the polyregular functions can also be described by multiple models, including: string-to-string pebble transducers, which are introduced in [14, Section 1] based on [15, Definition 1.5] and [21, Section 3.1], as well as an imperative programming language [5, Section 3], a functional programming language [5, Section 4], and a polynomial extension of MSO transductions [9, Definition 2]. For a survey of the polyregular functions, see [6].

Combinators. This paper studies the linear regular and polyregular functions by using systems based on prime functions and combinators. This approach dates back to the Krohn-Rhodes Theorem [19, p. 454], and was first applied to linear regular functions in [7], by describing them in terms of certain prime functions, such as

$$1 + \Sigma \times \Sigma^* \to \Sigma^* \qquad \text{list constructor,}$$

and combinators such as

$$\frac{\Sigma \to \Gamma \quad \Gamma \to \Delta}{\Sigma \to \Delta} \qquad \text{function composition.}$$

This system is further extended in [5, p. 64] to cover the polyregular functions, by adding extra prime functions of non-linear output size, such as the squaring operation.

The systems in [5, 7] have no constructions for iteration; because of this design decision, the hard part is proving completeness: every function of interest can be derived in the system. One reason for avoiding iteration is to have a minimal system. Another reason is that iteration constructions are powerful, and as we find out in this paper, it is hard to add them while retaining soundness (only functions of interest can be derived).

The fold combinator. In this paper, we take the opposite approach, by studying an iteration construction, namely the

[1]These are usually called the regular functions in the literature, but we add the word "linear" to distinguish them from the polyregular functions.

fold combinator. This combinator can be written as a rule

$$\frac{1 \to \Gamma \quad \Gamma \times \Sigma \to \Gamma}{\Sigma^* \to \Gamma} \quad \text{fold}.$$

The assumption of this rule can be seen as a deterministic automaton with input alphabet Σ and state space Γ, given by its initial state and transition function. In the conclusion of the rule, we have the function that maps an input string to the last state of the run of the automaton. The input alphabet and the state space need not be finite, e.g. the state space Γ could be the set 1^* which represents the natural numbers.

Folding is a fundamental construction in functional programming languages. For example, the fold combinator arises canonically from the inductive definition of the list type [18, Section 3]. Unfortunately, there is a price to pay for the power and elegance of the fold combinator: one can use it to derive all primitive recursive functions [18, Section 4.1]. Therefore, without any further restrictions, the fold combinator falls outside the scope of automata techniques, or any other techniques that can be used to decide semantic properties of programs, such as the halting problem.

This paper is devoted to identifying restrictions on the fold combinator that tame its expressive power. These restrictions are presented as a typing system, which ensures that applications of fold will stay in the class of polyregular functions. In particular, the resulting class of functions shares the decidability properties of the polyregular functions, e.g. one can decide if a function produces a nonempty output for at least one input.

There are two main contributions in the paper.

Quantifier-free interpretations. The first contribution is to identify the quantifier-free interpretations as an important class of functions in the context of fold. These are functions on structures in which the universe of the output is a subset of the universe of the input (in particular, the output size is linear), and all relations in the output structure are defined using quantifier-free formulas.

In Theorem 3.2 we show that applying the fold combinator to a quantifier-free interpretation yields a function that, although not necessarily quantifier-free, is at least linear regular. This result subsumes several existing results, in particular those about MSO definability of streaming transducers [2, 3]. Although quantifier-free interpretations are rather weak, they can describe most natural transformations that are used as primes in the calculi from [5, 7]; the remaining primes can then be derived using fold.

Having identified the importance of quantifier-free functions, in Theorem 4.1, we present a system of prime functions and combinators that derives exactly the quantifier-free functions. The completeness proof of the system is the longest proof in the paper. The quantifier-free system does not allow fold; fold is used in the next part of the paper, about polyregular functions.

Safe fold. The second main contribution is a type system that tames the power of fold. This system uses a type constructor ! and bears certain similarities to the parsimonius calculus of Mazza [20, Section 2.2]. The latter is part of a field called *implicit computational complexity*, which seeks to describe complexity classes using type systems. An influential example of this kind is a system of Bellantoni and Cook [4], which characterizes polynomial time. The present paper can be seen as part of implicit computational complexity, which targets regular languages instead of Turing complete models, such as logarithmic space or polymomial time. For a more detailed discussion of the connections between regular languages and λ-calculus, including a pioneering applicaton of linear types, see [22].

The usual application of ! is to restrict duplication, and this paper is no exception, as in the following example:

$$\underbrace{x \mapsto (x, x)}_{\text{not allowed}} \qquad \underbrace{!x \mapsto (!x, x)}_{\text{allowed}}.$$

However, apart from restricting duplication, ! is also used in this paper to restrict another, more mysterious, resource, namely quantifiers. The idea is that our system uses ! used to describe functions that are not necessarily quantifier-free, but are similar enough to quantifier-free functions so that the fold combinator can be applied to them.

The second main contribution of this paper is Theorem 5.3, which characterizes the polyregular functions using certain prime functions and combinators, in which the types involve ! and one of the combinators is fold. In Theorem 6.1 we also show that if we further restrict duplication

$$\underbrace{!x \mapsto (!x, x)}_{\text{not allowed}} \qquad \underbrace{!x \mapsto (x, x)}_{\text{allowed}},$$

then the resulting system derives exactly the linear functions. Finally, we also show that the results about the linear case can be extended from strings to trees without much difficulty.

2 Interpretations

In this section, we describe the polyregular functions. Among several equivalent definitions of the polyregular functions, our point of departure in this paper will be a definition that uses MSO interpretations [9, Section 2].

2.1 Definition of MSO interpretations

We assume that the reader is familiar with basic notions of monadic second-order logic MSO, see [17] for an introduction. We only describe the notation that we use. A *vocabulary* consists of a finite set of relation names, each one with an associated arity in $\{0, 1, \ldots\}$. Note that we allow nullary relations, i.e. relations of arity zero; such a relation takes no arguments and is "true" or "false" in each structure. A *structure* over such a vocabulary consists of a finite nonempty set, called the *universe* of the structure, and an interpretation of the vocabulary, which associates to each relation name in the vocabulary a relation over the universe of matching arity. The syntax and semantics of first-order logic and MSO are defined in the usual way. Whenever we speak of a *class of structures*, all structures in the class must be over the same vocabulary, and the class must be closed under isomorphism. The structures considered in this paper will be used to describe finite strings and similar objects, such as pairs of strings, or strings of pairs of strings.

Intuitive description. We begin with an intuitive description of string-to-string MSO intepretations. Following the classical Büchi-Elgot-Trakhtenbrot correspondence of automata and MSO logic, we view strings as structures.

Definition 2.1. *A string in Σ^* is viewed as a structure whose universe is the string positions, equipped with the relations*

$$\underbrace{x \le y}_{\text{order on positions}} \qquad \underbrace{a(x)}_{x \text{ has label } a \in \Sigma}.$$

A string-to-string MSO interpretation transforms strings using the above representation, such that the positions of the output string are represented by k-tuples of positions in the input string, for some $k \in \{0, 1, \dots\}$. The order[2] on output positions is defined by a formula

$$\varphi(\underbrace{x_1, \dots, x_k}_{\text{first output position}}, \underbrace{y_1, \dots, y_k}_{\text{second output position}})$$

with $2k$ free variables, while the labels of the output positions are defined by formulas with k free variables, one for each letter in the output alphabet. Finally, not all k-tuples of input positions need to participate in the output string; there is a formula with k free variables, called the *universe* formula, which selects those that do. All of these formulas need to be consistent – every k-tuple of positions in the input string that satisfies the universe formula must satisfy exactly one of the label formulas, and these k-tuples need to be linearly ordered by the order formula. Consistency is decidable, since it boils down to checking if some MSO formula is true in all strings, which in turn boils down to checking if automaton is nonempty by the equivalence of MSO and regular languages.

Formal definition. We now give a formal definition of MSO interpretations. The formal definition generalizes the above intuitive description in two ways of minor importance. First, the definition is presented not just for strings, but for general classes of structures; we intend to apply it to mild generalizations of strings, such as pairs of strings or strings of strings. Second, instead of the universe being k-tuples of some fixed dimension, it is created using a *polynomial functor*, which is an operation on sets of the form

$$F(A) = A^{k_1} + \cdots + A^{k_n}. \tag{1}$$

Typical polynomial functors include the identity functor A, or the functor $A^2 + A^2$ that produces two copies of the square of the input set. We use the following terminology for polynomial functors: each A^{k_i} is called a *component* of the polynomial functor, and $k_i \in \{0, 1, \dots\}$ is called the *dimension* of this component. This extra generality of polynomial functors[3] makes the definition more robust, it will be useful in a more refined analysis of MSO interpretations that will appear in Section 5.3. In case of linear functors (where all components have dimension at most one), the components correspond to the *copies* in an MSO transduction [13, p. 230].

In an MSO interpretation, the polynomial functor is used to define the universe of the output structure; if A is an input structure then elements of $F(A)$ are called *output candidates*. A subset of the output candidates will be the universe of the output structure. This subset is defined using an MSO *query of type F*, which is a family of MSO formulas, with one formula for each component in the functor, such that number of free variables in each formula is the dimension of the corresponding component. Here are some examples:

$$\underbrace{A^0 = 1}_{\substack{\text{a query of this type}\\ \text{is a formula without}\\ \text{free variables}}} \qquad \underbrace{A^4}_{\substack{\text{a query of this type}\\ \text{is a formula with}\\ \text{four free variables}}} \qquad \underbrace{A^2 + A^2}_{\substack{\text{a query of this type}\\ \text{is two formulas with}\\ \text{two free variables each}}}$$

The relations in the output structure are also defined using MSO queries, with a relation of arity m defined using a query of type

$$F^m(A) \overset{\text{def}}{=} \underbrace{F(A) \times \cdots \times F(A)}_{m \text{ times}}$$

The above type is also a polynomial functor, since polynomial functors are closed under taking products, e.g. the product of A^2 and $A + 1$ is $A^3 + A^2$. The discussion above is summarized in the following definition.

Definition 2.2 (MSO interpretation)**.** *A function $f : \Sigma \to \Gamma$ between two classes of structures is called an* MSO *interpretation if:*

1. ***Universe.*** *There is a polynomial functor F and a* MSO *query of type F such that for every input structure $A \in \Sigma$, the universe of the output structure is the subset of the output candidates $F(A)$ defined by this query; and*
2. ***Relations.*** *For every relation name R in the vocabulary of the output class, of arity m, there is an* MSO *query of type F^m, which defines the interpretation of R in every output structure.*

[2]For reasons described in [9, Theorem 4], the string positions are equipped with a linear order $x \geq y$ instead of successor $x = y + 1$.

[3]One can reduce the polynomial functor in an MSO interpretation to a single component A^k, at the cost of increasing the dimension k. This works for input structures with at least two elements. For this reason, [9] uses interpretations with just one component.

A *string-to-string* MSO *interpretation* is the special case of the above definition where the input type is Σ^* for some finite alphabet Σ, and the output type is Γ^* for some finite alphabet Γ.

Example 1. Consider the squaring operation on strings

$$[1,2,3] \mapsto [1,2,3,1,2,3].$$

Suppose that the input alphabet is Σ. This function is defined by an MSO interpretation as follows. The functor F is A^2, and the universe formula is "true", which means that the positions of the output string are all pairs of positions in the input string. The order formula describes the lexicographic order on A^2. Finally, the label of an output position is inherited from the input position on the second coordinate. □

2.2 String types

We are ultimately interested in functions that input and output strings over a finite alphabet. However, to create such functions using primes and combinators, it will be convenient to have more structured types for the simpler functions, such as pairs of strings. The idea to use such structured types comes from [7], in particular we use the same types, as described in the following definition.

Definition 2.3 (List types)**.** *A* list type *is any type constructed using the constructors*

$$\underbrace{1}_{\text{a type with one element}} \quad \underbrace{\Sigma_1 \times \Sigma_2}_{\text{pairs}} \quad \underbrace{\Sigma_1 + \Sigma_2}_{\text{co-pairs, i.e. disjoint union}} \quad \underbrace{\Sigma^*}_{\text{lists}}.$$

An example of a list type is

$$(1+1+1)^*.$$

This type can be seen as the type of strings over a three letter alphabet; in this way the list types generalize strings over finite alphabets. The generalization is minor, since elements of a list type can be seen as strings over a finite alphabet, which uses brackets and commas as in the following example:

$$\underbrace{([\texttt{left}\ 1, \texttt{right}\ 1, \texttt{left}\ 1], 1)}_{\text{an element of the list type } (1+1)^* \times 1}.$$

Structures for list types. We will be interested in MSO interpretations that transform one list type into another. We could simply represent list types as strings over a finite alphabet in the way described above, and then use MSO interpretations on strings over a finite alphabet. The resulting definition would be equivalent to the one that we will use in the paper. However, we choose to use a direct representation of list types as structures, without passing through a string encoding. The reason is that quantifiers would be needed to go between list types and their string encodings, and in this paper, we will be particularly interested in quantifier-free interpretations.

Definition 2.4. *To each list type we associate a class of structures, which is defined by induction as follows.*

(1) *The class 1 contains only one structure; this structure has one element in its universe and no relations.*

(+) *The vocabulary of the class $\Sigma_1 + \Sigma_2$ is the disjoint union of the vocabularies of the classes Σ_1 and Σ_2, plus one new nullary relation name (i.e. arity zero). A structure in this class is obtained by taking a structure in either of the classes Σ_1 or Σ_2, extending the vocabulary to the vocabulary of the other class by using empty sets, and interpreting the new nullary relation as "true" or "false" depending on whether the structure is from Σ_1 or Σ_2.*

(×) *The vocabulary of the class $\Sigma_1 \times \Sigma_2$ is the disjoint union of the vocabularies of the class Σ_1 and Σ_2, plus one new unary relation name (i.e. arity one). A structure in this class is obtained by taking the disjoint union (defined in the natural way) of two structures, one from Σ_1 and one from Σ_2, and using the new unary relation name to select the elements from the first structure.*

(∗) *The general idea is that a structure in the class Σ^* is obtained by taking a list $[A_1, \ldots, A_n]$ of nonempty[4] structures in Σ, creating a new structure using disjoint union (with a shared vocabulary), and adding a new binary relation $x \le y$ which holds whenever the structure containing x appears earlier in the list (or in the same place) than the structure containing y. The problem with this construction is that it would mix nullary relations that come from different structures in the list. To fix this problem, each nullary relation name $R()$ in the vocabulary of Σ is changed into a unary relation name $R(x)$ that selects elements x such that the corresponding structure satisfies $R()$.*

If we apply the above representation to a list type

$$(\underbrace{1+\cdots+1}_{n \text{ times}})^*$$

then we get the representation of strings as ordered structures from Definition 2.1, with the exception that the empty string has a universe with one element. Therefore, it is not important if we use Definition 2.1 or 2.4 for representing strings.

Definition 2.5. *A* polyregular function *is a function*

$$f : \Sigma \to \Gamma$$

[4]A structure is nonempty if its universe is nonempty. This leads to the following subtle point, which arises when considering lists of lists, and related structures. Since a list can be empty, it follows that we do not allow lists of empty lists such as $[[],[],[]]$. This means that the list constructor, as it is used in this paper and formalized in Definition 2.4, should be interpreted as possibly empty lists with nonempty list items. This distinction will not play a role for types such as $(1+1)^*$ where list elements cannot be empty, which is the case that we really care about.

between list types that can be defined by an MSO *interpretation, assuming that list types are viewed as classes of structures according to Definition 2.4.*

The original definition of polyregular functions [5] did not use MSO interpretations, however MSO interpretations were shown equivalent to the original definition in [9, Theorem 7]. Since the original definition was closed under composition, it follows that MSO interpretations are closed under composition (as long as the input and output classes are list types).

3 The fold combinator

In this section, we discuss dangers of the fold combinator

$$\frac{1 \to \Gamma \quad \Gamma \times \Sigma \to \Gamma}{\Sigma^* \to \Gamma} \quad \text{fold}$$

We also explain how some of the dangers can be avoided by using quantifier-free interpretations.

We begin this section with several examples illustrating the usefulness of fold.

Example 2. Consider a finite automaton with a n states and an input alphabet of m letters. Assuming some order on the states and alphabet, the transition function can be seen as a function between finite string types

$$\underbrace{(1+\cdots+1)}_{n \text{ times}} \times \underbrace{(1+\cdots+1)}_{m \text{ times}} \to \underbrace{1+\cdots+1}_{n \text{ times}}.$$

If we apply fold to this automaton, under some chosen initial state, then we get the function that inputs a string, and returns the last state in the run. A special case of this construction is when both the states and input letters of the automaton are elements of some finite group G, the initial state is the group identity, and the transition function is the group operation. By folding this transition function, we get the *group multiplication* function of type $G^* \to G$, which is one of the (less appealing) prime functions in the combinatory calculus from [5]. □

Example 3. There are two symmetric list constructors

$$\underbrace{1+\Sigma^* \times \Sigma \to \Sigma^*}_{\substack{\text{lists are constructed by adding} \\ \text{letters to the right of the list}}} \qquad \underbrace{1+\Sigma \times \Sigma^* \to \Sigma^*}_{\substack{\text{lists are constructed by adding} \\ \text{letters to the left of the list}}}.$$

If we apply fold to the two corresponding automata, then we get the reverse and identity functions on lists, respectively. The fold combinator corresponds in a canonical way to the first list constructor, which is why it is sometimes called *fold right.* □

3.1 On the dangers of folding

We now present two examples which show how the fold combinator, without any further restrictions, can define functions that are not polyregular. More generally, one can use fold to derive any primitive recursive function [18, Section 4.1].

Example 4. [Iterating duplication] Consider an automaton where the input alphabet is 1, and the states are 1^*. We view the states as natural numbers, with the list 1^n of length n representing the number n. The initial state in this automaton is 1, and the transition function is

$$(1^n, 1) \in 1^* \times 1 \quad \mapsto \quad 1^{2n} \in 1^*.$$

This is an example of a polyregular function, in fact it is a linear regular function. However, if we apply fold to it, then we get the function

$$1^n \in 1^* \quad \mapsto \quad 1^{2^n} \in 1^*.$$

which is not polyregular because of exponential growth. □

Example 5. [Subtraction] As illustrated in Example 4, we run into trouble if iterate duplication. But we can also run into trouble when the transition function does not create any new elements. Consider an automaton where the input alphabet is $1+1$, and the state space is the integers, represented as the list type

$$\underbrace{1^*}_{\substack{\text{represents} \\ \{-1,-2,\ldots\}}} + \underbrace{1^*}_{\substack{\text{represents} \\ \{0,1,\ldots\}}}$$

The initial state is zero, and the transition function increments or decrements the state depending on which of the two input letters from $1+1$ it gets. This transition function is easily seen to be polyregular, and it has the property that the output size is at most the input size, assuming that the input letter contributes to the input size. However, by folding this automaton, we get a function that subsumes integer subtraction and is therefore not polyrergular. Using similar ideas, one could simulate two-counter machines. □

3.2 Quantifier-free interpretations and their folding

As the two above examples show, we have to be careful when applying fold. Clearly we must avoid duplication (Example 4). This can be done by requiring the polynomial functor in the interpretation to be the identity, thus ensuring that the output is no larger than the input. It is less clear how to avoid the problem with Example 5. Our solution is to use quantifier-free interpretations, as defined below.

Definition 3.1. *A* quantifier-free interpretation *is the special case of* MSO *interpretations where the polynomial functor is the identity $F(A) = A$ and all formulas are quantifier-free.*

One could consider interpretations in which the formulas are quantifier-free, but the functor is not necessarily the identity; such interpretations will not be useful in this paper.

The transition function in Example 5 is not quanitifier-free, since decrementing a number, which corresponds to removing a list element, is not a quanitifier-free operation. The following theorem is the first main contribution of this paper: fold can be safely applied to quantifier-free interpretations.

Theorem 3.2. *Let Σ and Γ be any classes of structures, not necessarily list types. If the transition function*

$$\delta : \Gamma \times \Sigma \to \Gamma$$

in the assumption of the fold combinator is a quantifier-free interpretation, then the function in the conclusion is a linear MSO *interpretation.*

Proof

Consider an automaton as in the assumption of the theorem. For an input to this automaton $[A_1, \ldots, A_n]$, and $i \in \{0, \ldots, n\}$ we write $B_i \in \Gamma$ for the state of the automaton after reading the first i input letters. The state B_0 is the initial state, which is given by the assumption to the fold combinator, and the state B_n is the last state, which is the output of the function in the conclusion of the fold combinator. Our goal is to compute the last state using a linear MSO interpretation.

Since the functor in δ is the identity, the output candidates are simply the elements of the input structure. Therefore, the universe of B_n is contained in the disjoint union of the universe of B_{n-1} and the universe of A_n. By unfolding the induction, the universe of B_n is contained in the universe of the first state B_0 and the input structure $A = [A_1, \ldots, A_n]$. Therefore, to prove that the fold is an MSO interpretation, it will be enough to show that an MSO formula can tell us: (a) which elements of $B_0 + A$ belong to the output structure; and (b) which relations of the output structure are satisfied by which tuples from $B_0 + A$. The answers to these questions will be contained in the quantifier-free theory of the tuple, as defined below.

Definition 3.3. *Let A be a structure and let $\bar{a}$ be a list of distinguished elements, which need not belong to the universe of A. The* quantifier-free theory *of a $\bar{a}$ in A is the following information: which distinguished elements are in the universe, and which quantifier-free formulas are satisfied by those distinguished elements that are in the universe.*

Using the above terminology, to prove that the fold is definable in MSO, we need to show that for each tuple in $B_0 + A$, we can define in MSO the corresponding quantifier-free theory in the output structure B_0. This will be done in the following claim. The key property used by the claim is the following *continuity property* of quantifier-free interpretations: the quantifier-free theory of a tuple of output candidates in the output structure is uniquely determined by the quantifier-free theory of the same tuple in the input structure.

In the following claim, we consider a function which inputs structures with tuples of k distinguished elements, and has finitely many possible output values (quanitifier-free theories, in the case of the claim). Such a function is called MSO definable if for every chosen output value, there is an MSO formula with k free variables that selects inputs which give chosen output.

Claim 3.4. *For every $k \in \{1, 2, \ldots\}$ and every tuple $\bar{b}$ of elements in B_0, the following function is* MSO *definable:*

- **Input.** *A structure $A \in \Sigma^*$ with elements $\bar{a} \in A^k$.*
- **Output.** *The quantifier-free theory of $\bar{a}\bar{b}$ in B_n.*

Proof

By the continuity property mentioned earlier in this proof, the quantifier-free theory of $\bar{a}\bar{b}$ in B_n is uniquely determined by the quantifier-free theory of $\bar{a}\bar{b}$ in the structure (B_{n-1}, A_n), which in turn is uniquely determined (by compositionality) by the quantifier-free theories of $\bar{a}\bar{b}$ in the two individual structures B_{n-1} and A_n. Therefore, we can think of these quantifier-free theories as being computed by a finite automaton, where the initial state is the quantifier-free theory of $\bar{b}$ in B_0, and the input string is

$$[\text{qf theory of } \bar{a} \text{ in } A_1, \ldots, \text{qf theory of } \bar{a} \text{ in } A_n].$$

By the continuity property, one can design a transition function for this automaton, which does not depend on the input structure A or the tuple $\bar{a}$, such that its state after reading the first i letters is the quantifier-free theory of $\bar{a}\bar{b}$ in B_i. The state space of this automaton is finite, since there are finitely may quantifier-free theories once the vocabulary and number of arguments have been fixed. Since finite automata can be simulated in MSO, it follows that the last state in the run of this automaton, which is the theory in the conclusion of the claim, can be defined in MSO. □

We now use the claim to complete the proof of the lemma. The output candidates of the MSO interpretation are defined by the polynomial functor

$$F(A) = A + \underbrace{1 + \cdots + 1}_{\text{size of initial state } B_0}.$$

In other words, the output candidates are elements of the input list and the initial state. By the above claim, the quantifier-free theory of a single output candidate in the output structure can be defined in MSO, and since this theory tells us if the output candidate is present in the universe output structure, we can use it to define the universe. Similarly, if we want to know if a tuple of output candidates satisfies some relation from the output vocabulary, then we can find this information using MSO as in the above claim. □

On its own, the theorem above does not solve all of the problems with fold. One issue is that the theorem only supports one application of fold, since the folded function is no longer quantifier-free and cannot be folded again. Another issue is that applying the theorem stays within the class of functions that do not increase the output size, while we will also be interested in folding functions that increase the size.

These problems will be addressed later in the paper, by developing a suitable type system. Before continuing, we give some applications of the theorem.

Example 6. Consider a transition function of a finite automaton as in Example 2. In a list type of the form $1 + \cdots + 1$, the component of the disjoint union that is used can be accessed by a quantifier-free formula without free variables, since it is represented using nullary relations. Therefore, the transition function is a quantifier-free interpretation, and so we can apply Theorem 3.2 to conclude that the fold is an MSO transduction. This corresponds to the inclusion

$$\text{regular languages} \quad \subseteq \quad \text{MSO}.$$

Applying Theorem 3.2 to prove this inclusion is not the right way to prove it, since the inclusion itself is used in the proof of the theorem. □

In Example 6, we applied the fold combinator to a finite automaton. In the following example, we give a more interesting application, where the state space is infinite.

Example 7. [Streaming string transducers] Define a *simple streaming string transducer*, simple SST for short, as follows. It has two finite alphabets Σ and Γ, called the *input* and *output* alphabets. It has a *configuration space*, which is a list type of the form

$$\Delta = (\Gamma^*)^{k_1} + \cdots + (\Gamma^*)^{k_m}.$$

In other words, the set of configurations is obtained by applying some polynomial functor to the set of strings over the output alphabet. The idea is that a configuration consists of a state, which is one of the m components, and a register valuation which is a tuple of strings over the output alphabet. The configurations of the transducer are updated according to the following three functions, which are required to be quantifier-free, according the the representation of the input and output alphabets that was used in Example 6:

$$\underbrace{1 \to \Delta}_{\text{initial}} \qquad \underbrace{\Delta \times \Sigma \to \Delta}_{\text{transition function}} \qquad \underbrace{\Delta \to \Gamma^*}_{\text{final}}.$$

The semantics of the transducer is the function of type $\Sigma^* \to \Gamma^*$ that is obtained by folding the first two functions, and post-composing with the final function. By Theorem 3.2, this function is an MSO transduction.

The model described above subsumes (and in fact, is equivalent to) the classical model of SST [1, Section 3], with the only difference (which is why we call our model simple) being that our model allows the input letter to be used only once (as opposed to a constant number of times) in the registers. This is because string concatenation, which is the operation used to update registers in an SST, is a quantifier-free operation. Therefore, Theorem 3.2 can be seen as subsuming the implication

$$\text{copyless sst} \subseteq \text{deterministic MSO transductions}$$

proved in [1, Theorem 3]. The same idea will work for trees, as we will see in Section 6.1. □

Example 8. [Graphs] As mentioned in Theorem 3.2, the folded automaton need not operate on classes that are list types. For instance, we could adapt Example 7 to transducers in which the registers, instead of storing strings, store graphs with k distinguished vertices, as in Courcelle's algebras for treewidth [12, Section 1.4]. We could still apply Theorem 3.2, since the corresponding operations on graphs are quantifier-free. Similar ideas would also work for cliquewidth. □

4 Deriving quantifier-free functions

As we have shown in Theorem 3.2, the fold combinator can be safely applied to quantifier-free interpretations. Before discussing the fold combinator, we take a minor detour in this section, and present a complete system for the quantifier-free interpretations.

A few examples. We begin with examples and non-examples of quantifier-free interpretations operating on list types.

Example 9. [Commutativity of product] Consider the function of type

$$\Sigma_1 \times \Sigma_2 \to \Sigma_2 \times \Sigma_1,$$

which swaps the order in a pair. Like all examples in this section, this is actually an infinite family of functions, one for every choice of Σ_1 and Σ_2. The function is a quantifier-free interpretation. The only change between the input and output concerns the unary relation from the definition of the product class $\Sigma_1 \times \Sigma_2$ which tells us if an element is from the first coordinate; this relation needs to be complemented. □

Example 10. [List reverse and concatenation] Consider the list reverse function of type $\Sigma^* \to \Sigma^*$. This is clearly a quantifier-free interpretation – it is enough to replace the order $x \leq y$ with its reverse $y \leq x$. A similar idea works for the list concatenation function of type $\Sigma^{**} \to \Sigma^*$ which concatenates a list of lists into a list. In the input structure, there are two linear orders, corresponding to the inner and outer lists. To get the output structure, we use the lexicographic product of these two orders, which can be defined in a quantifier-free way. □

Example 11. [List constructor and destructor] Consider the (left) list constructor

$$1 + \Sigma \times \Sigma^* \to \Sigma^*,$$

that was discussed in Example 3. This is a quantifier-free interpretation. If the input is from 1, which can be tested in a quantifier-free way using the nullary relation from the co-product, then the output list is created in the natural way. Otherwise, if input is a pair from $\Sigma \times \Sigma^*$, then the order on the concatenated list can easily be defined by using the unary predicate that identifies the first argument of a pair.

$\Gamma \times \Sigma \leftrightarrow \Sigma \times \Gamma$	commutativity of $\times$
$\Gamma + \Sigma \leftrightarrow \Sigma + \Gamma$	commutativity of $+$
$\Gamma \times (\Sigma \times \Delta) \leftrightarrow (\Gamma \times \Sigma) \times \Delta$	associativity of $\times$
$\Gamma + (\Sigma + \Delta) \leftrightarrow (\Gamma + \Sigma) + \Delta$	associativity of $+$
$\Gamma \times (\Sigma + \Delta) \leftrightarrow (\Gamma \times \Sigma) + (\Gamma \times \Delta)$	distributivity
$\Gamma_1 \times \Gamma_2 \to \Gamma_i$	projections
$\Gamma_i \to \Gamma_1 + \Gamma_2$	co-projections
$\Gamma + \Gamma \to \Gamma$	co-diagonal
$\Sigma^* \times \Sigma \to \Sigma^*$	append
$\Sigma^* \to \Sigma^*$	reverse
$\Sigma^{**} \to \Sigma^*$	concat
$\Sigma \to \Sigma \times \Gamma^*$	create empty
$(\Sigma \times \Gamma)^* \to \Sigma^* \times \Gamma^*$	list distribute

Figure 1. The prime quantifier-free functions.

$\dfrac{\Gamma_1 \to \Sigma_1 \quad \Gamma_2 \to \Sigma_2}{\Gamma_1 \times \Gamma_2 \to \Sigma_1 \times \Sigma_2}$	functoriality of $\times$
$\dfrac{\Gamma_1 \to \Sigma_1 \quad \Gamma_2 \to \Sigma_2}{\Gamma_1 + \Gamma_2 \to \Sigma_1 + \Sigma_2}$	functoriality of $+$
$\dfrac{\Gamma \to \Sigma}{\Gamma^* \to \Sigma^*}$	functoriality of $*$
$\dfrac{\Gamma \to \Sigma \quad \Sigma \to \Delta}{\Gamma \to \Delta}$	function composition

Figure 2. The quantifier-free combinators.

The list constructor is bijective, and therefore it has a corresponding inverse of type

$$\Sigma^* \to 1 + \Sigma \times \Sigma^*,$$

which we call the *list destructor.* The list destructor is not a quantifier-free interpretation. The reason is that if the input is an nonempty list, then we would need to isolate in a quantifier-free way the elements from the head, i.e. from the first list element, which cannot be done. □

Example 12. [Diagonal] Another non-example is $x \mapsto (x, x)$. This is not a quantifier-free interpretation, since the output size is bigger than the input size. □

A complete system. We now present a complete characterization of quantifier-free interpretations on list types. The system will be used as a basis for the system in the next section, which will describe general MSO interpretations.

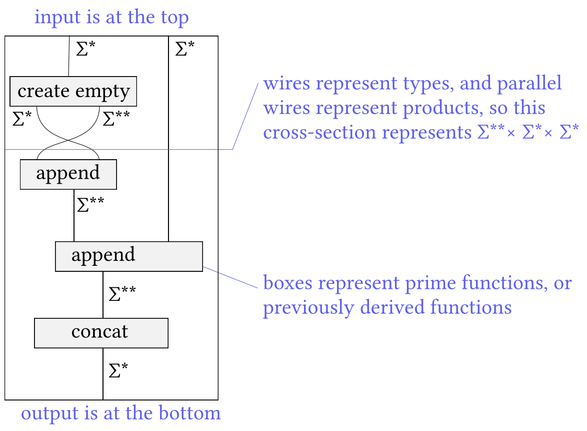

input is at the top
Σ^*
Σ^*
create empty
Σ^*
Σ^{**}
wires represent types, and parallel wires represent products, so this cross-section represents $\Sigma^{**} \times \Sigma^* \times \Sigma^*$
append
Σ^{**}
append
boxes represent prime functions, or previously derived functions
Σ^{**}
concat
Σ^*
output is at the bottom

Figure 3. A string diagram that derives the binary operation of type $\Sigma^* \times \Sigma^* \to \Sigma^*$ for list concatenation.

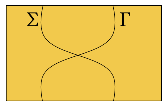
Theorem 4.1. *The quantifier-free interpretations between list types are exactly those that can be derived from the prime functions in Figure 1 by applying the combinators from Figure 2.*

The proof of the above theorem, with completeness being the non-trivial part, is in the appendix.

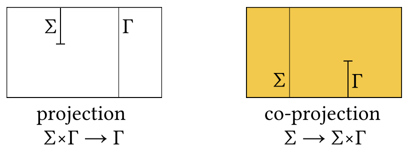
4.1 String diagrams

We conclude this section with several example derivations of quantifier-free functions using the system from Theorem 3.2. To present these derivations, we use string[5] diagrams based on [11, Chapter 3], as depicted in Figure 3.

We also use string diagrams with a yellow background, where parallel wires represent co-products. For example, the following diagram represents the prime function from Figure 1 that describes commutativity of +:

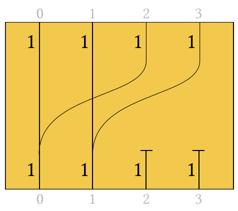


Here are two other examples of string diagrams, which use dead ends, and represent projections and co-projections:


Σ
Γ
projection
$\Sigma\times\Gamma \longrightarrow \Gamma$
Σ
Γ
co-projection
$\Sigma \longrightarrow \Sigma\times\Gamma$

Example 13. Recall the representation of finite sets as list types $1 + \cdots + 1$ used in Examples 2 and 6. Under this representation, every function between finite sets is derivable using the prime functions and combinators of Theorem 3.2. This is easily seen using string diagrams, as illustrated below:

[5]This is a name clash: the word "string" relates to the shape of the diagrams, and not to the fact that they manipulate types that represent strings.



The representation of finite sets as co-products is important here. For example, the diagonal function $1 \to 1 \times 1$ is not derivable, as explained in Example 12. □

5 Deriving polyregular functions

We now move beyond quantifier-free functions and present the main contribution of this paper, which is a system that derives exactly the polyregular functions. As explained in Example 5, we cannot simply add the fold combinator to the system from Theorem 3.2. Another idea would be to have two kinds of functions: quantifier-free functions, and general polyregular functions, with the fold combinator used to go from one kind to the other. In such a system, the only contribution of fold would be to define linear regular functions, since such are the functions in the conclusion of Theorem 3.2. We are more ambitious, and we want the fold combinator to be useful also for non-linear functions.

To define a system with fold, we add a new unary type constructor. This type constructor is denoted by ! and it is written on the left. The general idea is that an element $!x$ is essentially the same element as x, except that it is harder to obtain. The type constructor is not idempotent, and so $!!x$ is even harder to obtain than $!x$. The goal of this type constructor is to restrict the application of fold in a way that avoids the problems discussed in Section 3.1. This is done by using the following *safe fold* combinator:

$$\frac{!^k 1 \to \Gamma \quad \Gamma \times \Sigma \to \Gamma}{!^k(\Sigma^*) \to \Gamma} \qquad \text{safe fold}$$

In the combinator, $!^k$ refers to k-fold application of !. When applying the combinator, the number $k \in \{0, 1, \ldots\}$ must be strictly bigger than the grade of Γ, which is defined to be the maximal nesting of !, as in the following examples:

$$\underbrace{1^*}_{\text{grade zero}} \qquad \underbrace{1+!(1+!1)}_{\text{grade two}}.$$

For example, when Γ has grade zero, i.e. it does not use !, then safe fold can be used in the form

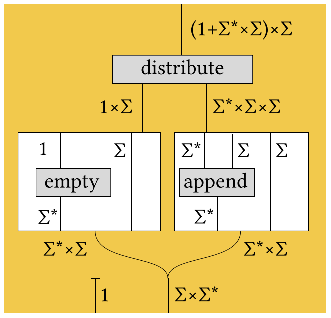
$$\frac{!1 \to \Gamma \quad \Gamma \times \Sigma \to \Gamma}{!(\Sigma^*) \to \Gamma} \qquad \text{safe fold when } \Gamma \text{ is without !}$$

The general idea is that the annotation with ! will disallow certain kinds of repeated applications of fold that would lead to functions that are not polyregular. Before giving a formal description of the system, we begin with an example.

Example 14. [List destructor] In this example, we use safe fold to derive a variant of the list destructor

$$\Sigma^* \to 1 + \Sigma^* \times \Sigma$$

that was discussed in Example 11. Consider an automaton where the state space is the output type of the list destructor, the initial state is 1, and the transition function is



By applying the safe fold to this automaton, we get the list deconstructor in a weaker type, namely

$$!(\Sigma^*) \to 1 + \Sigma^* \times \Sigma.$$

The weaker type avoids the issues from Example 5, since the input and output will have different numbers of !, and therefore we will be unable to apply fold again. □

5.1 Graded types and their derivable functions

We now give a formal description of the system. The type system is the same as previously, except that we have one more type constructor for !.

Definition 5.1. *A* graded list type *is any type that is constructed using the following type constructors*

$$\underbrace{1}_{\textit{a type with one element}} \qquad \underbrace{\Sigma_1 \times \Sigma_2}_{\textit{pair}} \qquad \underbrace{\Sigma_1 + \Sigma_2}_{\textit{co-pair, i.e. disjoint union}} \qquad \underbrace{\Sigma^*}_{\textit{lists}} \qquad !\Sigma.$$

The general idea is that ! does not change the underlying set, but only introduces some type annotation that controls the way fold and duplication can be applied. Apart from safe fold, the main way of dealing with ! is the duplicating operation

$$!\Sigma \to !\Sigma \times \Sigma \qquad \text{absorption},$$

which is named after the same rule in the parsimonious calculus of Mazza [20, p.1]. There are also prime functions for commuting ! with the remaining type constructors, for example $![x, y, z]$ and $[!x, !y, !z]$ are going to be equivalent in our system; for this reason we can write $!\Sigma^*$ without specifying the order in which the two constructors are applied.

Definition 5.2. *There are two kinds of derivability for functions between graded list types.*

1. ***Strongly derivable.*** *A function is called* strongly derivable *if it can be derived using quantifier-free prime functions and combinators from Figures 1 and 2, extended*

to graded list types that can use !, along with four new prime functions

$!(\Gamma + \Sigma) \leftrightarrow !\Gamma + !\Sigma$	*! commutes with +*
$!(\Gamma \times \Sigma) \leftrightarrow !\Gamma \times !\Sigma$	*! commutes with ×*
$(!\Gamma)^+ \leftrightarrow !(\Gamma^+)$	*! commutes with ∗*
$!\Gamma \to !\Gamma \times \Gamma$	*absorption*

and two new combinators

$\dfrac{\Sigma \to \Gamma}{!\Sigma \to !\Gamma}$	*functoriality of !*
$\dfrac{!^k 1 \to \Gamma \quad \Gamma \times \Sigma \to \Gamma}{!^k(\Sigma^*) \to \Gamma}$	*safe fold*

The safe fold combinator can only be applied when Γ has grade $< k$.

2. ***Weakly derivable.*** *A function is called* weakly derivable *if it is of the form $x \mapsto !^k f(x)$ for some k and some strongly derivable function f.*

In other words, a function is weakly derivable if it can be strongly derived for a sufficiently upgraded input type. For example, the list destructor of type

$$\Sigma^* \to 1 + \Sigma^* \times \Sigma$$

function is not strongly derivable (Example 11), but it is weakly derivable (Example 14).

In the following theorem, which is the main result of this paper, we are only interested in weak derivability for functions between (ungraded) string types, i.e. between types that do not use !. The purpose of ! is to get the strong derivations.

Theorem 5.3. *A function between (ungraded) list types is polyregular if and only if it is weakly derivable.*

The proof has two parts: soundness and completeness.

5.2 Completeness

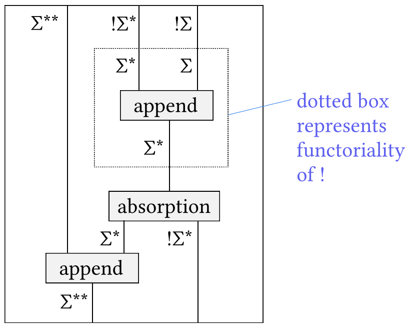
The completeness part of Theorem 5.3 is that every polyregular function can be weakly derived. Unlike the quantifier-free system in Theorem 4.1, completeness is relatively easy. This is because fold is a powerful combinator, and we can draw on a prior complete system for the polyregular functions [5, p. 64]. In the completeness proof, the polynomial growth output size will come from a single quadratic function.

Claim 5.4. *One can weakly derive the following function*

$$\underbrace{[a_1, \ldots, a_n] \quad \mapsto \quad [[a_n, \ldots, a_1], [a_{n-1}, \ldots, a_1], \ldots, [a_1]]}_{\textit{call this the } \text{prefixes } \textit{function}}.$$

Proof

Consider an automaton, where the input alphabet is $!\Sigma$, the state space is $\Sigma^{**} \times !\Sigma^*$, the initial state is the pair of empty lists, and the transition function is



By applying fold to this automaton, we get a function of type

$$!!\Sigma^* \to \Sigma^{**} \times !\Sigma^*$$

which returns the output of the prefixes function on the first output coordinate. Observe that in this proof, we applied the fold to a transition function that already uses !. □

Using the above function, in the appendix we show that the weakly derivable functions contain an already existing complete system for the polyregular functions [5, p. 64].

Before discussing the soundness proof in the theorem, let us comment on the minimality of its system. The system inherits all of the primes and combinators from the quantifier-free system in Theorem 4.1. In the presence of fold, some of these primes and combinators can be derived thus leading to a smaller system.

Theorem 5.5. *The system from Theorem 5.3 remains complete after removing the map combinator, as well as all prime functions and combinators that involve the list type, and adding*

$1 + \Sigma \to \Sigma^*$	*lists of length at most one*
$\Sigma^* \times \Sigma^* \to \Sigma^*$	*binary list concatenation.*

5.3 Soundness

The rest of this section is devoted to the proof of soundness for Theorem 5.3, which is that all weakly derivable functions are polyregular. We will define an invariant on strongly derivable functions, which is satisfied by the prime functions, is preserved by the combinators, and which implies that a function is polyregular. This invariant can be seen as giving a semantic explanation of the ! constructor and the strongly derivable functions.

The invariant uses a more refined notion of MSO interpretations, called *graded* MSO *interpretations*. These interpretations operate on graded structures, as described in the following definition.

Definition 5.6 (Graded structure). *A* graded structure *is a structure, together with a* grading function *that assigns to each element in the universe a grade in $\{0, 1, \ldots\}$.*

The idea is that the grade of an element is the number of times that ! has been applied, as in the following example

$$(\underbrace{1}_{\substack{\text{grade}\\\text{zero}}}, \underbrace{![1,1,1]}_{\substack{\text{grade}\\\text{one}}}).$$

A graded list type can be seen as describing a class of graded structures, with the constructor ! incrementing the grade of all elements, and the remaining constructors treated in the same way as in Definition 2.4.

If A is a graded structure, we write $A|\ell$ for the structure that is obtained from A by restricting its universe to elements that have grade at least ℓ. In the definition of a graded MSO interpretation, we use the grades to control how an MSO interpretation f uses quantifiers. The general idea is that $f(A)|\ell$ depends on $A|\ell$ in a quantifier-free way, and on $A|\ell+1$ in an MSO definable way.

Before presenting the formal definition, we introduce some notation, in which a polynomial functor F is applied to a tuple of elements $\bar{a}$, yielding a new (typically longer) tuple of elements $F(\bar{a})$. If an input set A for a polynomial functor F is equipped with some linear order, then this linear order can be extened to a linear order on the output set $F(A)$, by using some fixed order on the components, and ordering tuples lexicographically. This way we can think of a polynomial functor as transforming linearly ordered sets, i.e. lists. We will care about lists of fixed length, which we call tuples. For example if the polynomial functor is $A + A^2$, then applying it to the tuple $(1, 2)$ gives the tuple

$$(1, 2, 1, 2, (1, 1), (1, 2), (2, 1), (2, 2)) \in F(\{1, 2\})^6.$$

In the definition below, we will care about the theories of tuples of the form $F(\bar{a})$, with the theories defined as in Definition 3.3, but extended to MSO formulas of given quantifier rank (the quantifier rank of an MSO formula is the nesting depth of the quantifiers, with first-order and second-order quantifiers counted in the same way). Recall that these theories allow for distinguished elements that are not part of the universe in a structure. Equipped with this notation, we are ready define the graded version of MSO interpretations.

Definition 5.7. *A function $f : \Sigma \to \Gamma$ is called a* graded MSO interpretation *if there is some polynomial functor*

$$F(A) \quad = \quad \underbrace{A}_{\substack{\textit{this is called the}\\\text{quantifier-free}\\\textit{component}}} \quad + \quad \underbrace{F_0(A) + \cdots + F_m(A)}_{\substack{\textit{components from this part}\\\textit{of the functor are called the}\\\text{downgrading }\textit{components}}}$$

such that the following conditions hold:

1. **Universe and grades.** *The universe of the output structure is contained in*

$$A + F_0(A|1) + F_1(A|2) + \cdots + F_m(A|m+1).$$

The grades in the output structure are defined as follows: elements from F_ℓ have grade ℓ, and elements from the quantifier-free component inherit their grade from A.

2. **Continuity.** *For every $k, \ell \in \{0, 1, \ldots\}$ there is some quantifier rank $r \in \{0, 1, \ldots\}$ such that for every input structure A and distinguished elements $\bar{a} \in A^k$, the quantifier-free theory of the tuple $F(\bar{a})$ in $f(A)|\ell$ is uniquely determined by the following two theories:*
   a. *the quantifier-free theory of $\bar{a}$ in $A|\ell$;*
   b. *the rank r MSO theory of $\bar{a}$ in $A|\ell + 1$.*

If we ignore the grades, then a graded MSO interpretation is a special case of an MSO interpretation. This is because the quantifier-free type mentioned in the continuity condition will tell us which output candidates from $F(A)$ are in the universe of the output structure, and how the relations of the output structure are defined on them. Therefore, the continuity condition tells us that the output not only can be defined in MSO, but it can be defined in a way that respects the grades. In particular, in the special case when all input elements have nonzero grade, and all output elements have zero grade, the continuity condition collapses to the usual condition in an MSO interpretation. In this way, graded MSO interpretations generalize ungraded MSO interpretations.

Graded MSO interpretations also generalize quantifier-free interpretations – this happens in the case when all elements in the input and output structures have grade zero. In this case, only the quantifier-free component is useful, and all formulas are quantifier-free.

In the appendix, we show that all strongly derivable prime functions are graded MSO interpretations. This will imply that all weakly derivable functions are ungraded MSO interpretations, since the continuity condition becomes vacuous when the input type is sufficiently upgraded. The proof is an induction on the size of a strong derivation, with the most interesting cases being composition and safe fold. Composition is a corollary of composition closure for MSO interpretations on string types [9, Corollary 8], while safe fold is treated in the same way as in Theorem 3.2.

6 Linear regular functions

The last group of results from this paper concerns the linear regular functions, i.e. polyregular functions of linear growth. We show that a small change to the system from Theorem 5.3 will give exactly the linear regular functions. As we will see, superlinear growth in the system from Theorem 5.3 is not created by the fold combinator, with the culprit instead being

$$!\Gamma \to !\Gamma \times \Gamma \qquad\qquad \text{absorption.}$$

This function allows us to create an unbounded number of copies of an element of Γ, as witnessed in the proof of Claim 5.4. If we simply remove this function, then the system will become too weak, since all other prime functions and combinators preserve the property that the universe of the output structure is contained in the universe of the input structure. The solution is to add a weaker form of absorption

$$!\Gamma \to \Gamma \times \Gamma \qquad \text{linear absorption.}$$

In other words, removing *all* occurrences of ! is the price paid for copying. The corresponding system describes exactly the linear regular functions, as stated in the following theorem.

Theorem 6.1. *A function $f : \Sigma \to \Gamma$ between string types is linear regular if and only if it can be weakly derived in a system that is obtained from the one*[6] *in Theorem 5.3 by replacing absorption with linear absorption.*

The proof for the above theorem, which is in the appendix, is based on Example 7 about streaming string transducers. The idea is that linear absorption together with fold is enough to simulate streaming string transducers, which are expressively complete the linear regular functions.

6.1 Tree types

It turns out that the system for linear regular functions from Theorem 6.1 can be generalized without much further difficulty to trees. This is in contrast to a prior combinator system for trees [8, Theorem 7.1], which had an involved proof using approximately fifty prime functions. We belive that this is evidence for the usefulness of the fold combinator.

Consider a type for trees, defined inductively by

$$\mathsf{T}\Sigma = \underbrace{1 + \mathsf{T}\Sigma \times \Sigma \times \mathsf{T}\Sigma}_{\substack{\text{a tree is either a leaf, or has two}\\ \text{subtrees and a root label}}}$$

A *tree type* is a type that is constructed using the types from Definition 2.3, together with the tree type. Tree types can be seen as structures, using the same construction as for lists in Defintion 2.4, except that instead of one linear order, we have two orders: the *descendant order* (which is not a linear order) and the *document order* given by

$$\text{left subtree} \quad < \quad \text{root} \quad < \quad \text{right subtree}.$$

Define a *linear regular tree function* to be a function between tree types that is defined using linear MSO transductions.

Following Wilke [24], we view trees as an algebra. In this algebra, there is an additional type constructor $\mathsf{C}\Sigma$, which describes *contexts*. A context is a tree with a distinguished leaf (called the *hole*) where other trees can be inserted. This is not a primitive type constructor, only syntactic sugar for a certain combination of the list and tree type constructurs:

$$\mathsf{C}\Sigma \overset{\text{def}}{=} (\underbrace{(\mathsf{T}\Sigma \times \Sigma)}_{\substack{\text{the hole is in}\\ \text{the right subtree}}} + \underbrace{(\Sigma \times \mathsf{T}\Sigma)}_{\substack{\text{the hole is in}\\ \text{the left subtree}}})^*.$$

[6]One can also start with the smaller system from Theorem 5.5.

To operate on trees and contexts, we use the following operations, called *Wilke's operations*,see [24, Figure 1]:

$$\begin{aligned} 1 + \mathsf{T}\Sigma \times \Sigma \times \mathsf{T}\Sigma \to \mathsf{T}\Sigma &\qquad \text{tree constructor}\\ \mathsf{C}\Sigma \times \mathsf{T}\Sigma \to \mathsf{T}\Sigma &\qquad \text{replace hole by a tree}\\ \mathsf{C}\Sigma \times \mathsf{C}\Sigma \to \mathsf{C}\Sigma &\qquad \text{context composition}\\ 1 + (\mathsf{T}\Sigma \times \Sigma) + (\Sigma \times \mathsf{T}\Sigma) \to \mathsf{C}\Sigma &\qquad \text{context creation} \end{aligned}$$

All of these operations are quantifier-free interpretations, and we will use them as primes. The last two operations need not be explicitly added, since they can derived using the system from Theorem 3.2.

Theorem 6.2. *A function $f : \Sigma \to \Gamma$ between tree types is linear regular if and only if it can be derived in a system that is obtained from the system in Theorem 6.1 by adding the tree type, Wilke's operations, the prime function*

$$!\mathsf{T}\Sigma \leftrightarrow \mathsf{T}!\Sigma \qquad \textit{! commutes with } \mathsf{T}$$

and the following combinator

$$\frac{!^k 1 \to \Gamma \quad \Gamma \times \Sigma \times \Gamma \to \Gamma}{!^k \mathsf{T}\Sigma \to \Gamma} \qquad \textit{safe tree fold,}$$

which can be applied whenever Γ has grade $< k$.

Proof (Sketch)
As in Theorem 6.1. We use the same soundness proof, except that tree automata are used instead of string automata. For completeness, we use a result of Alur and D'Antoni, which says that every linear MSO interpretation is computed by a streaming tree transducer [3, Theorem 4.6]. Adjusting for notation, a streaming tree transducer is defined in the same way as in Example 7, except that instead of lists, registers store trees and contexts. The registers in the transducer are manipulated using Wilke's operations; and thus for the same reason as in Example 7, the corresponding tree function is weakly derivable. This completeness proof takes into account only functions of type $\mathsf{T}\Sigma \to \mathsf{T}\Gamma$ where Σ and Γ are finite alphabets, but the extension to other tree types is easily accomplished by encoding tree types into such trees. □

Tree polyregular functions. It is natural to ask about a polyregular system for trees. We conjecture that if we add absorption to the system from Theorem 6.2, and possibly a few extra prime functions, then the system will define exactly the MSO interpretations on tree types. This conjecture would imply that tree-to-tree MSO inprepretations are closed under composition, which is an open problem.

7 Perspectives

We finish the paper with some directions for future work.

In our proofs, we are careless about the number of times that ! is applied. Maybe a more refined approach can give a better understanding of the correspondence between the nesting of ! and the resources involved, such as quantifiers or copying. Alternatively, one could try to do away with ! entirely, and use some proof system where the safety of fold is captured by a structural property of the proof. One idea in this direction is to look at cyclic proofs [10]. Another idea would be to capture the structural property using the visual language of string diagrams.

Another question that concerns string diagrams is about the equivalence problem. Decidability of the equivalence problem for polyrergular functions is an open problem, but in the case of linear functions the problem is known to be decidable [16, Theorem 1]. Maybe one can express the decision procedure in terms of string diagrams, by designing equivalences on string diagrams which identify exactly those diagrams that describe the same function.

The system in this paper is based on combinators. A more powerful system would also allow for variables, λ, and higher-order types. Such a system exists without fold [6, Section 4], and it is tempting to see if it can be extended with fold. The result would be an expressive functional programming language that can only define regular functions.

References

[1] 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.

[2] Rajeev Alur and Loris D'Antoni. Streaming Tree Transducers. *J. ACM*, 64(5):31:1–31:55, August 2017.

[3] 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.

[4] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 283–293, 1992.

[5] Mikołaj Bojańczyk. Polyregular Functions. *CoRR*, abs/1810.08760, 2018.

[6] Mikołaj Bojańczyk. Transducers of polynomial growth. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[7] 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.

[8] Mikołaj Bojańczyk and Amina Doumane. First-order tree-to-tree functions. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 252–265. ACM, 2020.

[9] 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.

[10] James Brotherston and Alex Simpson. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation*, 21(6):1177–1216, 2011.

[11] Bob Coecke and Alex Kissinger. *Picturing quantum processes*. Cambridge University Press, 2017.

[12] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*, volume 138 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, 2012.

[13] 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.

[14] 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.

[15] Noa Globerman and David Harel. Complexity results for two-way and multi-pebble automata and their logics. *Theor. Comput. Sci.*, 169(2):161–184, 1996.

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

[17] Jörg Flum Heinz-Dieter Ebbinghaus. *Finite Model Theory*. Springer Monographs in Mathematics. Springer, 2nd edition, 2006.

[18] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

[19] Kenneth Krohn and John Rhodes. Algebraic theory of machines. i. prime decomposition theorem for finite semigroups and machines. *Transactions of the American Mathematical Society*, 116:450–450, 1965.

[20] Damiano Mazza. Simple parsimonious types and logarithmic space. In *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2015.

[21] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. *J. Comput. Syst. Sci.*, 66(1):66–97, 2003.

[22] Lê Thành Dung Nguyên, Camille Noûs, and Pierre Pradic. Comparison-free polyregular functions. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, pages 139:1–139:20, 2021.

[23] 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.

[24] Thomas Wilke. An algebraic characterization of frontier testable tree languages. *Theoretical Computer Science*, 154(1):85–106, 1996.

A The quantifier-free system

In this part of the appendix, we prove Theorem 4.1. In the proof, a *derivable function* is a function that can be derived using the system from Theorem 4.1. In other parts of the paper, derivable functions will refer to other systems.

The proof of Theorem 4.1 has two parts: soundness (i.e. all derivable functions are quantifier-free interpretations) and completeness (i.e. all quantifier-free interpretations are derivable).

A.1 Soundness

To prove soundness of the system, we show that all prime functions from Figure 1 are quantifier-free interpretations, and that the class of quantifier-free interpretations is closed under applying all combinators from Figure 2.

We only discuss one case, namely the combinator

$$\frac{\Sigma \to \Gamma}{\Sigma^* \to \Gamma^*} \qquad \text{functoriality of } *,$$

which is also known as the *map combinator*. The difficulty with this combinator is that in the structure that represents a list of elements $[A_1, \ldots, A_n] \in \Sigma$, as per Definition 2.4, the nullary predicates from the structures $A_1, \ldots, A_n$ are replaced by unary predicates. However, since the same replacement is done for the output list, it follows that a straightforward syntactic construction can be applied to transform the quantifier-free interpretation from the assumption of the combinator into a quantifier-free interpretation from the conclusion.

The rest of the soundness proof is left to the reader.

A.2 Completeness

The rest of this section is devoted to the completeness proof. We begin with some notation and preparatory lemmas that will be used in the proof.

Zero type. We will use an extended system, which has an additional type called 0. This type represents a class that contains one structure, and that structure has an empty universe. (This class is terminal, in the sense that every class of structures admits a unique quantifier-free interpretation to 0.) The corresponding prime functions are

$$\Sigma \to \Sigma \times 0 \qquad \text{add } 0$$

$$0 \to \Sigma^* \qquad \text{create an empty list}$$

One should not confuse 0 with the empty class $\varnothing$ (which anyway is not part of our type system). For example,

$$0 + \Sigma \neq \Sigma = \varnothing + \Sigma.$$

The extended system with 0 is equivalent to the original system, since we can view 0 as 1^*, but with only the empty list used. In particular, the extended system is conservative in the following sense: if a function between types that do not use 0 is derivable in the extended system, then it is also derivable in the non-extended system. For this reason, we can do the completeness proof in the extended system, which will be slightly more convenient. From now on, list types can use 0.

Disjunctive normal form. It will be useful to consider list types in a certain normal form, which is achieved using distributivity. We say that a list type is in *disjunctive normal form* if it is of the form

$$\coprod_{i \in I} \prod_{j \in I_j} \Sigma_{i,j}$$

where each $\Sigma_{i,j}$ is one of the types 0 or 1, or a list Σ^* where Σ is in disjunctive normal form. In other words, the list type does not contain any product of co-products.

In our proof, the main advantage of this normal form concerns nullary relations. Recall that the nullary relations in Definition 2.4, appear only in the co-product, and they are removed when applying the list constructor. Therefore, if a type in disjunctive normal form is not a co-product type, then its vocabulary contains no nullary relations.

The following lemma shows that every list type admits a derivable isomorphism with some list type in disjunctive normal form. Here, a *derivable isomorphism* is a derivable function that has a derivable inverse.

Lemma A.1. *Every list type admits a derivable isomorphism with some list type in disjunctive normal form.*

Proof
Using distributivity and functoriality. □

Thanks to the already proved soundness part of the theorem, the derivable isomorphism is also quantifier-free. Therefore, to prove completeness of the system, it is enough to prove completeness only for functions where both the input and output types are in disjunctive normal form. From now on, we only consider list types in disjunctive normal form.

Safe pairing. The last issue to be discussed before the completeness proof concerns pairing functions. Suppose that

$$f : \Sigma \to \Gamma_1 \times \Gamma_2$$

is a quantifier-free interpretation. In the completeness proof, we will want to show that it is derivable. A natural idea would be to use an inductive argument to derive the two quantifier-free interpretations

$$f_i : \Sigma \to \Gamma_i$$

that arise from f by projecting it onto the two output coordinates, and to then pair these two derivations into a derivation of f. Unfortunately, combining these two derviations would require some kind of pairing combinator, or a duplicating function of type $\Sigma \to \Sigma \times \Sigma$, none of which are available in our system (because they would be unsound).

For these reasons, we need to be a bit careful with pairing. The crucial observation is that pairing is not always unsound, because some functions can be paired. For example, the two functions f_1 and f_2 described above can be paired, because they use disjoint parts of the input structure. More formally, the universe formulas are disjoint, i.e. no element can be selected by both universe formulas. This view will be used in the completeness proof. To formalize it, we use the following lemma.

Lemma A.2. *Let Σ be a list type in disjunctive normal form, and let $\varphi(x)$ be a quantifier-free formula over its vocabulary. There is a list type, denoted by $\Sigma|\varphi$, and a quantifier-free interpretation*

$$\Sigma \xrightarrow{\text{projection of } \varphi} \Sigma|\varphi$$

such that the following conditions are satisfied.

1. *For every quantifier-free interpretation $f : \Sigma \to \Gamma$, such that the universe formula of f is contained in φ (which means that the universe formula of f implies the formula φ), there is a a decomposition*

$$\begin{tikzcd} \Sigma \arrow[rr, "f"] \arrow[dr, "\text{projection of } \varphi"'] & & \Gamma \\ & \Sigma|\varphi \arrow[ur, "f|\varphi"'] & \end{tikzcd}$$

where $f|\varphi$ is a quantifier-free interpetation.

2. ***Safe pairing.*** *Suppose that $\varphi_1, \dots, \varphi_n$ are formulas as in the assumption of the lemma, which are pairwise disjoint. Then one can derive the function*

$$\Sigma \longrightarrow (\Sigma|\varphi_1) \times \cdots \times (\Sigma|\varphi_n)$$

that produces all projections in parallel.

Proof

The purpose of the type 0 is this lemma; with the type used $\Sigma|\varphi$ when φ selects no elements. The lemma is proved by induction on the structure of the type Σ.

- Suppose that Σ is the zero type 0. In this case, the formula φ must be equivalent to "false". We define $0|\varphi$ to be the same type 0, and the projection is the identity. The safe pairing condition holds because of the prime function $\Sigma \to \Sigma \times 0$.
- Suppose that Σ is the unit type 1. In this case, the formula φ is equivalent to either "false" or "true", since the unique structure in 1 has a universe that has only one element. We define $1|\varphi$ to be the 0 or 1, depending on which of the two cases holds, with the projection being the unique function $1 \to 1|\varphi$. The safe pairing condition is proved using the prime function $\Sigma \to \Sigma \times 0$, since the list of quantifier-free formulas in the condition can have at most one formula that is not "false".
- Consider a list type of the form Σ^*. The main observation in the proof is that there is a bijective correspondence between quantifier-free formulas over the vocabularies of Σ and Σ^*. This correspondence is defined as follows: for every formula φ over the vocabulary of Σ, there is a formula φ^* over the vocabulary of Σ^* such that for every list

$$A = [A_1, \dots, A_n] \in \Sigma^*,$$

an element $a \in A_i$ is selected by φ^* in the entire list A if and only if a is selected by φ in the list element A_i. It is not hard to see that such a formula exists, and furthermore, every formula over the vocabulary of Σ^* is of equivalent to a formula of the form φ^*.
Therefore, in the case when the type is a list Σ^*, we can assume that the formula over the vocabulary of Σ^* is of the form φ^* for some formula φ over the vocabulary of Σ. Define

$$\Sigma^*|\varphi^* \quad \overset{\text{def}}{=} \quad (\Sigma|\varphi)^*,$$

with the projection function for φ^* being the result of applying the map combinator to the projection function for φ. The safe pairing property is proved by using the induction assumption, and using the function

$$(\Sigma_1 \times \cdots \times \Sigma_n)^* \to \Sigma_1^* \times \cdots \times \Sigma_n^*,$$

which can easily be seen to be derivable.
- The case when Σ is a co-product $\Sigma_1 + \Sigma_2$ is proved similarly to the list case. Here, we use a bijective correspondence between quantifier-free formulas φ over the vocabulary of Σ with pairs (φ_1, φ_2), where φ_i is a quantifier-free formula over the vocabulary of Σ_i.
- The case when Σ is a product $\Sigma_1 \times \Sigma_2$ is proved similarly to the co-product case. Again, there is a bijective correspondence between quantifier-free formulas φ over the vocabulary of Σ with pairs (φ_1, φ_2), where φ_i is a quantifier-free formula over the vocabulary of Σ_i. For the existence of such a bijective correspondence, we use the assumption that the type is in disjunctive normal form. Thanks to the assumption, the vocabulary has no nullary relations; if there would be nullary relations then there could be some communication between the two coordinates in the product.

□

Completeness. Consider a quantifier-free interpretation

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

Let φ be the universe formula of f, and let $\Sigma|\varphi$ be the type obtained by applying Lemma A.2. We write $\mathrm{dom} f$ for this type. The corresponding function in the decomposition as in item 1 is then

$$f|\mathrm{dom} f : \mathrm{dom} f \to \Gamma.$$

. We will use the following terminology for this decomposition: the type $\Sigma|\varphi$ will be called the *reduced domain of f*, the projection will be called the *domain reduction of f*, and the function g will be called *reduced f*. Here is a diagram that displays this terminology

$$\begin{tikzcd}
\Sigma \arrow[rr, "f"] \arrow[dr, "\text{domain reduction of } f"'] & & \Gamma \\
& \text{reduced domain of } f \arrow[ur, "\text{reduced } f"'] &
\end{tikzcd}$$

Because the domain reduction is derivable, and derivable functions are closed under composition, it is enough to show that for every quantifier-free interpretation, its reduced version is derivable. This will be shown in the following lemma.

Lemma A.3. *For every quantifier-free interpretation*

$$f : \Sigma \to \Gamma$$

with universe formula φ, one can derive the function

$$f|\varphi : \Sigma|\varphi \to \Gamma$$

from item 1 in Lemma A.2.

Proof

The lemma is proved by structural induction on the input and output types. In the induction step, we will replace either the input or output type by a simpler one. The induction step is shown in Sections A.2.2–A.2.5 below, which consider the following cases:

A.2.1 the input type is a co-product;
A.2.2 the output type is a co-product;
A.2.3 the output type is a product;
A.2.4 the input type is 0 or 1;
A.2.5 the input type is a list;
A.2.6 the input type is a product.

These cases are exhaustive, i.e. at least one of them always applied, but they are not disjoint. When applying some case, we assume that none of the previous cases can be applied. The induction basis corresponds to case A.2.4.

A.2.1 The input type is a co-product. In the representation of the co-product type from Definition 2.4, the information about whether the structure comes from the first or second case is stored in a nullary predicate. Therefore, by a straightforward syntactic manipulation of quantifier-free interpretations, from a quantifier-free interpetation

$$f : \Sigma_1 + \Sigma_2 \to \Gamma,$$

we can obtain two quantifier-free interpretations

$$f_1 : \Sigma_1 \to \Gamma \qquad f_2 : \Sigma_2 \to \Gamma$$

which describe the behaviour of f on inputs from Σ_1 and Σ_2, respectively. Let φ be the universe formula of f, and let φ_1 and φ_2 be the universe formulas of f_1 and f_2. By induction assumption, we can derive

$$f_i|\varphi_i : \Sigma_i|\varphi_i \to \Gamma$$

and derive their reduced versions. Since by definition we have

$$(\Sigma_1 + \Sigma_2)|\varphi = \Sigma_1|\varphi_1 + \Sigma_2|\varphi_2,$$

we can combine these two derivations into a derivation $f|\varphi$, by using the combinator

$$\frac{\Delta_1 \to \Gamma \quad \Delta_2 \to \Gamma}{\Delta_1 + \Delta_2 \to \Gamma} \qquad \text{cases},$$

which itself can be derived using functoriality of + and the co-diagonal.

A.2.2 The output type is a co-product. Consider a function

$$f : \Sigma \to \Gamma_1 + \Gamma_2$$

whose output type is a co-product. In this case, we assume that the previous case cannot be applied, i.e. the input type is not a co-product.

To produce the output structure, we need to define the nullary predicate that says which of the two cases in the output type is used. In a quantifier-free interpretation, this nullary predicate is defined by a quantifier-free formula, with no free variables, which is evaluated in the input structure. Since there are no nullary predicates in the input structure (because otherwise, the input type would be a co-product, and we could apply the case from the previous section), it follows that this quantifier-free formula is either "true" or "false". This means that the function f must always use the same variant Γ_1 or Γ_2 in the co-product from the output type, regardless of the choice of input structure. Therefore, we can replace f by a corresponding function of type $\Sigma \to \Gamma_i$, apply the induction assumption, and conclude by using composition and the co-projection.

A.2.3 The output type is a product. Consider a function

$$f : \Sigma \to \Gamma_1 \times \Gamma_2$$

whose output type is a product. We split this function into two quantifier-free interpretations

$$f_1 : \Sigma \to \Gamma_1 \quad f_2 : \Sigma \to \Gamma_2,$$

which produce the two coordinates in the output of f. These two functions must have disjoint universe formulas, since otherwise the same element in the output structure would belong to both coordinates of a pair. We can apply the induction assumption, and then combine these derivations into a derivation of f by using safe pairing from Lemma A.2.

A.2.4 The input type is 0 or 1. By cases A.2.2 and A.2.3, we can assume that the output type of the unique function in the family is either 0, 1, or a list type Γ^*.

When the output type is 0 or 1, then we are dealing with a quantifier-free interpretation which has one of the types

$$0 \to 0 \quad 0 \to 1 \quad 1 \to 0 \quad 1 \to 1.$$

There is no quantifier-free interpretation of the type $1 \to 0$, and for the remaining types there is exactly one quantifier-free interpretation, which is easily seen to be derivable.

We are left with the case when the output type is Γ^*. If the input type is 0, then the quantifier-free interpretation necessarily produces the empty list, and it is therefore derivable. If the input type is 1, then the function always produces the same output, which is either the empty list, in which case it can be derived using the list constructor, or a singleton list $[A]$ for some fixed structure $A \in \Gamma$. In the singleton case, we can use the induction assumption to derive the function $1 \mapsto A$, and pack the result as a list using the list unit operation.

A.2.5 The input type is a list. We now arrive at the most interesting case in the proof, which is when the input type is a list Σ^*. Because the previously studied cases A.2.2 and A.2.3 cannot be applied, the output type is one of 0, 1, or Γ^*. When the output type is 0, there is only one possible function, which is easily derivable. The output type 1 is impossible, since the function could not handle an empty list on the input. We are left with a list-to-list function. To prove the inductive step for such functions, we use the analysis from the following claim.

Claim A.4. *For every quantifier-free interpretation*

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

one can find quantifier-free interpretations

$$f_1, \ldots, f_k : \Sigma^* \to \Gamma^*$$

with disjoint universe formulas such that f is equal to

$$A \in \Sigma^* \quad \mapsto \quad \underbrace{f_1(A) \cdots f_k(A)}_{\text{list concatenation}}$$

and each f_i has one of the following properties:

1. *all output lists of f_i have length at most one.*
2. *there is some quantifier-free interpretation*

   $$g : \Sigma \to \Gamma^*$$

   such that f_i is equal to

   $$[A_1, \ldots, A_n] \mapsto \underbrace{g(A_1) \cdots g(A_n)}_{\text{list concatenation}}$$

3. *as in item 2, but with reverse list order $g(A_n) \cdots g(A_1)$.*

Before proving the claim, we use it to complete the induction step of the lemma in the present list-to-list case. Apply Claim A.4 to the function f, yielding a decomposition into functions $f_1, \ldots, f_k$. The induction assumption can be applied to these functions, since item 1 in the claim gives a smaller output type (namely Γ instead of Γ^* for the only list element), while the remaining two items give smaller input types. Finally, these derivations can be combined into a derivation of f, using the pairing operation from Lemma A.2, the function for list concatenation from Example **??**, and the prime function

$$(\Sigma \times \Gamma)^* \to \Sigma^* \times \Gamma^* \qquad \text{list distribute}$$

which is used to separate the domains of the functions $f_1, \ldots, f_k$ from the input list. It remains to prove the claim.

Proof (of Claim A.4)
Consider the universe formula $\varphi(x)$ of f. Decompose this formula as a finite union

$$\varphi(x) = \bigvee_{\sigma \in \Phi} \sigma(x)$$

of quantifier-free theories as in Definition 3.3, i.e. quantifier-free formulas that specify all relations satisfied by x. Take some input structure in Σ^*. For elements of this structure that satisfy the universe formula, there are two orders: the *input order* that describes the order in the input list

$$A = [A_1, \ldots, A_n] \in \Sigma^*$$

and the *output order* that describes the order in the output list

$$f(A) = [B_1, \ldots, B_m] \in \Gamma^*.$$

In the proof of the claim, we will analyze the relationship between these two orders. Both of these orders are reflexive, total, and transitive, but not necessarily anti-symmetric, since two elements may belong to the same list element.

For an element a in an input structure $A \in \Sigma^*$ that satisfies the universe formula $\varphi(x)$, the *unary theory* of a is defined to be the unique quantifier-free theory $\sigma \in \Phi$ that is satisfied by a. If a is strictly smaller than b in the input order, then by compositionality, the output order on a and b will be uniquely determined by the unary theories of the two individual elements a and b. This means that exactly of the following three implications must hold

$$\begin{tikzcd}
 & \text{a is strictly before b in the output order} \\
\text{a is strictly before b in the input order and the unary theories of a and b are σ and τ} \arrow[ur, Rightarrow, "\sigma<\tau"] \arrow[r, Rightarrow, "\sigma\sim\tau"] \arrow[dr, Rightarrow, "\sigma>\tau"'] & \text{a is equivalent to b in the output order} \\
 & \text{a is strictly after b in the output order}
\end{tikzcd}$$

Depending on which implication holds, we write one of

$$\sigma < \tau \quad \sigma \sim \tau \quad \sigma > \tau.$$

Before continuing, we make two cautionary remarks about the notation involving the relations < and > described above. The first cautionary remark is that < and > describe relations that are not necessarily converses of each other, since $\sigma < \tau$ and $\tau > \sigma$ do not mean the same thing; one of these conditions could be true without the other one being true. The second cautionary remark is that $\sigma < \tau$ is not necessarily obtained from some partial order by looking at strictly growing pairs. For example, we could have both $\sigma < \tau$ and $\tau < \sigma$.

To prove the claim, we make five observations about the relations <, > and ~. In these observations, we use *partial equivalence relations*; a partial equivalence relation is defined to be a binary relation that is symmetric and transitive but not necessarily reflexive. Equivalence classes of partial equivalence relations are defined in the expected way; the only difference is that some elements of the domain might not belong to any equivalence class.

1. The first observation is that $\sigma \sim \tau$ is a partial equivalence relation. It is easy to see that the relation $\sigma \sim \tau$ is transitive. We now argue that it is symmetric. (This is not immediately obvious.) Suppose that $\sigma \sim \tau$. Consider a list in $A \in \Sigma^*$ with four distinguished elements

   $$\underbrace{a_1}_{\substack{\text{unary}\\ \text{type } \sigma}} < \underbrace{a_2}_{\substack{\text{unary}\\ \text{type } \tau}} < \underbrace{a_3}_{\substack{\text{unary}\\ \text{type } \sigma}} < \underbrace{a_4}_{\substack{\text{unary}\\ \text{type } \tau}}$$

   with the order relationship describing the input order. From the assumption on $\sigma \sim \tau$ we can conclude that three pairs (depicted by lines in the following diagram) belong to the same elements in the output list:

$$\begin{tikzcd} a_1 \arrow[r, dash, "\sigma\sim\tau"'] \arrow[rrr, dash, bend left, "\sigma\sim\tau"] & a_2 & a_3 \arrow[r, dash, "\sigma\sim\tau"'] & a_4. \end{tikzcd}$$

Since belonging to the the same element in the output list is a transitive relation, we can deduce that a_2 and a_3 belong to the same element in the output list, thus establishing $\tau \sim \sigma$.

2. The next observation is that $(\sigma < \tau \wedge \tau < \sigma)$ is a partial equivalence relation. It is symmetric by definition, and it is transitive because each of the two conjuncts is transitive.
3. By the same proof as in the previous item, $(\sigma > \tau \wedge \tau > \sigma)$ is a partial equivalence relation.
4. We now show that the equivalence classes of the partial equivalence relations described in the first three observations are disjoint, and give a partition of
$$\Phi = \Phi_1 \cup \cdots \cup \Phi$$
of all unary types in Φ. For every $\sigma \in \Phi$, we have exactly one of the cases $\sigma \sim \sigma$, $\sigma < \sigma$, or $\sigma > \sigma$. This proves that every σ belongs to exactly one of the equivalence classes in the previous three items.
5. The last observation is that the order on equivalence classes in the previous item can be chosen so that for all $i < j$ we have
$$\sigma \in \Phi_i \text{ and } \tau \in \Phi_j \quad \Rightarrow \quad \sigma < \tau.$$
Let Φ_i and Φ_j be different equivalence classes from the previous item. For every $\sigma \in \Phi_i$ and $\tau \in \Phi_j$ we have exactly one of the three cases
$$\sigma < \tau \quad \text{or} \quad \sigma > \tau \quad \text{or} \quad \sigma \sim \tau.$$
The third case cannot hold, since otherwise Φ_i and Φ_j would be in the same equivalence class from the first observation. Therefore, one of the two first cases must hold. A short analysis, which is left to the reader, also shows that which of the two cases holds (first or second) does not depend on the choice of the σ and τ. This means that there is an unambiguous order relationship between Φ_i and Φ_j, and this relationship can be used to prove item 5 of the claim.

Let $\Phi_1, \ldots, \Phi_m$ be as in the last of the above observations. We know that for every input structure $A \in \Sigma^*$, the output list can be decomposed as
$$f(A) = f_1(A) \cdots f_n(A)$$
where f_i is the function obtained from f by restricting the output elements to those that have type from Φ_i in the input structure. To complete the proof of the claim, we will show that each function f_i has one of the three kinds in the statement of the claim.

Suppose first that Φ_i is an equivalence class defined by $\sigma \sim \tau$ as in the first observation. This means that all outputs produced by f_i are equivalent in the output order. Hence this f_i is of kind 1 as in the statement of the claim.

Suppose now that Φ_i is an equivalence class defined by $(\sigma < \tau \wedge \tau < \sigma)$ as in the second observation. This means that for every input list $A \in \Sigma^*$, if we take two elements a and b that have unary theory in Φ_i, then

$$\begin{array}{c} a \text{ is strictly before } b \text{ in the input order} \\ \Downarrow \\ a \text{ is strictly before } b \text{ in the output order} \end{array}$$

Hence this f_i is of kind 2 as in the statement of the claim.

A symmetric argument works for an equivalence class defined by $(\sigma > \tau \wedge \tau > \sigma)$, except that this time the output order is reversed, giving a function as in item 3 of the lemma.
□

A.2.6 The input type is a product. The final case in the proof of Lemma A.3 is when the input type is a product. Since all types are in disjunctive normal form, the input type is a product
$$\Sigma = \Sigma_1 \times \cdots \times \Sigma_m$$
where each Σ_i is either 1 or a list. (The type 0 can be removed from a product.) Because the previously studied cases A.2.2 and A.2.3 about output types that are products or co-products cannot be applied, the output type is either 0, 1, or a list type Γ^*.

If the output type is 0, then the function is easily derivable.

Consider now the case when the output type is 1. It cannot be the case that each of the input types $\Sigma_1, \ldots, \Sigma_m$ is a list, since the quantifier-free interpretation would be unable to handle the case when all lists are empty. Therefore, one of the input types is the unit type 1, and the conclusion of the lemma can be proved by using $1 \to 1$.

We are left with the case when the ouput type is of the form Γ^*. Here, we proceed in the same way as in Section A.2.5, with the corresponding version of Claim A.4 being the following claim. The proof of the claim, which uses a similar analysis of unary quantifier-free theories as in Claim A.4, is left to the reader.

Claim A.5. *For every quantifier-free interpretation*
$$f : \underbrace{\Sigma_1 \times \cdots \times \Sigma_m}_{\Sigma} \to \Gamma^*$$
one can find quantifier-free interpretations
$$f_1, \ldots, f_k : \Sigma_1 \times \Sigma_2 \to \Gamma^*$$
with disjoint universe formulas such that f is equal to
$$A \in \Sigma \quad \mapsto \quad \underbrace{f_1(A) \cdots f_k(A)}_{\text{list concatenation}}$$
and each f_i has one of the following properties:

1. *all output lists of f_i have length at most one; or*

$G^* \to G$	group multiplication
$\Sigma \to \Sigma \times \Sigma$	diagonal
$\Sigma^* \to 1 + \Sigma \times \Sigma^*$	list destructor
$(\Sigma + \Gamma)^* \to (\Sigma^* + \Gamma^*)^*$	block
$\Sigma^* \to (\Sigma^* \times \Sigma^*)^*$	split

Figure 4. Additional polyregular prime functions from [5].

2. *f_i factors through the projection*

$$\Sigma_1 \times \cdots \times \Sigma_m \to \Sigma_j \qquad \textit{for some } j \in \{1, \ldots, m\}.$$

This completes the last of the cases in the induction step, and thus also the proof of the lemma, which also completes the proof of Theorem 4.1. □

B Completeness for polyregular functions

In this section, we prove the completeness of the system in Theorem 5.3, i.e. we show that every polyregular function can be weakly derived. This implication is the less interesting one, since our system is designed to be powerful, i.e. it should be easy to derive functions in it. We will deduce the completeness of our system with fold from another completeness result that uses a system without fold.

We begin by describing the system that we reduce to. It has all of the combinators from Figure 2, and its prime functions are contained in those from Figure 1 plus certain additional functions that are described in Figure 4. The first three primes from Figure 4 have already been discussed in the paper, so we only explain the block and split functions. The split function of type

$$\Sigma^* \to \Sigma^* \times \Sigma^*$$

outputs all possible ways of splitting the input list into (prefix, suffix) pairs, as explained in the following example:

$$\begin{array}{c}[1,2,3] \\ \downarrow \\ [([],[1,2,3]),([1],[2,3]),([1,2],[3]),([1,2,3],[])].\end{array}$$

The other additional function is the block function of type

$$(\Sigma + \Gamma)^* \to (\Sigma^* + \Gamma^*)^*,$$

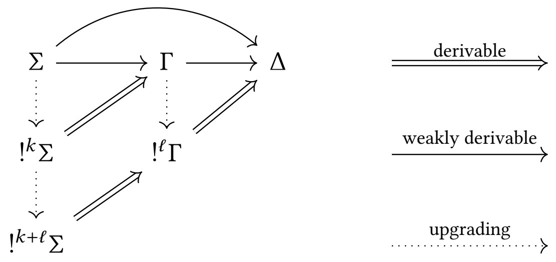
which blocks the elements of the input list into maximal blocks of same type, as illustrated in the following example that uses numbers for elements of Σ and letters for elements of Γ:

$$\begin{array}{c}[1,2,a,3,4,5,b,c] \\ \downarrow \\ [[1,2],[a],[3,4,5],[b,c]].\end{array}$$

Theorem B.1. *[5, p. 64] A function between list types is polyregular if and only if it can be derived using the prime functions and combinators from the quantifier-free system Theorem 4.1, plus the prime functions from Figure 4.*

In contrast to the system with fold from this paper, the system from the above theorem was designed to be minimal, and therefore, the completeness proof for the system with fold will be a simple corollary of completeness of the system from the above theorem. Thanks to Theorem B.1, to prove the completeness result for our system with fold, it is enough to show that (a) all prime functions in Theorem B.1 are weakly derivable; and (b) the combinators in Theorem B.1 preserve the weakly derivable functions.

Combinators. Consider first (b), about the combinators. The combinators are those from Figure 2. There is one combinator for function composition, and three combinators for functoriality. The combinators for functoriality are dealt with using the prime functions about ! commuting with the remaining constructors. The combinator for function composition is explained in the following diagram:


$$\begin{tikzcd}
\Sigma \arrow[r] \arrow[d, dotted] \arrow[rr, bend left] & \Gamma \arrow[r] \arrow[d, dotted] & \Delta & & \arrow[r, Rightarrow, "\text{derivable}"] & {} \\
!^k\Sigma \arrow[ur, Rightarrow] \arrow[d, dotted] & !^\ell\Gamma \arrow[ur, Rightarrow] & & & \arrow[r, "\text{weakly derivable}"] & {} \\
!^{k+\ell}\Sigma \arrow[ur, Rightarrow] & & & & \arrow[r, dotted, "\text{upgrading}"] & {}
\end{tikzcd}$$

Prime functions. Consider now (a), about the prime functions. Clearly all prime functions in the quantifier-free system are weakly derivable, since they are even strongly derivable. Weak derivability of the additional functions for group multiplication and the list destructor was already discussed in Examples 2 and 14. The diagonal function can easily be weakly derived using absorption. We are left with the split and block function.

Lemma B.2. *Split and block are weakly derivable.*

Proof

To weakly derive the split function, we use the prefixes function from Claim 5.4. If we take a list

$$[a_1, \ldots, a_n] \in \Sigma^*,$$

and then apply prefixes, reverse, followed prefixes again, then the output is a list in Σ^{***} of length n whose i-th element is

$$[[a_1, \ldots, a_n], [a_1, \ldots, a_{n-1}], \ldots, [a_1, \ldots, a_i]]. \qquad (2)$$

Since weakly derivable functions are closed under composition, this output can be produced by a weakly derivable function. Since weakly derivable functions are also closed under map, to complete the proof that split is weakly derivable, it remains to show that a weakly derivable function can transform the i-th element in (2) into the corresponding element in the output of split, namely

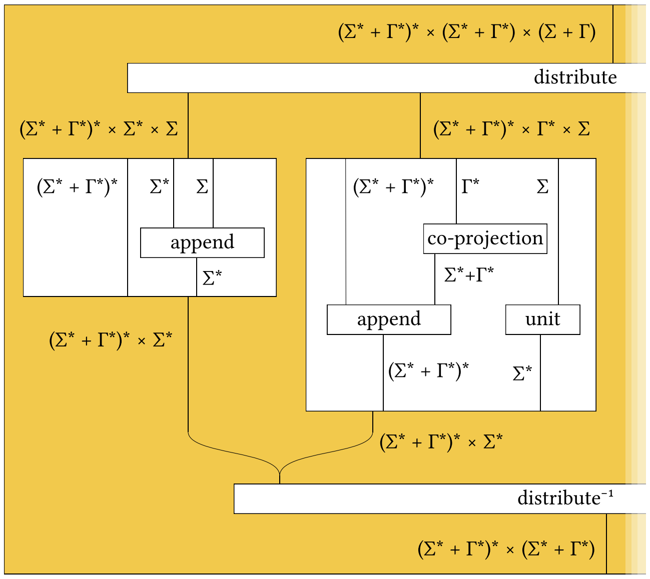
$$([a_1, \ldots, a_i], [a_{i+1}, \ldots, a_n]). \tag{3}$$

This is done as follows: using the list deconstructor, we split the list in (2) into its head and tail. The head is reversed, while the tail is transformed so that each list element is replaced by its own head.

We now turn to the block function. One approach is to derive the block function from split – thus showing that it is not needed in the system. This is shown in [5, p.90]. However, since we will later use a system that uses block but not split, we show how to derive block directly. To compute the block function, we use an automaton where the input alphabet is $\Sigma + \Gamma$, the state space is

$$\Delta = (\Sigma^* + \Gamma^*)^* \times \underbrace{(\Sigma^* + \Gamma^*)}_{\text{most recent block}}$$

and the transition function is illustrated in the following diagram (by symmetry, we only draw the left half):



In the diagram, the unit function is the function $x \mapsto [x]$ which can be derived as in Figure 3. If we set the initial state of the above automaton to be a pair of empty lists (the second one having type, say, Σ^*), then after reading a list in $!(\Sigma+\Gamma)^*$, its state will store the output of the block operation, except that the last list element will be held separately and will need to be added using append. □

B.1 A smaller system

A corollary of the completeness proof is Theorem 5.5, which shows that certain primes and combinators can be removed from the system in Theorem 5.3, while keeping it complete. We remove the map combinator, as well as all quantifier-free functions from Figure 1 that involve the list type, namely the functions

$\Sigma^* \times \Sigma \to \Sigma^*$	append
$\Sigma^* \to \Sigma^*$	reverse
$\Sigma^{**} \to \Sigma^*$	concat
$\Sigma \to \Sigma \times \Gamma^*$	create empty
$(\Sigma \times \Gamma)^* \to \Sigma^* \times \Gamma^*$	list distribute

In their place, we have only two functions

$1 + \Sigma \to \Sigma^*$	lists of length at most one
$\Sigma^* \times \Sigma^* \to \Sigma^*$	binary list concatenation.

We will show that the smaller system remains complete, because it can weakly derive the removed functions, and furthermore, the weakly derivable functions in the smaller system are closed under the map combinator.

Proof (of Theorem 5.5)
Consider first the prime functions that are removed from the smaller system. The append function can be (strongly) derived in the smaller system. Using append, we can (strongly) derive the left list constructor, whose safe folding gives the list reversal in type

$$!\Sigma^* \to \Sigma^*.$$

is obtained by composing a co-projection with the right list constructor. Applying the safe fold combinator to the left list constructor (after swapping the order of its arguments) shows that the reverse function can be derived in type

$$!\Sigma^* \to \Sigma,$$

and hence it is weakly derivable. The concat function is derived in type

$$!\Sigma^{**} \to \Sigma^*$$

by folding binary list concatenation. To weakly derive the create empty function, we observe that for every type Σ we can derive the unique function

$$!\Sigma \to 1,$$

and this derivation can be used together with absorption to derive the create empty function in type

$$!\Sigma \to \Sigma \times \Gamma^*.$$

Finally, the list distribute function can be derived in type

$$!(\Sigma \times \Gamma)^* \to \Sigma^* \times \Gamma^*$$

by a straightforward application of safe fold.

Finally, we can also eliminate the map combinator (functoriality of $*$), since using safe fold we obtain a version of the map combinator in type

$$\frac{\Gamma \to \Sigma}{!\Gamma^* \to \Sigma^*} \qquad \text{weak map,}$$

$1 + \Sigma \to \Sigma^*$	lists of length at most one
$\Sigma^* \times \Sigma^* \to \Sigma^*$	binary list concatenation
$!(\Gamma + \Sigma) \leftrightarrow !\Gamma + !\Sigma$	! commutes with +
$!(\Gamma \times \Sigma) \leftrightarrow !\Gamma \times !\Sigma$	! commutes with ×
$(!\Gamma)^* \leftrightarrow !(\Gamma^*)$	! commutes with ∗
$!\Gamma \to !\Gamma \times \Gamma$	absorption
$\Gamma \times \Sigma \leftrightarrow \Sigma \times \Gamma$	commutativity of ×
$\Gamma + \Sigma \leftrightarrow \Sigma + \Gamma$	commutativity of +
$\Gamma \times (\Sigma \times \Delta) \leftrightarrow (\Gamma \times \Sigma) \times \Delta$	associativity of ×
$\Gamma + (\Sigma + \Delta) \leftrightarrow (\Gamma + \Sigma) + \Delta$	associativity of +
$\Gamma \times (\Sigma + \Delta) \leftrightarrow (\Gamma \times \Sigma) + (\Gamma \times \Delta)$	distributivity
$\Gamma_1 \times \Gamma_2 \to \Gamma_i$	projections
$\Gamma_i \to \Gamma_1 + \Gamma_2$	co-projections
$\Gamma + \Gamma \to \Gamma$	co-diagonal

$\dfrac{!^k 1 \to \Gamma \qquad \Gamma \times \Sigma \to \Gamma}{!^k \Sigma^* \to \Gamma}$	safe fold
$\dfrac{\Gamma \to \Sigma \quad \Sigma \to \Delta}{\Gamma \to \Delta}$	function composition
$\dfrac{\Gamma_1 \to \Sigma_1 \quad \Gamma_2 \to \Sigma_2}{\Gamma_1 \times \Gamma_2 \to \Sigma_1 \times \Sigma_2}$	functoriality of ×
$\dfrac{\Gamma_1 \to \Sigma_1 \quad \Gamma_2 \to \Sigma_2}{\Gamma_1 + \Gamma_2 \to \Sigma_1 + \Sigma_2}$	functoriality of +
$\dfrac{\Gamma \to \Sigma}{!\Gamma \to !\Sigma}$	functoriality of !

Figure 5. A complete system for weakly deriving the polyregular functions. The safe fold combinator can only be applied when the type Γ has grade $< k$.

which is strong enough to replace the usual map combinator in the completeness proof of the system in Theorem 5.3. Summing, up we can reduce the system as stated in the following theorem. □

For easier reference, the system in the above theorem is described in Figure 5.

C Soundness for polyregular functions

In this section, we prove the soundness implication in Theorem 5.3. We prove that every strongly derivable function is a graded MSO interpretations. The prime functions from Figure 1 are quantifier-free, and therefore they are a special case of graded MSO interpretations. The extra prime functions from Theorem 5.3, namely absorption and those about ! commuting with the remaining type constructors, are easily seen to be the graded MSO interpretations. The combinators for functoriality are also easily seen to preserve graded MSO interpretations. There are two interesting cases, namely the combinators for function composition and safe fold.

C.1 Function composition

We first show that the graded MSO interpretations are closed under composition, as long as the input and output types are graded list types. Consider two graded MSO interpretations

$$\Sigma \xrightarrow{f_1} \Gamma \xrightarrow{f_2} \Delta.$$

We want to show that their composition

$$f_2 \circ f_1 : \Sigma \to \Delta$$

is a graded MSO interpretation. Let the corresponding polynomial functors be F_1 and F_2. The key tool is the following lemma.

Lemma C.1. *For every $k, r \in \{0, 1, \ldots\}$, the following function is* MSO *definable.*

Input *A structure $A \in \Sigma$ with distinguished elements $\bar{a} \in A^k$.*
Output *The rank r* MSO *theory of the tuple $F(\bar{a})$ in $f_1(A)$.*

Proof
This lemma reduces to closure under composition of MSO interpretations for list types [9, Corollary 8]. The result that we reduce to is non-trivial, and it depends on the fact that the input and output types are list types. □

Thanks to the above lemma, we can use a standard composition construction, with the polynomial functor for the composition being the composition $F_2 \circ F_1$ of the corresponding polynomial functors.

C.2 Safe fold

We are left with showing that graded MSO interpretations are closed under the safe fold combinator. All of the conceptual pieces are already in place, and we will simply show that the proof of Theorem 3.2 works, with minor adjustments to take into account the added generality of graded structures.

Suppose Γ is a type where all grades are $< k$, and we apply the safe fold combinator to graded MSO interpretations of types

$$!^k 1 \to \Gamma \qquad \text{and} \qquad \Gamma \to \Sigma \to \Gamma,$$

yielding a function of type

$$!^k \Sigma \to \Gamma.$$

By choice of k, in the resulting function every element in the input structure has strictly bigger grade than every element in the ouput structure. For such functions, the continuity condition in Definition 5.7 becomes trivial, and there is no difference between graded and un-graded MSO interpretations. Therefore, in order to prove the soundess of fold, it is enough to show that following lemma, that applying fold to a graded MSO interpretation yields an (ungraded) MSO interpretation.

Lemma C.2. *For every graded* MSO *interpretation*

$$\delta : \Gamma \times \Sigma \to \Gamma,$$

between graded list types, and every $B_0 \in \Gamma$, the following function is an (ungraded) MSO *interpretation*

$$A = \underbrace{[A_1, \ldots, A_n]}_{\substack{\text{list of structures in } \Sigma, \\ \text{with the grades forgotten}}} \quad \mapsto \quad \underbrace{B_n}_{\substack{\text{defined based on } A \\ \text{as in the proof of Calim 3.4}}}.$$

Proof

Let m be the maximal grade that appears in Γ, and let the polynomial functor in the transition function δ be

$$F(A) = F_0(A) + \cdots + F_m(A) + A.$$

By the continuity condition for the graded MSO interpretation δ, the elements of grade ℓ in B_n are the disjoint union of two sets:

1. grade ℓ elements in B_{n-1} or A_n; or
2. F_ℓ applied to grade $> \ell$ elements in B_{n-1} or A_n.

By unfolding the inductive definition of B_{n-1} in the first item of the above description, we see that the elements of grade ℓ in B_n are the disjoint union of two sets:

1*. grade ℓ elements in B_0 or $A_1, \ldots, A_n$; or
2*. F_ℓ applied to grade $> \ell$ elements in B_{i-1} or A_i for some $i \in \{1, \ldots, n\}$.

We will represent the elements that satisfy 1* or 2* as a subset of $G_\ell(A)$ for some polynomial functor G_ℓ. This functor is defined as follows by induction on ℓ, in reverse order $m, \ldots, 0$. Suppose that we want to define G_ℓ and assume that we have already defined $G_{\ell'}$ for $\ell' > \ell$. (In the induction basis of $\ell = m$ the assumption is empty.) To represent the elements in item 1*, we use the functor

$$A + \underbrace{1 + \cdots + 1}_{\substack{\text{number of elements} \\ \text{in } B_0 \text{ that have grade } \ell}}.$$

A tempting idea for item 2* is to use the functor

$$H_\ell(A) = F_\ell(G_{\ell+1}(A) + \cdots + G_m(A) + \underbrace{A}_{\substack{\text{represents elements of grade} > m \\ \text{in the input structure}}}).$$

Unfortunately, this idea is not correct. The reason is that in item 2*, there is a dijsoint union ranging over $i \in \{1, \ldots, n\}$, and the disjointness of this union is not taken into account by H_ℓ. The problem is that the universe of the structures $B_0, \ldots, B_n$ are not disjoint, and the functor H_ℓ can incorrectly identify elements that are obtained by applying F_ℓ to the same elements that appear in both B_i and B_j for $i \neq j$. To eliminate this problem, we will add an explicit identifier for the index j to the functor. To view the index i as an element of the input structure A_i, we use the first element in the universe of the corresponding list element A_i. Here, we when refer to the first element in the universe, we mean the natural linear order on the universe in a structure from a graded list type, which arises from the ordered nature of lists and pairs. Therefore, instead of $H_\ell(A)$, to represent item 2* we use the product $A \times H_\ell(A)$, with the A part representing the index i. Summing up, the functor G_ℓ that describes elements in each B_i is

$$G_\ell(A) = A + \underbrace{1 + \cdots + 1}_{\substack{\text{number of elements} \\ \text{in } B_0 \text{ that have grade } \ell}} + A \times H_\ell(A).$$

In the rest of this proof, we will view the universe of B_n as being a subset of

$$G(A) = G_0(A) + \cdots + G_m(A),$$

with $G_\ell(A)$ representing the elements of grade ℓ. The polynomial functor $G(A)$ will be the polynomial functor for the MSO interpretation in the conclusion of the lemma. To conclude the proof of the lemma, we need to show that in MSO we can define which elements of $G(A)$ belong to the universe of B_n, and what relations from the output vocabulary are satisfied by tuples of such elements. In other words, we need to define in MSO the quantifier-free theory of tuples from $G(A)$ in the output structure. This is done in the following claim, which completes the proof of the lemma.

Claim C.3. *For every $\ell, k \in \{0, 1, \ldots\}$ the following function is* MSO *definable:*

- **Input.** *A structure $A \in \Sigma^*$ with elements $\bar{a} \in A^k$.*
- **Output.** *The quantifier-free theory of $G(\bar{a})$ in $B_n|\ell$.*

Furthermore, the output depends only on A and $\bar{a}$ restricted to elements of grade at least ℓ.

Proof

Fix some ℓ and k as in the statement of the claim. The claim is proved by induction on ℓ, in reverse order $m, \ldots, 0$. Suppose that we want to prove the claim for some grade ℓ, and assume that it has already been proved for strictly bigger grades.

We use the same idea as in the proof of Claim 3.4. Consider a finite automaton, in which the states are all possible theories that arise by taking some k-tuple $\bar{a}$, and returning the quantifier-free theory of $G(\bar{a})$ in some structure from Γ. This set of states is finite, since the length of the tuple and the vocabulary are fixed.

We will design an automaton with this set of states, together with an input string (which will be called the *advice string*), so that it satisfies the following invariant: after reading the first i letters of the advice string, the state of the automaton is the quantifier-free theory of $G(\bar{a})$ in $B_i|\ell$.

The initial state of the automaton is determined by the invariant, it must be the quantifier-free theory of $G(\bar{a})$ in B_0. Since the universe of B_0 is equal to $G(\varnothing)$, it follows that the initial state does not depend on the tuple $\bar{a}$ or the input structure A.

We now describe the transition function of the automaton, as well as the advice string. By unfolding the definition of the graded MSO interpretation δ, there is some quantifier rank s such that the state of the automaton after reading i letters is uniquely determined by the following four pieces of information:

1. the quantifier-free theory of $G(\bar{a})$ in B_{i-1},
2. the quantifier-free theory of $G(\bar{a})$ in A_i,
3. the rank s MSO theory of $G(\bar{a})$ in $B_{i-1}|\ell + 1$,
4. the rank s MSO theory of $G(\bar{a})$ in $A_i|\ell + 1$.

The first piece of information is the previous state of the automaton. The remaining infomration will be the stored in the advice string; i.e. the i-th letter of the advice string will contain the information described the last three items above. Note that the advice string can be computed in MSO, by the induction assumption. Therefore, since the automaton can be simulated in MSO, it follows that the last state of this automaton can be defined in MSO, thus proving the claim. □

□

D Proof of Theorem 6.1

In this section, we prove that the system in Theorem 5.3 is sound and complete with respect to linear regular functions.

Soundness. The soundness proof follows the same lines as the soundness proof in Theorem 5.3. The general idea is that we use graded MSO interpretations where all components have dimension at most one. This, however, on its own is not going to be enough. To see why, let us compare the two absorption functions

$$\underbrace{!\Sigma \to \Sigma \times !\Sigma}_{\text{not allowed}} \quad \underbrace{!\Sigma \to \Sigma \times \Sigma}_{\text{allowed}}.$$

Both of them have linear size increase – each element of the input structure contributes two copies to the output structure. What is wrong with the function that is not allowed? The problem is that one of the copies has the same grade, and the other has lower grade. In the presence of folding, we can get an unbounded number of copies, by spawning a new lower grade copy in each iteration. This phenomenon will not occur in the allowed function, since both copies have lower grade. The phenomenon discussed above is formalised in the following definition:

Definition D.1. *A* linear graded MSO interpretation *is a graded* MSO *interpretation in which the underlying functor is linear, i.e. all components have dimension one, and which furthermore satisfies the following* downgrading *condition: if an element of the input structure has at least two copies in the output structure, then all of the copies have strictly lower grade.*

In the definition above, the copies of an element in the output structure are defined in the natural way; this definition makes sense when the functor is linear. For example, if the functor is

$$A + A + A + 1 + 1$$

then each input element spawns at most three copies. The components of dimension zero, of which there are two in the above example, are not counted as copies of any input elment.

To prove completeness of the system from Theorem 6.1, we show that all functions that are strongly derived in it are linear graded MSO interpretations. The proof is a simple inducton on the derivation. The most interesting cases are composition and folding. For composition, we simply observe that the condition on lower grades from Definition D.1 is preserved under composition.

We are left with folding. where we use the following lemma, which is the same as Lemma C.2 except that the functions in the assumption and conclusion are required to be linear. In the assumption, we use linearity as defined in Definition D.1, in particular the downgrading condition is assumed; in the conclusion we have an ungraded function, and therefore only the linearity of the functor and not the downgrading condition are assumed.

Lemma D.2. *For every linear graded* MSO *interpretation*

$$\delta : \Gamma \times \Sigma \to \Gamma,$$

between graded list types, and every $B_0 \in \Gamma$, the following function is an (ungraded) linear MSO *interpretation*

$$A = \underbrace{[A_1, \ldots, A_n]}_{\substack{\text{list of structures in } \Sigma,\\ \text{with the grades forgotten}}} \quad \mapsto \quad \underbrace{B_n}_{\substack{\text{defined based on A}\\ \text{as in the proof of Calim 3.4}}}.$$

Proof
We use the same proof as in Lemma C.2. However, there is one difficulty, which is that the functor G defined in that proof is not linear, even if δ is linear. This is because of the product $A \times H_\ell(A)$ which is used to code indexes. In fact, the functor G can have arbitrarily high dimension. However, thanks to the downgrading condition on δ, one show by induction that for every grade ℓ there is some constant $c_\ell \in \{0, 1, \ldots\}$ such that for every grade ℓ element a in the input structure, there are at most c_ℓ elements in the output structure which use a. Here, we say that an element uses a if it belongs to $G(A)$ but not to $G(A \setminus \{a\})$. Using this property, we can turn G into a linear functor. □

This finishes the soundness proof. Below, we give two completeness proofs.

First completeness proof. This proof uses the SST model from Example 7, which is complete for linear regular functions, in the case where the input and output types are strings over finite alphabets [1, Theorem 3]. In Example 7, we show

how to weakly derive every SST that uses each input letter at most once. To get the general form of SST, where an input letter can be used a constant number of times, it is enough to generalize the model from Example 7 so that the initial function is weakly derivable, and the transition function can be derived in type

$$\Delta \times !^k \Sigma \to \Delta$$

for some k. With these relaxations, we get all copyless SST, and retain weak derivability. This proof works only for functions of string-to-string type (admittedly, this is the case that we really care about), and for this reason we also present a second proof, which can also handle types such as strings of strings or pairs of strings.

Second completeness proof. In this proof, similarly to the completeness proof from Theorem 5.3, we reduce to a known complete system. In the case of linear MSO interpretations, the corresponding known system is from [7]. It is the same as in Theorem B.1, except that the split function is removed. In the completeness proof of Theorem 5.3, only the proof for split used general absorption (as opposed to linear absorption). Therefore, the system with linear absorption is complete for the linear regular functions.

This completes the second completeness proof, and thus also the proof of the theorem.