

# First-order tree-to-tree functions

Mikołaj Bojańczyk and Amina Doumane

## 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.

## 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, 11, 16].

There are also regular functions for trees, which can be defined using any of the following equivalent models: MSO tree-to-tree transductions [7, Section 3], single use attributed tree grammars [7], 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  $\lambda$ -calculus. As we explain in Section 6, first-order tree-to-tree transductions are expressive enough to capture evaluation of  $\lambda$ -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 [8, 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 (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 [11] – 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  $\Sigma_1 \times \Sigma_2 \rightarrow \Sigma_i$  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 [11]. 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 appendix.

---

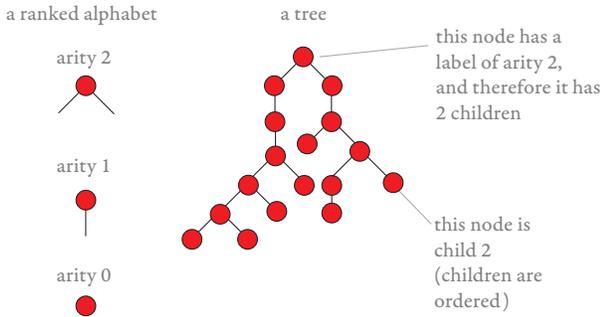
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

Conference'17, July 2017, Washington, DC, USA

© 2020 Association for Computing Machinery.  
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00  
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 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  $\{0, 1, 2, \dots\}$ . If  $a$  of a ranked set has arity  $n$ , then elements of  $\{1, \dots, n\}$  are called *ports of  $a$* . We adopt the convention that ranked sets are red, e.g.  $\Sigma$  or  $\Gamma$ , 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  $\text{trees}_\Sigma$  for the (unranked) set of trees over a ranked set  $\Sigma$ . This paper is about *tree-to-tree functions*, which are functions of the type

$$f : \text{trees}_\Sigma \rightarrow \text{trees}_\Gamma.$$

### 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  $\text{MSO}$ . 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  $\sigma$  or  $\tau$  for vocabularies.

**Definition 2.1** (Tree as a model). For a tree  $t$  over a ranked alphabet  $\Sigma$ , 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 ( $i \in \{1, 2, \dots\}$ )	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. We refer to this vocabulary as *the vocabulary of trees over  $\Sigma$* . A sentence of first-order logic (or  $\text{MSO}$ ) over this vocabulary describes a tree language, namely the set of trees whose associated

models satisfy the sentence. For example, the sentence

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

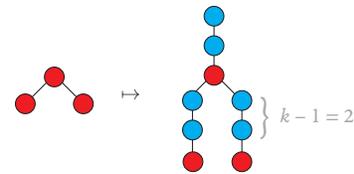
is true in (the models associated to) trees  $t$  where every node with label  $a$  has a descendant with label  $b$ . For more background about defining properties of trees using logic, see the survey of Thomas [32].

The regular tree languages are exactly those that can be defined in  $\text{MSO}$ , which was proved by Doner [17, Corollary 3.11], and also Thatcher and Wright [30, p. 74]. The tree languages definable in first-order logic are a proper subset of those definable in  $\text{MSO}$ , and it is an open problem whether or not one can decide if a regular tree language can be defined in first-order logic [9, 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<sup>1</sup> of the following two kinds:

1. *Copying.* Let  $k \in \{1, 2, \dots\}$ . 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  $\bullet$ , 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  $\Sigma$  and  $\Gamma$* , which are finite ranked sets. We use the name *input vocabulary* for the vocabulary of trees over the input alphabet  $\Sigma$ , likewise we define the *output vocabulary*.
  - b. A first-order formula over the input vocabulary, with one free variable, called the *universe formula*.

<sup>1</sup>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].

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.

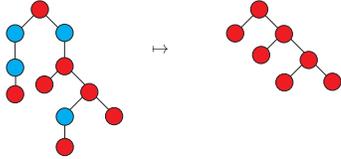
If we allowed monadic second-order logic MSO in items 2b and 2c (the free variables of the formulas would still be first-order variables ranging over tree nodes), then we would get the MSO tree-to-tree transductions of Bloem and Engelfriet [7, Section 3]. We discuss these in Section 7.

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:

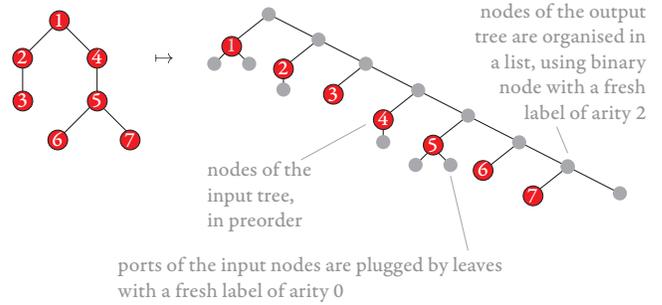


This is a non-copying first-order transduction. The universe formula selects nodes which have non-unary labels. The descendant relation is inherited from the input tree. To define the child relation on the output tree, we use the descendant relation in the input tree. A node  $x$  satisfies the unary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

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

This example shows the usefulness of first-order logic with descendant, as opposed to child only as used in [6].

**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.

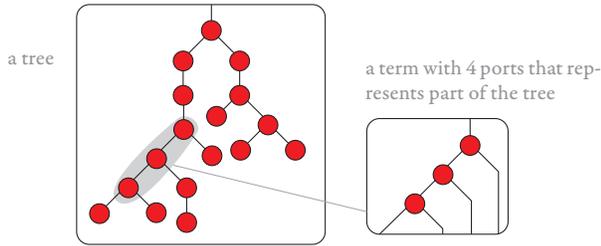
#### 3.1 Datatypes

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*  $\perp$  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  $\perp$  for partial functions: a partial function with output type  $\Sigma$  can be seen as a total function of output type  $\Sigma + \perp$ , which uses  $\perp$  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 are used to decompose trees (and other terms) into smaller pieces, as illustrated by the figure below.

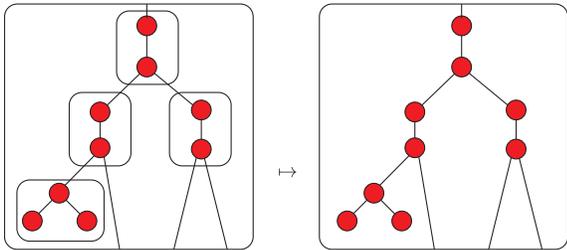


Formally speaking, terms are defined by induction as follows. As term over a ranked set  $\Sigma$  is either the *identity term* denoted by  $\square$ , which consists of a port and nothing else, or otherwise it is an expression of the form  $a(t_1, \dots, t_n)$  where  $a \in \Sigma$  has arity  $n$ , and  $t_1, \dots, t_n$  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  $T\Sigma$  for the ranked set of terms over a ranked set  $\Sigma$ . 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<sup>2</sup>. The unit of the monad, an operation of type  $\Sigma \rightarrow T\Sigma$ , is illustrated in the following picture:



The product of the monad, an operation of type  $T\Sigma \rightarrow T\Sigma$  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

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

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:

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

<sup>2</sup>An almost identical monad is used in [10, Section 9.2], which differs from ours in that it allows multiple uses of a single port.

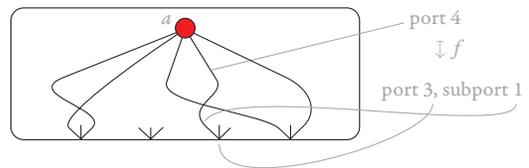
where  $\coprod$  denotes possibly infinite coproduct and  $X^n$  denotes the  $n$ -fold product of a ranked set  $X$  with itself.

**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  $F_k\Sigma$ , one for every  $k \in \{1, 2, 3, \dots\}$ . An  $n$ -ary element of  $F_k\Sigma$ , which is called a  $k$ -fold, consists of an element  $a \in \Sigma$  together with an injective *grouping function*

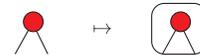
$$f : \underbrace{\{1, \dots, \text{arity of } a\}}_{\text{an element of this set is called a port of } a} \rightarrow \underbrace{\{1, \dots, n\} \times \{1, \dots, k\}}_{\text{these pairs are called sub-ports}}$$

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,  $F_1T\Sigma$  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).

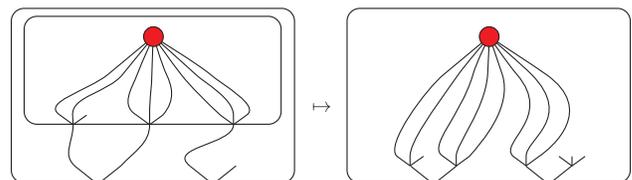
When viewed as a family of datatype constructors, folds have a monad-like structure: they are a graded monad in the sense of [21, p. 518]. The unit is the operation



of type  $\Sigma \rightarrow F_1\Sigma$ , while the product (or flattening) in the graded monad is the family of operations of type

$$F_{k_2}F_{k_1}\Sigma \rightarrow F_{k_1 \cdot k_2}\Sigma,$$

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



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

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

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

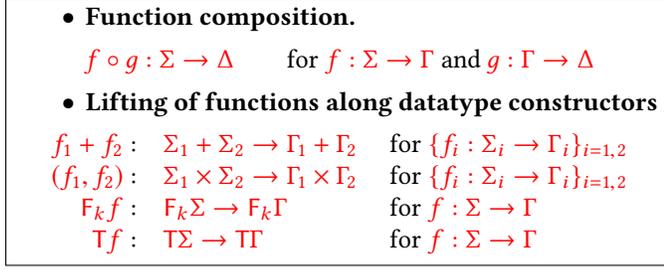


Figure 1. Combinators

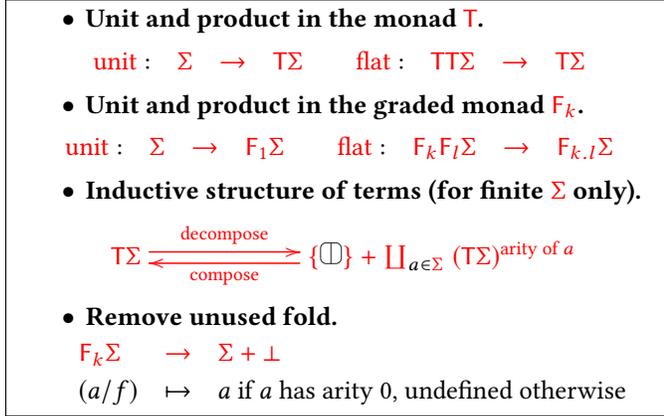


Figure 2. Prime functions for terms and fold.

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

$$T\Sigma \quad \Sigma_1 \times \Sigma_2 \quad \Sigma_1 + \Sigma_2 \quad F_k \Sigma.$$

### 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  $\Sigma$ , the unique arity preserving function  $\Sigma \rightarrow \perp$ ;
- all arity preserving functions with finite domain;
- the prime functions in Figures 2,3 and 4;

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.

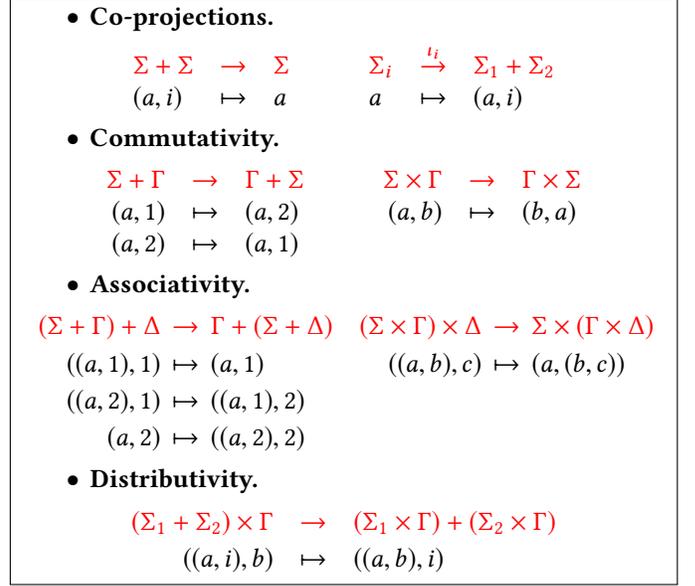


Figure 3. Prime functions for product and coproduct.

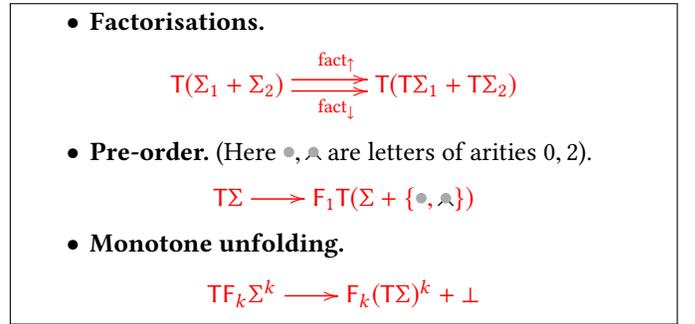


Figure 4. Functions explained in Section 3.3.

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

$$h : \Sigma \rightarrow T\Gamma$$

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:

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

More examples of derivable functions are in Appendix. B.

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

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

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

$$f : \mathbb{T}\Sigma \rightarrow \mathbb{T}\Gamma + \perp.$$

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 appendix, 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 nontrivial 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.

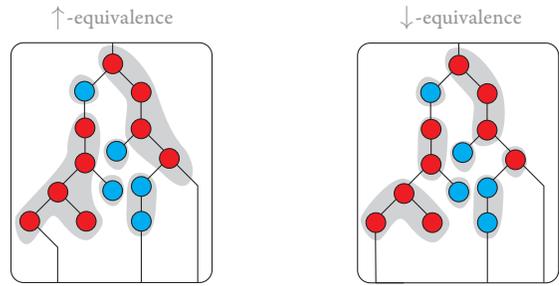
#### 3.3.1 Factorisations

We begin with the two factorisation functions

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

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  $t \in \mathbb{T}(\Sigma_1 + \Sigma_2)$ . We say that two nodes have the *same type* if both have labels in the same  $\Sigma_i$ ; 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  $\Sigma_1$  being red and  $\Sigma_2$  being blue:



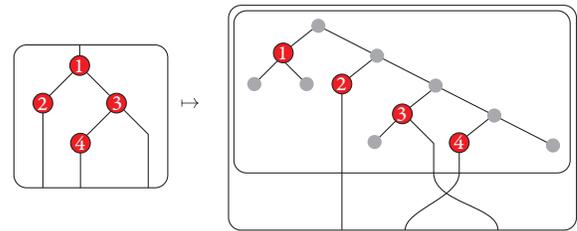
For both equivalence relations, the equivalence classes are connected under the parent-child relation, and therefore the equivalences can be seen as factorisations. These are the factorisations produced by the functions  $\text{fact}_\uparrow$  and  $\text{fact}_\downarrow$ .

#### 3.3.2 Pre-order traversal

The pre-order traversal function

$$\text{preorder} : \mathbb{T}\Sigma \rightarrow \mathbb{F}_1\mathbb{T}(\Sigma + \{\bullet, \blacktriangleleft\})$$

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:

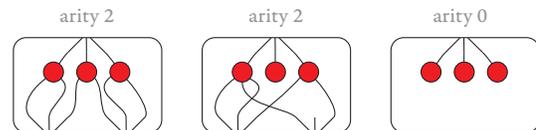


#### 3.3.3 Unfolding of the matrix power

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 [29, p. 268].

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

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

$$\mathbb{T}\Sigma^{[k]} \rightarrow (\mathbb{T}\Sigma)^{[k]},$$

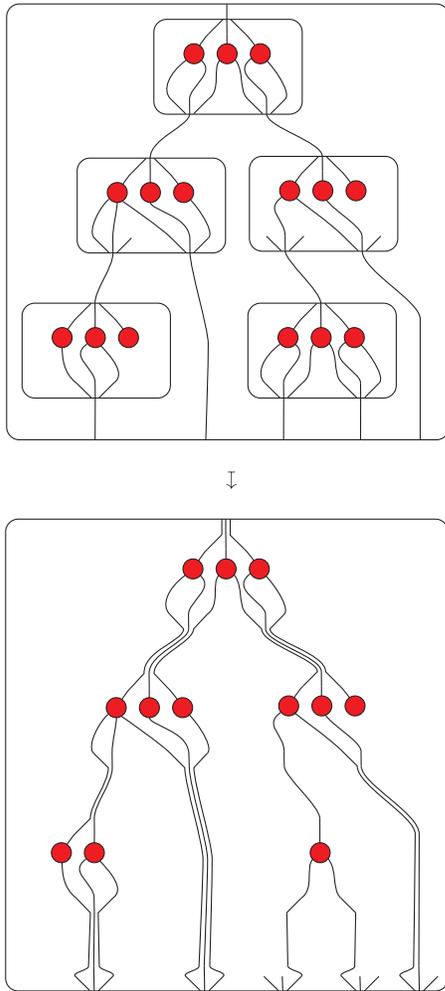


Figure 5. Unfolding the matrix power

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 the appendix.

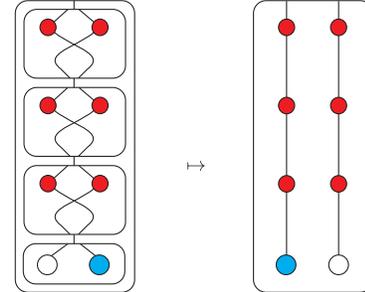
**Chain logic.** The general unfolding operation is too powerful to be included in the derivable functions, as we explain below. It does, however, admit a characterisation in terms of a fragment of MSO called *chain logic*, see [31, Section 2] or [8, Section 2.5.3], whose expressive power is strictly 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.

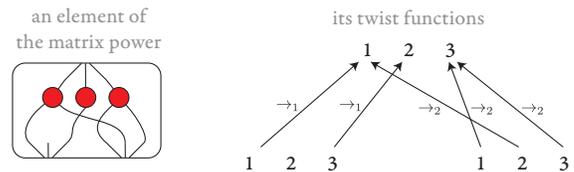
To see why chain logic is needed to describe general unfolding, consider the following unfolding, where two coordinates are swapped in each node of the input tree:



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  $p, q \in \{1, \dots, k\}$ , and let  $i$  be a port of  $a$ . Define the *twist function of port  $i$* , denoted by  $\rightarrow_i$ , as follows:  $q \rightarrow_i p$  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,  $\rightarrow_1$  is monotone, while  $\rightarrow_2$  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 appendix, 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

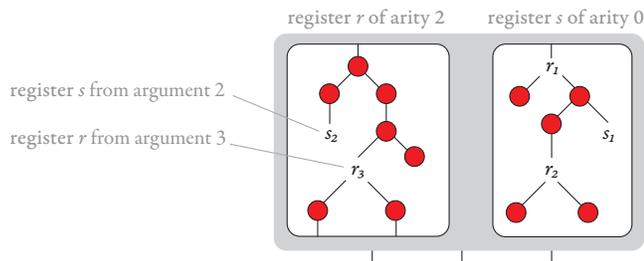
We now begin the proof of the harder implication in Theorem 3.4, which says that every first-order tree-to-tree transduction is derivable. Our proof passes through an automaton model, which is roughly based on existing transducer models for MSO transductions from [3, 7]. The automaton uses registers 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*  $\Gamma$ . A *register valuation* is defined to be any arity preserving function from the register names  $R$  to terms  $\mathbb{T}$ . 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  $n \in \{0, 1, \dots\}$ , an *n-ary register update* is defined to be any arity-preserving function

$$u : R \rightarrow \mathbb{T}(\Gamma + nR),$$

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 an 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 [7, 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  $\Sigma$  and  $\Gamma$ , 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  $\Sigma$ . 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

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

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*  $\Sigma$ , which is a finite ranked set;
- An *output alphabet*  $\Gamma$ , 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

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

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

1. *Single-use*<sup>3</sup>. 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*<sup>4</sup>. This condition uses the total order on registers. An  $n$ -ary register update  $u$  is called *monotone* if for every  $i \in \{1, \dots, n\}$ , the binary relation  $\rightarrow_i$  on register names  $r, s \in R$  defined by

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

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

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

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, 7]. 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 is in Appendix E, uses the composition method for logic, like similar proofs for [3, Theorem 4.6] and [7, 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 [7, 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.

<sup>3</sup>The single-use restriction is a standard feature of transducer models with linear size increase [1, 3, 7]. It prohibits iterated duplication of registers, which would lead to exponential size outputs.

<sup>4</sup>This is notion of monotonicity corresponds to the one used in Section 3.3.3, see the comments on page 11. A similar notion appears in [11, p. 7].

**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  $\Sigma$ . For a unary query, define its *characteristic function*, of type

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

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.

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

1. Letter-to-letter homomorphisms. For every finite  $\Gamma, \Sigma$  and  $f : \Sigma \rightarrow \Gamma$ , its *tree lifting*  $\text{trees}f : \text{trees}\Sigma \rightarrow \text{trees}\Gamma$ .
2. For every finite  $\Sigma$  and its subsets  $\Delta, \Gamma \subseteq \Sigma$ , the *characteristic functions of the following unary queries over alphabet  $\Sigma$* :
  - a. Child:  $x$  is an  $i$ -th child, for  $i \in \{1, 2, \dots\}$

$$\text{child}_i(x);$$

- b. Until:  $x$  has a descendant  $y$  with label in  $\Delta$ , such that all nodes strictly between  $x$  and  $y$  have label in  $\Gamma$

$$\exists y y > x \wedge \Delta(y) \wedge \forall z (x < z < y \Rightarrow \Gamma(z));$$

- c. Since:  $x$  has an ancestor  $y$  with label in  $\Delta$ , such that all nodes strictly between  $x$  and  $y$  have label in  $\Gamma$

$$\exists y y < x \wedge \Delta(y) \wedge \forall z (y < z < x \Rightarrow \Gamma(z)).$$

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 is in Appendix D.

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 Appendix D, 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  $\lambda$ -calculus. In Section 6.1, we discuss derivability of normalisation of  $\lambda$ -terms. In Section 6.2, we reduce evaluation of register updates to unfolding the matrix power and normalisation of  $\lambda$ -terms.

### 6.1 Normalisation of simply typed linear $\lambda$ -terms

We assume that the reader is familiar with the basic notions of the simply typed  $\lambda$ -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:

$$o \quad o \rightarrow o \quad (o \rightarrow o) \rightarrow (o \rightarrow o) \quad \dots$$

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  $\lambda$ -term is any expression that can be built from the variables, using  $\lambda$ -abstraction  $\lambda x.M$  and term application  $MN$ . We say that a  $\lambda$ -term is *well-typed* if one can associate to it a simple type according to the usual typing rules of simply typed  $\lambda$ -calculus, see [27, Definition 3.2.1]. Because the variables are typed, a  $\lambda$ -term has either a unique type, or is not well-typed. Here is an example of a well-typed  $\lambda$ -term, with the type annotation in blue:

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

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

A  $\lambda$ -term can be seen as a tree over the ranked alphabet

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

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

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

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

The first assumption is that the input  $\lambda$ -term is *linear*: every bound variable is exactly once in its scope<sup>5</sup>, but free variables are allowed to appear multiple times. The second assumption is that the input  $\lambda$ -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 Appendix F.1, 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  $\lambda$ -terms are represented as trees:*

- **Input.** A  $\lambda$ -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 Appendix F. A key role in the proof is played by the pre-order function.

### 6.2 Evaluation of register updates

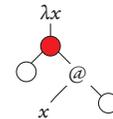
Equipped with Theorem 6.1, we prove derivability of evaluation of register updates. Fix a first-order register transducer. From now on, when speaking about register updates or register valuations, we mean those of the fixed transducer. Our goal is to prove the following lemma, which completes the proof 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  $\lambda$ -terms.** We will use  $\lambda$ -terms to represent register updates, which involve letters of the output alphabet  $\Gamma$ . Therefore, for the rest of Section 6.2, we use an extended notion of  $\lambda$ -terms, which allows building  $\lambda$ -terms of the form

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

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



<sup>5</sup>This restriction could easily be relaxed to “at most once”.

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

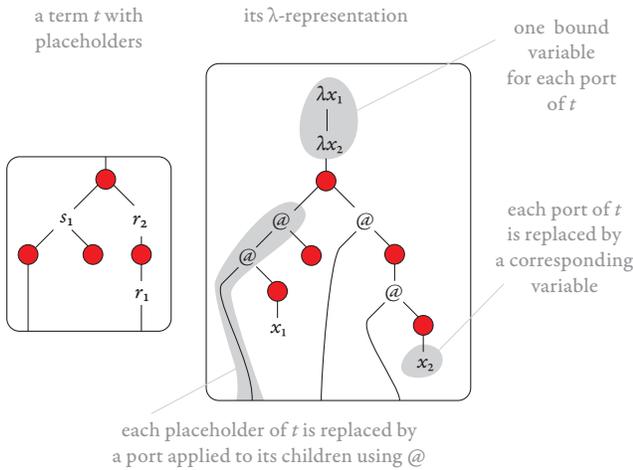
**$\lambda$ -representations of register updates.** To prove Lemma 6.2, we represent register updates using a matrix power of  $\lambda$ -terms. The idea is that the matrix power handles the parallel evaluation of registers.

Let  $X$  be a set of variables  $\{x_1 \dots, x_m\}$ , all of them having type  $o$ , where  $m$  is the maximal arity among registers. Define  $\Gamma_\lambda$  to be the output alphabet  $\Gamma$  plus the ranked alphabet defined in (1) for tree representations of  $\lambda$ -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  $\lambda$ -representation for terms in the family, which is a function of type

$$\mathbf{T}(\Gamma + nR) \xrightarrow{\lambda\text{-representation}} \mathbf{TT}_\lambda^k. \quad (3)$$

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  $i \in \{1, \dots, n\}$ . 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  $\{1, \dots, n\}$ .

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

$$\text{register updates} \xrightarrow{\lambda\text{-representation}} (\mathbf{TT}_\lambda)^{[k]}, \quad (4)$$

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

For a register update  $(t_1, \dots, t_k)$ , where  $t_i$  is the term with placeholders used in the  $i$ -th register, its  $\lambda$ -representation is

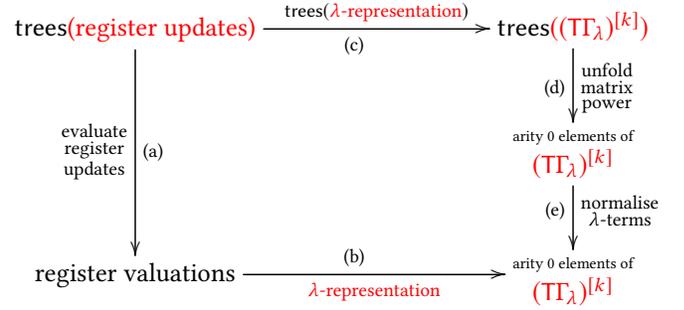
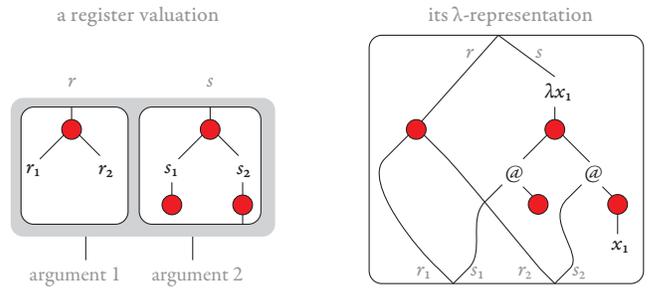


Figure 6

defined to be

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

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



The following three properties of the  $\lambda$ -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  $\lambda$ -representation is monotone (as defined in Section 3.3.3 for the matrix power).
- (P3) Every bound variable in the  $\lambda$ -representation is used exactly once, and the types that appear are of the form

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

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  $\lambda$ -representation – to unfolding the matrix power and normalising a  $\lambda$ -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.

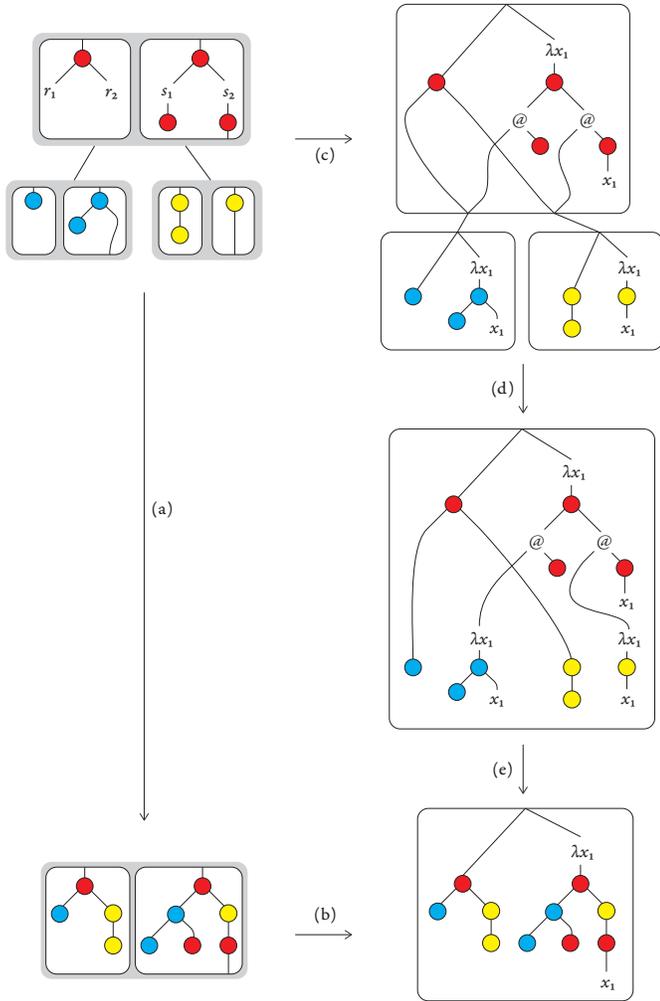


Figure 7. Example for Figure 6.

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  $\Delta$  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  $\lambda$ -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  $\lambda$ -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 monadic second-order 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.  $\square$

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 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 [9, 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] Augustin Baziramwabo, Pierre McKenzie, and Denis Thérien. Modular temporal logic. In *Logic in Computer Science LICS, Trento, Italy*, pages 344–351. IEEE, 1999.
- [6] Michael Benedikt and Luc Segoufin. Regular tree languages definable in FO and in FO<sub>mod</sub>. *ACM Trans. Comput. Log.*, 11(1):4:1–4:32, 2009.
- [7] 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.
- [8] Mikołaj Bojańczyk. *Decidable Properties of Tree Languages*. PhD Thesis, University of Warsaw, 2004.
- [9] Mikołaj Bojańczyk. Some open problems in automata and logic. *ACM SIGLOG News*, 2(4):3–15, 2014.
- [10] Mikołaj Bojańczyk. Recognisable languages over monads. *CoRR*, abs/1502.04898, 2015.
- [11] 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.
- [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 Jákł. 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. Log.*, 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] Howard Straubing. *Finite Automata, Formal Logic, and Circuit Complexity*. Birkhauser. 1994.
- [29] Walter Taylor. The fine spectrum of a variety. *Algebra Universalis*, 5(1):263–303, 1975.
- [30] 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.
- [31] Wolfgang Thomas. Infinite trees and automaton- definable relations over  $\omega$ -words. *Theoretical Computer Science*, 103(1):143 – 159, 1992.
- [32] Wolfgang Thomas. Languages, automata, and logic. In *Handbook of formal languages*, pages 389–455. Springer, 1997.

## A Unfolding the matrix power

In this part of the appendix, we define formally the unfolding function

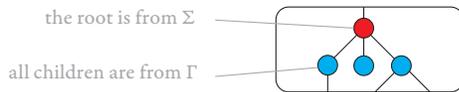
$$\mathbb{T}\Sigma^{[k]} \rightarrow (\mathbb{T}\Sigma)^{[k]}$$

that was described in Section 3.3.3. We present the definition in a slightly verbose manner, by decomposing unfolding into simpler operations. The presentation highlights the inductive character of unfolding, and the reasons why we are uneasy about it being a prime operation.

### A.1 Shallow terms

We begin by defining unfolding for terms of depth two, called *shallow terms*. Later, we extend the definition to all other terms by induction. We describe shallow terms as a separate datatype, since this datatype will also be used later, in Section G, to derive the (monotone) unfolding operation. For now, shallow terms are just an intermediate type used to define formally the unfolding function.

Let  $\Sigma$  and  $\Gamma$  be two ranked sets. The shallow terms datatype, which is denoted  $\Sigma.\Gamma$ , consists of expressions of the form  $a(b_1, \dots, b_n)$  where  $a$  is an  $n$ -ary element of  $\Sigma$  and  $b_1, \dots, b_n$  are elements of  $\Gamma$ . The arity of such an expression is the sum of arities of  $b_1, \dots, b_n$ . We draw shallow terms as terms of depth two, where the root is from  $\Sigma$  and the children are from  $\Gamma$ :



An equivalent definition of shallow terms, in terms of products and co-products, is

$$\Sigma.\Gamma \stackrel{\text{def}}{=} \coprod_{a \in \Sigma} \overbrace{\Gamma \times \dots \times \Gamma}^{\text{arity of } a \text{ times}} \quad (5)$$

### A.2 Terms as an inductive datatype

Using shallow terms, we can define the set of terms as the least solution of the equation

$$\mathbb{T}\Sigma = \{\emptyset\} + \Sigma.(\mathbb{T}\Sigma).$$

With this inductive definition, in order to define an operation of type  $\mathbb{T}\Sigma \rightarrow \Gamma$  on terms, it is enough to explain the induction base for the identity term and the induction step for shallow unfolding, as captured by two operations of types

$$\underbrace{\{\emptyset\} \rightarrow \Gamma}_{\text{induction base}} \quad \underbrace{\Sigma.\Gamma \rightarrow \Gamma}_{\text{induction step}}$$

We use such an induction below to define general unfolding. The crucial step is defining the induction step, which the unfolding for shallow terms defined in Section A.3 below.

As mentioned at the beginning of Section 3, the guiding principle behind our approach is to avoid iteration mechanisms. The inductive definition of general unfolding could be seen as such an iteration mechanism; this is the reason for Section G, where (monotone) unfolding is derived using simpler operations. In contrast, we believe that iteration is indeed avoided by the operations used in the induction step that are presented in Section A.3 below.

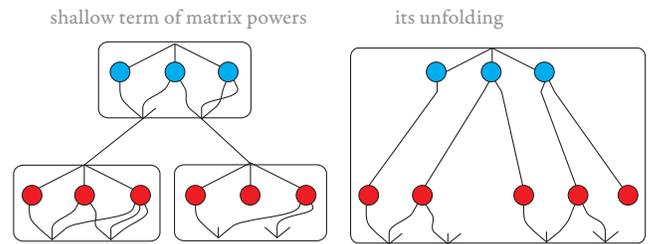
We do not formalise what we mean by “avoiding iteration”. One possible direction would be to say that an operation “avoids iteration” if it can be computed by a family of bounded depth circuits, as in the circuit class  $AC^0$ . A further requirement could be that the family of circuits not only exists, but it is also easy to see.

### A.3 Unfolding for shallow terms

The induction step in general unfolding is the operation

$$\Sigma^{[k]}. \Gamma^{[k]} \longrightarrow (\Sigma.\Gamma)^{[k]},$$

which we call shallow unfolding, and which is explained in the following picture:



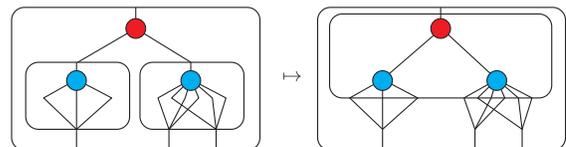
To define this operation formally, we further decompose it using three functions manipulating shallow terms. These functions, which are used here as intermediate functions in the definition of shallow unfolding, will become prime functions when we decompose the unfolding function in Appendix G.

#### A.3.1 Distribute shallow terms over fold

Let  $\Gamma$  and  $\Sigma$  be two datatypes. Consider the function  $f_1$

$$\Gamma.F_k\Sigma \xrightarrow{f_1} F_k(\Gamma.\Sigma)$$

which distributes shallow terms over folding. This function is illustrated by the following picture



and defined by

$$a(b_1/g_1, \dots, b_n/g_n) \mapsto a(b_1, \dots, b_n)/g$$

where  $g$  is the function defined as follows. For every  $i \in \{1, \dots, n\}$ , if  $j \in \{1, \dots, \text{arity}(b_i)\}$  then

$$\left( \underbrace{\pi_2(g_i(j) + \sum_{l < i} \text{arity}(b_l/g_l))}_{\text{Position of the group is shifted}}, \underbrace{\pi_1(g_i(j))}_{\text{Position inside the group is unchanged}} \right)$$

Position of the  $j$ -th port of  $b_i$  is shifted

$j + \sum_{l < i} \text{arity}(b_l)$

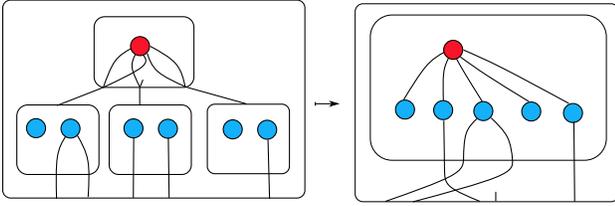
$\downarrow g$

### A.3.2 Matching function

We now define a function

$$(F_k \Gamma) \cdot (\Sigma^k) \xrightarrow{f_2} F_1(\Gamma \cdot \Sigma)$$

which matches the  $k$ -th fold with the  $k$ -th power<sup>6</sup>. The function  $f_2$  is illustrated by the following picture



and defined by

$$(a/g)((b_{1,1}, \dots, b_{1,k}), \dots, (b_{n,1}, \dots, b_{n,k})) \xrightarrow{f_2} a(b_{g(1)}, \dots, b_{g(m)})/g'$$

where  $m$  is the arity of  $a$  and the grouping function  $g'$  is the natural embedding of ports

$$\text{ports of } a(b_{g(1)}, \dots, b_{g(m)}) \downarrow \left( \prod_{\substack{i \in \{1, \dots, n\} \\ j \in \{1, \dots, k\}}} \text{ports of } b_{i,j}, 1 \right)$$

<sup>6</sup>In order to reduce the number of parentheses, in the rest of the paper we assume a notational convention where the unary datatype constructors – like folding, terms or powering – have priority over the binary shallow term constructor. Under this convention, the operation  $f_2$  is written as

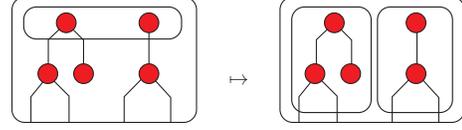
$$F_k \Gamma \cdot \Sigma^k \xrightarrow{f_2} F_1(\Gamma \cdot \Sigma)$$

### A.3.3 Distribute shallow terms over product

Finally, consider the function

$$\Gamma^k \cdot \Sigma \xrightarrow{f_3} (\Gamma \cdot \Sigma)^k$$

which distributes shallow terms over the  $k$ -th power. This function is illustrated by the following picture



and defined by

$$(a_1, \dots, a_k)(b_1, \dots, b_n) \xrightarrow{f_2} (a_1(b_1, \dots, b_{\text{arity}(a_1)}), a_2(b_{\text{arity}(a_1)+1}, \dots, b_{\text{arity}(a_2)}), \dots, a_k(b_{\text{arity}(a_{k-1})+1}, \dots, b_{\text{arity}(a_k)}))$$

where  $\text{arity}$  is the arity of  $a_i$  for  $i \in \{1, \dots, k\}$ .

### A.3.4 Unfolding shallow terms.

The following diagram defines unfolding of shallow terms in terms of the operations  $f_1, f_2, f_3$  defined above:

$$\begin{array}{ccc} \Sigma^{[k]} \cdot \Gamma^{[k]} = F_k \Sigma^k \cdot F_k \Gamma^k & \xrightarrow{\text{Shallow unfold}} & F_k(\Sigma \cdot \Gamma)^k = (\Sigma \cdot \Gamma)^{[k]} \\ f_1 \downarrow & & \uparrow F_k f_3 \\ F_k(F_k \Sigma^k \cdot \Gamma^k) & \xrightarrow{\text{flat} \circ F_k f_2} & F_k(\Sigma^k \cdot \Gamma) \end{array}$$

### A.4 Definition of unfolding

Having defined shallow unfolding, we apply the induction principle described in Section A.2 to define unfolding for general terms

$$\text{unfold} : T\Sigma^{[k]} \rightarrow (T\Sigma)^{[k]}.$$

If the input to general unfolding is the identity term  $\square$ , then the output is:



Otherwise, if the input is a nonempty term  $a(t_1, \dots, t_n)$  then the output is obtained by first applying term unfolding to the smaller terms  $t_1, \dots, t_n$ , and then applying the shallow unfold.

## B Examples

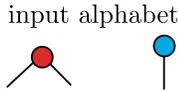
To illustrate derivable functions, we present a series of examples, some of them will be useful later. In the rest of this section, for every  $k \in \{1, 2, \dots\}$  the set  $k$  designates the ranked set containing a single element of arity  $k$  that we denote by simply by  $k$ .

**Example B.1** (Parent and children). Let  $\Gamma$  be a finite type. We define  $\Gamma_0$  to be the ranked set obtained from  $\Gamma$  by setting the arity of every element to 0. Consider the function:

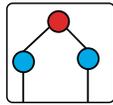
$$\text{Parent} : T\Gamma \rightarrow T(\Gamma \times (\Gamma_0 + 0))$$

which adds to every node of a term in  $T\Sigma$  the label of its parent if it has one, and 0 if it is the root.

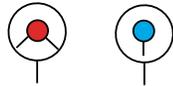
Let us explain how **Parent** can be derived. To illustrate this construction, we use the following alphabet  $\Gamma$



and the following term as a running example.



We denote by  $\Gamma_1$  the ranked set obtained from  $\Gamma$  by setting the arity of every element to 1. If  $a$  is a element of  $\Gamma$ , we denote by  $a_1$  the corresponding element of  $\Gamma_1$ . In our example, the alphabet  $\Gamma_1$  is



1. First, we apply the homomorphism

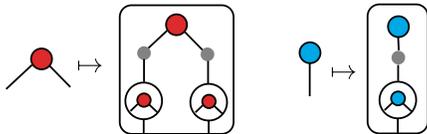
$$\text{Hom}_g : T\Gamma \rightarrow T(\Gamma + \Gamma_1 + 1)$$

where  $g$  is defined on the elements of  $\Gamma$  as follows

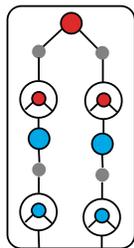
$$g : \Gamma \rightarrow T(\Gamma + \Gamma_1 + 1)$$

$$a \mapsto a(1(a_1(\square)), \dots, 1(a_1(\square)))$$

In our example, the action of  $g$  on the elements of  $\Gamma$  looks like this



Hence, after the application of the homomorphism **Hom<sub>g</sub>**, our initial term becomes

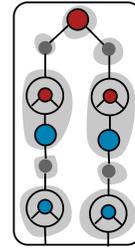


2. We apply the factorization

$$\text{fact}_\uparrow : T(\Gamma + \Gamma_1 + 1) \rightarrow T(T(\Gamma + \Gamma_1) + T1)$$

to separate the symbol 1 from the other symbols. After this operation, each node lies in the same factor as (the

element of  $\Gamma_1$  representing) its parent. In our example, the obtained term is the following



3. Consider the function

$$h : T1 \rightarrow T((\Gamma + \Gamma_1) \times (\Gamma_0 + 0))$$

which is the empty term constant function. It is derivable by lifting the empty term constant function over 1 to terms. And let  $k$  be the function

$$k : T(\Gamma + \Gamma_1) \rightarrow T((\Gamma + \Gamma_1) \times (\Gamma_0 + 0))$$

which is the identity function, except for the following terms in which it is defined as follows

$$a(\square, \dots, \square) \mapsto (a, 0)(\square, \dots, \square)$$

$$b_1(a(\square, \dots, \square)) \mapsto (a, b_0)(\square, \dots, \square)$$

$$b_1(\square) \mapsto \square$$

We apply the function  $h$  to the factors **T1** and the function  $k$  to the factors  $T(\Gamma + \Gamma_1)$ . Doing so, we obtain a term in  $TT((\Gamma + \Gamma_1) \times (\Gamma_0 + 0))$ , which we flatten, then we erase the symbols  $\Gamma_1$  using the function **Filter** of Example 3.3 to obtain the desired term.

If  $\Gamma$  is a finite ranked set, we define  $\Gamma^*$  as

$$\coprod_{i \leq \text{maximal arity in } \Gamma} \overbrace{\Gamma \times \dots \times \Gamma}^{i \text{ times}}$$

Now consider the function

$$\text{Children} : T\Gamma \rightarrow T(\Gamma \times (\Gamma_0 + 0)^*)$$

which tags every node of a term in  $T\Gamma$  by the list of its children symbols. When a child is a port, it is marked by 0 in the list. The function **Children** can be derived using a similar construction as above.

**Example B.2** (Root and leaves). Let  $\Sigma$  be a finite type and  $f : \Sigma \rightarrow \Gamma, g : \Sigma \rightarrow \Gamma$  be derivable functions. The function

$$\text{Root}_{f,g} : T\Sigma \rightarrow T\Gamma$$

which applies  $f$  to the root and  $g$  to the rest of the tree is a derivable function. To show this, we first start by applying the function **Parent**. Doing so, the root can be distinguished from the other nodes since it will be tagged by 0.

The function  $h$  defined below is derivable since its domain is finite.

$$h : \Sigma \times (\Sigma_0 + 0) \rightarrow \Gamma$$

$$(a, 0) \mapsto f(a)$$

$$(a, b) \mapsto g(a) \text{ if } b \neq 0.$$

We lift  $h$  to terms to conclude.

Similarly, the function

$$\text{Leaves}_{f,g} : T\Sigma \rightarrow T\Gamma$$

which applies  $f$  to the leaves and  $g$  to the rest of the tree is derivable. This is done using the same ideas as before, but invoking the function **Children** instead of the function **Parent**: leaves can be distinguished from the other nodes since they are tagged either by a list of 0 or the empty list.

**Example B.3** (Descendants and ancestors). If  $\Sigma$  is a finite type and  $\Gamma \subseteq \Sigma$ , then the functions

- $\text{Descendant}_{\Gamma} : T\Sigma \rightarrow T(\Sigma + \Sigma)$  which replaces the label of each node by its first or second copy, depending on whether it has a descendant in  $\Gamma$ ,
- $\text{Ancestor}_{\Gamma} : T\Sigma \rightarrow T(\Sigma + \Sigma)$  which replaces the label of each node by its first or second copy, depending on whether it has a descendant in  $\Gamma$ ,

are derivable.

To derive  $\text{Descendant}_{\Gamma}$ , we start by applying the factorization

$$\text{fact}_{\downarrow} : T\Sigma \rightarrow T(T\Gamma + T(\Sigma \setminus \Gamma))$$

which regroups the elements of  $\Sigma$  and the elements of  $\Sigma \setminus \Gamma$  into factors depending on whether they have the same ancestors of the same type.

Obviously, all the nodes of the  $\Gamma$  factors have a descendant in  $\Gamma$ . In the  $\Sigma \setminus \Gamma$  factors which are not leaves in the factorized term, all the nodes have a  $\Gamma$  descendant in the original term. To show this, take  $f$  to be one of these factors, and suppose by contradiction that one of its nodes does not have a descendant in  $\Gamma$ . By definition of  $\text{fact}_{\downarrow}$ , all the elements of  $f$  do not have a descendant in  $\Gamma$  as well. Since  $f$  is not a leaf, it has a child  $g$ . The factor  $g$  cannot be a  $\Gamma$  factor as the nodes of  $f$  would have a descendant in  $\Gamma$ . The factor  $g$  is then necessarily a  $\Sigma \setminus \Gamma$  factor. If a node of  $g$  has a descendant in  $\Gamma$ , this would give a  $\Gamma$  descendant to one of the node of  $f$ . Thus all the nodes of  $g$  are in  $\Sigma \setminus \Gamma$  and do not have a descendant in  $\Gamma$ , meaning that  $f$  and  $g$  are actually the same factor, which gives a contradiction. Finally, the  $\Sigma \setminus \Gamma$  factors which are leaves do not have a descendant in  $\Gamma$ . With these observations, we can now implement  $\text{Descendant}_{\Gamma}$ .

Let us consider the functions

$$\begin{aligned} \text{Yes}_{\Gamma} : \Gamma &\rightarrow \Sigma + \Sigma \\ \text{Yes}_{\Sigma \setminus \Gamma} : \Sigma \setminus \Gamma &\rightarrow \Sigma + \Sigma \\ \text{No}_{\Sigma \setminus \Gamma} : \Sigma \setminus \Gamma &\rightarrow \Sigma + \Sigma \end{aligned}$$

which replaces the label of each node by its first copy for  $\text{Yes}_{\Gamma}$  and  $\text{Yes}_{\Sigma \setminus \Gamma}$ , and by its second copy for  $\text{No}_{\Sigma \setminus \Gamma}$ . The three functions are derivable as their domains are finite. Consider the functions

$$\begin{aligned} f &:= T\text{Yes}_{\Gamma} + T\text{No}_{\Sigma \setminus \Gamma} : T\Gamma + T(\Sigma \setminus \Gamma) \rightarrow T(\Sigma + \Sigma) \\ g &:= T\text{Yes}_{\Gamma} + T\text{Yes}_{\Sigma \setminus \Gamma} : T\Gamma + T(\Sigma \setminus \Gamma) \rightarrow T(\Sigma + \Sigma) \end{aligned}$$

The descendant function is obtained by applying  $\text{leaves}_{f,g}$  followed by a flattening.

To derive the function  $\text{Ancestor}_{\Gamma}$ , we apply first a the factorization

$$\text{fact}_{\uparrow} : T\Sigma \rightarrow T(T\Gamma + T(\Sigma \setminus \Gamma))$$

which regroups the elements of  $\Sigma$  and the elements of  $\Sigma \setminus \Gamma$  into factors depending on whether they have the same descendants of the same type. Using similar arguments as before, we can conclude that:

- The nodes inside  $\Gamma$  factors have  $\Gamma$  ancestors.
- If a  $\Sigma \setminus \Gamma$  factor is the root of the factorized term, then its nodes do not have a  $\Gamma$  ancestor.
- If a  $\Sigma \setminus \Gamma$  factor is not the root of the factorized term, then its nodes do have a  $\Gamma[\text{Descendantsandancestors}]$  ancestor.

The ancestor function is obtained by applying  $\text{root}_{f,g}$  followed by a flattening.

**Example B.4** (Error raising.). We can think of the type  $\perp$  as an error type. Indeed, the following raising error functions are derivable.

**Lemma B.5.** *Let  $\Sigma$  and  $\Gamma$  be two datatypes. The functions*

$$\begin{aligned} T(\Sigma + \perp) &\rightarrow T\Sigma + \perp \\ (\Sigma + \perp) \times (\Gamma + \perp) &\rightarrow \Sigma \times \Gamma + \perp \\ (\Sigma + \perp).(\Gamma + \perp) &\rightarrow \Sigma.\Gamma + \perp \\ F_k(\Sigma + \perp) &\rightarrow F_k\Sigma + \perp \end{aligned}$$

which are defined as follows

$$t \text{ of arity } n \mapsto \begin{cases} t & \text{if } t \text{ does not contain any element of } \perp, \\ n & \text{otherwise.} \end{cases}$$

are derivable.

These functions can be easily derived using Proposition 5.1 and distributivity prime functions. The details of the proof are left as an exercise to the reader.

**Example B.6** (Partial functions.). Thinking of  $\perp$  as an error datatype, a function of type  $\Sigma \rightarrow \Gamma + \perp$  can be seen as a partial function from  $\Sigma$  to  $\Gamma$ . We write

$$\Sigma \rightarrow \Gamma$$

as a notation for the function type  $\Sigma \rightarrow \Gamma + \perp$ . Using the error raising mechanisms discussed earlier, we can manipulate transparently partial function. Indeed, all datatype constructors can be lifted to partial functions, by composing the liftings (1)–(4) with the error raising functions from Lemma B.5. For example, if  $f : \Sigma \rightarrow \Gamma$  is a partial function, then  $Tf : T\Sigma \rightarrow T\Gamma$  is defined as the composition

$$T\Sigma \xrightarrow{Tf} T(\Gamma + \perp) \xrightarrow{\text{Error raising}} T\Gamma + \perp.$$

## C Derivable functions can be described in first-order logic

The goal of this section is to show the right-to-left implication of Theorem 3.4, which says that derivable functions can be implemented by first-order transductions.

As discussed in the body of the paper, we proceed by induction on the derivation. During this induction, we will need to show that every prime function is a first-order transduction. Prime functions are not tree-to-tree functions, instead they transform datatypes into datatypes. This is the reason why we need

- to generalize tree-to-tree transductions into transductions that can transform models over arbitrary vocabularies (and not only the vocabulary of trees).
- show how datatypes (terms, pairs, copairs and folds) can be encoded as models over a well chosen vocabulary. More precisely, we will associate to every datatype  $\Sigma$  a relational vocabulary that we call *vocabulary of  $\Sigma$* . Structures over this vocabulary will be called *models over  $\Sigma$* . Then we will define a function

$$\Sigma \xrightarrow{x \mapsto \underline{x}} \text{models over } \Sigma$$

which assigns to each element  $x \in \Sigma$  a corresponding model over  $\Sigma$ , which is denoted by  $\underline{x}$ .

Right-to-left implication of Theorem 3.4 can be then generalized to the following statement, more suited to a proof by induction:

**Proposition C.1.** *Let  $\Gamma$  and  $\Sigma$  be two datatype. For every derivable function  $f$ , there is a first-order transduction  $g$  such that the following diagram commutes*

$$\begin{array}{ccc} \Sigma & \xrightarrow{f} & \Gamma \\ \downarrow x \mapsto \underline{x} & & \downarrow x \mapsto \underline{x} \\ \text{models over } \Sigma & \xrightarrow{g} & \text{models over } \Gamma \end{array}$$

The rest of this section is organized as follows. We define first-order transductions transforming arbitrary models in Section C.1. In Section C.2 we define the vocabularies for the datatypes and the model representation  $x \mapsto \underline{x}$ . Finally, we prove Proposition C.1 which gives as a corollary the right-to-left implication of Theorem 3.4.

### C.1 First-order transductions

The following definition introduces first-order transductions, which generalizes tree-to-tree transductions given in Definition 2.2 to arbitrary models.

**Definition C.2** (First-order transduction). *A first-order transduction is defined to be any composition of the following two kinds of transformations on structures:*

1. *Copying.* Fix some relational vocabulary  $\sigma$  and let  $k \in \{1, 2, \dots\}$ . Define  $k$ -copying to be the operation of type

$$\begin{array}{c} \text{models over } \sigma \\ \downarrow \\ \text{models over } \sigma \text{ extended} \\ \text{with a } k\text{-ary relation copy} \end{array}$$

which inputs a model  $\mathbb{A}$ , and outputs  $k$  disjoint copies of  $\mathbb{A}$ , where the copy relation is interpreted as the set of tuples  $(a_1, \dots, a_k)$  such that, for some  $a \in \mathbb{A}$ , the first copy of  $a$  is  $a_1$ , the second copy of  $a$  is  $a_2$ , etc. The copy relation is not commutative, because we distinguish the copies.

2. *Non-copying first-order transduction.* The syntax of a *non-copying first-order transduction* is given by:
  - a. Input relational vocabulary  $\sigma$  and output relational vocabulary  $\gamma$ .
  - b. A first-order *universe formula*  $\varphi(x)$  over  $\sigma$ .
  - c. For every relation  $R$  in vocabulary  $\gamma$ , a first-order formula  $\varphi_R(x_1, \dots, x_{\text{arity}(R)})$  over  $\sigma$ .

The semantics of a non-copying first-order transduction is a function

$$\begin{array}{c} \text{models over } \sigma \\ \downarrow \\ \text{models over } \gamma \end{array}$$

defined as follows. If the input model is  $\mathbb{A}$ , then the output model is defined as follows: the universe is elements of  $\mathbb{A}$  which satisfy the universe formula, and each relation  $R$  is interpreted as those tuples that satisfy  $\varphi_R$ .

The notion of copying used in the above definition is slightly different from the notion of copying used for tree-to-tree transductions in Definition 2.2, which was specifically tailored to stay within the realm of trees. Nevertheless, the two definitions are easily seen to define the same class of tree-to-tree functions.

### C.2 Datatypes as models.

Let us show how to encode datatypes as relational vocabularies and data as models over these vocabularies.

**Definition C.3** (Associated models for terms, pairs, co-pairs, folds.). To each type  $\Sigma$  we associate a vocabulary, called the *vocabulary of  $\Sigma$* , and a map

$$a \in \Sigma \quad \mapsto \quad \underbrace{a \in \text{models over the vocabulary of } \Sigma}_{\text{associated model of } a}$$

Furthermore, for each  $a \in \Sigma$  we distinguish a sequence (whose length is the arity of  $a$ ) of elements in  $\underline{a}$ , which are called the ports of  $\underline{a}$ . The definitions are by induction on the structure of  $\Sigma$ , as given below.

- *Finite ranked sets.* Elements of a ranked set

$$\Sigma = \{a_1, \dots, a_k\}$$

are modelled using a vocabulary which has unary relations  $a_1, \dots, a_k$  and  $P_1, \dots, P_m$  where  $m$  is the maximal arity of elements in  $\Sigma$ . For  $a \in \Sigma$  of arity  $n$ , the universe of  $\underline{a}$  is  $\{0, 1, \dots, n\}$ , with the ports being  $1, \dots, n$ . The relation  $P_i$  is interpreted as  $\{i\}$  when  $i \in \{1, \dots, n\}$  and as the empty set otherwise. The relation  $a_i$  is interpreted as  $\{0\}$  when  $a = a_i$  and as the empty set otherwise.

- *Coproduct.* Elements of the coproduct  $\Sigma_1 + \Sigma_2$  are modelled using the disjoint union of the vocabularies of  $\Sigma_1$  and  $\Sigma_2$ . If an element of the coproduct comes from  $\Sigma_1$ , then its associated model is defined as for the type  $\Sigma_1$ , with the remaining relations from the vocabulary of  $\Sigma_2$  interpreted as empty sets. The definition is analogous for elements from  $\Sigma_2$ .
- *Product.* Pairs in  $\Sigma_1 \times \Sigma_2$  are modelled using the disjoint union of the vocabularies of  $\Sigma_1$  and  $\Sigma_2$ . For  $(a_1, a_2)$ , the associated model is the disjoint union of models  $\underline{a_1} + \underline{a_2}$ , with the relations of  $\underline{a_1}$  using the vocabulary of  $\Sigma_1$ , and the relations of  $\underline{a_2}$  using the vocabulary  $\Sigma_1$ . If  $n_1$  is the arity of  $a_1$ , then the first  $n_1$  ports are inherited from  $\underline{a_1}$  and the remaining ports are inherited from  $\underline{a_2}$ .
- *Folding.* For  $k \in \{1, 2, \dots\}$ , elements of  $F_k \Sigma$  are modelled using the vocabulary of  $\Sigma$  plus two extra binary relations  $\sqsubset$  and  $R$ . If  $a \in \Sigma$  has arity  $nk$ , then the model associated to  $\underline{a/f}$  – which has arity  $n$  – is obtained from  $\underline{a}$  by adding a copy of the model below, where  $\sqsubset$  is the natural ordering on integers

$$(\{1, \dots, n\}, \sqsubset),$$

whose elements are used as the ports, and interpreting the binary relation  $R$  as

$$\{(i\text{-th port of } \underline{a}, f(i)) : i \in \{1, \dots, nk\}\}$$

- *Terms.* Terms in  $\mathbb{T}\Sigma$  are modelled using vocabulary of  $\Sigma$  extended with two fresh binary relations  $<$  and  $\sqsubset$ . Let  $t \in \mathbb{T}\Sigma$ . Consider the disjoint union of models

$$\bigsqcup_{x \in \text{non-port nodes in } t} \underline{a(x)}, \quad (6)$$

where  $\underline{a(x)}$  is the model over vocabulary of  $\Sigma$  that is defined by induction assumption. In the above disjoint union, the same vocabulary, namely the vocabulary of  $\Sigma$ , is used for all parts of the disjoint union. Next, consider the model

$$(\{1, \dots, n\}, \sqsubset) \quad (7)$$

where  $\sqsubset$  is the natural ordering on  $\{1, \dots, n\}$ . The model of  $t$  is defined by taking the disjoint union of the models in (6) and (7), and defining the descendent relation  $<$  as the set of pairs  $(u, v)$  such that:

- either  $u$  is the  $i$ -th port of  $\underline{a(x)}$  for some node  $x$  of  $a$ ,  $v$  is a port of  $\underline{a(y)}$  for some node  $y$  which is a descendent of the  $i$ -th child of  $x$ .
- or  $u$  is the  $i$ -th port of  $\underline{a(x)}$  for some node  $x$  of  $a$ ,  $v = j \in \{1, \dots, n\}$  and the  $j$ -th port of  $a$  is a descendent of the  $i$ -th child of  $x$ .

The above definition creates a certain ambiguity for trees, because if  $t$  is a tree over a finite ranked set  $\Sigma$ , then  $\underline{t}$  can be understood in two ways: as per Definition 2.1 for trees, or as per Definition C.3 when  $t$  is viewed as a special case of a term  $t \in \mathbb{T}\Sigma$ . Since we only use first-order transductions to transform relational structures, this ambiguity is not a problem, because one can easily define first-order transductions which map one definition of  $\underline{t}$  to the other.

### C.3 Proof of Proposition C.1

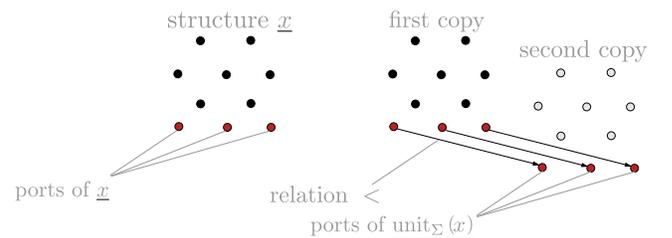
The proof proceeds by induction, following the definition of derivable functions. In the induction step, we have to deal with function composition and the lifting of function along the datatype constructors. First-order transductions are closed under composition by definition, while the liftings are immediate.

In the induction base, we need to show that all of the prime functions are first-order transductions. All the cases are easy, and consist mainly on unfolding the definitions; this is the point of calling these functions prime. There is one exception, which requires some more explanation, namely monotone unfolding. We explain below just one of the easy functions, the unit function  $\Sigma \rightarrow \mathbb{T}\Sigma$ , and the monotone unfolding. The other prime functions are left as an exercise.

#### C.3.1 A first-order transduction for the term unit

In the following, it will be convenient to use, as part of the vocabulary of  $\Sigma$ , a unary relation  $\text{Port}_\Sigma$  which selects the ports of the structures over the vocabulary of  $\Sigma$ ; and a binary relation  $\sqsubset_\Sigma$  which orders these ports. By induction on  $\Sigma$ , we can show that both relations are definable by first-order formulas over the vocabulary of  $\Sigma$ .

Given an element  $x$  of  $\Sigma$ , let us show how  $\text{unit}(x)$  can be implemented using a first-order transduction. The copying constant is 2, the first copy will contain the whole structure  $\underline{x}$  and the second copy will select only the ports of  $\underline{x}$  which will serve as the ports of the structure  $\text{unit}_\Sigma(x)$ , as illustrated by the following picture



The universe formulas are then:

$$\varphi_1(x) = \text{True} \quad \varphi_2(x) = \text{Port}_{\Sigma}(x)$$

In the first copy, the vocabulary of  $\Sigma$  will be interpreted as in the original structure, and as the empty set in the second copy. That is, for every unary relation  $R$  and for every binary relation  $S$  in the vocabulary of  $\Sigma$ , we set:

$$\begin{aligned} \varphi_R^1(x) &= R(x) & \varphi_S^{1,1}(x, y) &= S(x, y) \\ \varphi_R^2(x) &= \text{False} & \varphi_S^{2,2}(x, y) &= \text{False} \end{aligned}$$

Let us interpret the relations  $<$  and  $\sqsubset$  of the vocabulary of  $\mathbb{T}\Sigma$ . The ports of  $\text{unit}(x)$  inherit the order of the ports of  $\underline{x}$ , this is why we set:

$$\varphi_{\sqsubset}^{2,2}(x, y) = x \sqsubset_{\Sigma} y$$

The descendant relation  $<$  connects the  $i^{\text{th}}$  port of  $\underline{x}$  to the  $i^{\text{th}}$  port of  $\text{unit}(x)$ . Since these nodes come from the same node in the original structure, we set:

$$\varphi_{<}^{1,2}(x, y) = x = y$$

### C.3.2 A first-order transduction for monotone unfolding

Having illustrated the syntax of first-order transductions on the example of the unit function, we describe a first-order transduction for the monotone unfolding operation

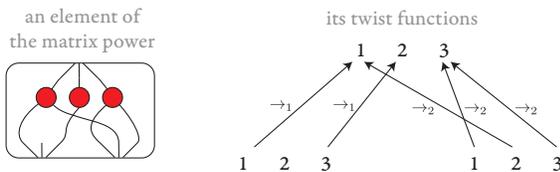
$$\mathbb{T}\Sigma^{[k]} \rightarrow (\mathbb{T}\Sigma)^{[k]} + \perp.$$

This is the only prime function whose corresponding first-order transduction is not obvious. Unlike in Section C.3.1, we focus more on the underlying conceptual difficulties than on the syntax of first-order transductions.

Recall that when defining the monotone unfolding operation, for each element  $a \in \mathbb{T}\Sigma^{[k]}$  of the matrix power, we used a family of (partial) twist functions

$$\rightarrow_i: \{1, \dots, k\} \rightarrow \{1, \dots, k\},$$

one for each port  $i$  of  $a$ . For the reader's convenience, we repeat a picture from Section 3.3.3, which explains the twist functions:



In this example, the twist function  $\rightarrow_1$  is monotone, but  $\rightarrow_2$  is not. The monotone unfolding operation works in the same way as general unfolding, except that it uses the undefined value  $\perp$  if the input term has at least one letter which uses at least one non-monotone twist.

The following lemma, whose simple proof is left to the reader, shows that the twist functions can be defined using first-order logic.

**Lemma C.4.** *Let  $\Sigma$  be a datatype and let  $k \in \{1, 2, \dots\}$ . For every partial function*

$$\tau: \{1, \dots, k\} \rightarrow \{1, \dots, k\}$$

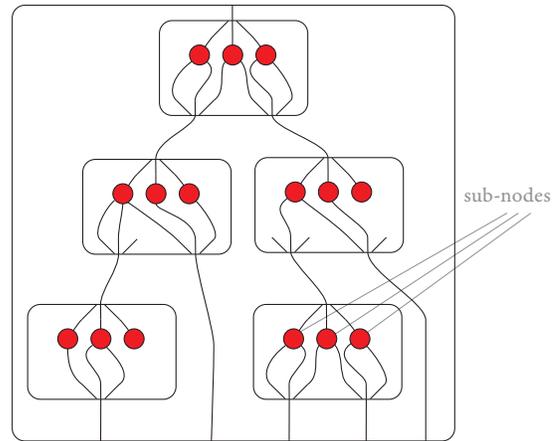
*there is a first-order formula  $\varphi_{\tau}(x)$  such that for every  $a \in \Sigma^{[k]}$ ,*

$$a \models \varphi_{\tau}(x)$$

*if and only if  $x$  represents a port with twist function  $\tau$ .*

By using the formulas from the above lemma, one can construct a first-order formula which checks if a term in  $\mathbb{T}\Sigma^{[k]}$  uses only monotone twists, i.e. whether or not the output of monotone unfolding should be  $\perp$ .

We now proceed to the more interesting part of monotone unfolding, i.e. actually doing the unfolding for monotone inputs. Consider an input  $t \in \mathbb{T}\Sigma^{[k]}$  to monotone unfolding. Define a *sub-node* of  $t$  to be a pair (node of  $t$ , number in  $\{1, \dots, k\}$ ), as explained in the following picture:



In the output of the monotone unfolding, which is of the form

$$(t_1, \dots, t_k)/f \in \mathbb{T}\Sigma^{[k]},$$

the nodes of the output terms  $t_1, \dots, t_k$  will correspond to the sub-nodes in the input  $t$ . The sub-nodes can be produced by copying the input term  $k$ -times.

The most interesting part of the structure in the output is the descendant relation in the terms  $t_1, \dots, t_k$ . This relation can be viewed as a descendant relation on the sub-nodes. We only describe how the descendant relation on the sub-nodes can be defined in first-order logic, and the rest of the transduction is left to the reader.

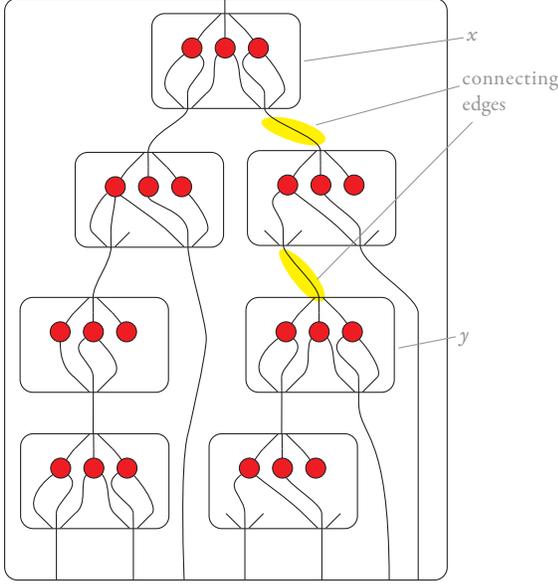
When defining the descendant relation on sub-nodes, the crucial part is composing the twist functions. Suppose that we want to check the descendant relationship between two sub-nodes

$$(x, i) \stackrel{?}{\leq} (y, j), \tag{8}$$

where  $x, y$  are nodes on the input term and  $i, j \in \{1, \dots, k\}$ . We will show that the descendant relationship (8) holds if and only if  $x$  is an ancestor of  $y$  in the input term, and the

twist functions on the path connecting  $x$  and  $y$  maps  $j$  to  $i$ , as explained below.

Consider a path in the input term, which connects node  $x$  with  $y$ , as illustrated in the following picture



Each edge in the input term corresponds to a chosen port in some node, which in turn corresponds to some twist function, and therefore it makes sense to talk about the twist function associated to an edge in the input term. Define

$$\tau_y^x : \{1, \dots, k\} \rightarrow \{1, \dots, k\}$$

to be the partial function, which is obtained by composing all of the twist functions corresponding to edges on the path connecting  $y$  to  $x$ , starting with  $y$  and ending with  $x$ . In the example from the above picture, we compose two twist functions, which correspond to edges marked in yellow.

Equipped with the above definitions, we can now characterise the descendant ordering on sub-nodes by

$$(x, i) \leq (y, j) \quad \text{iff} \quad x \leq y \wedge \tau_y^x(j) = i.$$

Therefore, to complete the proof, it remains to show the following lemma. This is where we use the monotonicity assumption.

**Lemma C.5.** *For every  $i, j \in \{1, \dots, k\}$  there is a first-order formula  $\psi_j^i(x, y)$  such that for every  $t \in \mathcal{T}\Sigma^{[k]}$*

$$t \models \psi_j^i(x, y) \quad \text{iff} \quad \tau_y^x(j) = i.$$

*Proof.* Let  $F$  be the set of monotone partial functions from  $\{1, \dots, k\}$  to itself. Define  $L \subseteq F^*$  to be the set of those words  $f_1 \cdots f_n$  such that the composition of functions  $f_n \circ \cdots \circ f_1$  maps  $i$  to  $j$ . We will show that – thanks to the monotonicity assumption – the language  $L$  is definable in first-order logic. To get the conclusion of the lemma, we check if the sequence

of twist functions on the path from  $x$  to  $y$  satisfies the first-order formula defining the language  $L$ .

The language  $L$  is recognised by a finite automaton, which has states  $\{1, \dots, k, \perp\}$ , and which simply applies the function in its input letter to the present state. We show below that this automaton is counter-free, in the sense of McNaughton and Papert [25, p. 6], and therefore it can be defined in first-order logic.

Recall that a counter in an automaton is a sequence of at least two pairwise distinct states  $q_1, \dots, q_n$  such that

$$q_1 \xrightarrow{w} q_2 \xrightarrow{w} \cdots \xrightarrow{w} q_n \xrightarrow{w} q_1$$

holds for some common input string  $w$ . In the automaton for the language  $L$  that we have discussed above, there is no counter. Indeed, if we would have  $q_1 \leq q_2$ , then by monotonicity of the function  $w \in F$  we would have

$$q_1 \leq q_2 \leq \cdots \leq q_n \leq q_1$$

and therefore all of  $q_1, \dots, q_n$  would be equal, contradicting the assumption that they are pairwise distinct. The same argument would work when  $q_1 \geq q_2$ . By [25, Theorem 10.5], if an automaton has no counter, then its language is definable in first-order logic.  $\square$

## D Appendix on first-order relabelling

The goal of this section is to show Proposition 5.1, which says that first-order relabeling are derivable. As discussed in the body of the paper, the proof of this proposition is based on an equivalence result between first-order queries on trees and a temporal logic, as stated in Lemma 5.2. While this result is deeply inspired from a similar result of Schlingloff [26], our frameworks are not exactly the same (he uses for ainstance unranked trees). In the rest of this section, we provide more details about the reduction from Schlingloff's result to our lemma (Section D.1). Then we show in Section D.2 how to use it in order to prove Proposition 5.1.

### D.1 Reduction to Schlingloff's theorem

Let us proceed to the proof of Lemma 5.2. Clearly the functions in the lemma are first-order tree relabeling, and first-order tree relabeling are easily seen to be closed under composition, which gives the right-to-left inclusion in the lemma. The hard part is the left-to-right inclusion, which says that every first-order tree relabeling can be decomposed into functions as in items 1,2a–2c. The first step in the proof of the right-to-left inclusion is the observation that every first-order tree relabeling can be decomposed as

$$g \circ f_1 \circ \cdots \circ f_n$$

where  $g$  is a relabeling as in item 1 of the lemma and each  $f_i$  is a characteristic function of some unary query (not necessarily of the simple form indicated in items 2a – 2c in the

lemma). This is a simple observation: the functions  $f_1, \dots, f_n$  annotate the tree with the truth values of the unary queries used in the definition of the first-order relabeling, and  $g$  uses these truth values to select the appropriate output label. The hard part of the lemma is showing that each  $f_i$  can be further decomposed into functions as indicated in the lemma. This is where we use the result of Schlingloff [26, Theorem 2.6], which says that all first-order definable tree properties can be defined using a temporal logic that has operators similar to the ones used in items 2a – 2c of the lemma.

The following table summarizes our framework (first column) and Schlingloff's one (second column). The first row describes the models under consideration, the second row the corresponding version of first-order logic, and the third row the corresponding temporal logic.

Models	Trees over a finite ranked alphabet $\Gamma$ : finitely branching, ranked trees, labeled from $\Gamma$ .	Models over a set of propositions $P$ : finitely branching, unranked trees, labeled from $2^P$ .
First-order logic	Usual first-order connectives ( $\exists, \vee, \neg$ ) with the descendant predicate $x \leq y$ and the following predicates:	
	$a(x)$ : $x$ is labeled $a$ ( $a \in \Gamma$ ). $\text{child}_i(x)$ : $x$ is an $i$ -th child. <b>We call it <math>\Gamma</math>-FO.</b>	$p(x)$ : label of $x$ contains $p$ ( $p \in P$ ). <b>We call it <math>P</math>-FO.</b>
Temporal logic	Usual CTL connectives ( $S$ (Since), $U$ (Until), $\vee, \neg$ ) together with:	
	$a \in \Gamma$ , $\odot_i \phi$ : the $i$ -th child satisfies $\phi$ . <b>We call it 2-CTL.</b>	$p \in P$ , $X_i \phi$ : at least $i$ children satisfy $\phi$ . <b>We call it 4-CTL.</b>

What is named  $\Gamma$ -FO in the table is what we simply called first-order logic along the paper. The operators of 2-CTL are those of Lemma 5.2. Using the notation of the table, Schlingloff's theorem says that  $P$ -FO formulas are equivalent to 4-CTL formulas, and Lemma 5.2 states that  $\Gamma$ -FO formulas are equivalent to 2-CTL ones. To deduce the later from the former, we will show how to translate every ranked tree  $t$  over  $\Gamma$  into a model  $[t]$  over a well chosen set of propositions  $P$ , then we will apply the following scheme

$$\begin{array}{ccc}
 \varphi \in \Gamma\text{-FO} & \xleftrightarrow[\text{Lemma D.1}]{\forall t, t \models \varphi \leftrightarrow [t] \models \psi} & \psi \in P\text{-FO} \\
 & & \uparrow \text{Schlingloff's theorem} \\
 \theta \in 2\text{-CTL} & \xleftrightarrow[\text{Lemma D.2}]{\forall t, t \models \theta \leftrightarrow [t] \models \delta} & \delta \in 4\text{-CTL}
 \end{array}$$

The translation  $[\_]$  and Lemmas D.1 and D.2 are explained below.

**From ranked trees to Schlingloff's models.** Let us fix a ranked alphabet  $\Gamma$ . Let  $P$  be the following set of propositions

$$P \stackrel{\text{def}}{=} \Gamma \cup \{i\text{-th-child} \mid i \in [1, \max \text{arity of } \Gamma]\}$$

Let  $t$  be a ranked tree over  $\Gamma$ . The translation  $[t]$  of  $t$  is the model defined as follows. It has the same set of nodes and the same descendant relation as  $t$ . The label of a node contains  $a$  if its label in  $t$  is  $a$ . It contains the proposition  $i$ -th-child if it is an  $i$ -th child in  $t$ .

**First-order logic for ranked and unranked trees.** Let  $\Gamma$  and  $P$  be as above. Let us show that  $\Gamma$ -FO and  $P$ -FO are equivalent.

**Lemma D.1.** *For every  $\Gamma$ -FO formula  $\phi$ , there is a  $P$ -FO formula  $\psi$  such that*

$$\forall t \in \text{trees } \Gamma, \quad t \models \phi \leftrightarrow [t] \models \psi$$

and conversely.

*Proof.* To show this lemma it is enough to show how to translate the specific predicates of each formalism into the other. The predicate  $a(x)$  of  $\Gamma$ -FO can be translated by the same predicate in  $P$ -FO and conversely. The predicate  $\text{child}_i(x)$  can be translated by  $i$ -th-child( $x$ ) and conversely. It is clear that these translations preserve the semantics.  $\square$

**Temporal logic for ranked and unranked trees.** Let  $\Gamma$  and  $P$  be as above. We show that 2-CTL and 4-CTL are equivalent.

**Lemma D.2.** *For every  $\Gamma$ -FO formula  $\phi$ , there is a  $P$ -FO formula  $\psi$  such that*

$$\forall t \in \text{trees } \Gamma, \quad t \models \phi \leftrightarrow [t] \models \psi$$

and conversely.

*Proof.* Here again, it is enough to translate the specific connectives of each formalism into the other. The connective  $X_i$  can be encoded in 2-CTL as follows, where  $b$  is the maximal arity of  $\Gamma$

$$\bigvee_{I \subseteq [1, b]} \bigwedge_{\substack{j \in I \\ \#I=i}} \odot_j \phi$$

Conversely, the connective  $\odot_i$  can be encoded in 4-CTL as follows:

$$X_1(i\text{-th-child} \wedge \phi)$$

$\square$

## D.2 First-order relabelling are derivable

To show Proposition 5.1, saying that first-order relabelling are derivable, we will show that each function appearing in Lemma 5.2 and corresponding to each operator of 2-CTL is derivable. This is the role of Lemmas D.3–D.5 presented below.

**Lemma D.3.** For every finite  $\Sigma, \Gamma \subseteq \Sigma$  and  $i \in \{1, 2, \dots\}$ , the characteristic function  $f : \mathbb{T}\Sigma \rightarrow \mathbb{T}(\Sigma + \Sigma)$  of the unary query

“The  $i$ -th child of  $x$  is in  $\Gamma$ ”

is derivable.

*Proof.* To show that  $f$  is derivable, we start applying the children function

$$\text{Children} : \mathbb{T}\Sigma \rightarrow \mathbb{T}(\Sigma \times (\Sigma_0 + 0)^*)$$

from Example B.1 which tags every nodes by the list of its children. Consider the function  $g$

$$\begin{aligned} g : \Sigma \times (\Sigma_0 + 0)^* &\rightarrow \Sigma + \Sigma \\ (a, l) &\mapsto (a, 1) \quad \text{if } l[i] \in \Gamma_0, \\ &\mapsto (a, 2) \quad \text{otherwise.} \end{aligned}$$

which maps an element of  $\Sigma$  tagged by a list to the first copy of  $\Sigma$  if the  $i$ -th element of the list is in  $\Gamma$  and to the second copy otherwise. The function  $g$  is derivable since its domain is finite. We finally get  $f$  by lifting  $g$  to terms.  $\square$

**Lemma D.4.** For every finite  $\Gamma, \Delta \subseteq \Sigma$ , the characteristic function  $f : \mathbb{T}\Sigma \rightarrow \mathbb{T}(\Sigma + \Sigma)$  of the unary query

$$\underbrace{\exists y y \geq x \wedge \Delta(y) \wedge \forall z (x < z < y \Rightarrow \Gamma(z))}_{\substack{x \text{ has a descendant } y \text{ with label in } \Delta, \text{ such that} \\ \text{all nodes between } x \text{ and } y \text{ have label in } \Gamma}}$$

is derivable.

*Proof.* We start by applying the factorization

$$\text{fact}_\Gamma : \mathbb{T}\Sigma \rightarrow \mathbb{T}(\mathbb{T}(\Sigma \setminus (\Gamma \cup \Delta)) + \mathbb{T}(\Gamma \cup \Delta))$$

which decomposes our terms into factors, depending on whether their node labels are in  $\Gamma \cup \Delta$  or not. Note that the value of a node w.r.t. the until query depends only on the node labels of its factor.

The nodes of the  $\mathbb{T}(\Sigma \setminus (\Gamma \cup \Delta))$  factors do not satisfy the query, thus we will apply to them the function  $\mathbb{T}g$  obtained by lifting the function

$$\begin{aligned} g : \Sigma \setminus (\Gamma \cup \Delta) &\rightarrow \Sigma + \Sigma \\ a &\mapsto (a, 2). \end{aligned}$$

Nodes of the  $\mathbb{T}(\Gamma \cup \Delta)$  factors satisfy the query if and only if they have a descendant in  $\Delta$ . Consider the function  $h$  obtained by composing the descendant function  $\text{Descendant}_\Delta$  from Example B.3 with an injection  $\mathbb{T}(\iota + \iota)$

$$\underbrace{\mathbb{T}(\Gamma \cup \Delta) \xrightarrow{\text{Descendant}_\Delta} \mathbb{T}(\Gamma \cup \Delta + \Gamma \cup \Delta) \xrightarrow{\mathbb{T}(\iota + \iota)} \mathbb{T}(\Sigma + \Sigma)}_h$$

Finally, to get the characteristic function  $f$ , we apply  $\mathbb{T}g$  to the  $\mathbb{T}(\Sigma \setminus (\Gamma \cup \Delta))$  factors and  $h$  to the other factors using the co-pairing combinator, then we flat the obtained term.  $\square$

**Lemma D.5.** For every finite  $\Gamma, \Delta \subseteq \Sigma$ , the characteristic function of the unary query

$$\underbrace{\exists y y \leq x \wedge \Delta(y) \wedge \forall z (y < z < x \Rightarrow \Gamma(z))}_{\substack{x \text{ has a descendant } y \text{ with label in } \Delta, \text{ such that} \\ \text{all nodes strictly between } x \text{ and } y \text{ have label in } \Gamma}}$$

is derivable.

*Proof.* The same proof as above, one only needs to replace the use of the function  $\text{Descendant}_\Delta$  by that of  $\text{Ancestor}_\Delta$ , introduced in Example B.3.  $\square$

## E Proof of Theorem 4.3

In this part of the appendix, we prove Theorem 4.3, which says that every first-order tree-to-tree transduction is recognised by a register transducer.

According to Definition 2.2, a first-order transductions is a composition of any number of functions each of which is either copying (item 1) or a non-copying first-order transduction (item 2). In other words:

$$\begin{aligned} &\text{first-order transductions} \\ &\stackrel{\text{def}}{=} (\text{copying} \cup (\text{non-copying first-order transductions}))^* \end{aligned}$$

where the star denotes closure under composition. Although register transducers are closed under composition, this is not very easy to show directly, and therefore we begin by simplifying the function composition in the definition of first-order transductions. It is not hard to see that copying commutes with non-copying first-order transductions in the following sense:

$$\begin{aligned} &\text{copying} \circ (\text{non-copying first-order transductions}) \\ &\subseteq (\text{non-copying first-order transductions}) \circ \text{copying}. \end{aligned}$$

Furthermore, since the class of copying functions is closed under composition, and the same is true for non-copying first-order transductions, we get the following normal form of first-order transductions:

$$\begin{aligned} &\text{first-order transductions} \\ &= (\text{non-copying first-order transductions}) \circ \text{copying}. \end{aligned}$$

Therefore, in order to prove Theorem 4.3, it suffices to show that a register transducer can compute any function which first copies the nodes of the input tree a fixed number of times, and then applies a non-copying first-order transduction.

For the rest of this section, fix a tree-to-tree function

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

which is a composition of first copying (some fixed number of times), followed by a non-copying first-order transduction. We will show that  $f$  is computed by some register transducer.

In the proof, we use the origin information associated to  $f$ , i.e. how nodes of the output tree can be traced back to

nodes in the input tree. For an input tree  $t \in \text{trees}\Sigma$ , define its origin map to be the function of type

$$\text{nodes in } f(t) \rightarrow \text{nodes in } t$$

which maps a node  $x$  of the output tree to the node of the input tree that was used to define it. (The origin in a copying function is the node that is being copied, while the node in a non-copying transduction is the node of the input structure that represents the node of the output structure.) For a node  $x$  in an input tree  $t$ , define the origin colouring of  $x$  to be the function

$$y \in \text{nodes in } f(t) \mapsto \begin{cases} \text{below} & \text{if the origin of } y \text{ is } x \\ & \text{or a descendant of } x \\ \text{not below} & \text{otherwise.} \end{cases}$$

Define the name *origin factorisation of  $x$  in  $t$* , which is an element of  $\text{trees}\mathbb{T}$ , to be the factorisation of the output tree where the factors are connected parts of same type (“below” or “not below”). The origin factorisation is obtained by applying the ancestor factorisation  $\text{fact}_\uparrow$  to the output tree extended with its origin colouring.

The general idea behind the register transducer is that, after processing the subtree of a node  $x$  in the input tree, its registers will store the “below” factors in the origin factorisation of  $x$ . We only store the “below” factors, and not the “not below” factors, because only the “below” factors can be computed using register updates based on the subtree of the node  $x$  in the input tree. The key observation is the following lemma, which shows that the a constant number of registers will be enough.

**Lemma E.1.** *For every input tree  $t$  and node  $x$  in  $t$ , the origin factorisation of  $x$  in  $t$  has at most a constant (i.e. depending only on the fixed transduction) number of factors.*

*Proof.* For an input tree  $t$  and a node  $x$  in it, we say that an edge in the output tree  $f(t)$  is  *$x$ -sensitive* if its the two endpoints are in different factors of the origin colouring of  $x$  in  $t$ . The number of factors in the origin factorisation is one plus the number of sensitive edges, and therefore to prove the lemma, it is enough to show that:

(\*) for every input tree  $t$  and node  $x$  in  $t$ , there is at most a constant number of  $x$ -sensitive edges.

Let us write  $\rightarrow$  for the image – along the origin mapping – of the child relation in the output tree. In other words, nodes  $y, z$  in the input tree satisfy  $y \rightarrow z$  if some node in the output tree with origin  $z$  is a child of some node in the output tree with origin  $y$ . It is not hard to see that  $\rightarrow$  can be defined in first-order logic, using the formulas from the transduction. Let  $r$  be the quantifier rank of the first-order formula used to define  $\rightarrow$ . Using Ehrenfeucht-Fraïssé argument, one can show that if  $x, y, z$  are nodes in the input tree such that  $y$  and  $z$  are on different sides of  $x$  (i.e. any path connecting  $y$  and  $z$  must necessarily pass through  $x$ ), then the truth value

of any rank  $r$  first-order formula  $\varphi(y, z)$  depends only on the following information:

- the  $r$ -type of  $(y, x)$  in the input tree, i.e. the rank  $r$  first-order formulas satisfied by  $(y, x)$ ; and
- the  $r$ -type of  $(z, x)$  in the input tree, i.e. the rank  $r$  first-order formulas satisfied by  $(y, x)$ .

Since the relation  $\rightarrow$  has constant outdegree and indegree, and it can be defined using quantifier rank  $r$ , follows that if  $y \rightarrow z$  are on different sides of  $x$  then there can only be a constant number of nodes  $y'$  such that  $(y, x)$  and  $(y', x)$  have the same  $r$ -type in the input tree. Since the number of  $r$ -types is constant, it follows that number of pairs  $y \rightarrow z$  which are on different sides of  $x$  is constant; these pairs are the sensitive edges.  $\square$

Apply the above lemma, yielding an upper bound  $k \in \{1, 2, \dots\}$  on the number of factors in the origin factorisations. Note that each of the factors in the origin factorisation has arity  $< k$ , since the ports of the factors must lead to the other factors. It follows that, in order to store the “below” factors in registers, it is enough to have  $k$  groups of registers, with each group having one registers for every arity in  $\{0, \dots, k-1\}$ :

$$R \stackrel{\text{def}}{=} \{r_i^j \text{ of arity } j : i \in \{1, \dots, k\}, j \in \{0, \dots, k-1\}\}$$

We now define the invariant that will be satisfied by the register transducer. (We use a slightly extended model of register transducers, where some register contents can be undefined; this model is easily seen to reduce to the original one, by filling the undefined registers with some fixed nonces.

- **Invariant.** Let  $t$  be an input tree, let  $x$  be a node in  $t$ , and let  $s_1, \dots, s_n$  be the “below” factors of  $x$ , viewed as subsets of nodes in the output tree, ordered so that

$$\text{root}(s_1) \leq \dots \leq \text{root}(s_n)$$

where  $\leq$  is the pre-order on nodes in the output tree and  $\text{root}(s_i)$  denotes the unique node in  $s_i$  which is an ancestor of all other nodes in  $s_i$ . After processing the subtree of  $x$  in the input tree, the register valuation of the register transducer is

$$r_i^j \mapsto \begin{cases} s_i \text{ viewed as a term} & \text{if } s_i \text{ has arity } j \\ \text{over the output alphabet} & \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The output register of the transducer is  $r_1^0$ . When  $x$  is the root of the input tree, then there is only one “below” factor, namely the entire output tree (which has arity 0) and therefore – thanks to the invariant – the output tree will be found in the output register.

The following lemma gives the register updates of the transducer.

**Lemma E.2.** *There is a finite set  $\Delta$  of register updates with the following property. For every input tree  $t$  and every node  $x$  in  $t$ , there is some  $u \in \Delta$  such that the register valuation of  $x$  (as defined in the invariant) is obtained by applying  $u$  to the register valuations of the children  $x_1, \dots, x_n$  of  $x$ , in listed in left-to-right order. Furthermore, there is a family  $\{\varphi_u(x)\}_{u \in \Delta}$  of unary queries over the input alphabet such that the update associated to a node  $x$  is  $u$  if and only if the node satisfies  $\varphi_u(x)$  in the input tree.*

*Proof.* The crucial observation is that each of the “below” factors in the origin factorisation for  $x$  – seen as subsets of nodes in the output tree – is a (disjoint) set union of the the “below” factors in the origin factorisations for the children of  $x$ , plus the nodes in the output tree which have origin in  $x$ . Since there is at most a constant number of children and nodes with origin  $x$ , there is a finite number – depending only on the transduction – of ways in which these factors can be combined; this finite set of possible combinations is the set  $\Delta$ . The “furthermore” part of the lemma, about computing the update using first-order queries, follows from a simple inspection of the first-order formulas used in defining the transduction.  $\square$

The above lemma completes the definition of the register transducer. Its register updates are  $\Delta$  as in the lemma, and its transition function assigns label  $u \in \Delta$  to each node that satisfies  $\varphi_u(x)$ . The final part of the proof is showing that the register updates are monotone. We use the following order on the registers:

$$\underbrace{r_1^0 < r_1^1 < \dots < r_1^{k-1} < r_2^0 < r_2^1 < \dots < r_k^{k-2} < r_k^{k-1}}_{\text{lexicographic, with the lower index having priority}}$$

Let  $x$  be a node in an input tree  $t$ , and let  $s_1, s_2$  be a “below” factor in the origin factorisation of  $x$ , which are register contents in register valuation of  $x$ . The registers storing  $s_1$  and  $s_2$  will be ordered – according to the invariant – with respect to the pre-order on the root nodes of  $s_1$  and  $s_2$ . Let  $x'$  be the parent of  $x$ . By the reasoning in the proof of Lemma E.2, there are “below” factors in the origin factorisation of  $x'$  which contain the factors  $s_1$  and  $s_2$ ; call these factors  $s'_1$  and  $s'_2$  (possibly  $s'_1 = s'_2$ ). Since  $s'_1$  contains  $s_1$  (as a set of nodes in the output tree), and the same is true for  $s'_2$  and  $s_2$ , we have

$$\text{root}(s_1) \leq \text{root}(s_2) \quad \text{implies} \quad \text{root}(s'_1) \leq \text{root}(s'_2)$$

which establishes monotonicity of the register updates.

This completes the proof of Theorem 4.3.

## F Normalisation of $\lambda$ -terms is a first-order transduction

In this part of the appendix, we show Theorem 6.1, which says that under some restrictions, normalisation of  $\lambda$ -terms is a first-order transduction. Before proving this result in F.3,

we will first explain in F.1 why these restrictions are unavoidable. Then we show in F.2 that the set of  $\lambda$ -terms satisfying these restrictions form a first-order tree language. This result will be useful for the proof of Theorem 6.1.

### F.1 Explaining the restrictions

Recall that Theorem 6.1 says that normalisation of  $\lambda$ -terms is derivable under two assumptions: the input term should be linear, and could be typed using a fixed finite set of types.

If the linearity condition is removed, and because of iterated duplication, the normal form of a well-typed  $\lambda$ -term can be exponential (or worse, see [27, Section 3.6]), as shown by the following example.

**Example F.1.** Assume that we have two variables  $x^o$  and  $y^{o \rightarrow o \rightarrow o}$  and consider the  $\lambda$ -terms defined by:

$$M_0 \stackrel{\text{def}}{=} x^o \quad M_{n+1} = (\lambda x^o. y^{o \rightarrow o \rightarrow o} x^o x^o) M_n.$$

The  $\lambda$ -term  $M_n$  is well-typed and of type  $o$ . It has size linear in  $n$ , but its normal form has size at least  $2^n$ .

If there was a first-order transduction normalising these terms, it would be exponential-size increase, which is not possible since all first-order transductions are linear-size increase.

Being linear alone is not enough to normalise terms with first-order transductions. Another obstacle is terms that use types of unbounded complexity, as illustrated in the following example.

**Example F.2.** Consider the following  $\lambda$ -terms, which have types of unbounded size:

$$M_n = \overbrace{\lambda x^o. \lambda x^o. \dots \lambda x^o. x^o}^{n \text{ times}}$$

This is a well-typed affine term, whose type is

$$o^n \rightarrow o \quad \stackrel{\text{def}}{=} \quad \overbrace{o \rightarrow o \rightarrow \dots \rightarrow o}^{n+1 \text{ arrows}}$$

To  $M_n$ , apply  $m$  arguments of type  $o$ :

$$M_n \overbrace{y^o y^o \dots y^o}^{m \text{ times}}. \quad (9)$$

We claim that the above  $\lambda$ -term cannot be normalised using a first-order transduction, or even a monadic second-order transduction. In order to normalise, a transduction would need to be able to compare the numbers  $n$  and  $m$  as follows: if  $m < n$  the normal form contains  $\lambda$ , if  $m = n$  the normal form does not contain  $\lambda$ , and if  $m > n$  then the normal form is undefined because the  $\lambda$ -term is not well-typed. Whether or not a  $\lambda$ -term (seen as a tree over a finite alphabet) contains  $\lambda$  is a first-order definable property, and first-order definable properties are preserved under inverse images of first-order transductions. Therefore, if normalisation would be a first-order transduction, then there would be a first-order formula

which would be true for terms of the form (9) with  $m > n$  and which would be false for terms of the form (9) with  $m = n$ . Such a formula cannot exist, which can be shown using a pumping argument or Ehrenfeucht-Fraïssé games.

## F.2 Restrictions of Theorem 6.1 are first-order definable

In this section, we show that the restrictions of Theorem 6.1 discussed above, are first-order definable, as stated in the following theorem.

**Proposition F.3.** *Let  $X$  be a finite set of simply typed variables and let  $\mathcal{T}$  be a finite set of simple types. The tree language of linear  $\lambda$ -terms which can be typed using  $\mathcal{T}$  is first-order definable.*

In the rest of this appendix, we denote by  $\Lambda_{\mathcal{T}X}$  this tree language. To prove Proposition F.3, we first show that for  $\lambda$ -terms in  $\Lambda_{\mathcal{T}X}$ , checking if their type is  $\tau$ , where  $\tau$  is a type in  $\mathcal{T}$ , is a FO property:

**Lemma F.4.** *For every type  $\tau$  in  $\mathcal{T}$ , there is a first-order query  $\varphi_{\tau}$  such that:*

$$\forall M \in \Lambda_{\mathcal{T}X} \quad M, u \models \varphi_{\tau} \iff M|_u : \tau$$

where  $M|_u$  is the sub-tree of  $M$  rooted in  $u$ .

Before establishing this lemma, let us see how Proposition F.3 can be derived from it. Linearity can be easily seen as a first-order property. The hard part is to show that the set of  $\lambda$ -terms which can be typed using  $\mathcal{T}$  is first-order. Suppose for convenience that  $\mathcal{T}$  is downward closed. For every type  $\tau$  in  $\mathcal{T}$ , let  $\varphi_{\tau}$  be the formula given by Lemma F.4. In the following, we use the binary formula  $\text{Succ}_i(u, v)$  which is valid when  $v$  is the  $i$ -th child of  $u$ , and which is easily expressible in first-order logic.

Consider the unary formula  $\text{Lambda}(u)$ , which expresses that  $u$  is a binder node, that its type and the type of its child match well and both belong to  $\mathcal{T}$ :

$$\begin{aligned} \text{Lambda}(u) := & \underbrace{\lambda x(u)}_{u \text{ has label } \lambda x} \wedge \bigvee_{\substack{\sigma \rightarrow \tau \in \mathcal{T} \\ x:\sigma}} \underbrace{\varphi_{\sigma \rightarrow \tau}(u)}_{u \text{ has type } \sigma \rightarrow \tau} \\ & \wedge \underbrace{\exists v \text{ Succ}_1(u, v) \wedge \varphi_{\tau}(v)}_{\text{the child of } u \text{ has type } \tau} \end{aligned}$$

Similarly, consider the unary formula  $\text{Application}(u)$  which checks that a node is an application node, that the type of its children match well and that both belong to  $\mathcal{T}$ :

$$\begin{aligned} \text{Application}(u) := & \underbrace{@(u)}_{u \text{ has label } @} \wedge \\ & \exists v, w \bigvee_{\sigma \rightarrow \tau \in \mathcal{T}} \underbrace{\text{Succ}_1(u, v) \wedge \varphi_{\sigma \rightarrow \tau}(v)}_{\text{the left child of } u \text{ has type } \sigma \rightarrow \tau} \wedge \underbrace{\text{Succ}_2(u, w) \wedge \varphi_{\sigma}(w)}_{\text{the right child of } u \text{ has type } \sigma} \end{aligned}$$

Finally, consider the formula  $\text{Variable}(u)$ , which expresses that  $u$  is a variable node, whose type is in  $\mathcal{T}$ :

$$\text{Variable}(u) := \bigvee_{x:\sigma \in \mathcal{T}} \underbrace{x(u)}_{u \text{ has label } x}$$

We claim that the following (nullary) formula  $\phi$  recognizes the tree language  $\Lambda_{\mathcal{T}X}$

$$\phi = \forall u. \text{Variable}(u) \vee \text{Lambda}(u) \vee \text{Application}(u)$$

If a  $\lambda$ -term is in  $\Lambda_{\mathcal{T}X}$ , then it clearly satisfies  $\phi$ . Suppose by contradiction that there is a  $\lambda$ -term  $M$  which is not in  $\Lambda_{\mathcal{T}X}$  and yet satisfies  $\phi$ . Let  $u$  be the deepest node of  $M$  which is not in  $\Lambda_{\mathcal{T}X}$  (we identify in this proof a node  $u$  and the sub-term  $M|_u$ ). In particular, the descendants of  $u$  are all in  $\Lambda_{\mathcal{T}X}$ . The node  $u$  cannot be a variable, since variable nodes are well-typed and their type is in  $\mathcal{T}$  by the first disjunct of  $\phi$ . If  $u$  was labeled by  $\lambda x$ , where  $x$  is of type  $\sigma$ , then by the second disjunct of  $\phi$  there is a type  $\tau$  such that  $\sigma \rightarrow \tau \in \mathcal{T}$  and the child  $v$  of  $u$  satisfies  $\varphi_{\tau}$ . Since  $v$  is in  $\Lambda_{\mathcal{T}X}$ , its type is  $\tau$  by Lemma F.4. Hence  $u$  is well-typed and its type is  $\sigma \rightarrow \tau \in \mathcal{T}$ . As a consequence  $u$  is in  $\Lambda_{\mathcal{T}X}$  which is a contradiction. Finally, if  $u$  was labeled by  $@$ , then by the third disjunct of  $\phi$ , its two children  $u_1$  and  $u_2$  would satisfy respectively  $\varphi_{\sigma \rightarrow \tau}$  and  $\varphi_{\sigma}$  and by Lemma F.4 they are of type  $\sigma \rightarrow \tau$  and  $\sigma$  respectively. The node  $u$  is then well-typed and its type is  $\tau$  (which is a type of  $\mathcal{T}$  thanks to downward closeness). As a consequence,  $u$  is in  $\Lambda_{\mathcal{T}X}$ , which gives a contradiction and concludes the proof.

We can go back now to the proof of Lemma F.4.

*Proof of Lemma F.4.* Let us show that the following unary query is expressible in first-order logic

“if  $t$  is a  $\lambda$ -term of  $\Lambda_{\mathcal{T}X}$ , then its type is  $\tau$ ”:

For that, notice that the type of a well-typed term depends only on its left-most branch. In fact, the type of a term is exactly the type of its left-most branch in the following sens.

Consider the (unranked) alphabet  $X^{\lambda} := X \cup \{ @, \lambda x \mid x \in X \}$ . We can equip the words over  $X^{\lambda}$  with the following typing rules:

$$\frac{}{x : \sigma} \quad \frac{u : \tau}{u \lambda x : \sigma \rightarrow \tau} \quad \frac{u : \sigma \rightarrow \tau}{u @ : \tau}$$

where  $x$  is of type  $\sigma$  and  $\sigma, \tau \in \mathcal{T}$ .

We say that  $w$  is of type  $\tau$  and write  $w : \tau$  if there is a typing derivation for  $w : \tau$ .

We can associate to every branch of a  $\lambda$ -term a word over  $X^{\lambda}$  corresponding to the sequence of its labels read bottom-up. By induction on  $\lambda$ -terms, we can easily show that the type of a  $\lambda$ -term is the type of the word corresponding to its leftmost branch.

By this last observation, we can reduce the query asking if the type of a term is  $\tau$ , to the same query but on  $X^{\lambda}$  words.

To show that the former is a first-order query, it is then sufficient to show that the following word language

$$W_\tau = \{w \in X.\{@, \lambda x|x \in X\}^* \mid w : \tau\}$$

is first-order definable, or equivalently that  $W_\tau$  is recognized by a counter-free finite automaton. For that we proceed as follows: first, we show that  $W_\tau$  is recognized by a pushdown automaton  $P_\tau$ . Then we will show that the stack height of  $P_\tau$  is bounded, thus it can be turned into a deterministic finite automaton  $D_\tau$ . Finally, we show that the obtained automaton  $D_\tau$  is actually counter-free.

Consider the pushdown automaton  $P_\tau$  whose

- set of states is  $\{i, p, f\}$ , where  $i$  is the initial state and  $f$  the accepting state;
- input alphabet is the alphabet  $X^\lambda$ ;
- stack alphabet is the set of types  $\mathcal{T}$ ;
- and whose transition function is described as follows:
  - If the automaton is in the initial state  $i$  with an empty stack, and if the symbol it reads is a variable  $x$  of type  $\sigma_1 \rightarrow \dots \rightarrow \sigma_n$ , then we go to the state  $p$  and push the symbols  $\sigma_n, \dots, \sigma_1$  in the stack in this order. The top-level symbol of the stack is then  $\sigma_1$ .
  - If the automaton is in the state  $p$  and it reads the symbol  $\lambda y$ , where  $y$  is of type  $\sigma$ , then push the symbol  $\sigma$  in the stack, and stay in the state  $p$ .
  - If the automaton is in the state  $p$ , if it reads the symbol  $@$  and if the stack is non empty, then pop the top-level symbol and stay in the state  $p$ .
  - If the automaton reaches the end of the word being in state  $p$ , and if the stack contains the symbols  $\tau_1, \dots, \tau_m$  in this order,  $\tau_1$  being the top-level symbol, where  $\tau_1 \rightarrow \dots \rightarrow \tau_m$  is the type  $\tau$ , then pop them all and go to the final state  $f$ .

A word  $w$  is accepted by  $P_\tau$  if there is a run that reaches the end of  $w$  in the accepting state  $f$  with an empty stack. We write  $(r, s) \xrightarrow{w} (r', s')$  if there is a run over the word  $w$  which starts in the state  $r \in \{i, p, f\}$  and with a stack  $s$  and ends up in the state  $r' \in \{i, p, f\}$  and with a stack  $s'$ .

By induction on the length of the word  $w$ , we can easily show that:

**Lemma F.5.** *For every word  $w \in X.\{@, \lambda x|x \in X\}^*$ , we have that:*

$$(i, \epsilon) \xrightarrow{w} (p, \sigma_n \dots \sigma_1) \quad \text{iff} \quad w : \sigma_1 \rightarrow \dots \rightarrow \sigma_n$$

A direct consequence of this lemma is that  $P_\tau$  recognizes  $W_\tau$ . Another direct consequence is that the stack height of  $P_\tau$  is bounded by  $m$ , the size of the longest type in  $\mathcal{T}$ . Thus  $P_\tau$  can be turned into a DFA  $D_\tau$ , by encoding the stack information in the states. More precisely, the states of  $D_\tau$  are pairs  $(r, s)$  where  $r \in \{i, p, f\}$  and  $s$  is a stack of height at most  $m$ , the initial state is  $(i, \epsilon)$  and there is a transition  $(r, s) \xrightarrow{a} (r', s')$  where  $a \in X^\lambda \cup \{\epsilon\}$  if there is a corresponding run in  $P_\tau$ . We show in the following that  $D_\tau$  is counter-free.

Let us start with some observations. In the pushdown automaton  $P_\tau$ , the effect of a word  $w$  on a stack  $s$ , starting from the state  $p$  is the following: it erases the first  $n$  top level elements of  $s$ , and replaces them by a word  $u$ . The number  $n$  and the word  $u$  do not depend on the stack  $s$  but only on the word  $w$ . This is exactly what the following lemma claims.

**Lemma F.6.** *For every word  $w$  over  $X^{\lambda^*}$ , there is a natural number  $n$  and a word  $u \in \mathcal{T}^*$  such that if  $(p, s) \xrightarrow{w} (p, s')$  then  $s$  and  $s'$  can be decomposed as follows:*

$$s = t.v, \quad s' = t.u \quad \text{and} \quad |v| = n.$$

The proof is an easy induction on the length of  $w$ . As a consequence we have that:

- If  $(p, s_1) \xrightarrow{w} (p, s_2) \xrightarrow{w} (p, s_3)$  and  $|s_2| > |s_1|$  then  $|s_3| > |s_2|$ .
- If  $(p, s_1) \xrightarrow{w} (p, s_2) \xrightarrow{w} (p, s_3)$  and  $|s_2| < |s_1|$  then  $|s_3| < |s_2|$ .
- If  $(p, s_1) \xrightarrow{w} (p, s_2) \xrightarrow{w} (p, s_3)$  and  $|s_2| = |s_1|$  then  $s_3 = s_2$ .

Let us show that  $D_\tau$  is counter-free. Suppose by contradiction that there is a word  $w$  and pairwise distinct stacks  $s_1, \dots, s_n$  such that

$$(p, s_1) \xrightarrow{w} (p, s_2) \xrightarrow{w} \dots (p, s_n) \xrightarrow{w} (p, s_1).$$

By the first two properties above, we have necessarily that

$$|s_1| = \dots = |s_n|$$

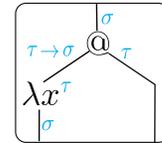
Thus by the third property, we have that

$$s_1 = \dots = s_n$$

which concludes the proof.  $\square$

### F.3 Normalisation of $\lambda$ -terms

This section is dedicated to the proof of Theorem 6.1. Let us first introduce some terminology. In a  $\lambda$ -term, we call *redex* a pattern of the following form



that is, an application node whose left child is an abstraction node. In a well-typed  $\lambda$ -term, we define the *type of a redex* to be the type of the left sub-term of its application node. In the figure above, the type of the redex is  $\tau \rightarrow \sigma$ . If the abstraction node of a redex is labeled by  $\lambda x$ , we say that  $x$  is *the variable of the redex* and that this redex is an  *$x$ -redex*.

Let us go back to the proof of Theorem 6.1. Before normalising  $\lambda$ -terms, the first thing to do is to discriminate those  $\lambda$ -terms satisfying the restrictions of Theorem 6.1 from the

others. This amounts to pre-processing the normalisation process by the function

$$\text{trees}^{X^\lambda} \rightarrow \text{trees}^{X^\lambda} + \perp$$

which is the identity for inputs satisfying the restrictions and is undefined otherwise. Let us see how this function can be derived. Thanks to Proposition F.3, the restrictions of Theorem 6.1 are first-order definable, say by a first-order query  $\phi$ . By virtue of Proposition 5.1, the characteristic function of  $\phi$  is derivable. Now following the label's root of the input, we either output the input tree if the label says that it satisfies the query  $\phi$ , or outputs the undefined symbol otherwise. This last function can be easily derived. From now on, we suppose that our  $\lambda$ -terms satisfy the restrictions of Theorem 6.1.

To normalise  $\lambda$ -terms, we proceed by induction on the set of types  $\mathcal{T}$ . The main observation is that, if the evaluation of a redex of type  $\sigma \rightarrow \tau$  creates new redexes, then their types are either  $\sigma$  or  $\tau$ . They belong, in particular, to  $\mathcal{T} \setminus \{\sigma \rightarrow \tau\}$ . As a consequence, we only need to show that the function that evaluates all the redexes of a fixed type is derivable. As we only create strictly smaller redexes, we need to iterate this process only finitely many times, the bound being the size of  $\mathcal{T}$ .

Since we have only finitely many typed variables, it is enough to show that the function that evaluates all the redexes of a fixed type  $\sigma$  and a fixed variable  $x$  is derivable. This will be our goal in the rest of this section.

**Proposition F.7.** *Let  $X$  be a finite set of simply typed variables,  $\mathcal{T}$  be a finite set of simple types,  $x$  be a variable in  $X$  and  $\sigma$  a simple type in  $\mathcal{T}$ . The following tree-to-tree function is derivable:*

- **Input.** A  $\lambda$ -term  $t$  over variables  $X$ .
- **Output.** The  $\lambda$ -term obtained by evaluating in  $t$  all the  $x$ -redexes of type  $\sigma$ , if  $t$  is in  $\Lambda_{\mathcal{T}}X$ , and undefined otherwise.

To show this proposition, we will factorise (via a derivable function) our  $\lambda$ -terms into factors satisfying the following properties:

- (P1) All the  $x$ -redexes of type  $\sigma$  fall entirely into one of the factors.
- (P2) Each factor have a a very specific shape called *thin*. These factors are those  $\lambda$ -terms with ports whose normal form have the shape of a word (by opposition to trees, which is the general case).

By properties (P1) and (P2), it is enough to show that normalisation of thin  $\lambda$ -terms (with ports) is derivable. For this purpose, our strategy will be to prove that the word obtained by normalising a thin  $\lambda$ -term results from a pre-order traversal. Since pre-order traversal is a prime function, this implies that normalisation of thin  $\lambda$ -terms is derivable.

The last ingredient to conclude the proof is to notice that  $\beta$ -reducing the factors of (a factorisation of) a  $\lambda$ -term, then

applying a flattening, is the same thing as  $\beta$ -reducing the original  $\lambda$ -term, which follows directly from the fact that  $\beta$ -reduction is a congruence on terms. This concludes the proof.

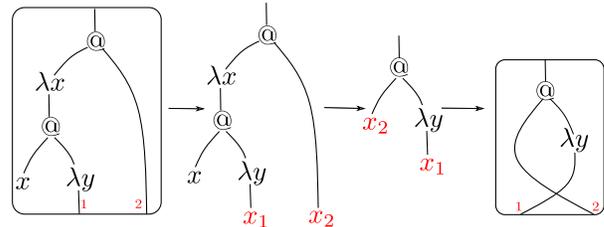
In the rest of this section, we develop on each of the two main steps of the proof, namely proving Properties (P1) and (P2). In Section F.3.1 we present thin  $\lambda$ -terms with ports and show how to normalise them. Then we show in Section F.3.2 how to factorise a  $\lambda$ -term into thin factors.

### F.3.1 Normalisation of thin $\lambda$ -terms

As discussed earlier, we will need to normalise  $\lambda$ -terms with ports (the factors of our factorisation). In the following, we will denote by  $X^\lambda$  the ranked set

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

With this notation,  $\lambda$ -terms with ports are the inhabitants of  $\text{TX}^\lambda$ . Normalisation of these terms generalizes that of usual  $\lambda$ -terms in a straightforward way: the  $i$ -th port is replaced by a fresh variable  $x_i$ , the obtained  $\lambda$ -term (without ports) is evaluated as usual, then the variable  $x_i$  is replaced back by the port  $i$ , as one can see in the following example.



Note that when a  $\lambda$ -term is linear, its normal form has the same number of ports. Note also that respecting the original order of ports in the normal form (which is important for compositionality) may twist ports, as in the example above. As a consequence, normalisation of linear  $\lambda$ -terms with ports is an arity preserving function of type:

$$\text{TX}^\lambda \rightarrow \text{F}_1 \text{TX}^\lambda + \perp$$

Let us present now the class of *thin  $\lambda$ -terms with ports*.

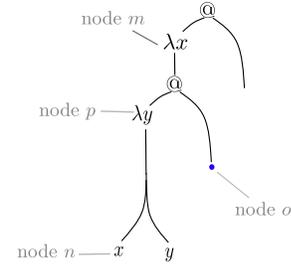
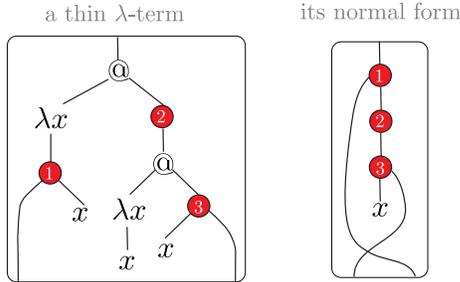
**Definition F.8.** We say that the node of a  $\lambda$ -term is *branching* if its has at two distinct children which are not ports.

A *thin  $\lambda$ -term with ports* is a term from  $\text{TX}^\lambda$  in which every branching node is the application node of a redex.

In the remaining of this section we will omit the mention “with ports” if clear from the context.

Since thin  $\lambda$ -terms branch only on redexes, the result of their normalisation is a “word”, in the sens that every node has at most one non-port child. We will show that this word can actually be obtained by a pre-order traversal of the original  $\lambda$ -term. We will then use the prime **preorder** function to show that normalisation of thin  $\lambda$ -terms is derivable.

The left  $\lambda$ -term below is linear and thin. The red nodes are the ones which are not redexes nor the variables of these redexes. The right  $\lambda$ -term is its normal form: we can see that nodes appear top-down in the pre-order of the original  $\lambda$ -term.



By induction hypothesis, the variable bound by  $p$  is strictly greater than  $n$ . It is also strictly smaller than  $o$ , which gives a contradiction and concludes the proof.  $\square$

**Proposition F.9.** Let  $X$  be a finite set of simply typed variables,  $\mathcal{T}$  be a finite set of simple types. The following tree-to-tree function is derivable:

- **Input.** A  $\lambda$ -term  $t$  over variables  $X$ .
- **Output.** The normal form of  $t$ , if it is linear and thin, and undefined otherwise.

Let  $t$  be a thin  $\lambda$ -term and let  $u$  be its normal form. As noticed before,  $u$  has the shape of a word. Moreover, since  $t$  is linear, the nodes of  $u$  are exactly the nodes of  $t$  which are not redexes, nor their variables.

**Proposition F.10.** Let  $t$  be a linear thin  $\lambda$ -term and let  $u$  be its normal form. The order in which the inner nodes (ie. non ports) of  $u$  appear top-down is the pre-order of  $t$ .

*Proof.* To establish this proposition, we need the following lemma.

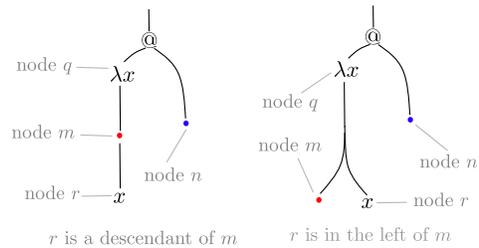
**Lemma F.11.** Let  $t$  be a linear thin  $\lambda$ -term and let  $r$  be one of its redexes. Consider  $m$  to be the binder node of  $r$  and  $n$  to be its variable node.

The node  $n$  is the greatest (that is the left-most) node in the sub-term  $t|_m$  w.r.t. the pre-order.

*Proof.* We proceed by induction on the length of the path between  $m$  and  $n$ . When it is 0 the result is clear. Suppose by contradiction that it is strictly greater than 0 and that there is a node  $o$  which is strictly greater than  $n$ . We take  $o$  to be the smallest node which is greater than  $n$ . Since  $t$  is thin, the least common ancestor  $l$  between  $n$  and  $o$  is an application node of a redex. Since  $n$  is smaller than  $o$ ,  $n$  is the left descendant of  $l$ , in other words it is the descendant of the left child  $p$  of  $l$ , which is a binder. The node  $m, n, o$  and  $p$  are illustrated below:

Let us go back to the proof of our proposition. Consider two inner nodes  $n, m$  of  $t$  which are also nodes of  $u$ , and such that  $m$  is smaller than  $n$  in the pre-order of  $t$  (we will call it simply pre-order in the rest of the proof). We show that  $n$  is a descendant of  $m$  in  $u$ . There are two cases to consider:

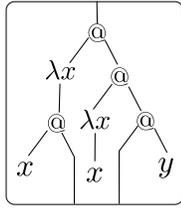
- Either  $n$  is a descendant of  $m$  in  $t$ , in this case we can conclude easily since  $\beta$ -reduction preserves the descendant relation. Indeed, by a small analysis of  $\beta$ -reduction, one can notice that a reduction step may extend the descendant relation, but can never change (or break) the order of two comparable nodes in the original  $\lambda$ -term.
- Otherwise, let us consider the lowest common ancestor  $p$  of  $m$  and  $n$ . We proceed by induction on the length of the path between  $m$  and  $p$ . By definition of thin  $\lambda$ -terms, since  $p$  is branching it is necessarily an application node, whose left child  $q$  is a binder node, let us say  $\lambda x$ . By Lemma F.11,  $m$  is smaller w.r.t. the pre-order than the node  $r$  of the variable bound by  $q$ . We are then left with the following two situations. The first case, illustrated by the left figure below, is when  $r$  is a descendant of  $m$  in  $t$ . In this case, after one reduction step  $n$  will be a descendant of  $m$ . The other case is when  $m$  is in the left of  $r$  in  $t$ , as illustrated by the right figure below. In this case, after one reduction step, the lowest common ancestor between  $m$  and  $n$  will be a descendant of  $p$ , and we can conclude by induction hypothesis.



This concludes the proof of the first claim.  $\square$

Let us construct now a derivable function which computes the normal form of linear thin  $\lambda$ -terms. We illustrate this

construction on the term  $t$  below which will be our running example in this proof.



*Proof of Proposition F.9.* Let  $t$  be a linear thin  $\lambda$ -term in  $\mathbf{TX}^\lambda$ .

1. We start by distinguishing the redexes of  $t$  and their variables from the other nodes. For that, we apply the characteristic function of the following first-order query  $\varphi$ :

“The node  $u$  is a redex or a variable of a redex”

This query is first-order expressible. Indeed it is the disjunction of the following queries

$$\text{@Redex}(u) = \text{@}(u) \wedge \exists v \text{Child}_1(u, v) \wedge \forall_{x \in X} \lambda x(v)$$

$$\lambda\text{Redex}(u) = \lambda x(u) \wedge \exists v \text{Child}_1(v, u) \wedge \text{@}(v)$$

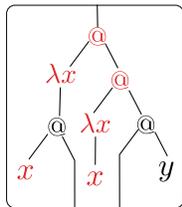
$$X\text{Redex}(u) = \forall_{x \in X} x(u) \wedge \exists v \lambda\text{Redex}(v) \wedge v \text{ binds } u$$

where  $\text{@Redex}(u)$  says that  $u$  is the application node of a redex,  $\lambda\text{Redex}(u)$  says that it is the abstraction node of a redex and  $X\text{Redex}(u)$  says that it is the variable of a redex. The formula  $u$  binds  $v$ , defined below, is a binary first-order query expressing that the node  $u$  is an abstraction node that binds  $v$ .

$$\bigvee_{x \in X} \lambda x(u) \wedge x(v) \wedge u < v \wedge \forall u < w < v \neg \lambda x(w)$$

The query  $\varphi$  being first-order, its characteristic function is derivable thanks to Proposition 5.1.

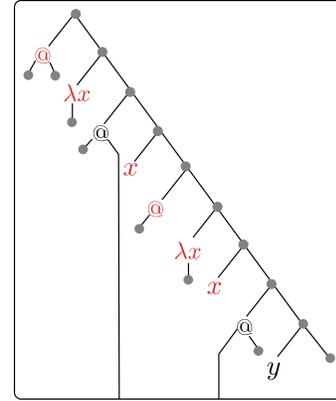
When we apply this function to  $t$ , we get a term in  $\mathbf{T}(X^\lambda + X^\lambda)$  term below. Below is the effect of this first step on our running example. We colored in red the nodes belonging to the first copy of  $X^\lambda$ , that is the nodes satisfying the query  $\varphi$ . These nodes are the ones that will disappear in the normal form of  $t$ .



2. After that, we apply the **preorder** function

$$\text{preorder} : \mathbf{T}(X^\lambda + X^\lambda) \rightarrow \mathbf{F}_1\mathbf{T}(X^\lambda + X^\lambda + 0 + 2)$$

After this step, our initial term becomes



In this term, the nodes of the normal form appear in the right order thanks to Prop. F.10. Now, we only need to get rid of the redexes and the variable nodes that participated in the computation of the normal form (that is the ones colored in red) together with the nodes  $\bullet$  and  $\blacktriangle$  introduced by the **preorder** function.

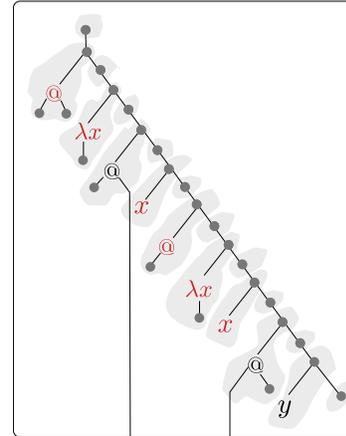
3. For this purpose, we apply the function

$$\mathbf{T}(X^\lambda + X^\lambda + 0 + 2) \rightarrow \mathbf{T}(X^\lambda + X^\lambda + 0 + 2 + 1)$$

which adds the unary symbol 1 as the parent of every node 2. This function can be easily implemented using the derivable homomorphism function of Example ???. Then we apply the factorisation **fact<sub>↑</sub>** to separate the symbol 1 from the others:

$$\text{fact}_\uparrow : \mathbf{T}(X^\lambda + X^\lambda + 0 + 2 + 1) \rightarrow \mathbf{T}(\mathbf{T}(X^\lambda + X^\lambda + 0 + 2) + \mathbf{T}1)$$

After this step, our example term becomes like this



4. Now consider the function

$$g : \mathbf{T}(X^\lambda + X^\lambda + 0 + 2) \rightarrow \mathbf{F}_1\mathbf{T}(X^\lambda + X^\lambda + 0 + 2)$$

which is the identity function, except for the following finite set of terms for which it is defined in figure 8. The red elements are those belonging to the first copy of  $X^\lambda$ .

Now back to our term, we replace the **T1** factors by the empty term, and to the other factors we apply the

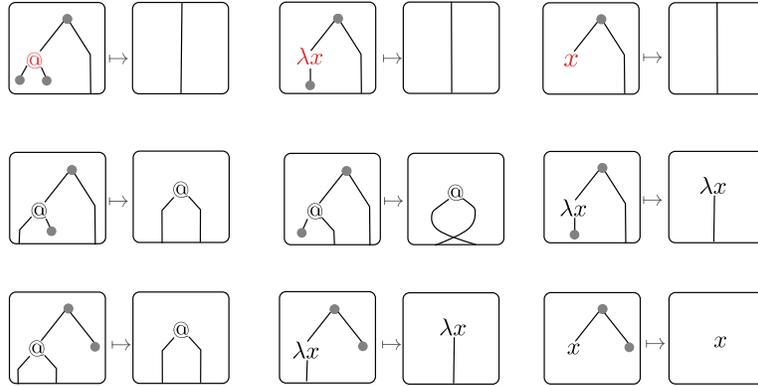


Figure 8. Definition of the function  $g$ .

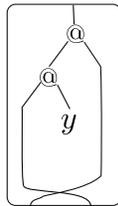
function  $g$ . After that, we apply the function

$$F_1 F_1 \Sigma \rightarrow F_1 \Sigma$$

which untwists two consecutive applications of  $F_1$ . Doing so, we get a term of type

$$F_1 T(X^\lambda + X^\lambda + 0 + 2)$$

which is the normal form of  $t$ . Our running example becomes then

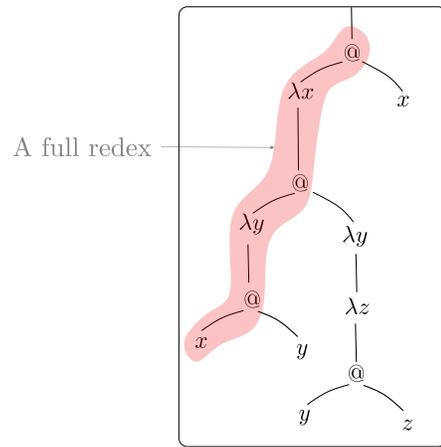


- Note that we obtained the desired term, but not with the desired type. To obtain a term in  $F_1 TX^\lambda$ , we get rid of the labels  $0 + 2$  by transforming them respectively into variables and application nodes. The choice of which variables to choose is not important, since the only terms that will actually have  $0 + 2$  in their results are  $\lambda$ -terms which are not linear or not thin.

□

### F.3.2 Factorising $\lambda$ -terms into blocks of thin $\lambda$ -terms

In a linear  $\lambda$ -term, we call *full redex* a set of nodes containing a redex, the node of its variable, together with the set of nodes between them, as illustrated below



**Proposition F.12.** For every finite set of typed variables  $X$ , for every finite set of types  $\mathcal{T}$  and for every  $x \in X$  and  $\sigma \in \mathcal{T}$ , there is a factorisation

$$f : TX^\lambda \rightarrow \Pi TX^\lambda + \perp$$

which satisfies, for every linear  $\lambda$ -term  $t$ , that

- every full  $x$ -redex of type  $\sigma$  in  $t$  is entirely contained in one of the factors of  $f(t)$ ;
- the factors of  $f(t)$  are thin.

and is undefined otherwise.

*Proof.* We define the function  $f$  as the composition of the following three functions

$$TX^\lambda \xrightarrow{g} T(X^\lambda + 1) \xrightarrow{\text{block}^\uparrow} T(TX^\lambda + T1) \xrightarrow{\text{erase}} \Pi TX^\lambda$$

The function  $g$  will indicate, using the unary symbol 1, the places where two distinct blocks of  $f$  will be separated. We will describe it more precisely a bit later. The function  $\text{block}^\uparrow$  will create these blocks and finally, we erase all the factors  $T1$ .

The function  $\text{block}^\uparrow$  is a prime function and erase can be easily derivable. Let us show how to derive the function  $g$ , so that the 1-nodes it introduces creates blocks satisfying the

conditions (1) and (2) of Proposition F.12 (when the input is a linear  $\lambda$ -term).

We define  $g$  as the composition of the characteristic function of three first-order unary queries: @redex, Right and Left, followed by a homomorphism  $h$ . We define them in the following:

- The property @Redex checks whether a node is the application node of an  $x$ -redex of type  $\sigma$ . It can be expressed by the following first-order formula, where  $\varphi_\sigma$  is a first-order formula which decides if the type of a node is  $\sigma$  (for instance the one given by Lemma F.4):

$$\text{App}(u) := @ (u) \wedge \exists v \text{ Succ}_1(u, v) \wedge \lambda x(v) \wedge \varphi_\sigma(v)$$

- The query Right (resp. Left) checks if the node is an application node, which lies, together with his right (resp. left) child, between the application node of an  $x$ -redex of type  $\sigma$  and the node it binds. Those properties can be easily expressed by a first-order formula.

When we apply the characteristic functions of these queries to a term in  $\text{TX}^\lambda$ , each node will be decorated by three informations: whether it satisfies or not App, whether it satisfies or not Right and whether it satisfies or not Left. Note that for linear  $\lambda$ -terms, some combinations of these properties cannot hold in the same node. For instance, a node cannot satisfy Right and Left simultaneously, as this would contradict linearity.

Now we define the homomorphism  $h$ , which maps the  $\lambda$ -terms with these three informations to terms of  $\text{T}(X^\lambda + 1)$ . We define the action of  $h$  on each node, depending on its label and the three informations it contains:

- If the label of the node is  $y$  or  $\lambda y$  for some variable  $y \in X$ , or if the label is @ and satisfies App, then  $h$  returns the same node (seen as a term), forgetting the extra three informations.
- If the node is an application node satisfying

$$\neg \text{App} \wedge \neg \text{Right} \wedge \neg \text{Left}$$

then  $h$  adds 1 to the two children of the node.

- If the node is an application node satisfying

$$\neg \text{App} \wedge \neg \text{Right} \quad (\text{resp. } \neg \text{App} \wedge \neg \text{Left})$$

then  $h$  adds 1 to the left (resp. right) child of the node.

Let  $t$  be a linear  $\lambda$ -term. We show that the factors induced by  $g$  satisfy the two conditions of Proposition F.12. First of all, by analyzing the action of  $h$  on each node, note that every application node will receive 1 as one of its children, except when it satisfies App. Thus the only branching nodes in a factor are redexes, hence the factors are thin. Now suppose by contradiction that there is some full  $x$ -redex of type  $\sigma$  of  $t$  which is not entirely contained in a factor. This means that in  $g(t)$  there is a 1 between the application node of some  $x$ -redex of type  $\sigma$  and its variable. By construction of  $h$ , 1 is the child of an application node (call it  $n$ ). Suppose w.l.o.g. that it is the right child of  $n$ . The node  $n$  cannot satisfy App

because it got 1 as a child by  $h$ . It satisfies Right by the contradiction hypothesis. Thus it satisfies  $\neg \text{App} \wedge \neg \text{Right}$ , therefore it receives also 1 as its left child by  $h$ . This means that  $n$  received 1 for its both children, and the only way to get that is to satisfy  $\neg \text{App} \wedge \neg \text{Right} \wedge \neg \text{Left}$ , which gives a contradiction.  $\square$

## G Decomposing the unfolding function

As discussed in the main body of the paper, the unfolding function may be regarded as unsatisfactory. In this section, we will decompose it into a collection of small functions containing no form of iteration.

We present these new prime functions in Section G.1, and state the main result of this section which is that term unfolding can be derived from these new prime functions (and the other prime functions of Section 3). To prove this result, our strategy is to show that term unfolding can be derived for a restricted class of terms that we call *homogeneous*, and then to show that every term can be factorised into homogeneous terms.

The notion of homogeneous terms, and the result about decomposing arbitrary terms into homogeneous ones, are presented in Section G.2. Next, in Section G.3, we show how term unfolding can be done for homogeneous inputs. Finally, in Section G.4 we prove the main result of the section by combining the results of Sections G.2 and G.3.

### G.1 New prime functions replacing the unfolding

In order to decompose the unfolding function, we enrich datatypes with the constructor of shallow terms introduced in Section A.1.

We present the prime functions which will replace the unfolding in Figures 9–12. Prime functions of Figure 9 describe the behaviour of the shallow term datatype. Figure 10 contains some additional laws for the fold datatype and Figure 11 contains some new distributivity laws. Prime functions of Figure 12 are weak versions of the unfolding function, containing no form of iteration.

Note that some of these functions were already presented in Appendix A.3 to define formally the unfolding function: distributivity of shallow terms over fold, distributivity of shallow terms over product (Figure 11), and the matching function (Figure 12). In appendix A.3, those functions were introduced in a very formal (hence verbose) way. In this appendix, we made the opposite choice of giving only informal definitions through some hopefully clear and unambiguous pictures.

The main result of this section is that the unfolding can be replaced by the more atomic functions of Figures 9–12, in presence of the prime functions presented in Section 3, as stated in the following theorem

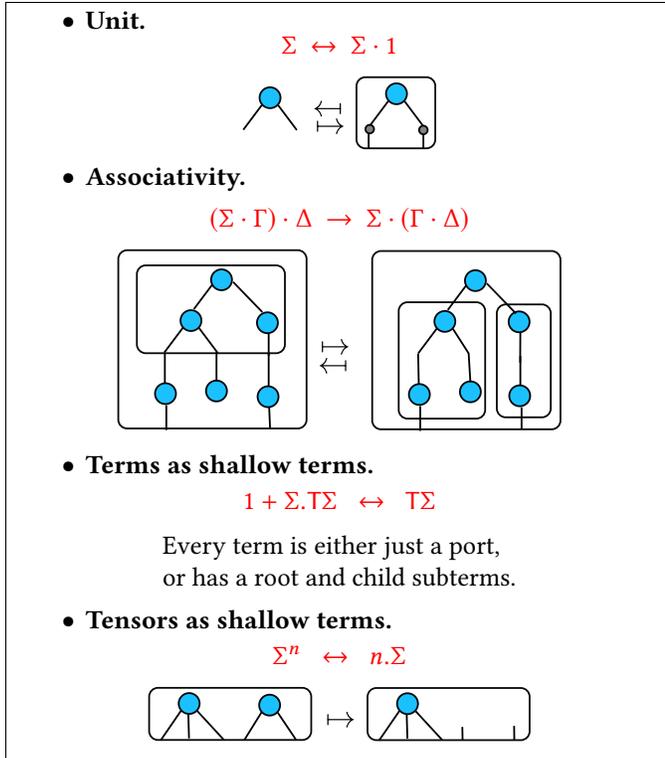


Figure 9. Prime functions for shallow terms.

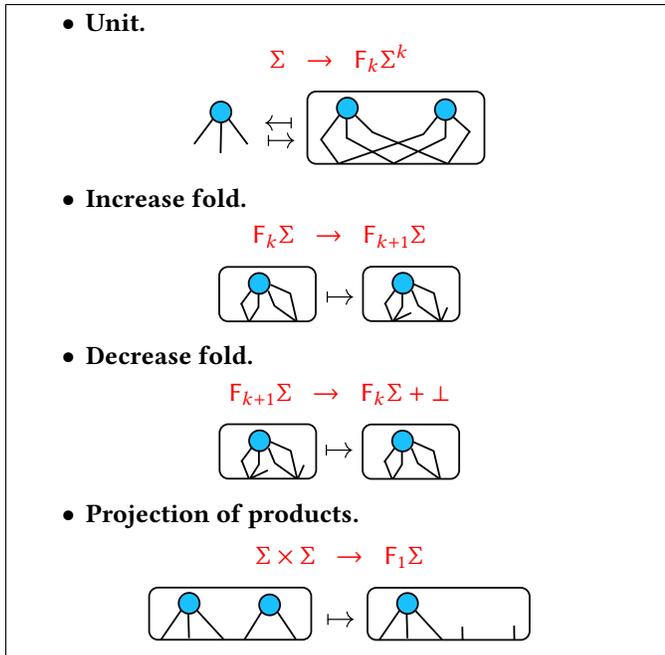


Figure 10. Additional prime functions for folds.

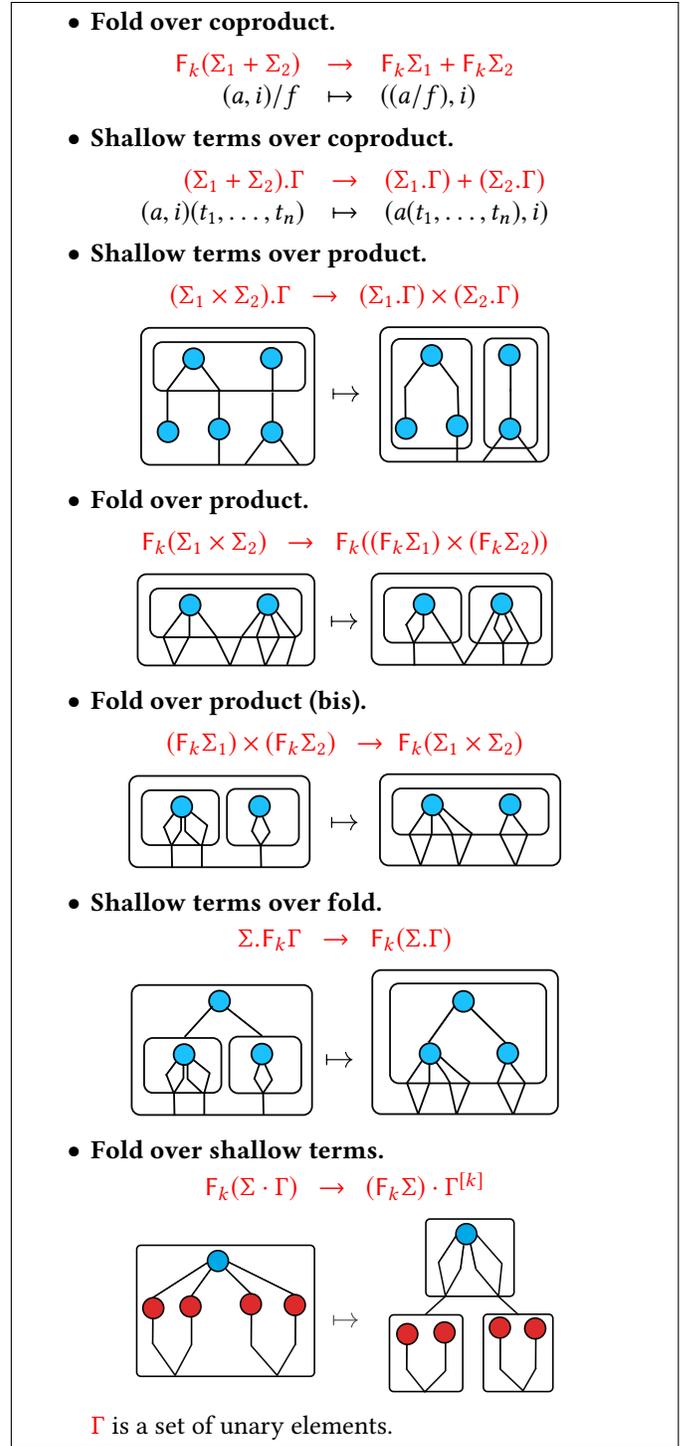


Figure 11. Additional distributivity prime functions.

**Theorem G.1.** *The unfolding function can be derived using the functions of Figures 9–12 and the prime functions of Section 3.*

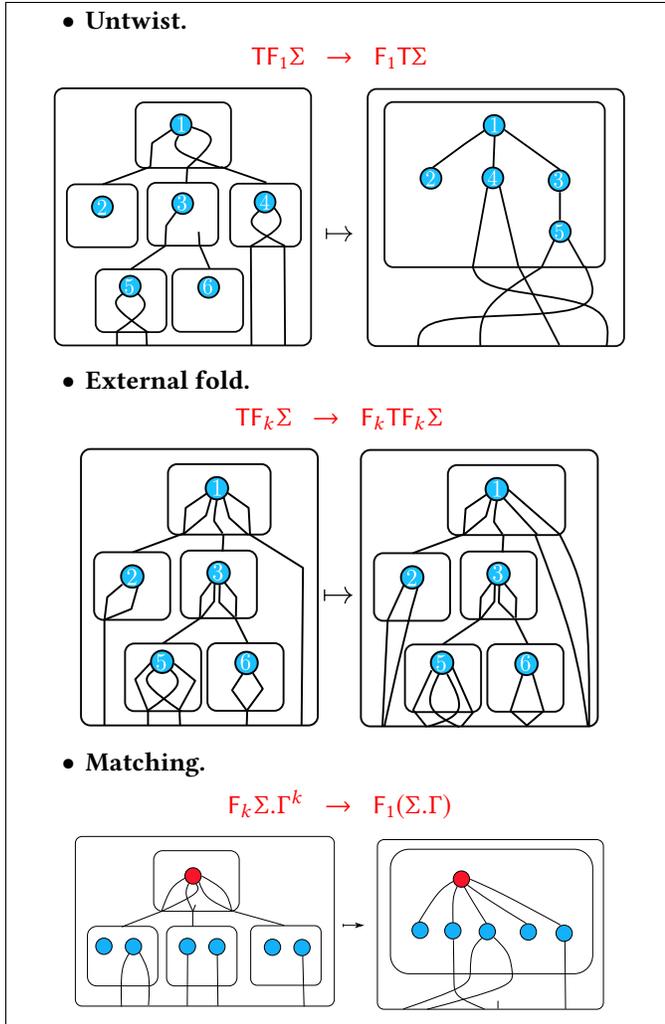


Figure 12. Weak forms of unfolding.

In the rest of Appendix G, derivable means derivable from the prime functions of Figures 9–12 and the prime functions of Section 3 except from unfolding.

### G.2 Factorisation forests

This section is devoted to stating and proving a tree version of the Factorisation Forest Theorem of Imre Simon. Our result differs from the original Factorisation Forest Theorem in the following ways: (a) we consider trees instead of strings; (b) we use aperiodic finite monoids instead of arbitrary finite monoids; and (c) the factorisation in the conclusion of the theorem can be computed by a derivable function. A tree generalisation of the Factorisation Forest Theorem was already proved by Colcombet [14, Theorem 1 and Section 3.3], but Colcombet’s result is proved for monadic second-order logic, and therefore it does not satisfy condition (c).

**Factorisation forests** The idea behind factorisation forests is to split a term into a nested factorisation, which is a term of terms of terms, and so on up to a certain depth. Define a *nested factorisation* of depth  $k \in \{1, 2, \dots\}$  over alphabet  $\Sigma$  to be an element of  $T^k\Sigma$  which is defined by

$$T^0\Sigma = \Sigma \quad \text{and} \quad T^{k+1}\Sigma = TT^k\Sigma.$$

Nested factorisations can be flattened to terms by using an operation  $\text{flat}^k : T^k\Sigma \rightarrow T\Sigma$  defined by

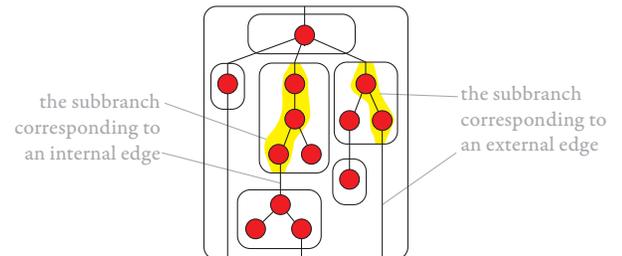
$$\text{flat}^1 = \text{identity} \quad \text{and} \quad \text{flat}^{k+1} \stackrel{\text{def}}{=} \text{flat} \circ T(\text{flat}^k).$$

An equivalent definition of  $\text{flat}^{k+1}$  would be  $\text{flat}^k \circ T^{k-1}\text{flat}$ , the equivalence of these definitions corresponds to the fact that  $T$  is a monad.

**Branches and subbranches** Define a *branch* in a ranked set to be an element of the ranked set together with a distinguished port. We draw branches like this:



We write  $B\Sigma$  for the (unranked) set of branches over a ranked set  $\Sigma$ . For a term, we classify its edges as internal (linking a non-port node with a non-port child) and external (linking a non-port node with a child port). Each edge in a term  $t \in T\Sigma$  corresponds to a branch over  $\Sigma$ , namely the branch which leads to the edge. Any branch obtained this way is called a *subbranch* of  $t$ . Here is a picture of subbranches in the case of a term of terms:



Branches in terms form a monoid. Using the monoid structure of branches in terms, we can extend any function  $h : B\Sigma \rightarrow M$ , with  $M$  a monoid, to a monoid homomorphism

$$h^{(n)} : BT^n\Sigma \rightarrow M$$

which maps a branch of a term to the product – in the monoid  $M$  – of all of its subbranches (after flattening). A more formal definition is that  $h^{(0)}$  is the same as  $h$ , while  $h^{(n+1)}$  is the unique monoid homomorphism which makes the following diagram commute

$$\begin{array}{ccc} BT^n\Sigma & & \\ \text{Bunit} \downarrow & \searrow h^{(n)} & \\ BT^{n+1}\Sigma & \xrightarrow{h^{(n+1)}} & M \end{array}$$

The idea behind factorisation forests, as expressed in Definition G.2 below, is to factorise a term into a term of terms of terms (etc.) so that the depth of nesting is bounded, and at each level all branches behave regularly with respect to some monoid homomorphism.

**Definition G.2** (Homogeneous factorisations). Let  $h : B\Sigma \rightarrow M$  be a function into a monoid  $M$ .

- We say that a factorisation  $t \in T\Sigma$  is *homogeneous with respect to  $h$*  if it either:
  1. it is a shallow term (which means that all internal edges originate from the root); or
  2. all internal subbranches of  $t$  have the same value under  $h^{(1)}$ ; or
  3. if  $a, b \in M$  appear as values – under  $h^{(1)}$  – of internal branches in  $t$ , then  $ab = a$ .
- We say that a nested factorisation  $t \in T^n\Sigma$  is *hereditarily homogeneous with respect to  $h$*  if either  $n = 1$  and  $t$  is the unit of a letter, or  $n \geq 2$  and both:
  1. it is homogeneous with respect to  $h^{(n-1)}$ ; and
  2. every node has a label in  $T^{n-1}\Sigma$  that is hereditarily homogeneous with respect to  $h$ .

Recall that a finite monoid is aperiodic if it has only trivial subgroups. An equivalent definition is that every element  $m$  of the monoid satisfies

$$\exists n \in \{1, 2, \dots\} m^n = m^{n+1}.$$

A famous theorem of Schützenberger, McNaughton and Papert, see [28, Theorem VI.1.1] says that the languages of words recognised by homomorphisms into finite aperiodic monoids are exactly those that can be defined in first-order logic. This is the reason why we consider aperiodic monoids.

**Example G.3.** Let  $k \in \{1, \dots\}$  and consider the monoid of partial functions

$$\{1, \dots, k\} \rightarrow \{1, \dots, k\}.$$

This monoid is not aperiodic, because it contains the group of all permutations of  $\{1, \dots, k\}$ . Consider now the restriction of this monoid to partial functions which are monotone (this is a monoid, because such functions are closed under composition). This monoid is aperiodic, because if  $f$  is a partial function, then for every  $i \in \{1, 2, \dots, k\}$  the sequence

$$f^1(k), f^2(k), f^3(k), \dots$$

reaches a fixpoint (or becomes undefined) in at most  $k$  steps.

We are now ready to state our version of the Factorisation Forest Theorem.

**Theorem G.4** (Factorisation Forest Theorem). *Let  $\Sigma$  be a ranked set and let  $h : B\Sigma \rightarrow M$  be a function into a finite aperiodic monoid  $M$ . There is some  $n \in \{1, 2, \dots\}$  and a function*

$$f : T\Sigma \rightarrow T^n\Sigma$$

*such that  $\text{flat}^n \circ f$  is the identity on  $T\Sigma$ , and all outputs of  $f$  are hereditarily homogeneous with respect to  $h$ . Furthermore, if  $\Sigma$  is finite<sup>7</sup> then  $f$  is derivable.*

In the proof below, the constructions are designed so that they can be formalised using derivable functions, however we leave the details of the “Furthermore” part to the reader.

Define a *good set* to be any subset  $X \subseteq T\Sigma$  which admits a function

$$f : T\Sigma \rightarrow T^n\Sigma \quad \text{for some } n \in \{1, 2, \dots\}$$

such that  $\text{flat}^n \circ f$  is the identity on  $T\Sigma$ , and  $f$  restricted to  $X$  produces only hereditarily homogeneous outputs. Our goal is to show that the entire set  $T\Sigma$  is good. To prove this, we use a more refined result, stated below, which has a parameter that can be used for induction. We say that a term  $t \in T\Sigma$  uses  $A \subseteq M$  for internal subbranches if all internal subbranch have image under  $h^{(1)}$  that belongs to  $A$ .

**Lemma G.5.** *Let  $\Sigma$  be a ranked set and let  $h : B\Sigma \rightarrow M$  be a monoid homomorphism into a finite aperiodic monoid  $M$ . For every  $A \subseteq M$ , the terms that use  $A$  for inner subbranches is good.*

Theorem G.4 follows immediately from the lemma, by taking  $A$  to be the entire monoid. The rest of Section G.2 is therefore devoted to proving the lemma. The proof is by induction on two parameters: (a) the size of  $A$ ; and (b) the size of the semigroup  $\langle A \rangle \subseteq M$  that is generated by  $A$ . These parameters are ordered lexicographically, with the size of the semigroup being more important.

The induction base is when  $A$  contains only one element  $a$  of the monoid. If a term uses  $\{a\}$  for internal subbranches, then applying **Tunit** leads to a factorisation that is homogeneous according to item 2 of Definition G.2, which is also hereditarily homogeneous because all nodes are labelled by units. This completes the proof of the induction base.

In the proof of the induction step, we consider two cases.

- The first case is when every  $a \in A$  satisfies

$$\langle \{ba : b \in \langle A \rangle\} \rangle = \langle A \rangle$$

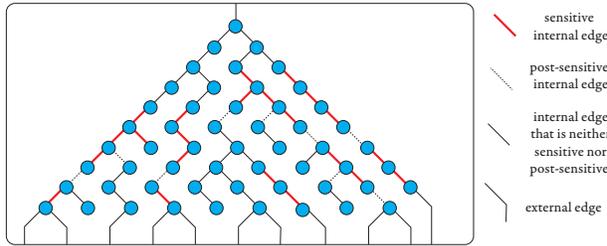
This means that every for every  $a \in A$ , the function  $b \mapsto ba$  is a permutation of  $\langle A \rangle$ . Since the monoid is aperiodic, this permutation must necessarily be the identity. Therefore, we have  $ab = a$  for every  $a, b \in \langle A \rangle$ . This means that if all a term uses  $A$  for internal subbranches, then applying **Tunit** gives a factorisation which is hereditarily homogeneous according to item 3 of Definition G.2.

- If the previous item does not hold, then there is some  $a \in A$  such that

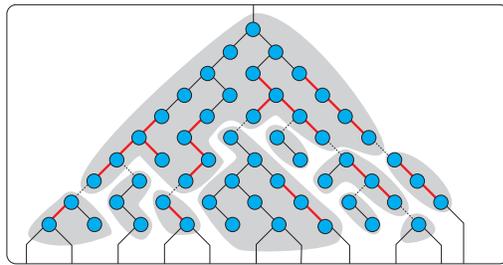
$$\langle \{ba : b \in \langle A \rangle\} \rangle$$

<sup>7</sup>This finiteness assumption could be relaxed by saying that  $\Sigma$  is possibly infinite but the function  $h$  is derivable, in the sense that a derivable function can decorate the ports of an element in  $\Sigma$  by their values under  $M$ .

is a proper subsemigroup of  $\langle A \rangle$ . Fix some such  $a$ . Define a *sensitive edge* in a term  $t \in \mathbb{T}\Sigma$  to be any internal edge where the corresponding subbranch has value  $a$  under  $h^{(1)}$ . Call an internal edge *post-sensitive* if it is not sensitive, but its parent edge is. Here is a picture:

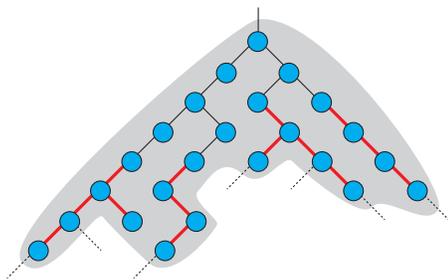


Define the *split* of a term to be the factorisation which cuts along post-sensitive edges, as shown in the following picture:

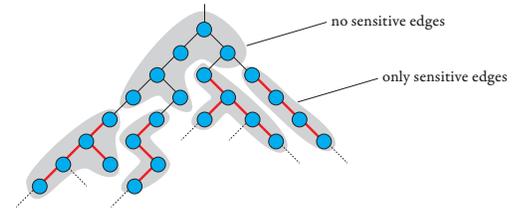


We only consider splits for terms which use  $A$  for internal subbranches. Roughly speaking, we will show that all factors in the split are good, and the split itself is good. Combining these two observations, we will see that all terms

We begin by looking at the factors in the split (of a term where  $A$  is used for internal subbranches). Here is a picture of such a factor:



If we follow a branch in an factor of the split, from root to port, we first have a sequence of non-sensitive edges from the original term, followed by a sequence of sensitive edges. Group the non-sensitive edges together, and group the sensitive edges together, resulting in a shallow term from  $\mathbb{T}\Sigma.\mathbb{T}\Sigma$ , which is illustrated in the following picture:



In the resulting shallow term, the root is labelled by a term without sensitive edges (i.e. it is a term which uses  $A - \{a\}$  for internal subbranches), while the children are labelled by terms where all edges are sensitive (i.e. they are terms which use  $\{a\}$  for internal subbranches). We can apply the induction in both cases, and combine the resulting nested factorisations using a shallow term, as in item 1 of Definition G.2.

Having established that the factors of the split are good, we turn to the split itself. By construction, every subbranch of the split is mapped by  $h^{(1)}$  to the smaller semigroup

$$\langle \{ba : b \in \langle A \rangle\} \rangle,$$

We can view the split as a term over alphabet  $\Gamma = \mathbb{T}\Sigma$ . Since all internal subbranches of the split are in the smaller subsemigroup, we can apply the induction assumption of the lemma (with  $\Gamma$  and  $h^{(1)}$ ), showing that the split is good. More formally, the set

$$\{\text{split of } t : t \in \mathbb{T}\Sigma \text{ uses } A \text{ for internal subbranches}\}$$

is good. To show now that original set of terms  $t$  that use  $A$  for internal branches is good, we first apply the split, then compute the nested factorisation for the split, and finally we compute the nested factorisations for the factors of the split (the letters from  $\Gamma$ ).

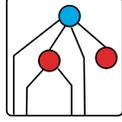
### G.3 Term unfolding for homogeneous inputs

The goal of this section is to show that term unfolding is derivable for homogeneous inputs. Actually, we will first show that unfolding is derivable for another particular case of inputs which we call *constant-twists*. Then we will use this function as a macro to unfold the homogeneous inputs.

#### G.3.1 Unfolding constant-twist functions

In the proof of this section, it will be sometimes convenient to manipulate *partial shallow terms*, that is shallow terms where some children of the root maybe ports. We will define them more precisely, and show that unfolding matrix power of partial shallow terms is derivable.

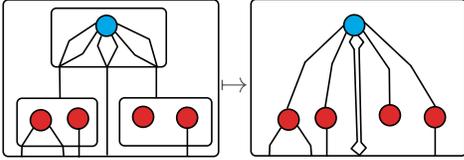
**Partial shallow unfold** If  $\Gamma$  and  $\Delta$  are types, we define  $\Gamma \odot \Delta$  to be  $\Gamma.\langle \Delta + \{1\} \rangle$ . We call its inhabitants the *partial shallow terms*. A partial shallow term looks like this, where we omitted to draw the element 1



We define the *partial shallow unfolding* function as the extension of the shallow unfold function of Figure 12 to partial shallow terms. It is the function of type

$$F_k \Sigma \odot \Gamma^k \rightarrow F_k(\Sigma \odot \Gamma)$$

defined as in the following picture



the partial shallow unfolding function can be derived as follows. Consider the functions  $f$  and  $g$  defined as follows

$$\begin{aligned} f : \Gamma^k &\xrightarrow{\text{Increase-fold}} F_k \Gamma^k \xrightarrow{F_k(t_1)^k} F_k(\Gamma + 1)^k \\ g : 1 &\xrightarrow{k\text{-Unit}} F_k 1^k \xrightarrow{F_k(t_2)^k} F_k(\Gamma + 1)^k \end{aligned}$$

We start by lifting  $f$  and  $g$  as follows

$$F_k \Sigma \odot \Gamma^k \stackrel{\text{def}}{=} F_k \Sigma \cdot (\Gamma^k + 1) \rightarrow F_k \Sigma \cdot F_k(\Gamma + 1)^k$$

We compose the obtained function with the prime function which distributes the shallow product over the fold:

$$F_k \Sigma \cdot F_k(\Gamma + 1)^k \rightarrow F_k(F_k \Sigma \cdot (\Gamma + 1)^k)$$

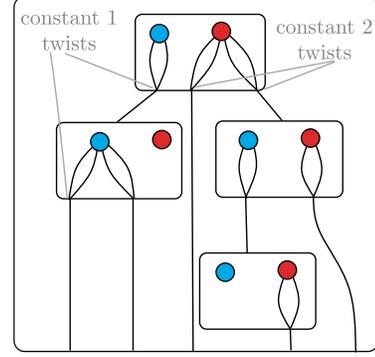
Now we can apply the shallow unfold function, more precisely we lift it along the constructor  $F_k$ . Then we compose the result with the product of the graded monad:

$$F_k(F_k \Sigma \cdot (\Gamma + 1)^k) \xrightarrow{F_k \text{Shallow unfold}} F_k F_1(\Sigma \cdot (\Gamma + 1))$$

Then we compose the result with the product of the graded monad, to obtain the desired function

$$F_k F_1(\Sigma \cdot (\Gamma + 1)) \xrightarrow{\text{flat}} F_k \Sigma \cdot (\Gamma + 1) \stackrel{\text{def}}{=} F_k \Sigma \odot \Gamma$$

**Term unfolding for constant-twist inputs** We say that a term  $t \in \mathbb{T}\Sigma^{[k]}$  is a *constant-twist term* if each twist of an internal branches is a constant function. Note that the internal twists need not to be the same constant function. Here is an example of a constant-twist term



This section is devoted to proving the following lemma.

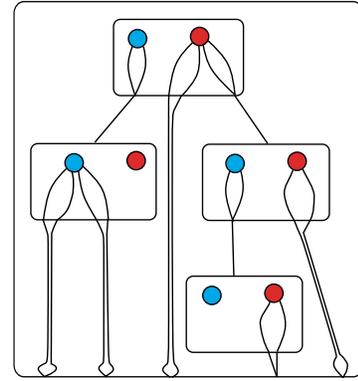
**Lemma G.6.** *Let  $k \in \{1, 2, \dots\}$ . There is a derivable function*

$$f : \mathbb{T}\Sigma^{[k]} \rightarrow (\mathbb{T}\Sigma)^{[k]}$$

*which coincides with unfolding for all constant-twist inputs.*

*Proof.* The function  $f$  can be derived using the following steps. We use the example of the constant-twists term above as a running example.

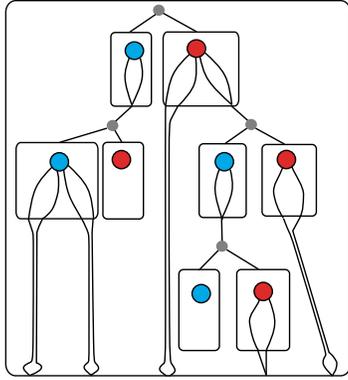
1. We start by applying the external unfolding function. We get a term in  $F_k \mathbb{T}\Sigma^{[k]}$ . Our example becomes like this



2. Next, we will transform each matrix power node into a tensor product as follows

$$\Sigma^{[k]} \rightarrow F_k(F_k \Sigma \times \dots \times F_k \Sigma) \rightarrow F_1(F_k \Sigma)^k$$

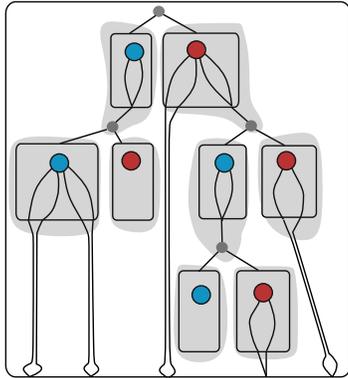
The idea here is that, since the image of each twist is a singleton, the ports of the matrix power are independent. We can then transform safely each node into a tensor product. After that, we transform each tensor product  $(F_k \Sigma)^k$  into a shallow term  $k \cdot F_k \Sigma$  which we see itself as a term of type  $\mathbb{T}(k + F_k \Sigma)$ . After the application of the unfolding function  $\text{unfold}_1$  followed by a flattening, and the simplification of  $F_k F_1 \mathbb{T}(k + F_k \Sigma)$  into  $F_k \mathbb{T}(k + F_k \Sigma)$  we get a term in  $F_k \mathbb{T}(k + F_k \Sigma)$ . Our running example becomes as follows after this step



3. Now we apply the factorization

$$T(k + F_k\Sigma) \rightarrow TT(k + F_k\Sigma)$$

which regroups each element  $F_k\Sigma$  with its children of type  $k$  in the same factor, and leaves the other nodes in isolated factors. At this point our term looks like this



Note that this factorization have the following shape: the root is labeled by  $k$ , and all the other nodes have labels in  $(F_k\Sigma) \odot k$ . We want to reflect this structure in the type by applying the following function

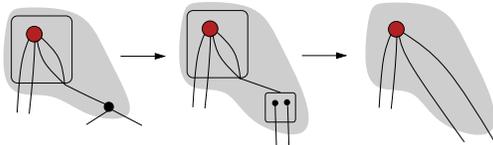
$$TT(k + F_k\Sigma) \rightarrow k.T((F_k\Sigma) \odot k)$$

This function can be implemented easily using the decomposition function, the functions which maps every type to  $1$  and mechanisms of raising errors.

4. In each node  $(F_k\Sigma) \odot k$ , we transform  $k$  into  $1^k$  (this function is basic since its domain is finite). After that, we apply the partial shallow unfold function, composed with the prime functions eliminating the  $1$  and decreasing the fold:

$$(F_k\Sigma) \odot k \rightarrow F_k(\Sigma \odot 1) \rightarrow F_k\Sigma \rightarrow F_1\Sigma$$

as illustrated by the following picture



5. At this point we have a term of type  $F_k(k \cdot TF_1\Sigma)$ . We apply the untwist function

$$TF_1\Sigma \rightarrow F_1T\Sigma$$

then the function which transforms shallow terms into tensor product

$$k \cdot F_1T\Sigma \rightarrow (F_1T\Sigma)^k$$

Now our term is of type  $F_k(F_1T\Sigma)^k$ . To conclude, we apply the prime function which permutes the tensor product with the fold, then we apply the product of the graded monad. □

### G.3.2 Term unfolding for $\alpha$ -homogeneous inputs

For a monotone function

$$\alpha : \{1, \dots, k\} \rightarrow \{1, \dots, k\}$$

we say that a term  $t \in T\Sigma^{[k]}$  is  $\alpha$ -homogeneous if all internal branches have twist  $\alpha$ . This section is devoted to proving the following lemma.

**Lemma G.7.** *Let  $k \in \{1, 2, \dots\}$  and let  $\alpha : \{1, \dots, k\} \rightarrow \{1, \dots, k\}$  be a monotone function. There is a derivable operation*

$$f : T\Sigma^{[k]} \rightarrow (T\Sigma)^{[k]}$$

which coincides with term unfolding for all inputs which are  $\alpha$ -homogeneous.

*Proof.* We proceed by induction on  $k$ . When  $k = 1$ , the unfolding coincides with the basic distributivity function

$$TF_1\Sigma \rightarrow F_1T\Sigma$$

Let us treat the inductive case. For that, we introduce a tool that will be useful to analyze the function  $\alpha$ . For a function

$$\alpha : \{1, \dots, k\} \rightarrow \{1, \dots, k\}$$

define its *graph* as the directed graph whose set of vertices is  $\{1, \dots, k\}$ , and which contains an edge  $i \rightarrow j$  if  $\alpha(i) = j$ . Note that the out-degree of the nodes is 1.

In the proof of the inductive case, we distinguish two cases. The first one is when the graph of  $\alpha$  is not weakly connected. In this case, by monotonicity of  $\alpha$ , we can find  $m \in \{1, k-1\}$  such that  $\alpha(\{1, m\}) \subseteq \{1, m\}$  and  $\alpha(\{m+1, k\}) \subseteq \{m+1, k\}$ . The idea is then to create two copies of the original term: in the first one we keep only the first  $m$  elements of the tensor product of each node, and in the second one we keep the last  $k-m$  copies. Then we unfold these terms by applying the induction hypothesis, and finally we gather them to obtain the unfolding of the original term.

Let us now implement the ideas we discussed above. We start by unfolding the external twists, using the basic external unfold function. This way, the domain of every external twist cannot be shared by the two disconnected components of

the domain of  $\alpha$ . Then, we duplicate the input term using the basic function

$$\mathbb{T}\Sigma^{[k]} \rightarrow F_2(\mathbb{T}\Sigma^{[k]} \times \mathbb{T}\Sigma^{[k]})$$

To the first copy, we apply the function

$$f_1 : \mathbb{T}\Sigma^{[k]} \rightarrow (\mathbb{T}\Sigma)^{[m]}$$

which keeps only the first  $m$  elements of the tensor product, then applies the induction hypothesis to the obtained term. To the second copy, we apply the function

$$f_2 : \mathbb{T}\Sigma^{[k]} \rightarrow (\mathbb{T}\Sigma)^{[k-m]}$$

which keeps only the last  $k - m$  elements of the tensor product, then applies the induction hypothesis to the obtained term.

The function  $f_1$  can be derived using the tensor projection function, the merge of folds, then reducing the fold and finally invoking the induction hypothesis.

When we apply  $f_1$  and  $f_2$  to the two copies of the original term, we get a term of type

$$F_2((\mathbb{T}\Sigma)^{[m]} \times (\mathbb{T}\Sigma)^{[k-m]})$$

At this point, we are almost done, we only need to transform the type in order to match the desired type. For that we increase the fold by applying the following prime functions

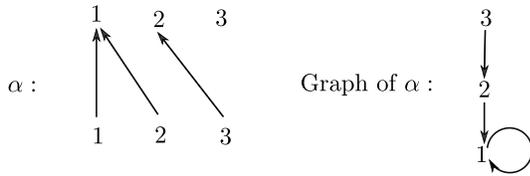
$$F_m(\mathbb{T}\Sigma)^m \rightarrow F_k(\mathbb{T}\Sigma)^m \quad F_{k-m}(\mathbb{T}\Sigma)^{k-m} \rightarrow F_k(\mathbb{T}\Sigma)^k$$

We swap the fold with the tensor product using the corresponding prime function, then we decrease the fold. This concludes the proof of the first case.

Now consider the case where the graph of  $\alpha$  is weakly connected. By monotonicity, we can show that either

$$\alpha^{-1}(1) = \emptyset \quad \text{or} \quad \alpha^{-1}(k) = \emptyset$$

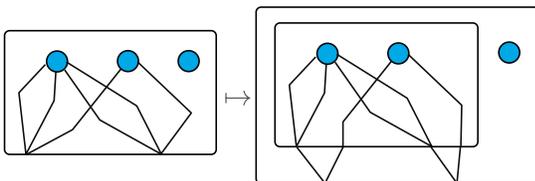
By symmetry, we suppose wlog that  $\alpha^{-1}(k) = \emptyset$ . We suppose also that  $\alpha(k) = k - 1$ , the general case can be treated in a similar way. We consider as example the following function  $\alpha$ , whose graph, drawn below, is weakly connected



Consider the function

$$F_k \Sigma^k \rightarrow F_2(F_{k-1} \Sigma^{k-1} \times \Sigma)$$

which acts as in the following picture



If we inject both  $F_{k-1} \Sigma^{k-1}$  and  $\Sigma$  into  $\Delta \stackrel{\text{def}}{=} F_{k-1} \Sigma^{k-1} + \Sigma$ , we get a term in the matrix power  $\Delta^{[2]}$ . Note that the twists of such elements are the constant 1. We can then apply the unfolding function for constant-twists inputs

$$\mathbb{T}\Delta^{[2]} \rightarrow \mathbb{T}\Delta^{[2]}$$

We can decompose the term  $\mathbb{T}\Delta$  into  $(\mathbb{T}F_{k-1} \Sigma^{k-1}) \odot \Sigma$ , by analyzing its structure. Now we can apply the induction hypothesis to unfold  $\mathbb{T}F_{k-1} \Sigma^{k-1}$  into  $F_{k-1}(\mathbb{T}\Sigma)^{k-1}$ . By applying the prime functions which permute the fold with the tensor product then increase the fold, we obtain the desired term.  $\square$

### G.3.3 Term unfolding for homogeneous inputs

we say that a term  $t \in \mathbb{T}\Sigma^{[k]}$  is homogeneous if for every two internal branches  $b_1, b_2$  having twists  $\alpha_1, \alpha_2$  respectively and such that  $b_2$  is a child of  $b_1$ , we have that

$$\alpha_1 \alpha_2 = \alpha_1$$

The rest of this section is devoted to proving the following lemma.

**Lemma G.8.** *Let  $k \in \{1, 2, \dots\}$ . There is a derivable operation*

$$f : \mathbb{T}\Sigma^{[k]} \rightarrow (\mathbb{T}\Sigma)^{[k]}$$

which coincides with term unfolding for all inputs which are homogeneous.

**Lemma G.9.** *Let  $\alpha : [1, k] \rightarrow [1, k]$  and  $\beta : [1, k] \rightarrow [1, k]$  be two monotone functions such that*

$$\alpha \beta = \alpha.$$

*If the graph of  $\alpha$  is not weakly connected, then so is the graph of  $\beta$ . Moreover, if  $m \in \{1, \dots, k\}$  is such that*

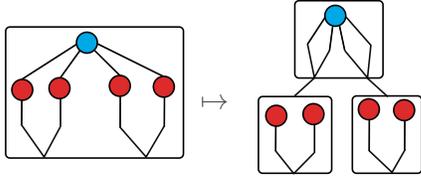
$$\alpha[1, m] \subseteq [1, m] \quad \text{and} \quad \alpha[m+1, k] \subseteq [m+1, k]$$

*then we have also*

$$\beta[1, m] \subseteq [1, m] \quad \text{and} \quad \beta[m+1, k] \subseteq [m+1, k]$$

*If the graphs of  $\alpha$  and  $\beta$  are both weakly connected, then  $\alpha$  and  $\beta$  are both constant functions.*

*Proof of Lemma G.8.* We proceed by induction on  $k$ . The base case, ie when  $k = 1$  is realized by the untwist prime function. Let us treat the inductive case. First, we factorize our term in such a way that in each factor, either all the internal twists are weakly connected, or all of them are not weakly connected. To realize this factorization, it is enough to detect the first nodes (that is the closest to the root) where the twist becomes not weakly connected. Indeed, by Lemma G.9, we know that the twists of the sub-tree rooted in such nodes are all not weakly connected. To detect these node, the following prime function is of particular interest



If we analyze this factorization, it has the form  $\mathbb{T}\Sigma^{[k]} \odot \mathbb{T}\Sigma^{[k]}$ , where the root contains only connected internal twists and the leaves only weakly connected internal twists. By Lemma G.9, we know that the internal twists of the root are constant, and that in each leaf, there is an integer  $m$  such that every internal twist  $\alpha$  satisfies

$$\alpha[1, m] \subseteq [1, m] \text{ and } \alpha[m + 1, k] \subseteq [m + 1, k].$$

To unfold the root, we apply Proposition G.6. To unfold each leaf, we proceed by induction, in the exact same way as the non-connected case in the proof of Proposition G.7. Finally to untwist the whole term, we apply the prime shallow unfold function.  $\square$

#### G.4 Proof of Theorem G.1

In this section, we complete the proof of Theorem G.1. We say that a nested factorisation in  $\mathbb{T}^n \Sigma^{[k]}$  is *monotone* if all of the labels from  $\Sigma^{[k]}$  that appear in it are monotone. Consider the homomorphism which maps a branch to its corresponding twist, and which gives the completely undefined function in case the twist is not monotone. The homomorphism uses an aperiodic monoid, as discussed in Example G.3. Apply the Factorisation Forest Theorem with respect to this homomorphism, yielding a derivable function

$$f : \mathbb{T}\Sigma^{[k]} \rightarrow \mathbb{T}^n \Sigma^{[k]}$$

which produces only nested factorisations that are hereditarily homogeneous. (Also, because monotone functions are closed under composition, it follows that if an input to  $f$  is monotone, then the same is true for the output.) Therefore, Theorem G.1 follows by composing the function  $f$  with the function  $g_n$  from the following lemma.

**Lemma G.10.** *For every finite ranked set  $\Sigma$  and  $n \in \{1, 2, \dots\}$  there is a derivable function*

$$g_n : \mathbb{T}^n \Sigma^{[k]} \rightarrow \mathbb{T}^{[k]} \Sigma$$

which makes the following diagram commute for inputs that are monotone and hereditarily homogeneous:

$$\begin{array}{ccc} \mathbb{T}^n \Sigma^{[k]} & & \\ \text{flat}^n \downarrow & \searrow g & \\ \mathbb{T}\Sigma^{[k]} & \xrightarrow{\text{unfold}} & \mathbb{T}^{[k]} \Sigma \end{array}$$

*Proof.* Induction on  $n$ . To make the induction pass through, we also show that each function  $g_n$  is consistent with the twist homomorphism in the following sense: for every input  $t \in \mathbb{T}^n \Sigma^{[k]}$ , and every port  $i \in \{1, \dots, \text{arity}(t)\}$ , the same value is

obtained by: (a) recursive flattening  $t$  and then composing all of the twists that are found on the path from the root to port  $i$ ; (b) applying  $g_n$  and then computing the twist corresponding to port  $i$ .

For the induction base  $n = 1$ , hereditarily homogeneous inputs are units, and there are finitely many of them and the function can be derived on a case by case basis.

Consider the induction step, where the lemma has already been proved for  $n$  and we want to prove it for  $n + 1$ . The function is the composition

$$\mathbb{T}^{n+1} \Sigma^{[k]} \xrightarrow{\mathbb{T}g_n} \mathbb{T}(\mathbb{T}\Sigma)^{[k]} \xrightarrow{\text{Lemma G.7}} (\mathbb{T}\mathbb{T}\Sigma)^{[k]} \xrightarrow{\text{flat}^{[k]}} \Sigma^{[k]}$$

Consider a hereditarily homogeneous input  $t \in \mathbb{T}^{n+1} \Sigma^{[k]}$ .

1. Apply the function from the induction assumption to every label of  $t$ , i.e. apply

$$\mathbb{T}^{n+1} \Sigma^{[k]} \xrightarrow{\mathbb{T}g_n} \mathbb{T}(\mathbb{T}\Sigma)^{[k]}$$

2. Let  $t_1$  be the output from the previous step. Because  $g_n$  is consistent with twists, and  $t$  is hereditarily homogeneous, it follows that  $t_1$  is either a shallow term, or it is homogeneous with respect to the twist homomorphism. If  $t_1$  is a shallow term, then we apply the shallow unfolding operation from ... Otherwise, we  $t_1$  is homogeneous, because  $t$  is hereditarily homogeneous and  $g_n$  is consistent with twists. Therefore, we can apply the function from Lemma G.7, with the alphabet being  $\mathbb{T}\Sigma$ .

3. The result of the previous step is a term  $t_2 \in (\mathbb{T}\mathbb{T}\Sigma)^{[k]}$ . To this term, we apply  $\text{flat}^{[k]}$ , yielding the final result.

A routine check shows that the function  $g_{n+1}$  defined above satisfies the property in the statement of the lemma, and that it is furthermore consistent with the twist homomorphism.  $\square$

## H Chain logic and general unfold

In this section, we prove Theorem 3.6, which says that adding general unfold to MSO yields exactly the chain logic tree-to-tree transductions. For the rest of this section, we use the word “derivable” to mean derivable in the extension of Definition 3.2 where general unfold is used instead of monotone unfold.

To prove that every derivable function is a chain logic transduction, we use the same proof as in Appendix C. The only difference is that we need to deal with general unfolding instead of monotone unfolding. For general unfolding, we use the same proof as in Section C.3.2, with the only difference being in Lemma C.5. As opposed to the monotone case in Lemma C.5, we need to compose not necessarily monotone partial functions. In the presence of non-monotone functions, the language corresponding to  $L$  from Lemma C.5 is no longer first-order definable, but it is still a regular language, and therefore it is definable in MSO. Chain logic can evaluate

arbitrary MSO properties on paths in a tree, and therefore a formula of chain logic can be used to compute the twist function between two nodes in an input tree.

The rest of this appendix is devoted to the converse implication in Theorem 3.6, which says that every chain logic tree-to-tree transduction is derivable, in the presence of general unfolding.

**Chain logic relabellings.** Define *chain logic relabellings* in the same way as the first-order relabellings from Definition 4.1, except that chain logic is used instead of first-order logic. As in Theorem 7.1 about MSO transductions, we push all of the power of chain logic into tree relabellings.

**Lemma H.1.** *Every chain logic tree-to-tree transduction can be decomposed as: (a) a chain logic relabelling; followed by (b) a first-order tree-to-tree transduction.*

*Proof sketch.* Same proof as in [14, Corollary 1], except that MSO is replaced by chain logic. The key property is that the compositionality method, which is used in Lemmas 1 and 2 of [14], also works for chain logic.  $\square$

Thanks to the above lemma, and derivability of first-order tree-to-tree transductions from our main theorem, in order to finish the proof of Theorem 3.6, it suffices to show that every chain logic relabelling is derivable. To prove derivability of chain logic relabellings, we decompose them into simpler pieces. Unlike for first-order relabellings, where the decomposition was based on Schlingloff's theorem about temporal logic, in the case of chain logic we use an approach based on top-down tree automata<sup>8</sup>.

**Top-down tree automata.** We begin by defining top-down tree automata. These are automata which process the input tree in a deterministic top-down (i.e. root-to-leaves) pass. Since we do not use nondeterministic top-down tree automata, we implicitly assume that the automata are deterministic.

**Definition H.2.** A *top-down tree automaton* is given by:

1. an *input alphabet*  $\Sigma$ , which is a finite ranked set;
2. a finite unranked set of *states*  $Q$ ;
3. a designated initial state in  $Q$ ;
4. for each input letter  $a \in \Sigma$ , a transition function

$$\delta_q : Q \rightarrow Q^{\text{arity of } a};$$

5. an *accepting set*, which is a subset of

$$Q \times (\text{input letters of arity zero}).$$

For an input tree  $t \in \text{trees}\Sigma$ , the *run* of the automaton is defined to be the labelling of the nodes by states, which is defined as follows by induction on the distance from the

<sup>8</sup>The results of this section could be translated into an apparently new result, which says that chain logic has the same expressive power as an extension of Schlingloff's logic obtained by adding group modalities as defined by Baziramwabo, McKenzie and Thérien in [5, Section 4].

root. The state in the root is the initial state. Suppose that we have already defined the state  $q$  in a node  $x$  of the input tree. Apply the transition function, corresponding to the label of node  $x$ , to the state  $q$ , yielding a tuple of states  $q_1, \dots, q_n$ . These are the states of the run in the children of node  $x$ . An input tree is accepted if for every leaf, the accepting set contains the pair (state in the leaf, label of the leaf).

**Definition H.3** (Tree relabellings associated to a top-down tree automaton). We associate two tree-to-tree functions to a top-down tree automaton  $\mathcal{A}$  with input alphabet  $\Sigma$ . Each of these is a special cases of a chain logic relabelling.

- The *ancestor relabelling*, is denoted by

$$\mathcal{A}^\uparrow : \text{trees}\Sigma \rightarrow \text{trees}(Q \times \Sigma),$$

where  $Q \times \Sigma$  is the ranked set which consists of one copy of the alphabet  $\Sigma$  for each state. The ancestor relabelling simply extends the input tree with the run of the automaton. Note that the accepting set of the automaton does not play a role in the definition of the ancestor relabelling.

- The *descendant relabelling*, denoted by

$$\mathcal{A}^\downarrow : \text{trees}\Sigma \rightarrow \text{trees}(\Sigma + \Sigma),$$

is the characteristic function, in the sense of Section 5, of the query which selects nodes whose subtree is accepted by  $\mathcal{A}$ . In other words, for each node  $x$  in the input tree, its label is replaced by the corresponding label in the first copy of  $\Sigma$  if the subtree of  $x$  is accepted by  $\mathcal{A}$ , and otherwise it is replaced by the corresponding label in the second copy of  $\Sigma$ .

The reason for notation in the above definition is that, in the ancestor relabelling, the label of a node depends on its ancestors, while in the descendant relabelling, the label of a node depends on its descendants. It is worth pointing out that many different runs of the automaton are used in the descendant relabelling, because for each node the automaton is started again with the initial state in that node.

We begin with the following lemma, which states a connection between chain logic and (nestings of) top-down tree automata that was described in [8].

**Lemma H.4.** *Every chain logic relabelling is a composition of functions which are either:*

- (a) a *letter-to-letter homomorphism*; or
- (b) the *descendant relabelling* of a top-down tree automaton.

*Proof.* Adjusting for a slightly different terminology, this lemma is the same as [8, Theorem 2.5.9]. To help with the terminology, we note that the wordsum automata (WS) from [8] are the same as top-down tree automata here, while the cascade product of wordsum automata is the same as composing descendant relabellings.  $\square$

Since letter-to-letter homomorphisms are derivable, in order to finish the proof of Theorem 3.6, it remains to prove that every function of kind (b) in the above lemma is derivable. We prove this by doing a further decomposition, which reduces the descendant relabelling to the ancestor relabelling.

**Lemma H.5.** *For every top-down tree automaton, its descendant relabelling is a composition of functions which are either:*

- (c) a first-order relabelling; or
- (d) the ancestor relabelling of a top-down tree automaton.

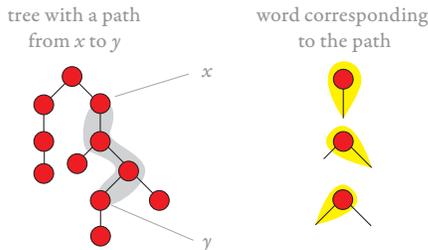
*Proof.* In this proof, we use the forward Ramseyan splits of Colcombet [14].

Fix a top-down tree automaton  $\mathcal{A}$ . For the proof of this lemma, as well as for subsequent results, it will be convenient to use a different perspective on top-down tree automata, which uses automata on words. Recall the set of branches  $B\Sigma$  that was defined in page 34: a branch is a letter together with a distinguished port. Define the *branch automaton* of  $\mathcal{A}$  to be the deterministic word automaton, where the input alphabet is  $B\Sigma$ , the states are the same, the initial state is the same as in  $\mathcal{A}$ , and the transition function is defined by

$$\left( \underbrace{q}_{Q}, \underbrace{(a, i)}_{B\Sigma} \right) \mapsto i\text{-th state in the tuple } \delta_a(q).$$

The branch automaton does not have accepting states. Roughly speaking, the run of a top-down tree automaton corresponds to running the branch automaton on every root-to-leaf path in the tree. This correspondence is spelled out in more detail below.

Consider two nodes in a tree, called the *source* and *target*, such that the source is an ancestor of the target. The source can be equal to the target. The *path* between these two nodes is defined to be the set of edges in the tree which connects them. We can view the path as a word over the alphabet  $B\Sigma$ , as illustrated in the following picture:



The correspondence between the top-down tree automaton  $\mathcal{A}$  and its branch automaton can now be phrased as follows: for a node  $x$ , the state of the top-down tree automaton in node  $x$  is the same as the state of the branch automaton after reading the (word corresponding to the) path from the root to node  $x$ .

Equipped with the above terminology, we complete the proof of the lemma. For a path in an input tree, define its

*state transformation* to be the function of type  $Q \rightarrow Q$  which describes the state transformation of the branch automaton over the (word corresponding to the) path. By Colcombet's results on forward Ramseyan splits [14, Lemma 3], there is a top-down tree automaton  $\mathcal{B}$  with input alphabet  $\Sigma$  and a family of first-order formulas

$$\{\varphi_f(x, y)\}_{f:Q \rightarrow Q}$$

with the following property. For every input tree  $t \in \text{trees}\Sigma$  and nodes  $x \leq y$  in that tree, the state transformation for the path from  $x$  to  $y$  is equal to  $f$  if and only if

$$\mathcal{B}^\uparrow(t) \models \varphi_f(x, y).$$

The idea is that the top-down tree automaton  $\mathcal{B}$  computes the forward Ramseyan split associated to state transformations in the branch automaton of  $\mathcal{A}$ . It follows that there is a formula  $\varphi(x)$  of first-order logic such that for every  $t \in \text{trees}\Sigma$ ,

$$\mathcal{B}^\uparrow(t) \models \varphi(x)$$

holds if and only if the subtree of node  $x$  is accepted by the automaton  $\mathcal{A}$ . The formula says that for all leaves  $y \leq x$ , the corresponding state transformation of the branch automaton leads to an accepting state. Therefore, the descendant relabelling of  $\mathcal{A}$  can be computed by first applying the ancestor relabelling of  $\mathcal{B}$ , and then a first-order relabelling.  $\square$

We now show that the ancestor relabellings produced by the previous lemma can be further decomposed, so that the underlying automata are reversible. Call a top-down tree automaton *reversible* if the corresponding branch automaton, as defined in the proof of Lemma H.5, is reversible, which means that for every input letter the corresponding transition function is a permutation of the states.

**Lemma H.6.** *For every top-down automaton, its ancestor function is a composition of functions which are either:*

- (c) a first-order relabelling; or
- (e) the ancestor relabelling of a reversible top-down automaton.

*Proof.* A corollary of the original Krohn-Rhodes theorem.

Define a *Mealy machine* to be a string-to-string transducer, which is obtained from a deterministic word automaton by adding an output function, which maps every transition to a letter of an output alphabet. The original Krohn-Rhodes theorem says that every Mealy machine is a composition of Mealy machines where the underlying automaton is either aperiodic or reversible.

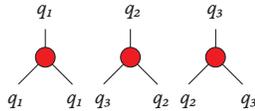
Take a top-down tree automaton. We can view its associated branch automaton as a Mealy machine which decorates each position in the input word by the state after reading the input word up to and including that position. To this Mealy machine apply the Krohn-Rhodes theorem. The relabellings for the aperiodic Mealy machines can be computed by the

functions of kind (c), while the relabellings for the reversible ones correspond to kind (e).  $\square$

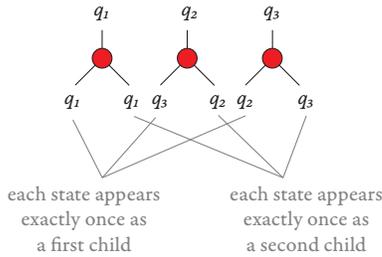
Putting together Lemmas H.4, H.5 and H.6, we see that every chain logic relabelling is a composition of functions which have kinds (c) or (e) as in the statement of Lemma H.6. Since first-order relabellings are derivable, it remains to derive the functions of kind (e).

**Lemma H.7.** *For every reversible top-down tree automaton, its ancestor relabelling is derivable (in the presence of general unfolding).*

*Proof.* Let the states of the automaton be  $Q = \{q_1, \dots, q_k\}$ . We assume that  $q_1$  is the initial state. Consider an input letter  $a$ , and its associated transition function as in item 4. Here is a picture of such a transition function, where the letter  $a$  is binary and the number of states is  $k = 3$ .



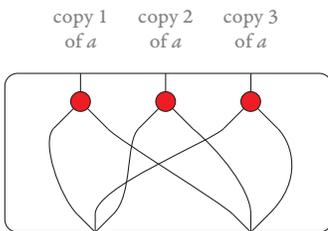
In terms of the above picture, the reversibility of the automaton can be described as follows:



We can represent the above transition function as an element of the  $k$ -th matrix power of  $Q$  copies of the states, denoted by

$$\hat{a} \in (Q \times \Sigma)^{[k]},$$

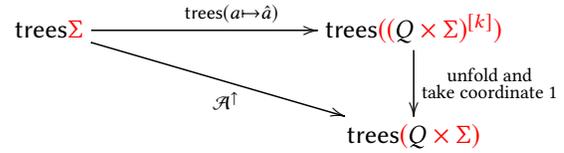
which is illustrated in the following picture:



More formally,  $\hat{a}$  is defined so that for every port  $i$  of the letter  $a$ , the  $i$ -th twist function (see Section 3.3.3) is equal to the state transformation of the branch automaton when reading the letter  $(a, i) \in B\Sigma$ . The twist functions need not be monotone, since the branch automaton need not be monotone. The reversibility of the automaton is crucial here; for a

non-reversible automaton we might need to use a sub-part of the matrix power several times.

The transformation  $a \mapsto \hat{a}$  is defined so that unfolding the matrix power captures exactly run computation in the top-down tree automaton, as described in the following commuting diagram



This completes the proof of the lemma. Note how general unfolding is used, since the twists involved need not be monotone.  $\square$