First-order tree-to-tree functions

Mikołaj Bojańczyk
University of Warsaw
Warsaw, Poland
bojan@mimuw.edu.pl

Amina Doumane
CNRS, ENS Lyon
Lyon, France
amina.doumane@ens-lyon.fr

Abstract

We study tree-to-tree transformations that can be defined in first-order logic or monadic second-order logic. We prove a decomposition theorem, which shows that every transformation can be obtained from prime transformations, such as tree-to-tree homomorphisms or pre-order traversal, by using combinators such as function composition.

CCS Concepts: • **Theory of computation → Regular languages**; **Tree languages**; **Logic and verification**.

Keywords: First-order logic, monadic second-order logic, tree transducer, automata

ACM Reference Format:
Mikołaj Bojańczyk and Amina Doumane. 2020. First-order tree-to-tree functions. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '20), July 8–11, 2020, Saarbrücken, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373718.3394785

Acknowledgments

Supported by ERC under the ERC Consolidator Grant LIPA 683080.

1 Introduction

The purpose of this paper is to decompose tree transformations into simple building blocks. An important inspiration is the Krohn-Rhodes theorem [24, p. 454], which says that every string-to-string function recognised by a Mealy machine can be decomposed into certain prime functions.

Regular functions. The transformations studied in this paper are the regular functions.

In [19, Theorem 13], Engelfriet and Hoogeboom proved that deterministic two-way transducers recognise the same string-to-string functions as MSO transductions. Because of this and other properties – such as closure under composition [13, Theorem 1] and decidable equivalence [22, Theorem 1] – this class of functions is now called the *regular string-to-string functions*. Other equivalent descriptions of the regular functions include: string transducers of Alur and Černý [2], and several models based on combinators [4, 10, 16].

There are also regular functions for trees, which can be defined using any of the following equivalent models: MSO tree-to-tree transductions [6, Section 3], single use attributed tree grammars [6], macro tree transducers of linear size increase [18, Theorem 7.1], and streaming tree transducers [3, Theorem 4.6].

The goal of this paper is to prove a decomposition result for regular tree-to-tree functions. As in the Krohn-Rhodes theorem, we want to show that every such function can be obtained by combining certain prime functions.

First-order transductions. Although MSO transductions are the more popular model, we work mainly with the less expressive model of first-order transductions. Why?

As we explain in Section 7, every MSO tree-to-tree transduction can be decomposed as: (a) first, a relabelling defined in MSO, which does not change the tree structure; followed by (b) a first-order tree-to-tree transduction. In this sense, as far as transformations of the tree structure are concerned, first-order and MSO transductions have the same expressive power. Another argument for the importance of first-order tree-to-tree transductions is a connection with the λ-calculus. As we explain in Section 6, first-order tree-to-tree transductions are expressive enough to capture evaluation of λ-terms (assuming linearity, i.e. every variable is used once), and such evaluation turns out to be one of the core computational steps implicit in a tree-to-tree transduction.

Another advantage of first-order logic on trees, compared to MSO, is a better decomposition theory, in the sense of decomposing formulas into simpler ones [7, 20, 23].

For our paper, the most useful decomposition is a remarkable theorem of Schlingloff, which says that first-order logic on trees is equivalent to a certain two-way variant of CTL [26, Theorem 4.5]. In contrast, there are no such results for MSO.

Summing up, we believe that first-order tree transformations are expressive, have a strong theory, and deserve to leave the shadow of their better known MSO cousin.

Structured datatypes. We present our main decomposition result in a formalism based on functional programming

LICS '20, July 8–11, 2020, Saarbrücken, Germany

ACM ISBN 978-1-4503-7104-9/20/07…$15.00
https://doi.org/10.1145/3373718.3394785

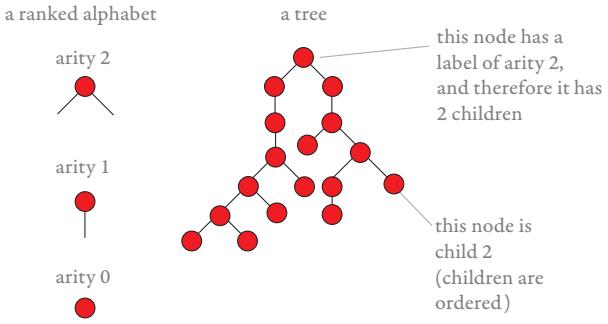
(in a combinatory variant, i.e. without variables), with structured datatypes such as pairs or co-pairs. The motivation behind this approach – which is inspired by [10] – is to avoid encoding datatypes in our constructions using syntactic annotation such as endmarkers and separators. Thanks to the structured datatypes, we can use established operations such as `map`, and we can assign informative types to our functions, such as $\Sigma_1 \times \Sigma_2 \to \Sigma_i$ for projection, as opposed to saying that all functions input and output trees.

The choice of datatypes for trees is harder than for the string case that was studied in [10]. The difficulty is in splitting the input into smaller pieces. A piece of a string is also a string, but this is no longer true for trees, where the pieces have dangling edges (or variables). As a result, more complicated datatypes are needed; and our design choices lead us to functions that operate on ranked sets, where each element has an associated arity.

This is a long paper. Given the limited space, we have decided to prioritise explaining design choices and intuitions, with examples and many pictures. As a result, almost all of the proofs are in the extended version of this paper [11].

2 Trees and tree-to-tree functions

In this section, we describe the trees and tree-to-tree functions that are discussed in this paper. A *ranked set* is a set where each element has an associated *arity* in $\{0, 1, 2, \ldots\}$. If a of a ranked set has arity n, then elements of $\{1, \ldots, n\}$ are called *ports of* a. We adopt the convention that ranked sets are red, e.g. Σ or Γ, and other objects (elements of ranked sets, or unranked sets) are black. We use ranked sets as building blocks for trees. The following picture describes the notion of trees that we use and some terminology:



We use standard tree terminology, such as ancestor, descendant, child, parent. We write treesΣ for the (unranked) set of finite trees over a ranked set Σ. This paper is about *tree-to-tree functions*, of the type

$$f : \text{trees}\Sigma \to \text{trees}\Gamma.$$

2.1 First-order logic and transductions

To define tree-to-tree functions and tree languages, we use logic, mainly first-order logic and monadic second-order logic MSO. The idea is to view a tree as a model, and to use logic to describe properties and transformations of such models.

A *vocabulary* is defined to be a set of relation names, each one with associated arity. We do not use function symbols in this paper. A vocabulary can be formalised as a ranked set, which is why we use red letters like σ or τ for vocabularies.

Definition 2.1 (Tree as a model)**.** For a tree t over a ranked alphabet Σ, its *associated model* is defined as follows. The universe is the nodes of the tree, and it is equipped with the following relations:

$x < y$	x is an ancestor of y	arity 2
$\text{child}_i(x)$	x is an i-th child ($i \in \{1, 2, \ldots\}$)	arity 1
$a(x)$	x has label a ($a \in \Sigma$)	arity 1

The i-th child predicates are only needed for i up to the maximal arity of letters in the ranked alphabet, and hence the vocabulary in the above definition is finite as soon as the ranked alphabet is finite, which will be the case all along the paper. We refer to this vocabulary as *the vocabulary of trees over* Σ. A sentence of first-order logic (or MSO) over this vocabulary describes a tree language, namely the set of trees whose associated models satisfy the sentence. For example, the sentence

$$\forall x\ a(x) \Rightarrow \exists y\ x < y \wedge b(x)$$

is true in (the models associated to) trees t where every node with label a has a descendant with label b. For more background about defining properties of trees using logic, see the survey of Thomas [31].

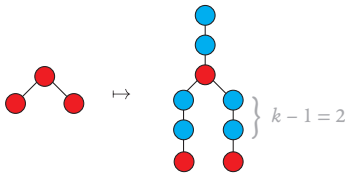
The regular tree languages are exactly those that can be defined in MSO, which was proved by Doner [17, Corollary 3.11], and also Thatcher and Wright [29, p. 74]. The tree languages definable in first-order logic are a proper subset of those definable in MSO, and it is an open problem whether or not one can decide if a regular tree language can be defined in first-order logic [8, Section 3]. This is in contrast to the case of words, where the decidable characterisation of first-order logic by Schützenberger-McNaughton-Papert [25, Theorem 10.5] is a cornerstone of algebraic language theory.

Tree-to-tree functions. Apart from defining tree languages, logic can also be used to define transformations on models. In the context of this paper, we are interested mainly in first-order transductions, defined below. Roughly speaking, a first-order transduction uses first-order logic to define a new tree structure on the input tree.

Definition 2.2 (First-order tree-to-tree transduction)**.** A tree-to-tree function is called a *first-order transduction* if it can be obtained by composing any number of operations[1] of the following two kinds:

[1]There is a normal form of first-order transductions, where at two phases are used: first item 1, then item 2. We do not need the normal form, so we do not prove it, but it can be shown similarly to [15, Section 7.1.5].

1. *Copying.* Let $k \in \{1, 2, \ldots\}$. Define k-copying to be the operation which inputs a tree and outputs a tree where every node is preceded by a chain of $k-1$ unary nodes with a fresh label ●, as in the following picture:



   After k-copying, the number of nodes grows k times.
2. *Non-copying first-order transductions.* This is a tree-to-tree function which uses first-order logic to define a new tree structure over the nodes of the input tree. The syntax of such a transduction is given by:
   a. *Input and output alphabets* Σ and Γ, which are finite ranked sets. We use the name *input vocabulary* for the vocabulary of trees over the input alphabet Σ, likewise we define the *output vocabulary*.
   b. A first-order formula over the input vocabulary, with one free variable, called the *universe formula*.
   c. For each relation of the output vocabulary, of arity n, a corresponding first-order formula over the input vocabulary with n free variables.

   The transduction inputs a tree over the input alphabet, and outputs a tree over the output alphabet where:
   - the nodes are those nodes of the input tree that satisfy the universe formula in item 2b;
   - the labels, descendant, and child relations are defined by the formulas in item 2c.

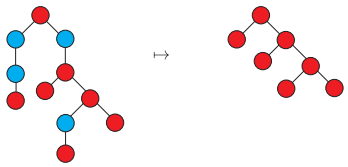
   In order for the transduction to be well defined, the formulas in item 2c must be such that they produce a tree model for every input tree.

If we allowed monadic second-order logic MSO in items 2b and 2c (the free variables of the formulas would still be first-order variables ranging over tree nodes), then we would get the MSO tree-to-tree transductions of Bloem and Engelfriet [6, Section 3]. We discuss these in Section 7.

We conclude this section with two examples of first-order tree-to-tree transductions.

Example 2.3. Let the input and output alphabets be:

and consider the function which removes the unary nodes:

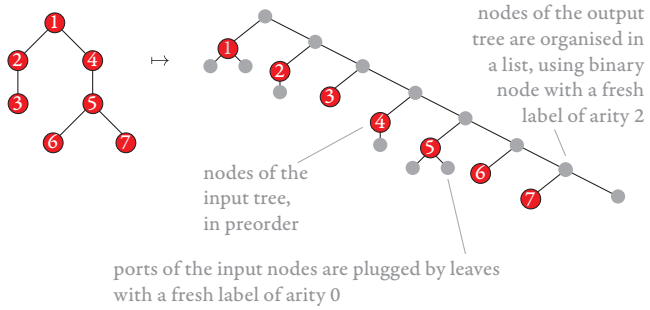


This is a non-copying first-order transduction. The universe formula selects nodes which have non-unary labels. The descendant relation is inherited from the input tree. To define the child relation on the output tree, we use the descendant relation in the input tree. A node x satisfies the unary i-th child predicate in the output tree if it satisfies the following first-order formula in the input tree:

$$\exists y\ \mathrm{child}_i(y) \wedge \underbrace{y \leq x \wedge \forall z\ (y \leq z < x \Rightarrow \bullet(z))}_{\substack{y \text{ is the farthest ancestor that can be} \\ \text{reached from } x \text{ using only unary nodes}}}.$$

This example shows the usefulness of first-order logic with descendant, as opposed to child only as used in [5].

Example 2.4. Define *pre-order* on nodes in a tree as follows: x is before y if either $x \leq y$, or there exist nodes x' and y' such that $x' \leq x$, $y' \leq y$, and x' is a sibling of y' with a smaller child number. Consider the tree-to-tree function which transforms a tree into a list of its nodes in pre-order traversal, as explained in the following picture:



This function is a first-order tree-to-tree transduction, because the pre-order is first-order definable. Unlike Example 2.3, we need copying, because a node of arity n in the input tree corresponds to $n + 2$ nodes in the output tree.

3 Derivable functions

In this section, we state the main result of this paper, which says that the first-order tree-to-tree transductions are exactly those that can be obtained by starting with certain prime functions (such as pre-order traversal from Example 2.4) and applying certain combinators (such as function composition).

The guiding principle behind our approach is to describe tree-to-tree functions without using any iteration mechanisms, such as states or `fold` functions. This principle validates the choice of first-order logic. If we were to use MSO, at the very least we would need to have some mechanism for groups, which are a basic building block for Krohn-Rhodes decompositions, or for evaluating Boolean formulas.

3.1 Datatypes

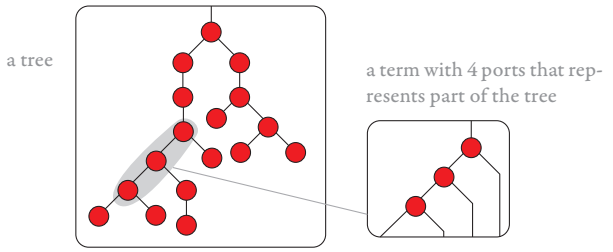
The prime functions and combinators use datatypes such as pairs of trees, or pairs of trees of pairs, etc. Although these

datatypes could be encoded in trees, we avoid this encoding and use explicit datatype constructors.

An important property of our datatypes is that they represent ranked sets, i.e. each element of a datatype has an arity. The datatypes are obtained from the atomic datatypes by applying four datatype constructors, as described below.

Atomic datatypes. Every finite ranked set is an atomic datatype. Apart from finite ranked sets, we allow one more atomic datatype: the *terminal ranked set* $\bot$ which contains exactly one element of every arity. The set is called terminal because it admits a unique arity preserving function from every ranked set. We use $\bot$ for partial functions: a partial function with output type Σ can be seen as a total function of output type $\Sigma + \bot$, which uses $\bot$ for undefined values.

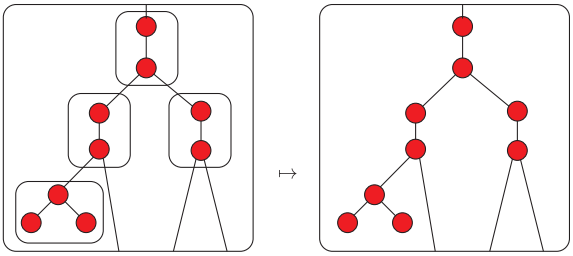
Terms. The central datatype constructor is the *term* constructor, which is a generalisation of trees to higher arities. A term is a tree with dangling edges, called ports. The dangling edges ares used to decompose trees (and other terms) into smaller pieces, as illustrated by the figure below.



Formally speaking, terms are defined by induction as follows. A term over a ranked set Σ is either the *identity term* denoted by $\square$, which consists of a port and nothing else, or otherwise it is an expression of the form $a(t_1, \ldots, t_n)$ where $a \in \Sigma$ has arity n, and $t_1, \ldots, t_n$ are already defined terms. The arity of a term is the number of ports. Terms of arity zero are the same as trees. We write $\mathsf{T}\Sigma$ for the ranked set of terms over a ranked set Σ. Because the term constructor – like other datatype constructors – outputs a ranked set, it makes sense to talk about terms of terms, etc.

Terms are a monad, in the category of ranked sets and arity preserving functions[2]. The unit of the monad, an operation of type $\Sigma \to \mathsf{T}\Sigma$, is illustrated in the following picture:

The product of the monad, an operation of type $\mathsf{TT}\Sigma \to \mathsf{T}\Sigma$ that we call *flattening*, is illustrated in the following picture:



This monad structure will be part of our prime functions.

Products and coproducts. There are two binary datatype constructors

$$\underbrace{\Sigma_1 \times \Sigma_2}_{\text{product}} \qquad \underbrace{\Sigma_1 + \Sigma_2}_{\text{coproduct}}.$$

An element of the product is a pair (a_1, a_2) where $a_i \in \Sigma_i$. The arity of the pair is the sum of arities of its two coordinates a_1 and a_2.

An element of the coproduct is a pair (i, a) where $i \in \{1, 2\}$ and $a \in \Sigma_i$. The arity is inherited from a.

The set of terms can be defined in terms of products and coproducts, as the least solution of the equation:

$$\mathsf{T}\Sigma = \{\square\} + \coprod_{a \in \Sigma} (\mathsf{T}\Sigma)^{\text{arity of } a}$$

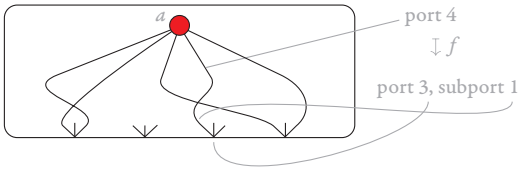
where $\coprod$ denotes possibly infinite coproduct and X^n denotes the n-fold product of a ranked set X with itself.

Folding. The final – and maybe least natural – datatype constructor called *folding*. Folding has two main purposes: (1) reordering ports in a term; and (2) reducing arities by grouping ports into groups.

Folding is not one constructor, but a family of unary constructors $\mathsf{F}_k\Sigma$, one for every $k \in \{1, 2, 3, \ldots\}$. An n-ary element of $\mathsf{F}_k\Sigma$, which is called a *k-fold*, consists of an element $a \in \Sigma$ together with an injective *grouping* function

$$f : \underbrace{\{1, \ldots, \text{arity of } a\}}_{\substack{\text{an element of this set is} \\ \text{called a port of } a}} \to \underbrace{\{1, \ldots, n\} \times \{1, \ldots, k\}}_{\text{these pairs are called } \textit{sub-ports}}$$

We denote such an element as a/f and draw it like this:



Already for $k = 1$, the constructor F_1 is non-trivial. For example, $\mathsf{F}_1\mathsf{T}\Sigma$ is a generalisation of terms where ports are not necessarily ordered left-to-right (because the grouping function need not be monotone), and some ports need not appear (because the grouping function need not be total); in other words this is the same as terms in the usual sense of universal algebra, with the restriction that each variable is used at most once (sometimes called linearity).

[2] An almost identical monad is used in [9, Section 9.2], which differs from ours in that it allows multiple uses of a single port.

- **Function composition.**

$$f \circ g : \Sigma \to \Delta \qquad \text{for } f : \Sigma \to \Gamma \text{ and } g : \Gamma \to \Delta$$

- **Lifting of functions along datatype constructors**

$$\begin{array}{rll} f_1 + f_2 : & \Sigma_1 + \Sigma_2 \to \Gamma_1 + \Gamma_2 & \text{for } \{f_i : \Sigma_i \to \Gamma_i\}_{i=1,2} \\ (f_1, f_2) : & \Sigma_1 \times \Sigma_2 \to \Gamma_1 \times \Gamma_2 & \text{for } \{f_i : \Sigma_i \to \Gamma_i\}_{i=1,2} \\ \mathsf{F}_k f : & \mathsf{F}_k\Sigma \to \mathsf{F}_k\Gamma & \text{for } f : \Sigma \to \Gamma \\ \mathsf{T} f : & \mathsf{T}\Sigma \to \mathsf{T}\Gamma & \text{for } f : \Sigma \to \Gamma \end{array}$$

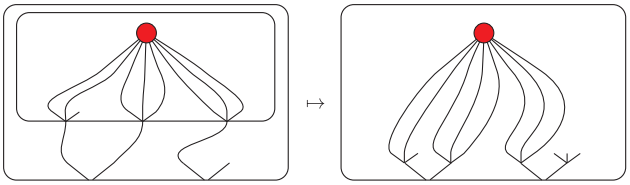
Figure 1. Combinators

When viewed as a family of datatype constructors, folds have a monad-like structure: they are a graded monad in the sense of [21, p. 518]. The unit is the operation

of type $\Sigma \to \mathsf{F}_1\Sigma$, while the product (or flattening) in the graded monad is the family of operations of type

$$\mathsf{F}_{k_2}\mathsf{F}_{k_1}\Sigma \to \mathsf{F}_{k_1 \cdot k_2}\Sigma,$$

indexed by $k_1, k_2 \in \{1, 2, \ldots\}$, that is illustrated below:



More formally, the flattening of a double fold $(a/f_1)/f_2$ has the grouping function defined by

$$i \mapsto (i_2, \pi(p_1, p_2)) \qquad \text{where} \begin{cases} (i_1, p_1) & = f_1(i) \\ (i_2, p_2) & = f_2(i_1) \end{cases}$$

and π is the natural bijection between $\{1, \ldots, k_1\} \times \{1 \ldots, k_2\}$ and $\{1, \ldots, k_1 k_2\}$.

This completes the list of datatype constructors.

Definition 3.1 (Datatypes). The *datatypes* are the least class of ranked sets which contains all finite ranked sets, the terminal set, and which is closed under applying the constructors

$$\mathsf{T}\Sigma \qquad \Sigma_1 \times \Sigma_2 \qquad \Sigma_1 + \Sigma_2 \qquad \mathsf{F}_k\Sigma.$$

3.2 Derivable functions

We now present the central definition of this paper.

Definition 3.2 (Derivable function). An arity preserving function between two datatypes is called *derivable* if it can be generated, by using the combinators in Figure 1, from the following prime functions:

- for every Σ, the unique arity preserving function $\Sigma \to \bot$;
- all arity preserving functions with finite domain;
- the prime functions in Figures 2,3 and 4;

- **Unit and product in the monad T.**

$$\text{unit} : \quad \Sigma \to \mathsf{T}\Sigma \qquad \text{flat} : \quad \mathsf{T}\mathsf{T}\Sigma \to \mathsf{T}\Sigma$$

- **Unit and product in the graded monad F_k.**

$$\text{unit} : \quad \Sigma \to \mathsf{F}_1\Sigma \qquad \text{flat} : \quad \mathsf{F}_k\mathsf{F}_l\Sigma \to \mathsf{F}_{k \cdot l}\Sigma$$

- **Inductive structure of terms (for finite Σ only).**

$$\mathsf{T}\Sigma \underset{\text{compose}}{\overset{\text{decompose}}{\rightleftarrows}} \{\square\} + \coprod_{a \in \Sigma} (\mathsf{T}\Sigma)^{\text{arity of } a}$$

- **Remove unused fold.**

$$\begin{array}{rcl} \mathsf{F}_k\Sigma & \to & \Sigma + \bot \\ (a/f) & \mapsto & a \text{ if } a \text{ has arity 0, undefined otherwise} \end{array}$$

Figure 2. Prime functions for terms and fold.

- **Co-projections.**

$$\begin{array}{rcl} \Sigma + \Sigma & \to & \Sigma \\ (a, i) & \mapsto & a \end{array} \qquad \begin{array}{rcl} \Sigma_i & \overset{\iota_i}{\to} & \Sigma_1 + \Sigma_2 \\ a & \mapsto & (a, i) \end{array}$$

- **Commutativity.**

$$\begin{array}{rcl} \Sigma + \Gamma & \to & \Gamma + \Sigma \\ (a, 1) & \mapsto & (a, 2) \\ (a, 2) & \mapsto & (a, 1) \end{array} \qquad \begin{array}{rcl} \Sigma \times \Gamma & \to & \Gamma \times \Sigma \\ (a, b) & \mapsto & (b, a) \end{array}$$

- **Associativity.**

$$\begin{array}{rcl} (\Sigma + \Gamma) + \Delta & \to & \Gamma + (\Sigma + \Delta) \\ ((a, 1), 1) & \mapsto & (a, 1) \\ ((a, 2), 1) & \mapsto & ((a, 1), 2) \\ (a, 2) & \mapsto & ((a, 2), 2) \end{array} \qquad \begin{array}{rcl} (\Sigma \times \Gamma) \times \Delta & \to & \Sigma \times (\Gamma \times \Delta) \\ ((a, b), c) & \mapsto & (a, (b, c)) \end{array}$$

- **Distributivity.**

$$\begin{array}{rcl} (\Sigma_1 + \Sigma_2) \times \Gamma & \to & (\Sigma_1 \times \Gamma) + (\Sigma_2 \times \Gamma) \\ ((a, i), b) & \mapsto & ((a, b), i) \end{array}$$

Figure 3. Prime functions for product and coproduct.

- **Factorisations.**

$$\mathsf{T}(\Sigma_1 + \Sigma_2) \underset{\text{fact}_\downarrow}{\overset{\text{fact}_\uparrow}{\rightrightarrows}} \mathsf{T}(\mathsf{T}\Sigma_1 + \mathsf{T}\Sigma_2)$$

- **Pre-order.** (Here $\bullet$, $\frown$ are letters of arities 0, 2).

$$\mathsf{T}\Sigma \longrightarrow \mathsf{F}_1\mathsf{T}(\Sigma + \{\bullet, \frown\})$$

- **Monotone unfolding.**

$$\mathsf{T}\mathsf{F}_k\Sigma^k \longrightarrow \mathsf{F}_k(\mathsf{T}\Sigma)^k + \bot$$

Figure 4. Functions explained in Section 3.3.

The combinators in Figure 1 are function composition, and the obvious liftings of functions along the datatype constructors. The prime functions in Figure 2 describe the monad structure of terms and folds, and were explained in Section 3.1. The prime functions in Figure 3 are simple syntactic transformations, which are intended to have no computational content. Figure 4 contains less obvious operations, whose definitions are deferred to Section 3.3.

Example 3.3. Define a *term homomorphism* to be any function of type $T\Sigma \to T\Gamma$ which is obtained by applying some function

$$h : \Sigma \to T\Gamma$$

to every node of the input term. Examples of term homomorphisms include the function from Example 2.3 which removes all unary letters, or the k-copying function in item 1 of the definition of first-order tree-to-tree transductions. We claim that every term homomorphism with a finite input alphabet is derivable. The function h is a prime function, because it has a finite domain thanks to the assumption that the input alphabet is finite. We can lift h to terms using the combinator of Figure 1, and then compose it with the product operation of terms monad, thus giving the homomorphism:

$$T\Sigma \xrightarrow{Th} TT\Gamma \xrightarrow{\text{flat}} T\Gamma$$

We are now ready to state the main theorem of this paper. We say that a tree-to-tree function

$$f : \text{trees}\Sigma \to \text{trees}\Gamma$$

is *derivable* if it agrees on arguments that are trees with some derivable partial function

$$f : T\Sigma \to T\Gamma + \bot.$$

The main result of this paper is the following theorem.

Theorem 3.4. *A tree-to-tree function is a first-order transduction if and only if it is derivable.*

The right-to-left implication in the above theorem is proved by a relatively straightforward induction on the derivation. The general idea is that we associate to each datatype a relational structure; for example the relational structure associated to a pair (a_1, a_2) is the disjoint union of the relational structures associated to a_1 and a_2. In the long version of this paper [11], we show that all prime functions are first-order transductions (adapted suitably to structures other than trees); and that this property is preserved under applying the combinators. There is one non trivial step in the proof, which concerns monotone unfolding, and will be discussed below.

The left-to-right implication in the theorem, which says that every first-order transduction is derivable, is the main contribution of this paper, and is discussed in Sections 4–6.1.

3.3 The prime functions from Figure 4

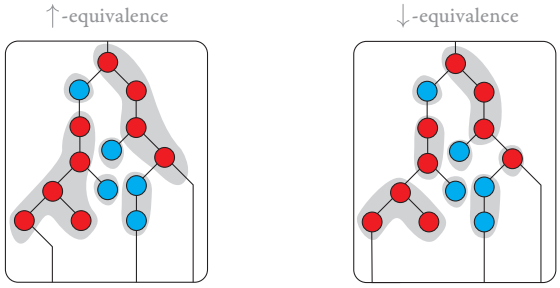
In this section, we describe the prime functions from Figure 4. Each of these functions will play a key role in one of the main results of the paper.

3.3.1 Factorisations. We begin with the two factorisation functions

$$\text{fact}_\uparrow, \text{fact}_\downarrow : T(\Sigma_1 + \Sigma_2) \to T(T\Sigma_1 + T\Sigma_2),$$

which are used to cut terms into smaller parts. Define a *factorisation* of a term to be any term of terms that flattens to it. An alternative view is that a factorisation is an equivalence relation on nodes in a term, where every equivalence class is connected via the parent-child relation.

Consider a term $t \in T(\Sigma_1 + \Sigma_2)$. We say that two nodes have the *same type* if both have labels in the same Σ_i; otherwise we say that nodes have *opposing type*. Define two equivalence relations on nodes in a term as follows: (a) nodes are called $\uparrow$-*equivalent* if they have the same type and the same proper ancestors of opposing type; (b) nodes are called $\downarrow$-*equivalent* if they are $\uparrow$-equivalent and have the same proper descendants of opposing type. Here is a picture of the equivalence classes, with Σ_1 being red and Σ_2 being blue:

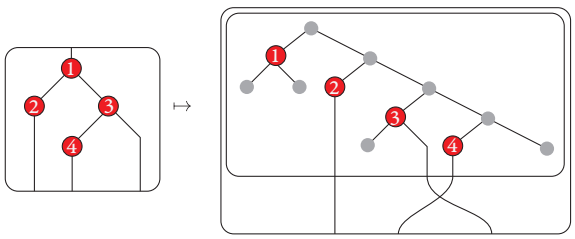


For both equivalence relations, the equivalence classes are connected under the parent-child relation, and therefore the equivalences can be seen as factorisations. These are the factorisations produced by the functions $\text{fact}_\uparrow$ and $\text{fact}_\downarrow$.

3.3.2 Pre-order traversal. The pre-order traversal function

$$\text{preorder} : T\Sigma \to F_1 T(\Sigma + \{\bullet, \curvearrowright\})$$

is the natural extension – from trees to terms – of the pre-order function in Example 2.4. The fold in the output type is used to reorder the ports in a way which matches the input term, as illustrated in the following picture:



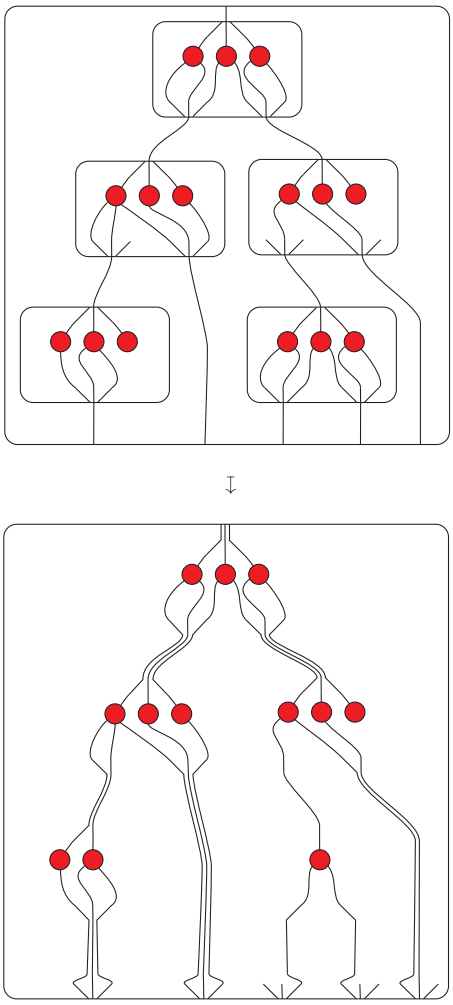
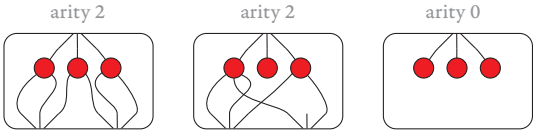


Figure 5. Unfolding the matrix power

3.3.3 Unfolding of the matrix power. The final prime function is called monotone unfolding. The general idea is that unfolding unpacks a representation of several trees inside a single tree. Before describing this function in more detail, we introduce some notation, inspired by the matrix power in universal algebra [28, p. 268].

Definition 3.5 (Matrix power). For $k \in \{1, 2, \ldots\}$ define the k-th matrix power of a ranked set Σ, denoted by $\Sigma^{[k]}$, to be the ranked set $\mathsf{F}_k \Sigma^k$.

Here is a picture of elements in the third matrix power:



An element of the k-th matrix power can be seen as having a group of k incoming edges, and each of its ports can be seen as a group of k outgoing edges. The *general unfolding* operation, which has type

$$\mathsf{T}\Sigma^{[k]} \to (\mathsf{T}\Sigma)^{[k]},$$

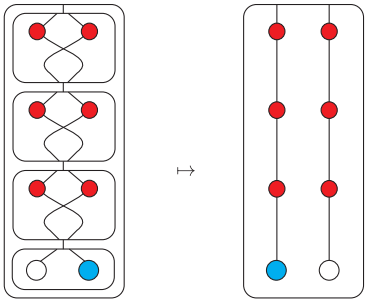
matches the k incoming edges in a node with the k outgoing edges in the parent port; it also removes the unreachable nodes. This operation is illustrated in Figure 5, and a formal definition is in [11].

Chain logic. The general unfolding operation is too powerful to be included in the derivable functions, as we explain below. It does, however, admit a characterisation in terms of a fragment of MSO called *chain logic*, see [30, Section 2] or [7, Section 2.5.3], whose expressive power is strictly between first-order logic and MSO. Chain logic is defined to be the fragment of MSO where set quantification is restricted to sets where all nodes are comparable by the descendant relation.

Theorem 3.6. *The following conditions are equivalent for tree-to-tree functions:*

- *is derivable, as in Definition 3.2, except that general unfold is used instead of monotone unfold;*
- *is a transduction, as in Definition 2.2, except that chain logic is used instead of first-order logic.*

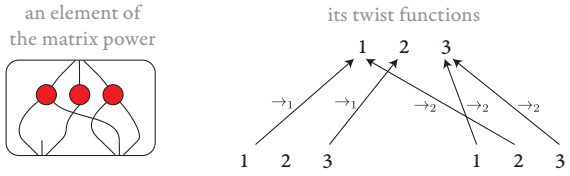
To see why chain logic is needed to describe general unfolding, consider the following unfolding, where two coordinates are swapped in each node of the input tree:



For inputs with an odd number of swaps, the output of unfolding has a white leaf in the first coordinate, and for inputs with an even number of swaps, the output has a white leaf in the first coordinate. Checking if a path has even length can be done in chain logic, but not in first-order logic.

Monotone unfolding. To avoid the problems with cyclic swaps, the unfolding function in Figure 4 imposes a monotonicity requirement on the matrix power, described below.

Let $a \in \Sigma^{[k]}$ be an element of the matrix power, let $p, q \in \{1, \ldots, k\}$, and let i be a port of a. Define the *twist function of port i*, denoted by $\to_i$, as follows: $q \to_i p$ if coordinate q in the i-th outgoing edge is connected to coordinate p in root, as described in the following picture:



The twist function is partial. Call an element of the matrix power *monotone* if for every port, its twist functions is monotone (when restricted to inputs where it is defined). In the picture above, $\to_1$ is monotone, while $\to_2$ is not. Also, the problems with an even number of swaps discussed earlier arise from a non-monotone twist function:

The *monotone unfolding* operation in Figure 5 defined to be the restriction of general unfolding, which is undefined if the input contains at least one label which is non-monotone, and otherwise returns the output of the general unfolding.

Is unfolding derivable? The prime functions in our main theorem are meant to be simple syntactic rewritings. It is debatable whether the unfolding operation – even in its monotone variant – is of this kind. For example, our proof that monotone unfolding is a first-order transduction requires an invocation of the Schützenberger-McNaughton-Papert theorem about first-order logic on words being the same as counter-free automata.

Is it possible to break down monotone unfolding into simpler primitives? In the long version of this paper [11], we devote considerable resources to answering this question. We propose one new datatype

and seventeen additional prime functions, which can be called syntactic rewriting without straining the reader's patience. Then, we show that monotone unfolding can be derived using the new datatype and functions. The proof of this result is one of the main technical contributions of this paper.

4 Register tree transducers

We now begin the proof of the harder implication in Theorem 3.4, which says that every first-order tree-to-tree transduction is derivable. Our proof passes through an automaton model, which is roughly based on existing transducer models for MSO transductions from [3, 6]. The automaton uses registers to store parts of the output tree. The semantics of the automaton involves two phases: (a) mapping the input tree to an expression that uses register updates; (b) evaluating the expression. These phases are described in more detail below.

Register valuations and updates. We begin by explaining how the registers work. The registers store terms that are used to construct the output tree. Each register has an arity: registers of arity zero store trees, registers of arity one store unary terms, etc.

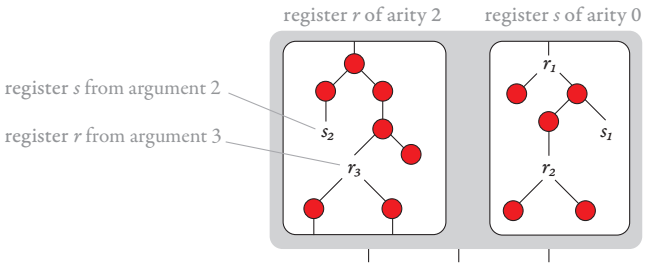
Fix two finite ranked sets: the *register names* R and the *output alphabet* Γ. A *register valuation* is defined to be any arity preserving function from the register names R to terms $\mathsf{T}\Gamma$.

To transform register valuations, we use *register updates*. A register update is an operation which inputs several register valuations and outputs a single register valuation. For $n \in \{0, 1, \ldots\}$, an *n-ary register update* is defined to be any arity-preserving function

$$u : R \to \mathsf{T}(\Gamma + nR),$$

where nR stands for the disjoint union of n copies of R. The i-th copy of R represents the register contents in the i-th argument.

Here is a picture of a register update which has arity 3 and uses two registers r and s:



An n-ary register update u induces a operation, which inputs n register valuations and outputs the register valuation obtained by taking u and replacing the i-th copy of a register name with the contents of that register in the i-th input register valuation. Register updates have arities, and therefore the ranked set of register updates is written in red, and can be used for labels in a tree. For such a tree

$$t \in \text{trees}(\text{register updates}),$$

define its *evaluation* to be the register valuation defined by induction in the natural way. Note that register updates of arity zero are the same as register valuations, which gives the induction base.

First-order relabellings. Our automaton model has no states. Instead, it uses a first-order relabelling, as defined below, to directly assign to each node of the input tree a register update that will be applied in that node. A similar model is used by Bloem and Engelfriet [6, Theorem 17], except that in their case, the first phase uses MSO relabellings, and the second phase is an attribute grammar.

Definition 4.1 (First-order relabelling). A *first-order relabelling* is given by two finite ranked sets Σ and Γ, called the *input and output alphabets*, and a family

$$\{\varphi_a(x)\}_{a \in \Gamma}$$

of first-order formulas over the vocabulary of trees over Σ. These formulas need to satisfy the following restriction:

(*) for every tree over the input alphabet and node in that tree, there is a unique output letter $a \in \Gamma$ such that $\varphi_a(x)$ selects the node; furthermore, the arity of a is the same as the arity of (the label of) the node.

The semantics of a first-order tree relabelling is a function

$$\text{trees}\Sigma \to \text{trees}\Gamma,$$

which changes the label of every node in the input tree to the unique letter described in (*).

A first-order tree relabelling is a very special case of a first-order tree-to-tree transduction, where only the labelling of the input tree is changed, while the universe as well as the child and descendant relations are not affected.

Register transducers. Having defined registers, register updates, and first-order tree relabellings, we are now ready to define our automaton model.

Definition 4.2 (First-order register transducer)**.** The syntax of a *first-order register transducer* consists of:

- An *input alphabet* Σ, which is a finite ranked set;
- An *output alphabet* Γ, which is a finite ranked set;
- A set R of *registers*, which is a finite ranked set;
- A total order on the registers.
- A designated *output register* in R, of arity zero.
- A *transition function*, which is a first-order relabelling

$$\text{trees}\Sigma \to \text{trees}\Delta,$$

for some finite set Δ of register updates over registers R and output alphabet Γ. We require all register updates in Δ to be single-use and monotone, as defined below:

1. *Single-use*[3]. An n-ary register update u is called *single-use* if every $r \in nR$ appears in at most one term from $\{u(s)\}_{s \in R}$, and it appears at most once in that term.
2. *Monotone*[4]. This condition uses the total order on registers. An n-ary register update u is called monotone if for every $i \in \{1, \ldots, n\}$, the binary relation $\to_i$ on register names $r, s \in R$ defined by

$$r \to_i s \quad \text{if} \quad \text{the } i\text{-th copy of } r \text{ appears in } u(s),$$

which is a partial function from r to s when u is single-use, is monotone:

$$r_1 \leq r_2 \wedge r_1 \to_i s_1 \wedge r_2 \to_i s_2 \quad \Rightarrow \quad s_1 \leq s_2$$

The semantics of the transducer is a tree-to-tree function, defined as follows. The input is a tree over the input alphabet. To this tree, apply the transition function, yielding a tree of register updates. Next, evaluate the tree of register updates, yielding a register valuation. The output tree is defined to be the contents of the designated output register.

The main difference of our model with respect to prior work is that we want to capture tree transformations defined in first-order logic, as opposed to MSO used in [1, 3, 6]. This is why we use first-order relabellings instead of MSO relabellings. For the same reason, we require the register updates to be monotone, see the discussion in Section 3.3.3.

The main result of this section is that first-order register transducers are expressively complete for first-order tree-to-tree transductions.

Theorem 4.3. *Every first-order tree-to-tree transduction is recognised by a first-order register transducer.*

The proof, which can be found in [11], uses the composition method for logic, like similar proofs for [3, Theorem 4.6] and [6, Theorem 14]. The converse inclusion in the theorem is also true. This is can be shown directly without much difficulty, following the same lines as in [6, Section 5]. The converse inclusion also follows from other results in this paper: (a) we show in the following sections that every function computed by the transducer is derivable; and (b) derivable functions are first-order tree-to-tree transductions by the easy implication in Theorem 3.4.

Proof strategy for Sections 5–6. By Theorem 4.3, to prove derivability of every first-order tree-to-tree transduction, and thus finish the proof of our main theorem, it suffices to prove derivability for first-order register transducers. In a first-order register transducer, the computation has two steps: a first-order relabelling, followed by evaluation of the register updates. The first step is handled in Section 5, and the second step is handled in Section 6.

5 First-order relabellings

In this section we prove derivability of the first computation step used in first-order register transducers.

Proposition 5.1. *Every first-order relabelling is derivable.*

To prove the proposition, we use a decomposition of first-order relabellings into simpler functions, in the style of the Krohn-Rhodes theorem.

We use the name *unary query* for a first-order formula with one free variable over the vocabulary of trees. This assumes some implicit alphabet Σ. For a unary query, define its *characteristic function*, of type

$$\text{trees}\Sigma \to \text{trees}(\Sigma + \Sigma),$$

to be the function which replaces the label of each node by its first or second copy, depending on whether the node is selected by the query. This is a special case of a first-order relabelling. The key to Proposition 5.1 is the following lemma, which decomposes first-order relabellings into characteristic functions of certain basic unary queries.

[3] The single-use restriction is a standard feature of transducer models with linear size increase [1, 3, 6]. It prohibits iterated duplication of registers, which would lead to exponential size outputs.

[4] This is notion of monotonicity corresponds to the one used in Section 3.3.3, see the comments on page 12. A similar notion appears in [10, p. 7].

Lemma 5.2. *Every first-order relabelling can be obtained by composing the following functions:*

1. Letter-to-letter homomorphisms. *For every finite* Γ, Σ *and* $f : \Sigma \to \Gamma$, *its tree lifting* $\mathrm{trees}f : \mathrm{trees}\Sigma \to \mathrm{trees}\Gamma$.
2. *For every finite* Σ *and its subsets* $\Delta, \Gamma \subseteq \Sigma$, *the characteristic functions of the following unary queries over alphabet* Σ:
   a. Child: x *is an* i*-th child, for* $i \in \{1, 2, \ldots\}$
   $$\mathrm{child}_i(x);$$
   b. Until: x *has a descendant* y *with label in* Δ, *such that all nodes strictly between* x *and* y *have label in* Γ
   $$\exists y\ y > x \wedge \Delta(y) \wedge \forall z\ (x < z < y \Rightarrow \Gamma(z));$$
   c. Since: x *has an ancestor* y *with label in* Δ, *such that all nodes strictly between* x *and* y *have label in* Γ
   $$\exists y\ y < x \wedge \Delta(y) \wedge \forall z\ (y < z < x \Rightarrow \Gamma(z)).$$

The lemma uses a theorem of Schlingloff [26, Theorem 2.6], which says that all first-order definable tree properties can be defined using a temporal logic with operators similar to the ones used in items 2 of the lemma. Note that the temporal logic is a two-way logic, because *until* depends on the descendants of the node x, while *since* depends on the ancestors. In fact, there is no temporal logic which characterises first-order logic, uses only descendants, and has finitely many operators [12, Theorem 5.5]. The exact reduction to Schlingloff's theorem can be found in [11].

It remains to show that all of the functions from Lemma 5.2 are derivable. The letter-to-letter homomorphisms from item 1 are a special case of homomorphisms discussed in Example 3.3, and hence derivable. In the long version of the paper [11], we show that the functions from item 2 are also derivable. In the proof, a key role is played by the factorisation functions discussed in Section 3.3.1.

6 Evaluation of register updates

In this section, we deal with the second computation phase in a first-order register transducer, namely evaluating register updates. As discussed in the end of Section 4, this completes the proof of our main theorem.

Our proof uses the language of λ-calculus. In Section 6.1, we discuss derivability of normalisation of λ-terms. In Section 6.2, we reduce evaluation of register updates to unfolding the matrix power and normalisation of λ-terms.

6.1 Normalisation of simply typed linear λ-terms

We assume that the reader is familiar with the basic notions of the simply typed λ-calculus; more detailed definitions can be found in [27]. Define *simple types* to be expressions generated from an atomic type o using a binary arrow constructor, as in the following examples:

$$o \qquad o \to o \qquad (o \to o) \to (o \to o) \qquad \cdots$$

In this paper, the atomic type o represents trees over the output alphabet. Let X be a set of variables, each one with an associated simple type. A λ-term is any expression that can be built from the variables, using λ-abstraction $\lambda x.M$ and term application MN. We say that a λ-term is *well-typed* if one can associate to it a simple type according to the usual typing rules of simply typed λ-calculus, see [27, Definition 3.2.1]. Because the variables are typed, a λ-term has either a unique type, or is not well-typed. Here is an example of a well-typed λ-term, with the type annotation in blue:

$$\overbrace{\lambda y^{o\to o}.\ \lambda x^{o}.\ \underbrace{y(yx)}_{o}.}^{(o\to o)\to o\to o}$$

We use the standard notion of β-reduction for λ-terms, see [27, Definition 1.2.1]. Because of normalisation and confluence for the simply typed λ-calculus, every well-typed λ-term has a unique normal form, i.e a λ-term to which it β-reduces (in zero or more steps), and which cannot be further β-reduced.

A λ-term can be seen as a tree over the ranked alphabet

$$\overbrace{\{x : x \in X\}}^{\text{arity } 0} \cup \overbrace{\{\lambda x : x \in X\}}^{\text{arity } 1} \cup \overbrace{\{@\}}^{\text{arity } 2} \tag{1}$$

where @ represents term application. Using this representation, and assuming that the set of variables is finite, it makes sense to view normalisation as a tree-to-tree function

$$\lambda\text{-term} \qquad \mapsto \qquad \text{its normal form},$$

and ask about its derivability. We show that this function is derivable, under two assumptions on the input λ-term.

The first assumption is that the input λ-term is *linear*: every bound variable is exactly once in its scope[5], but free variables are allowed to appear multiple times.

The second assumption is that the input λ-term can be typed using a fixed finite set of types $\mathcal{T}$: it has type in $\mathcal{T}$, and the same is true for all of its sub-terms. In [11], we explain why the assumptions are needed.

Theorem 6.1. *Let* X *be a finite set of simply typed variables, and let* $\mathcal{T}$ *be a finite set of simple types. The following tree-to-tree function is derivable, assuming that* λ*-terms are represented as trees:*

- **Input.** *A* λ*-term over variables* X.
- **Output.** *Its normal form, if it is linear and can be typed using* $\mathcal{T}$, *and undefined otherwise.*

This is one of our main technical contributions, and its proof is in the extended version of the paper [11]. A key role in the proof is played by the pre-order function.

[5] This restriction could easily be relaxed to "at most once".

6.2 Evaluation of register updates

Equipped with Theorem 6.1, we prove derivability of evaluation of register updates. Fix a first-order register transducer.

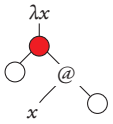
From now on, when speaking about register updates or register valuations, we mean those of the fixed transducer. Our goal is to prove the following lemma, which completes the proof of our main theorem.

Lemma 6.2. *Consider the tree-to-tree function, which inputs a tree of register updates, evaluates it, and outputs the contents of the designated output register. This function is derivable.*

Output letters in λ-terms. We will use λ-terms to represent register updates, which involve letters of the output alphabet Γ. Therefore, for the rest of Section 6.2, we use an extended notion of λ-terms, which allows building λ-terms of the form

$$a(M_1, \ldots, M_n) \qquad \text{for every } a \in \Gamma \text{ of arity } n. \tag{2}$$

The typing rules are extended as follows: if the arguments $M_1, \ldots, M_n$ all have type o (no other type is allowed for arguments of a), then (2) has type o. These λ-terms can be represented as trees, as in the following picture:



Theorem 6.1 works without change for the extended notion of λ-terms used in this section. Note that there is no β-reduction rule for λ-terms of the form (2).

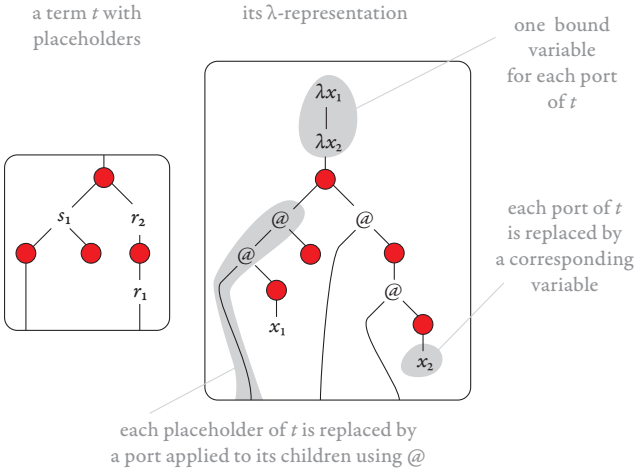
λ-representations of register updates. To prove Lemma 6.2, we represent register updates using a matrix power of λ-terms. The idea is that the matrix power handles the parallel evaluation of registers.

Let X be a set of variables $\{x_1 \ldots, x_m\}$, all of them having type o, where m is the maximal arity among registers. Define Γ_λ to be the output alphabet Γ plus the ranked alphabet defined in (1) for tree representations of λ-terms.

Recall that a register update – of arity say n – consists of a family of terms over alphabet $\Gamma + nR$, one for each register $r \in R$. We begin by explaining the λ-representation for terms in the family, which is a function of type

$$\mathsf{T}(\Gamma + nR) \xrightarrow{\lambda\text{-representation}} \mathsf{T}\Gamma_\lambda \ . \tag{3}$$

This function is not arity preserving, which is why it is not written in red. Define a *placeholder* to be an element of nR; we write placeholders as r_i with $r \in R$ and $i \in \{1, \ldots, n\}$. The function (3) is explained in the following picture:



Note how the arities need not be preserved: the arity of the output is the number of placeholders in the input, which need not be the same as the number of ports in the input. The correspondence of ports in the output term with placeholders in the input term is defined with respect to some arbitrary order on the set nR of placeholders, say lexicographic with respect to the order on registers and $\{1, \ldots, n\}$.

Having defined the λ-representation of terms with placeholders, we lift it a λ-representation of register updates

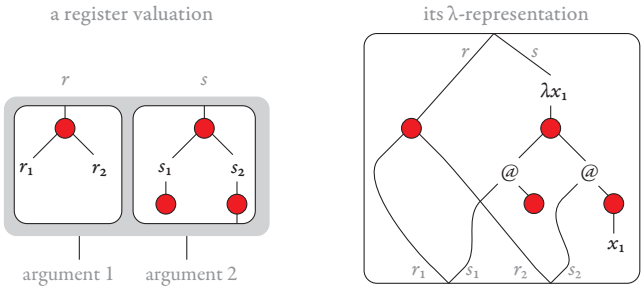
$$\text{register updates} \xrightarrow{\lambda\text{-representation}} (\mathsf{T}\Gamma_\lambda)^{[k]} \ , \tag{4}$$

where k is the number of registers. This function is arity preserving.

For a register update $(t_1, \ldots, t_k)$, where t_i is the term with placeholders used in the i-th register, its λ-representation is defined to be

$$(\lambda\text{-representation of } t_1, \ldots, \lambda\text{-representation of } t_k)/f,$$

where the grouping function f connects a placeholder r_i to the r-th sub-port of port i. Here is a picture



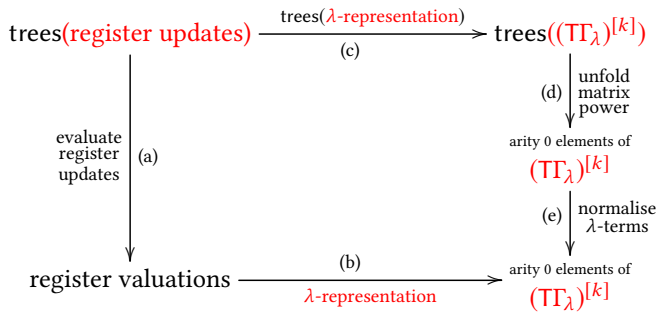


Figure 6

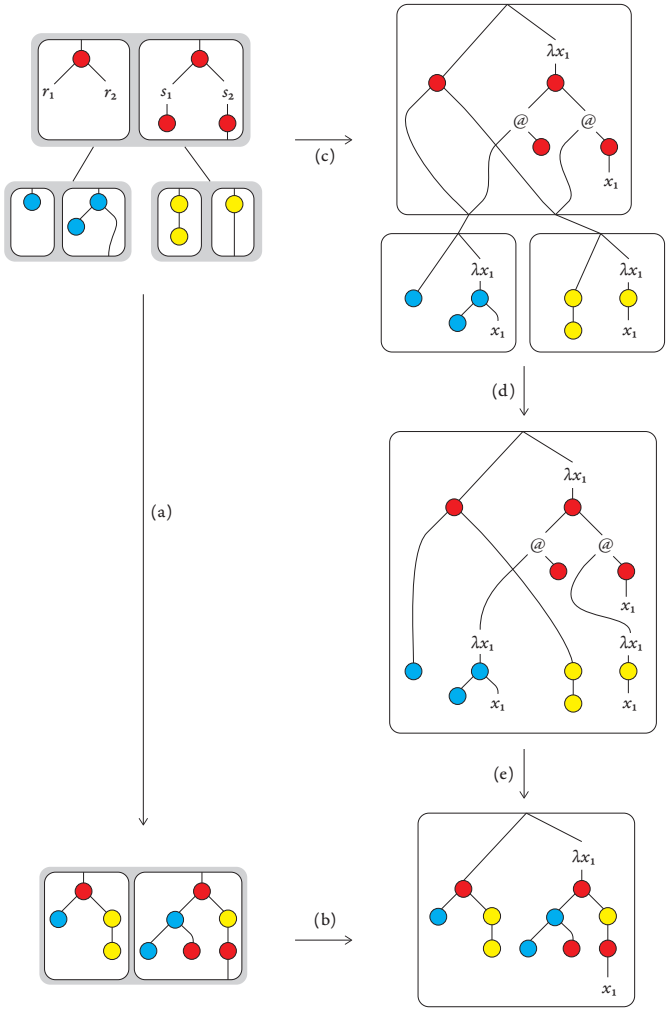


Figure 7. Example for Figure 6.

The following three properties of the λ-representation for register updates will be used later in the proof:

(P1) If we restrict the domain to a finite set of register updates, e.g. those used in the transducer, then it is a prime function, by virtue of having finite domain.
(P2) A register update is monotone (as in Definition 4.2) if and only if its λ-representation is monotone (as defined in Section 3.3.3 for the matrix power).
(P3) Every bound variable in the λ-representation is used exactly once, and the types that appear are of the form

$$\overbrace{o \to o \to \cdots \to o \to o}^{\text{at most (maximal arity in } \Gamma\text{) times}} \to o,$$

hence Theorem 6.1 can be applied.

Putting it all together. To finish the proof of Lemma 6.2, we observe that the semantics of a register automaton are translated – under the λ-representation – to unfolding the matrix power and normalising a λ-term. This observation is formalised by saying that the diagram in Figure 6 commutes, and it follows directly from the definitions. Instead of giving a proof, we illustrate it on an example in Figure 7.

We claim that all of the arrows (c), (d) and (e) on the right-down path in Figure 6 are derivable:

(c) Since we work with a fixed register transducer, there is a finite subset Δ of register updates used, and therefore operation (a) in the figure is derivable by property (P1).
(d) Arrow (d) represents the unfolding of the matrix power. By property (P2), the outputs of arrow (c) are monotone, and so we can use the monotone unfolding operation, which is a prime function and therefore derivable.
(e) Finally, arrow (e) represents normalisation of λ-terms. This arrow is derivable by Theorem 6.1. The assumptions of this theorem are met by property (P3).

Since the arrows (c), (d), (e) are derivable, and the diagram commutes, it follows that the composition of the arrows (a) and (b) is derivable. In other words, there is a derivable function which maps a tree of register updates to the λ-representation of the resulting register valuation (when viewing a register valuation as a special case of a register

update of arity zero). Finally, to get the contents of the output register, we get rid of the fold in the matrix power by using the last function from Figure 2, and project onto the coordinate for the output register.

This completes the proof of Lemma 6.2, and therefore also of the main theorem.

7 Monadic second-order transductions

We finish the paper by discussing a variant of our main theorem for MSO tree-to-tree transductions. We simply add, as prime functions, all MSO relabellings, which are defined the same way as the first-order relabellings from Definition 4.1, except that the unary queries can use MSO logic instead of first-order logic.

Theorem 7.1. *A tree-to-tree function is an* MSO *transduction if and only if it can be derived using Definition 3.2 extended by adding all* MSO *relabellings as prime functions.*

Proof. In [14, Corollary 1], Colcombet shows that every MSO formula on trees can be replaced by a first-order formula that runs on an MSO relabelling of the input tree. Applying that result to transductions, we see that every MSO tree-to-tree transduction can be decomposed as: (a) an MSO relabelling; followed by (b) a first-order tree-to-tree transduction. The theorem follows. □

The solution above is not particularly subtle, and contrasts our results for first-order logic and chain logic, where we took care to have a small number of primitives. This was possible thanks in part to the decomposition of first-order queries into simpler ones that was is in Section 5, and the Krohn-Rhodes theorem that is used in the proof of Theorem 3.6 about chain logic. In principle, a decomposition of MSO relabellings could be possible, but proving it would likely require developing a new decomposition theory for regular tree languages, in the style of the Krohn-Rhodes theorem, which we feel is beyond the scope of this paper. One would expect a Krohn-Rhodes theorem for trees to yield an effective characterisation of first-order logic – as it does for words – but finding such a characterisation remains a major open problem [8, Section 3].

References

[1] Rajeev Alur. Streaming String Transducers. In *Workshop on Logic, Language, Information and Computation, WoLLIC 2011, Philadelphia, USA*, volume 6642 of *Lecture Notes in Computer Science*, page 1. Springer, 2011.

[2] 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.

[3] Rajeev Alur and Loris D'Antoni. Streaming tree transducers. *Journal of the ACM (JACM)*, 64(5):31, 2017.

[4] 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.

[5] Michael Benedikt and Luc Segoufin. Regular tree languages definable in FO and in FO_{mod}. *ACM Trans. Comput. Log.*, 11(1):4:1–4:32, 2009.

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

[7] Mikołaj Bojańczyk. *Decidable Properties of Tree Languages*. PhD Thesis, University of Warsaw, 2004.

[8] Mikołaj Bojańczyk. Some open problems in automata and logic. *ACM SIGLOG News*, 2(4):3–15, 2014.

[9] Mikołaj Bojańczyk. Recognisable languages over monads. *CoRR*, abs/1502.04898, 2015.

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

[11] Mikołaj Bojańczyk and Amina Doumane. First-order tree-to-tree functions. *CoRR*, abs/2002.09307, 2020.

[12] Mikołaj Bojańczyk, Howard Straubing, and Igor Walukiewicz. Wreath Products of Forest Algebras, with Applications to Tree Logics. *Logical Methods in Computer Science*, 8(3), 2012.

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

[14] Thomas Colcombet. A Combinatorial Theorem for Trees. In *International Colloquium on Automata, Languages and Programming, ICALP, Wrocław, Poland*, Lecture Notes in Computer Science, pages 901–912. Springer, 2007.

[15] Bruno Courcelle. The monadic second-order logic of graphs v: on closing the gap between definability and recognizability. *Theoretical Computer Science*, 80(2):153 – 202, 1991.

[16] Vrunda Dave, Paul Gastin, and Shankara Narayanan Krishna. Regular transducer expressions for regular transformations. In *Logic in Computer Science, LICS 2018, Oxford, UK,*, pages 315–324, 2018.

[17] John Doner. Tree acceptors and some of their applications. *J. Comput. System Sci.*, 4:406–451, 1970.

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

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

[20] Z. Ésik and P. Weil. On logically defined recognizable tree languages. In *FSTTCS*, volume 2914 of *LNCS*, pages 195–207, 2003.

[21] Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. Towards a formal theory of graded monads. In *Foundations of Software Science and Computation Structures, FoSSaCS, Eindhoven, the Netherlands*, Lecture Notes in Computer Science, pages 513–530. Springer, 2016.

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

[23] Thilo Hafer and Wolfgang Thomas. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In *International Colloquium on Automata, Languages and Programming, ICALP, Turku, Finland*, pages 269–279. Springer, 1987.

[24] 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.

[25] Robert McNaughton and Seymour Papert. *Counter-free automata*. The M.I.T. Press, Cambridge, Mass.-London, 1971.

[26] Bernd-Holger Schlingloff. Expressive completeness of temporal logic of trees. *Journal of Applied Non-Classical Logics*, 2(2):157–180, 1992.

[27] Morten Heine Sorensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, July 2006.

[28] Walter Taylor. The fine spectrum of a variety. *Algebra Universalis*, 5(1):263–303, 1975.

[29] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical systems theory*, 2(1):57–81, March 1968.

[30] Wolfgang Thomas. Infinite trees and automaton- definable relations over ω-words. *Theoretical Computer Science*, 103(1):143 – 159, 1992.

[31] Wolfgang Thomas. Languages, automata, and logic. In *Handbook of formal languages*, pages 389–455. Springer, 1997.