Algorithmic aspects of grammar-compressed trees

DISSERTATION
zur Erlangung des Grades eines Doktors
der Naturwissenschaften

vorgelegt von
Dipl.-Inf. Carl Philipp Reh

eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen
Siegen 2019

Betreuer und erster Gutachter
Prof. Dr. Markus Lohrey
Universität Siegen

Zweiter Gutachter
Prof. Dr. Sebastian Maneth
Universität Bremen

Tag der mündlichen Prüfung
19. September 2019

ZUSAMMENFASSUNG

We introduce forest straight-line programs (FSLPs) as a compressed representation for unranked trees (forests). These are compared to other compression formalisms for forests, namely top dags and TSLPs (tree straight-line programs) for fcns (first-child next-sibling) encoding. We show that FSLPs are equally succinct to TSLPs for fcns encodings but converting them to top dags requires a blowup of the alphabet size. All of these formalisms are treated uniformly using an algebraic setting.

We then use FSLPs to implement various data structures and algorithms: We can answer in polynomial time if two FSLPs produce equal forests modulo associativity and/or modulo commutativity. Allowing linear preprocessing time, we implement a data structure that allows navigation steps on FSLPs (go to the first/last child, the next/previous neighbor or to the parent, or print the symbol at the current location) to occur in constant time. Allowing polynomial time preprocessing, we extend this to include a subtree equality check that works in constant time.

We also study the effects of unorderedness on compression and derive various bounds. In the best case, compression using DAGs can be improved from n to $\log n$ when trees are allowed to be reordered, i.e. a ratio of $n/\log n$. For TSLPs/FSLPs instead of DAGs, we show a ratio of $n \cdot \log\log n/\log^2 n$.

We show how to evaluate a form of visibly one-counter automata on FSLPs in polynomial time by implementing them as an algebra.

Finally, we implement an algorithm that produces a top dag and is worst-case optimal, i.e. it always finds a top dag that compresses to at least $n/\log n$, while also retaining other beneficial properties from previous top dag compressors.

ZUSAMMENFASSUNG

Wir stellen Forest-Straight-Line-Programme vor, die ungerankte Bäume (Forests) komprimieren. Diese werden mit anderen komprimierten Darstellungen verglichen: Top-Dags und TSLPs (Tree-Straight-Line-Programme) für fcns (First-Child-Next-Sibling)-Kodierung. Wir zeigen, dass FSLPs und TSLPs für fcns gleich stark komprimieren. Beim Konvertieren von FSLPs zu Top-Dags ist allerdings ein Blowup der Alphabetgröße unvermeidbar. All diese Formalismen werden gleichermaßen in einem algebraischen Setting behandelt.

Wir implementieren verschiedene Algorithmen und Datenstrukturen auf FSLPs: In Polynomialzeit können wir beantworten, ob zwei FSLPs die gleichen Forests modulo Assoziativität und/oder Kommutativität produzieren. Mit linearer Preprocessing-Zeit können wir eine Datenstruktur implementieren, die Navigationsschritte auf FSLPs (gehe zum ersten/letzten Kind, zum nächsten/vorherigen Nachbarn oder zum Elternknoten, oder gib das Symbol an der aktuellen Position aus) in konstanter Zeit durchführen. Mit polynomieller Preprocessing-Zeit können wir dies um einen Subtree-Equality-Check erweitern.

Wir untersuchen außerdem, wie sich Unorderedness auf Kompression auswirkt, wobei wir verschiedene Schranken zeigen. Im besten Fall kann die Kompression eines DAGs von n zu $\log n$ verbessert werden, wenn man Bäume umordnen darf, was einem Verhältnis von $n/\log n$ entspricht. Im Falle von TSLPs/FSLPs zeigen wir ein Verhältnis von $n \cdot \log\log n/\log^2 n$.

Wir zeigen, wie man eine Art Visibly One-Counter-Automat auf FSLPs auswerten kann, indem wir diese als Algebra implementieren.

Zuletzt implementieren wir einen Algorithmus, der einen Top-Dag produziert, welcher worst-case-optimal ist, d.h. er findet immer einen Top-Dag, der mindestens zu $n/\log n$ komprimiert. Außerdem erhält dieser andere nützliche Eigenschaften von vorher eingeführten Top-Dag-Kompressoren.

CONTENTS

1

INTRODUCTION

Context-free grammars that produce a single word are called straight-line programs, SLPs for short, and can be thought of as a succinct representation of this word. Nonterminals can be used to identify repetitive content, so for example a word *abababab* can be represented as $A \to ab$, $B \to AA$, $C \to BB$, where C is the start symbol. SLPs have been widely studied: Examples are *compression algorithms* (see e.g. [16]) that take a word as an input and try to produce a small SLP for this word, and *algorithms on compressed data* (see [34] for an overview), that work on the SLP directly, instead of on the word that it produces. The best compression that can be achieved with an SLP is logarithmic, i.e. a word of length n can at best be represented by an SLP of size $\Theta(\log n)$. Furthermore, every word of length n using at most σ many different symbols can be represented by an SLP of size $\mathcal{O}(n/\log_\sigma n)$. Since an SLP can compress exponentially, answering questions about the word an SLP produces by first uncompressing the SLP is unattractive since this implies exponential runtime. Algorithms that work on the SLP directly can be more efficient. For example, it is easy to output a character at a certain position using linear time in the size of the SLP.

Apart from string compression, *tree compression* has also been widely studied. Instead of strings, trees are compressed. These trees are ordered (i.e. the children of each node are linearly ordered), in contrast to the unordered trees that we consider later. Multiple formalisms for tree compression are in use: The simplest among them is a so-called directed acyclic graph, or DAG for short. A DAG can identify the same subtrees by directing multiple edges at the same node. Tree straight-line programs (TSLPs) are another formalism which have been widely used (see [36] for a survey). They are similar to SLPs in that they are also context-free grammars but they produce a single tree instead of a single string. TSLPs can make use of *subtree repeats*. For example, a tree $a\langle a\langle \dots a \dots \rangle\rangle$, that is basically a very tall tree where each layer only consists of a single subtree, cannot be compressed using a DAG. It can however be compressed using a TSLP which can introduce a context $a(x)$. Contexts can be concatenated, so for example $a(x)a(x)$ yields the tree $a\langle a\langle x\rangle\rangle$, and they can be applied to a tree, e.g. $a(x)b$ yields the tree $a\langle b\rangle$. A small TSLP for the tree $a\langle \dots a\langle a\rangle \dots\rangle$ then basically compresses a word of the form $a(x)\dots a(x)a$. In this work, we only use linear TSLPs, which means that x may not occur in multiple places, e.g. $a(xx)$ is not allowed. A non-linear TSLP can achieve more than exponential compression, but they are also a lot more difficult to deal with when designing algorithms for them. TSLPs in general have the problem of not being able to compress horizontal repetition in a tree. For example, a tree $a\langle b \dots b\rangle$ cannot be compressed at all. This does not matter when ranked trees, i.e. trees where the number of b depends on a, are used, but in a more general setting this is a limiting factor.

A string of trees is called a *forest* which play a central role in this work. Compression of forests has been studied in various forms. For example, TSLPs can be used to compress forests (see for example the TreeRePair compressor from [37]) using the so-called *first-child next-sibling* (fcns) encoding (see e.g. [46] and Paragraph 2.3.2 of Knuth's first book [30]). In this encoding, a forest is represented as a binary tree which is basically a "head-tail" representation. TSLPs in this setting can make use of contexts to identify repeating trees on the same level, e.g. the tree $a\langle b \dots b\rangle$ can be represented succinctly. The fcns encoding however has the disadvantage of pulling a tree and its parent far apart which makes the design of algorithms on TSLPs for fcns encodings difficult. Another formalism that has been used are *top dags* ([5, 27]), which are DAGs for *top trees*. The simplest top tree is a pair (a, b), called an *atomic cluster*, that represents the tree $a\langle b\rangle$. Top trees can be combined via horizontal composition and via vertical composition, making it possible to use repeating contexts similar to TSLPs, but also horizontal repeats. For example, two top trees (a, b), (b, c) can be combined vertically, yielding a top tree for $a\langle b\langle c\rangle\rangle$. Here, the common node b is merged. Similarly, the top trees (a, b) and (a, c) can be combined horizontally, yielding a top tree for $a\langle bc\rangle$. By choosing a pair of nodes as the most primitive data structure, it is not possible to represent a tree with a single node or the empty forest using top trees. Dealing with such pairs as the most primitive data structure also makes designing algorithms on top trees inconvenient.

This work mainly deals with forests, forest compression and algorithms on compressed forests. We start by establishing a unified view of all the previously mentioned formalisms by discussing them in an algebraic setting, which is done in Chapter 2. A Σ-algebra consists of multiple parts: The syntax is given by a set Σ of operator names, where each one has a specific type. Expressions are built by using these operators, and optionally by using *variables*, which can appear anywhere a subexpression can appear. They will allow us to define our generalized view of SLPs: An SLP basically maps variables to expressions that may contain variables themselves. This gives us a purely syntactic view of compression. For example, an expression $+(+(1,1),+(1,1))$ can be compressed by introducing $+(X, X)$ and then mapping X to $+(1,1)$. We use the term SLP to refer to this general view of a compressed expression and rename SLPs to string straight-line programs (SSLPs). An algebra itself gives *semantics* to an expression. It evaluates them over a specific universe and interprets the operators by actual functions. We define a *forest algebra* as a reference point (for a forest algebra similar to this one, see [8]). Much like top dags, it allows for horizontal and for vertical composition. The most primitive building blocks are ε, the empty forest, and x, a forest context. Like with linear TSLPs, only one occurrence of x is allowed. Syntactically, we deal with this by introducing two distinct types: $\mathcal{F}$, which stands for forest expressions that do not contain x, and $\mathcal{F}_x$, which stands for forest expressions that do contain x. Similar to TSLPs, we call SLPs for forest expressions FSLPs.

Chapter 3 introduces *normal forms* of SLPs. They restrict the forms expressions may take for the purpose of making it easier to implement algorithms on them, especially syntactic transformations. Given an SLP, we want to transform it into a given normal form in *linear time*, which implies that the resulting SLP will not be larger (up to a constant factor). Our most important normal form is the *normal form for FSLPs*, which we will use in many places, for example when comparing FSLPs to top dags and the

fcns encoding, and when checking for equality up to commutativity and associativity.

Chapter 4 deals with navigating in strings given by SSLPs, ranked trees given by TSLPs and forests given by FSLPs. The goal each time is to precompute a navigation structure that requires space linear in the size of the SLP, such that the navigation operations each take constant time. In the case of strings, we can move from one position of the string to the next position, to the previous position, or obtain the character at the current position. In the case of ranked trees, we can go to the parent node or to the i'th child node, or obtain the character at the current node. In the case of forests we can go to the parent node, the first child node, the last child node, the left neighbor, the right neighbor, or obtain the character at the current node. The tree and forest structures are then expanded to allow subtree equality checks: Given two nodes that were reached by two sequences of navigation steps, are the trees at the current nodes equal? The problem of checking equality of subtrees occurs in several different contexts, see for instance [15] for details. Typical applications are common subexpression detection, unification, and non-linear pattern matching. For instance, checking whether the pattern $f\langle xf\langle yy\rangle\rangle$ is matched at a certain tree node needs a constant number of navigation steps and a single subtree equality check. To support subtree equality checks, we allow more preprocessing time and we allow to compare two numbers in the size of the tree, resp. forest, which technically takes logarithmic time in the size of the tree, resp. forest.

Chapter 5 compares FSLPs to the already mentioned top dags and to TSLPs for fcns encodings. These comparisons are done by implementing translations between the different formalisms: TSLPs for fcns encodings and FSLPs can be translated into each other in linear time, which implies that they are equally succinct. Top dags can also be translated into FSLPs in linear time. However, when translating FSLPs to top dags, a blowup-factor of the size of the alphabet is needed.

Chapter 6 introduces *associative* and *commutative* symbols. The basic idea is that FSLPs might encode expressions that have associative and/or commutative operators, like + on natural numbers, in them, i.e. the expression $1 + (2 + 3)$ should be considered equal to $(2 + 1) + 3$ because + is associative and commutative. We present a way to test if two forests produced by FSLPs are equal under these rules, which generalizes a result from [38] that only deals with TSLPs and commutative operators. This is implemented by transforming the FSLPs into the *associative normal form*, resp. *commutative normal form*. These normal forms have the property that they produce the same forests if and only if the forests produced by the original FSLPs are equal under the rules of associativity, resp. commutativity. Testing if two FSLPs produce the same forest is implemented by transforming them into SSLPs that produce strings which are the same if and only if the forests produced by the FSLPs are the same. Polynomial time equality checks on SSLPs can be found for example in [44, 29], see [35] for a survey.

Chapter 7 introduces the notion of *unordered forests*. In an unordered forest the children of a node are not ordered (in contrast to the ordered forests that have been considered so far). In particular, we consider two unordered forests to be equal if one can be transformed into the other by reordering children. For example, $a\langle bc\rangle$ is the same unordered forest as $a\langle cb\rangle$. Unordered forests play a role in "data-centric" XML (see e.g. [1, 7, 9, 47, 48]). We look at how much we can improve compression if we allow to compress, instead of an unordered forest f, *any* other unordered forest f' that is equal

to f. This is first studied for DAGs, where we look at two different size measures: First the number of edges of a DAG and second the number of nodes. For the number of edges, there are trees of size n that basically have a DAG with $\Theta(n)$ edges, but they can be reordered such that their DAG only has $\Theta(\log n)$ edges. For the number of nodes, this gap slightly increases since trees again need $\Theta(n)$ nodes for the DAG, but they can be reordered such that their DAG only needs $\Theta(\log n / \log\log n)$ nodes. In addition to DAGs, we study the same question for FSLPs and TSLPs: There are forests of size n that can be reordered such that they are compressible using FSLPs of size $\Theta(\log n)$. However, the FSLP for the original forests need at least size $\Theta(n \cdot \log\log n / \log n)$. We show a similar result for TSLPs.

Chapter 8 shows how to evaluate forest automata (see e.g. [17]) on FSLPs by combining some results with our transformation from FSLPs to TSLPs for their fcns encodings. We also show how to evaluate a form of visibly one-counter automata on FSLPs (see e.g. [31, 2]).

Chapter 9 presents an algorithm that extends the results of [5]: They implement an algorithm that given a tree t of size n produces a top dag of height $\mathcal{O}(\log n)$ and a size that is at most the size of the minimal DAG of t times $\log n$. We extend this algorithm by introducing a preprocessing step that ensures that the resulting top dag also has size $\mathcal{O}(n / \log_\sigma n)$, where σ is the alphabet size.

2

ALGEBRAS AND SLPS

Algebras are an abstract way to define expressions and evaluate them. For an introduction, see for example [41]. The first step in defining an algebra is to specify its syntax. This is done by introducing a set (which might be infinite) of typed operations, e.g. given types N and 2, we could have the operations $*\colon N^2 \to N$, $\mathbf{cond}\colon 2 \times N^2 \to N$, $\mathbf{true}\colon 2$, $\mathbf{false}\colon 2$, $\mathbf{0}\colon N$ and $\mathbf{1}\colon N$. Terms of a given type over a set of operations are defined by induction, e.g. $\mathbf{1}$ is a term of type N, $*(\mathbf{1},\mathbf{1})$ is a term of type N, and so on. The next step in defining an algebra is to define how it evaluates expressions. For this, we need to specify a carrier set for every type, e.g. $\mathbb{N}$ for N and $\{t,f\}$ for 2. We then have to interpret every operation with an actual (n-ary) function, e.g. $*$ is interpreted with multiplication on natural numbers. To define algebras formally, we start by introducing typed sets and function types.

Definition 1 (Typed set)**.** Let **type** be a set. A *typed set* over **type** is a pair (A,τ), where $\tau\colon A \to \mathbf{type}$. For every $t \in \mathbf{type}$ we set $A_t = \{a \in A \mid \tau(a) = t\}$.

In the above definition, **type** serves as a set of types, e.g. $\mathbf{type} = \{\mathbf{Bool},\mathbf{Int}\}$ and the function τ maps each element of A to its type. For example, we could have $A = \mathbb{N} \cup \{t,f\}$ with $\tau(t) = \tau(f) = \mathbf{Bool}$ and $\tau(i) = \mathbf{Int}$ for $i \in \mathbb{N}$.

Definition 2 (Function types)**.** Let **type** be a set. A (first-order) *function type over* ***type*** is of the form $t_1 \times \cdots \times t_n \to t$, where $t_1,\ldots,t_n,t \in \mathbf{type}$ and $n \in \mathbb{N}$. In case $n = 0$ we simply write t instead of $\to t$. We denote with $\mathbf{type}^{\to}$ the set of all function types over **type**.

As an example, **Int**, **Bool** and $\mathbf{Bool} \times \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$ are function types. Note that we allow there to be zero arguments to a function.

Definition 3 (Signatures)**.** Let Σ and **type** be sets. A *signature* over Σ and **type** is a function $\mathcal{S}\colon \Sigma \to \mathbf{type}^{\to}$. Instead of specifying Σ and $\mathcal{S}$ explicitly, they can both be given by a list: $a_1\colon t_1,\ldots,a_n\colon t_n$ ($n \geq 0$) with $a_i \neq a_j$ for all $1 \leq i,j \leq n$ with $i \neq j$ and $t_i \in \mathbf{type}$ for $1 \leq i \leq n$. We obtain $\Sigma = \{a_1,\ldots,a_n\}$ and $\mathcal{S}(a_i) = t_i$ for $1 \leq i \leq n$.

We can think of Σ as a set of operators, e.g. $\Sigma = \{*,\mathbf{cond},\mathbf{0},\mathbf{1},\mathbf{true},\mathbf{false}\}$. The signature $\mathcal{S}$ maps each operator to its type, e.g. if $\mathbf{type} = \{\mathbf{Bool},\mathbf{Int}\}$, we could have $\mathcal{S}(\mathbf{cond}) = \mathbf{Bool} \times \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$, $\mathcal{S}(\mathbf{true}) = S(\mathbf{false}) = \mathbf{Bool}$, $\mathcal{S}(*) = \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$, and $\mathcal{S}(\mathbf{0}) = S(\mathbf{1}) = \mathbf{Int}$. Instead of defining $\mathcal{S}$ explicitly as a function, it could also be given by the following list: $\mathbf{cond}\colon \mathbf{Bool} \times \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$, $\mathbf{true}\colon \mathbf{Bool}$, $\mathbf{false}\colon \mathbf{Bool}$, $*\colon \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$, $0\colon \mathbf{Int}$, $1\colon \mathbf{Int}$.

Definition 4 (Variables and Typing)**.** Let **type** be a set, let $\mathcal{V}$ be a (countably) infinite set of variables and let $V \subseteq \mathcal{V}$. A *typing* of V is a function $\Gamma\colon V \to \mathbf{type}$. Again, we may give Γ as a list of the form $X_1\colon \tau_1,\ldots,X_n\colon \tau_n$, where $n \geq 0$,

$X_i \in \mathcal{V}$ and $\tau_i \in \mathbf{type}$ for $1 \le i \le n$. This then defines $V = \{V_1, \ldots, V_n\} \subseteq \mathcal{V}$, $\Gamma : V \to \mathbf{type}$ and $\Gamma(V_i) = \tau_i$ for $1 \le i \le n$.

We assume that all variables are elements of $\mathcal{V}$ and also that $\mathcal{V}$ does not interfere with any signatures that we use.

Definition 5 (Expressions)**.** Let $\mathcal{S} : \Sigma \to \mathbf{type}^{\to}$ and $\Gamma : V \to \mathbf{type}$, where $V \subseteq \mathcal{V}$. For every $t \in \mathbf{type}$, the set $\mathcal{E}(\mathcal{S}, \Gamma, t)$ of *expressions of type* t is defined as follows:

- $x \in \mathcal{E}(\mathcal{S}, \Gamma, t)$ for every $x \in V$ with $\Gamma(x) = t$,
- $f(e_1, \ldots, e_n) \in \mathcal{E}(S, \Gamma, t)$ for every $e_1 \in \mathcal{E}(\mathcal{S}, \Gamma, t_1), \ldots, e_n \in \mathcal{E}(\mathcal{S}, \Gamma, t_n)$, $n \in \mathbb{N}$, $f \in \Sigma$, and $\mathcal{S}(f) = t_1 \times \cdots \times t_n \to t$.

The *set of all expressions* is defined as $\mathcal{E}(\mathcal{S}, \Gamma) = \bigcup \{\mathcal{E}(\mathcal{S}, \Gamma, t) \mid t \in \mathbf{type}\}$. In case $V = \varnothing$ (so $\Gamma : \varnothing \to \mathbf{type}$) we do not mention Γ and write $\mathcal{E}(\mathcal{S})$ for $\mathcal{E}(\mathcal{S}, \Gamma)$, and $\mathcal{E}(\mathcal{S}, t)$ for $\mathcal{E}(\mathcal{S}, \Gamma, t)$ where $t \in \mathbf{type}$. For $e \in \mathcal{E}(\mathcal{S}, \Gamma)$ we define $\tau_{\mathcal{E}} : \mathcal{E}(\mathcal{S}, \Gamma) \to \mathbf{type}$, which maps an expression to its type, as $\tau_{\mathcal{E}}(e) = t$ where $e \in \mathcal{E}(\mathcal{S}, \Gamma, t)$. (Note that $\mathcal{E}(\mathcal{S}, \Gamma)$ together with $\tau_{\mathcal{E}}$ is a typed set.) If $\mathcal{S}$ and Γ are known from the context, we will simply write $e : t$ instead of $e \in \mathcal{E}(\mathcal{S}, \Gamma, t)$.

The *size* $|e|$ of an expression $e \in \mathcal{E}(\mathcal{S}, \Gamma)$ is defined as: $|v| = 1$ for $v \in V$ and $|f(e_1, \ldots, e_n)| = 1 + \sum \{|e_i| \mid 1 \le i \le n\}$ for $f(e_1, \ldots, e_n) \in \mathcal{E}(\mathcal{S}, \Gamma) \setminus V$.

Continuing our previous example, we would have

$$\mathbf{cond}(\mathbf{false}, *(0, *(0, 1), *(1, 1))) \in \mathcal{E}(\mathcal{S}, \mathbf{Int}).$$

Let Γ be given by $x : \mathbf{Bool}$ and $y : \mathbf{Int}$, so we have two variables. Then $\mathbf{cond}(x, y, 1) \in \mathcal{E}(\mathcal{S}, \Gamma, \mathbf{Int})$.

Definition 6 (Algebra)**.** Given a signature $\mathcal{S} : \Sigma \to \mathbf{type}^{\to}$, an *algebra* is a pair $\mathcal{A} = ((\mathcal{U}, \tau), \mathcal{I})$, where $\mathcal{U}$ together with $\tau : \mathcal{U} \to \mathbf{type}$ is a typed set and $\mathcal{I}$ is a function such that $\mathcal{I}(f) : \tau(t_1) \times \cdots \times \tau(t_n) \to \tau(t)$ for every $f \in \Sigma$ with $S(f) = t_1 \times \cdots \times t_n \to t$.

The standard algebra for our example would be $\mathcal{A} = ((\mathcal{U}, \tau), \mathcal{I})$ with $\mathcal{U} = \mathbb{N} \cup \{t, f\}$, $\tau(n) = \mathbf{Int}$ for $n \in \mathbb{N}$ and $\tau(t) = \tau(f) = \mathbf{Bool}$, where the mapping $\mathcal{I}$ is defined as follows:

- $\mathcal{I}(\mathbf{0}) = 0$, $\mathcal{I}(\mathbf{1}) = 1$, $\mathcal{I}(\mathbf{true}) = t$, $\mathcal{I}(\mathbf{false}) = f$,
- $\mathcal{I}(*)(x, y) = x * y$, where $* : \mathbb{N}^2 \to \mathbb{N}$ is multiplication on $\mathbb{N}$, and
- $\mathcal{I}(\mathbf{cond})(b, x, y) = \begin{cases} x & \text{if } b = t, \\ y & \text{if } b = f. \end{cases}$

Another algebra would be $\mathcal{A}' = ((\mathcal{U}', \tau'), \mathcal{I}')$ that counts the number of occurrences of **true** and **false**. We define $\mathcal{U}' = \{1'\} \cup \mathbb{N}$ with $\tau'(1') = \mathbf{Bool}$ and $\tau'(n) = \mathbf{Int}$ for all $n \in \mathbb{N}$. Note that we have to map **true** and **false** to an element that gets type **Bool**. The idea is that, since each occurrence of **true** and **false** counts as 1, we map them to a copy of 1. Let $i : \{1'\} \to \mathbb{N}$ be $i(1') = 1$. We define $\mathcal{I}$ as follows:

- $\mathcal{I}'(\mathbf{0}) = 0$, $\mathcal{I}'(\mathbf{1}) = 0$, $\mathcal{I}'(\mathbf{true}) = 1'$, $\mathcal{I}'(\mathbf{false}) = 1'$,
- $\mathcal{I}'(*)(x, y) = x + y$, where $+ : \mathbb{N}^2 \to \mathbb{N}$ is addition on $\mathbb{N}$, and
- $\mathcal{I}'(\mathbf{cond})(b, x, y) = i(b) + x + y$.

Definition 7 (Environment and Evaluation)**.** Let $\mathcal{A} = ((\mathcal{U}, \tau), \mathcal{I})$ be an algebra with signature $\mathcal{S}: \Sigma \to \mathbf{type}^{\to}$, $\tau: \mathcal{U} \to \mathbf{type}$, $V \subseteq \mathcal{V}$ and typing $\Gamma: V \to \mathbf{type}$. An *environment* is a function $\eta: V \to \mathcal{U}$ such that $\tau \circ \eta = \Gamma$. The evaluation function $[\![]\!]_{\mathcal{A},\eta}: \mathcal{E}(\mathcal{S}, \Gamma) \to \mathcal{U}$ is defined as follows:

- For $x \in V$ we set $[\![x]\!]_{\mathcal{A},\eta} = \eta(x)$.
- For $f(e_1, \ldots, e_n) \in \mathcal{E}(\mathcal{S}, \Gamma) \setminus V$ we set

$$[\![f(e_1, \ldots, e_n)]\!]_{\mathcal{A},\eta} = \mathcal{I}(f)([\![e_1]\!]_{A,\eta}, \ldots, [\![e_n]\!]_{A,\eta}).$$

For expressions without variables we define $[\![]\!]_{\mathcal{A}}: \mathcal{E}(\mathcal{S}) \to \mathcal{U}$ as $[\![e]\!]_{\mathcal{A}} = [\![e]\!]_{\mathcal{A},\eta}$ with $\eta: \varnothing \to \mathcal{U}$.

Going back to our previous examples, we have for example

- $[\![*(\mathbf{1}, \mathbf{1})]\!]_{\mathcal{A}} = [\![\mathbf{1}]\!]_{\mathcal{A}} * [\![\mathbf{1}]\!]_{\mathcal{A}} = 1 * 1 = 1$,
- $[\![*(\mathbf{1}, \mathbf{1})]\!]_{\mathcal{A}'} = [\![\mathbf{1}]\!]_{\mathcal{A}'} + [\![\mathbf{1}]\!]_{\mathcal{A}'} = 0 + 0 = 0$,
- $[\![\mathbf{cond}(\mathbf{false}, \mathbf{cond}(\mathbf{true}, \mathbf{0}, \mathbf{1}), \mathbf{1})]\!]_{\mathcal{A}} = 1$ and
- $[\![\mathbf{cond}(\mathbf{false}, \mathbf{cond}(\mathbf{true}, \mathbf{0}, \mathbf{1})\mathbf{1})]\!]_{\mathcal{A}'} = 2$.

With $\eta: \{x\} \to \mathbb{N}$ and $\eta(x) = 5$, we have $[\![+(x, \mathbf{1})]\!]_{\mathcal{A},\eta} = \eta(x) + [\![\mathbf{1}]\!]_{A} = 5 + 1 = 6$.

Definition 8 (Acyclic Relation)**.** Let $R \subseteq A \times A$ be a relation. R is called *acyclic* if its transitive closure $R^+ = \bigcup\{R^i \mid i \geq 1\}$ of R is a strict (partial) order. Since R^+ is transitive by definition, this means that R^+ must be asymmetric, i.e. there are no $v, v' \in V$ such that $(v, v') \in R^+$ and $(v', v) \in R^+$.

Definition 9 (SLPs)**.** Let $\mathcal{S}: \Sigma \to \mathbf{type}^{\to}$. An *SLP* with signature $\mathcal{S}$ is a tuple $G = (V, \Gamma, \rho)$ with $V \subseteq \mathcal{V}$, $\Gamma: V \to \mathbf{type}$ and $\rho: V \to \mathcal{E}(\mathcal{S}, \Gamma)$. Let $\leq_G \subseteq V \times V$ with $\leq_G = \{(v', v) \mid v' \text{ occurs in } \rho(v)\}$. An SLP must fulfil the following:

- $\tau_{\mathcal{E}} \circ \rho = \Gamma$.
- $\leq_G$ is acyclic.

The *size* of G is defined as $|G| = \sum\{|\rho(X)| \mid X \in V\}$. The *height* $h(e)$ of an expression $e \in \mathcal{E}(\mathcal{S}, \Gamma)$ is defined as $h(A) = h(\rho(A))$ for $A \in V$ and $h(f(e_1, \ldots, e_n)) = 1 + \max\{h(e_1), \ldots, h(e_n)\}$. The *height* of G is defined as $h(G) = \max\{h(A) \mid A \in V\}$.

Definition 10 (Evaluating SLPs)**.** Let $G = (V, \Gamma, \rho)$ be an SLP with signature $\mathcal{S}: \Sigma \to \mathbf{type}^{\to}$ and let $\mathcal{A}((\mathcal{U}, \tau), \mathcal{I})$ with $\tau: \mathcal{U} \to \mathbf{type}$ be an algebra. The function $[\![]\!]_{G,\mathcal{A}}: \mathcal{E}(\mathcal{S}, \Gamma) \to \mathcal{U}$ evaluates an expression and is defined as follows: Let $\eta: V \to \mathcal{U}$ be defined as $\eta = [\![]\!]_{G,\mathcal{A}} \circ \rho$ and define $[\![]\!]_{G,\mathcal{A}} = [\![]\!]_{\mathcal{A},\eta}$. In case $\mathcal{A}$ is clear from the context, we simply write $[\![]\!]_G$.

Note that this recursive definition of $[\![]\!]_{G,\mathcal{A}}$ is well-defined since $\leq_G$ is acyclic. The environment η evaluates a variable to an element from $\mathcal{U}$ by first using ρ to get an expression from $\mathcal{E}(\mathcal{S}, \Gamma)$ and then using $[\![]\!]_{G,\mathcal{A}}$ recursively. The definition of $[\![]\!]_{G,\mathcal{A}}$ depends on both an SLP G and an algebra $\mathcal{A}$. We chose to define it this way since we almost always have a concrete algebra in mind. Alternatively, there is a different, equivalent view of evaluation: Since an SLP is defined independently from algebras, it can *unfold* expressions by recursively replacing variables, i.e. an SLP can turn any expression $e \in \mathcal{E}(\mathcal{S}, \Gamma)$ into an expression $e' \in \mathcal{E}(\mathcal{S})$ without variables by recursively applying ρ. This final expression e' can then be evaluated in any algebra $\mathcal{A} = ((\mathcal{U}, \tau), \mathcal{I})$ using $\eta: \varnothing \to \mathcal{U}$.

Definition 11 (Unfolding SLPs)**.** Let $G = (V, \Gamma, \rho)$ be an SLP with signature $\mathcal{S}$. We define $\mathrm{unfold}_G \colon \mathcal{E}(\mathcal{S}, \Gamma) \to \mathcal{E}(\mathcal{S})$ as

$$\mathrm{unfold}_G(f(e_1, \ldots, e_n)) = f(\mathrm{unfold}_G(e_1), \ldots, \mathrm{unfold}_G(e_n))$$

for an expression $f(e_1, \ldots, e_n) \in \mathcal{E}(\mathcal{S}, \Gamma)$ and $\mathrm{unfold}_G(X) = \mathrm{unfold}_G(\rho(X))$ for a variable $X \in V$.

We are often interested in the value of a particular variable of an SLP, which we will call the start variable.

Definition 12 (SLP with start variable)**.** An *SLP with start variable* S is a tuple $G = (V, \Gamma, \rho, S)$, where (V, Γ, ρ) is an SLP and $S \in V$. Given an algebra $\mathcal{A}$ we write $[\![G]\!]_{\mathcal{A}} = [\![S]\!]_{G,\mathcal{A}}$. In case $\mathcal{A}$ is clear from the context, we simply write $[\![G]\!]$. We also set $\mathrm{unfold}(G) = \mathrm{unfold}_G(S)$.

2.1 Algebras for trees and forests

We now come to the first formalism that represents terms for trees. These trees, called ranked trees, have the property that the label of a node determines exactly how many children this node must have. For example, we could say that a node labelled with + must have two children and a node labelled with 1 must have none.

Definition 13 (Ranked alphabet)**.** A *ranked alphabet* is a pair (Σ, r), where Σ is a set and $r \colon \Sigma \to \mathbb{N}$.

Definition 14 (Ranked Trees)**.** Let (Σ, r) be a ranked alphabet. The set $\mathcal{T}(\Sigma, r)$ of *ranked trees over* (Σ, r) is defined by induction: If $t_1, \ldots, t_{r(a)} \in \mathcal{T}(\Sigma, r)$ and $a \in \Sigma$ then $a\langle t_1, \ldots, t_{r(a)}\rangle \in \mathcal{T}(\Sigma, r)$. The set $\mathcal{T}_x(\Sigma, r)$ of *ranked trees over* (Σ, r) *with a parameter* is also defined by induction: $x \in \mathcal{T}_x(\Sigma, r)$, and if $t_1, \ldots, t_{r(a)-1} \in \mathcal{T}(\Sigma, r)$, $1 \le i \le r(a)$, $t \in \mathcal{T}_x(\Sigma, r)$ and $a \in \Sigma$ then

$$a\langle t_1, \ldots, t_{i-1}, t, t_i, \ldots, t_{r(a)-1}\rangle \in \mathcal{T}_x(\Sigma, r).$$

Note that in order to produce leaves, we need an $a \in \Sigma$ with $r(a) = 0$. Also note that in the last part of the definition $r(a) = 0$ is not allowed since that would lead to a ranked tree without a parameter.

To substitute a ranked tree (with or without a parameter) into a ranked tree with a parameter, we define the substitution function on ranked trees:

Definition 15 (Ranked tree substitution)**.** Let (Σ, r) be a ranked alphabet. We define the function

$$[\,] \colon \mathcal{T}_x(\Sigma, r) \times (\mathcal{T}(\Sigma, r) \cup \mathcal{T}_x(\Sigma, r)) \to (\mathcal{T}(\Sigma, r) \cup \mathcal{T}_x(\Sigma, r)),$$

where $x[t'] = t'$ for $t' \in \mathcal{T}_x(\Sigma, r)$, and

$$a\langle t_1, \ldots, t_{i-1}, t, t_i, \ldots, t_{r(a)-1}\rangle[t'] = a\langle t_1, \ldots, t_{i-1}, t[t'], t_i, \ldots, t_{r(a)-1}\rangle$$

for $t \in \mathcal{T}_x(\Sigma, r)$, $a \in \Sigma$ and $t_1, \ldots, t_{r(a)-1} \in \mathcal{T}(\Sigma, r)$.

We will use the following signature to represent ranked trees as expressions.

Definition 16 (Tree expressions)**.** Let $\mathbf{type}_{\mathcal{T}} = \{\mathcal{T}, \mathcal{T}_x\}$ and let (Σ, r) be a ranked alphabet. The *tree signature* $\mathcal{S}_{\mathcal{T}}(\Sigma, r)$ over Σ and r is defined by the following operations:

- $a\colon \mathcal{T}$ for every $a \in \Sigma$ with $r(a) = 0$,
- $a_i\colon \mathcal{T}^{r(a)-1} \to \mathcal{T}_x$ for every $a \in \Sigma$ and $1 \leq i \leq r(a)$,
- $\boxplus\colon \mathcal{T}_x^2 \to \mathcal{T}_x$ and
- $\circledcirc\colon \mathcal{T}_x \times \mathcal{T} \to \mathcal{T}$.

Instead of $a_i(e_1, \ldots, e_{r(a)-1})$ we may also write $a(e_1, \ldots, e_{i-1}, x, e_i, \ldots, e_{r(a)-1})$ where $e_1, \ldots, e_{r(a)-1} \in \mathcal{E}(\mathcal{S}_{\mathcal{T}}(\Sigma, r))$ and $a \in \Sigma$. In addition, we may write $a(e_1, \ldots, e_{r(a)})$ instead of $a_1(e_2, \ldots, e_{r(a)}) \circledcirc e_1$, where $e_1, \ldots, e_{r(a)} \in \mathcal{E}(\mathcal{S}_{\mathcal{T}}(\Sigma, r))$ and $a \in \Sigma$. An SLP for tree expressions (over Σ) is called a TSLP (over Σ). For a TSLP (V, Γ, ρ) we set $V_0 = \Gamma^{-1}(\mathcal{T})$ and $V_1 = \Gamma^{-1}(\mathcal{T}_x)$. For a TSLP (V, Γ, ρ, S) with start variable S we require that $S \in V_0$.

Definition 17 (Standard tree algebra)**.** Let (Σ, r) be a ranked alphabet and let $\tau_{\mathcal{T}}\colon \mathcal{T}(\Sigma, r) \cup \mathcal{T}_x(\Sigma, r) \to \{\mathcal{T}, \mathcal{T}_x\}$ with

$$\tau_{\mathcal{T}}(t) = \begin{cases} \mathcal{T} & \text{if } t \in \mathcal{T}(\Sigma), \\ \mathcal{T}_x & \text{if } t \in \mathcal{T}_x(\Sigma). \end{cases}$$

The *standard tree algebra* over (Σ, r) is

$$\mathcal{A}_{\mathcal{T},\Sigma,r} = ((\mathcal{T}(\Sigma, r) \cup \mathcal{T}_x(\Sigma, r), \tau_{\mathcal{T}}), \mathcal{I}_{\mathcal{T}}),$$

where $\mathcal{I}_{\mathcal{T}}$ is defined as follows:

- $\mathcal{I}_{\mathcal{T}}(a) = a$, for $a \in \Sigma$, $r(a) = 0$,
- $\mathcal{I}_{\mathcal{T}}(a_i)(t_1, \ldots, t_{r(a)-1}) = a\langle t_1, \ldots, t_{i-1}, x, t_i, \ldots, t_{r(a)-1}\rangle$, for $a \in \Sigma$, $r(a) \geq 1$,
- $\mathcal{I}_{\mathcal{T}}(\boxplus)(t_1, t_2) = t_1[t_2]$, and
- $\mathcal{I}_{\mathcal{T}}(\circledcirc)(t_1, t_2) = t_1[t_2]$.

Instead of $\mathcal{A}_{\mathcal{T},\Sigma,r}$ we will often simply write $\mathcal{A}_{\mathcal{T}}$ if (Σ, r) is clear from the context. Also, we will often write $[\![e]\!]$ instead of $[\![e]\!]_{\mathcal{A}_{\mathcal{T},\Sigma,r}}$.

When we use $\mathcal{A}_{\mathcal{T}}$ we are also allowed to omit some parentheses in expressions. For example, since $\boxplus$ is associative in $\mathcal{A}_{\mathcal{T}}$, we simply write $e_1 \boxplus e_2 \boxplus e_3$ instead of $(e_1 \boxplus e_2) \boxplus e_3$.

We now come to a more general notion of trees. Here, the number of children of a node is not determined by its label. We also define forests as a list of trees.

Definition 18 (Forests and Trees)**.** Let Σ be an alphabet. The set $\mathcal{F}(\Sigma)$ of forests over Σ and the set $\mathcal{T}(\Sigma)$ of trees over Σ are defined by induction:

- $\varepsilon \in \mathcal{F}(\Sigma)$.
- If $f_1, f_2 \in \mathcal{F}(\Sigma)$ then $f_1 f_2 \in \mathcal{F}(\Sigma)$.
- If $f \in \mathcal{F}(\Sigma)$ then $a\langle f\rangle \in \mathcal{F}(\Sigma)$ and $a\langle f\rangle \in \mathcal{T}(\Sigma)$ for every $a \in \Sigma$.

As a short-hand notation, we may write a instead of $a\langle\varepsilon\rangle$, where $a \in \Sigma$.

The set $\mathcal{F}_x(\Sigma)$ of forests over Σ with a parameter and the set $\mathcal{T}_x(\Sigma)$ of trees over Σ with a parameter are also defined by induction:

- $x \in \mathcal{F}_x(\Sigma)$.
- If $f_1 \in \mathcal{F}(\Sigma)$ and $f_2 \in \mathcal{F}_x(\Sigma)$ then $f_1 f_2 \in \mathcal{F}_x(\Sigma)$ and $f_2 f_1 \in \mathcal{F}_x(\Sigma)$.

- If $f \in \mathcal{F}_x(\Sigma)$ then $a\langle f\rangle \in \mathcal{F}_x(\Sigma)$ and $a\langle f\rangle \in \mathcal{T}_x(\Sigma)$ for every $a \in \Sigma$.

The *rank* $\rho\colon \mathcal{T}(\Sigma) \to \mathbb{N}$ of a tree is defined as

$$\rho(a\langle t_1 \dots t_n\rangle) = \max\{n, \rho(t_1), \dots, \rho(t_n)\},$$

where $a \in \Sigma$ and $t_1, \dots, t_n \in \mathcal{T}(\Sigma)$. Let $r \in \mathbb{N}$. The set of all Σ-labelled trees $t \in \mathcal{T}(\Sigma)$ with $\rho(t) \leq r$ is denoted with $\mathcal{T}_r(\Sigma)$. The *height* $h\colon \mathcal{T}(\Sigma) \to \mathbb{N}$ of a tree is defined as

$$h(a\langle t_1 \dots t_n\rangle) = 1 + \max\{h(t_1), \dots, h(t_n)\}.$$

Definition 19 (Forest and tree substitution)**.** The substitution function on forests $[]\colon \mathcal{F}_x(\Sigma) \times (\mathcal{F}(\Sigma) \cup \mathcal{F}_x(\Sigma)) \to (\mathcal{F}(\Sigma) \cup \mathcal{F}_x(\Sigma))$ is defined as follows: Let $f, f_1 \in \mathcal{F}_x(\Sigma)$ and $f_2 \in \mathcal{F}(\Sigma)$. We set

$$\begin{aligned} x[f'] &= f', \\ (f_1 f_2)[f'] &= (f_1[f'])f_2, \\ (f_2 f_1)[f'] &= f_2(f_1[f']), \\ a\langle f\rangle[f'] &= a\langle f[f']\rangle. \end{aligned}$$

Substitution on trees, $[]\colon \mathcal{T}_x(\Sigma) \times (\mathcal{T}(\Sigma) \cup \mathcal{T}_x(\Sigma)) \to (\mathcal{T}(\Sigma) \cup \mathcal{T}_x(\Sigma))$, is defined using substitution on forests (which is the last case of the previous definition).

We are going to use the following signature to represent forests as expressions.

Definition 20 (Forest expressions)**.** Let $\mathbf{type}_{\mathcal{F}} = \{\mathcal{F}, \mathcal{F}_x\}$. The *forest signature* $S_{\mathcal{F}}(\Sigma)$ over Σ is defined by the following operations:

- $\varepsilon\colon \mathcal{F}$,
- $x\colon \mathcal{F}_x$,
- $a(x)\colon \mathcal{F}_x$ for every $a \in \Sigma$,
- $\boxminus\colon \mathcal{F} \times \mathcal{F} \to \mathcal{F}$,
- $\circledcirc\colon \mathcal{F}_x \times \mathcal{F} \to \mathcal{F}_x$,
- $\circledcirc\colon \mathcal{F} \times \mathcal{F}_x \to \mathcal{F}_x$,
- $\circledcirc\colon \mathcal{F}_x \times \mathcal{F} \to \mathcal{F}$, and
- $\boxminus\colon \mathcal{F}_x \times \mathcal{F}_x \to \mathcal{F}_x$.

An SLP for forest expressions (over Σ) is called an FSLP (over Σ). For an FSLP (V, Γ, ρ) we set $V_0 = \Gamma^{-1}(\mathcal{F})$ and $V_1 = \Gamma^{-1}(\mathcal{F}_x)$. For an FSLP (V, Γ, ρ, S) with start variable S we require that $S \in V_0$. Instead of $\mathcal{E}(\mathcal{S}_{\mathcal{F}}(\Sigma))$ we may write $\mathcal{E}_{\mathcal{F}}(\Sigma)$.

Definition 21 (Standard forest algebra)**.** Let $\tau_{\mathcal{F}}\colon \mathcal{F}(\Sigma) \cup \mathcal{F}_x(\Sigma) \to \{\mathcal{F}, \mathcal{F}_x\}$ with

$$\tau_{\mathcal{F}}(f) = \begin{cases} \mathcal{F} & \text{if } f \in \mathcal{F}(\Sigma), \\ \mathcal{F}_x & \text{if } f \in \mathcal{F}_x(\Sigma). \end{cases}$$

The *standard forest algebra* over Σ is

$$\mathcal{A}_{\mathcal{F},\Sigma} = ((\mathcal{F}(\Sigma) \cup \mathcal{F}_x(\Sigma), \tau_{\mathcal{F}}), \mathcal{I}_{\mathcal{F}}),$$

where $\mathcal{I}_{\mathcal{F}}$ is defined as follows:

- $\mathcal{I}_{\mathcal{F}}(\varepsilon) = \varepsilon$,
- $\mathcal{I}_{\mathcal{F}}(x) = x$,
- $\mathcal{I}_{\mathcal{F}}(a(x)) = a\langle x\rangle$,
- $\mathcal{I}_{\mathcal{F}}(\boxminus)(f, f') = ff'$,
- $\mathcal{I}_{\mathcal{F}}(\oslash)(f, f') = ff'$,
- $\mathcal{I}_{\mathcal{F}}(\oslash)(f, f') = ff'$,
- $\mathcal{I}_{\mathcal{F}}(\oslash)(f, f') = f[f']$,
- $\mathcal{I}_{\mathcal{F}}(\boxplus)(f, f') = f[f']$.

As with $\mathcal{A}_{\mathcal{T}}$, we may write $\mathcal{A}_{\mathcal{F}}$ instead of $\mathcal{A}_{\mathcal{F},\Sigma}$ if Σ is clear from the context. Also, instead of $[\![e]\!]_{A_{\mathcal{F},\Sigma}}$ we will often simply write $[\![e]\!]$.

Similar to $\mathcal{A}_{\mathcal{T}}$, when we use $\mathcal{A}_{\mathcal{F}}$ we are also allowed to omit some parentheses in expressions: For example, since $\boxplus$ and $\circ$ are associative in $\mathcal{A}_{\mathcal{F}}$, we simply write $e_1 \boxplus e_2 \boxplus e_3$ instead of $(e_1 \boxplus e_2) \boxplus e_3$, and $e_1 \boxminus e_2 \boxminus e_3$ instead of $(e_1 \boxminus e_2) \boxminus e_3$. We also write $e_1 \oslash e_2 \oslash e_3$ instead of $(e_1 \oslash e_2) \oslash e_3$, and so on.

Example 1. Let $n \in \mathbb{N}$. Consider the FSLP

$$F = (\{S, A_0, \ldots, A_n, B_0, \ldots, B_n\}, \Gamma, \rho, S)$$

over $\{a, b, c\}$ with $\Gamma(A_i) = \mathcal{F}$ and $\Gamma(B_i) = \mathcal{F}_x$ for $0 \le i \le n$, and where ρ is defined by

$$\begin{aligned}
\rho(A_0) &= a,\\
\rho(A_i) &= A_{i-1} \boxminus A_{i-1} \text{ for } 1 \le i \le n,\\
\rho(B_0) &= b(x) \boxplus (A_n \oslash x \oslash A_n),\\
\rho(B_i) &= B_{i-1} \boxplus B_{i-1} \text{ for } 1 \le i \le n, \text{ and}\\
\rho(S) &= B_n \boxplus (c(x) \boxplus \varepsilon).
\end{aligned}$$

The forest produced by this FSLP is

$$[\![F]\!] = b\langle a^{2^n} b\langle a^{2^n} \cdots b\langle a^{2^n}\, a^{2^n}\rangle \cdots a^{2^n}\rangle a^{2^n}\rangle,$$

where b occurs 2^n many times. This is a forest of exponential width and height. See Figure 1 for $n = 2$.

Finally, we also define expressions for strings, which we will need for various constructions.

Definition 22 (String expressions)**.** Let $\mathbf{type}_{\mathcal{S}} = \{\mathcal{S}\}$. The string signature $\mathcal{S}(\Sigma)$ over Σ is defined by the following operations:

- $\varepsilon : \mathcal{S}$,
- $a : \mathcal{S}$ for every $a \in \Sigma$, and
- $\circ : \mathcal{S}^2 \to \mathcal{S}$.

Instead of $\mathcal{E}(\mathcal{S}(\Sigma))$ we may write $\mathcal{E}_{\mathcal{S}}(\Sigma)$.

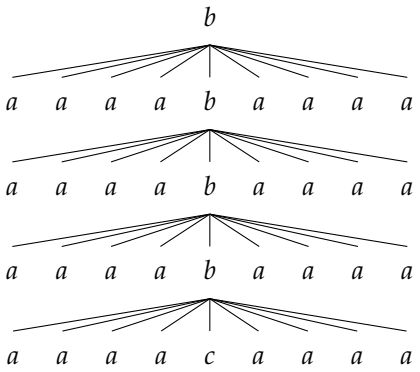


Figure 1: Forest $[\![F]\!]$ for $n = 2$ from Example 1.

An SLP for string expressions (over Σ) is called an SSLP (over Σ). Since the typed set $\mathbf{type}_S$ only uses one type, the second component of an SSLP over Σ is of the form $\Gamma : \Sigma \to \{S\}$, which is always uniquely determined. We are therefore allowed to leave Γ out, i.e. instead of saying that (V, Γ, ρ) is an SSLP, we simply say that (V, ρ) is an SSLP. Similarly, for $S \in V$ we say that (V, ρ, S) is an SSLP with a start variable.

Definition 23 (Strings)**.** For an alphabet Σ, the *set of strings over* Σ is Σ^*. Let $w = a_1 \dots a_n \in \Sigma^n$. For an index $i \in \{1, \dots, n\}$ we write $w[i] = a_i$. For indices $i, j \in \{1, \dots, n\}$ we write $w[i : j] = a_i \dots a_j$, which is ε if $i > j$. We also write $w[: i]$ instead of $w[1 : i]$, and $w[i :]$ instead of $w[i : n]$. For $0 \le k \le j$ we may write $w[-k : j]$ instead of $w[j - k + 1 : j]$, and for $0 \le k \le n$ we may also write $w[-k :]$ instead of $w[n - k + 1 : n]$. In both cases, we set $w[n + 1 : n] = \varepsilon$.

The notation $w[-k : j]$ means to take k symbols to the left starting from j and $w[-k :]$ denotes the last k symbols.

Definition 24 (Standard string algebra)**.** Let $\tau : \Sigma^* \to \{\mathcal{S}\}$. The *standard string algebra over* Σ is defined as $\mathcal{A}_\Sigma = ((\Sigma^*, \tau), \mathcal{I})$, where $\mathcal{I}(\varepsilon) = \varepsilon$, $\mathcal{I}(a) = a$ for $a \in \Sigma$ and $\mathcal{I}(\circ)(s, t) = st$. Instead of $[\![e]\!]_{\mathcal{A}_\Sigma}$ we will often simply write $[\![e]\!]$.

Again, we allow to write $e_1 \circ e_2 \circ e_3$ instead of $(e_1 \circ e_2) \circ e_3$ when using $\mathcal{A}_\Sigma$, since $\circ$ is associative in it.

Example 2. Consider the SSLP $G = (\{S, A, B, C\}, \rho, S)$ over the alphabet $\{a, b\}$ with $\rho(S) = A \circ A \circ B$, $\rho(A) = C \circ B \circ B$, $\rho(B) = C \circ a \circ C$ and $\rho(C) = b$. We have $[\![B]\!]_G = bab$, $[\![A]\!]_G = bbabbab$, and $[\![G]\!] = bbabbabbbabbabbab$. The size of $\rho(S)$, $\rho(A)$ and $\rho(B)$ is 5 and the size of $\rho(C)$ is 1. As an example, let us calculate $|\rho(S)|$, which is (in prefix notation)

$$|\circ(\circ(A, A), B)| = 1 + |\circ(A, A)| + |B| = 1 + 3 + 1.$$

DAGs are a simple form of tree compression, where identical subtrees may be identified. They are usually defined as acyclic graphs, where nodes represent subtrees. Each node v is assigned a label from some alphabet and a list of child nodes $u_1, \dots, u_n$, $n \ge 0$, i.e. a list of edges $(v, u_1), \dots, (v, u_n)$. Here, we want to view DAGs as expressions over a very simple algebra that expresses this relationship (label of a node plus children) directly:

Definition 25 (Flat Tree expressions)**.** Let $\mathbf{type}_\mathcal{D} = \{\mathcal{T}\}$. The *DAG signature* $\mathcal{S}_\mathcal{D}(\Sigma)$ over Σ has the following operations:

- $a_i : \mathcal{T}^i \to \mathcal{T}$ for all $i \in \mathbb{N}$ and $a \in \Sigma$.

We write $\mathcal{E}_\mathcal{D}(\Sigma)$ instead of $\mathcal{E}(\mathcal{S}_\mathcal{D}(\Sigma))$. We will usually leave the i from a_i out since it is clear from the context how many arguments are passed to it. Technically, however, all of these are different symbols.

Definition 26 (Standard Flat Tree algebra)**.** Let $\tau_\mathcal{D} : \mathcal{T}(\Sigma) \to \{\mathcal{T}\}$. The *standard flat tree algebra over* Σ is defined as $\mathcal{A}_{\mathcal{D},\Sigma} = ((\mathcal{T}(\Sigma), \tau_\mathcal{D}), \mathcal{I}_\mathcal{D})$, where

- $\mathcal{I}_\mathcal{D}(a_i)(t_1, \ldots, t_i) = a\langle t_1 \ldots t_i \rangle$, for every $a \in \Sigma$ and $i \in \mathbb{N}$.

In case Σ is clear from the context, we may write $\mathcal{A}_\mathcal{D}$ instead of $\mathcal{A}_{\mathcal{D},\Sigma}$.

In Chapter 3 we generalize the notion of DAGs to various algebras.

2.2 EQUALITY CHECKS

We will need to perform equality checks on various expressions. We start with string expressions which will serve as the basis for other equality checks.

Lemma 1. *Let $G = (V, \rho)$ be an SSLP and $A, B \in V$. We can test in polynomial time if $[\![A]\!]_G = [\![B]\!]_G$.*

There are many different algorithms that solve this problem in polynomial time, but they are not trivial, see [35].

We can also check if two tree expressions produce the same tree.

Lemma 2. *Let $T = (V, \Gamma, \rho)$ be a TSLP and $A, B \in V$. We can test in polynomial time whether $[\![A]\!]_T = [\![B]\!]_T$.*

This has been proven in [13]. The idea is to translate A and B into SSLPs G_A and G_B that produce string representations of $[\![A]\!]_T$ and $[\![B]\!]_T$ and then use Lemma 1 to check if $[\![G_A]\!] = [\![G_B]\!]$ which is true if and only if $[\![A]\!]_G = [\![B]\!]_G$.

A similar argument can be made for FSLPs:

Definition 27. Let $\Sigma' = \Sigma \cup \{\langle, \rangle\}$ with $\langle, \rangle \notin \Sigma$. The *depth-first-left-right* traversal of a forest is defined as $\mathrm{dflr} : \mathcal{F}(\Sigma) \to (\Sigma')^*$ with

- $\mathrm{dflr}(a\langle f \rangle) = a\langle f \rangle$ for $a \in \Sigma$ and $f \in \mathcal{F}(\Sigma)$, and
- $\mathrm{dflr}(t_1 \ldots t_n) = \mathrm{dflr}(t_1) \ldots \mathrm{dflr}(t_n)$ for $t_1 \ldots t_n \in \mathcal{T}(\Sigma)$ with $n \geq 0$.

We extend this to FSLPs as follows:

Lemma 3. *Let $F = (V, \Gamma, \rho)$ be an FSLP. We can construct in linear time an SSLP $F_\mathrm{dflr} = (V', \rho')$ such that $V_0 \subseteq V'$ and for every $A \in V_0$ we have $[\![A]\!]_{F_\mathrm{dflr}} = \mathrm{dflr}([\![A]\!]_F)$.*

Proof. Define the SSLP $F_\mathrm{dflr} = (V', \rho')$ with

$$V' = V_0 \cup \{A_\ell, A_r \mid A \in V_1\}.$$

To define ρ' we introduce the functions

$$\begin{aligned} \mathrm{str} &: \mathcal{E}(\mathcal{S}_\mathcal{F}(\Sigma), \Gamma, \mathcal{F}) \to \mathcal{E}(\mathcal{S}(\Sigma'), V'), \\ \mathrm{str}_\ell, \mathrm{str}_r &: \mathcal{E}(\mathcal{S}_\mathcal{F}(\Sigma), \Gamma, \mathcal{F}_x) \to \mathcal{E}(\mathcal{S}(\Sigma'), V') \end{aligned}$$

where str maps forest expressions to string expressions, and str_ℓ and str_r generate the string part left of x, resp. the right part of x:

- For $A \in V_0$ we set $\mathrm{str}(A) = A$.
- For $e = \varepsilon$ we set $\mathrm{str}(\varepsilon) = \varepsilon$.
- For $e = e_l \boxminus e_r$ we set $\mathrm{str}(e) = \mathrm{str}(e_l) \circ \mathrm{str}(e_r)$.
- For $e = e_t \oslash e_b$ we set $\mathrm{str}(e) = \mathrm{str}_\ell(e_t) \circ \mathrm{str}(e_b) \circ \mathrm{str}_r(e_t)$.
- For $A \in V_1$ we set $\mathrm{str}_\ell(A) = A_\ell$ and $\mathrm{str}_r(A) = A_r$.
- For $e = x$ we set $\mathrm{str}_\ell(e) = \mathrm{str}_r(e) = \varepsilon$.
- For $e = a(x)$ we set $\mathrm{str}_\ell(e) = a \circ \langle$ and $\mathrm{str}_r(e) = \rangle$.
- For $e = e_l \oslash e_r$ we set $\mathrm{str}_\ell(e) = \mathrm{str}_\ell(e_l)$ and $\mathrm{str}_r(e) = \mathrm{str}_r(e_l) \circ \mathrm{str}(e_r)$.
- For $e = e_l \oslash e_r$ we set $\mathrm{str}_\ell(e) = \mathrm{str}(e_\ell) \circ \mathrm{str}_\ell(e_r)$. and $\mathrm{str}_r(e) = \mathrm{str}_r(e_r)$.
- For $e = e_t \boxplus e_b$ we set $\mathrm{str}_\ell(e) = \mathrm{str}_\ell(e_t) \circ \mathrm{str}_\ell(e_b)$ and $\mathrm{str}_r(e) = \mathrm{str}_r(e_b) \circ \mathrm{str}_r(e_t)$.

We then set $\rho'(A) = \mathrm{str}(\rho(A))$ if $A \in V_0$ and $\rho'(A_\ell) = \mathrm{str}_\ell(\rho(A))$ and $\rho'(A_r) = \mathrm{str}_r(\rho(A))$ for $A \in V_1$. □

Lemma 4. *Let $F = (V, \Gamma, \rho)$ be an FSLP and $A, B \in V_0$. We can test in polynomial time if $[\![A]\!]_F = [\![B]\!]_F$.*

Proof. We use Lemma 3 to construct F_{dflr}. Then we use Lemma 1 to check in polynomial time if $[\![A]\!]_{F_{\mathrm{dflr}}} = [\![B]\!]_{F_{\mathrm{dflr}}}$. The correctness follows because

- for every $A \in V_0$ we have $[\![A]\!]_{F_{\mathrm{dflr}}} = \mathrm{dflr}([\![A]\!]_F)$ and
- for every $f, f' \in \mathcal{F}(\Sigma)$ we have $f = f'$ if and only if $\mathrm{dflr}(f) = \mathrm{dflr}(f')$.

□

Proposition 1. *There are polynomial time algorithms that, given an SSLP $G = (V, \rho)$ over Σ as input, compute the following.*

- *Given $X \in V$ and $1 \le i \le j \le |[\![X]\!]_G|$, compute an SSLP G' such that $[\![X]\!]_{G'} = [\![X]\!]_G[i:j]$*
- *Given $X, Y \in V$, compute the length of the longest common prefix of $[\![X]\!]_G$ and $[\![Y]\!]_G$.*

Proof. See [34] for details. □

Definition 28. A TSLP $T = (V, \Gamma, \rho)$ (resp. FSLP) is *reduced* if for every $A, B \in V_0$ we have $[\![A]\!]_T \neq [\![B]\!]_T$.

Lemma 5. *A TSLP $T = (V, \Gamma, \rho)$ (resp. FSLP) can be converted in polynomial time into a reduced TSLP (resp. FSLP) $T' = (V', \Gamma', \rho')$ such that for all $A \in V$ there is an $A' \in V'$ with $[\![A']\!]_{T'} = [\![A]\!]_T$.*

Proof. For every pair $A, B \in V$ we test if $[\![A]\!]_T = [\![B]\!]_T$ using Lemma 2 (resp. Lemma 4). In case this is true, we remove B and replace it by A. □

3 NORMAL FORM SLPS

In general, an SLP (V, Γ, ρ) is not restricted in the way it can assign expressions, i.e. $\rho(A)$ can be an expression that is arbitrarily complicated. When transforming SLPs, designing algorithms on them, etc., it is usually desirable that $\rho(A)$ is restricted in some way so that constructions have to deal with less cases, or the cases that they have to deal with somehow make the construction in question easier. In this section we will look at a few such restrictions which we call *normal forms*. The most important one will be the normal form for FSLPs. We start with a very simple normal form that is defined for all SLPs, regardless of what algebra they use. Its name is derived from the *Chomsky normal form* for context-free grammars.

Definition 29 (Chomsky normal form for SLPs)**.** An SLP (V, Γ, ρ) is in *Chomsky normal form* if for every $X \in V$ the form of $\rho(X)$ is $\rho(X) = f(X_1, \ldots, X_n)$ for some $X_1, \ldots, X_n \in V$.

Lemma 6. *Given an SLP $G = (V, \Gamma, \rho)$ over some signature $\mathcal{S}$, we can transform it in linear time into an SLP $G' = (V', \Gamma', \rho')$ with $|G'| \in \mathcal{O}(|G|)$ that is in Chomsky normal form and for each $A \in V$ we have* $\operatorname{unfold}_G(A) = \operatorname{unfold}_{G'}(A)$.

Proof. The proof of this is straight-forward: $V' \supseteq V$ will be a set with new variables that we introduce as follows: Let $\rho(X) = f(e_1, \ldots, e_n)$. Every e_i $(1 \leq i \leq n)$ with $e_i \notin V$ is replaced by a new variable $X_i \notin V$ that we add to V'. We set $\rho'(X_i) = e_i$ and $\Gamma'(X_i) = \tau_i$, where $\mathcal{S}(f) = \tau_1 \times \cdots \times \tau_n \to \tau$. We iterate this procedure until all $\rho'(A)$ for $A \in V'$ have the desired form. For all $X \in V$ we set $\Gamma'(X) = \Gamma(X)$. □

Instead of defining DAGs as graphs, we define them as SLPs that are in Chomsky normal form, where every variable is reachable. This way, the notion DAG can be used for any signature, which is for example needed later when we talk about top dags.

Definition 30 ($\mathcal{S}$-DAGs)**.** An SLP $G = (V, \Gamma, \rho, S)$ over the signature $\mathcal{S}$ is called an *$\mathcal{S}$-DAG* if it is in Chomsky normal form and every variable $A \in V$ is reachable from S. It is called a *minimal $\mathcal{S}$-DAG* if there are no two distinct variables $A, B \in V$ with $\operatorname{unfold}_G(A) = \operatorname{unfold}_G(B)$. A $\mathcal{D}$-DAG (an SLP over flat tree expressions) is simply called a DAG.

Lemma 7. *Given an expression $e \in \mathcal{E}(\mathcal{S}, \tau)$ over some signature $\mathcal{S}$ of type τ we can compute a minimal DAG* $\operatorname{mdag}(e)$ *with* $\operatorname{unfold}(\operatorname{mdag}(e)) = e$ *in linear time.*

For a proof, see [20]. An implementation that achieves amortized linear time using hashing, see e.g. [12], works as follows: We start with an SLP $G = (V, \Gamma, \rho, S)$ with $V = \{S\}$, $\Gamma(S) = \tau$ and $\rho(S) = e$. The idea is to replace subexpressions with variables bottom-up: Let $e' = f(A_1, \ldots, A_n)$ be a subexpression of $\rho(S)$, but not $\rho(S)$ itself, and assume $A_1, \ldots A_n \in V$. We test if we

already have a variable with $\rho(A) = e'$, which we implement by maintaining a hash map that maps keys from V to pairs from $\Sigma \times V^*$. If we have such an A, we replace e' in e with A. If not, we introduce a new variable A', set $\rho(A') = e'$, replace e' in e with A' and also insert $A' \to (f, A_1 \dots A_n)$ into our hash map.

A minimal DAG for a flat tree expression is what is usually called a minimal DAG of a tree, i.e. a graph that identifies all subtrees that are the same. We therefore extend the notation of mdag to trees:

Definition 31 (Minimal DAG of a tree)**.** Let $\mathrm{flat}\colon \mathcal{T}(\Sigma) \to \mathcal{E}_{\mathcal{D}}(\Sigma)$ be defined by

$$\mathrm{flat}(a\langle t_1, \dots, t_n\rangle) = a_n(\mathrm{flat}(t_1), \dots, \mathrm{flat}(t_n))$$

for $a \in \Sigma$ and $t_1, \dots, t_n \in \mathcal{T}(\Sigma)$, where $n \geq 0$. For $t \in \mathcal{T}(\Sigma)$ we define $\mathrm{mdag}(t)$ as $\mathrm{mdag}(\mathrm{flat}(t))$.

For TSLPs, we simply say that a TSLP $T = (V, \Gamma, \rho)$ over a ranked alphabet (Σ, r) in Chomsky normal form is in *normal form*. $\rho(A)$ for $A \in V$ therefore has one of the following forms:

- $\rho(A) = a$, where $a \in \Sigma$,
- $\rho(A) = B \mathbin{⦶} C$, where $B \in V_1$ and $C \in V_0$,
- $\rho(A) = B \boxplus C$, where $B, C \in V_1$, or
- $\rho(A) = a_i(A_1, \dots, A_{r(a)-1})$ where $A_1, \dots, A_{r(a)-1} \in V_0$ and $a \in \Sigma$.

Corollary 1. *Given a TSLP $T = (V, \Gamma, \rho)$, we can transform it in linear time into a TSLP $T' = (V', \Gamma', \rho')$ with $|T'| \in \mathcal{O}(|T|)$ such that T' is in normal form, $V \subseteq V'$ and $[\![A]\!]_{T'} = [\![A]\!]_T$ for all $A \in V$.*

A TSLP basically starts with an expression of the form $B \mathbin{⦶} C$, where B is transformed into a series of expressions of the form $D \boxplus E$, which in turn eventually end in expressions of the form $a_i(A_1, \dots, A_{r(a)-1})$. We can produce these expressions as a string, which will be useful in later constructions. This is done by the following SSLP:

Definition 32 (Spine SSLP for TSLPs)**.** Let $T = (V, \Gamma, \rho)$ be a TSLP over Σ in normal form. The *spine SSLP* of T is the SSLP $T_{\boxplus} = (V, \rho')$ over $\Sigma_{\boxplus}$ with

- $\Sigma_{\boxplus} = \Sigma \cup \{a_i(A_1, \dots, A_{r(a)-1}) \mid a \in \Sigma, 1 \leq i \leq r(a), A_1, \dots, A_{r(a)-1} \in V_0\}$,
- $\rho'(A) = \begin{cases} a & \text{if } \rho(A) = a, \\ B \circ C & \text{if } \rho(A) = B \mathbin{⦶} C, \\ B \circ C & \text{if } \rho(A) = B \boxplus C, \\ \rho(A) & \text{if } \rho(A) = a_i(A_1, \dots, A_{r(a)}). \end{cases}$

3.1 FACTORIZATION

This section is preparation work for the normal form for FSLPs that will be introduced shortly.

Definition 33 (Sigma-Factorization)**.** Let $\Sigma_1 \subseteq \Sigma$ and $\Sigma_2 = \Sigma \setminus \Sigma_1$. Given an SSLP $G = (V, \rho)$, a Σ_1-factorization of G is an SSLP $G' = (V', \rho')$ with $V' = \mathcal{U} \uplus \mathcal{L} \uplus V$, $[\![A]\!]_G = [\![A]\!]_{G'}$ for all $A \in V$ and where ρ' only uses the following forms:

- For $A \in V$ we have $\rho'(A) = B$ with $B \in \mathcal{L}$ or $\rho'(A) = B \circ C \circ a \circ D$ with $B, D \in \mathcal{L}$, $C \in \mathcal{U}$ and $a \in \Sigma_1$.
- For $A \in \mathcal{U}$ we have $\rho'(A) = \varepsilon$, $\rho'(A) = a \circ B$ with $a \in \Sigma_1$ and $B \in \mathcal{L}$ or $\rho'(A) = B \circ C$ with $B, C \in \mathcal{U}$.
- For $A \in \mathcal{L}$ we have $\rho'(A) = \varepsilon$, $\rho'(A) = b$ with $b \in \Sigma_2$ or $\rho'(A) = B \circ C$ with $B, C \in \mathcal{L}$.

Lemma 8. *Given $\Sigma_1 \subseteq \Sigma$ and an SSLP G we can transform it in linear time into an SSLP G' that is a Σ_1-factorization of G and $|G'| \in \mathcal{O}(|G|)$.*

Proof. Let $w \in \Sigma^*$ and $w = v_0 a_1 v_1 \dots a_n v_n$ for $n \geq 0$ with $a_i \in \Sigma_1$ and $v_0, v_i \in \Sigma_2^*$ for $1 \leq i \leq n$ which we call the Σ_1-factorization of w. Let

$$M = (\Sigma_2^* \times (\Sigma_1 \Sigma_2^*)^* \times \Sigma_1 \Sigma_2^*) \cup \Sigma_2^*.$$

We define $s \colon \Sigma^* \to M$ as $s(w) = (w_\ell, w_m, w_r)$ with

$$\begin{aligned} w_\ell &= v_0, \\ w_m &= a_1 v_1 \dots a_{n-1} v_{n-1} \text{ and} \\ w_r &= a_n v_n. \end{aligned}$$

Let us shorten $(\Sigma_2^* \times (\Sigma_1 \Sigma_2^*)^* \times \Sigma_1 \Sigma_2^*)$ to Σ_1^ϵ.

Let $G = (V, \rho)$. We assume that G is in Chomsky normal form using Lemma 6. The first step is to calculate for each $A \in V$ whether $s([\![A]\!]_G) \in \Sigma_2^*$ or $s([\![A]\!]_G) \in \Sigma_1^\epsilon$, which is straightforward:

- If $\rho(A) = \varepsilon$ then $s([\![A]\!]_G) \in \Sigma_2^*$.
- If $\rho(A) = c$ then $s([\![A]\!]_G) \in \Sigma_1^\epsilon$ if $c \in \Sigma_1$, otherwise $s([\![A]\!]_G) \in \Sigma_2^*$.
- If $\rho(A) = B \circ C$ then $s([\![A]\!]_G) \in \Sigma_1^\epsilon$ if $s([\![B]\!]_G) \in \Sigma_1^\epsilon$ or $s([\![C]\!]_G) \in \Sigma_1^\epsilon$, otherwise $s([\![A]\!]_G) \in \Sigma_2^*$.

Define $G' = (V', \rho')$ with $V' = \mathcal{U} \uplus \mathcal{L} \uplus V$ and

$$\begin{aligned} \mathcal{U} = {} & \{U_\varepsilon\} \\ & \cup \{A_m \mid A \in V,\ s([\![A]\!]_G) \in \Sigma_1^\epsilon\} \\ & \cup \{U_{BC}, U'_{BC} \mid A \in V,\ \rho(A) = B \circ C,\ s([\![B]\!]_G) \in \Sigma_1^\epsilon,\ s([\![C]\!]_G) \in \Sigma_1^\epsilon\}, \\ \mathcal{L} = {} & \{A_\ell, A_r \mid A \in V,\ s([\![A]\!]_G) \in \Sigma_1^\epsilon\} \\ & \cup \{L_{BC} \mid A \in V,\ \rho(A) = B \circ C,\ s([\![B]\!]_G) \in \Sigma_1^\epsilon,\ s([\![C]\!]_G) \in \Sigma_1^\epsilon\} \\ & \cup \{A' \mid A \in V,\ s([\![A]\!]_G) \in \Sigma_2^*\}. \end{aligned}$$

We now proceed to define ρ'. In addition, we use induction to define $\sigma_A \in \Sigma$ for each $A \in V$ with $[\![A]\!]_G \in \Sigma_1^\epsilon$. First, we set $\rho'(U_\varepsilon) = \varepsilon$.

- For $\rho(A) = \varepsilon$ we set $\rho'(A') = \varepsilon$.
- For $\rho(A) = b \in \Sigma_2$ we set $\rho(A') = b$.
- For $\rho(A) = a \in \Sigma_1$ we set $\rho'(A_\ell) = \rho'(A_m) = \rho'(A_r) = \varepsilon$ and $\sigma_A = a$.
- For $\rho(A) = B \circ C$ we have the following cases:
  - If $s([\![B]\!]_G) \in \Sigma_2^*$ and $s([\![C]\!]_G) \in \Sigma_2^*$ we set $\rho'(A') = B' \circ C'$.
  - If $s([\![B]\!]_G) \in \Sigma_2^*$ and $s([\![C]\!]_G) \in \Sigma_1^\epsilon$ we set $\rho'(A_\ell) = B' \circ C_\ell$, $\rho'(A_m) = \rho'(C_m)$, $\rho'(A_r) = \rho'(C_r)$, and $\sigma_A = \sigma_C$.

– If $s([\![B]\!]_G) \in \Sigma_1^\epsilon$ and $[\![C]\!]_G \in \Sigma_2^*$ we set $\rho'(A_\ell) = \rho'(B_\ell)$, $\rho'(A_m) = \rho'(B_m)$, $\rho'(A_r) = B_r \circ C'$, and $\sigma_A = \sigma_B$.
– If $s([\![B]\!]_G) \in \Sigma_1^\epsilon$ and $s([\![C]\!]_G) \in \Sigma_1^\epsilon$ we set $\rho'(A_\ell) = \rho'(B_\ell)$, $\rho'(A_r) = \rho'(C_r)$, and $\sigma_A = \sigma_C$. For A_m we want that

$$[\![\rho'(A_m)]\!]_{G'} = [\![B_m \circ (\sigma_B \circ (B_r \circ C_\ell)) \circ C_m]\!]_{G'}$$

but in order to produce the desired rules, we have to split $\rho'(A_m)$ as follows: $\rho'(A_m) = B_m \circ U_{BC}$, $\rho'(U_{BC}) = U'_{BC} \circ C_m$, $\rho'(U'_{BC}) = \sigma_B \circ L_{BC}$, $\rho'(L_{BC}) = B_r \circ C_\ell$.

Finally, we have to define the rules for each $A \in V$:

- If $s([\![A]\!]_G) \in \Sigma_2^*$, then $\rho'(A) = A' \circ U_\varepsilon$.
- If $s([\![A]\!]_G) \in \Sigma_1^\epsilon$, then $\rho'(A) = A_\ell \circ A_m \circ \sigma_a \circ A_r$.

□

3.2 NORMAL FORM FOR FSLPS

To obtain the desired normal form for FSLPs, we first start with the so-called weak normal form. This normal form is easy to obtain and it is only used in the construction for the actual normal form.

Definition 34 (Weak normal form)**.** An FSLP (V, Γ, ρ) is in *weak normal form* if $\rho(X)$ for all $X \in V$ only takes on the following forms:

- ε,
- $T \otimes B$, where $T \in V_1$, $B \in V_0$,
- $T \boxplus B$, where $T, B \in V_1$,
- $a(x)$, where $a \in \Sigma$,
- $L \oslash x \oslash R$, where $L, R \in V_0$.

Lemma 9. *Given an FSLP $F = (V, \Gamma, \rho)$ we can in linear time transform it into a new FSLP $F' = (V', \Gamma', \rho')$ that is in weak normal form, $[\![F]\!] = [\![F']\!]$ and $|F'| \in \mathcal{O}(|F|)$.*

Proof. We assume that F is in Chomsky normal form using Lemma 6. The set of new variables are

$$\begin{aligned} V' = V &\cup \{E_\varepsilon\} \\ &\cup \{A' \mid \rho(A) = B \boxminus C,\ B, C \in V\} \\ &\cup \{A' \mid \rho(A) = B \oslash C,\ B, C \in V\} \\ &\cup \{A' \mid \rho(A) = B \oslash C,\ B, C \in V\}. \end{aligned}$$

We set $\Gamma'(A) = \Gamma(A)$ for all $A \in V$, $\Gamma'(E_\varepsilon) = \mathcal{F}$ and $\Gamma'(A') = \mathcal{F}_x$ for all $A \in V$. For ρ', we first set $\rho'(E_\varepsilon) = \varepsilon$. The other cases are as follows:

- In case $\rho(A) = \varepsilon$, $\rho(A) = B \otimes C$, $\rho(A) = B \boxplus C$, or $\rho(A) = a(x)$ we set $\rho'(A) = \rho(A)$.
- For $\rho(A) = x$ we set $\rho'(A) = E_\varepsilon \oslash x \oslash E_\varepsilon$.
- For $\rho(A) = B \boxminus C$ we set $\rho'(A') = B \oslash x \oslash C$ and $\rho'(A) = A' \otimes E_\varepsilon$.

- For $\rho(A) = B \oslash C$ we set $\rho'(A') = E_\varepsilon \oslash x \oslash C$ and $\rho'(A) = A' \boxplus B$.
- For $\rho(A) = B \oslash C$ we set $\rho'(A') = B \oslash x \oslash E_\varepsilon$ and $\rho'(A) = A' \boxplus C$.

□

We now finally come to the definition of our normal form.

Definition 35 (Normal form for FSLPs)**.** An FSLP (V, Γ, ρ) is in *normal form* if $\rho(X)$ for all $X \in V$ only takes on the following forms:

- ε,
- $T \oslash (a(x) \oslash B)$, where $a \in \Sigma$, $T \in V_1$ and $B \in V_0$,
- $L \boxminus R$, where $L, R \in V_0$,
- x,
- $a(x) \boxplus (L \oslash x \oslash R)$, where $a \in \Sigma$ and $L, R \in V_0$,
- $T \boxplus B$, where $T, B \in V$.

We basically have two kinds of expressions: *Vertical expressions*, which are x, $a(x) \boxplus (L \oslash x \oslash R)$ and $T \boxplus B$, and *horizontal expressions*, which are ε, $T \oslash (a(x) \oslash B)$ and $L \boxminus R$.

Horizontal expressions build forests using expressions of the form $L \boxminus R$, which in turn yields expressions of the form

$$T_1 \oslash (a_1(x) \oslash B_1) \boxminus \cdots \boxminus T_n \oslash (a_n(x) \oslash B_n).$$

Every of these sub-expressions starts with T_i, which uses expressions of the form $T \boxplus B$. These in turn each yield an expression of the form

$$b_1(x) \boxplus (L_1 \oslash x \oslash R_1) \boxplus \cdots \boxplus b_m(x) \boxplus (L_m \oslash x \oslash R_m).$$

This structure has several advantages: Similar to the spine SSLP for TSLPs, we define a spine SSLP for FSLPs, which is built out of the vertical expressions. In addition, we define the rib SSLP which is built out of the horizontal expressions. Furthermore, in $a(x) \boxplus (L \oslash x \oslash R)$ we know that x is always bound to a tree. The only place in which it can be bound to a forest is the $a(x) \oslash B$ part of horizontal expressions.

In the following, we show how to obtain the normal form for FSLPs.

Theorem 1. *Given an FSLP $F = (V, \Gamma, \rho)$ we can transform it in linear time into a new FSLP $F' = (V', \Gamma', \rho')$ that is in normal form with $V_0 \subseteq V'$, $[\![X]\!]_F = [\![X]\!]_{F'}$ for all $X \in V_0$ and $|F'| \in \mathcal{O}(|F|)$.*

Proof. We first assume that F is in weak normal form using Lemma 9. Let

$$\Sigma_h = \{L \oslash x \oslash R \mid X \in V,\ \rho(X) = L \oslash x \oslash R\},$$
$$\Sigma_v = \{a(x) \mid X \in V,\ \rho(X) = a(x)\}.$$

We define the *Spine SSLP for the weak normal form* as the SSLP $F_\boxplus = (V_1, \rho')$ over $\Sigma_h \cup \Sigma_v$ with

- $\rho'(X) = A \circ B$ if $\rho(X) = A \boxplus B$ and
- $\rho'(X) = \rho(X)$ if $\rho(X) = a(x)$ or $\rho(X) = L \oslash x \oslash R$.

Then we apply Lemma 33 to create a Σ_v-factorization (V', ρ'') out of $F_{\boxplus}$. Here, we have that $V' = \mathcal{L} \uplus \mathcal{U} \uplus V_1$ with the appropriate $\mathcal{L}$ and $\mathcal{U}$. The new variable set is defined as

$$\begin{aligned} V'' = \mathcal{U} \cup V_0 \cup \{A_\ell, A_r \mid A \in \mathcal{L}\} \\ \cup \{A' \mid A \in V_0, \rho(A) = B \circledcirc C, \rho''(B) \in \mathcal{L}\} \\ \cup \{A_1, A_2 \mid A \in V_0, \rho(A) = B \circledcirc C, \rho''(B) = D \circ E \circ a(x) \circ F\}. \end{aligned}$$

The new FSLP is $F' = (V'', \Gamma', \rho^*)$ with $\Gamma'(B) = \mathcal{F}_x$ for all $B \in \mathcal{U}$ and $\Gamma'(X) = \mathcal{F}$ for all $X \in V'' \setminus \mathcal{U}$. The cases for ρ^* are as follows:

- If $A \in V_0$ with $\rho(A) = \varepsilon$ then $\rho^*(A) = \varepsilon$.
- If $A \in V_0$ with $\rho(A) = B \circledcirc C$, $B \in V_1$, $C \in V_0$ then
  - if $\rho''(B) = D$, so $D \in \mathcal{L}$, then we want that $[\![A]\!]_{F'} = [\![D_\ell \boxminus C \boxminus D_r]\!]_{F'}$ which is achieved by splitting $\rho^*(A)$ as follows: $\rho^*(A) = A' \boxminus D_r$ and $\rho^*(A') = D_\ell \boxminus C$.
  - if $\rho''(B) = D \circ E \circ a(x) \circ F$, so $D, F \in \mathcal{L}$ and $E \in \mathcal{U}$, then we want that $[\![A]\!]_{F'} = [\![D_\ell \boxminus (E \circledcirc (a(x) \circledcirc F)) \boxminus D_r]\!]_{F'}$ which is achieved by splitting $\rho^*(A)$ as follows: $\rho^*(A_1) = E \circledcirc (a(x) \circledcirc F)$, $\rho^*(A_2) = A_1 \boxminus D_r$ and $\rho^*(A) = D_\ell \boxminus A_2$.
- If $A \in \mathcal{L}$ with $\rho''(A) = \varepsilon$ then $\rho^*(A_\ell) = \rho^*(A_r) = \varepsilon$.
- If $A \in \mathcal{L}$ with $\rho''(A) = B \circ C$, $B, C \in \mathcal{L}$, then $\rho^*(A_\ell) = B_\ell \boxminus C_\ell$ and $\rho^*(A_r) = C_r \boxminus B_r$.
- If $A \in \mathcal{L}$ with $\rho''(A) = L \circledcirc x \circledcirc R$ then $\rho^*(A_\ell) = \rho^*(L)$ and $\rho^*(A_r) = \rho^*(R)$.
- If $A \in \mathcal{U}$ with $\rho''(A) = \varepsilon$ then $\rho^*(A) = x$.
- If $A \in \mathcal{U}$ with $\rho''(A) = a(x) \circ B$, $B \in \mathcal{L}$ then $\rho^*(A) = a(x) \boxplus (B_\ell \circledcirc x \circledcirc B_r)$.
- If $A \in \mathcal{U}$ with $\rho''(A) = B \circ C$, $B, C \in \mathcal{U}$, then $\rho^*(A) = B \boxplus C$.

□

Definition 36 (Spine SSLP for FSLPs)**.** Let $F = (V, \Gamma, \rho)$ be an FSLP in normal form. The *spine SSLP* of F is the SSLP $F_{\boxplus} = (V_1, \rho')$ over $\Sigma_{\boxplus}$ with

- $\Sigma_{\boxplus} = \{a(x) \boxplus (L \circledcirc x \circledcirc R) \mid a \in \Sigma, L, R \in V_0\}$,
- $\rho'(A) = \begin{cases} \varepsilon & \text{if } \rho(A) = x, \\ \rho(A) & \text{if } \rho(A) = a(x) \boxplus (L \circledcirc x \circledcirc R), \\ B \circ C & \text{if } \rho(A) = B \boxplus C. \end{cases}$

The *spine* of a given variable $B \in V_1$ is defined as

$$\operatorname{spine}_F(B) = [\![B]\!]_{F_{\boxplus}}.$$

Definition 37 (Rib SSLP)**.** Let $F = (V, \Gamma, \rho)$ be an FSLP in normal form. The *rib SSLP* of F is the SSLP $F_{\boxminus} = (V_0, \rho')$ over $\Sigma_{\boxminus}$ with

- $\Sigma_{\boxminus} = \{B \circledcirc (a(x) \circledcirc C) \mid B \in V_1, C \in V_0, a \in \Sigma\}$,
- $\rho'(A) = \begin{cases} \varepsilon & \text{if } \rho(A) = \varepsilon, \\ \rho(A) & \text{if } \rho(A) = B \circledcirc (a(x) \circledcirc C), \\ B \circ C & \text{if } \rho(A) = B \boxminus C. \end{cases}$

Since we can take any FSLP and transform it in linear time into an equivalent FSLP in normal form, we may always assume that our FSLPs are in normal form. However, in some proofs, we need slight variations of the normal form in which the rib and/or spine SSLPs do not use ε. These normal forms are easy to obtain in linear time, which is summarized in the following lemma:

Lemma 10. *We can transform an FSLP $F = (V, \Gamma, \rho, S)$ in linear time into a new FSLP $F' = (V', \Gamma', \rho', S)$ with $[\![F]\!] = [\![F']\!]$ and $|F'| \in \mathcal{O}(|F|)$ such that:*

1. *For every $A \in V'$ we have $[\![A]\!]_{F'} \neq x$. The allowed expressions are*
   - ε,
   - $L \boxminus R$,
   - $B \circledcirc (a(x) \circledcirc C)$,
   - $a(x) \circledcirc C$,
   - $a(x) \boxplus (L \oslash x \oslash R)$ *and*
   - $T \boxplus B$.
2. *If $[\![F]\!] \neq \varepsilon$ then for every $A \in V'$ we have $[\![A]\!]_{F'} \neq \varepsilon$. The allowed expressions are*
   - $L \boxminus R$,
   - $B \circledcirc (a(x) \circledcirc C)$,
   - $B \circledcirc (a(x) \circledcirc \varepsilon)$,
   - $a(x) \boxplus (L \oslash x \oslash R)$,
   - $a(x) \boxplus (L \oslash x)$,
   - $a(x) \boxplus (x \oslash R)$,
   - $a(x)$ *and*
   - $T \boxplus B$.
3. *If $[\![F]\!] \neq \varepsilon$ then for every $A \in V'$ we have $[\![A]\!]_{F'} \neq x$ and $[\![A]\!]_{F'} \neq \varepsilon$. The allowed expressions are*
   - $L \boxminus R$,
   - $B \circledcirc (a(x) \circledcirc C)$,
   - $B \circledcirc (a(x) \circledcirc \varepsilon)$,
   - $a(x) \circledcirc C$,
   - $a(x) \circledcirc \varepsilon$,
   - $a(x) \boxplus (L \oslash x \oslash R)$,
   - $a(x) \boxplus (L \oslash x)$,
   - $a(x) \boxplus (x \oslash R)$,
   - $a(x)$ *and*
   - $T \boxplus B$.

Proof. The transformations are similar to ε-elimination in context-free grammars.

1. First, we need to calculate for which variables $A \in V_1$ we have $[\![A]\!]_F = x$: We have $[\![A]\!]_F = x$ if $\rho(A) = x$ and $[\![A]\!]_F \neq x$ if $\rho(A) = a(x) \boxplus (B \oslash x \oslash C)$. In case $\rho(A) = B \boxplus C$ then $[\![A]\!]_F = x$ if and only if $[\![B]\!]_F = [\![C]\!]_F = x$. We remove all $A \in V$ from V' with $[\![A]\!]_F = x$, i.e. we initially set V' to $V \setminus \{A \in V \mid [\![A]\!]_F = x\}$, and set $\rho'(A) = \rho(A)$ except in two cases: If $\rho(A) = B \boxplus C$, we may have removed B, C or both. In case B has been removed but not C, we have $[\![B]\!]_F = x$ and therefore $[\![A]\!]_F = [\![C]\!]_F$. We replace every occurrence of A by C and also remove A from V'. Similarly, in case C has been removed but not B we have $[\![A]\!]_F = [\![B]\!]_F$, so we replace every occurrence of A by B and also remove A from V'. In case both variables have been removed, then A has also been removed. Finally, if $\rho(A) = B \otimes (a(x) \otimes C)$ we might have removed B. If this is the case, we set $\rho'(A) = a(x) \otimes C$.

2. The transformation is very similar to the previous one. First, we calculate for which variables $A \in V_0$ we have $[\![A]\!]_F = \varepsilon$: We have $[\![A]\!]_F = \varepsilon$ if $\rho(A) = \varepsilon$ and $[\![A]\!]_F \neq \varepsilon$ if $\rho(A) = B \otimes (a(x) \otimes C)$. In case $\rho(A) = B \boxminus C$ then $[\![A]\!]_F = \varepsilon$ if and only if $[\![B]\!]_F = [\![C]\!]_F = \varepsilon$. We initially set V' to $V \setminus \{A \in V \mid [\![A]\!]_F = \varepsilon\}$, Again, we set $\rho'(A) = \rho(A)$ except in two cases: If $\rho(A) = B \boxminus C$, we may have removed B, C or both. In case we only removed C we have $[\![A]\!]_F = [\![B]\!]_F$, in which case we replace every occurrence of A by B and remove A from V'. Similarly, in case we only removed B we have $[\![A]\!]_F = [\![C]\!]_F$, in which case we replace every occurrence of A by C and remove A from V'. If we removed both B and C, then A has also been removed. If $\rho(A) = B \otimes (a(x) \otimes C)$ we may have removed C. If this is the case then we set $\rho'(A) = B \otimes (a(x) \otimes \varepsilon)$. If $\rho(A) = a(x) \otimes (B \oslash x \oslash C)$ we set $\rho'(A) = a(x)$ if $[\![B]\!]_F = [\![C]\!]_F = \varepsilon$, $\rho'(A) = a(x) \otimes (B \oslash x)$ if $[\![C]\!]_F = \varepsilon$ and $[\![B]\!]_F \neq \varepsilon$, and $\rho'(A) = a(x) \otimes (x \oslash C)$ if $[\![B]\!]_F = \varepsilon$ and $[\![C]\!]_F \neq \varepsilon$.

3. If we want to both disallow $[\![A]\!]_F = x$ and $[\![A]\!]_F = \varepsilon$, then we can apply the previous two transformations in succession. After the first step, we may end up with $a(x) \otimes C$ and after the second step, we might have removed C. If this is the case, we have to replace C with ε and obtain $a(x) \otimes \varepsilon$.

□

4 NAVIGATION

The ability to navigate efficiently in a tree is a basic prerequisite for most tree querying procedures. For instance, the DOM representation available in web browsers through JavaScript provides tree navigation primitives (see, e.g., [19]). Tree navigation has been intensively studied in the context of succinct tree representations. Here, the goal is to represent a tree by a bit string, whose length is asymptotically equal to the information-theoretic lower bound. For instance, for trees with n nodes the information-theoretic lower bound is $2n + o(n)$ and there exist succinct representations (e.g., the balanced parentheses representation) that encode an ordered tree of size n by a bit string of length $2n + o(n)$. In addition there exist such encodings that allow to navigate in the tree in constant time (and support many other tree operations), see e.g. [42] for a survey.

Navigation of trees is also studied in [5]. The authors show that a single navigation step in a tree t can be carried out in time $O(\log|t|)$ in the top dag. Nodes are represented by their preorder numbers, which need $O(\log|t|)$ bits. In [6] an analogous result has been shown for unranked trees that are represented by SSLPs for the balanced parentheses representation of the tree. This covers also TSLPs: from a TSLP T for t one can easily compute in linear time an SSLP for the balanced parentheses representation of t. In some sense our results for trees are orthogonal to the results of [6]:

- We can navigate, determine node labels, and check equality of subtrees in time $\mathcal{O}(1)$, but our representation of tree nodes needs space $\mathcal{O}(|T|)$.
- Bille et al. [6] can navigate and execute several other tree queries (e.g. lowest common ancestor computations) in time $\mathcal{O}(\log|t|)$, but their node representation (preorder numbers) only need space $\mathcal{O}(\log|t|) \leq \mathcal{O}(|T|)$.

An implementation of navigation over TSLP-compressed trees is given in [40]. Their worst-case time per navigation step is $\mathcal{O}(h)$ where h is the height of the TSLP. The authors demonstrate that on XML trees, full traversals take about 5–7 longer than over succinct trees (based on an implementation by Sadakane) while using 3–15 times less space; thus, their implementation provides a competitive space/time trade-off.

Checking equality of subtrees is trivial for minimal DAGs, since every subtree is uniquely represented. For so called SL grammar-compressed DAGs (which can be seen as TSLPs with certain restrictions) it was shown in [10] that equality of subtrees can be checked in time $\mathcal{O}(\log|t|)$ for given preorder numbers.

In this section, we use the word RAM model, in which registers contain integers of a certain bit length. This bit length depends on the input of the algorithm. It takes constant time to compare two registers, add two registers,

and so on. The space needed by the algorithm is the number of registers written. Let G be an SSLP.

- For navigation on S we allow a bit length of $\mathcal{O}(\log|G|)$, since we only have to store numbers of length at most $|G|$.
- For equality checks we need a bit length of $\mathcal{O}(\log|[\![G]\!]|) \leq \mathcal{O}(|G|)$, which is the same assumption as in [5, 6].

In the following sections we have to deal with many operations that can fail. Instead of treating these as partial functions, we define $M_\perp := M \uplus \{\perp\}$ for any set M, where $\perp$ takes the role of an error value. This way, we have to explicitly test for $\perp$ or return $\perp$ where appropriate, and hopefully make error handling more visible.

4.1 NAVIGATING SSLPS

In [26] a data structure was presented that allows to produce the string of an SSLP from left to right, where each step requires constant time. The data structure itself requires linear space in the size of the SSLP. Here, we modify this slightly so that we can move left or right through the string and print the current character, where each step requires constant time.

Lemma 11. *Let $G = (V, \rho)$ be an SSLP. We precompute in time $\mathcal{O}(|G|)$ some data structure $P(G)$ of size $\mathcal{O}(|G|)$ that makes it possible to implement the following operations in constant time, where by $\mathcal{N}(G)$ (which will be defined later) we denote the set of possible states an SSLP traversal of G can be in:*

$\triangleleft : V \to \mathcal{N}(G)_\perp$*: Go to the first character.*

$\triangleright : V \to \mathcal{N}(G)_\perp$*: Go to the last character.*

$\mathrm{st} : \mathcal{N}(G) \to V$*: Get the start variable.*

$z : \mathcal{N}(G) \to \Sigma$*: Get the current character.*

$\rightarrow : \mathcal{N}(G) \to \mathcal{N}(G)_\perp$*: Go to the next character.*

$\leftarrow : \mathcal{N}(G) \to \mathcal{N}(G)_\perp$*: Go to the previous character.*

The size of each element from $\mathcal{N}(G)$ is bounded by $\mathcal{O}(h(G))$.

Most of the above operations can fail, which is indicated by the special value $\perp$: $\triangleleft(A)$ and $\triangleright(A)$ fail if $[\![A]\!]_G = \varepsilon$, and $\leftarrow$ and $\rightarrow$ fail if we leave the word $[\![A]\!]_G$. Semantically, an element $X \in \mathcal{N}(G)$ with $\mathrm{st}(X) = S$ is an index, called $\mathrm{idx}(X)$, of $[\![S]\!]_G$. The operations manipulate the index as follows: $\triangleleft(S)$ sets the index to 1, $\triangleright(S)$ sets the index to $|[\![S]\!]_G|$, $\rightarrow$ increments the index and $\leftarrow$ decrements the index. The implementation will not actually store $\mathrm{idx}(X)$ in X. Instead, we store a certain representation of a root-leaf path of the syntax tree.

Consider a binary tree in which you want to move from one leaf to the next. The standard algorithm for this operates on the sequence of nodes that go from the root to the leaf, and is done in four steps:

1. As long as the current node is a right child, go to the parent node.
2. Now the current node is a left child. Go to its parent node.
3. Go to the right child.

4. As long as the current node is not a leaf, go to the left child.

We want to implement this procedure in constant time on the syntax tree of an SSLP G. For this, we first assume that G is in Chomsky normal form, using Lemma 6. In addition, we want that for every $A \in V$ with $\rho(A) = B \circ C$ we have that $[\![B]\!]_G \neq \varepsilon$ and $[\![C]\!]_G \neq \varepsilon$, which is easy to implement and works in a way similar to the construction in Lemma 10. Each element from $D \in V$ with $[\![D]\!]_G \neq \varepsilon$ therefore has a syntax tree, where the inner nodes are labelled with elements from $A \in V$ with $\rho(A) = B \circ C$ for some $B, C \in V$ and the leaf nodes are labelled with elements from $A \in V$ where $\rho(A) = a$ for some $a \in \Sigma$. Using a sequence of nodes as our data structure makes it impossible to do steps 1 and 4 in constant time, which is why we use a different data structure.

Instead of a sequence of nodes that goes from the root to a leaf, we could use a data structure that records the labels of nodes we visit and if we went into the left child or the right child. Formally, this is an element of $(V \times \{\ell, r\})^* \times V$. A crucial observation is that a sequence $w(A_1, d) \dots (A_n, d)w'$ (so we have n steps into the same direction $d \in \{\ell, r\}$) can be compressed to $w(A_1, d)w'$ without losing any information: We can reobtain A_{i+1} (where $1 \leq i < n$) by looking at $\rho(A_i) = B \circ C$: If $d = \ell$ then $A_{i+1} = B$ and if $d = r$ then $A_{i+1} = C$. Since a sequence of nodes that go only to the left or only to the right now have succinct representations, steps 1 and 4 can now be implemented in constant space. We still have to argue that they can also be implemented in constant time. From now on, we only want to use the compressed versions of traversals: A sequence $(A_1, d_1) \dots (A_n, d_n)A_{n+1} \in (V \times \{\ell, r\})^* \times V$ is called *valid* if $d_i \neq d_{i+1}$ for all $1 \leq i < n$. We choose

$$\mathcal{N}(G) = \{X \in (V \times \{\ell, r\})^* \times V \mid X \text{ is valid}\}.$$

The difficulty with this data structure is to reobtain a parent node. Consider an original sequence of left steps $w(A_1, \ell) \dots (A_n, \ell)A$ (the argument is similar for right steps). Here, we can simply go from A to A_n. In the compressed representation however, we only have $w(A_1, \ell)A$. To reobtain A_n, we cannot expand $A_1 \dots A_{n-1}$ since this would require linear time. In addition, what A_n is depends on both A_1 and A.

Formally, let $\mathcal{L}\colon V \to V_\perp^*$ be defined by $\mathcal{L}(A) = B\mathcal{L}(B)$ if $\rho(A) = B \circ C$ for some $B, C \in V$, $\mathcal{L}(A) = \varepsilon$ if $\rho(A) = a$ for some $a \in \Sigma$, and $\mathcal{L}(A) = \perp$ if $\rho(A) = \varepsilon$. Let $\text{reduce}_\ell\colon V^2 \to V_\perp$ be defined by

$$\text{reduce}_\ell(A, A') = \begin{cases} B & \text{if } \mathcal{L}(A) = wBA'w' \text{ for some } w, w' \in V^*, \\ \perp & \text{otherwise.} \end{cases}$$

We will have to argue that reduce_ℓ can be implemented in linear space and constant time (it can obviously be implemented in quadratic space and constant time using a lookup-table). This is however done after presenting the traversal algorithm itself.

The function $\text{expand}_\ell\colon V \to V_\perp$ retrieves the last element of $\mathcal{L}(A)$ or $\perp$ if $\mathcal{L}(A) = \varepsilon$. It can easily be represented as an array of size $|V|$. The operations $\lhd$, $\rhd$ and z are implemented as follows:

- $\lhd(A) = \begin{cases} (A, \ell)\text{expand}_\ell(A) & \text{if } \text{expand}_\ell(A) \neq \perp, \\ A & \text{if } \text{expand}_\ell(A) = \perp \text{ and } \rho(A) \neq \varepsilon, \\ \perp & \text{if } \rho(A) = \varepsilon. \end{cases}$
- $\rhd$ is similar to $\lhd$ but uses r instead of ℓ.

- $z(wA) = a$ where $\rho(A) = a$, $w \in (V \times \{\ell, r\})^*$ and $A \in V$.

The operation $\to$ is split into four functions $r_\nwarrow, r_\nearrow, r_\searrow, r_\swarrow : \mathcal{N}(G) \to \mathcal{N}(G)_\perp$ which correspond to the four operations on the syntax tree we discussed earlier. Let $w \in (V \times \{\ell, r\})^*$ and $A \in V$. We set:

- $r_\nwarrow(wA) = \begin{cases} \perp & \text{if } w = \varepsilon, \\ wA & \text{if } w = w'(A', \ell), \\ w'A' & \text{if } w = w'(A', r). \end{cases}$
- $r_\nearrow(wA) = \begin{cases} \perp & \text{if } w = \varepsilon \text{ or } w = w'(A', r), \\ w(A', \ell)B & \text{if } w = w'(A', \ell) \text{ and } \mathrm{reduce}_\ell(A', A) = B \neq \perp, \\ w'A' & \text{if } w = w'(A', \ell) \text{ and } \mathrm{reduce}_\ell(A', A) = \perp. \end{cases}$
- $r_\searrow(wA) = \begin{cases} \perp & \text{if } \rho(A) = a \in \Sigma, \\ w(A, r)C & \text{if } w = w'(A', \ell) \text{ and } \rho(A) = B \circ C, \\ w'(A', r)C & \text{if } w = w'(A', r) \text{ and } \rho(A) = B \circ C. \end{cases}$
- $r_\swarrow(wA) = \begin{cases} wA & \text{if } \rho(A) = a \in \Sigma, \\ w(A, \ell)\mathrm{expand}_\ell(A) & \text{if } \rho(A) = B \circ C. \end{cases}$

We set $\to = r_\swarrow \circ r_\searrow \circ r_\nearrow \circ r_\nwarrow$, where $\perp$ is returned as soon as one of the functions returns it. When we use $\to$, some cases where the individual functions return $\perp$ cannot occur, for example we cannot actually have the case $r_\searrow(wA) = \perp$ because $\rho(A) = a \in \Sigma$, since $r_\nearrow$ uses reduce_ℓ to replace the last $A \in V$. Each of these functions can be implemented in constant time, since they modify their arguments by adding or removing a single element to resp. from the back. They also make use of ρ, reduce_ℓ and expand_ℓ, all of which are implemented in constant time. The implementation of $\leftarrow$ is basically the mirrored version of $\to$, where expand_r and reduce_r are used: Formally, let $\mathcal{R} : V \to V_\perp^*$ be defined by $\mathcal{R}(A) = C\mathcal{R}(C)$ if $\rho(A) = B \circ C$ for some $B, C \in V$, $\mathcal{R}(A) = \varepsilon$ if $\rho(A) = a$ for some $a \in \Sigma$ and $\mathcal{R}(A) = \perp$ if $\rho(A) = \varepsilon$. Let $\mathrm{reduce}_r : V^2 \to V_\perp$ be defined by

$$\mathrm{reduce}_r(A, A') = \begin{cases} B & \text{if } \mathcal{R}(A) = wBA'w' \text{ for some } w, w' \in V^*, \\ \perp & \text{otherwise.} \end{cases}$$

We also split $\leftarrow$ into four functions $\ell_\nwarrow, \ell_\nearrow, \ell_\searrow, \ell_\swarrow : \mathcal{N}(G) \to \mathcal{N}(G)_\perp$ which are all defined in a similar way to their r-counterparts.

All that is left to do is to argue that we can precompute reduce_ℓ (resp. reduce_r) in linear time and such that it takes constant time to execute, which we do as follows: Given two nodes u and v in a tree, where u is an ancestor of v, the *next link*-query returns the child of u that is also an ancestor of v. The following result is mentioned in [26]:

Proposition 2. *A trie T can be represented in space $\mathcal{O}(|T|)$ such that any next link-query can be answered in time $\mathcal{O}(1)$. Moreover, this representation can be computed in time $\mathcal{O}(|T|)$ from T.*

If we have two variables A and B, where B appears in $\mathcal{L}(A)$ somewhere, then we want to know what the symbol left of B is. This basically corresponds to the next link-query for the nodes B and A in a trie (suffix tree) for the strings

$$\mathcal{L} = \{A\mathcal{L}(A) \mid A \in V, \rho(A) \neq \varepsilon\} \circ \{\$\} \subseteq V^* \circ \{\$\},$$

where the symbol $\$ \notin \Sigma$ is used to give the trie a "dummy root node". This trie has size $\mathcal{O}(|V|)$ since there are at most $|V|+1$ nodes in it. Then we have that $\text{reduce}_\ell(A,B) = C$ if and only if the next link-query for the nodes labelled with A and B is the node labelled with C. Consider the following example:

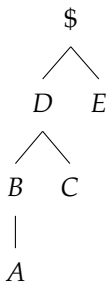
Example 3. Let $G = (V,\rho)$ be an SSLP over $\{a,b\}$ with $V = \{A,B,C,D,E,F\}$ and

$$\begin{aligned}\rho(A) &= B \circ C,\\ \rho(B) &= D \circ D,\\ \rho(C) &= D \circ E,\\ \rho(D) &= a,\\ \rho(E) &= b,\\ \rho(F) &= \varepsilon.\end{aligned}$$

Then we have $\mathcal{L}(A) = BD$, $\mathcal{L}(B) = D$, $\mathcal{L}(C) = D$, $\mathcal{L}(D) = \mathcal{L}(E) = \varepsilon$, $\mathcal{L}(F) = \perp$ and

$$\mathcal{L} = \{ABD, BD, CD, D, E\} \circ \{\$\}$$

such that the suffix tree is



So for example, we have $\text{reduce}_\ell(A,D) = B$, since B is the child of D on the path from A to D. On the other hand, $\text{reduce}_\ell(B,D) = \perp$, since D is the parent of B, i.e. there is no node between B and D on the path from B to D.

Since each $X = (A_1,d_1)\dots(A_n,d_n)A_{n+1} \in \mathcal{N}(G)$ represents a path into the syntax tree of G, so $n+1 \in \mathcal{O}(h(A_1))$, the size $|X|$ is bounded by $h(G)$, which is in $\mathcal{O}(|G|)$. We can use the result from [23] to reduce $h(G)$ to $\mathcal{O}(\log|G|)$. This can also be applied in the following sections where we navigate TSLPs and FSLPs using their spine (and rib) SSLPs.

4.2 NAVIGATING TSLPS

Theorem 2. *Let $T = (V,\Gamma,\rho)$ be a TSLP over (Σ,r) in normal form. We can precompute in time $\mathcal{O}(|T|)$ some data structure $P(T)$, and we define some data structure $\mathcal{N}(T)$, such that the size of each element from $\mathcal{N}(T)$ is bounded by $\mathcal{O}(h(T))$, with the following operations that work in constant time:*

$\triangle : V_0 \to \mathcal{N}(T)$*: Go to a root node.*

$z : \mathcal{N}(T) \to \Sigma$*: Get the character at the current node.*

$\uparrow : \mathcal{N}(T) \to \mathcal{N}(T)_\perp$*: Go to the parent node.*

$\downarrow_i : \mathcal{N}(T) \to \mathcal{N}(T)_\perp$*: Go to the i'th child, where* $1 \le i \le \max\{r(a) \mid a \in \Sigma\}$.

An element $X \in \mathcal{N}(T)$ can be thought of as describing a path into the tree $[\![T]\!]$, which is a sequence of natural numbers, where each one dictates into which child we went. Again, this sequence is not actually stored in X.

To implement the navigation on T we reuse the navigation on SSLPs, namely on the spine SSLP of T. Each spine starts with a (possibly empty) sequence of symbols of the form $a(A_1, \ldots, A_{j-1}, x, A_{j+1}, \ldots, A_{r(a)})$ and ends on a symbol a. When we are on a symbol of the form

$$a(A_1, \ldots, A_{j-1}, x, A_{j+1}, \ldots, A_{r(a)})$$

we can enter $r(a)$ many different children. In case we enter the j'th child, we move the spine traversal one to the right. In case we enter a different child j' (so $j' \neq j$), where $1 \leq j' \leq r(a)$, then we start a new spine traversal at $A_{j'}$ instead. As our navigation structure we use $\mathcal{N}(T) = \mathcal{N}(T_{\square})^+$ and as our precomputed data structure we use $P(T_{\square})$, which is in $\mathcal{O}(|T|)$. The individual operations are implemented as follows: For the root node, we set $\triangle(A) = \triangleleft(A)$. Let $w \in \mathcal{N}(T_{\square})^*$ and $X \in \mathcal{N}(T_{\square})$. To go to the parent, we set:

$$\uparrow(wX) = \begin{cases} w \leftarrow(X) & \text{if } \leftarrow(X) \neq \bot, \\ w & \text{if } \leftarrow(X) = \bot \text{ and } w \neq \varepsilon, \\ \bot & \text{if } \leftarrow(X) = \bot \text{ and } w = \varepsilon. \end{cases}$$

For the current character we set $z(wX) = a$ if $z(X) = a_j(A_1, \ldots, A_{r(a)-1})$ or if $z(X) = a$. Going to the i'th child is defined as follows:

$$\downarrow_i(wX) = \begin{cases} w \rightarrow(X) & \text{if } z(X) = a(A_1, \ldots, A_{j-1}, x, A_{j+1}, \ldots A_{r(a)}) \\ & \text{and } j = i, \\ w \triangleleft(A_i) & \text{if } z(X) = a(A_1, \ldots, A_{j-1}, x, A_{j+1}, \ldots A_{r(a)}), \\ & j \neq i \text{ and } i \leq r(a), \\ \bot & \text{if } z(X) = a(A_1, \ldots, A_{j-1}, x, A_{j+1}, \ldots A_{r(a)}) \\ & \text{and } i > r(A) \text{ or } z(X) = a. \end{cases}$$

4.3 NAVIGATING FSLPS

Theorem 3. *Let $F = (V, \Gamma, \rho)$ be an FSLP in normal form. We precompute in time $\mathcal{O}(|F|)$ some data structure $P(F)$, and we define a set of states $\mathcal{N}(F)$, such that the size of each element from $\mathcal{N}(F)$ is bounded by $\mathcal{O}(h(F))$, with the following operations that work in constant time:*

$\triangle_{\triangleleft}: V_0 \to \mathcal{N}(F)_\bot$*: Go to the root node of the first tree of a forest.*

$\triangle_{\triangleright}: V_0 \to \mathcal{N}(F)_\bot$*: Go to the root node of the last tree of a forest.*

$z: \mathcal{N}(F) \to \Sigma$*: Get the character at the current node.*

$\swarrow: \mathcal{N}(F) \to \mathcal{N}(F)_\bot$*: Go to the root node of the first child.*

$\searrow: \mathcal{N}(F) \to \mathcal{N}(F)_\bot$*: Go to the root node of the last child.*

$\uparrow: \mathcal{N}(F) \to \mathcal{N}(F)_\bot$*: Go to the parent node.*

$\rightarrow: \mathcal{N}(F) \to \mathcal{N}(F)_\bot$*: Go to the root node of the right sibling.*

$\leftarrow: \mathcal{N}(F) \to \mathcal{N}(F)_\bot$*: Go to the root node of the left sibling.*

The idea to implement this is as follows: We are going to interleave spine navigations with rib navigations. Each spine navigation starts with an expression of the form $\rho(A) = B \circledcirc (a(x) \circledcirc C)$. The navigation of $[\![A]\!]_F$ starts on $[\![B]\!]_{F_{\boxplus}}$ (which can be empty). The symbols appearing in this navigation are of the form $a(x) \boxplus (L \circledcirc x \circledcirc R)$. If we reach the parameter x in $[\![B]\!]_F$, we are actually on $[\![a(x) \circledcirc C]\!]_F$. Rib navigations can start on the above variables L, R and C. The symbols appearing in them are all of the form $B \circledcirc (a(x) \circledcirc C)$, which again start spine navigations. The "current node" will always be represented by a symbol of the form $a(x) \boxplus (L \circledcirc x \circledcirc R)$ or $a(x) \circledcirc C$. For this, we need a slight addition to the spine SSLP, since it does not contain the $a(x) \circledcirc C$ parts yet: Let $F_{\boxplus} = (V_1, \rho_{\boxplus})$. We define $F_{\boxplus}' = (V', \rho_{\boxplus}')$ over Σ' as follows:

- $\Sigma' = \Sigma_{\boxplus} \cup \{a(x) \circledcirc C \mid a \in \Sigma, C \in V_0\}$,
- $V' = V_1 \cup \{B \circledcirc (a(x) \circledcirc C) \mid A \in V_0, \rho(A) = B \circledcirc (a(x) \circledcirc C)\}$,
- $\rho_{\boxplus}'(A) = \rho_{\boxplus}(A)$ for all $A \in V_1$ and
- $\rho_{\boxplus}'(B \circledcirc (a(x) \circledcirc C)) = B \circ (a(x) \circledcirc C)$ for all $B \circledcirc (a(x) \circledcirc C) \in V' \setminus V_1$.

Therefore, the spine words we navigate on are of the form

$$(a_1(x) \boxplus (L_1 \circledcirc x \circledcirc R_1)) \dots (a_n(x) \boxplus (L_n \circledcirc x \circledcirc R_n))(a(x) \circledcirc C),$$

where $n \geq 0$. As our precomputed data structure we will use the pair $(P(F_{\boxplus}'), P(F_{\boxminus}))$ and $\mathcal{N}(F) = (\{\ell, m, r\} \times \mathcal{N}(F_{\boxminus}) \times \mathcal{N}(F_{\boxplus}'))^+$ as our navigation structure. The first component of each element tells us from where we entered the rib navigation: If we are coming from a symbol of the form $a(x) \boxplus (A \circledcirc x \circledcirc A)$ it is important to remember if we entered the left A or the right A. The m-part is used for symbols of the form $a(x) \circledcirc C$. When we start a new navigation on the first tree of a forest, we always have to start a rib navigation first and then navigate to the first element of the spine that corresponds to the first tree of the rib navigation. To ease the notation, we define a short-hand operator $\triangleleft : \{\ell, m, r\} \times V_0 \to \mathcal{N}(F)_\perp$ with

$$\triangleleft(d, A) = \begin{cases} (d, \triangleleft(A), \triangleleft(z(\triangleleft(A)))) & \text{if } \triangleleft(A) \neq \perp, \\ \perp & \text{if } \triangleleft(A) = \perp. \end{cases}$$

The individual operations are implemented as follows: Going to the root of the first tree is defined as $\triangle_{\triangleleft}(A) = \triangleleft(m, A)$. Going to the root of the last tree is similar. For the current character we set $z(w(d, Y, X)) = a$ if $z(X) = a(x) \boxplus (L \circledcirc x \circledcirc R)$ or $z(X) = a(x) \circledcirc C$.

Going to the first child works as follows: In case the current character is of the form $a(x) \circledcirc (L \circledcirc x \circledcirc R)$ we enter the first tree of $[\![L]\!]_F$ if it exists. If not we enter the x on the spine and therefore have to move the spine navigation one position to the right. This is always possible, since spines always end in

symbols of the form $a(x) \oslash C$. The other case is that the current character is of the form $a(x) \oslash C$. Here, we have to enter the first tree of $[\![C]\!]_F$ if it exists.

$$\swarrow(w(d, Y, X)) = \begin{cases} w(d, Y, X) \lhd (\ell, L) & \text{if } z(X) = a(x) \boxslash (L \oslash x \oslash R) \\ & \text{and } \lhd(L) \neq \bot, \\ w(d, Y, \rightarrow(X)) & \text{if } z(X) = a(x) \boxslash (L \oslash x \oslash R) \\ & \text{and } \lhd(L) = \bot, \\ w \lhd (m, C) & \text{if } z(X) = a(x) \oslash C \\ & \text{and } \lhd(C) \neq \bot, \\ \bot & \text{if } z(X) = a(x) \oslash C \\ & \text{and } \lhd(C) = \bot. \end{cases}$$

Going to the last child works in a similar way. Instead of entering the first tree of $[\![L]\!]_F$ in the first case, we enter the last tree of $[\![R]\!]_F$. In the second case we enter the last tree of $[\![C]\!]_F$.

Going to the parent node is straightforward: We move the current spine navigation one to the left if possible. If not, we remove the latest spine and tree navigation.

$$\uparrow(w(d, Y, X)) = \begin{cases} w(d, Y, \leftarrow(X)) & \text{if } \leftarrow(X) \neq \bot, \\ w & \text{if } \leftarrow(X) = \bot \text{ and } w \neq \varepsilon, \\ \bot & \text{if } \leftarrow(X) = \bot \text{ and } w = \varepsilon. \end{cases}$$

The most difficult case is going to the right neighbor. Whether we are at a node of the form $a(x) \boxslash (L \oslash x \oslash R)$ or $a(x) \oslash C$ does not matter, since we have to look at the node above. If we can move the spine position one to the left, say to $a'(x) \boxslash (L' \oslash x \oslash R')$, the right neighbor is the first tree of $[\![R']\!]_F$, if it exists. If not, there is no right neighbor. In case we are already at the first symbol of the spine we try to move the previous rib navigation one to the right. If this is not possible we look at the previous spine navigation if it exists, and then we are in any of the two situations: If the current symbol of the previous spine navigation is of the form $a(x) \oslash C$ then we have no right neighbor. If it is of the form $a(x) \boxslash (L \oslash x \oslash R)$ then there are two more cases. If we were navigating R then there is also no right neighbor. If we were navigating L then we reach the x on the spine and move the current spine navigation one to the right (which again is always defined).

$$\rightarrow(w(d, Y, X)) = \begin{cases} w(d, Y, \leftarrow(X)) \lhd (r, R) & \text{if } \leftarrow(X) \neq \bot, \\ & z(\leftarrow(X)) = a(x) \boxslash (L \oslash x \oslash R) \\ & \text{and } \lhd(R) \neq \bot, \\ w(d, \rightarrow(Y), \lhd(z(\rightarrow(Y)))) & \text{if } \leftarrow(X) = \bot \\ & \text{and } \rightarrow(Y) \neq \bot, \\ w'(\ell, Y', \rightarrow(X')) & \text{if } \leftarrow(X) = \bot,\ \rightarrow(Y) = \bot, \\ & w = w'(\ell, Y', X') \\ & \text{and } \rightarrow(X') \neq \bot, \\ \bot & \text{if } \leftarrow(X) = \bot,\ \rightarrow(Y) = \bot, \\ & w = w'(d, Y', X') \text{ and } d \neq \ell \\ & \text{or } d = \ell \text{ and } \rightarrow(X') = \bot. \end{cases}$$

Going to the left neighbor is again similar.

The goal of this section is to create a navigation structure with the same operations we used previously (go to the parent or to the i'th child) and to support the following operation in constant time: Given two navigation structures, are the subtrees at their current positions equal? To make this possible, we are going to allow polynomial time preprocessing in $|T|$ instead of linear time preprocessing. Also, like we mentioned earlier, we now allow to compare two numbers which have a size of $\mathcal{O}(\log(|[\![T]\!]|))$ bits.

Formally, $u \in \mathcal{T}(\Sigma)$ is a subtree of $t \in \mathcal{T}(\Sigma)$ if and only if there is a context $v \in \mathcal{T}_x(\Sigma)$ such that $v[u] = t$.

Let $T = (V, \Gamma, \rho)$ be a TSLP in normal form that is also reduced. Assuming this already requires polynomial time preprocessing. Let

$$V_0' = \{A \in V_0 \mid \rho(A) = B \circledcirc C,\ B, C \in V\}.$$

We characterize subtrees of $[\![T]\!]$ using spine SSLPs: Let $A \in V_0'$ where $\rho(A) = B \circledcirc C$ for some $B, C \in V$. The length of the spine of A is defined by $\ell(A) = |[\![B]\!]_{T_\square}|$ and the subtree produced at depth i, $1 \le i \le \ell(A) + 1$, is defined by

$$A_\triangle(i) = [\![A[i] \square \cdots \square A[\ell(A)] \circledcirc C]\!]_T.$$

Especially, $A_\triangle(1) = [\![A]\!]_T$ and $A_\triangle(\ell(A) + 1) = [\![C]\!]_T$. We introduce the following short-hand notations: Let $i, j \in \{1, \ldots, \ell(A)\}$. We write $A[i : j] = [\![A]\!]_{T_\square}[i : j]$, $A[i :] = [\![A]\!]_{T_\square}[i :]$, $A[: i] = [\![A]\!]_{T_\square}[: i]$ and $A[i] = [\![A]\!]_{T_\square}[i]$.

For each subtree t of $[\![A]\!]_T$ there is an index i with $A_\triangle(i) = t$ or there is a variable $D \in V_0$ such that t is a subtree of $[\![D]\!]_T$ and $[\![D]\!]_T$ is a subtree of $[\![A]\!]_T$. Both cases can actually overlap, which we call a *nontrivial occurrence* of a subtree. In this case there is an index $i \ge 1$ *and* a $D \in V_0$ with $A_\triangle(i) = t = [\![D]\!]_T$. The positions 1 and $\ell(A) + 1$ are always nontrivial occurrences since $A_\triangle(1) = [\![A]\!]_T$ and $A_\triangle(\ell(A) + 1) = [\![C]\!]_T$. We are interested in the first nontrivial occurrence below A itself, which may be $\ell(A) + 1$:

$$s(A) = \min\{i \in \{2, \ldots, \ell(A) + 1\} \mid D \in V_0,\ A_\triangle(i) = [\![D]\!]_T\}.$$

We can now reformulate where subtrees occur as follows: For each subtree t of $[\![A]\!]_T$ that is not $[\![A]\!]_T$ itself, there is *either* an index i with $A_\triangle(i) = t$ and $2 \le i < s(A)$ *or* there is a variable $D \in V_0$ such that t is a subtree of $[\![D]\!]_T$ and $[\![D]\!]_T$ is a subtree of $[\![A]\!]_T$. By definition of s, all $2 \le i < s(A)$ are *not* equal to any $[\![D]\!]_T$ with $D \in V_0$. Let $\overline{A}$ be the variable such that $A_\triangle(s(A)) = [\![\overline{A}]\!]_T$ and let $\rho_A = A[s(A) - 1]$.

Example 4. We give an example of a TSLP $T = (V, \Gamma, \rho)$ that has two non trivially equal subtrees in $[\![A]\!]_T$ and $[\![D]\!]_T$, where $A, D \in V$. The variables for the spine of A are:

$$\begin{aligned}
\rho(A) &= B \circledcirc C,\\
\rho(B) &= B_1 \square B_2,\\
\rho(B_1) &= g(x, K),\\
\rho(B_2) &= B_3 \square B_4,\\
\rho(B_3) &= f(I, x),\\
\rho(B_4) &= f(K, x).
\end{aligned}$$

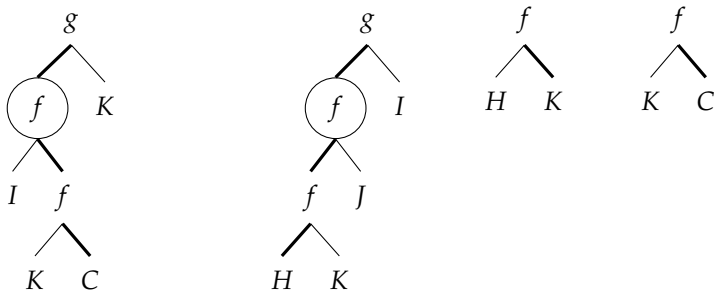


Figure 1: Trees of A, D, I and J of the TSLP T from Example 4. Thick lines represent spine paths. The subtrees at the two nodes marked by circles derive equal trees.

Additionally, the variables for the spine of D are:

$$\begin{aligned}\rho(D) &= E \mathbin{\textcircled{\scriptsize :}} H,\\ \rho(E) &= E_1 \mathbin{\square\!\!\square} E_2,\\ \rho(E_1) &= g(x, I),\\ \rho(E_2) &= E_3 \mathbin{\square\!\!\square} E_4,\\ \rho(E_3) &= f(x, J),\\ \rho(E_4) &= f(x, K).\end{aligned}$$

Finally, the variables for the spines of I and J and some basic variables are added:

$$\begin{aligned}\rho(I) &= L \mathbin{\textcircled{\scriptsize :}} K,\\ \rho(L) &= f(H, x),\\ \rho(J) &= M \mathbin{\textcircled{\scriptsize :}} C,\\ \rho(M) &= f(K, x),\\ \rho(C) &= c,\\ \rho(K) &= k,\\ \rho(H) &= h.\end{aligned}$$

The reader can check that this TSLP is reduced. See Figure 1 for a visualization. For the spine SSLP, we have

$$\begin{aligned}[\![A]\!]_{T_{\square\!\!\square}} &= g(x, K) f(I, x) f(K, x) c \text{ and}\\ [\![D]\!]_{T_{\square\!\!\square}} &= g(x, I) f(x, K) f(x, K) h.\end{aligned}$$

The trees produced by A and D are

$$\begin{aligned}A_\triangle(1) &= [\![B_1 \mathbin{\square\!\!\square} B_3 \mathbin{\square\!\!\square} B_4 \mathbin{\textcircled{\scriptsize :}} C]\!]_T = g\langle f\langle f\langle h, k\rangle, f\langle k, c\rangle\rangle, k\rangle \text{ and}\\ D_\triangle(1) &= [\![E_1 \mathbin{\square\!\!\square} E_3 \mathbin{\square\!\!\square} E_4 \mathbin{\textcircled{\scriptsize :}} H]\!]_T = g\langle f\langle f\langle h, k\rangle, f\langle k, c\rangle\rangle, f\langle h, k\rangle\rangle.\end{aligned}$$

The equal subtrees, marked by circles in Figure 1, are

$$\begin{aligned}A_\triangle(2) &= [\![B_3 \mathbin{\square\!\!\square} B_4 \mathbin{\textcircled{\scriptsize :}} C]\!]_T\\ &= f\langle f\langle h, k\rangle, f\langle k, c\rangle\rangle\\ &= [\![E_3 \mathbin{\square\!\!\square} E_4 \mathbin{\textcircled{\scriptsize :}} H]\!]_T\\ &= D_\triangle(2).\end{aligned}$$

Moreover, $s(A) = 3$, $s(D) = 3$, $\overline{A} = J$, $\overline{D} = I$, $\rho_A = f(I, x)$, and $\rho_D = f(x, J)$.

Lemma 12. *For every variable $A \in V_0$ we can compute $s(A)$, ρ_A and $\overline{A}$ in polynomial time.*

Proof. For $2 \le i \le \ell(A)$ let $T_{A,i}$ be a TSLP with $[\![T_{A,i}]\!] = A_\triangle(i)$. Such a TSLP can be constructed using Proposition 1: Compute an SSLP for $A[i] \circ \ldots \circ A[\ell(A)+1]$ from $T_{\boxplus}$, which can then be converted into an appropriate TSLP for $A_\triangle(i)$.

For every $D \in V_0$, we now test if it is $\overline{A}$ and if so, what $s(A)$ is. Once $s(A)$ is known, ρ_A can be easily computed from $T_{\boxplus}$ using Proposition 1. To test whether there is a position i, where $2 \le i \le \ell(A)$, with $A_\triangle(i) = [\![D]\!]_T$ can be done using binary search on the interval $[2, \ldots, \ell(A)]$: For a given i we can first compute $T_{A,i}$ and then use that to test if $|A_\triangle(i)|$ is less, equal or greater than $|[\![D]\!]_T|$. In case the binary search returns an index i such that $|A_\triangle(i)| = |[\![D]\!]_T|$, we have to test if $[\![B \otimes D]\!]_T = A_\triangle(i)$, which can be implemented using Lemma 4. If this is true, then $\overline{A} = D$ and $s(A) = i$. □

Proposition 3. *Let $D, E \in V_1$, $D \neq E$ with*

$$\rho(D) = f_i(D_1, \ldots, D_n),$$
$$\rho(E) = g_j(E_1, \ldots, E_m),$$

and $t, u \in \mathcal{T}(\Sigma)$. If $[\![D]\!]_T[t] = [\![E]\!]_T[u]$, then there exist $A, B \in V_0$ such that $[\![A]\!]_T = t$ and $[\![B]\!]_T = u$.

Proof. Since T is reduced, we have $[\![D]\!]_T[t] = [\![E]\!]_T[u]$ if and only if $f = g$ (and thus $m = n$), $i \neq j$, $D_k = E_k$ for $k \in \{1, \ldots, m\} \setminus \{i, j\}$, $t = [\![E_i]\!]_T$ and $u = [\![D_j]\!]_T$. Note that if $i = j$, then, since T is reduced, we would obtain $D = E$, which contradicts the assumption. □

Lemma 13. *For all $A, B \in V_0'$ and all $1 \le i < s(A)$, $1 \le j < s(B)$, the following two conditions are equivalent:*

(a) $A_\triangle(i) = B_\triangle(j)$

(b) $A[i : s(A) - 2] = B[j : s(B) - 2]$ *and* $[\![\rho_A \otimes \overline{A}]\!]_T = [\![\rho_B \otimes \overline{B}]\!]_T$.

Proof. We will make use of the following facts, which are easy to prove: Let $A, B \in V_0'$ and $1 \le i < \ell(A)$, $1 \le j < \ell(B)$.

$$A_\triangle(s(A) - 1) = [\![A[s(A) - 1] \otimes \overline{A}]\!]_T = [\![\rho_A \otimes \overline{A}]\!]_T. \tag{1}$$
$$A_\triangle(i) = [\![A[i]]\!]_T[A_\triangle(i+1)]. \tag{2}$$
$$\text{If } A_\triangle(i+1) = B_\triangle(j+1) \text{ and } A[i] = B[j], \text{ then } A_\triangle(i) = B_\triangle(j). \tag{3}$$
$$\text{If } A_\triangle(i) = B_\triangle(j) \text{ and } A[i] = B[j], \text{ then } A_\triangle(i+1) = B_\triangle(j+1). \tag{4}$$

To obtain a from b, note that by equation (1) $[\![\rho_A \otimes \overline{A}]\!]_T = [\![\rho_B \otimes \overline{B}]\!]_T$ implies $A_\triangle(s(A) - 1) = B_\triangle(s(B) - 1)$. Repeated application of equation (3) implies $A_\triangle(i) = B_\triangle(j)$. Now assume that a holds, so $A_\triangle(i) = B_\triangle(j)$. By induction on i and j, we show that $A[i : s(A) - 2] = B[j : s(B) - 2]$ and $[\![\rho_A \otimes \overline{A}]\!]_T = [\![\rho_B \otimes \overline{B}]\!]_T$.

- Let $i = s(A) - 1$. Then equation (1) becomes

  $$A_\triangle(i) = [\![\rho_A \otimes \overline{A}]\!]_T = [\![A[s(A) - 1] \otimes \overline{A}]\!]_T.$$

  By equation (2) we have $B_\triangle(j) = [\![B[j]]\!]_T[B_\triangle(j+1)]$. Therefore, we obtain

  $$[\![A[s(A) - 1] \otimes \overline{A}]\!]_T = [\![B[j]]\!]_T[B_\triangle(j+1)].$$

Now there are two cases: either $A[s(A)-1] = B[j]$ in which case $[\![\overline{A}]\!]_T = B_\triangle(j+1)$, or $A[s(A)-1] \neq B[j]$ in which case we obtain from Proposition 3 that there is a variable in V_0 that expands to $B_\triangle(j+1)$. In both cases, there is a variable in V_0 that expands to $B_\triangle(j+1)$. Since $j+1 \leq s(B)$, we must have $j+1 = s(B)$, $B_\triangle(j+1) = [\![\overline{B}]\!]_T$, and $\rho_B = B[j]$. Therefore we have

$$A[i : s(A)-2] = \varepsilon = B[j : s(B)-2]$$

and $[\![\rho_A \circledcirc \overline{A}]\!]_T = [\![\rho_B \circledcirc \overline{B}]\!]_T$. The latter implies $[\![\rho_A \circledcirc A']\!]_T = [\![\rho_B \circledcirc B']\!]_T$ since T is reduced.

- The case $j = s(B)-1$ is symmetric to the previous case.
- Let $i < s(A)-1$ and $j < s(B)-1$. We claim that $A[i] = B[j]$. Assume that $A[i] \neq B[j]$. From $A_\triangle(i) = B_\triangle(j)$ we obtain

  $$[\![A[i]]\!]_T[A_\triangle(i+1)] = [\![B[j]]\!]_T[B_\triangle(j+1)]$$

  by equation (2). Proposition 3 implies that there are variables in V_0 that expand to $A_\triangle(i+1)$ and $B_\triangle(j+1)$, respectively. This contradicts $i+1 < s(A)$ as well as $j+1 < s(B)$. Hence we have $A[i] = B[j]$. Because $A_\triangle(i) = B_\triangle(j)$ it follows from equation (4) that $A_\triangle(i+1) = B_\triangle(j+1)$. We can now conclude with induction.

□

To support subtree equality checks during navigation, we modify the old navigation implementation so that we always stay above $s(A)$. To achieve this, we count the number of steps we went down into each spine. If we reach $s(A)$ we instead go into the spine of $\overline{A}$, since $[\![\overline{A}]\!]_T$ is equal to $A_\triangle(s(A))$. This modification then allows us to use Lemma 13 for equality checks: Instead of testing Point a from the Lemma, we test Point b. We will later argue why we can implement this efficiently.

The data structure used to navigate our TSLP with support for subtree equality checks is $\mathcal{N}(T) \times (\mathbb{N}^+)$. In the second component, $\mathbb{N}^+$, we record a sequence of numbers that tell us how deep into each of the spines we went. The length of this sequence should be the same as the length of the first component, so the navigation structure we will use is

$$\mathcal{N}(T)' = \{(w, v) \in \mathcal{N}(T) \times (\mathbb{N}^+) \mid |w| = |v|\}.$$

To implement the new operations, we basically reuse the old operations, expect when we enter a spine split, in which case we start a new traversal. To count the number of steps we went into each spine, we have to know when the old operations enter a new one. This can easily be checked by comparing the length $|w|$ of $w \in \mathcal{N}(T)$ and the length of $\downarrow_i(w)$. Comparing these lengths means comparing two $O(|T|)$ bit numbers, which we use constant time for. Keeping track of these lengths during the traversal is also easily done in constant time and will be omitted here.

To query the start symbol of the latest spine traversal, let $\mathrm{st}\colon \mathcal{N}(T) \to V_0$ be defined as $\mathrm{st}(X_1 \dots X_n) = \mathrm{st}(X_n)$ (where $n \geq 1$). To go to the root of $A \in V_0$ we define $\triangle(A) = (\triangle(A), 1)$, reusing the old operation $\triangle$. In the following definitions let $w \in \mathcal{N}(T)$, $v \in \mathbb{N}^*$ and $j \in \mathbb{N}$. To go to the parent, we reuse the old $\uparrow$-operation in the first component. In addition to that, we have to

remove the latest spine counter in case it is 1. Alternatively, we could have tested if $|\uparrow(w)|$ shrinks.

$$\uparrow(w, vi) = \begin{cases} (\uparrow(w), v) & \text{if } \uparrow(w) \neq \bot \text{ and } i = 1, \\ (\uparrow(w), v(i-1)) & \text{if } \uparrow(w) \neq \bot \text{ and } i > 1, \\ \bot & \text{if } \uparrow(w) = \bot. \end{cases}$$

Going to the i'th child is defined as follows:

$$\downarrow_i(w, vj) = \begin{cases} (\downarrow_i(w), v(j+1)) & \text{if } \downarrow_i(w) \neq \bot,\ |\downarrow_i(w)| = |w| \\ & \text{and } s(\mathrm{st}(w)) < j+1, \\ (w \lhd (\overline{\mathrm{st}(w)}), vj1) & \text{if } \downarrow_i(w) \neq \bot,\ |\downarrow_i(w)| = |w| \\ & \text{and } s(\mathrm{st}(w)) = j+1, \\ (\downarrow_i(w), vj1) & \text{if } \downarrow_i(w) \neq \bot \text{ and } |\downarrow_i(w)| > |w|, \\ \bot & \text{if } \downarrow_i(w) = \bot. \end{cases}$$

In the first case, we stay on the spine, since $|\downarrow_i(w)| = |w|$ and we are still above the split position (meaning higher in the tree). In the second case, we arrive at the split position. Here, the original operation stays on the traversal of the current spine, but we instead start a new traversal that goes into the split. In the third case the original operation starts a new spine traversal, so we do this as well. Note that $s(A) \geq 2$ for every $A \in V_0$ because T is reduced. Also note that we never go beyond spine splits.

Given two navigation structures $(w, vj), (w', v'j') \in \mathcal{N}(T)'$, the goal is now to test if $\mathrm{st}(w)_\triangle(j) = \mathrm{st}(w')_\triangle(j')$. Let $\mathrm{st}(w) = A$ and $\mathrm{st}(w') = B$. If $A, B \notin V_0'$, so $\rho(A) = a$ and $\rho(B) = b$ for some $a, b \in \Sigma$, we simply have to test if $a = b$ (which is true if and only if $A = B$ since T is reduced). If $A \notin V_0'$ and $B \in V_0$ (or vice versa) then the result is false. Now let $A, B \in V_0'$. Instead of testing Point a from Lemma 13, we test Point b. The first test is if $[\![\rho_A \circledcirc \overline{A}]\!]_T = [\![\rho_B \circledcirc \overline{B}]\!]_T$. Let $\rho_A = a_i(A_1, \ldots, A_n)$ and $\rho_B = b_j(B_1, \ldots, B_m)$. By Proposition 3, since T is reduced this is true if and only if $a = b$ and

$$A_1, \ldots, A_{i-1}, \overline{A}, A_{i+1}, \ldots, A_n = B_1, \ldots, B_{j-1}, \overline{B}, B_{j+1}, \ldots, B_m.$$

This is a simple comparison of two strings from V_0^* of constant length. The second test, $A[i : s(A) - 2] = B[j : s(B) - 2]$, can be reformulated as follows: Is there a $k \in \mathbb{N}$ such that $i + k = s(A) - 2$, $j + k = s(B) - 2$ and

$$A[-k : s(A) - 2] = B[-k : s(B) - 2]?$$

This is equivalent to testing if

- $j - i = s(A) - s(B)$
- and if so, do $A[: s(A) - 2]$ and $B[: s(B) - 2]$ have a common suffix of at least length $s(A) - 2 - i$ (resp. $s(B) - 2 - j$)?

Definition 38. Let Σ be an alphabet and $w, v \in \Sigma^*$. The *length of the longest common suffix* of w and v, written as $\mathrm{lcs}(w, v)$, is the largest number $s \in \mathbb{N}$ such that $w[-s :] = v[-s :]$.

Lemma 14. *Let $u, w, v \in \Sigma^*$.*

1. *If $\mathrm{lcs}(u, v) \geq \mathrm{lcs}(u, w)$ then $\mathrm{lcs}(u, w) = \mathrm{lcs}(v, w)$.*
2. *If $\mathrm{lcs}(u, v) \leq \mathrm{lcs}(u, w)$ then $\mathrm{lcs}(v, w) = \mathrm{lcs}(u, v)$.*

lcs	A	B	C	D
B	3			
C	0	0		
D	0	0	2	
E	0	0	3	3

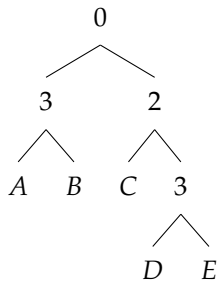


Figure 2: An example for storing lcs in a tree of linear size.

Proof. We only prove the first point, since the proof of the second one is almost the same. Let $\mathrm{lcs}(u,v) \geq \mathrm{lcs}(u,w)$, so i and j are the largest numbers such that $u[-i:] = v[-i:]$ and $u[-j:] = w[-j:]$ with $i \geq j$. It follows that $v[-j:] = u[-j:] = w[-j:]$ and we have to show that $v[-(j+1):] \neq w[-(j+1):]$ which implies that $\mathrm{lcs}(v,w) = j$. In case $i = j$ this is clear, since $v[-(j+1):] = w[-(j+1):]$ would imply $u[-(j+1):] = v[-(j+1)]$ as well as $u[-(j+1):] = w[-(j+1)]$, which contradicts both $u[-(j+1)] \neq v[-(j+1)]$ and $u[-(j+1)] \neq w[-(j+1)]$. Now assume that $i > j$ and that $v[-(j+1):] = w[-(j+1):]$. We have $u[-(j+1):] = v[-(j+1)]$, since $\mathrm{lcs}(u,v)$ is at least one longer than $\mathrm{lcs}(u,w)$. Therefore, we obtain that $u[-(j+1)] = w[-(j+1)]$. This is a contradiction, since $u[-(j+1)] \neq w[-(j+1)]$. □

The longest common suffix of the spines of two variables is defined as $\mathrm{lcs}: V_0' \times V_0' \to \mathbb{N}$ with

$$\mathrm{lcs}(A,B) = \mathrm{lcs}(A[:s(A)-2], B[:s(B)-2]).$$

We can precompute $\mathrm{lcs}(A,B)$ for all $A,B \in V_0'$, using Proposition 1, which takes polynomial time. The difficult part is to store these $|V_0'|^2$ pairs in space $\mathcal{O}(|V_0'|)$. To achieve this, we construct a tree of size $\Theta(|V_0'|)$ with the following properties:

- Leaf nodes are labelled with elements from V_0'. Each $A \in V_0'$ is mapped to a unique leaf node, which for simplicity we will also call A.
- Inner nodes are labelled with numbers from $\mathbb{N}$. With $\mathrm{lca}(x,y)$ we denote the lowest common ancestor of two nodes x,y, and with $\ell(x)$ the label of node x. Then for every $A,B \in V_0'$ it will hold that

$$\ell(\mathrm{lca}(A,B)) = \mathrm{lcs}(A,B).$$

See Figure 2 for an illustration. The construction of the tree works as follows: Let $V_0' = \{A_1,\ldots,A_m\}$. If $m = 0$ then the resulting tree is empty. Let $m \neq 0$. We start with a tree that has a root node labelled with 0, which has a single child node labelled with A_1. We now iterate over V_0 from $i = 2$ to m and do the following:

- Assume that we have already constructed the tree for $\{A_1,\ldots,A_{i-1}\}$.
- Take an $A \in \{A_1,\ldots,A_{i-1}\}$ such that

$$\mathrm{lcs}(A,A_i) = \max\{\mathrm{lcs}(A_i,A_j) \mid 1 \leq j < i\}.$$

  Since we have $i \geq 2$, this is well-defined.

- Let X be the leaf-node labelled with A.
- As long as the parent of X is labelled with a number that is greater than $\mathrm{lcs}(A, A_i)$, set X to the parent of X. Now the parent node P of X is labelled with a number $n \leq \mathrm{lcs}(A, A_i)$. This node exists since the root node is labelled with 0.
- In case $n = \mathrm{lcs}(A, A_i)$, add a new child node labelled with A_i to P.
- In case $n < \mathrm{lcs}(A, A_i)$, remove X as child node from P. Add a new node Y to P labelled with $\mathrm{lcs}(A, A_i)$. Add X and a new node labelled with A_i as children to Y.

For each element from V_0' we add at most two nodes. Every internal node is labelled with a $\log(|[\![T]\!]|)$-bit number. Hence, on the word RAM model we can store the tree in space $\mathcal{O}(|T|)$. Finally, we obtain $\mathrm{lcs}(A, B)$ for $A, B \in V_0$ by computing the lowest common ancestor of the two leaf nodes labelled with A and B. Using a data structure for computing lowest common ancestors in time $\mathcal{O}(1)$ [3, 45] we obtain an $\mathcal{O}(1)$-time implementation of subtree equality checking.

We now prove that the constructing of the tree is correct. For nodes X and Y we use $\mathrm{lca}(X, Y)$ to denote the lowest common ancestor of X and Y. We also use $\ell(X)$ to denote the label of node X. We need to show that in the i'th step of the algorithm it holds for every $A \neq B \in \{A_1, \ldots, A_i\}$ that $\ell(\mathrm{lca}(A, B)) = \mathrm{lcs}(A, B)$. For $i = 1$ this is clear. Let $i > 1$. In the first case, we found a node labelled with $\mathrm{lcs}(A, A_i)$ and added A_i to its children. In the second case, we inserted a new node labelled with $\mathrm{lcs}(A, A_i)$ and added A_i to its children. In both cases, we must show that the alterations made to the tree do not change $\ell(\mathrm{lca}(B, C))$ for all $B, C \in \{A_1, \ldots, A_{i-1}\}$ and that $\ell(\mathrm{lca}(B, A_i)) = \mathrm{lcs}(B, A_i)$ for all $B \in \{A_1, \ldots, A_{i-1}\}$. The only alterations to existing nodes that we make is in the last case in which we remove X as the child of P, add a new node Y, make it the child of P and add X as the child of Y. This does not change lca for any already existing nodes. To show that lca for the new variable is correct, we distinguish two cases:

- Let $\mathrm{lca}(\mathrm{lca}(A, B), A_i) = \mathrm{lca}(A, A_i) = \mathrm{lca}(B, A_i)$. Since we can assume that $\ell(\mathrm{lca}(A, B)) = \mathrm{lcs}(A, B)$, it must be that $\mathrm{lcs}(A, A_i) \leq \mathrm{lcs}(A, B)$, otherwise we would have stopped below $\mathrm{lca}(A, B)$. From Point 1 of Lemma 14 we obtain that $\mathrm{lcs}(A, A_i) = \mathrm{lcs}(B, A_i)$. Altogether, we obtain
$$\ell(\mathrm{lca}(B, A_i))) = \ell(\mathrm{lca}(A, A_i)) = \mathrm{lcs}(A, A_i) = \mathrm{lcs}(B, A_i).$$
- Let $\mathrm{lca}(\mathrm{lca}(A, A_i), B) = \mathrm{lca}(A, B) = \mathrm{lca}(A_i, B)$. Because A has the longest common suffix with A_i, we know that $\mathrm{lcs}(A, A_i) \geq \mathrm{lcs}(A_i, B)$. From Point 2 of Lemma 14 we obtain that $\mathrm{lcs}(A, B) = \mathrm{lcs}(A_i, B)$ and therefore
$$\ell(\mathrm{lca}(A_i, B)) = \ell(\mathrm{lca}(A, B)) = \mathrm{lcs}(A, B) = \mathrm{lcs}(A_i, B).$$

4.5 SUBTREE EQUALITY CHECK FOR FORESTS

The construction for subtree equality checking in FSLPs will work similarly to the one for TSLPs. Again, we allow polynomial time preprocessing in $|F|$ and we allow to compare two numbers of $\mathcal{O}(\log(|[\![F]\!]|))$ bits.

The first difference to the construction for TSLPs is the following: Remember that for TSLPs we defined spine splits for $A \in V_0'$ as the highest

index $i \geq 2$ such that $A_\triangle(i) = D$ for some $D \in V$. Since $\rho(A) = B \circledcirc C$ for some $B, C \in V_0$, this position is always well-defined because $A_\triangle(\ell(A)+1) = [\![C]\!]_T$. In case of FSLPs, this is slightly different: The spines start with variables of the form $B \circledcirc (a(x) \circledcirc C)$, but we do not necessarily have a variable that produces $[\![a(x) \circledcirc C]\!]_F$. We can however add one easily. Let $F = (V, \Gamma, \rho)$ be an FSLP in normal form that is reduced. We add a new variable E with $\rho(E) = x$ and for every $A \in V$ with $\rho(A) = B \circledcirc (a(x) \circledcirc C)$ we add a new variable $A_\perp$ with $\rho(A_\perp) = E \circledcirc (a(x) \circledcirc C)$, but only if there is no variable $B \in V$ with $[\![B]\!]_F = [\![a(x) \circledcirc C]\!]_F$ already. This preprocessing can be implemented in polynomial time. The resulting FSLP, which we still call F, is in normal form and reduced. The next part that works differently from TSLPs is Proposition 3, which we will reformulate:

Proposition 4. *Let $D, D' \in V_1$, $D \neq D'$ with*

$$\rho(D) = a(L \circledcirc x \circledcirc R),$$
$$\rho(D') = a'(L' \circledcirc x \circledcirc R'),$$

and $t, u \in \mathcal{T}(\Sigma)$. If $[\![D]\!]_T[t] = [\![D']\!]_T[u]$, then there exist $A, B \in V_0$ such that $[\![A]\!]_F = t$ and $[\![B]\!]_F = u$.

Proof. Recall the definition of the rib SLP. Let $[\![L_\boxminus]\!] = \alpha_1 \dots \alpha_\ell$, $[\![R_\boxminus]\!] = \beta_1 \dots \beta_r$, $[\![L'_\boxminus]\!] = \alpha'_1 \dots \alpha'_{\ell'}$ and $[\![R'_\boxminus]\!] = \beta'_1 \dots \beta'_{r'}$. Therefore, we must have variables

$$\{L_1, \dots, L_\ell, R_1, \dots, R_r, L'_1, \dots, L'_{\ell'}, R'_1, \dots, R'_{r'}\} \subseteq V_0$$

with $\rho(L_i) = \alpha_i$ for every $1 \leq i \leq \ell$, $\rho(R_i) = \beta_i$ for every $1 \leq i \leq r$, $\rho(L'_i) = \alpha'_i$ for every $1 \leq i \leq \ell'$ and $\rho(R'_i) = \beta'_i$ for every $1 \leq i \leq r'$. Since $[\![D]\!]_T[t] = [\![D']\!]_T[u]$, we have $a = a'$ and

$$[\![L_1 \boxminus \dots \boxminus L_\ell]\!]_F t [\![R_1 \boxminus \dots \boxminus R_r]\!]_F = [\![L'_1 \boxminus \dots \boxminus L'_{\ell'}]\!]_F u [\![R'_1 \boxminus \dots \boxminus R'_{r'}]\!]_F.$$

Since F is reduced, we must have $\ell \neq \ell'$ (and also $r \neq r'$). Otherwise, $L_i = L'_i$ for all $1 \leq i \leq \ell$ and $R_i = R'_i$ for all $1 \leq i \leq r$, so $L = L'$, $R = R'$ and therefore $D = D'$ which contradicts that $D \neq D'$. Assume that $\ell < \ell'$. Then we must have $t = [\![L_{\ell+1}]\!]_F$ and $u = [\![R'_{\ell'-(\ell+1)}]\!]_F$. The case in which $\ell > \ell'$ is similar. □

Similar to TSLPs we define V'_0 as

$$V'_0 = \{A \in V_0 \mid \rho(A) = B \circledcirc (a(x) \circledcirc C),\ a \in \Sigma,\ B, C \in V\}.$$

Let $A \in V'_0$ and $\rho(A) = B \circledcirc (a(x) \circledcirc C)$. We define $\ell(A) = |[\![A]\!]_{F_\boxplus}|$ as the length of the spine, which is 0 in case $[\![B]\!]_F = x$. The i'th element of the spine is again defined as $A[i] = [\![A]\!]_{F_\boxplus}[i]$, where $1 \leq i \leq \ell(A)$. The subtree produced by the spine path of A at depth i, $1 \leq i \leq \ell(A)+1$, is defined by

$$A_\triangle(i) = [\![A[i] \boxplus \dots \boxplus A[\ell(A)] \circledcirc (a(x) \circledcirc C)]\!]_F.$$

Especially, $A_\triangle(1) = [\![A]\!]_F$ and $A_\triangle(\ell(A)+1) = [\![a(x) \circledcirc C]\!]_F$. The spine split is defined in the same way like it was for TSLPs:

$$s(A) = \min\{i \in \{2, \dots, \ell(A)+1\} \mid D \in V_0,\ A_\triangle(i) = [\![D]\!]_F\}.$$

Also let $\rho_A = A[s(A)-1]$ and $\overline{A} = D$ if and only if $[\![D]\!]_F = A_\triangle(s(A))$ for $D \in V_0$. This is always well-defined since we made sure that there is a variable that produces $[\![a(x) \circledcirc C]\!]_F$ and because F is reduced. Lemma 13 can be directly translated to FSLPs:

Lemma 15. *For all $A, B \in V_0'$ and all $1 \le i < s(A)$, $1 \le j < s(B)$, the following two conditions are equivalent:*

(a) $A_\triangle(i) = B_\triangle(j)$

(b) $A[i : s(A) - 2] = B[j : s(B) - 2]$ *and* $[\![\rho_A \circledcirc \overline{A}]\!]_F = [\![\rho_B \circledcirc \overline{B}]\!]_F$.

The proof of this lemma is similar to the one for TSLPs. Again, we would like to use this lemma for equality testing. However, testing the second condition cannot be done in constant time directly: Let $\rho_A = a(x) \boxplus (L_A \circledcirc x \circledcirc R_A)$ and $\rho_B = b(x) \boxplus (L_B \circledcirc x \circledcirc R_B)$. We have to test if $a = b$ (which is again easy) but also if

$$[\![L_A]\!]_{F_\boxplus} \overline{A} [\![R_A]\!]_{F_\boxplus} = [\![L_B]\!]_{F_\boxplus} \overline{B} [\![R_B]\!]_{F_\boxplus},$$

which is a comparison of two strings of exponential length. These comparisons can be implemented using equality checking on SSLPs using Lemma 1 and we can carry all of them out in polynomial time. Let $R \subseteq V_0' \times V_0'$ be the relation such that $(A, B) \in R$ if and only if the previous statement is true. Notice that R is an equivalence relation, and therefore we can store it in linear space as follows: For each A we compute its equivalence class, assign each equivalence class a unique number and save an array of size $|V_0'|$ that maps each variable to its equivalence class. This way, for two variables $A, B \in V_0'$ we can quickly check if they belong to the same equivalence class.

Next, we need a new navigation structure. The idea is again to record for every spine traversal how deep into the spine we went. We also start a new spine traversal in case the spine split is reached. For trees, this required a rather simple change, since the navigation structure was a sequence of spine traversals. In case of forests, this is slightly different. Consider the following example: Let our current traversal be $w(d, Y, X)(\ell, Y', X')$, where the current character of X is $a(x) \boxplus (L \circledcirc x \circledcirc R)$ and the traversal Y' is on the last tree of $[\![L]\!]_F$. If we now use $\rightarrow$, we end up on the parameter x, which means we have to remove (ℓ, Y', X') and move one to the right on X. Suppose that this is the spine split position, so there is $\overline{A} \in V_0$ that evaluates to the tree we want to navigate to. Instead of moving X one to the right, we start a new rib traversal on $\overline{A}$ using m as the direction, so we obtain $w(d, Y, X) \triangleleft (m, \overline{A})$. The navigation structure is $\mathcal{N}(F)' = \{(w, v) \in \mathcal{N}(F) \times (\mathbb{N}^+) \mid |w| = |v|\}$.

The operations are implemented as follows: To query the start symbol of the latest spine traversal, let $\mathrm{st}\colon \mathcal{N}(F) \to V_0$ be defined as

$$\mathrm{st}((d_1, Y_1, X_1) \dots (d_n, Y_n, X_n)) = \mathrm{st}(X_n),$$

where $n \ge 1$. Going to the first tree works similar to $\triangle$ from the tree navigation:

$$\triangleleft(A) = \begin{cases} (\triangleleft(A), 1) & \text{if } \triangleleft(A) \neq \bot, \\ \bot & \text{if } \triangleleft(A) = \bot. \end{cases}$$

Going to the last tree is defined in an analogous way. Going to the parent also works similarly to $\uparrow$ from the tree navigation:

$$\uparrow(w, vi) = \begin{cases} (\uparrow(w), v) & \text{if } \uparrow(w) \neq \bot \text{ and } i = 1, \\ (\uparrow(w), v(i-1)) & \text{if } \uparrow(w) \neq \bot \text{ and } i > 1, \\ \bot & \text{if } \uparrow(w) = \bot. \end{cases}$$

Going to the first child works similarly to $\downarrow_i$ from the tree navigation: In case we reach the spine split we start a new navigation. Otherwise, we simply follow along the original navigation.

$$\swarrow(w, vj) = \begin{cases} (w \lhd (m, \overline{\mathrm{st}(w)}), vj1) & \text{if } \swarrow(w) \neq \bot, |\swarrow(w)| = |w| \\ & \text{and } s(\mathrm{st}(w)) = j+1, \\ (\swarrow(w), v(j+1)) & \text{if } \swarrow(w) \neq \bot, |\swarrow(w)| = |w| \\ & \text{and } s(\mathrm{st}(w)) \neq j+1, \\ (\swarrow(w), vj1) & \text{if } \swarrow(w) \neq \bot \text{ and } |\swarrow(w)| = |w|+1, \\ \bot & \text{if } \swarrow(w) = \bot. \end{cases}$$

Going to the last child is similar. When going to the right neighbor (going to the left neighbor is again similar), we have three cases to consider:

- The length of the original navigation stays the same. In this case we left a spine navigation and entered a new one, so we remove the previous index and add a new index of 1.
- The length of original navigation increases by one, in which case we moved the old spine navigation one to the left and added a new spine navigation. We therefore have to decrease the previous index by one and add a new index of 1.
- The length of the original navigation decreases by one, which means that we increase the spine index of the previous navigation, so we may reach its split position. If that is not the case, we simply increase the index of the previous spine navigation by 1. If we reach the spine split, we have to remove the last part of the current navigation and leave the previous spine navigation as it is.

$$\rightarrow(w, vj) = \begin{cases} (\rightarrow(w), v1) & \text{if } \rightarrow(w) \neq \bot \text{ and } |\rightarrow(w)| = |w|, \\ (\rightarrow(w), v(j-1)1) & \text{if } \rightarrow(w) \neq \bot \text{ and } |\rightarrow(w)| = |w|+1, \\ (\rightarrow(w), v'(j'+1)) & \text{if } \rightarrow(w) \neq \bot, |\rightarrow(w)| = |w|-1, \\ & w = w'(d', Y', X'), v = v'j' \\ & \text{and } s(\mathrm{st}(w')) \neq (j'+1), \\ (w' \lhd (m, \overline{\mathrm{st}(w')}), v'1) & \text{if } \rightarrow(w) \neq \bot, |\rightarrow(w)| = |w|-1, \\ & w = w'(d', Y', X'), v = v'j' \\ & \text{and } s(\mathrm{st}(w')) = (j'+1), \\ \bot & \text{if } \rightarrow(w) = \bot. \end{cases}$$

5 RELATIVE SUCCINCTNESS

In this section we study the relative succinctness of FSLPs, top dags and fcns encodings. It turns out that up to multiplicative factors of size $|\Sigma|$ (number of node labels) all three formalisms are equally succinct. Moreover, the transformations between the formalisms can be computed in linear time. This allows us to transfer algorithmic results for FSLPs to top dags and TSLPs for fcns encodings, and vice versa. We take a look at the following two formalisms:

- The fcns (first child - next sibling) algebra. Instead of arbitrary horizontal concatenation $f \boxminus g$, it is only allowed to prepend a single tree in the form of $(a(x) \circledcirc f) \boxminus g$. The fcns encoding is similar to the head-tail representation used for list expressions and has the advantage of giving horizontal concatenation a unique representation.
- The cluster algebra. Cluster expressions are similar to forest expressions, but they do start with a single edge $a(x) \circledcirc b$, or with a single edge with a context at the bottom $a(x) \boxdot b(x)$. Instead of directly concatenating or replacing x, the nodes next to each other are "merged", which is similar to saying that $[\![(a(x) \circledcirc b) \boxminus (a(x) \circledcirc c)]\!] = a\langle bc \rangle$.

We will introduce new signatures for fcns encodings and clusters to distinguish them from forest expressions.

Definition 39 (fcns expressions)**.** Let Σ be an alphabet, let $\Sigma^{\text{fcns}} = \Sigma \uplus \{\bot\}$ and let $r^{\text{fcns}} \colon \Sigma^{\text{fcns}} \to \mathbb{N}$ with $r^{\text{fcns}}(a) = 2$ for every $a \in \Sigma$ and $r^{\text{fcns}}(\bot) = 0$. The *fcns signature* is $\mathcal{S}_{\text{fcns}}(\Sigma) = \mathcal{S}_{\mathcal{T}}(\Sigma^{\text{fcns}}, r^{\text{fcns}})$, i.e. the tree signature over the ranked alphabet $(\Sigma^{\text{fcns}}, r^{\text{fcns}})$. A TSLP for fcns expressions over Σ^{fcns} is called an fcns-SLP over Σ.

Definition 40 (fcns of trees)**.** Let Σ be an alphabet. The function $\text{fcns} \colon \mathcal{F}(\Sigma) \to \mathcal{T}(\Sigma^{\text{fcns}}, r^{\text{fcns}})$ is defined as follows:

- $\text{fcns}(\varepsilon) = \bot$,
- $\text{fcns}(a\langle f \rangle f') = a\langle f, f' \rangle$, where $f, f' \in \mathcal{F}(\Sigma)$ and $a \in \Sigma$.

Its inverse, $\text{fcns}^{-1} \colon \mathcal{T}(\Sigma^{\text{fcns}}, r^{\text{fcns}}) \to \mathcal{F}(\Sigma)$, is

- $\text{fcns}^{-1}(\bot) = \varepsilon$,
- $\text{fcns}^{-1}(a\langle t_1, t_2 \rangle) = a\langle \text{fcns}^{-1}(t_1) \rangle \, \text{fcns}^{-1}(t_2)$.

Top dags were introduced by Bille et al. [5] as a formalism for the compression of trees. Roughly speaking, the top dag for such a tree t is the DAG representation of an expression that evaluates to t, where the expression builds t from edges using two merge operations (horizontal and vertical merge). Since we have an algebraic setting, we introduce a new signature for clusters.

Definition 41 (Cluster expressions)**.** Let Σ be an alphabet. Define

$$\mathbf{type}_{\mathcal{C},\Sigma} = \{\mathcal{C}^a \mid a \in \Sigma\} \cup \{\mathcal{C}^a_b \mid a, b \in \Sigma\}.$$

The *cluster signature* $\mathcal{S}_\mathcal{C}(\Sigma)$ over Σ is given by the following operations:

- $\binom{a}{b} : \mathcal{C}^a$ for every $a, b \in \Sigma$,
- $\underline{\binom{a}{b}} : \mathcal{C}^a_b$ for every $a, b \in \Sigma$,
- ${}^a\odot^a : \mathcal{C}^a \times \mathcal{C}^a \to C^a$ for every $a \in \Sigma$,
- ${}^a_b\odot^a : \mathcal{C}^a_b \times \mathcal{C}^a \to \mathcal{C}^a_b$ for every $a, b \in \Sigma$,
- ${}^a\odot^a_b : \mathcal{C}^a \times \mathcal{C}^a_b \to \mathcal{C}^a_b$ for every $a, b \in \Sigma$,
- ${}^a_b\oplus^b : \mathcal{C}^a_b \times \mathcal{C}^b \to \mathcal{C}^a$ for every $a, b \in \Sigma$,
- ${}^a_b\oplus^b_c : \mathcal{C}^a_b \times \mathcal{C}^b_c \to \mathcal{C}^a_c$ for every $a, b, c \in \Sigma$.

We write $\mathcal{E}_\mathcal{C}(\Sigma) = \mathcal{E}(\mathcal{S}_\mathcal{C}(\Sigma))$. A DAG (V, Γ, ρ, S) for cluster expressions is called a *top dag*. We require that $\Gamma(S) = \mathcal{C}^a$ for some $a \in \Sigma$. Expressions of the form $\binom{a}{b}$ and $\underline{\binom{a}{b}}$, where $a, b \in \Sigma$, are also called *atomic* clusters. Cluster expressions are sometimes also called *top trees*.

Clusters themselves are special types of trees. A regular cluster is a tree that consists of at least two nodes. The most primitive clusters are therefore of the form $a\langle b\rangle$ for $a, b \in \Sigma$. Clusters with a bottom-boundary node are trees with a parameter that also consist of at least two nodes. In addition, the parameter must not have any siblings. The most primitive clusters with a bottom-boundary node are of the form $a\langle b\langle x\rangle\rangle$ with $a, b \in \Sigma$. We define this as follows:

Definition 42 (Clusters)**.** The top node of a tree $\triangle : \mathcal{T}(\Sigma) \cup \mathcal{T}_x(\Sigma) \to \Sigma$ is defined by $\triangle(a\langle f\rangle) = a$, where $f \in \mathcal{F}(\Sigma) \cup \mathcal{F}_x(\Sigma)$. The *clusters with top-boundary node* $a \in \Sigma$ are

$$\mathcal{C}^a(\Sigma) = \{t \in \mathcal{T}(\Sigma) \mid \triangle(t) = a\} \setminus \{a\langle\rangle \mid a \in \Sigma\}.$$

Let $\triangledown : \mathcal{T}_x(\Sigma) \setminus \{x\} \to \Sigma$ return the label above x. The *clusters with top-boundary node* $a \in \Sigma$ and *bottom-boundary node* $b \in \Sigma$ are

$$\begin{aligned}\mathcal{C}^a_b(\Sigma) = \{t \in \mathcal{T}(\Sigma) \mid \triangle(t) = a \wedge \triangledown(t) = b \wedge \exists t' \in \mathcal{T}_x(\Sigma).t = t'[b\langle x\rangle]\}\\ \setminus \{c\langle x\rangle\} \mid c \in \Sigma\}.\end{aligned}$$

The *clusters of rank* 0 are $\mathcal{C}(\Sigma) = \bigcup\{\mathcal{C}^a(\Sigma) \mid a \in \Sigma\}$ and the *clusters of rank* 1 are $\mathcal{C}_x(\Sigma) = \bigcup\{\mathcal{C}^a_b(\Sigma) \mid a, b \in \Sigma\}$.

The definition of $\mathcal{C}^a_b(\Sigma)$ is rather technical but can be explained as follows: Since we ensure that every element of $\mathcal{C}^a_b(\Sigma)$ has a subtree of the form $b\langle x\rangle$, we make sure that x has no siblings. By further ensuring that $a\langle x\rangle$ itself is not included in $\mathcal{C}^a_a(\Sigma)$, we have that elements from $\mathcal{C}^a_a(\Sigma)$ must be at least of the form $a\langle a\langle x\rangle\rangle$, since $a\langle f_1 x f_2\rangle \notin \mathcal{C}^a_a(\Sigma)$ if $f_1 \neq \varepsilon$ or $f_2 \neq \varepsilon$.

Definition 43 (Standard cluster algebra)**.** Let $\tau_\mathcal{C} : \mathcal{C}(\Sigma) \cup \mathcal{C}_x(\Sigma) \to \mathbf{type}_{\mathcal{C},\Sigma}$ be defined by

$$\tau_\mathcal{C}(t) = \begin{cases}\mathcal{C}^a & \text{if } t \in \mathcal{C}^a(\Sigma),\\ \mathcal{C}^a_b & \text{if } t \in \mathcal{C}^a_b(\Sigma).\end{cases}$$

The *standard cluster algebra* $\mathcal{A}_{\mathcal{C},\Sigma} = ((\mathcal{C}(\Sigma) \cup \mathcal{C}_x(\Sigma), \tau_{\mathcal{C}}), \mathcal{I}_{\mathcal{C}})$ over Σ evaluates cluster expressions to clusters, where the following functions (which get the same name as the operators in the signature) are used to define $\mathcal{I}_{\mathcal{C}}$: Let $a, b, c \in \Sigma$. We define

- ${}^{a}\odot^{a} : \mathcal{C}^a(\Sigma) \times \mathcal{C}^a(\Sigma) \to \mathcal{C}^a(\Sigma)$ with ${}^{a}\odot^{a}\,(a\langle f\rangle, a\langle f'\rangle) = a\langle ff'\rangle$,
- ${}^{a}_{b}\odot^{a} : \mathcal{C}^a_b(\Sigma) \times \mathcal{C}^a(\Sigma) \to \mathcal{C}^a_b(\Sigma)$ with ${}^{a}_{b}\odot^{a}\,(a\langle f\rangle, a\langle f'\rangle) = a\langle ff'\rangle$,
- ${}^{a}\odot^{a}_{b} : \mathcal{C}^a(\Sigma) \times \mathcal{C}^a_b(\Sigma) \to \mathcal{C}^a_b(\Sigma)$ with ${}^{a}\odot^{a}_{b}\,(a\langle f\rangle, a\langle f'\rangle) = a\langle ff'\rangle$,
- ${}^{a}_{b}\oplus^{b} : \mathcal{C}^a_b(\Sigma) \times \mathcal{C}^b(\Sigma) \to \mathcal{C}^a(\Sigma)$ with ${}^{a}_{b}\oplus^{b}\,(a\langle f\rangle, b\langle f'\rangle) = a\langle f\rangle[f']$,
- ${}^{a}_{b}\oplus^{b}_{c} : \mathcal{C}^a_b(\Sigma) \times \mathcal{C}^b_c(\Sigma) \to \mathcal{C}^a_c(\Sigma)$ with ${}^{a}_{b}\oplus^{b}_{c}\,(a\langle f\rangle, b\langle f'\rangle) = a\langle f\rangle[f']$.

The evaluation is defined as follows:

- $\mathcal{I}_{\mathcal{C}}(\binom{a}{b}) = a\langle b\rangle$,
- $\mathcal{I}_{\mathcal{C}}(\binom{a}{\underline{b}}) = a\langle b\langle x\rangle\rangle$,
- $\mathcal{I}_{\mathcal{C}}({}^{a}\odot^{a})(t, t') = {}^{a}\odot^{a}\,(t, t')$,
- $\mathcal{I}_{\mathcal{C}}({}^{a}_{b}\odot^{a})(t, t') = {}^{a}_{b}\odot^{a}\,(t, t')$,
- $\mathcal{I}_{\mathcal{C}}({}^{a}\odot^{a}_{b})(t, t') = {}^{a}\odot^{a}_{b}\,(t, t')$,
- $\mathcal{I}_{\mathcal{C}}({}^{a}_{b}\oplus^{b})(t, t') = {}^{a}_{b}\oplus^{b}\,(t, t')$,
- $\mathcal{I}_{\mathcal{C}}({}^{a}_{b}\oplus^{b}_{c})(t, t') = {}^{a}_{b}\oplus^{b}_{c}\,(t, t')$.

5.1 COMPARISON WITH TOP DAGS

Proposition 5. *For a given top dag $D = (V, \Gamma, \rho, S)$ one can compute in linear time an FSLP F such that $[\![F]\!] = [\![D]\!]$ and $|F| \in \mathcal{O}(|D|)$.*

Proof. Let $\sqcap : \mathcal{C}(\Sigma) \to \mathcal{F}(\Sigma)$ be the function that removes the top node of a cluster, i.e. $\sqcap(a\langle f\rangle) = f$. We translate every cluster expression using $\phi : \mathcal{E}(\mathcal{S}_{\mathcal{C}}(\Sigma), \Gamma) \to \mathcal{E}(\mathcal{S}_{\mathcal{F}}(\Sigma), \Gamma)$ such that $[\![\,]\!]_F \circ \phi = \sqcap \circ [\![\,]\!]_D$. The fact that ϕ has this property can be easily verified. The individual cases for ϕ are as follows:

- $\phi(A) = A$ for $A \in V$,
- $\phi(\binom{a}{b}) = b(x) \mathbin{\textcircled{\scriptsize$\cdot$}} \varepsilon$,
- $\phi(\binom{a}{\underline{b}}) = b(x)$,
- $\phi(t\ {}^{a}\odot^{a}\ t') = \phi(t) \boxminus \phi(t')$,
- $\phi(t\ {}^{a}_{b}\odot^{a}\ t') = \phi(t) \oslash \phi(t')$,
- $\phi(t\ {}^{a}\odot^{a}_{b}\ t') = \phi(t) \oslash \phi(t')$,
- $\phi(t\ {}^{a}_{b}\oplus^{b}\ t') = \phi(t) \mathbin{\textcircled{\scriptsize$\cdot$}} \phi(t')$,
- $\phi(t\ {}^{a}_{b}\oplus^{b}_{c}\ t') = \phi(t) \boxplus \phi(t')$.

To define the typing of our FSLP, let $\tau\colon \mathbf{type}_{\mathcal{C},\Sigma} \to \mathbf{type}_{\mathcal{F}}$ with $\tau(\mathcal{C}^a) = \mathcal{F}$ and $\tau(\mathcal{C}^a_b) = \mathcal{F}_x$, where $a, b \in \Sigma$. We define $F = (V \uplus \{S'\}, \Gamma', \rho', S')$ with $\Gamma'(A) = \tau(\Gamma(A))$ for all $A \in V$ and $\Gamma'(S') = \mathcal{F}$. To define ρ' we set $\rho'(A) = \phi(\rho(A))$ for every $A \in V$. Let $\widehat{D} = \triangle([\![D]\!])$. We set $\rho'(S') = \widehat{D}(x) \otimes S$. This yields

$$[\![F]\!] = [\![S']\!]_F = [\![\widehat{D}(x) \otimes S]\!]_F = \widehat{D}\langle [\![S]\!]_F \rangle = \widehat{D}\langle \sqcap([\![S]\!]_D) \rangle = [\![S]\!]_D = [\![D]\!].$$

□

Proposition 6. *For a given FSLP F with $[\![F]\!] \in \mathcal{C}(\Sigma)$ one can compute in time $\mathcal{O}(|\Sigma| \cdot |F|)$ a top dag D such that $[\![D]\!] = [\![F]\!]$ and $|D| \in \mathcal{O}(|\Sigma| \cdot |F|)$.*

Proof. Let $F = (V, \Gamma, \rho, S)$ be an FSLP with $[\![F]\!] \in \mathcal{C}(\Sigma)$. We use Lemma 10 and assume that F is in normal form such that for all $A \in V$ we have $[\![A]\!]_F \neq x$ and $[\![A]\!]_F \neq \varepsilon$. Every $A \in V$ that is not of the form $\rho(A) = B \boxminus C$ produces a tree, i.e. $[\![A]\!]_F \in \mathcal{T}(\Sigma) \cup \mathcal{T}_x(\Sigma)$. Hence, for these A we can define $\widehat{A} = \triangle([\![A]\!]_F) \in \Sigma$, which is the label of the root node of the tree (context) $[\![A]\!]_F$. Let

$$U_0 = \{A \in V \mid [\![A]\!]_F \in \mathcal{T}(\Sigma),\ \rho(A) \neq a(x) \otimes \varepsilon,\ a \in \Sigma\}.$$

We define a top dag $D = (V', \Gamma', \rho', S)$ with

$$V'_0 = U_0 \uplus \{A^a \mid A \in V_0,\ a \in \Sigma\},$$
$$V'_1 = \{A_b \mid A \in V_1,\ b \in \Sigma\},$$

and where Γ' is defined as

- $\Gamma'(A) = \mathcal{C}^{\widehat{A}}$ for all $A \in \mathcal{U}_0$,
- $\Gamma'(A^a) = \mathcal{C}^a$ for all $A^a \in V'_0$,
- $\Gamma'(A_b) = \mathcal{C}^{\widehat{A}}_b$ for all $A_b \in V'_1$, $b \in \Sigma$.

We will define the right-hand side mapping ρ' of D such that the following identities hold:

1. $[\![A]\!]_D = [\![A]\!]_F$ for every $A \in U_0$,
2. $[\![A^a]\!]_D = a\langle [\![A]\!]_F \rangle$ for every $A \in V_0$,
3. $[\![A_b]\!]_D = [\![A]\!]_F[b\langle x \rangle]$ for every $A \in V_1$.

In order to obtain these identities, we define ρ' as follows:

- If $\rho(A) = b(x) \otimes \varepsilon$ for $A \in V_0$ then $\rho'(A^a) = \binom{a}{b}$.
- If $\rho(A) = a(x)$ for $A \in V_1$ then $\rho'(A_b) = \binom{a}{\underline{b}}$.
- If $\rho(A) = B \boxminus C$ for $A, B, C \in V_0$ then $\rho'(A^a) = B^a \, {}^a\!\odot^a C^a$.
- If $A \in U_0$ then $\rho'(A^a) = \binom{a}{\widehat{A}}\ {}^a_{\widehat{A}}\!\oplus^{\widehat{A}} A$.
- If $\rho(A) = a(x) \otimes B$ (hence $A \in U_0$) then $\rho'(A) = B^a$.
- If $\rho(A) = B \otimes (a(x) \otimes C)$, $A \in U_0$, $C \in V_0$ and $B \in V_1$, then

$$\rho'(A) = B_a \, {}^{\widehat{B}}_a\!\oplus^a C^a.$$

- If $\rho(A) = B \otimes (a(x) \otimes \varepsilon)$, $a \in \Sigma$ and $C \in V_1$ (so $A \in U_0$) then $\rho'(A) = B_a$.

- If $\rho(A) = B \boxplus C$ for $A, B, C \in V_1$ then $\rho'(A_b) = B_{\widehat{C}} \,{}^{\widehat{B}}_{\widehat{C}}\!\oplus^{\widehat{C}}_{b} C_b$.
- If $\rho(A) = a(x) \boxplus (B \boxtimes x \boxtimes C)$ for $A \in V_1$, $B, C \in V_0$ then
$$\rho'(A_b) = B^a \,{}^a\!\odot^a \left(\binom{a}{\underline{b}} \,{}^a\!\odot^a C^a \right).$$
- If $\rho(A) = a(x) \boxplus (B \boxtimes x)$ for $A \in V_1$, $B \in V_0$ then $\rho'(A_b) = B^a \,{}^a\!\odot^a \binom{a}{\underline{b}}$.
- If $\rho(A) = a(x) \boxplus (x \boxtimes C)$ for $A \in V_1$, $C \in V_0$ then $\rho'(A_b) = \binom{a}{\underline{b}} \,{}^a\!\odot^a C^a$.

The correctness of this construction follows by induction, using 1–3. To conclude the proof, note that since $[\![F]\!] \in \mathcal{C}(\Sigma)$, the start symbol S of F must belong to U_0. Hence, the above point 1 implies $[\![D]\!] = [\![F]\!]$. □

The following example shows that the size bound in Proposition 6 is sharp:

Example 5. Let $\Sigma = \{a, a_1, \ldots, a_\sigma\}$ and let $t_n = a\langle a_1\langle a^m\rangle \cdots a_\sigma\langle a^m\rangle\rangle$ where $n \geq 1$ and $m = 2^n$. For every $n > \sigma$ the tree t_n can be produced by an FSLP of size $\mathcal{O}(n)$: using $n = \log_2 m$ many variables we can produce the forest a^m and then $\mathcal{O}(n)$ many additional variables suffice to produce t_n. On the other hand, every top dag for t_n has size $\Omega(\sigma \cdot n)$: consider a top tree e that evaluates to t_n. Then e must contain a subexpression e_i that evaluates to the subtree $a_i\langle a^m\rangle$ $(1 \leq i \leq \sigma)$ of t_n. The subexpression e_i has to produce $a_i\langle a^m\rangle$ using the ${}^{a_i}\!\odot^{a_i}$-operation from copies of $a_i\langle a\rangle$. Hence, the expression for $a_i\langle a^m\rangle$ has size $n = \log_2 m$ and different e_i contain no identical subexpressions. Therefore every top dag for t_n has size at least $\sigma \cdot n$.

5.2 COMPARISON WITH FCNS

In contrast to top dags, FSLPs and TSLPs for fcns encodings turn out to be equally succinct up to constant factors. First, we show that we can convert an FSLP for a ranked tree into a TSLP. Let (Σ, r) with $r\colon \Sigma \to \mathbb{N}$ be a ranked alphabet. The partial function $\mathrm{tr}_r\colon \mathcal{F}(\Sigma) \to \mathcal{T}(\Sigma, r)$ converts a forest into a ranked tree if in every node the number of children satisfies r, i.e.

$$\mathrm{tr}_r(a\langle t_1 \ldots t_n\rangle) = a\langle \mathrm{tr}_r(t_1), \ldots, \mathrm{tr}_r(t_n)\rangle$$

if $n = r(a)$, where $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$, and $\mathrm{tr}_r(t_1), \ldots, \mathrm{tr}_r(t_n)$ are also defined.

Lemma 16. *Given an FSLP F with $\mathrm{tr}_r([\![F]\!]) \in \mathcal{T}(\Sigma, r)$ we can in linear time produce a TSLP T with $[\![T]\!] = \mathrm{tr}_r([\![F]\!])$ and $|T| \in \mathcal{O}(|F|)$.*

Proof. Let $F = (V, \Gamma, \rho, S)$ be an FSLP in normal form. The TSLP is $T = (V', \Gamma', \rho', S)$ with

$$\begin{aligned} V' = {} & \{A \in V_1 \mid [\![A]\!]_F \neq x\} \cup \{A \in V_0 \mid [\![A]\!]_F \in \mathcal{T}(\Sigma)\} \\ & \cup \{A' \mid A \in V,\ \rho(A) = B \circledcirc (a(x) \circledcirc C),\ a \in \Sigma,\ B, C \in V\}. \end{aligned}$$

Let $\tau\colon \mathbf{type}_{\mathcal{F}} \to \mathbf{type}_{\mathcal{T}}$ with $\tau(\mathcal{F}) = \mathcal{T}$ and $\tau(\mathcal{F}_x) = \mathcal{T}_x$. We set $\Gamma'(A) = \tau(\Gamma(A))$ for all $A \in V$ and $\Gamma'(A) = \mathcal{T}$ for all $A \in V' \setminus V$. We translate the individual cases as follows:

- In case $\rho(A) = \varepsilon$ or $\rho(A) = x$ then $A \notin V'$.
- In case $\rho(A) = B \boxminus C$ then $\rho'(A) = \rho(B)$ if $[\![C]\!]_F = \varepsilon$ and $[\![B]\!]_F \neq \varepsilon$, $\rho'(A) = \rho(C)$ if $[\![B]\!]_F = \varepsilon$ and $[\![C]\!]_F \neq \varepsilon$. If $[\![A]\!]_F = \varepsilon$ or $[\![A]\!]_F \notin \mathcal{T}(\Sigma)$ then $A \notin V'$.

- In case $\rho(A) = B \boxplus C$ then $\rho'(A) = \rho(B)$ if $[\![C]\!]_F = x$ and $[\![B]\!]_F \neq x$, $\rho'(A) = \rho(C)$ if $[\![B]\!]_F = x$ and $[\![C]\!]_F \neq x$. If $[\![A]\!]_F = x$ then $A \notin V'$.
- In case $\rho(A) = a(x) \boxplus (L \boxminus x \boxminus R)$ let $[\![L]\!]_{F_\boxminus} = \beta_1 \dots \beta_{i-1}$ and $[\![R]\!]_{F_\boxminus} = \beta_i \dots \beta_{r(a)-1}$ for some $1 \leq i < r(a)$ and $\beta_j \in \mathcal{E}(\mathcal{S}_\mathcal{F}(\Sigma), \Gamma)$ for $1 \leq j < r(a)$. By the definition of $F_\boxminus$ we must have variables $B_1, \dots, B_{r(a)-1} \in V$ with $[\![B_j]\!]_F = [\![\beta_j]\!]_F$ for all $1 \leq j < r(a)$. Since for all $1 \leq j < r(a)$ we have $[\![B_j]\!]_F \in \mathcal{T}(\Sigma)$, we can set $\rho(A) = a_i(B_1, \dots, B_{r(a)-1})$.
- In case $\rho(A) = B \boxtimes (a(x) \boxtimes C)$ let $[\![C]\!]_{F_\boxminus} = C_1 \dots, C_{r(a)}$. In case $r(a) > 0$ we set $\rho'(A') = a(x, C_2, \dots, C_{r(a)-1}) \boxtimes C_1$. In case $r(a) = 0$ we set $\rho'(A') = a$. Additionally, we set $\rho(A) = B \boxtimes A'$ if $B \in V'$, and $\rho(A) = \rho(A')$ otherwise.

□

Proposition 7. *Let T be an fcns-SLP over Σ. We can transform T into an FSLP F with $|F| \in \mathcal{O}(|T|)$ such that $[\![T]\!] = \text{fcns}([\![F]\!])$.*

Proof. Let $T = (V, \Gamma, \rho, S)$. By Lemma 6, we may assume that T is in normal form. Since every $a \in \Sigma$ has rank 2, the possible cases for ρ are

- $\rho(A) = \bot$,
- $\rho(A) = a(B, x)$,
- $\rho(A) = a(x, B)$,
- $\rho(A) = B \boxtimes C$,
- $\rho(A) = B \boxplus C$.

Let $F = (V, \Gamma', \rho', S)$ with $\Gamma'(A) = \mathcal{F}$ if $\Gamma(A) = \mathcal{T}$ and $\Gamma'(A) = \mathcal{F}_x$ if $\Gamma(A) = \mathcal{T}_x$. We can easily define ρ' by translating right-hand sides of the above forms into right-hand sides for fcns^{-1}:

- $\rho(A) = \bot$ becomes $\rho'(A) = \varepsilon$.
- $\rho(A) = a(B, x)$ becomes $\rho'(A) = (a(x) \boxtimes B) \boxminus x$.
- $\rho(A) = a(x, B)$ becomes $\rho'(A) = (a(x) \boxplus x) \boxminus B$.
- $\rho(A) = B \boxtimes C$ and $\rho'(A) = B \boxplus C$ stay the same.

For the correctness of the construction, we have to show that $\text{fcns}([\![F]\!]) = [\![T]\!]$. In order to do this, we show the following properties:

- $\text{fcns}([\![A]\!]_F) = [\![A]\!]_T$ for all $A \in V_0$,
- $\text{fcns}([\![A]\!]_F[f]) = [\![A]\!]_T[\text{fcns}(f)]$ for all $A \in V_1$, $f \in \mathcal{F}(\Sigma)$.

These are shown using a simple induction and case analysis:

- $\rho(A) = \bot$: $\text{fcns}([\![A]\!]_F) = \text{fcns}(\varepsilon) = \bot = [\![A]\!]_T$.
- $\rho(A) = a(B, C)$: We obtain ("ind" refers to induction on B and C)

$$\begin{aligned}
\text{fcns}([\![A]\!]_F) &= \text{fcns}([\![(a(x) \boxtimes B) \boxplus C]\!]_F) \\
&= \text{fcns}(a\langle [\![B]\!]_F \rangle [\![C]\!]_F) \\
&= a\langle \text{fcns}([\![B]\!]_F), \text{fcns}([\![C]\!]_F) \rangle \\
&\overset{\text{ind}}{=} a\langle [\![B]\!]_T, [\![C]\!]_T \rangle \\
&= [\![A]\!]_T.
\end{aligned}$$

- $\rho(A) = a(B,x)$: We obtain

$$\begin{aligned}
\mathrm{fcns}(\llbracket A \rrbracket_F[f]) &= \mathrm{fcns}(\llbracket (a(x) \varovee B) \varolessthan x \rrbracket_F[f]) \\
&= \mathrm{fcns}(a\langle \llbracket B \rrbracket_F \rangle f) \\
&= a\langle \mathrm{fcns}(\llbracket B \rrbracket_F), \mathrm{fcns}(f) \rangle \\
&\stackrel{\text{ind}}{=} a\langle \llbracket B \rrbracket_T, \mathrm{fcns}(f) \rangle \\
&= \llbracket a(B,x) \rrbracket_T[\mathrm{fcns}(f)] \\
&= \llbracket A \rrbracket_T[\mathrm{fcns}(f)].
\end{aligned}$$

- $\rho(A) = a(x,B)$: We obtain

$$\begin{aligned}
\mathrm{fcns}(\llbracket A \rrbracket_F[f]) &= \mathrm{fcns}(\llbracket (a(x) \boxbar x) \varogreaterthan B \rrbracket_F[f]) \\
&= \mathrm{fcns}(a\langle f \rangle \llbracket B \rrbracket_F) \\
&= a\langle \mathrm{fcns}(f), \mathrm{fcns}(\llbracket B \rrbracket_F) \rangle \\
&\stackrel{\text{ind}}{=} a\langle \mathrm{fcns}(f), \llbracket B \rrbracket_T \rangle \\
&= \llbracket a(x,B) \rrbracket_T[\mathrm{fcns}(f)] \\
&= \llbracket A \rrbracket_T[\mathrm{fcns}(f)].
\end{aligned}$$

- $\rho(A) = B \varovee C$: We obtain the following, where the first (resp., second) induction step uses induction on B (resp., C):

$$\begin{aligned}
\mathrm{fcns}(\llbracket A \rrbracket_F) &= \mathrm{fcns}(\llbracket B \varovee C \rrbracket_F) \\
&= \mathrm{fcns}(\llbracket B \rrbracket_F[\llbracket C \rrbracket_F]) \\
&\stackrel{\text{ind}}{=} \llbracket B \rrbracket_T[\mathrm{fcns}(\llbracket C \rrbracket_F)] \\
&\stackrel{\text{ind}}{=} \llbracket B \rrbracket_T[\llbracket C \rrbracket_T] \\
&= \llbracket B \varovee C \rrbracket_T \\
&= \llbracket A \rrbracket_T.
\end{aligned}$$

- $\rho(A) = B \boxbar C$: We obtain

$$\begin{aligned}
\mathrm{fcns}(\llbracket A \rrbracket_F[f]) &= \mathrm{fcns}(\llbracket B \boxbar C \rrbracket_F[f]) \\
&= \mathrm{fcns}((\llbracket B \rrbracket_F[\llbracket C \rrbracket_F])[f]) \\
&= \mathrm{fcns}(\llbracket B \rrbracket_F[\llbracket C \rrbracket_F[f]]) \\
&\stackrel{\text{ind}}{=} \llbracket B \rrbracket_T[\mathrm{fcns}(\llbracket C \rrbracket_T[f])] \\
&\stackrel{\text{ind}}{=} \llbracket B \rrbracket_T[\llbracket C \rrbracket_F[\mathrm{fcns}(f)]] \\
&= (\llbracket B \rrbracket_T[\llbracket C \rrbracket_T])[\mathrm{fcns}(f)] \\
&= \llbracket B \boxbar C \rrbracket_T[\mathrm{fcns}(f)] \\
&= \llbracket A \rrbracket_T[\mathrm{fcns}(f)].
\end{aligned}$$

We used the fact that $(f[g])[h] = f[g[h]]$ for every $f, g \in \mathcal{F}_x(\Sigma)$ and $h \in \mathcal{F}(\Sigma) \cup \mathcal{F}_x(\Sigma)$, which follows from an easy induction. This concludes the proof of the proposition. □

By combining Lemma 16 and Proposition 7, we obtain the following:

Corollary 2. *Let Σ be an alphabet. Given an FSLP F with $\mathrm{tr}_{\Sigma^{\mathrm{fcns}}}(\llbracket F \rrbracket) = \mathrm{fcns}(f)$ for some $f \in \mathcal{F}(\Sigma)$, we can compute in linear time an FSLP F' with $|F'| \in \mathcal{O}(|F|)$ such that $\llbracket F' \rrbracket = f$.*

Proposition 8. *For every FSLP F over Σ, we can construct in linear time an fcns-SLP T over Σ with $[\![T]\!] = \mathrm{fcns}([\![F]\!])$ and $|T| \in \mathcal{O}(|F|)$.*

Proof. We start with the definition of two functions which are closely related to fcns. The first function $\pi\colon \mathcal{F}(\Sigma) \to \mathcal{T}_x(\Sigma^{\mathrm{fcns}}, r^{\mathrm{fcns}})$ is defined inductively by

$$\begin{aligned} \pi(\varepsilon) &= x, \\ \pi(a\langle f\rangle g) &= a\langle \mathrm{fcns}(f), \pi(g)\rangle \text{ for all } a \in \Sigma,\ f, g \in \mathcal{F}(\Sigma); \end{aligned}$$

in particular its restriction to $\mathcal{T}(\Sigma)$ is given by

$$\pi(a\langle f\rangle) = a\langle \mathrm{fcns}(f), x\rangle \text{ for all } a \in \Sigma,\ f \in \mathcal{F}(\Sigma). \tag{1}$$

Simple induction proofs show that

$$\mathrm{fcns}(f) = \pi(f)[\bot] \text{ for all } f \in \mathcal{F}(\Sigma), \tag{2}$$
$$\pi(fg) = \pi(f)[\pi(g)] \text{ for all } f, g \in \mathcal{F}(\Sigma). \tag{3}$$

The second function $\varphi\colon \mathcal{F}(\Sigma) \cup \mathcal{F}_x(\Sigma) \to \mathcal{T}(\Sigma^{\mathrm{fcns}}, r^{\mathrm{fcns}}) \cup \mathcal{T}_x(\Sigma^{\mathrm{fcns}}, r^{\mathrm{fcns}})$ is defined inductively by

$$\begin{aligned} \varphi(\varepsilon) &= \bot, \\ \varphi(xg) &= x \text{ for all } g \in \mathcal{F}(\Sigma), \\ \varphi(a\langle f\rangle g) &= a\langle \varphi(f), \varphi(g)\rangle \text{ for all } a \in \Sigma,\ f, g \in \mathcal{F}(\Sigma) \cup \mathcal{F}_x(\Sigma) \\ &\qquad \text{with } a\langle f\rangle g \in \mathcal{F}(\Sigma) \cup \mathcal{F}_x(\Sigma). \end{aligned}$$

Simple induction proofs show that

$$\mathrm{fcns}(f) = \varphi(f) \text{ for all } f \in \mathcal{F}(\Sigma), \tag{4}$$
$$\varphi(fxg) = \pi(f) \text{ for all } f, g \in \mathcal{F}(\Sigma). \tag{5}$$

The most important equation for φ is

$$\begin{aligned} \varphi(f[a\langle g\rangle]) = \varphi(f)[a\langle \varphi(g), \mathrm{fcns}(\mathrm{sib}(f))\rangle] \\ \text{for all } f \in \mathcal{F}_x(\Sigma),\ a \in \Sigma,\ g \in \mathcal{F}(\Sigma), \end{aligned} \tag{6}$$

where $\mathrm{sib}(f) \in \mathcal{F}(\Sigma)$ denotes the sequence of all right siblings of x in $f \in \mathcal{F}_x(\Sigma)$, i.e.,

$$\begin{aligned} \mathrm{sib}(xg) &= g \text{ for all } g \in \mathcal{F}(\Sigma), \\ \mathrm{sib}(a\langle f\rangle g) &= \begin{cases} \mathrm{sib}(f) \text{ if } f \in \mathcal{F}_x(\Sigma) \text{ and } g \in \mathcal{F}(\Sigma), \\ \mathrm{sib}(g) \text{ if } f \in \mathcal{F}(\Sigma) \text{ and } g \in \mathcal{F}_x(\Sigma). \end{cases} \end{aligned}$$

Equation (6) tells us how to obtain $\varphi(f[a\langle g\rangle])$ from $\varphi(f)$ and $\varphi(g)$. For its proof note that $a\langle \varphi(g), \mathrm{fcns}(\mathrm{sib}(f))\rangle = a\langle \varphi(g), \varphi(\mathrm{sib}(f))\rangle = \varphi(a\langle g\rangle\, \mathrm{sib}(f))$. Hence it suffices to prove

$$\varphi(f[t]) = \varphi(f)[\varphi(t\,\mathrm{sib}(f))] \text{ for all } f \in \mathcal{F}_x(\Sigma), t \in \mathcal{T}(\Sigma) \cup \mathcal{T}_x(\Sigma),$$

which can be done by the following induction on the length of the sequence f and case distinction on the first element of f:

- $f = xg$: Then $g \in \mathcal{F}(\Sigma)$ and we have

$$\varphi(f[t]) = \varphi(tg) = x[\varphi(tg)] = \varphi(f)[\varphi(tg)] = \varphi(f)[\varphi(t\,\mathrm{sib}(f))].$$

- $f = a\langle g\rangle h$ with $a \in \Sigma$ and $g \in \mathcal{F}(\Sigma)$: Then $h \in \mathcal{F}_x(\Sigma)$ and we obtain

$$\begin{aligned}\varphi(f[t]) &= \varphi(a\langle g\rangle\, h[t])\\ &= a\langle \varphi(g), \varphi(h[t])\rangle \text{ by the definition of } \varphi\\ &= a\langle \varphi(g), \varphi(h)[\varphi(t\,\mathrm{sib}(h))]\rangle \text{ by induction for } h\\ &= a\langle \varphi(g), \varphi(h)[\varphi(t\,\mathrm{sib}(f))]\rangle \text{ because } \mathrm{sib}(f) = \mathrm{sib}(h)\\ &= a\langle \varphi(g), \varphi(h)\rangle[\varphi(t\,\mathrm{sib}(f))]\\ &= \varphi(f)[\varphi(t\,\mathrm{sib}(f))] \text{ by the definition of } \varphi.\end{aligned}$$

- $f = a\langle g\rangle h$ with $a \in \Sigma$ and $g \in \mathcal{F}_x(\Sigma)$: Then $h \in \mathcal{F}(\Sigma)$ and the proof is the same as in the previous step except that the roles of g and h are exchanged.

Equations (1) to (6) are a guideline for the construction of the TSLP T. Let $F = (V, \Gamma, \rho, S)$ be an FSLP over Σ. In case $[\![F]\!] = \varepsilon$, we simply translate to a TSLP that only produces $\bot$. Now assume that $[\![F]\!] \neq \varepsilon$. By Lemma 10 we assume that F is in normal form and that for all $A \in V$ we have $[\![A]\!]_F \neq x$ and $[\![A]\!]_F \neq \varepsilon$. Let

$$\begin{aligned}V_0^{\bot} = \{A \in V_0 \mid \rho(A) = a(x) \mathbin{\textcircled{\scriptsize :}} B,\ a \in \Sigma,\ B \in V\}\\ \cup \{A \in V_0 \mid \rho(A) = B \mathbin{\textcircled{\scriptsize :}} (a(x) \mathbin{\textcircled{\scriptsize :}} C),\ B, C \in V,\ a \in \Sigma\}.\end{aligned}$$

We then define $T = (V', \Gamma', \rho', S)$ where

$$V' = V_0 \uplus \{A_{\sqcap} \mid A \in V_0^{\bot}\} \uplus \{A^{\pi} \mid A \in V_0\} \uplus \{A_{\sqcap} \mid A \in V_1\}$$

with $\Gamma'(A) = \mathcal{T}$ for $A \in V_0$, $\Gamma'(A_{\sqcap}) = \mathcal{T}$ for $A \in V_0^{\bot}$, $\Gamma'(A^{\pi}) = \mathcal{T}_x$ for $A \in V_0$ and $\Gamma'(A_{\sqcap}) = \mathcal{T}_x$ for $A \in V_1$. To explain the role of these variables let $\Delta\colon \mathcal{T}(\Sigma) \setminus \{x\} \to \mathcal{F}(\Sigma)$ be defined by $\Delta(a\langle f\rangle) = f$. We want to achieve that

$$[\![A]\!]_T = \mathrm{fcns}([\![A]\!]_F) \text{ for every } A \in V_0, \tag{7}$$

$$[\![A^{\pi}]\!]_T = \pi([\![A]\!]_F) \text{ for every } A \in V_0, \tag{8}$$

$$[\![A_{\sqcap}]\!]_T = \varphi(\Delta([\![A]\!]_F)) \text{ for every } A \in V_0^{\bot} \cup V_1. \tag{9}$$

From (7) we obtain $[\![T]\!] = [\![S]\!]_T = \mathrm{fcns}([\![S]\!]_F) = \mathrm{fcns}([\![F]\!])$ which concludes the proof of the proposition (assuming that T satisfies the size bound $|T| \in \mathcal{O}(|F|)$).

It remains to define ρ' in such a way that (7), (8) and (9) are satisfied. For every $A \in V_0^{\bot} \cup V_1$ let α_A denote the root label of $[\![A]\!]_F$, and for every $A \in V_1$ let $R_A \in V_0$ be a variable with $[\![R_A]\!]_F = \mathrm{sib}([\![A]\!]_F)$. Such a variable exists in V_0, namely

- $R_A = C$ if $\rho(A) = a(x) \mathbin{\square} (B \mathbin{\textcircled{\scriptsize $\leftarrow$}} x \mathbin{\textcircled{\scriptsize $\rightarrow$}} C)$,
- $R_A = R_C$ if $\rho(A) = B \mathbin{\square} C$ for $B, C \in V_1$.

Then we define ρ' by

- $\rho'(A) = A^{\pi} \mathbin{\textcircled{\scriptsize :}} \bot$ if $A \in V_0$,
- $\rho'(A^{\pi}) = \widehat{A}(A_{\sqcap}, x)$ if $A \in V_0^{\bot}$,
- $\rho'(A^{\pi}) = B^{\pi} \mathbin{\square} C^{\pi}$ if $A \in V_0^{\top}$ with $\rho(A) = B \boxminus C$,
- $\rho'(A_{\sqcap}) = C$ if $A \in V_0^{\bot}$ with $\rho(A) = a(x) \mathbin{\textcircled{\scriptsize :}} C$,
- $\rho'(A_{\sqcap}) = B_{\sqcap} \mathbin{\textcircled{\scriptsize :}} a(C, R_B)$ if $A \in V_0^{\bot}$ with $\rho(A) = B \mathbin{\textcircled{\scriptsize :}} (a(x) \mathbin{\textcircled{\scriptsize :}} C)$,

- $\rho'(A_\sqcap) = B^\pi$ if $A \in V_1$ with $\rho(A) = a(x) \boxplus (B \oslash x \oslash C)$,
- $\rho'(A_\sqcap) = B_\sqcap \boxplus \widehat{C}(C_\sqcap, R_B)$ if $A \in V_1$ with $\rho(A) = B \boxplus C$.

This concludes the definition of the TSLP T. It is clear that T can be constructed from F in linear time and that $|T| \in \mathcal{O}(|F|)$.

Equations (7), (8) and (9) are proved by the following induction on $\leq_T$. Note that $A_\sqcap \leq_T A^\pi$ for all $A \in V_0^\perp$ and $A^\pi \leq_T A$ for all $A \in V_0$.

- If $A \in V_0$ then $\rho'(A) = A^\pi \otimes \perp$ and thus

$$\begin{aligned}[\![A]\!]_T &= [\![A^\pi]\!]_T[\perp] \\ &= \pi([\![A]\!]_F)[\perp] \text{ by induction for } A^\pi \\ &= \mathrm{fcns}([\![A]\!]_F) \text{ by equation (2).}\end{aligned}$$

- If $A \in V_0^\perp$ with $[\![A]\!]_F = a\langle f\rangle$ then $\rho'(A^\pi) = a(A_\sqcap, x)$ and thus

$$\begin{aligned}[\![A^\pi]\!]_T &= a\langle [\![A_\sqcap]\!]_T, x\rangle \\ &= a\langle \varphi(\Delta([\![A]\!]_F)), x\rangle \text{ by induction for } A_\sqcap \\ &= a\langle \varphi(f), x\rangle \\ &= a\langle \mathrm{fcns}(f), x\rangle \text{ by equation (4)} \\ &= \pi(a\langle f\rangle) \text{ by equation (1)} \\ &= \pi([\![A]\!]_F).\end{aligned}$$

- If $A \in V_0^\top$ with $\rho(A) = B \boxminus C$ then $\rho'(A^\pi) = B^\pi \boxplus C^\pi$ and thus

$$\begin{aligned}[\![A^\pi]\!]_T &= [\![B^\pi]\!]_T[[\![C^\pi]\!]_T] \\ &= \pi([\![B]\!]_F)[\pi([\![C]\!]_F)] \text{ by induction for } B^\pi \text{ and } C^\pi \\ &= \pi([\![B]\!]_F[\![C]\!]_F) \text{ by equation (3)} \\ &= \pi([\![A]\!]_F).\end{aligned}$$

- If $A \in V_0^\perp$ with $\rho(A) = a(x) \otimes B$ then $\rho'(A_\sqcap) = B$ and thus

$$\begin{aligned}[\![A_\sqcap]\!]_T &= [\![B]\!]_T \\ &= \mathrm{fcns}([\![B]\!]_F) \text{ by induction for } B \\ &= \varphi([\![B]\!]_F) \text{ by equation (4)} \\ &= \varphi(\Delta([\![A]\!]_F)).\end{aligned}$$

- If $A \in V_0^\perp$ with $\rho(A) = B \otimes (a(x) \otimes C)$ then $\rho'(A_\sqcap) = B_\sqcap \otimes a(C, R_B)$, so

$$\begin{aligned}[\![A_\sqcap]\!]_T &= [\![B_\sqcap]\!]_T[a\langle \mathrm{fcns}([\![C]\!]_F), \mathrm{fcns}([\![R_B]\!]_F)\rangle] \text{ by induction for } C \text{ and } R_B \\ &= [\![B_\sqcap]\!]_T[\varphi(a\langle [\![C]\!]_F\rangle [\![R_B]\!]_F)] \\ &= \varphi(\Delta([\![B]\!]_F))[\varphi(a\langle [\![C]\!]_F\rangle [\![R_B]\!]_F)] \text{ by induction for } B \\ &= \varphi(\Delta([\![B]\!]_F))[a\langle [\![C]\!]_F\rangle] \text{ by equation (6)} \\ &= \varphi(\Delta([\![A]\!]_F)).\end{aligned}$$

- If $A \in V_1$ with $\rho(A) = a(x) \boxplus (B \oslash x \oslash C)$ then $\rho'(A_\sqcap) = B^\pi$ and thus

$$\begin{aligned}[\![A_\sqcap]\!]_T &= [\![B^\pi]\!]_T \\ &= \pi([\![B]\!]_F) \text{ by induction for } B \\ &= \varphi([\![B]\!]_F \, x \, [\![C]\!]_F) \text{ by equation (5)} \\ &= \varphi(\Delta([\![A]\!]_F)).\end{aligned}$$

- If $A \in V_1$ with $\rho(A) = B \boxplus C$, $[\![B]\!]_F = b\langle f \rangle$ and $[\![C]\!]_F = c\langle g \rangle$ then $\rho'(A) = B_\sqcap \boxplus (c(x) \boxplus (C_\sqcap \oslash R_B))$ and we have $[\![R_B]\!]_F = \mathrm{sib}(b\langle f \rangle) = \mathrm{sib}(f)$. Thus we obtain

$$\begin{aligned}
[\![A_\sqcap]\!]_T &= [\![B_\sqcap]\!]_T[c\langle [\![C_\sqcap]\!]_T, [\![R_B]\!]_T \rangle] \\
&= \varphi(\Delta([\![B]\!]_F))[c\langle \varphi(\Delta([\![C]\!]_F)), \mathrm{fcns}([\![R_B]\!]_F) \rangle] \\
&\quad \text{by induction for } B_\sqcap,\ C_\sqcap \text{ and } R_B \\
&= \varphi(f)[c\langle \varphi(g), \mathrm{fcns}(\mathrm{sib}(f)) \rangle] \\
&= \varphi(f[c\langle g \rangle]) \text{ by equation (6)} \\
&= \varphi(\Delta(b\langle f \rangle[c\langle g \rangle])) \\
&= \varphi(\Delta([\![B]\!]_F[[\![C]\!]_F])) \\
&= \varphi(\Delta([\![A]\!]_F)).
\end{aligned}$$

This concludes the proof of the proposition. □

Finally, let us define the function $\mathrm{u}\colon \mathcal{T}(\Sigma, r) \to \mathcal{F}(\Sigma)$, where (Σ, r) is a ranked alphabet, by setting $\mathrm{u}(a\langle t_1, \ldots, t_n \rangle) = a\langle t_1 \ldots t_n \rangle$. The following is easy to show:

Lemma 17. *Given a TSLP T, we can compute in linear time an FSLP F with $|F| \in \mathcal{O}(|T|)$ and $[\![F]\!] = \mathrm{u}([\![T]\!])$.*

By combining Proposition 8 and Lemma 17, we obtain the following:

Corollary 3. *Given an FSLP F, we can compute in linear time an FSLP F' with $|F'| \in \mathcal{O}(|F|)$ and $[\![F']\!] = \mathrm{u}(\mathrm{fcns}([\![F]\!]))$.*

6 TESTING EQUALITY MODULO ASSOCIATIVITY AND COMMUTATIVITY

In this section we will give an algorithmic application which proves the usefulness of FSLPs (even if we deal with binary trees). We fix two subsets $\mathcal{A} \subseteq \Sigma$ (the set of *associative symbols*) and $\mathcal{C} \subseteq \Sigma$ (the set of *commutative symbols*). This means that we impose the following identities for all $a \in \mathcal{A}$, $c \in \mathcal{C}$, all trees $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$, all permutations $\sigma: \{1, \ldots, n\} \to \{1, \ldots, n\}$, and all $1 \le i \le j \le n+1$:

$$a\langle t_1 \cdots t_n \rangle = a\langle t_1 \cdots t_{i-1} a\langle t_i \cdots t_{j-1} \rangle t_j \cdots t_n \rangle, \tag{ASSOC}$$

$$c\langle t_1 \cdots t_n \rangle = c\langle t_{\sigma(1)} \cdots t_{\sigma(n)} \rangle. \tag{COMM}$$

Note that the standard law of associativity for a binary symbol $\circ$ (i.e., $x \circ (y \circ z) = (x \circ y) \circ z$) can be captured by making $\circ$ an associative symbol in the sense of (ASSOC).

6.1 ASSOCIATIVE SYMBOLS

Below, we define the associative normal form $\mathrm{nf}_{\mathcal{A}}(f)$ of a forest f and show that from an FSLP F we can compute in linear time an FSLP F' with $[\![F']\!] = \mathrm{nf}_{\mathcal{A}}([\![F]\!])$. For trees $s, t \in \mathcal{T}(\Sigma)$ we have that $s = t$ modulo the identities in (ASSOC) if and only if $\mathrm{nf}_{\mathcal{A}}(s) = \mathrm{nf}_{\mathcal{A}}(t)$. The generalization to forests is needed for the induction, where a slight technical problem arises. Whether the forests $t_1 \cdots t_{i-1} a\langle t_i \cdots t_{j-1} \rangle t_j \cdots t_n$ and $t_1 \cdots t_n$ are equal modulo the identities in (ASSOC) actually depends on the symbol on top of these two forests. If it is an a, and $a \in \mathcal{A}$, then the two forests are equal modulo associativity, otherwise not. To cope with this problem, we use for every associative symbol $a \in \mathcal{A}$ a function $\phi_a: \mathcal{F}(\Sigma) \to \mathcal{F}(\Sigma)$ that pulls up occurrences of a whenever possible.

For every $a \in \Sigma_\perp := \Sigma \uplus \{\perp\}$, where $\perp$ means that no symbol is on top, let $\phi_a: \mathcal{F}(\Sigma) \to \mathcal{F}(\Sigma)$ be defined as follows, where $f \in \mathcal{F}(\Sigma)$, $n \ge 0$ and $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$:

$$\phi_a(b\langle f \rangle) = \begin{cases} \phi_a(f) & \text{if } a \in \mathcal{A} \text{ and } a = b, \\ b\langle \phi_b(f) \rangle & \text{otherwise,} \end{cases}$$

$$\phi_a(t_1 \cdots t_n) = \phi_a(t_1) \cdots \phi_a(t_n).$$

Note that $\phi_a(\varepsilon) = \varepsilon$. Moreover, define $\mathrm{nf}_{\mathcal{A}}: \mathcal{F}(\Sigma) \to \mathcal{F}(\Sigma)$ by $\mathrm{nf}_{\mathcal{A}}(f) = \phi_\perp(f)$.

Example 6. Let $t = a\langle a\langle cd \rangle b\langle cd \rangle a\langle e \rangle \rangle$ and $\mathcal{A} = \{a\}$. We obtain

$$\begin{aligned} \phi_a(t) &= \phi_a(a\langle cd \rangle b\langle cd \rangle a\langle e \rangle) = \phi_a(a\langle cd \rangle)\phi_a(b\langle cd \rangle)\phi_a(a\langle e \rangle) \\ &= \phi_a(cd) b\langle \phi_b(cd) \rangle \phi_a(e) = cdb\langle cd \rangle e, \\ \phi_b(t) &= a\langle \phi_a(a\langle cd \rangle b\langle cd \rangle a\langle e \rangle) \rangle = a\langle cdb\langle cd \rangle e \rangle. \end{aligned}$$

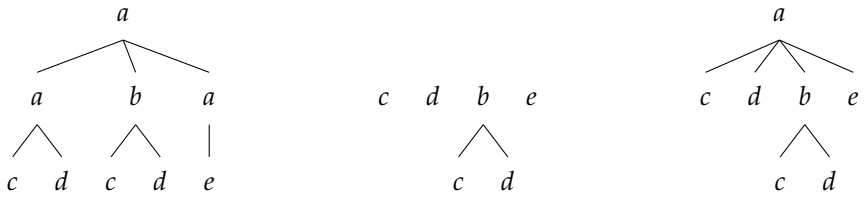


Figure 1: t on the left, $\phi_a(t)$ in the middle and $\phi_b(t)$ on the right from Example 6

Lemma 18. *From a given FSLP F over Σ one can construct in time $\mathcal{O}(|F| \cdot |\Sigma|)$ an FSLP F' with $[\![F']\!] = nf_{\mathcal{A}}([\![F]\!])$.*

Proof. By Lemma 10, we may assume that $F = (V, \Gamma, \rho, S)$ is in normal form such that there is no $A \in V$ with $[\![A]\!]_F = x$. We define $\omega\colon V_1 \to \Sigma$ such that $\omega(B)$ returns the symbol above x in $[\![B]\!]_F$:

- $\omega(A) = a$ if $\rho(A) = a(x) \boxplus (L \circledcirc x \circledcirc R)$,
- $\omega(A) = \omega(C)$ if $\rho(A) = B \boxplus C$.

These symbols can be computed all together in linear time. We introduce new variables A_a for all $a \in \Sigma_\perp$ and define the right-hand sides of F' such that $[\![A_a]\!]_{F'} = \phi_a([\![A]\!]_F)$ for all $A \in V_0$ and $[\![B_a]\!]_{F'}[\phi_{\omega(B)}(f)] = \phi_a([\![B]\!]_F[f])$ for all $B \in V_1$, $f \in \mathcal{F}(\Sigma)$. Now let $F' = (V', \Gamma', S_\perp, \rho')$ where $V' = \{A_a \mid A \in V, a \in \Sigma_\perp\}$, $\Gamma'(A_a) = \Gamma(A)$ for $A_a \in V'$ and ρ' is defined by

- $\rho'(A_a) = \varepsilon$ if $\rho(A) = \varepsilon$,
- $\rho'(A_a) = B_a \boxminus C_a$ if $\rho(A) = B \boxminus C$,
- $\rho'(A_a) = B_a \circledcirc C_b$ if $\rho(A) = B \circledcirc (b(x) \circledcirc C)$, $b \in \mathcal{A}$ and $b = \omega(B)$,
- $\rho'(A_a) = B_a \circledcirc (b(x) \circledcirc C_b)$ if $\rho(A) = B \circledcirc (b(x) \circledcirc C)$ with $b \neq a$ or $b \notin \mathcal{A}$,
- $\rho'(A_a) = B_a \boxplus C_{\omega(B)}$ if $\rho(A) = B \boxplus C$,
- $\rho'(A_a) = B_a$ if $\rho(A) = a(x) \circledcirc B$ and $a \in \mathcal{A}$,
- $\rho'(A_a) = b(x) \circledcirc B_b$ if $\rho(A) = b(x) \circledcirc B$ with $b \neq a$ or $b \notin \mathcal{A}$,
- $\rho'(A_a) = B_a \circledcirc x \circledcirc C_a$ if $\rho(A) = a(x) \boxplus (B \circledcirc x \circledcirc C)$ with $a \in \mathcal{A}$,
- $\rho'(A_a) = b(x) \boxplus (B_b \circledcirc x \circledcirc C_b)$ if $\rho(A) = b(x) \circledcirc (B \circledcirc x \circledcirc C)$ with $b \neq a$ or $b \notin \mathcal{A}$.

An induction shows:

1. $[\![A_a]\!]_{F'} = \phi_a([\![A]\!]_F)$ for all $A \in V_0$ and $a \in \Sigma\perp$, and
2. $[\![B_a]\!]_{F'}[\phi_{\omega(B)}(f)] = \phi_a([\![B]\!]_F[f])$ for all $B \in V_1$, $a \in \Sigma_\perp$ and $f \in \mathcal{F}(\Sigma)$.

From 1 we obtain $[\![F']\!] = [\![S_\perp]\!]_{F'} = \phi_\perp([\![S]\!]_F) = \mathrm{nf}_{\mathcal{A}}([\![S]\!]_F) = \mathrm{nf}_{\mathcal{A}}([\![F]\!])$. □

To test whether two trees over Σ are equivalent with respect to commutativity, we define a *commutative normal form* $\mathrm{nf}_{\mathcal{C}}\colon \mathcal{T}(\Sigma) \to \mathcal{T}(\Sigma)$ such that $\mathrm{nf}_{\mathcal{C}}(t_1) = \mathrm{nf}_{\mathcal{C}}(t_2)$ if and only if t_1 and t_2 are equivalent with respect to the identities in (COMM) for all $c \in \mathcal{C}$. We choose to sort the trees directly below a commutative symbol length-lexicographically, using a total order on trees. This actually gives us a normal form since two trees equal modulo commutativity turn into actually equal trees. It is also extendable to FSLPs, i.e. we can transform an FSLP in polynomial time into another FSLP that produces length-lexicographically sorted forests.

Definition 44 (llex on strings)**.** Let Δ be an alphabet. Let $<$ be a strict total order on Δ. We define the *length-lexicographic* order $<_{\mathrm{llex}} \subseteq \Delta^* \times \Delta^*$ in the following way: Let $w = a_1 \dots a_n \in \Delta^n$ and $v = b_1 \dots b_m \in \Delta^m$, where $n, m \geq 0$. Set $w <_{\mathrm{llex}} v$ if either $n < m$ or $n = m$ and there is an i with $1 \leq i \leq n$ such that $a_j = b_j$ for all $1 \leq j < i$ and $a_i < b_i$.

Definition 45 (llex on forests)**.** Let $<$ be a strict total order on $\Sigma \cup \{\langle, \rangle\}$. The *length-lexicographic order* $<_{\mathrm{llex}} \subseteq \mathcal{F}(\Sigma)^2$ on forests is defined as

$$f <_{\mathrm{llex}} f' \Leftrightarrow \mathrm{dflr}(f) <_{\mathrm{llex}} \mathrm{dflr}(f').$$

Definition 46 (Sorting)**.** Let $<$ be a total order on a possibly infinite alphabet Δ, and let $\leq$ be its reflexive closure. Then we define $\mathrm{sort}^{<}\colon \Delta^* \to \Delta^*$ by $\mathrm{sort}^{<}(a_1 \cdots a_n) = a_{i_1} \cdots a_{i_n}$ with $\{i_1, \dots, i_n\} = \{1, \dots, n\}$ and $a_{i_1} \leq \cdots \leq a_{i_n}$.

Lemma 19. *Let G be an SSLP over Δ and let $<$ be some total order on Δ. We can construct in time $\mathcal{O}(|\Delta| \cdot |G|)$ an SSLP G' such that $[\![G']\!] = \mathrm{sort}^{<}([\![G]\!])$.*

Proof. Let $G = (V, \rho, S)$. We define the SSLP $G' = (V', \rho', S)$ over Δ where $V' = \{S\} \cup \{A_a \mid A \in V, a \in \Delta\}$ with new variables $A_a \notin V$, and ρ' defined by

- $\rho'(A_a) = \varepsilon$ if $\rho(A) = \varepsilon$ or $\rho(A) = b$ for some $b \in \Sigma$ with $b \neq a$,
- $\rho'(A_a) = a$ if $\rho(A) = a$,
- $\rho'(A_a) = B_a \circ C_a$ if $\rho(A) = B \circ C$,
- $\rho'(S) = S_{a_1} \circ \dots \circ S_{a_n}$ where $\Delta = \{a_1, \dots, a_n\}$ with $a_1 < \cdots < a_n$.

A straightforward induction shows that $[\![A_a]\!]_{G'} = a^{m_a}$ where m_a is the number of occurrences of a in $[\![A]\!]_G$. This implies

$$[\![G']\!] = [\![S_{a_1} \circ \dots \circ S_{a_m}]\!]_{G'} = \mathrm{sort}^{<}([\![G]\!]).$$

□

Lemma 20. *For two FSLPs F_1 and F_2 over Σ we can check in polynomial time whether $[\![F_1]\!] <_{\mathrm{llex}} [\![F_2]\!]$.*

Proof. We use Lemma 3 to construct SSLPs G_1, G_2 with $[\![G_1]\!] = \mathrm{dflr}([\![F_1]\!])$ and $[\![G_2]\!] = \mathrm{dflr}([\![F_2]\!])$. Since by definition we have $[\![G_1]\!] <_{\mathrm{llex}} [\![G_2]\!]$ if and only if $\mathrm{dflr}(f) <_{\mathrm{llex}} \mathrm{dflr}(f')$, it is enough to test if $[\![G_1]\!] <_{\mathrm{llex}} [\![G_2]\!]$, which we can do in polynomial time using [38, Lemma 3]. □

From $<_{\mathrm{llex}}$ on trees we obtain the function $\mathrm{sort}^{<_{\mathrm{llex}}}\colon \mathcal{F}(\Sigma) \to \mathcal{F}(\Sigma)$ and we define $\mathrm{nf}_{\mathcal{C}}\colon \mathcal{F}(\Sigma) \to \mathcal{F}(\Sigma)$ inductively by

$$\mathrm{nf}_{\mathcal{C}}(a\langle f\rangle) = \begin{cases} a\langle \mathrm{sort}^{<_{\mathrm{llex}}}(\mathrm{nf}_{\mathcal{C}}(f))\rangle & \text{if } a \in \mathcal{C}, \\ a\langle \mathrm{nf}_{\mathcal{C}}(f)\rangle & \text{otherwise,} \end{cases}$$
$$\mathrm{nf}_{\mathcal{C}}(t_1 \cdots t_n) = \mathrm{nf}_{\mathcal{C}}(t_1) \cdots \mathrm{nf}_{\mathcal{C}}(t_n).$$

Obviously, $f_1, f_2 \in \mathcal{F}(\Sigma)$ are equal modulo the identities in (COMM) if and only if $\mathrm{nf}_{\mathcal{C}}(f_1) = \mathrm{nf}_{\mathcal{C}}(f_2)$. Furthermore, $f_1, f_2 \in \mathcal{F}(\Sigma)$ are equal modulo the identities in (ASSOC) if and only if $\mathrm{nf}_{\mathcal{A}}(f_1) = \mathrm{nf}_{\mathcal{A}}(f_2)$. This is because the rewriting system consisting of the rules

$$a\langle t_1 \cdots t_{i-1} a\langle t_i \cdots t_{j-1}\rangle t_j \cdots t_n\rangle \to a\langle t_1 \cdots t_n\rangle, \qquad (\to\text{-ASSOC})$$

for $a \in \mathcal{A}$, $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$ and $1 \le i \le j \le n+1$, is confluent and terminating. We now show the following:

Lemma 21. *For $f_1, f_2 \in \mathcal{F}(\Sigma)$ we have $nf_{\mathcal{C}}(nf_{\mathcal{A}}(f_1)) = nf_{\mathcal{C}}(nf_{\mathcal{A}}(f_2))$ if and only if f_1 and f_2 are equal modulo the identities in* (ASSOC) *and* (COMM)*.*

Proof. Let $\to_{\mathcal{A}}$ be the resulting rewrite relation obtained from the rules of (→-ASSOC). It suffices to show that $\mathrm{nf}_{\mathcal{C}}(\mathrm{nf}_{\mathcal{A}}(f_1)) = \mathrm{nf}_{\mathcal{C}}(\mathrm{nf}_{\mathcal{A}}(f_2))$ if f_1 and f_2 can be transformed into each other by a single application of (ASSOC) or (COMM); let us write $f_1 =_{\mathcal{A}} f_2$ or $f_1 =_{\mathcal{C}} f_2$, respectively, for the latter. The case $f_1 =_{\mathcal{A}} f_2$ is clear, since this implies $\mathrm{nf}_{\mathcal{A}}(f_1) = \mathrm{nf}_{\mathcal{A}}(f_2)$. Now assume that $f_1 =_{\mathcal{C}} f_2$. The crucial observation is that $f =_{\mathcal{C}} g \to_{\mathcal{A}} h$ implies $f \to_{\mathcal{A}} g' =_{\mathcal{C}} h$ for some $g' \in \mathcal{F}(\Sigma)$ (a single application of (→-ASSOC) commutes with a permutation of the children of a node). Since $f_1 =_{\mathcal{C}} f_2 \to_{\mathcal{A}}^* \mathrm{nf}_{\mathcal{A}}(f_2)$, it follows that $f_1 \to_{\mathcal{A}}^* f_1' =_{\mathcal{C}} \mathrm{nf}_{\mathcal{A}}(f_2)$ for some $f_1' \in \mathcal{F}(\Sigma)$. But $f_1' =_{\mathcal{C}} \mathrm{nf}_{\mathcal{A}}(f_2)$ implies that f_1' is irreducible with respect to $\to_{\mathcal{A}}$, i.e., $f_1' = \mathrm{nf}_{\mathcal{A}}(f_1)$. Thus we obtain $\mathrm{nf}_{\mathcal{A}}(f_1) =_{\mathcal{C}} \mathrm{nf}_{\mathcal{A}}(f_2)$ and hence $\mathrm{nf}_{\mathcal{C}}(\mathrm{nf}_{\mathcal{A}}(f_1)) = \mathrm{nf}_{\mathcal{C}}(\mathrm{nf}_{\mathcal{A}}(f_2))$. □

From an FSLP $F = (V, \Gamma, \rho, S)$ in normal form we want to obtain in polynomial time an FSLP $F' = (V', \Gamma', \rho', S)$ with $[\![F']\!] = \mathrm{nf}_{\mathcal{C}}([\![F]\!])$. The new FSLP will fulfil that $V_0 \subseteq V_0'$, $V_1 \subseteq V_1'$ and $[\![A]\!]_{F'} = \mathrm{nf}_{\mathcal{C}}([\![A]\!]_f)$ for every $A \in V_0$. The new right-hand sides $\rho'(A)$ for all $A \in V$ will be defined by induction on $\le_F$. Before we present the details we want to point out the main difficulties.

Let $A \in V_0$ with $\rho(A) = B \otimes (a(x) \otimes C)$ and let $\mathrm{spine}_F(B) = \beta_1 \cdots \beta_N$, where $N \ge 0$. Remember that by the definition of the spine, the β_i ($1 \le i \le N$) are all expressions of the form $a(x) \boxplus (L \oslash x \oslash R)$. Also recall the definition of $A_\triangle(i)$ from Section 4.4. Let $N = \ell(A)$ and for $0 \le p \le N$ let

$$t_p := A_\triangle(p+1) = [\![\beta_{p+1} \boxplus \cdots \boxplus \beta_N \otimes (a(x) \otimes C)]\!]_F$$

be the tree which is substituted for the parameter x in β_p, in particular $t_0 = [\![A]\!]_F$ and $t_N = [\![a(x) \otimes C]\!]_F$. Note that $|t_0| > \cdots > |t_N|$ and that the length N of the spine may be exponential in $|F|$, hence we may have exponentially many different trees t_p.

Let p with $0 \le p \le N$ be some position, let $\beta_p = b(x) \boxplus (L \oslash x \oslash R)$ and

$$[\![L \circ R]\!]_{F_\boxplus} = B_1 \otimes (a_1(x) \otimes C_1) \ldots B_n \otimes (a_n(x) \otimes C_n),$$

where $n \ge 0$. By induction we may assume that

$$[\![B_i \otimes (a_i(x) \otimes C_i)]\!]_{F'} = \mathrm{nf}_{\mathcal{C}}([\![B_i \otimes (a_i(x) \otimes C_i)]\!]_F)$$

for every $1 \le i \le n$.

In case $b \notin \mathcal{C}$, we can leave β_p as is, since

$$\begin{aligned}\mathrm{nf}_{\mathcal{C}}(t_{p-1}) &= \mathrm{nf}_{\mathcal{C}}(b\langle [\![L]\!]_F t_p [\![R]\!]_F\rangle)\\ &= b\langle \mathrm{nf}_{\mathcal{C}}([\![L]\!]_F)\mathrm{nf}_{\mathcal{C}}(t_p)\mathrm{nf}_{\mathcal{C}}([\![R]\!]_F)\rangle\\ &= b\langle [\![L]\!]_{F'}\mathrm{nf}_{\mathcal{C}}(t_p)[\![R]\!]_{F'}\rangle\\ &= [\![\beta_p]\!]_{F'}[\mathrm{nf}_{\mathcal{C}}(t_p)].\end{aligned}$$

In case $b \in \mathcal{C}$ we have to replace β_p, since

$$\begin{aligned}\mathrm{nf}_{\mathcal{C}}(t_{p-1}) &= \mathrm{nf}_{\mathcal{C}}(b\langle [\![L]\!]_F t_p [\![R]\!]_F\rangle)\\ &= b\langle \mathrm{sort}^{<_{\mathrm{llex}}}(\mathrm{nf}_{\mathcal{C}}([\![L \boxminus R]\!]_F)\,\mathrm{nf}_{\mathcal{C}}(t_p))\rangle.\end{aligned}$$

Unfortunately, what to replace β_p with depends on p, since the position of $\mathrm{nf}_{\mathcal{C}}(t_p)$ in

$$\mathrm{sort}^{<_{\mathrm{llex}}}(\mathrm{nf}_{\mathcal{C}}([\![L \boxminus R]\!]_F)\,\mathrm{nf}_{\mathcal{C}}(t_p))$$

depends on p, too. We cannot replace expressions at arbitrary positions on the spine, since that might lead to an exponential blowup. Instead, we require that F already is in a form such that all occurrences of the same expression are translated in the same way, so we can translate β_p regardless of what p is. More specifically, we require that the last position is always the correct position for $\mathrm{nf}_{\mathcal{C}}(t_p)$. This will be true for FSLPs that are in strong normal form.

Let $F = (V, \Gamma, \rho, S)$ be an FSLP in normal form. For every $B \in V_1$ let

$$\mathrm{big\_args}_F(B) = \{t \in \mathcal{T}(\Sigma) \mid |t| \geq |[\![\beta \oslash \varepsilon]\!]_F| \text{ for every } \beta \text{ in } \mathrm{spine}_F(B)\}.$$

We say that F is in *strong normal form* if $[\![a(x) \oslash C]\!]_F \in \mathrm{big\_args}_F(B)$ for every $A \in V_0$ with $\rho(A) = B \oslash (a(x) \oslash C)$. If we assume that F is in strong normal form, then for all q with $\beta_q = \beta_p$ and all A' which occur in w we have

$$|\mathrm{nf}_{\mathcal{C}}(t_q)| = |t_q| \geq |[\![a(x) \oslash C]\!]_F| \geq |[\![\beta_q \oslash \varepsilon]\!]_F| > |[\![A']\!]_F| = |\mathrm{nf}_{\mathcal{C}}([\![A']\!]_F)|.$$

This means that the tree $\mathrm{nf}_{\mathcal{C}}(t_p)$ goes to the last position in

$$\mathrm{sort}^{<_{\mathrm{llex}}}(\mathrm{nf}_{\mathcal{C}}([\![L \boxminus R]\!]_F\,\mathrm{nf}_{\mathcal{C}}(t_p)).$$

Hence we can replace β_p with $b(x) \boxplus (S_w \oslash x)$, where S_w is a new variable with $[\![S_w]\!]_{F'} = \mathrm{sort}^{<_{\mathrm{llex}}}(\mathrm{nf}_{\mathcal{C}}([\![L \boxminus R]\!]_F))$. Such a variable S_w can be obtained with Lemma 20 and Lemma 19, see the proof of Theorem 4 for details. Thus we have

$$\begin{aligned}\mathrm{nf}_{\mathcal{C}}(t_{p-1}) &= b\langle \mathrm{sort}^{<_{\mathrm{llex}}}(\mathrm{nf}_{\mathcal{C}}([\![w]\!]_F))\,\mathrm{nf}_{\mathcal{C}}(t_p)\rangle\\ &= [\![b(x) \boxplus (S_w \oslash x)]\!]_{F'}[\mathrm{nf}_{\mathcal{C}}(t_p)]\\ &= [\![\beta_p]\!]_{F'}[\mathrm{nf}_{\mathcal{C}}(t_p)].\end{aligned}$$

Finally, we set $\rho'(A) = \rho(A)$. By induction for C we have $[\![C]\!]_{F'} = \mathrm{nf}_{\mathcal{C}}([\![C]\!]_F)$, so $\mathrm{nf}_{\mathcal{C}}(t_N) = \mathrm{nf}_{\mathcal{C}}([\![a(x) \oslash C]\!]_{F'})$. Thus we obtain

$$\begin{aligned}\mathrm{nf}_{\mathcal{C}}([\![A]\!]_F) &= \mathrm{nf}_{\mathcal{C}}(t_0)\\ &= [\![\beta_1 \boxplus \cdots \boxplus \beta_N]\!]_{F'}[\mathrm{nf}_{\mathcal{C}}(t_N)]\\ &= [\![\beta_1 \boxplus \cdots \boxplus \beta_N \oslash (a(x) \oslash C)]\!]_{F'}\\ &= [\![A]\!]_{F'}\end{aligned}$$

as desired.

If F is only in normal form we cannot expect that $|t_q| \geq |[\![\beta_q \otimes \varepsilon]\!]_F|$ holds for *all* indices q with $\beta_p = \beta_q$. However, this can only fail if β_q is the rightmost occurrence of β_p in the spine, i.e. $q \geq p$. Otherwise, we have $p < q$ and therefore

$$|t_q| = |[\![\beta_{q+1} \boxplus \cdots \boxplus \beta_N \otimes (a(x) \otimes C)]\!]_F| \geq |[\![\beta_p \otimes \varepsilon]\!]_F| = |[\![\beta_q \otimes \varepsilon]\!]_F|.$$

The important point is that we have at most polynomially (even linearly) many of these last positions. This gives us the idea for the proof of the following lemma:

Lemma 22. *From a given FSLP $F = (V, \Gamma, \rho, S)$ in normal form we can construct in polynomial time an FSLP $F' = (V', \Gamma', \rho', S)$ in strong normal form with $[\![F]\!] = [\![F']\!]$.*

Proof. We obtain F' from F by modifying (only) ρ of variables $A \in V_0$ with $\rho(A) = B \otimes (a(x) \otimes C)$: Let $\mathrm{spine}_F(B) = \beta_1 \cdots \beta_N$ ($N \geq 0$), and let $\{\delta_1, \ldots, \delta_m\} = \{\beta_1, \ldots, \beta_N\}$ with $\delta_i \neq \delta_j$ for all $1 \leq i \neq j \leq m$ be the set of all these expressions. For $1 \leq i \leq m$ let $p_i = \max\{1 \leq p \leq N \mid \beta_p = \delta_i\}$ be the position of the last occurrence of δ_i in $\mathrm{spine}_F(B)$. The set $\{\delta_1, \ldots, \delta_m\}$ and the positions $p_1, \ldots, p_m$ can be computed from F in polynomial time, hence we may assume w.l.o.g. that $p_m < \cdots < p_1$ by (re-)ordering the symbols δ_i in this way. This means in particular that $p_1 = N$. Additionally, we set $p_{m+1} = 0$.

The idea for the construction of F' is to divide the spine $\beta_1 \cdots \beta_N$ into the maximal spine segments which do not contain any last occurrences of the variables δ_i, i.e., the spine segments $\beta_{p_{i+1}+1} \cdots \beta_{p_i - 1}$. The details of the construction are as follows: For $1 \leq i \leq m$ we construct in polynomial time SSLPs $G_i = (N_i, \rho_i, E_i)$ with $[\![G_i]\!] = \beta_{p_{i+1}+1} \cdots \beta_{p_i - 1}$ (see e.g. [39, Lemma 1]). We may assume that the variable sets N_i are pairwise disjoint and that they only contain new variables, i.e., variables which are not in V and have not been added to V' by previous steps. Moreover, we assume that G_i is in Chomsky normal form. We add each $X \in N_i$ to the variable set V'_1 of F', set $\Gamma'(X) = \mathcal{F}_x$ and define ρ' by

- $\rho'(X) = x$ if $\rho_i(X) = \varepsilon$,
- $\rho'(X) = Y \boxplus Z$ if $\rho_i(X) = Y \circ Z$,
- $\rho'(X) = \rho(X)$ if $\rho_i(X) \in \{\delta_1, \ldots, \delta_m\}$.

By induction on $\leq_{G_i}$ we obtain $\mathrm{spine}_{F'}(X) = [\![X]\!]_{G_i}$ for every $X \in N_i$, in particular $\mathrm{spine}_{F'}(E_i) = [\![E_i]\!]_{G_i} = [\![G_i]\!] = \beta_{p_{i+1}+1} \cdots \beta_{p_i - 1}$ for every $1 \leq i \leq m$.

Now we add new variables A_i for $1 \leq i \leq m-1$ with $\Gamma'(A_i) = \mathcal{F}$ and C_i, C'_i for $1 \leq i \leq m$ with $\Gamma'(C_i) = \Gamma'(C'_i) = \mathcal{F}$ to the variable set V'_0 of F'. Additionally, we set $A_0 = C$ and $A_m = A$. Let $\delta_i = b_i(x) \otimes (L_i \oslash x \oslash R_i)$. For the new variables and for $A = A_m$ we define ρ' by

1. $\rho'(C_i) = L_i \boxminus C'_i$ and $\rho'(C'_i) = A_{i-1} \boxminus R_i$
2. $\rho'(A_i) = E_i \otimes (b_i(x) \otimes C_i)$,

for $1 \leq i \leq m$. Equation 1 means that

$$[\![b_i(x) \otimes C_i]\!]_{F'} = [\![b_i(x) \otimes (L_i \boxminus A_{i-1} \boxminus R_i)]\!]_{F'} = [\![\delta_i \otimes A_{i-1}]\!]_{F'} = [\![\beta_{p_i} \otimes A_{i-1}]\!]_{F'}.$$

From this and 2 we obtain

$$[\![A_i]\!]_{F'} = [\![\beta_{p_{i+1}+1} \boxplus \cdots \boxplus \beta_{p_i} \otimes A_{i-1}]\!]_{F'}.$$

Using induction on i and $[\![A_0]\!]_{F'} = [\![C]\!]_{F'}$ yields

$$[\![A_i]\!]_{F'} = [\![\beta_{p_{i+1}+1} \boxplus \cdots \boxplus \beta_N \circledcirc C]\!]_{F'}$$

for $0 \le i \le m$, in particular

$$[\![A]\!]_{F'} = [\![A_m]\!]_{F'} = [\![\beta_1 \boxplus \cdots \boxplus \beta_N \circledcirc C]\!]_{F'}.$$

Note that this holds for *every* $A \in V_0$ with $\rho(A) = B \circledcirc (b(x) \circledcirc C)$ for some $B, C \in V$ and $b \in \Sigma$. In addition, the right-hand sides of other variables in V are not modified. Thus we obtain $[\![A]\!]_{F'} = [\![A]\!]_F$ for every $A \in V$ by induction on $\le_F$, in particular $[\![F']\!] = [\![S]\!]_{F'} = [\![S]\!]_F = [\![F]\!]$.

It remains to be shown that F' is in strong normal form. The only variables $A' \in V_0'$ with $\rho'(A') = B \circledcirc (b(x) \circledcirc C)$ for some $B, C \in V'$ and $b \in \Sigma$ are the variables A_i ($1 \le i \le m$) with $\rho'(A_i) = E_i \circledcirc (b_i(x) \circledcirc C_i)$. Hence it suffices to prove $[\![b_i(x) \circledcirc C_i]\!]_{F'} \in \text{big\_args}_{F'}(E_i)$, i.e., $|[\![b_i(x) \circledcirc C_i]\!]_{F'}| \ge |[\![\delta_j \circledcirc \varepsilon]\!]_{F'}|$ for all δ_j that occur in $\text{spine}_{F'}(E_i)$. If $j > i$, then $p_j \le p_{i+1}$ is the last position of δ_j in $\beta_1 \cdots \beta_N$, hence D_j does *not* occur in $\text{spine}_{F'}(E_i) = \beta_{p_{i+1}+1} \cdots \beta_{p_i - 1}$. If $j \le i$, then $p_i \le p_j \le N$ and thus

$$\begin{aligned} |[\![b_i(x) \circledcirc C_i]\!]_{F'}| &= |[\![\beta_{p_i} \circledcirc A_{i-1}]\!]_{F'}| \\ &= |[\![\beta_{p_i} \boxplus \cdots \boxplus \beta_N \circledcirc C]\!]_{F'}| \\ &\ge |[\![\beta_{p_j} \circledcirc \varepsilon]\!]_{F'}| \\ &= |[\![\delta_j \circledcirc \varepsilon]\!]_{F'}|, \end{aligned}$$

which concludes the proof. □

Example 7. For every $m \ge 1$ let $F_m = (V, \Gamma, \rho, S)$ be an FSLP over $\{a\}$ with $a \in \mathcal{C}$, $|F_m| \in \mathcal{O}(m)$, $V \supseteq \{S, B, E, X\} \cup \{U_i \mid 0 \le i \le 2m\} \cup \{D_i, L_i, R_i \mid 1 \le i \le m\}$ and

$$\begin{aligned} \rho(E) &= \varepsilon, \\ \rho(X) &= x, \\ \rho(R_1) &= X \circledcirc (a(x) \circledcirc E) \\ \rho(R_i) &= R_{i-1} \boxminus R_{i-1} \text{ for } 2 \le i \le m, \\ \rho(U_0) &= a(x) \boxplus (E \circledcirc x \circledcirc E), \\ \rho(U_i) &= U_{i-1} \boxplus U_{i-1}, \text{ for } 1 \le i \le 2m, \\ \rho(L_i) &= U_{2i} \circledcirc R_1, \text{ for } 1 \le i \le m, \\ \rho(D_i) &= \delta_i = a(x) \boxplus (L_i \circledcirc x \circledcirc R_i), \text{ for } 1 \le i \le m, \\ \text{spine}_{F_m}(B) &= \delta_1^{2^{m-1}} \delta_m \delta_m \delta_1^{2^{m-2}} \delta_{m-1} \delta_{m-1} \cdots \delta_1^{2^1} \delta_2 \delta_2 \delta_1, \\ \rho(S) &= B \circledcirc R_1. \end{aligned}$$

Note that $\mathcal{O}(m)$ variables whose ρ-expressions have constant size are sufficient to produce the spine of B. We do not present them in detail because they are irrelevant for the following illustration.

Let p_i ($1 \le i \le m$) be the last position of δ_i in $\text{spine}_{F_m}(B)$, and let t_{p_i} be the tree which is substituted for x in δ_i at this last position, i.e., $t_{p_1} = [\![R_1]\!]_{F_m}$, $t_{p_2} = [\![\delta_1 \circledcirc R_1]\!]_{F_m}$ and

$$t_{p_{i+1}} = [\![\underbrace{\delta_1 \boxplus \cdots \boxplus \delta_1}_{2^{i-1}} \boxplus \delta_i \boxplus \delta_i \circledcirc t_{p_i}]\!]_{F_m}$$

for $2 \le i \le m-1$. Let

$$u_i := [\![U_i]\!]_{F_m} = \underbrace{a\langle \dots a\langle}_{2^i} x\rangle \dots\rangle \text{ for } 0 \le i \le 2m,$$
$$\ell_i := [\![L_i]\!]_{F_m} = u_{2i}\langle a\rangle = \underbrace{a\langle \dots a\langle}_{4^i} a\rangle \dots\rangle \text{ for } 1 \le i \le m,$$
$$r_i := [\![R_i]\!]_{F_m} = a^{2^{i-1}} \text{ for } 1 \le i \le m.$$

For $1 \le i \le m$ we have $|[\![D_i]\!]_{F_m}[t]| = |a\langle \ell_i t r_i\rangle| = |t| + 4^i + 2^{i-1} + 2$ for every $t \in \mathcal{T}(\{a\})$, hence $|t_{p_{i+1}}| = |t_{p_i}| + 2\cdot(4^i + 2^{i-1} + 2) + 2^{i-1}\cdot 7 \le |t_{p_i}| + 3\cdot 4^i$ for every $i \ge 3$. By induction on i this implies $|t_{p_i}| \le 4^i < |\ell_i|$ and thus $\mathrm{nf}_{\mathcal{C}}(t_{p_i}) <_{\mathrm{llex}} \ell_i = \mathrm{nf}_{\mathcal{C}}(\ell_i)$ for $1 \le i \le m$, which explains the shape of the tree $\mathrm{nf}_{\mathcal{C}}([\![F_3]\!])$ in Figure 3.

By the construction of Lemma 22 we obtain the strong normal form FSLP F'_m from F_m by modifying (only) the right-hand sides of the variables S and $L_1, \dots, L_m$. The modification of $\rho(L_i)$ is easy since $\rho(L_i) = U_{2i} \otimes R_1$ and only the variable U_0 occurs in the spine of U_{2i}. Hence we focus on the modification of $\rho(S) = B \otimes R_1$.

The spine of B contains all the expressions $\delta_1, \dots, \delta_m$ and they are already ordered in such a way that $p_m < \dots < p_1$ holds for their last positions p_i. Hence we obtain $F'_m = (V', \Gamma', \rho', S)$ with

$$V' \supseteq V \cup \{C_i, C'_i \mid 1 \le i \le m\} \cup \{E_i \mid 2 \le i \le m\} \cup \{A_i \mid 1 \le i < m\},$$

$A_0 = R_1$, $A_m = S$ and

$$\begin{aligned}
\mathrm{spine}_{F'_m}(E_i) &= \delta_1^{2^{i-1}}\delta_i, \text{ for } 2 \le i \le m,\\
\rho'(A_i) &= E_i \otimes (a(x) \otimes C_i) \text{ for } 1 \le i \le m,\\
\rho'(C_i) &= L_i \boxminus C'_i \text{ for } 1 \le i \le m,\\
\rho'(C'_i) &= A_{i-1} \boxminus R_i \text{ for } 1 \le i \le m.
\end{aligned}$$

Note that $\mathrm{spine}_{F'_m}(E_i)$ is the spine segment between the last positions of δ_{i+1} and δ_i and that $\mathrm{spine}_{F'_m}(E_1) = \varepsilon$.

The case $m = 3$ is illustrated in Figure 2 and Figure 3. We have

$$\mathrm{spine}_{F_3}(B) = \delta_1^4\delta_3\delta_3\delta_1^2\delta_2\delta_2\delta_1,$$

hence $|\mathrm{spine}_{F_3}(B)| = 11$. The positions of the last occurrences of $\delta_1, \delta_2, \delta_3$ in the spine are $p_1 = 11$, $p_2 = 10$ and $p_3 = 6$, respectively. These are the occurrences that are replaced by C_1, C_2 and C_3. We therefore obtain

$$\begin{aligned}
[\![C_1]\!]_{F'_3} &= [\![L_1 \boxminus A_0 \boxminus R_1]\!]_{F'_3},\\
\mathrm{spine}_{F'_3}(E_1) &= \varepsilon\\
\rho'(A_1) &= E_1 \otimes (a(x) \otimes C_1),\\
[\![C_2]\!]_{F'_3} &= [\![L_2 \boxminus A_1 \boxminus R_2]\!]_{F'_3},\\
\mathrm{spine}_{F'_3}(E_2) &= \delta_1^2\delta_2,\\
\rho'(A_2) &= E_2 \otimes (a(x) \otimes C_2),\\
[\![C_3]\!]_{F'_3} &= [\![L_3 \boxminus A_2 \boxminus R_3]\!]_{F'_3},\\
\mathrm{spine}_{F'_3}(E_3) &= \delta_1^4\delta_3,\\
\rho'(A_3) &= E_3 \otimes (a(x) \otimes C_3).
\end{aligned}$$

This concludes our example.

It remains to be shown how an FSLP F in strong normal form can be turned into an FSLP F' with $[\![F']\!] = \mathrm{nf}_{\mathcal{C}}([\![F]\!])$. We have already given an outline of this construction. Now we present the details.

Theorem 4. *From a given FSLP F we can construct in polynomial time an FSLP F' with $[\![F']\!] = nf_{\mathcal{C}}([\![F]\!])$.*

Proof. Let $F = (V, \Gamma, \rho, S)$. By Theorem 1 and Lemma 22 we may assume that F is already in strong normal form. We want to construct an FSLP $F' = (V', \Gamma', \rho', S)$ with $V_0 \subseteq V_0'$ and $V_1 = V_1'$, so for all $A \in V$ we have $\Gamma'(A) = \Gamma(A)$, such that

1. $[\![A]\!]_{F'} = \mathrm{nf}_{\mathcal{C}}([\![A]\!]_F)$ for all $A \in V_0$,
2. $[\![A]\!]_{F'}[\mathrm{nf}_{\mathcal{C}}(t)] = \mathrm{nf}_{\mathcal{C}}([\![A]\!]_F[t])$ for all $A \in V_1$, $t \in \mathrm{big\_args}_F(A)$.

From 1 we obtain $[\![F']\!] = [\![S]\!]_{F'} = \mathrm{nf}_{\mathcal{C}}([\![S]\!]_F) = \mathrm{nf}_{\mathcal{C}}([\![F]\!])$ which will be enough to conclude the proof. To define ρ', let

$$\begin{aligned} V^c = \{A \in V_0 \mid \rho(A) = B \circledcirc (a(x) \circledcirc C),\ a \in \mathcal{C},\ B, C \in V\} \\ \cup \{A \in V_1 \mid \rho(A) = a(x) \boxplus (L \circledcirc x \circledcirc R),\ a \in \mathcal{C},\ L, R \in V_0\} \end{aligned}$$

be the set of *commutative variables* of F. We set $\rho'(A) = \rho(A)$ for every $A \in V \setminus V^c$. For $A \in V^c$ we define $\rho'(A)$ by induction on $\leq_F$:

- If $\rho(A) = B \circledcirc (a(x) \circledcirc C)$ with $a \in \mathcal{C}$, let $w = [\![C]\!]_{F_\boxminus}$. This is a string of the form
$$w = B_1 \circledcirc (a_1(x) \circledcirc C_1) \cdots B_n \circledcirc (a_n(x) \circledcirc C_n) \in (\Sigma_\boxminus)^n,$$
where $n \geq 0$. Let $M_A \subseteq \Sigma_\boxminus$ be the set of symbols that appear in w. By induction, $\rho'(D)$ and hence $[\![D]\!]_{F'}$ are already defined for every $D \in \{B_1, \ldots, B_n, C_1, \ldots, C_n\}$. Therefore, by Lemma 20, we can compute in polynomial time a total order $<$ on M_A such that $\delta < \delta'$ implies $[\![\delta]\!]_{F'} \leq_{\mathrm{llex}} [\![\delta']\!]_{F'}$ for all $\delta, \delta' \in M_A$. By Lemma 19, we can construct in linear time an SSLP $G_w = (V_w, \rho_w, S_w)$ with $[\![G_w]\!] = \mathrm{sort}^<(w)$, and we may assume that all variables $D \in V_w$ are new. We add these variables D to V_0', set $\Gamma'(D) = \mathcal{F}$, and define $\rho'(D)$ as follows:
  - $\rho'(D) = \varepsilon$ if $\rho_w(D) = \varepsilon$,
  - $\rho'(D) = B \circledcirc (a(x) \circledcirc C)$ if $\rho_w(D) = B \circledcirc (a(x) \circledcirc C)$ for some variables B, C and $a \in \Sigma$, and
  - $\rho'(D) = L \boxminus R$ if $\rho_w(D) = L \circ R$ for some variables L, R.

  Finally, we set $\rho'(A) = B \circledcirc (a(x) \circledcirc S_w)$. We obtain that $[\![S_w]\!]_{G_w} = \alpha_1 \cdots \alpha_n \in M_A^n$ if and only if $[\![S_w]\!]_{F'} = \mathrm{sort}^{<_{\mathrm{llex}}}([\![\alpha_1]\!]_{F'} \cdots [\![\alpha_n]\!]_{F'})$, since $\alpha_i < \alpha_j$ implies $[\![\alpha_i]\!]_{F'} <_{\mathrm{llex}} [\![\alpha_j]\!]_{F'}$ for every $1 \leq i, j \leq m$.
- If $\rho(A) = a(x) \boxplus (L \circledcirc x \circledcirc R)$ with $a \in \mathcal{C}$, then define $G_w = (V_w, \rho_w, S_w)$ as before, but with $w = [\![L \circ R]\!]_{F_\boxminus}$ instead of $w = [\![C]\!]_{F_\boxminus}$, and set $\rho'(A) = a(x) \boxplus (S_w \circledcirc x)$.

Properties 1 and 2 are proved by induction on $\leq_F$. We only consider the interesting cases, i.e., those in which $<_{\mathrm{llex}}$ plays a role.

1. $A \in V_0$ with $\rho(A) = B \circledcirc (a(x) \circledcirc C)$ with $B \in V_0$ and $a \in \mathcal{C}$:

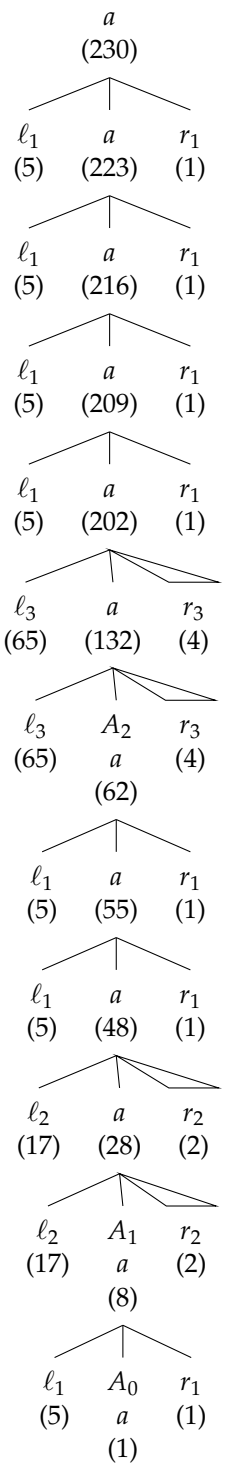


Figure 2: The tree $[\![F_3]\!] = [\![F_3']\!]$ for the FSLP F_3 from Example 7. The size of each subtree is written in ( ) after the node label. See Figure 3 for $\mathrm{nf}_{\mathcal{C}}([\![F_3]\!]) = \mathrm{nf}_{\mathcal{C}}([\![F_3']\!])$.

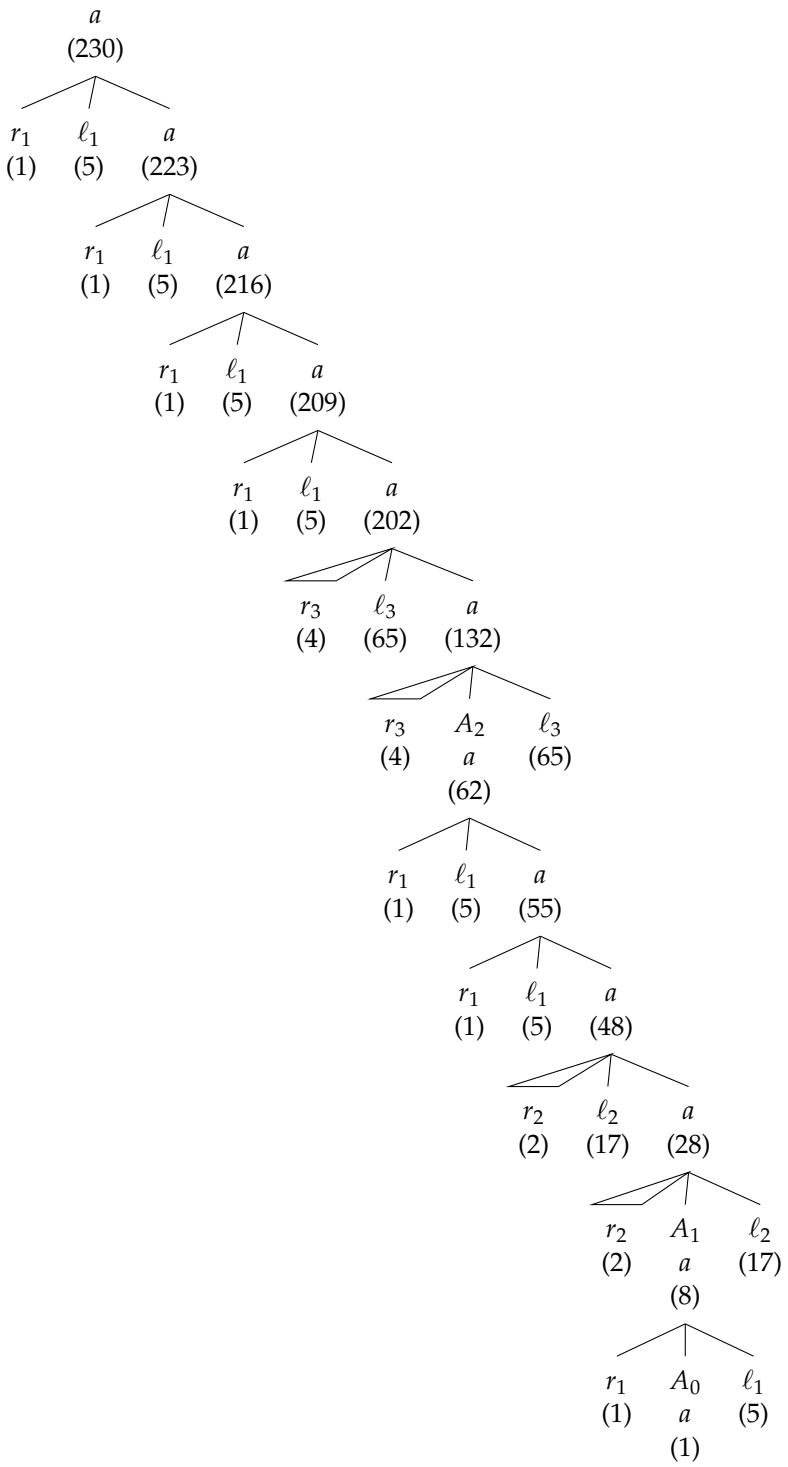


Figure 3: The tree $\mathrm{nf}_{\mathcal{C}}([\![F_3]\!]) = \mathrm{nf}_{\mathcal{C}}([\![F_3']\!])$ for the FSLP F_3 from Example 7. The size of each subtree is written in ( ) after the node label. The last positions of $\delta_1, \delta_2, \delta_3$ are $p_1 = 11, p_2 = 10$ and $p_3 = 6$, respectively. The roots of the trees t_{p_i} and $\mathrm{nf}_{\mathcal{C}}(t_{p_i})$ are marked with A_i since $[\![A_i]\!]_{F_3'} = t_{p_i}$. They are the only argument trees which are smaller than their siblings ℓ_i, hence $\mathrm{nf}_{\mathcal{C}}(t_{p_i})$ goes to the left of ℓ_i in $\mathrm{nf}_{\mathcal{C}}([\![F_3]\!])$ and all the others go to the right.

Let $w = [\![C]\!]_{F_\boxminus} = \alpha_1 \dots \alpha_m$ with $m \geq 0$. By definition of the strong normal form we have $[\![a(x) \circledcirc C]\!]_F \in \text{big\_args}_F(B)$. We have

$$\begin{aligned}
\text{nf}_{\mathcal{C}}([\![a(x) \circledcirc C]\!]_F) &= \text{nf}_{\mathcal{C}}(a\langle [\![C]\!]_F \rangle) \\
&= a\langle \text{sort}^{<_{\text{llex}}}(\text{nf}_{\mathcal{C}}([\![C]\!]_F)) \rangle \text{ since } a \in \mathcal{C} \\
&= a\langle \text{sort}^{<_{\text{llex}}}(\text{nf}_{\mathcal{C}}([\![\alpha_1]\!]_F) \dots \text{nf}_{\mathcal{C}}([\![\alpha_m]\!]_F)) \rangle \\
&= a\langle \text{sort}^{<_{\text{llex}}}([\![\alpha_1]\!]_{F'} \dots [\![\alpha_m]\!]_{F'}) \rangle \text{ by induction} \\
&= a\langle [\![S_w]\!]_{F'} \rangle \\
&= [\![a(x) \circledcirc S_w]\!]_{F'}.
\end{aligned}$$

From this we obtain

$$\begin{aligned}
\text{nf}_{\mathcal{C}}([\![A]\!]_F) &= \text{nf}_{\mathcal{C}}([\![B \circledcirc (a(x) \circledcirc C)]\!]_F) \\
&= \text{nf}_{\mathcal{C}}([\![B]\!]_F[[\![a(x) \circledcirc C]\!]_F]) \\
&= [\![B]\!]_{F'}[\text{nf}_{\mathcal{C}}([\![a(x) \circledcirc C]\!]_F)] \text{ by induction for } B \\
&= [\![B]\!]_{F'}[[\![a(x) \circledcirc S_w]\!]_{F'}] \\
&= [\![B \circledcirc (a(x) \circledcirc S_w)]\!]_{F'} \\
&= [\![A]\!]_{F'}.
\end{aligned}$$

2. $A \in V_1$ with $\rho(A) = a(x) \boxplus (B \oslash x \oslash C)$:

   Let $w = [\![B \circ C]\!]_{F_\boxminus} = \alpha_1 \cdots \alpha_m$ with $m \geq 0$, $[\![B]\!]_{F_\boxminus} = \alpha_1 \cdots \alpha_k$ and $[\![C]\!]_{F_\boxminus} = \alpha_{k+1} \cdots \alpha_m$ with $0 \leq k \leq m$. For every $t \in \text{big\_args}_F(A)$ and $1 \leq i \leq m$ we have

$$|\text{nf}_{\mathcal{C}}(t)| = |t| \geq |[\![A \circledcirc \varepsilon]\!]_F| > |[\![B \boxminus C]\!]_F| \geq |[\![\alpha_i]\!]_F| = |\text{nf}_{\mathcal{C}}([\![\alpha_i]\!]_F)|,$$

   which implies $\text{nf}_{\mathcal{C}}([\![\alpha_i]\!]_F) \leq_{\text{llex}} \text{nf}_{\mathcal{C}}(t)$. Hence we obtain

$$\begin{aligned}
\text{nf}_{\mathcal{C}}([\![A]\!]_F[t]) &= \text{nf}_{\mathcal{C}}(a\langle [\![B]\!]_F t [\![C]\!]_F \rangle) \\
&= a\langle \text{sort}^{<_{\text{llex}}}(\text{nf}_{\mathcal{C}}([\![B]\!]_F t [\![C]\!]_F)) \rangle \text{ since } a \in \mathcal{C} \\
&= a\langle \text{sort}^{<_{\text{llex}}}(\text{nf}_{\mathcal{C}}([\![\alpha_1]\!]_F \cdots [\![\alpha_k]\!]_F t [\![\alpha_{k+1}]\!]_F \cdots [\![\alpha_m]\!]_F)) \rangle \\
&= a\langle \text{sort}^{<_{\text{llex}}}(\text{nf}_{\mathcal{C}}([\![\alpha_1]\!]_F) \cdots \text{nf}_{\mathcal{C}}([\![\alpha_k]\!]_F) \text{nf}_{\mathcal{C}}(t) \\
&\qquad \text{nf}_{\mathcal{C}}([\![\alpha_{k+1}]\!]_F) \cdots \text{nf}_{\mathcal{C}}([\![\alpha_m]\!]_F)) \rangle \\
&= a\langle \text{sort}^{<_{\text{llex}}}(\text{nf}_{\mathcal{C}}([\![\alpha_1]\!]_F) \cdots \text{nf}_{\mathcal{C}}([\![\alpha_m]\!]_F)) \text{nf}_{\mathcal{C}}(t) \rangle \\
&\quad \text{since } \text{nf}_{\mathcal{C}}([\![\alpha_i]\!]_F) \leq_{\text{llex}} \text{nf}_{\mathcal{C}}(t) \text{ for } 1 \leq i \leq m \\
&= a\langle [\![S_w]\!]_{F'} \text{nf}_{\mathcal{C}}(t) \rangle \\
&\quad \text{since } [\![S_w]\!]_{F'} = \text{sort}^{<_{\text{llex}}}(\text{nf}_{\mathcal{C}}([\![\alpha_1]\!]_F) \cdots \text{nf}_{\mathcal{C}}([\![\alpha_m]\!]_F)) \text{ (1)} \\
&= [\![a(x) \boxplus (S_w \oslash x)]\!]_{F'}[\text{nf}_{\mathcal{C}}(t)] \\
&= [\![A]\!]_{F'}[\text{nf}_{\mathcal{C}}(t)] \text{ by definition of } \rho'(A).
\end{aligned}$$

3. $A \in V_1$ with $\rho(A) = B \boxplus C$:

   Then $\rho'(A) = B \boxplus C$. Let $t \in \text{big\_args}_F(A) \subseteq \text{big\_args}_F(B) \cap \text{big\_args}_F(C)$. Since $|[\![C]\!]_F[t]| \geq |t|$, we obtain $[\![C]\!]_F[t] \in \text{big\_args}_F(B)$ and thus

$$\begin{aligned}
\text{nf}_{\mathcal{C}}([\![A]\!]_F[t]) &= \text{nf}_{\mathcal{C}}([\![B]\!]_F[[\![C]\!]_F[t]]) \\
&= [\![B]\!]_{F'}[\text{nf}_{\mathcal{C}}([\![C]\!]_F[t])] \text{ by induction for } B \\
&= [\![B]\!]_{F'}[[\![C]\!]_{F'}[\text{nf}_{\mathcal{C}}(t)]] \text{ by induction for } C \\
&= [\![A]\!]_{F'}[\text{nf}_{\mathcal{C}}(t)].
\end{aligned}$$

This concludes the proof of the theorem. □

Example 8. Let F be the strong normal form FSLP $F'_m = (V', \Gamma', \rho', S)$ $(m \geq 1)$, which we obtained in Example 7. From F we construct a new FSLP which produces $\mathrm{nf}_{\mathcal{C}}([\![F]\!])$. Again, we will only consider the spine of B and ignore the spines of the U_{2i}. We have to replace the right-hand sides of A_i and D_i for $1 \leq i \leq m$. We have

$$w = [\![C_i]\!]_{F_{\boxminus}} = [\![L_i \circ A_{i-1} \circ R_i]\!]_{F_{\boxminus}} = \rho(L_i)\rho(A_{i-1})\rho(R_1)^{2^{i-1}}.$$

Let $<$ be the total order on $M_{C_i} \supseteq \{\rho(L_i), \rho(A_{i-1}), \rho(R_1)\}$ from Theorem 4. Since $[\![A_{i-1}]\!]_F = t_{p_i}$ and $[\![R_1]\!]_F <_{\mathrm{llex}} t_{p_i} <_{\mathrm{llex}} [\![L_i]\!]_F$ for $2 \leq i \leq m$, we must have that $\rho(R_1) < \rho(A_{i-1}) < \rho(L_i)$. For $i = 1$ we have a special case because $R_1 = A_0$, for which we have $[\![A_0]\!]_F <_{\mathrm{llex}} [\![L_1]\!]_F$, so $\rho(R_1) = \rho(A_0) < \rho(L_1)$. Altogether, we obtain

$$\mathrm{sort}^{<}(w) = \rho(R_1)^{2^{i-1}}\rho(A_{i-1})\rho(L_i).$$

Using Lemma 19, we introduce an SLP $G_w = (V_w, \rho_w, S_w)$ with $[\![G_w]\!] = \mathrm{sort}^{<}(w)$. Finally, we set the new right-hand side of A_i to $E_i \mathbin{\textcircled{\scriptsize :}} (a(x) \mathbin{\textcircled{\scriptsize :}} S_w)$.

For the right-hand sides $\rho'(D_i) = a(x) \boxplus (L_i \circledcirc x \circledcirc R_i)$ for $1 \leq i \leq m$ we have

$$w = [\![L_i \circ R_i]\!]_{F_{\boxminus}} = \rho(L_i)\rho(R_1)^{2^{i-1}}.$$

Let $<$ be the total order on $M_{D_i} \supseteq \{\rho(R_1), \rho(L_i)\}$ from Theorem 4. Since $[\![R_1]\!]_F <_{\mathrm{llex}} [\![L_i]\!]_F$ for $1 \leq i \leq m$, we must have that $\rho(R_1) < \rho(L_i)$, and thus $\mathrm{sort}^{<}(w) = \rho(R_1)^{2^{i-1}}\rho(L_i)$. Since F is in strong normal form, x always goes to the last position. We introduce $G_w = (V_w, \rho_w, S_w)$ with $[\![G_w]\!] = \mathrm{sort}^{<}(w)$ and set the new right-hand side of D_i to $a(x) \boxplus (S_w \circledcirc x)$. This concludes our example.

As an immediate consequence of Theorem 4 we obtain our main result.

Theorem 5. *For trees s, t we can test in polynomial time whether s and t are equal modulo the identities in* (ASSOC) *and* (COMM)*, if s and t are given succinctly by one of the following formalisms: FSLPs, top dags or TSLPs for their fcns encodings.*

Proof. By Proposition 5 and Proposition 7 it suffices to show Theorem 5 for the case that t_1 and t_2 are given by FSLPs F_1 and F_2, respectively. By Lemma 21 and Lemma 20 it suffices to compute in polynomial time FSLPs F'_1 and F'_2 for $\mathrm{nf}_{\mathcal{C}}(\mathrm{nf}_{\mathcal{A}}(t_1))$ and $\mathrm{nf}_{\mathcal{C}}(\mathrm{nf}_{\mathcal{A}}(t_2))$. This can be achieved using Lemma 18 and Theorem 4. □

7 UNORDERED FORESTS

Understanding the interplay between ordered and unordered structures is an important topic of database research. For XML this interplay has received considerable attention, see, e.g., [1, 9, 48, 7, 47]. A document is deemed *document-centric*, if the order of elements matters. Examples of such documents include web pages (e.g., in XHTML). In contrast, a document is *data-centric* if the order of elements is unimportant. For instance, the order of author-, title-, and year-elements in a bibliographic entry is unimportant. Of course, there could be mixtures of both, unordered and ordered nodes. For instance, an author-node could be marked "ordered" to contain subtrees for the first author, second author, etc. JSON naturally supports ordered nodes (arrays) and unordered nodes (objects). The absence of order bears many opportunities such as query optimization and set-oriented parallel processing, cf. [1]. Unordered XML has also been studied recently with respect to schema language definitions [9], a topic already considered during the birth years of XML [43]. Here we study the question whether *forest compression* can benefit from unorderedness. In XML compression, document trees are typically stored (and compressed) separately from the data values, see, e.g., [33]. It was observed early on that DAGs provide high compression ratios for common XML document trees [12] (10% on average for their documents).

Intuitively, considering a forest to be unordered means that the order of trees in a subforest does not matter. For example, the unordered forest $a\langle bc\rangle$ is considered to be the same unordered forest as $a\langle cb\rangle$. We have already looked at this in Chapter 6 where we studied commutative symbols. Here, we make every symbol commutative, which allows us to reorder every subforest. We formally define an unordered forest as the set of all of its reorderings, or equivalently, all forests that have the same commutative normal form:

Definition 47 (Unordered forests)**.** Let $\mathcal{C} = \Sigma$. For $f \in \mathcal{F}(\Sigma)$ let

$$f^u = \{f' \in \mathcal{F}(\Sigma) \mid \mathrm{nf}_{\mathcal{C}}(f) = \mathrm{nf}_{\mathcal{C}}(f')\}.$$

For example, $(a\langle bc\rangle)^u = \{a\langle bc\rangle, a\langle cb\rangle\}$. We use a similar definition for trees with maximal rank $\mathcal{T}_k(\Sigma)$ and ranked trees $\mathcal{T}(\Sigma, r)$, since the number of children does not change by reordering them, i.e. for every ranked alphabet (Σ, r), where $r\colon \Sigma \to \mathbb{N}$, and every $t \in \mathcal{T}(\Sigma, r)$ we have $t^u \subseteq \mathcal{T}(\Sigma, r)$ and for every $k \in \mathbb{N}$ and $t \in \mathcal{T}_k(\Sigma)$ we have $t^u \subseteq \mathcal{T}_k(\Sigma)$.

Our question is how much we can gain if we allow to compress a reordered version of a forest instead of the original one. Let us first define this using DAGs.

Definition 48 (Unordered DAGs)**.** We define the set $\mathrm{dag}(t)$ for a given tree $t \in \mathcal{T}(\Sigma)$ to contain all DAGs D with $[\![D]\!] = t$. We also define the set $\mathrm{dag}^u(t)$ that contains all DAGs for *all reorderings* of t, i.e.

$$\mathrm{dag}^u(t) = \{D \in \mathrm{dag}(t') \mid t' \in t^u\}.$$

For a DAG $D = (V, \rho)$, we look at the following two size measures:

- The number of its variables, $|D|_V = |V|$. This is the same as the number of nodes in the graph representation of a DAG.
- The number of edges in its graph representation, which is basically the same as the number of subexpressions. For $A \in V$ where $\rho(A) = f(A_1, \ldots, A_n)$ we define $|A|_E = n$. We also define $|D|_E = \sum\{|A|_E \mid A \in V\}$.

We are interested in what the smallest DAGs according to these size measures are. Let $t \in \mathcal{T}(\Sigma)$. We define

$$\begin{aligned}
\min|\mathrm{dag}(t)|_V &= \min\{|D|_V \mid D \in \mathrm{dag}(t)\},\\
\min|\mathrm{dag}(t)|_E &= \min\{|D|_E \mid D \in \mathrm{dag}(t)\},\\
\min|\mathrm{dag}^u(t)|_V &= \min\{|D|_V \mid D \in \mathrm{dag}^u(t)\},\\
\min|\mathrm{dag}^u(t)|_E &= \min\{|D|_E \mid D \in \mathrm{dag}^u(t)\}.
\end{aligned}$$

Since we are sometimes only interested in the size of an alphabet but not its elements, we introduce the following shorthand notation: For $r, k \in \mathbb{N}$ let $\mathcal{T}_{r,k} = \mathcal{T}_r(\{1, \ldots, k\})$ and $\mathcal{T}_{\infty,k} = \mathcal{T}(\{1, \ldots, k\})$. The trees in $\mathcal{T}_{\infty,1}$ are also called *unlabelled*. Formally, their nodes are all labelled with 1, but we will simply write $\langle f \rangle$ instead of $1\langle f \rangle$.

With $e = 2{,}71828\ldots$ we always denote Euler's constant and with $\ln n$ (resp. $\log n$) we denote the logarithm of n to base e (resp., 2).

For every $t \in \mathcal{T}(\Sigma)$, since $t \in t^u$, we have

$$\begin{aligned}
\min|\mathrm{dag}^u(t)|_E &\le \min|\mathrm{dag}(t)|_E,\\
\min|\mathrm{dag}^u(t)|_V &\le \min|\mathrm{dag}(t)|_V.
\end{aligned}$$

In this section we study the question of how much smaller a DAG for an unordered version of a tree can be compared to a DAG of the original one. Formally, we study the growth of the following two worst case ratios, where $n, r, k \in \mathbb{N}$ and $r, k \le n$:

$$\begin{aligned}
\alpha_E(n,r,k) &= \max\left\{ \frac{\min|\mathrm{dag}(t)|_E}{\min|\mathrm{dag}^u(t)|_E} \,\middle|\, t \in \mathcal{T}_{r,k},\ |t| \le n \right\},\\
\alpha_N(n,r,k) &= \max\left\{ \frac{\min|\mathrm{dag}(t)|_V}{\min|\mathrm{dag}^u(t)|_V} \,\middle|\, t \in \mathcal{T}_{r,k},\ |t| \le n \right\}.
\end{aligned}$$

Let $x \in \{N, E\}$. Note that $\alpha_x(n,1,k) = 1$ since each $t \in \mathcal{T}_{1,k}$ is a linear chain, and therefore nothing can be reordered. Hence, we only consider the ratio $\alpha_x(n,r,k)$ for $r \ge 2$. Note that $\alpha_x(n,r,k) \le \alpha_x(n,r',k')$ for all $r, r', k, k' \le n$ with $r \le r'$ and $k \le k'$, since $\mathcal{T}_{r,k} \subseteq \mathcal{T}_{r',k'}$. In the following we mainly concentrate on the extreme cases $\alpha_x(n,2,1)$ and $\alpha_x(n,n,n)$, which we abbreviate by $\alpha_x(n) = \alpha_x(n,n,n)$.

We will show the following bounds:

$$\begin{aligned}
\alpha_N(n,2,1) &= \Theta\left(\frac{n}{\log n}\right), & \alpha_N(n) &= \Theta\left(\frac{n \cdot \log\log n}{\log n}\right),\\
\alpha_E(n,2,1) &= \Theta\left(\frac{n}{\log n}\right), & \alpha_E(n) &= \Theta\left(\frac{n}{\log n}\right).
\end{aligned}$$

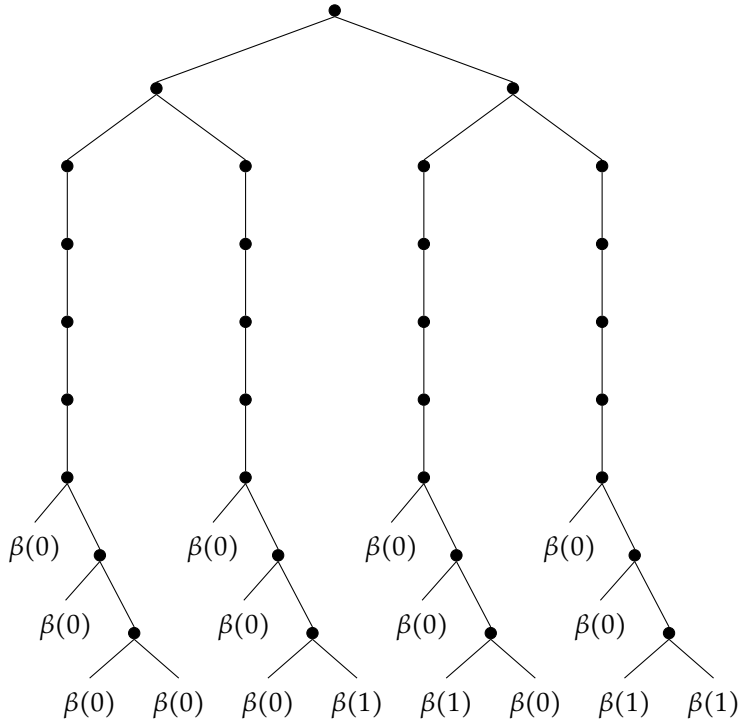


Figure 1: A possible choice for the tree t_{16} from Lemma 23 with $n = 16$, $r = h = 4$ and $k = 2$.

7.1 LOWER BOUNDS FOR NODES/EDGES OF DAGS

In this section, we prove two lower bounds. In the first part we derive a lower bound of $\Omega(n/\log n)$ for $\alpha_E(n, 2, 1)$. For this, we construct a family of binary trees, where DAGs achieve almost no compression, but by reordering the trees we achieve an exponential compression ratio. Later, we show that this bound is tight by providing a matching upper bound for $\alpha_E(n)$. Getting a lower bound on unlabelled binary trees also gives us a lower bound for all $\alpha_E(n, r, k)$ where $r, k \in \mathbb{N}$. Furthermore, it gives us a lower bound on $\alpha_N(n, r, k)$ because $\alpha_E(n, 2, 1) \in \Theta(\alpha_N(n, 2, 1))$, since for binary trees it does not matter whether we count the number of variables in a DAG or the number of its edges. This is because by the definition of a DAG, all variables must be reachable from the start variable. Therefore, in the following theorem and its proof, we only consider trees from $\mathcal{T}_{2,1}$ (binary unlabelled trees). By $B_h \in \mathcal{T}_{2,1}$ we denote the complete unlabelled binary tree with 2^h leaves and height h. This tree has size $2^{h+1} - 1$ and its minimal DAG has $\Theta(h)$ variables. For a tree t with k leaves and trees $t_1, \ldots, t_k$ we write $t[t_1, \ldots, t_k]$ to denote the tree obtained from t by replacing the i-th leaf (in pre-order) by t_i. For $k \geq 1$ let $c_k = \langle^k\rangle^k$ denote a chain of k nodes. We encode non-empty bit strings by trees from $\mathcal{T}_{2,1}$ using the function β that is inductively defined as follows: $\beta(0) = \langle\langle\rangle\langle\langle\rangle\rangle\rangle$, $\beta(1) = \langle\langle\langle\rangle\rangle\langle\rangle\rangle$, and $\beta(ds) = \langle\beta(d)\beta(s)\rangle$ for $d \in \{0, 1\}$ and $s \in \{0, 1\}^+$. The construction in the proof of the following theorem is similar to a construction from [22].

Lemma 23. *For every $n \geq 2$ there exists a tree $t_n \in \mathcal{T}_{2,1}$ with*

- $|t_n| \in \Theta(n)$,
- $h(t_n) \in \Theta(\log n)$,

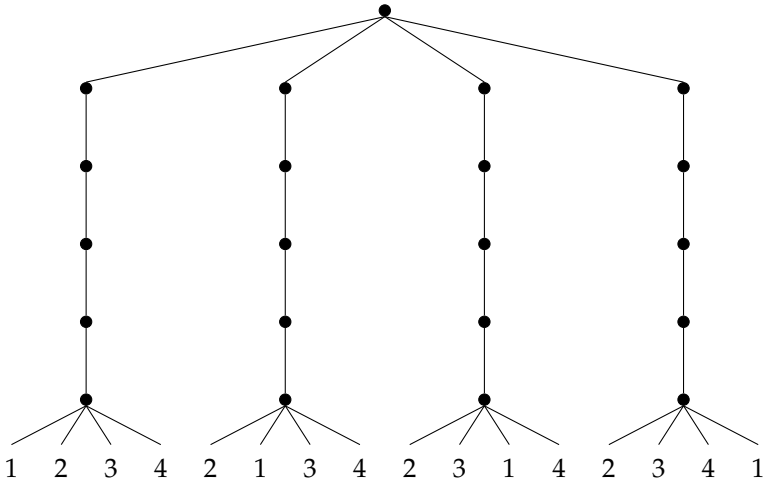


Figure 2: A possible choice for the tree t_{16} from Theorem 6 with $n = 16$, $x = \frac{1}{2}\log\log n = 1$, and $r = k = 4$.

- $\min|\mathrm{dag}(t_n)|_E \in \Theta(n)$,
- $\min|\mathrm{dag}^u(t_n)|_E \in \Theta(\log n)$.

Hence, we have $\alpha_E(n,2,1) \in \Omega(n/\log n)$ *and* $\alpha_N(n,2,1) \in \Omega(n/\log n)$.

Proof. Let $n \in \mathbb{N}$ and $h = \lceil \log n \rceil$. Let $r = 2^k \in \Theta(n/\log n)$ be the smallest power of two that is at least n/h. Let $u_1, \ldots, u_r$ be r distinct bit strings of length h (note that $r \le n \le 2^h$). Consider the trees $s_1 = \beta(u_1), \ldots, s_r = \beta(u_r)$. We add to s_i a chain of length h and obtain the tree $s_i' = c_h[s_i]$ $(1 \le i \le r)$. Finally, set $t_n = B_k[s_1', \ldots, s_r']$. A possible choice for the tree t_{16} is shown in Figure 1.

Let us first bound the size of t_n. For B_k we have $|B_k| \in \Theta(r) = \Theta(n/\log n)$. The total size of all r copies of the chain c_h is $\Theta(r \cdot h) = \Theta(n)$. Finally, every s_i has size $\Theta(h)$, so their sizes sum up to $\Theta(r \cdot h) = \Theta(n)$. Altogether, we get $|t_n| \in \Theta(n)$. For the height we have $h(B_k) \in \Theta(\log r) = \Theta(\log n)$, $h(c_h) = h \in \Theta(\log n)$ and $h(\beta(u_i)) \in \Theta(\log(h)) = \Theta(\log\log n)$ for $1 \le i \le r$, so altogether $h(t_n) \in \Theta(\log n)$. To bound $\min|\mathrm{dag}(t_n)|_E$, note that the trees $s_1, \ldots, s_r$ are pairwise different. This implies that the r copies of the chains c_h cannot be identified. Therefore, every DAG has at least $r \cdot h \in \Theta(n)$ many variables, so $\min|\mathrm{dag}(t_n)|_V \in \Theta(n)$. Since every node of t_n has rank at most 2, we also get $\min|\mathrm{dag}(t_n)|_E \in \Theta(n)$. However, we can reorder all $s_1, \ldots, s_r$ into the same tree s, which also lets us reorder the $c_k[s_i]$ $(1 \le i \le r)$ into the same tree $c' = c_k[s]$. This lets us reorder t_n into the tree $B_k[c', \ldots, c']$, where c' appears r times. A DAG for this tree can represent the B_k part with $\Theta(\log r) = \Theta(\log n)$ variables, the c_k part with $k \in \Theta(\log n)$ variables and s with $\Theta(h) = \Theta(\log n)$ variables. We therefore get $\min|\mathrm{dag}^u(t_n)|_V \in \Theta(\log n)$ and since t_n is a binary tree we also get $\min|\mathrm{dag}^u(t_n)|_E \in \Theta(\log n)$. □

In the next section, we will prove $\alpha_E(n) \in \mathcal{O}(n/\log n)$, which yields the same upper bound for $\alpha_E(n,2,1)$. Moreover, also the lower bound of $\Omega(n/\log n)$ for $\alpha_N(n,2,1)$ turns out be sharp (see Corollary 5 for $k = 1$). On the other hand, for $\alpha_N(n) = \alpha_N(n,n,n)$ we can improve the lower bound to $\Omega(n \log\log n/\log n)$:

Theorem 6. *Fix a constant* $\delta > 1$. *For every* $n \ge 1$ *large enough (depending on* δ*) there exists a tree* $t_n \in \mathcal{T}_{r,k}$ *with the following properties:*

- $k = \lceil \delta \cdot \log n / \log\log n \rceil$ *and* $r \in \Theta(n \cdot \log\log n / \log n)$,
- $|t_n| \in \Theta(n)$,
- $\min|\mathrm{dag}(t_n)|_V \in \Theta(n)$,
- $\min|\mathrm{dag}^u(t_n)|_V \in \Theta(\log n / \log\log n)$.

Hence, we have $\alpha_N(n) \in \Omega(n \cdot \log\log n / \log n)$.

Proof. Fix $n \geq 1$ and let $x = \frac{1}{\delta} \log\log n$, $k = \lceil (\log n)/x \rceil = \lceil \delta \cdot \log n / \log\log n \rceil$, and $r = \lfloor n/k \rfloor \in \Theta(n \cdot \log\log n / \log n)$. Let us first show that with this choice we have $k! \geq r$. With Stirling's formula (or, more precisely, the inequality $z! \geq \sqrt{2\pi z} \cdot (z/e)^z$) we get

$$\begin{aligned} k! \geq (k/e)^k &\geq (\log n / ex)^{(\log n)/x} \\ &= 2^{(\log\log n - \log(ex))(\log n)/x} \\ &= n^{(\log\log n - \log(ex))/x}. \end{aligned}$$

Since moreover $n \geq n/k \geq r$, it suffices to show

$$n^{(\log\log(n) - \log(ex))/x} \geq n,$$

i.e.,

$$\log\log n - \log(ex) \geq x = \frac{1}{\delta} \log\log n,$$

or, equivalently

$$\left(1 - \frac{1}{\delta}\right) \log\log n \geq \log(ex) = \log(e/\delta) + \log\log\log n,$$

which holds for n large enough. This shows that, indeed, $k! \geq r$.

We now construct the tree $t_n \in \mathcal{T}_{r,k}$ as follows: take r many pairwise different trees $s_1, \ldots, s_r$ consisting of a root node with k many children, which are leaves. The sequence of labels of these k leaves forms a permutation of $\{1, \ldots, k\}$. Since $k! \geq r$, these r pairwise different trees exist. From s_i we next construct s_i' by adding the chain c_k on top of s_i. Finally, the tree t_n is obtained by taking a new root node, whose children are the roots of the trees $s_1', \ldots, s_r'$. A possible choice for the tree t_{16} is shown in Figure 2. Note that for n large enough we have $k \leq r$ since $k \in \Theta(\log n / \log\log n)$ and $r \in \Theta(n \cdot \log\log n / \log n)$. Hence, $t_n \in \mathcal{T}_{r,k}$.

We get $|t_n| = 1 + 2rk \in \Theta(n)$. For the number of variables of the DAGs we obtain $\min|\mathrm{dag}(t_n)|_V \in \Theta(1 + rk + k) \subseteq \Theta(n)$, since the root node, the root nodes of s_i and the k leaves have to remain distinct in every DAG for t_n. Finally, note that all $s_1, \ldots, s_r$ can be reordered into the same tree, and therefore the $c_k[s_i]$ $(1 \leq i \leq r)$ can be reordered into the same tree. We therefore obtain that

$$\min|\mathrm{dag}^u(t_n)|_V \in \Theta(k) = \Theta(\log n / \log\log n),$$

which proves the statement. □

7.2 UPPER BOUND FOR EDGES OF DAGS

In this section we prove an upper bound for $\alpha_E(n)$ via a lower bound of $\Omega(\log n)$ for the function

$$\mu(n) := \min\left\{\min|\mathrm{dag}^u(t)|_E \mid t \in \mathcal{T}_{n,n},\ |t| = n\right\}.$$

Thus, if we take all trees of size n and all DAGs for all of their reorderings, the smallest of them has size $\Theta(\log n)$. Note that for binary trees this is obvious since the height of such a tree is at least $\log n$ and the number of edges of a DAG is at least the height of the tree. Also note that

$$\mu(n) = \min\left\{\min|\mathrm{dag}(t)|_E \mid t \in \mathcal{T}_{n,n},\ |t| = n\right\}.$$

since the set of all reorderings of all trees of size n is the same as the set of all trees of size n. Moreover, it holds that

$$\mu(n) = \min\left\{\min|\mathrm{dag}(t)|_E \mid t \in \mathcal{T}_{n,1},\ |t| = n\right\},$$

i.e., it suffices to consider unlabelled trees. This is because adding labels to a tree can make the minimal DAG only larger. Let $D = (V, \rho, S)$ be a DAG for such a tree. We define the existence of a *path of length* k, where $k \in \mathbb{N}$, between $A, B \in V$ as follows: A *path of length* 1 exists from $A \in V$ to $B \in V$ if B appears in $\rho(A)$ as a subexpression (remember that DAGs have to be in Chomsky normal form, so a path of length 1 from A to B means that $[\![B]\!]_D$ is a direct subtree of $[\![A]\!]_D$). A *path of length* $k + k'$, where $k, k' \in \mathbb{N}$ and $k, k' \geq 1$, exists from A to B if there is a $C \in V$ such that there is a path of length k from A to C and a path of length k' from C to B. Furthermore, every $A \in V$ has a path of length 0 to itself. We consider the *depth* $d(A)$ of a variable $A \in V$ which is defined as the length of a longest path from S to A (remember that DAGs cannot have variables that are not reachable from S, so this is well-defined). Thus $d(S) = 0$ and $h(D) = \max\{d(A) \mid A \in V\}$. For $1 \leq i \leq h(D) + 1$ let $V_i(D) = \{A \in V \mid d(A) = i - 1\}$ be the set of variables at depth $i - 1$. Finally, let $\rho_i(D) = \sum\{|\rho(A)|_E \mid A \in V_i(D)\}$ for $1 \leq i \leq h(D)$. This is the total number of edges that start in a node at depth $i - 1$. Every such edge goes to a node at depth $j \geq i$. We write V_i and ρ_i for $V_i(D)$ and $\rho_i(D)$, respectively, if D is clear from the context.

Lemma 24. *Let $D = (V, \rho, S)$ be a DAG of height $h = h(D)$. The number of leaves of $[\![D]\!]$ is bounded by $\prod_{i=1}^{h} \rho_i$.*

Proof. Consider the DAG $D' = (\{1, \ldots, h+1\}, \rho', 1)$ with $[\![D']\!] \in \mathcal{T}_{\infty,1}$. Formally, this is a tree labelled with 1, but for clarity, let us set $a = 1$. Let $\rho'(i) = a_{\rho_i}(i+1, \ldots, i+1)$ (a node labelled with a that has ρ_i many children, which are all $i+1$) for $1 \leq i \leq h$ and $\rho'(h+1) = a_0$. The tree $[\![D']\!]$ is a chain of $h+1$ nodes with ρ_i many edges from node i to node $i+1$, which means that $[\![D']\!]$ has $\prod_{i=1}^{h} \rho_i$ many leaves. It therefore suffices to transform D into D' and show that this transformation does not reduce the number of leaves of $[\![D]\!]$.

First of all, we can identify in D all variables A with $\rho(A) = a_0$. This does not change $[\![D]\!]$, $h(D)$, nor ρ_i for any $1 \leq i \leq h+1$. Hence, V_{h+1} consists of the unique sink node of D; let us call this variable s. Next, we construct from D the DAG $D_1 = (V, \rho_1, S)$, where ρ_1 is defined as follows: For s we set $\rho_1(s) = a_0$. Now, let $A \in V_i$ with $1 \leq i \leq h$ and $\rho(A) = a_r(A_1, \ldots, A_r)$. We set $\rho_1(A) = a_r(A'_1, \ldots, A'_r)$, where the variables A'_j are defined as follows: if $A_j \in V_{i+1}$ then set $A'_j = A_j$. Otherwise, i.e., if $A_j \in V_k$ with $k > i+1$, then let A'_j be a variable in V_{i+1} such that there exists a path from A'_j to A_j. Note that such a variable A'_j exists, since every variable in V_k ($k > 1$) has a predecessor in V_{k-1}. Note that $[\![A_j]\!]$ is a subtree of $[\![A'_j]\!]$. Therefore, $[\![D_1]\!]$ has indeed at least as many leaves as $[\![D]\!]$.

The DAG D_1 has still height h and $\rho_i(D_1) = \rho_i(D)$ for $1 \leq i \leq h$. But in contrast to D, each variable $A \in V_i$ only uses variables from V_{i+1} in $\rho(A)$

$(1 \le i \le h)$. Moreover, no variable $A \in V_i$ $(1 \le i \le h)$ has $\rho_1(A) = a_0$. If we now merge all variables in V_i into a single variable, we obtain (up to naming of variables) the DAG D'. Clearly, this merging increases the number of paths from the root S to the sink s. But the number of such paths is exactly the number of leaves in $[\![D]\!]$. This shows the lemma. □

Theorem 7. *We have* $\mu(n) \ge \frac{e}{2} \cdot \ln(n/2)$.

Proof. Let t be an arbitrary (unlabelled tree) of size n. We first transform t into a new tree t' by adding exactly one additional child node to every non-leaf of t. These new children are leaves in t'. Now t' has the property that every non-leaf node has at least two children. Note that $|t| \le |t'| \le 2|t|$ and therefore $\min|\mathrm{dag}(t)|_E \in \Theta(\min|\mathrm{dag}(t')|_E)$. Let ℓ be the number of leaves of t'. Since every non-leaf node of t' has at least two children, we have $\ell \ge |t'|/2 \ge n/2$. Moreover, let h be the height of t', let $D \in \mathrm{dag}(t')$ and let $\rho_i = \rho_i(D)$. From Lemma 24 we obtain $\ell \le \prod_{i=1}^{h} \rho_i$. On the other hand, we have $|D|_E = \sum_{i=1}^{h} \rho_i$. The well-known inequality between the arithmetic and geometric mean (see [24]) states that for all $x_1, \ldots, x_m \in \mathbb{R}$,

$$\frac{1}{m} \cdot \sum_{i=1}^{m} x_i \ge \left(\prod_{i=1}^{m} x_i \right)^{1/m}.$$

Applying this to the numbers ρ_i $(1 \le i \le h)$, we get

$$|D|_E = \sum_{i=1}^{h} \rho_i \ge h \cdot \left(\prod_{i=1}^{h} \rho_i \right)^{1/h} \ge h \cdot \ell^{1/h}.$$

To further bound the term $h \cdot \ell^{1/h}$, we consider it as a function of h: let $f(x) = x \cdot \ell^{1/x}$. Its derivative is

$$f'(x) = \ell^{1/x} \left(1 - \frac{\ln(\ell)}{x} \right).$$

Therefore $f(x)$ has a minimum at $x = \ln \ell$ in the interval $(0, \infty)$, from which it follows that

$$h \cdot \ell^{1/h} \ge \ell^{1/\ln(\ell)} \cdot \ln \ell = e \cdot \ln \ell.$$

With $\ell \ge n/2$ we finally get

$$\min|\mathrm{dag}(t)|_E \ge \frac{1}{2} \cdot |d|_E \ge \frac{e}{2} \cdot \ln \ell \ge \frac{e}{2} \cdot \ln(n/2).$$

□

For every tree t of size n we have $\min|\mathrm{dag}(t)|_E \le n$. Moreover, by Theorem 7 it holds that $\min|\mathrm{dag}^u(t)|_E \ge \frac{e}{2} \cdot \ln(n/2)$. Hence, we obtain

$$\frac{\min|\mathrm{dag}(t)|_E}{\min|\mathrm{dag}^u(t)|_E} \le \frac{2n}{e \cdot \ln(n/2)} \in \Theta(n/\log n),$$

which is stated in the next corollary.

Corollary 4. *It holds that* $\alpha_E(n) \in O(n/\log n)$.

In this section, we derive an upper bound on the node size of the minimal DAG. We use $\log(k+1)$ instead of $\log k$ in order to avoid $\log k = 0$ for $k = 1$.

Theorem 8. *For every tree $t \in \mathcal{T}_{n,k}$ of size n and height h, it holds that*

$$\min|\mathrm{dag}(t)|_V \in \mathcal{O}\left(\frac{n \cdot h \cdot \log(k+1)}{\log n}\right).$$

Proof. Let $t \in \mathcal{T}_{n,k}$ be a tree of size n and height h. Note that $\min|\mathrm{dag}(t)|_V$ is the number of different subtrees of t. Let t' be the tree that is obtained from t by removing all maximal subtrees of size at most

$$m := \frac{1}{2} \cdot \log_{4k} n = \frac{\log n}{2 \cdot \log 4k}.$$

Let f be the forest consisting of all these removed subtrees. Then the number of different subtrees of t (i.e., $\min|\mathrm{dag}(t)|_V$) is bounded by $|t'|$ plus the number of different subtrees in f. But the latter is bounded by the number of trees $s \in \mathcal{T}_{\infty,k}$ with $|s| \le m$, which by [22, Lemma 1] is at most $\frac{4}{3}(4k)^m = \frac{4}{3}n^{1/2}$.

Let us now bound $|t'|$. Consider a leaf v of t'. Then, the subtree of t rooted in v must have size larger than m; otherwise v would not belong to t'. Therefore, t' has at most n/m many leaves. Clearly, if every internal node in t' would have at least two children, then we could conclude that t' has at most $2n/m$ many nodes. But t' may contain nodes with a single child. Let us call such nodes *unary*. Moreover, let ℓ be the length of a longest path in t' in which all nodes except the last one are unary. Then, we get $|t'| \le 2(\ell+1)n/m \le 2(h+1)n/m$. In total, we get

$$\begin{aligned}\min|\mathrm{dag}(t)|_V &\le \frac{2(h+1)n}{m} + \frac{4}{3} \cdot n^{1/2} \\ &= \frac{4 \cdot (h+1) \cdot n \cdot \log 4k}{\log n} + \frac{4}{3} \cdot n^{1/2} \in \mathcal{O}\left(\frac{n \cdot h \cdot \log(k+1)}{\log n}\right).\end{aligned}$$

□

Corollary 5. *It holds that $\alpha_N(n,n,k) \in \mathcal{O}\left(\frac{n \cdot \log(k+1)}{\log n}\right)$ and $\alpha_N(n) \in \mathcal{O}\left(\frac{n \cdot \log\log n}{\log n}\right)$.*

Proof. Let us first show $\alpha_N(n,n,k) \in \mathcal{O}\left(\frac{n \cdot \log(k+1)}{\log n}\right)$. Let $t \in \mathcal{T}_{\infty,k}$ be a tree of size n and height h. By Theorem 8 we have

$$\min|\mathrm{dag}(t)|_V \in \mathcal{O}\left(\frac{n \cdot h \cdot \log(k+1)}{\log n}\right).$$

On the other hand, we clearly have $\min|\mathrm{dag}^u(t)|_V \ge h$. Therefore, we get

$$\frac{\min|\mathrm{dag}(t)|_V}{\min|\mathrm{dag}^u(t)|_V} \in \mathcal{O}\left(\frac{n \cdot \log(k+1)}{\log n}\right).$$

Let us now prove $\alpha_N(n) \in \mathcal{O}\left(\frac{n \cdot \log\log n}{\log n}\right)$. Consider an arbitrary tree $t \in \mathcal{T}_{\infty,n}$ of size n. If more than $\log n$ labels occur in t, then we clearly have $\min|\mathrm{dag}^u(t)|_V > \log n$. Since $\min|\mathrm{dag}(t)|_V \le n$ we get (for n large enough)

$$\frac{\min|\mathrm{dag}(t)|_V}{\min|\mathrm{dag}^u(t)|_V} \le \frac{n}{\log n} \le \frac{n \cdot \log\log n}{\log n}.$$

On the other hand, if at most $\log n$ many different labels occur in t then the bound $\alpha_N(n,n,k) \in \mathcal{O}\left(\frac{n\cdot\log(k+1)}{\log n}\right)$ implies

$$\frac{\min|\mathrm{dag}(t)|_V}{\min|\mathrm{dag}^u(t)|_V} \in \mathcal{O}\left(\frac{n\cdot\log\log n}{\log n}\right).$$

This proves the bound $\alpha_N(n) \in \mathcal{O}\left(\frac{n\cdot\log\log n}{\log n}\right)$. □

7.4 FSLPs for unordered forests

For a forest $f \in \mathcal{F}(\Sigma)$ we write $\min|\mathrm{FSLP}(f)|$ for the size of a smallest FSLP for f and $\min|\mathrm{FSLP}^u(f)|$ for the size of a smallest FSLP for any of the unordered versions of f. Let $k \in \mathbb{N}$. Similar to trees, we write $\mathcal{F}_k = \mathcal{F}(\{1,\ldots,k\})$. The forests from $\mathcal{F}_k$ that use exactly k many different labels are denoted with $\mathcal{F}_{=k}$. For $n \in \mathbb{N}$ we define

$$\alpha_{\mathrm{FSLP}}(n) = \max\left\{\frac{\min|\mathrm{FSLP}(f)|}{\min|\mathrm{FSLP}^u(f)|} \mid f \in \mathcal{F}_n,\ |f| = n\right\}.$$

Theorem 9.

$$\alpha_{\mathrm{FSLP}}(n) \in \Theta\left(\frac{n\cdot\log\log n}{\log^2 n}\right).$$

We prove the upper bound in Lemma 25 and the lower bound in Lemma 27.

Lemma 25.

$$\alpha_{\mathrm{FSLP}}(n) \in \mathcal{O}\left(\frac{n\cdot\log\log n}{\log^2 n}\right).$$

Proof. For $k,n \in \mathbb{N}$ with $k \le n$ we define

$$\widehat{\alpha}(n,k) = \max\left\{\frac{\min|\mathrm{FSLP}(f)|}{\min|\mathrm{FSLP}^u(f)|} \mid f \in \mathcal{F}_{=k},\ |f| = n\right\}.$$

Note that $\alpha_{\mathrm{FSLP}}(n) = \max\{\widehat{\alpha}(n,i) \mid 0 \le i \le n\}$. To show the lemma, we prove that for all $n,k \in \mathbb{N}$, $k \le n$ and $n \ge 2$ we have

$$\widehat{\alpha}(n,k) \in \mathcal{O}\left(\frac{n\cdot\log\log n}{\log^2 n}\right).$$

Let $k' = \max\{k,2\}$. For every $f \in \mathcal{F}_{=k}$ with $|f| = n$ we have that

$$\min|\mathrm{FSLP}(f)| \in \mathcal{O}\left(\frac{n}{\log_{k'} n}\right) = \mathcal{O}\left(\frac{n\cdot\log k'}{\log n}\right).$$

On the other hand, we also have

$$\min|\mathrm{FSLP}^u(f)| \in \Omega(\max\{k,\log n\}).$$

Therefore, we get

$$\widehat{\alpha}(n,k) \in \mathcal{O}\left(\frac{n\cdot\log k'}{\log n\cdot\max\{k,\log n\}}\right).$$

In case $k \le \log_2 n$, so $k' \in \mathcal{O}(\log n)$, we obtain

$$\widehat{\alpha}(n,k) \in \mathcal{O}\left(\frac{n\cdot\log\log n}{\log^2 n}\right).$$

In case $k > \log_2 n$ we have $k \geq 2 > \log_2 2$. Since the function $f(x) = \log_2 x/x$ is monotonically decreasing in the interval $[2, \infty]$ we have $f(k) \leq f(\log_2 n)$, so $f(k) \in \mathcal{O}(f(\log n))$ and thus

$$\widehat{\alpha}(n,k) \in \mathcal{O}\left(\frac{n \cdot \log k}{\log n \cdot k}\right) \subseteq \mathcal{O}\left(\frac{n \cdot \log\log n}{\log n \cdot \log n}\right).$$

□

Lemma 26. *Let $k \in \mathbb{N}$ and $A \subseteq \mathcal{F}_k$ be a finite, non empty set. There is a forest $f \in A$, where $m = \min|\mathrm{FSLP}(f)|$, such that $m \cdot \log m \in \Omega(\log|A|)$.*

Proof. For $f \in \mathcal{F}_k$ let $M(f)$ be a minimal FSLP for f. We then define the set $B = \{M(f) \mid f \in A\}$. Since all elements from A must have different minimal FSLPs we have $|A| = |B|$. Consider an injective function $c\colon B \to \{0,1\}^*$ (which is a binary encoding of these FSLPs). Let $b = \max\{|w| \mid w \in \mathrm{Img}(c)\}$. We have $\mathrm{Img}(c) \subseteq \{0,1\}^{\leq b}$ and since c is injective we also have $|B| \leq |\{0,1\}^{\leq b}|$ and thus $b \in \Omega(\log|B|)$. This means that there is a forest $f \in A$ such that $M(f)$ needs $b \in \Omega(\log|B|) = \Omega(\log|A|)$ many bits to encode. Let m be the size of this FSLP. Encoding an FSLP G over an arbitrary alphabet is possible with $\mathcal{O}(|G| \cdot \log|G|)$ many bits, so $m \cdot \log m \in \Omega(b)$ and thus $m \cdot \log m \in \Omega(\log|A|)$. □

Lemma 27.

$$\alpha_{\mathrm{FSLP}}(n) \in \Omega\left(\frac{n \cdot \log\log n}{\log^2 n}\right).$$

Proof. Let $n \in \mathbb{N}$, $n \geq 2$ and $k = \lfloor \log_2 n \rfloor$ (so $k \geq 1$). Let $\Gamma \subseteq \mathcal{F}_k$ be the set of forests that are permutations of $\{1, \ldots, k\}$, i.e. $f \in \Gamma$ if and only if $f = x_1 \ldots x_k$ with $\{x_1, \ldots, x_k\} = \{1, \ldots, k\}$ for some $x_1, \ldots, x_k \in \{1, \ldots, k\}$. We have that $|\Gamma| = k! = \lfloor \log_2 n \rfloor!$ and every $f \in \Gamma$ has size $|f| = k$. Let $m = \lfloor n/\log_2 n \rfloor$. Consider the forests

$$A = \{1\langle f_1\rangle \ldots 1\langle f_m\rangle \in \mathcal{F}_k \mid f_1, \ldots, f_m \in \Gamma\}.$$

Let $f \in A$, so $f = 1\langle f_1\rangle \ldots 1\langle f_m\rangle$ for some $f_1, \ldots, f_m \in \Gamma$. We have that $|f| \in \Theta(n)$. Furthermore, f can be reordered into, say, $(1\langle f_1\rangle)^m$, because the f_i ($1 \leq i \leq m$) are permutations of each other. For the reordered version we can implement an FSLP of size $\Theta(k + \log m) = \Theta(\log n)$, which means that $\min|\mathrm{FSLP}^u(f)| \in \mathcal{O}(\log n)$. We now show that there is an $f \in A$ with $\min|\mathrm{FSLP}(f)| \in \Omega(n \cdot \log\log n/\log n)$. We have $|A| = (\lfloor \log_2 n \rfloor!)^{\lfloor \frac{n}{\log_2 n} \rfloor}$, so

$$\log|A| \in \Theta\left(\frac{n}{\log n} \cdot \log((\log n)!)\right).$$

By Stirling's formula we have for every $x \in \mathbb{N}$ that $\log(x!) \in \Theta(x \cdot \log x)$, so

$$\log|A| \in \Theta\left(\frac{n \cdot \log n \cdot \log\log n}{\log n}\right) = \Theta(n \cdot \log\log n).$$

Using Lemma 26 we obtain that there is a forest $f \in A$ such that

$$\min|\mathrm{FSLP}(f)| \cdot \log \min|\mathrm{FSLP}(f)| \in \Omega(\log|A|) = \Omega(n \cdot \log\log n).$$

We have $\min|\mathrm{FSLP}(f)| \in \mathcal{O}(n)$, so $\log \min|\mathrm{FSLP}(f)| \in \mathcal{O}(\log n)$ and thus

$$\min|\mathrm{FSLP}(f)| \in \Omega\left(\frac{n \cdot \log\log n}{\log \min|\mathrm{FSLP}(f)|}\right) \subseteq \Omega\left(\frac{n \cdot \log\log n}{\log n}\right),$$

which together with $\min|\mathrm{FSLP}^u(f)| \in \mathcal{O}(\log n)$ finishes the proof. □

For $k \in \mathbb{N}$ let $\mathcal{T}_{[k]} = \bigcup\{\mathcal{T}(\{1,\dots,k\}, r) \mid r\colon\{1,\dots,k\} \to \mathbb{N}\}$, which is the set of all ranked trees that use at most k many different labels. The trees from $\mathcal{T}_{[k]}$ that use exactly k many different labels are denoted with $\mathcal{T}_{=k}$. For a tree $t \in \mathcal{T}_{[k]}$ let $\min|\mathrm{TSLP}(t)|$ be the size of a minimal TSLP for t and let $\min|\mathrm{TSLP}^u(t)|$ be the size of a minimal TSLP for one of the reorderings of t. For $n \in \mathbb{N}$ we define

$$\alpha_{\mathrm{TSLP}}(n) = \max\left\{\frac{\min|\mathrm{TSLP}(t)|}{\min|\mathrm{TSLP}^u(t)|} \mid t \in \mathcal{T}_{[n]},\ |t| = n\right\}.$$

Theorem 10.

$$\alpha_{\mathrm{TSLP}}(n) \in \Theta\left(\frac{n \cdot \log\log n}{\log^2 n}\right).$$

Proof. For the upper bound we basically use the same argument as for FSLPs: For $k, n \in \mathbb{N}$ with $k \le n$ we define

$$\widehat{\alpha}_{\mathcal{T}}(n,k) = \max\left\{\frac{\min|\mathrm{TSLP}(t)|}{\min|\mathrm{TSLP}^u(t)|} \mid t \in \mathcal{T}_{=k},\ |t| = n\right\}.$$

As for FSLPs, we have for all $n, k \in \mathbb{N}$, $k \le n$ and $n \ge 2$ that

$$\widehat{\alpha}_{\mathcal{T}}(n,k) \in \mathcal{O}\left(\frac{n \cdot \log\log n}{\log^2 n}\right).$$

This again follows from the fact that every tree $t \in \mathcal{T}_{=k}$ of size $|t| = n$ can be represented by a TSLP of size $\mathcal{O}(n/\log_k n)$, while on the other hand every TSLP for one of t's unordered versions needs at least size $\max\{k, \log n\}$.

For the lower bound we basically change all the forests into their fcns form. Let $n \in \mathbb{N}$, $n \ge 2$ and $k = \lfloor \log_2 n \rfloor$ (so $k \ge 1$). Let $\sigma = \{1,\dots,k\} \cup \{\overline{k}, \overline{2}\}$ where $r\colon\sigma \to \mathbb{N}$ is defined by $r(\overline{i}) = i$ for $i \in \{k, 2\}$ and $r(a) = 0$ for all $a \in \{1,\dots,k\}$. Let $\Gamma \subseteq \mathcal{T}(\sigma, r)$ be the set of trees that have their root node labelled with $\overline{k}$ and the children of the root node are a permutation of $\{1,\dots,k\}$. Let $m = \lfloor n/\log_2 n \rfloor$. Consider the trees

$$A = \{2\langle t_m, \cdots 2\langle t_2, t_1\rangle \cdots\rangle \in \mathcal{T}(\sigma, r) \mid t_1, \dots, t_m \in \Gamma\}.$$

Lemma 26 works the same for TSLPs, which is why we obtain that there is a tree $t \in A$ such that every TSLP for t has size at least $\Omega(n \cdot \log\log n/\log n)$. On the other hand, we can reorder every $t \in A$ into, say, $2\langle t_1, \cdots 2\langle t_1, t_1\rangle \cdots\rangle$, where t_1 appears m times. This tree has a TSLP of size $\mathcal{O}(k + \log m) = \mathcal{O}(\log n)$. □

7.6 EXPERIMENTAL RESULTS

We contrast our theoretical results by experimental data for two corpora of XML trees. In addition to minimal DAGs, we are interested in experiments to measure the impact that unorderedness has on other tree compression methods. The ones we test are the DAG variants introduced in [10] and the grammar-based tree compressor "TreeRePair" [37]. These are the strongest tree compressors that we are aware of. Instead of using these compressors on a tree t directly, we apply them to its *canonical tree* $\mathsf{canon}(t)$. The tree $\mathsf{canon}(a\langle t_1 \dots t_n\rangle)$ is obtained by sorting the trees $\mathsf{canon}(t_1), \dots, \mathsf{canon}(t_n)$ according to their size, and in case of equal sizes, according to the lexicographical order of their traversal strings (see for example [14]). This is the

same as setting $\mathcal{C} = \Sigma$ and defining $\mathsf{canon}(t) = \mathrm{nf}_{\mathcal{C}}(t)$ from Chapter 6. We clearly have $\mathsf{canon}(t) \in t^u$ for all trees t. Additionally, for all trees s, t we have $s^u = t^u$ if and only if $\mathsf{canon}(s) = \mathsf{canon}(t)$. It should be understood that our experiment only gives a rough indication of the benefit of unorderedness for compressors other than the DAG. We expect that a more careful adaptation of those compressors to unordered trees will provide stronger compression. We only report number of edges, so "size" in this section always refers to number of edges.

We compare seven known tree compressors, which are considered in [10]:

1. minimal DAG,
2. minimal binary DAG,
3. minimal reverse binary DAG,
4. minimal hybrid DAG,
5. minimal reverse hybrid DAG,
6. DS, and
7. TreeRePair.

We choose these compressors, since they all produce a graph-based representation of the input tree. This makes the output sizes of the compressors comparable.

The *minimal binary DAG (bdag)* of a tree t is the minimal DAG of the fcns encoding of t. The *minimal reverse binary DAG (rbdag)* is the minimal DAG of the "first-child/previous-sibling encoding" (fcps), defined in the obvious way. Binary DAGs and reverse binary DAGs share end- and begin-sequences, respectively, of subtrees. This implies that both the bdag and rbdag of a canonical tree can be *larger* than the corresponding minimal DAG of the original tree. As an example consider the following tree

$$t = f\langle g\langle cdbah\rangle g\langle cdb\rangle g\langle bdcdb\rangle\rangle.$$

This tree has 16 edges. Its minimal binary DAG has only 14 edges, because the end-sequence of subtrees "cdb" occurs twice and can be shared. Similarly, the minimal reverse binary DAG has size 14 (because "cdb" appears twice). In contrast, the canonical tree of t

$$\mathsf{canon}(t) = f\langle g\langle bcd\rangle g\langle abcdh\rangle g\langle bbcdd\rangle\rangle$$

has a bdag and rbdag of 16 edges. Interestingly, such scenarios where bdag and rbdag become larger for the canonical tree appear frequently in practice.

The *hybrid DAG (hdag)* (and *reverse hybrid DAG (rhdag)*) were introduced in [10] as data structures that are guaranteed to be smaller than or equal in size to both the DAG and the bdag (rbdag) of an unranked tree. The hdag (resp., rhdag) is obtained from a DAG by applying the fcns encoding (resp., fcps-encoding) to the rules of the DAG (where the DAG is viewed as a regular tree grammar), and then computing the minimal DAG of the resulting forest of encoded rules; see [10] for a precise definition. Similarly as with bdag and rbdag, the hdag and rhdag can be *larger* for the canonical tree than for the original one.

The acronym *DS* stands for "DAG and string compression". The idea is to compute a minimal DAG and to then apply a string compressor to the

Corpus	Docs	Edges	aD	mD	aR	mR
I	21	$3.1 \cdot 10^6$	6.6	36	5.7	$3.9 \cdot 10^6$
II	1131	79465	7.9	65	6.0	2925

Table 1: Document characteristics, Edges = average number of edges in a tree, Docs = number of documents, aD = average depth of a node, mD = maximum depth of a node in any tree, aR = average rank of a node, mR = maximum rank of a tree

above mentioned rules of the DAG. As in [10], we use RePair [32] as our string compressor. Finally, *TR* refers to the grammar-based tree compressor TreeRePair of [37]. The sizes are numbers of edges of the compressed structures, see [10] for details.

We use two different Corpora of XML documents. These corpora were also used in [10]. For each document we consider the unranked tree of its element nodes, i.e., we ignore attribute and text values. *Corpus I* consists of XML documents from the web which are often used in XML compression research. Many of the files of this corpus can be downloaded from the XMLCompBench site[1] (see [10] for details). *Corpus II* is a subset of files from the *University of Amsterdam XML Web Collection*[2]. We have verified by hand that, according to the tag names, all of the documents in Corpus I appear to be order independent. By sampling Corpus II we also did not find order dependent documents.

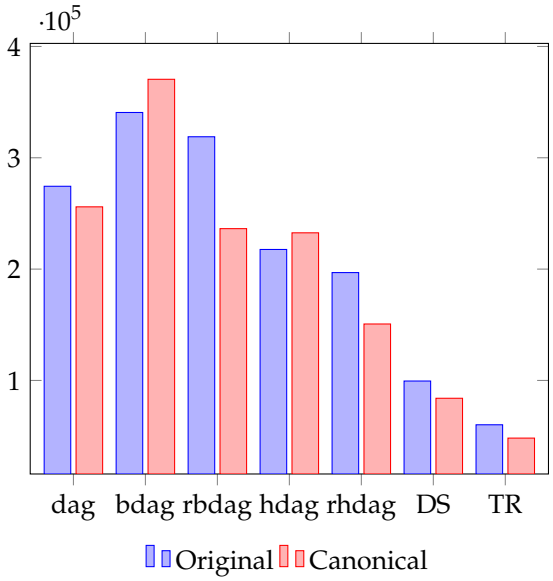
The characteristics of the Corpora are quite different: Corpus I consists of few and very large files while Corpus II has many small files. Some characteristics are shown in Table 1. As can be seen, the average size of documents from Corpus I is about 40 times larger than that of Corpus II, and the rank (=maximum length of sibling lists) of documents from Corpus I is about 1300 times larger; this indicates that most of the documents from Corpus I are indeed very long lists of (small) subtrees.

The implementations for dag, bdag, rbdag, hdag, and DS are the same ones as used in [10]. Note that DS uses Gonzalo Navarro's implementation of RePair for strings[3]. For TreeRePair, called "TR" in what follows, we use Roy Mennicke's implementation[4]; we do not change any parameters and run it plain from the command line (thus, the maxRank parameter of TR is at its default value of 4). We do not report running times (they are provided in [10]). The canonizer was implemented from scratch in java using integer and string sorting as provided by java (this runs quite slow and can take several hours for some of the documents).

The results of applying the different compressors to the documents of Corpus I are shown in Table 2. The first line shows how the compression ratio on the canonical tree changes with respect to the compression ratio for the original tree (a percentage of more than 100% means that the compression ratio is better on the canonical tree). The second row shows the sizes of the compressed canonical trees (in number of edges). For instance, the compression ratio of the hdag of the canonical tree of document "sprot39.dat" is 67% of the ratio for the original tree. On the other hand, DS compressor over the canonical tree of document "EnWikTionary" has a compression that is 191-times better than the ratio for the original tree. For each document we

[1] http://xmlcompbench.sourceforge.net
[2] http://data.politicalmashup.nl/xmlweb
[3] http://www.dcc.uchile.cl/~gnavarro/software/
[4] http://code.google.com/p/treerepair

Figure 3: Comparison of average sizes of Corpus I.



indicate in bold the unique best increase of compression, and underline the smallest size.

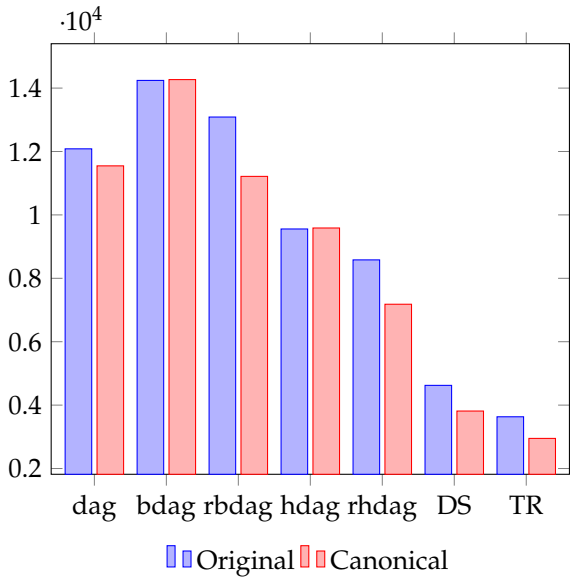
Note that the minimal DAG of the canonical tree can never be larger than the minimal DAG of the original tree. Intuitively, every original repeating subtree (that gets shared in the DAG) is also repeating in the canonical tree. Thus, there cannot be percentages below 100 in the column for the DAG. In every other column the percent number can potentially be below 100. This is because these compressors take into account sibling sequences and hence are effected by the change of sibling orders due to canonization. In fact, this happens for the file "EXI-factbook": here *all* compression ratios (except that for the DAG) become worse for the canonical tree. It means the the ordering of the canonization removes repetitions that are meaningful for the compressor. It is interesting to see that for this outlier, the strongest overall compressor TR (with respect to size) is affected the most: the compression goes down to 81% of the original; this is also the only file where this ever happens for TR. Another outlier that comes from the EXI group is EXI-weblog, where no compression ratio changes; this document is in an order that is isomorphic to that of the canonical tree.

The majority (almost one half) of documents have the strongest increase for the DS compressor. In particular, all the EnWik documents belong to this group. It is interesting to observe that for all the EnWik documents *only* DS and TR give (*massive*) compression, while for all the DAG variants the compression ratio does not change. This means that after canonization there are (i) no repeating subtrees and (ii) no repeating prefixes or suffixes of sibling lists that were different before canonization. Note also that for this group, DS achieves the smallest size values for each document. It means that there are no complex tree patterns that are repeating, and hence would be compressed by TR but not by DS; all repetition seems to be purely on the level of sibling lists. In contrast to that, observe the treebank file which features (by far) the most complex tree structure of all the documents: here the size of TR is almost twice smaller than that of DS. Curiously, the rhdag has the highest increase for this document. There is another interesting

document	dag	bdag	rbdag	hdag	rhdag	DS	TR
1998statistics	118%	352%	**373**%	306%	333%	239%	211%
	1164	682	632	422	373	200	238
catalog-01	146%	82%	177%	84%	**182**%	146%	117%
	5856	8514	5830	5302	3295	2994	3390
catalog-02	114%	98%	111%	98%	113%	**473**%	331%
	28496	53647	50858	27912	25823	5761	8072
dictionary-01	107%	102%	**225**%	104%	187%	130%	136%
	54575	75827	33386	45214	24960	24634	16434
dictionary-02	116%	116%	**254**%	117%	207%	138%	145%
	469915	588665	257228	353365	197197	194324	115932
EnWikiNew	100%	100%	100%	100%	100%	**2712**%	2282%
	35075	70018	70025	35057	35054	341	422
EnWikiQuote	100%	100%	100%	100%	100%	**2370**%	2091%
	23904	47692	47699	23888	23887	267	316
EnWikiVersity	100%	100%	100%	100%	100%	**2672**%	2287%
	43693	87258	87263	43676	43673	264	326
EnWikTionary	100%	100%	100%	100%	100%	**19108**%	15839%
	726221	1452273	1452279	726197	726191	428	531
EXI-Array	100%	100%	100%	100%	100%	**425**%	375%
	95584	128009	128011	95562	95563	213	267
EXI-factbook	100%	100%	91%	96%	96%	93%	81%
	4477	5090	3227	3766	2225	1937	1708
EXI-Invoice	100%	100%	100%	100%	100%	98%	**102**%
	1073	2073	2067	1071	1066	98	106
EXI-Telecomp	100%	100%	100%	100%	100%	99%	**102**%
	9933	19807	19808	9932	9931	111	137
EXI-weblog	100%	100%	100%	100%	100%	100%	100%
	8504	16997	16997	8504	8504	44	58
JST_gene.chr1	100%	99%	100%	99%	100%	**430**%	396%
	9176	14718	14103	7840	7206	917	1062
JST_snp.chr1	100%	98%	101%	97%	101%	**382**%	347%
	23509	41444	37425	22805	19111	2571	2980
medline	165%	150%	**240**%	141%	222%	145%	141%
	395754	493136	158984	326638	113932	122270	88109
NCBI_gene.chr1	100%	93%	110%	91%	116%	**148**%	137%
	16038	15504	9839	11606	5912	4237	3764
NCBI_snp.chr1	100%	100%	100%	100%	100%	100%	100%
	404704	809394	809394	404704	404704	61	83
sprot39.dat	102%	60%	**269**%	67%	237%	102%	102%
	1724689	2394532	586523	1484814	376067	328469	257376
treebank	101%	99%	106%	99%	**107**%	104%	103%
	1292198	1455300	1171666	1246195	1039933	1073301	510683

Table 2: Difference (in %) of canonical versus original tree compression, and size of canonical compression output (largest in bold and smallest underlined, respectively).

Figure 4: Comparison of average sizes of Corpus II.



group of documents, namely those where the rbdag has the largest increase. It means that after canonization there are a lot of repeating prefixes of sibling sequences; thus, optional elements which typically appear at the end of sibling lists (the reverse DAGs profit from that) have, after canonization, remained to appear at the end. Apparently, this is less often the case for the reverse hybrid DAG, i.e., after building the DAG there is less profit from canonization. An interesting document that has always been challenging with respect to compression [11] is medline: with 165% it has the largest increase within the DAG column. This means that many permutations of the same subtree sequences exist. This could be because these bibliography entries have been entered manually by different persons, each having their own preferences of ordering sibling lists. Observe also that every single compressor has an increase of at least 140% for the medline document. Similar to this is the 1998statistics document: here the DAG only increases by 118%, but all others increase by 210% or more. Thus, there are not many subtrees with precisely the same subtrees (possibly in different orders), but, there is a large number of repetitions of subsequences of sibling lists, in particular of prefix subsequences (viz. the highest increases of rbdag and bdag).

In summary, Figure 3 and Figure 4 show the average sizes for the different compressors for Corpus I and Corpus II, respectively. For Corpus I, all compressors, except bdag (91%) and hdag (93%), show an improvement of the compression ratio. DS (118%) and TR (124%), which already give very high compression ratios, also have high increases. The biggest increases, however, are seen for rbdag (134%) and rhdag (130%). For Corpus II, we see that there is almost no difference in the case of bdag and hdag. Again, DS (121%) and TR (123%) improve on their already high compression ratios, while rbdag (117%) and rhdag (120%) achieve improvements as well.

Finally, we also tried a different canonizer: It assigns to every subtree t a number $i(t)$ such that for every pair of subtrees t_1, t_2 it holds that $i(t_1) = i(t_2)$ if and only if the unordered trees of t_1 and t_2 are equal. The children $t_1, \ldots, t_n$ of a subtree t are then sorted with respect to $i(t_1), \ldots, i(t_n)$. While this

algorithm runs a lot faster than sorting the whole subtrees, the compression ratios only change very slightly.

8 EVALUATING AUTOMATA

This section is motivated by so-called forest automata (see, for example [17]). Like regular automata can be used to check if a word is a member of a (regular) language, forest automata can be used to check if a forest is part of a certain language. A naive approach to check if the forest of an FSLP is accepted by a forest automaton would be to evaluate the FSLP first, which has exponential runtime cost. We will present a faster implementation that uses the fcns encoding. Later, we will show how to evaluate visibly one-counter automata on FSLPs.

8.1 FOREST AUTOMATA

The following definition from forest automata is from [17] where they are called hedge automata.

Definition 49 (Forest automaton)**.** For an alphabet Σ, the set of regular expressions over Σ is denoted with $\mathcal{R}(\Sigma)$ and the language of a regular expression $e \in \mathcal{R}(\Sigma)$ is denoted with $\mathcal{L}(e)$. A *forest automaton* is a tuple $A = (\Sigma, P, F, \triangle)$, where P is a finite set of states, $F \subseteq P$ and $\triangle\colon P \times \Sigma \to \mathcal{R}(P)$. Here, $\triangle$ means that the automaton may label a tree $a\langle f \rangle$ with q if f can be labelled with a word from $\mathcal{L}(\triangle(q, a))$, i.e. a tree t is labelled with the following set of states $S(t)$:

$$S(a\langle t_1 \dots t_n \rangle) = \{q \in Q \mid q_1 \in S(t_1), \dots, q_n \in S(t_n),\, q_1 \dots q_n \in \mathcal{L}(\triangle(q, a))\}$$

The accepted language is $\mathcal{L}(A) = \{t \in \mathcal{T}(\Sigma) \mid S(t) \cap F \neq \varnothing\}$. The size $|A|$ is $|Q|$ plus the sizes of the regular expressions appearing in $\triangle$.

We make use of the fact that FSLPs can be converted into TSLPs for their fcns encodings. Then we evaluate these on the corresponding tree automata:

Corollary 6. *Given a forest automaton A and an FSLP F we can check in polynomial time in both $|A|$ and $|F|$ whether A accepts $[\![F]\!]$.*

Proof. We first use Proposition 8 to construct a TSLP T of size $\mathcal{O}(|F|)$ such that $[\![T]\!] = \mathrm{fcns}([\![F]\!])$. Also, we use the construction from [17] (Proposition 8.3.2) to convert A into a tree automaton A' such that for every $f \in \mathcal{F}(\Sigma)$ we have A' accepts $\mathrm{fcns}(f)$ if and only if A accepts f. Whether A' accepts $[\![T]\!] = \mathrm{fcns}([\![F]\!])$ can be tested in polynomial time using the construction from [39]. □

8.2 VISIBLY ONE-COUNTER AUTOMATA

In the following, we use a form of visibly one-counter automata on forests. One-counter automata on strings basically have a natural number (a counter)

that during transitions is incremented, decremented or left as is. In the case of *visibly* one-counter automata the input symbol completely determines which of these three cases happen (i.e. how the counter is modified does not depend on the state the automaton is in). The input alphabet is therefore divided into three parts: When reading *call symbols* the stack is incremented, when reading *internal symbols* the stack is kept as is, and when reading *return symbols* the stack is decremented, if possible. When return symbols are encountered and the stack is already 0, the word is rejected. In e.g. [31, 2], visibly one-counter automata may distinguish finitely many counter states, e.g. if the counter is 5 it may do a different state transition than when the counter is not 5. Here, we omit these capabilities and only focus on the counter itself.

We start with a function $c\colon \Sigma \to \{-1, 0, +1\}$ and extend this to $c^*\colon \Sigma^* \to \mathbb{Z}$ with $c^*(\varepsilon) = 0$, $c^*(a) = c(a)$ for $a \in \Sigma$ and $c^*(vw) = c^*(v) + c^*(w)$ for $v, w \in \Sigma^*$. We say that a word $w \in \Sigma^*$ is *accepted by c starting with $k \in \mathbb{N}$* if there is no prefix v of w, i.e. $w = vv'$ for some $v' \in \Sigma^*$, such that $k + c^*(v) < 0$. For example, let $\Sigma = \{a, b\}$, $c(a) = +1$ and $c(b) = -1$. The word $abba$ is accepted starting with $k \geq 1$ but not with $k = 0$, because $c^*(abb) = -1$. We say that a word is *accepted by c* if it is accepted by c starting with 0. In case c is clear from the context, we just say that a word is accepted (starting from k). The goal now is to calculate if the string of an SLP is accepted by c. For this, we compute a pair $(m, d) \in \mathbb{N} \times \mathbb{Z}$ for each string expression $s \in \mathcal{E}_{\mathcal{S}}(\Sigma)$ with the following properties:

- m is the smallest number such that $[\![s]\!]$ is accepted starting with m.
- d is $c^*([\![s]\!])$.

We define the algebra $((Z, \tau), \mathcal{I}_{\mathcal{S}})$ with $Z = \mathbb{N} \times \mathbb{Z}$, $\tau\colon Z \to \{\mathcal{S}\}$ and

- $\mathcal{I}_{\mathcal{S}}(\varepsilon) = (0, 0)$,
- $\mathcal{I}_{\mathcal{S}}(a) = (\max(0, -c(a)), c(a))$,
- $\mathcal{I}_{\mathcal{S}}(\circ)((m_\ell, d_\ell), (m_r, d_r)) = (\max(m_\ell, m_r - d_\ell), d_\ell + d_r)$.

For example, if $c(a) = -1$, then $\mathcal{I}_{\mathcal{S}}(a) = (1, -1)$, meaning that 1 is the smallest number that a is accepted with. If $c(a) = 1$, then $\mathcal{I}_{\mathcal{S}}(a) = (0, 1)$, so 0 is the smallest number that a is accepted with.

Let us now extend the visibly one-counter setting to forests. A forest $a_1\langle f_1\rangle \dots a_n\langle f_n\rangle \in \mathcal{F}(\Sigma)$ is *accepted by c* if $f_1, \dots, f_n \in \mathcal{F}(\Sigma)$ are accepted by c and $a_1 \dots a_n \in \Sigma^*$ is accepted by c.

Like we did in Chapter 4, we use the notation $M_\perp$ as a short-hand for $M \uplus \{\perp\}$ for any set M. To test if a forest is accepted, we implement the following algebra $\mathcal{A}(c) = ((\mathcal{U}, \tau_{\mathcal{U}}), \mathcal{I})$. For a forest expression $e \in \mathcal{E}_{\mathcal{F}}(\Sigma)$ with $e\colon \mathcal{F}$ we basically calculate the same information as we did for strings, but we also have to remember if every subforest is accepted. We therefore use $Z_\perp$, where $\perp$ means that a subforest is rejected. For example, if $f \in \mathcal{F}(\Sigma)$ is already rejected, then $f_\ell a\langle f\rangle f_r$ cannot be accepted, regardless of what f_ℓ and f_r are. For forest expressions $e \in \mathcal{E}_{\mathcal{F}}(\Sigma)$ with $e\colon \mathcal{F}_x$ we compute two pairs from Z, one for the forest left of x and one for the forest right of x. In addition to that, if x is not part of the roof of the forest, then we compute a third pair from Z for the roof. Furthermore, forests with parameters can be rejected if a subforest is already rejected, regardless of what we substitute for x or what we concatenate at the roof. We therefore choose $(Z \times Z_\perp \times Z)_\perp$ as carrier for forests with parameters. In total our carrier set

is $(\mathcal{U}, \tau_{\mathcal{U}})$ with $\mathcal{U} = (Z_\perp \cup (Z \times Z_\perp \times Z)_\perp)$, $\tau_{\mathcal{U}}(x) = \mathcal{F}$ if $x \in Z_\perp$ and $\tau_{\mathcal{U}}(x) = \mathcal{F}_x$ if $x \in (Z \times Z_\perp \times Z)_\perp$. We evaluate forest expressions over it as follows: Let $e = (0,0)$, $x \circ y = \mathcal{I}_{\mathcal{S}}(\circ)(x,y)$ for $x, y \in Z$ and let $A = \{x \in Z \mid x \text{ is accepted}\}$. The result of any operation is defined as $\perp$ if one of the arguments is $\perp$, i.e. for all $s \in Z$ we have $\mathcal{I}(\boxminus)(\perp, s) = \perp$, $\mathcal{I}(\boxminus)(s, \perp) = \perp$, $\mathcal{I}(\boxminus)(\perp, \perp) = \perp$, and so on. The remaining cases where none of the arguments are $\perp$ are defined as follows:

- $\mathcal{I}(\varepsilon) = e$,
- $\mathcal{I}(x) = (e, \perp, e)$,
- $\mathcal{I}(a(x)) = (e, \mathcal{I}_{\mathcal{S}}(a), e)$,
- $\mathcal{I}(\boxminus)(\ell, r) = \ell \circ r$,
- $\mathcal{I}(\boxbslash)((\ell,t,r),b) = \begin{cases} \ell \circ b \circ r & \text{if } t = \perp, \\ t & \text{if } t \neq \perp \text{ and } \ell \circ b \circ r \in A, \\ \perp & \text{if } t \neq \perp \text{ and } \ell \circ b \circ r \notin A, \end{cases}$
- $\mathcal{I}(\boxslash)((\ell,t,r),s) = (\ell, t, r \circ s)$,
- $\mathcal{I}(\boxbslash)(s,(\ell,t,r)) = (s \circ \ell, t, r)$,
- $\mathcal{I}(\boxplus)((\ell,t,r),(\ell',t',r')) = \begin{cases} (\ell', t, r') & \text{if } t \neq \perp,\ t' \neq \perp \\ & \text{and } \ell \circ t' \circ r \in A, \\ \perp & \text{if } t \neq \perp,\ t' \neq \perp \\ & \text{and } \ell \circ t' \circ r \notin A, \\ (\ell \circ \ell', t, r' \circ r) & \text{if } t \neq \perp \text{ and } t' = \perp, \\ (\ell', \ell \circ t' \circ r, r') & \text{if } t = \perp \text{ and } t' \neq \perp, \\ (\ell \circ \ell', \perp, r' \circ r) & \text{if } t = \perp \text{ and } t' = \perp. \end{cases}$

Theorem 11. *Given $c\colon \Sigma \to \{-1, 0, 1\}$ and an FSLP $F = (V, \Gamma, \rho, S)$ we can test if $[\![F]\!]$ is accepted by c in time $\mathcal{O}(|F| \cdot \log([\![F]\!]))$ and space requirements $\mathcal{O}(\log([\![F]\!]))$.*

Proof. Since elements from $\mathcal{U}$ contain numbers in the size of $[\![F]\!]$, the space requirements for such an element are in $\mathcal{O}(\log([\![F]\!]))$. Each operation from $\mathcal{I}$ has to do arithmetic on these numbers, which has a runtime cost of $\mathcal{O}(\log([\![F]\!]))$ per operation. We can therefore test if $[\![F]\!]$ is accepted by doing the following: Let $r = [\![S]\!]_{F, \mathcal{A}(c)}$. If $r = \perp$ then $[\![F]\!]$ is rejected. Otherwise, $r = (m, d)$ and $[\![F]\!]$ is accepted if and only if $m = 0$. □

9 OPTIMAL WORST-CASE COMPRESSION

Let Σ be an alphabet and $\sigma = |\Sigma|$. Since we need σ as the base of logarithms, we require that $\sigma \geq 2$. Let $\min|\mathrm{tdag}(t)|$ denote the size of a smallest top dag for a tree $t \in \mathcal{C}(\Sigma)$. A simple counting argument shows that $\Omega(n/\log_\sigma n)$ is the information-theoretic lower bound for

$$\max\{\min|\mathrm{tdag}(t)| \mid t \in \mathcal{C}(\Sigma), |t| = n\}.$$

We present a new linear-time top dag construction that achieves this bound. In addition, our construction has two properties that are also true for the original construction of Bille et al. [5], which are the following: Let $t \in \mathcal{C}(\Sigma)$ with $|t| = n$.

- The size of the top dag is bounded by $\mathcal{O}(|\mathrm{mdag}(t)| \cdot \log n)$ and
- the height of the top dag is bounded by $\mathcal{O}(\log n)$.

Concerning the first point it was shown in [4] that the factor $\log n$ for the size actually occurs in the construction of [5]. The logarithmic bound on the height is important to obtain the logarithmic time bounds for the querying operations (e.g. computing the label, parent node, first child, right sibling, depth, height, nearest common ancestor, etc. of nodes given by their preorder numbers) in [5].

It was shown in [5] that their algorithm produces a top dag of size $\mathcal{O}(n/\log_\sigma^{0.19} n)$. In [27] this bound was improved to $\mathcal{O}(n \log\log n/\log_\sigma n)$. After the arXiv-version of this result had appeared, an alternative construction of top dags of size $\mathcal{O}(n/\log_\sigma n)$ was presented in [21]. In that paper, it is also shown that the $\mathcal{O}(n \log\log n/\log_\sigma n)$ bound for the top dag construction from [5, 27] cannot be improved.

Theorem 12. *There is a linear time algorithm that computes from a given tree $t \in \mathcal{C}(\Sigma)$ with $|t| = n$ a top dag G with $|G| \in \mathcal{O}(n/\log_\sigma n)$, $h(G) \in \mathcal{O}(\log n)$ and $|G| \in \mathcal{O}(|\mathrm{mdag}(t)| \cdot \log n)$.*

An integral part of our construction is a modification of the algorithm BU-Shrink (bottom-up shrink) from [22], which constructs in linear time a TSLP of size $\mathcal{O}(n/\log_\sigma n)$ for a given binary tree t. In addition to that, the algorithm of Bille et al. from [5] is used. Our construction and the algorithm of Bille et al. work on edge-labelled trees that label edges with cluster expressions. We represent these using expressions over the following signature:

Definition 50 (Edge-labelled cluster trees)**.** Let

$$\mathbf{type}_{\mathcal{Z}} = \{\triangle_a \mid a \in \Sigma\} \cup \{*_i \mid i \in \mathbb{N}, i \geq 1\}.$$

The signature $\mathcal{S}_{\mathcal{Z}}$ over Σ with types $\mathbf{type}_{\mathcal{Z}}$ has the following operations:

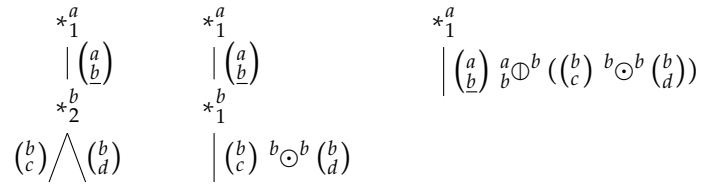


Figure 1: Example of three equivalent edge-labelled cluster trees.

- $c\colon \triangle_a$ for every $c \in \mathcal{E}_{\mathcal{C}}(\Sigma)$, $c\colon \mathcal{C}^a$, where $a \in \Sigma$,
- $*_n^a\colon \triangle_a^n \to *_a$ for every $n \in \mathbb{N}$ with $n \geq 1$ and $a \in \Sigma$, and
- $c\colon *_b \to \triangle_a$ for every $c \in \mathcal{E}_{\mathcal{C}}(\Sigma)$, $c\colon \mathcal{C}_b^a$, where $a, b \in \Sigma$.

We set $\mathcal{E}_{\mathcal{Z}}(\Sigma) = \mathcal{E}(\mathcal{S}_{\mathcal{Z}}(\Sigma))$.

Consider a tree $t = a\langle b\langle cd\rangle\rangle$. The most basic expression for t that only uses atomic clusters is $*_1^a\left(\binom{a}{\underline{b}}\left(*_2^b\left(\binom{b}{c}, \binom{b}{d}\right)\right)\right)$. Here, the $*_i^a$-operations take the roles of (inner) nodes with $i \geq 1$ outgoing edges, the cluster expressions from $\mathcal{C}^a$ are labels of edges that lead to a leaf node (which we do not model explicitly) and the cluster expressions from $\mathcal{C}_b^a$ are labels of edges that go to an internal node. An algorithm that wants to build a cluster expression for t can now perform simplifications on the $\mathcal{E}_{\mathcal{Z}}(\Sigma)$-part of the expression, while integrating multiple cluster expressions. We can transform the above expression into $*_1^a\left(\binom{a}{\underline{b}}\left(*_1^b\left(\left(\binom{b}{c}\ {}^b\!\odot^b\ \binom{b}{d}\right)\right)\right)\right)$, which can in turn be transformed into $*_1^a\left(\binom{a}{\underline{b}}\ {}_b^a\!\oplus^b\left(\binom{b}{c}\ {}^b\!\odot^b\ \binom{b}{d}\right)\right)$. See Figure 1 for a visualization.

To start our algorithm, we are first going to convert a cluster into an expression $\mathcal{E}_{\mathcal{Z}}(\Sigma)$, which is defined using the following two functions

$$s_{\mathcal{T}}\colon \Sigma \to \mathcal{C}(\Sigma) \to \mathcal{E}_{\mathcal{Z}}(\Sigma),$$
$$s_{\mathcal{F}}\colon \Sigma \to \mathcal{C}(\Sigma)^+ \to \mathcal{E}_{\mathcal{Z}}(\Sigma),$$

that receive the current symbol as an extra parameter and are defined as

$$s_{\mathcal{T}}(a)(b\langle f\rangle) = \begin{cases} \binom{a}{b} & \text{if } f = \varepsilon, \\ \binom{a}{\underline{b}}(s_{\mathcal{F}}(b)(f)) & \text{if } f \neq \varepsilon, \end{cases}$$
$$s_{\mathcal{F}}(a)(t_1 \dots t_n) = *_n^a(s_{\mathcal{T}}(a)(t_1), \dots, s_{\mathcal{T}}(a)(t_n)).$$

We define $s\colon \mathcal{C}(\Sigma) \to \mathcal{E}_{\mathcal{Z}}(\Sigma)$ as $s(a\langle f\rangle) = s_{\mathcal{F}}(a)(f)$. Following our previous example, where $t = a\langle b\langle cd\rangle\rangle$, we get

$$\begin{aligned} s(t) &= s_{\mathcal{F}}(a)(b\langle cd\rangle) \\ &= *_1^a(s_{\mathcal{T}}(a)(b\langle cd\rangle)) \\ &= *_1^a\left(\binom{a}{\underline{b}} s_{\mathcal{F}}(b)(cd)\right) \\ &= *_1^a\left(\binom{a}{\underline{b}}(*_2^b(s_{\mathcal{T}}(b)(c), s_{\mathcal{T}}(b)(d))\right) \\ &= *_1^a\left(\binom{a}{\underline{b}}\left(*_2^b\left(\binom{b}{c}, \binom{b}{d}\right)\right)\right). \end{aligned}$$

Any expression from $\mathcal{E}_{\mathcal{Z}}(\Sigma)$ can be evaluated into a cluster expression from $\mathcal{E}_{\mathcal{C}}(\Sigma)$, which we do with the algebra $\mathcal{A}_{\mathcal{Z},\Sigma} = (\mathcal{U}_{\mathcal{Z}}, \mathcal{I}_{\mathcal{Z}})$, that is defined as follows: Let $\overline{\mathcal{E}_{\mathcal{C}}(\Sigma)} = \{\bar{e} \mid e \in \mathcal{E}_{\mathcal{C}}(\Sigma)\}$ be a copy of $\mathcal{E}_{\mathcal{C}}(\Sigma)$. We set $\mathcal{U}_{\mathcal{Z}} = (\mathcal{E}_{\mathcal{C}}(\Sigma) \cup \overline{\mathcal{E}_{\mathcal{C}}(\Sigma)}, \tau_{\mathcal{Z}})$ where $\tau_{\mathcal{Z}}(c) = *_a$ if $c\colon \mathcal{C}^a$ for some $a \in \Sigma$ and $\tau_{\mathcal{Z}}(\bar{c}) = \triangle_a$ if $c\colon \mathcal{C}^a$ for some $a \in \Sigma$. Then $\mathcal{I}_{\mathcal{Z}}$ is defined as follows:

- $\mathcal{I}_{\mathcal{Z}}(c) = \overline{c}$ if $c{:}\mathcal{C}^a$ for some $a \in \Sigma$,
- $\mathcal{I}_{\mathcal{Z}}(*_n)(\overline{c_1}, \ldots, \overline{c_n}) = c_1 \, {}^a\!\odot^a \ldots {}^a\!\odot^a \, c_n$ if $c_1, \ldots, c_n{:}\mathcal{C}^a$ for some $a \in \Sigma$, and
- $\mathcal{I}_{\mathcal{Z}}(c)(d) = \overline{c \, {}^a_b\!\oplus^b d}$ if $c{:}\mathcal{C}^a_b$ for some $a, b \in \Sigma$.

We can then further evaluate this expression using $[\![\,]\!]$ of the cluster algebra, so let $[\![\,]\!]_{\mathcal{Z}}{:}\mathcal{E}_{\mathcal{Z}}(\Sigma) \to \mathcal{T}(\Sigma)$ be defined as $[\![\,]\!]_{\mathcal{Z}} = [\![\,]\!]_{\mathcal{A}_{\mathcal{C},\Sigma}} \circ [\![\,]\!]_{\mathcal{A}_{\mathcal{Z},\Sigma}}$.

The algorithm of Bille et al. starts with a tree $t \in \mathcal{T}(\Sigma)$ and converts it into a helper tree $\widetilde{T}$ that is an edge-labelled tree, where at first every edge is labelled with the corresponding atomic cluster expression. For our purpose, we say that $\widetilde{T} \in \mathcal{E}_{\mathcal{Z}}(\Sigma)$ and that we can hand *any* such expression to the algorithm, not just the ones that only consist of atomic clusters expressions. The algorithm shrinks $\widetilde{T}$ down to a single edge, i.e. an expression of the form $*_1(T)$, which yields the resulting cluster expression $T \in \mathcal{E}_{\mathcal{C}}(\Sigma)$. Finally, it returns $\mathrm{mdag}(T)$. The difference between our algorithm and applying the algorithm of Bille et al. directly is that we do some preprocessing on $\widetilde{T}$. The guarantees the algorithm of Bille et al. gives about the size and the height of the result only apply to the $\mathcal{E}_{\mathcal{Z}}$-portion of $\widetilde{T}$, not the clusters that are in it. We therefore use the following view of Bille et al.'s algorithm. For an expression $e \in \mathcal{E}_{\mathcal{Z}}(\Sigma)$ let $\mathrm{Cl}(e)$ be all cluster expressions appearing in e.

Lemma 28. *Let $\widetilde{T} \in \mathcal{E}_{\mathcal{Z}}(\Sigma)$. We can construct in linear time a $T \in \mathcal{E}_{\mathcal{C}}(\Sigma)$ with*

- $[\![T]\!]_{\mathcal{A}_{\mathcal{C},\Sigma}} = [\![\widetilde{T}]\!]_{\mathcal{Z}}$,
- $|\mathrm{mdag}(T)| \in \mathcal{O}(|\mathrm{mdag}(\widetilde{T})| \cdot \log|\widetilde{T}| + \sum\{|e| \mid e \in \mathrm{Cl}(\widetilde{T})\})$ *and*
- $h(T) \in \mathcal{O}(\log|\widetilde{T}| + \max\{h(e) \mid e \in \mathrm{Cl}(\widetilde{T})\})$.

We present two versions of our algorithm. The first one does not achieve the mdag bound but is easier to explain, while the second one will be a slight modification of the first one and will achieve the mdag bound. A high-level overview of the steps performed during the first version of the algorithm is as follows:

- Transform $t \in \mathcal{C}(\Sigma)$ into $d := s(t) \in \mathcal{E}_{\mathcal{Z}}(\Sigma)$.
- Perform our modification of BU-Shrink which yields $d' \in \mathcal{E}_{\mathcal{Z}}(\Sigma)$.
- Use the algorithm of Bille et al. to produce a top tree $T \in \mathcal{E}_{\mathcal{C}}(\Sigma)$.
- Construct $G := \mathrm{mdag}(T)$.

9.1 MODIFIED BU-SHRINK

We fix a number $k \in \mathbb{N}$. For each $e \in \mathcal{E}_{\mathcal{C}}(\Sigma)$ we define its *weight* as $w(e) = |[\![e]\!]|$. The algorithm performs the following operations on d as long as possible:

- We can change subexpressions of the form

  $$e(*_1^b(e'))$$

  into

  $$e \, {}^a_b\!\oplus^b e'$$

  if $w(e) < k$ and $w(e') < k$, where $e{:}\mathcal{C}^a_b$ and $e'{:}\mathcal{C}^b$.

- We can change subexpressions of the form

  $$e(*_1^b(e'(f)))$$

  into

  $$(e\ {}_b^a\!\oplus_c^b e')(f)$$

  if $w(e) < k$ and $w(e') < k$, where $e{:}\mathcal{C}_b^a$ and $e'{:}\mathcal{C}_c^b$.

- We can change subexpressions of the form

  $$*_n^a(u_1, \ldots, u_{j-1}, e, e', u_{j+2}, \ldots, u_n)$$

  into

  $$*_{n-1}^a(u_1, \ldots, u_{j-1}, e\ {}^a\!\odot^a e', u_{j+2}, \ldots, u_n)$$

  if $w(e) < k$ and $w(e') < k$, where $e, e'{:}\mathcal{C}^a$.

- We can change subexpressions of the form

  $$*_n^a(u_1, \ldots, u_{j-1}, e, e'(f), u_{j+2}, \ldots, u_n)$$

  into

  $$*_{n-1}^a(u_1, \ldots, u_{j-1}, (e\ {}^a\!\odot_b^a e')(f), u_{j+2}, \ldots, u_n)$$

  if $w(e) < k$ and $w(e') < k$, where $e{:}\mathcal{C}^a$ and $e'{:}\mathcal{C}_b^a$.

- We can change subexpressions of the form

  $$*_n^a(u_1, \ldots, u_{j-1}, e(f), e', u_{j+2}, \ldots, u_n)$$

  into

  $$*_{n-1}^a(u_1, \ldots, u_{j-1}, (e\ {}_b^a\!\odot^a e')(f), u_{j+2}, \ldots, u_n)$$

  if $w(e) < k$ and $w(e') < k$, where $e{:}\mathcal{C}_b^a$ and $e'{:}\mathcal{C}^a$.

The algorithm is not deterministic since the cases can overlap. Which way these ambiguities are resolved is not important (one could always try the cases in order, for example). It is important however, that the algorithm can be implemented in linear time. The arguments are more or less the same as for the analysis of BU-Shrink in [22]: We replace at most $|d|$ many subexpressions, where each update step can be done in constant time. Which updates we still have to do can be maintained as a set of positions into d, which is initially populated by going over all subexpressions. When we update a subexpression, it must be removed from the set, and some of the surrounding expressions might need to be inserted into the set.

Before we continue, we need the following lemma which states that most subexpressions represent forests of one or zero trees:

Lemma 29. *Let $e \in \mathcal{E}_{\mathcal{Z}}(\Sigma)$. Then $\Theta(|e|)$ many subexpressions of e are either of type $\mathcal{C}^a$ for some $a \in \Sigma$ or have the form $e(*_1^b(t))$ for some $e{:}\mathcal{C}_b^a$ and $t{:}\triangle_b$ for some $a, b \in \Sigma$.*

Proof. Let $E{:}\mathcal{E}_{\mathcal{Z}}(\Sigma) \to \mathbb{N}$ be the number of subexpressions of the above form and $\triangle{:}\mathcal{E}_{\mathcal{Z}}(\Sigma) \to \mathbb{N}$ the number of subexpressions that have a type from $\triangle := \bigcup\{\triangle_a \mid a \in \Sigma\}$. We will show that $2E(e) \geq \triangle(e) + 1$. Since $\Theta(|e|)$ many subexpressions of e have a type from $\triangle$, this shows the lemma. A subexpression with type from $\triangle$ has either the form e' with $e'{:}\mathcal{C}^a$ for some $a \in \Sigma$, in which case we have $2E(e') = 2 \geq 1 + 1 = \triangle(e') + 1$, or it has the form $e'(*_n(u_1, \ldots, u_n))$ with $n \geq 1$, where $e' \in \mathcal{C}_b^a$ for some $a, b \in \Sigma$ and $u_1, \ldots, u_n \in \triangle_b$. We proceed using induction:

- In case $n = 1$ we have

$$2E(e'(*_1^b(u))) = 2 + 2E(u) \geq 2 + \triangle(u) + 1 \geq \triangle(e'(*_1^b(u))) + 1.$$

- In case $n > 1$ we have

$$\begin{aligned} 2E(e'(*_n^b(u_1, \ldots, u_n))) &= 2 \sum \{E(u_i) \mid 1 \leq i \leq n\} \\ &\geq \sum \{\triangle(u_i) \mid 1 \leq i \leq n\} + n \\ &\geq \triangle(e'(*_n^b(u_1, \ldots, u_n))) + 1. \end{aligned}$$

  The last step holds since $\triangle(e'(*_n^b(u_1, \ldots, u_n))) = 1 + \sum\{\triangle(u_i) \mid 1 \leq i \leq n\}$ and $n \geq 1$.

□

We now show that after our preprocessing step that turns d into d', we have shrunk d by a factor of k:

Lemma 30. $|d'| \in \mathcal{O}(n/k)$.

Proof. The idea is as follows: We count the number of subexpressions of d' with weight more than k and we will show that this is a constant fraction of $|d'|$. Therefore the sum of the weights of these subexpressions must be $\Omega(n)$, and thus $|d'| \in \mathcal{O}(n/k)$. To show that a constant fraction of subexpressions have weight more than k we do the following: By Lemma 29 a constant fraction of subexpressions are of the form e with $e{:}\mathcal{C}^a$ for some $a \in \Sigma$ or $c(*_1^b(u))$ for some $c{:}\mathcal{C}_b^a$ and some $u{:}\delta_b$, where $a, b \in \Sigma$. We go through all of these and for each one find at least one subexpression of weight more than k in its vicinity (which might be the subexpression itself). Each time a subexpression is found this way we assign it a marking. We will argue that each subexpression can only be marked at most four times, i.e. we do not overcount them. For each $c(u) \in \mathcal{E}_{\mathcal{Z}}(\Sigma, \triangle)$ we set $w(c(u)) = w(c)$. Let $e \neq d'$ be a subexpression such that either $e{:}\mathcal{C}^a$ for some $a \in \Sigma$ or $e = c(*_1^b(u))$ for some $c{:}\mathcal{C}_b^a$ and some $u{:}\triangle_b$, where $a, b \in \Sigma$. Since e is not the whole expression, it must occur in a subexpression of the form $*_n^a(u_1, \ldots, u_{j-1}, e, u_{j+1}, \ldots, u_n)$. We assign the following markings:

- If $w(e) > k$ we mark e.
- If $e{:}\mathcal{C}^a$, $j > 1$ and $w(u_{j-1}) > k$ we mark u_{j-1}.
- If $e{:}\mathcal{C}^a$, $j < i$ and $w(u_{j+1}) > k$ we mark u_{j+1}.
- If $e = c(*_1^b(u))$ and $w(u) > k$ we mark u.

At least one of the previous four cases must occur, because otherwise our algorithm would not have stopped. In addition to that, every subexpression can get marked at most four times. This means we found $\Omega(|d'|)$ subexpressions of weight greater than k. □

To bound the size of the resulting top dag G, we have to count the number of different cluster expressions in T. This number can be upper-bounded by the size of d', which is in $\mathcal{O}(n/k)$, plus the number of different cluster expressions of size at most $2k$, since our algorithm cannot merge two expressions where one has size larger than k. The latter number can be bounded as follows: A cluster expression of size m can be seen as a binary tree with m nodes, where every node has one of $\ell := 2\sigma^2 + 5$ many different

node labels, since there are σ^2 many different cluster expressions of the form $\binom{a}{b}$, σ^2 many of the form $\binom{a}{\underline{b}}$ and five internal operations. Since the number of unlabelled binary trees of size m can be bounded by 4^m, we obtain an upper bound of $4^m \cdot \ell^m = (4\ell)^m$ for the number of cluster expressions of size m. The number of cluster expressions of size at most m can therefore be upper-bounded by $(4\ell)^1 + \cdots + (4\ell)^m \leq m \cdot (4\ell)^m$. In our case, $m = 2k$, so we have at most $2k(4\ell)^{2k}$ many different cluster expressions. We choose $k = c \log_r n$ where $c = \frac{1}{4}$ (any $0 < c < \frac{1}{2}$ is fine) and $r = 4\ell$. The size of d' is therefore in $\mathcal{O}(n/\log_r n)$ and the number of different cluster expressions of size at most $2k$ is at most

$$2\left(\frac{1}{4}\log_r n\right) r^{2(\frac{1}{4}\log_r n)} \in \mathcal{O}\left(\log_r(n) \cdot \sqrt{n}\right) \subseteq \mathcal{O}\left(\frac{n}{\log_r n}\right).$$

Since $\log_r x \in \Theta(\log_\sigma x)$ for any x, we obtain the desired size bound of $\mathcal{O}(n/\log_\sigma n)$. Moreover, the algorithm of Bille et al. (Lemma 28) guarantees that the height of T is in $\mathcal{O}(\log n)$ since all cluster expressions in d' have height $\mathcal{O}(k) \subseteq \mathcal{O}(\log n)$.

9.2 DAG-version of the algorithm

The second version of our algorithm, which achieves the $\mathcal{O}(|\mathrm{mdag}(t)| \cdot \log n)$ bound, basically performs the merge steps on the minimal DAG instead of on the tree. Earlier, we needed a representation of an edge-labelled cluster tree and now we need one of an edge-labelled cluster DAG.

Definition 51 (Edge-labelled cluster DAG)**.** An *edge-labelled cluster DAG* is an SLP $G = (V, \Gamma, \rho, S)$ over $\mathcal{S}_{\mathcal{Z}}$ where for each variable $A \in V$ we have $\rho(A) = *_n^a(e_1, \ldots, e_n)$, where $n \geq 1$ and $a \in \Sigma$ and for each $1 \leq i \leq n$ we either have $e_i : \mathcal{C}^a$ for some $a \in \Sigma$ or $e_i = c_i(B_i)$, $c_i : \mathcal{C}_b^a$ for some $a, b \in \Sigma$ and $B_1, \ldots B_n \in V$.

A high-level overview of our modified algorithm is as follows:

- Instead of working on the previous $d \in \mathcal{E}_{\mathcal{Z}}(\Sigma)$, we first construct $\mathrm{mdag}(t)$ which is an SLP over $\mathcal{S}_{\mathcal{D}}$.
- Then we construct an edge-labelled cluster DAG D for t, which is an SLP over $\mathcal{S}_{\mathcal{Z}}$, that has roughly the same size as $\mathrm{mdag}(t)$.
- The modified version of our algorithm converts D into D' which is also an edge-labelled cluster DAG for $\mathcal{E}_{\mathcal{Z}}(\Sigma)$.
- We unfold D' into $d' = \mathrm{unfold}(D) \in \mathcal{E}_{\mathcal{Z}}(\Sigma)$.
- We continue like we did previously.

The first step is implemented as follows: Let $\mathrm{mdag}(t) = (V', \rho', S)$. We define $D = (V, \rho, S)$, where

$$V = V' \setminus \{A \in V' \mid \rho(A) = a, a \in \Sigma\}.$$

Let $A \in V'$ with $\rho'(A) = b(A_1, \ldots, A_n)$ for some $A_1, \ldots, A_n \in V'$. We define $A^\triangle : \Sigma \to \mathcal{E}_{\mathcal{C}}(\Sigma)$ as

$$A^\triangle(a) = \begin{cases} \binom{a}{b} & \text{if } n = 0, \\ \binom{a}{\underline{b}} & \text{otherwise.} \end{cases}$$

Let $A \in V$ with $\rho'(A) = a(A_1, \ldots, A_n)$, where $A_1, \ldots, A_n \in V'$. Since $A \in V$ we have $n \geq 1$. We define

$$\rho(A) = *_n^a(A_1^{\triangle}(a), \ldots, A_n^{\triangle}(a)).$$

For example, let $\mathrm{mdag}(t) = (\{A, B, S\}, \rho', S)$ with $\rho'(S) = a(A, A)$, $\rho'(A) = b(B)$ and $\rho'(B) = c$. We then have $D = (\{A, S\}, \rho, S)$ with

$$\rho(S) = *_2^a\left(\binom{a}{b}(A), \binom{a}{b}(A)\right),$$
$$\rho(A) = *_1^b\left(\binom{b}{c}\right).$$

The possible transformations our algorithm does on D are as follows:

- If there is a variable $A \in V$ with $\rho(A) = *_n^a(\alpha_1, \ldots, \alpha_n)$, $\alpha_i = e_i(B_i)$, $\rho(B_i) = *_1^b(e)$ for some $1 \leq i \leq n$, where $w(e_i), w(e) < k$, $e_i : \mathcal{C}_b^a$ and $e : \mathcal{C}^b$ then we can change $\rho(A)$ into

$$*_n^a(\alpha_1, \ldots, \alpha_{i-1}, e_i \,{}_b^a\!\oplus^b e, \alpha_{i+1}, \ldots, \alpha_n).$$

- If there is a variable $A \in V$ with $\rho(A) = *_n^a(\alpha_1, \ldots, \alpha_n)$, $\alpha_i = e_i(B_i)$, $\rho(B_i) = *_1^b(e(B))$ for some $1 \leq i \leq n$, where $w(e_i), w(e) < k$, $e_i : \mathcal{C}_b^a$ and $e : \mathcal{C}_c^b$ then we can change $\rho(A)$ into

$$*_n^a(\alpha_1, \ldots, \alpha_{i-1}, (e_i \,{}_b^a\!\oplus_c^b e)(B), \alpha_{i+1}, \ldots, \alpha_n).$$

- If there is a variable $A \in V$ with $\rho(A) = *_n^a(\alpha_1, \ldots, \alpha_n)$ and there is an $1 \leq i < n$ such that $\alpha_i = e_i$, $\alpha_{i+1} = e_{i+1}$, $w(e_i), w(e_{i+1}) < k$ and $e_i, e_{i+1} : \mathcal{C}^a$ then we can change $\rho(A)$ into

$$*_{n-1}^a(\alpha_1, \ldots, \alpha_{i-1}, e_i \,{}^a\!\odot^a e_{i+1}, \alpha_{i+2}, \ldots, \alpha_n).$$

- If there is a variable $A \in V$ with $\rho(A) = *_n^a(\alpha_1, \ldots, \alpha_n)$ and there is an $1 \leq i < n$ such that $\alpha_i = e_i(B)$, $\alpha_{i+1} = e_{i+1}$, $w(e_i), w(e_{i+1}) < k$, $e_i : \mathcal{C}_b^a$ and $e_{i+1} : \mathcal{C}^a$ then we can change $\rho(A)$ into

$$*_{n-1}^a(\alpha_1, \ldots, \alpha_{i-1}, (e_i \,{}_b^a\!\odot^a e_{i+1})(B), \alpha_{i+2}, \ldots, \alpha_n).$$

- If there is a variable $A \in V$ with $\rho(A) = *_n^a(\alpha_1, \ldots, \alpha_n)$ and there is an $1 \leq i < n$ such that $\alpha_i = e_i$, $\alpha_{i+1} = e_{i+1}(B)$, $w(e_i), w(e_{i+1}) < k$, $e_i : \mathcal{C}^a$ and $e_{i+1} : \mathcal{C}_b^a$ then we can change $\rho(A)$ into

$$*_{n-1}^a(\alpha_1, \ldots, \alpha_{i-1}, (e_i \,{}^a\!\odot_b^a e_{i+1})(B), \alpha_{i+2}, \ldots, \alpha_n).$$

In all three steps, neither $|D|$ nor $h(D)$ increase. It is possible to "orphan" a variable in one of these steps, i.e. it is not used in $[\![S]\!]_D$ anymore, in which case we remove it. The size bound $\mathcal{O}(n/\log_\sigma n)$ and the height bound $\mathcal{O}(\log n)$ for $\mathrm{mdag}(T)$ follow from our previous arguments. It remains to show that $|\mathrm{mdag}(T)| \in \mathcal{O}(|\mathrm{mdag}(t)| \cdot \log n)$. The algorithm of Bille et al. (Lemma 28) guarantees that

$$|\mathrm{mdag}(T)| \in \mathcal{O}\left(|\mathrm{mdag}(d')| \cdot \log |d'| + \sum\{|e| \mid e \in \mathrm{Cl}(d')\}\right).$$

First, we have $\sum\{|e| \mid e \in \mathrm{Cl}(d')\} \in \mathcal{O}(|\mathrm{mdag}(t)| \cdot \log n)$ because $|d'| \leq |\mathrm{mdag}(t)|$ and every $e \in \mathrm{Cl}(d')$ has size $|e| \in \mathcal{O}(\log n)$. It therefore remains to show that

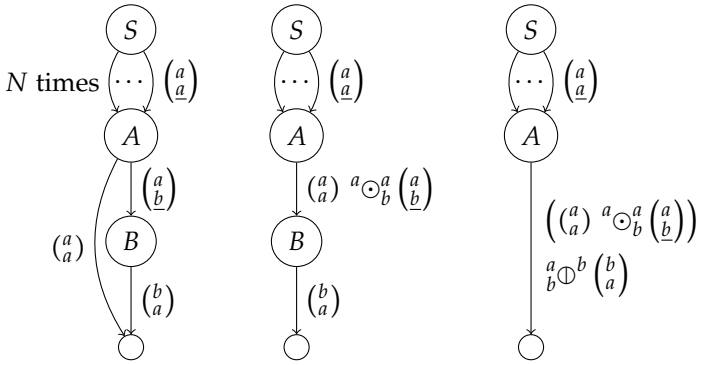


Figure 2

$|\mathrm{mdag}(d')| \cdot \log |d'| \in \mathcal{O}(|\mathrm{mdag}(t)| \cdot \log n)$. We obviously have $|d'| \leq n$, so $\log |d'| \leq \log n$. We also have $|\mathrm{mdag}(d')| \leq |\mathrm{mdag}(t)|$ because

$$|\mathrm{mdag}(d')| \in \mathcal{O}(|D'|) \subseteq \mathcal{O}(|D|) \subseteq \mathcal{O}(|\mathrm{mdag}(t)|).$$

In more detail, we have $|\mathrm{mdag}(d')| \leq |D'|$ since D' is a DAG for d' (which might be larger than the minimal DAG), $|D'| \leq |D|$ because the algorithm only reduces the size of the current DAG, and $|D| \in \mathcal{O}(|\mathrm{mdag}(t)|)$ because the minimal edge-labelled DAG and the minimal DAG have roughly the same size.

Example 9. Consider the following edge-labelled cluster DAG D, that is also presented in Figure 2. Let $N \in \mathbb{N}$, $N \geq 1$, and $D = (V, \Gamma, \rho, S)$ with $V = \{S, A, B\}$ and

$$\begin{aligned}
\rho(S) &= *_N^a \left(\binom{a}{\underline{a}}(A), \ldots, \binom{a}{\underline{a}}(A) \right), \\
\rho(A) &= *_2^a \left(\binom{a}{a}, \binom{a}{\underline{b}}(B) \right), \\
\rho(B) &= *_1^b \left(\binom{b}{a} \right).
\end{aligned}$$

Our algorithm performs the following steps: In the first step, we merge horizontally in $\rho(A)$ and obtain

$$\rho(A) = *_1^a \left(\left(\binom{a}{a} \; {}^a\odot_b^a \binom{a}{\underline{b}} \right)(B) \right).$$

In the second step, we merge $\rho(B)$ into $\rho(A)$ and obtain

$$\rho(A) = *_1^a \left(\left(\binom{a}{a} \; {}^a\odot_b^a \binom{a}{\underline{b}} \right) \; {}_b^a\oplus^b \binom{b}{a} \right).$$

Since B is not used anymore, we remove it. We choose N to be large enough such that the previous steps are actually allowed.

10

FUTURE WORK

In Chapter 4 we showed how to navigate in trees and forests and how to do subtree equality checks in constant time. We allowed polynomial time preprocessing, during which we need to find all equal subtrees. This in turn needs equality checks of TSLPs/FSLPs, which we implemented using equality checks on SSLPs. Currently, the best known algorithm for this is the one by Jeż [29], which needs quadratic time. It would be interesting to see if we can implement the whole preprocessing step in quadratic time as well.

In Chapter 6 we implemented equality checks of FSLPs, allowing symbols to be associative and/or commutative. It would be interesting for which other algebraic laws this can be tested in polynomial time. We are confident that we can extend this to idempotent symbols, meaning that $a\langle fttg\rangle = a\langle ftg\rangle$ for all $f, g \in \mathcal{F}(\Sigma)$ and $t \in \mathcal{T}(\Sigma)$. Another interesting open problem concerns context unification modulo associative and commutative symbols. The decidability of (plain) context unification was a long standing open problem finally solved by Jeż [28], who showed the existence of a polynomial space algorithm. Jeż's algorithm uses his recompression technique for TSLPs. One might try to extend this technique to FSLPs with the goal of proving decidability of context unification for terms that also contain associative and commutative symbols. For first-order unification and matching [25], context matching [25], and one-context unification [18] there exist algorithms for TSLP-compressed trees that match the complexity of their uncompressed counterparts. One might also try to extend these results to the associative and commutative setting.

In Chapter 7 we investigated how large the difference between the DAG of a tree and the DAG of one of its unordered versions can be (counting edges or nodes). Another question would be what this difference is *on average*.

The syntax we chose for forest algebras is the one that we found most natural. In [8] a slightly different approach is taken. There, the authors also introduce laws (equalities on forest expressions) that the algebras must fulfil. It would be interesting to see for which forest algbra (including laws) the forests and forest contexts are the initial algebra.

BIBLIOGRAPHY

[1] Serge Abiteboul, Pierre Bourhis, and Victor Vianu. "Highly Expressive Query Languages for Unordered Data Trees". In: *Theory Comput. Syst.* 57.4 (2015), pp. 927–966. DOI: 10.1007/s00224-015-9617-5. URL: https://doi.org/10.1007/s00224-015-9617-5.

[2] Vince Bárány, Christof Löding, and Olivier Serre. "Regularity Problems for Visibly Pushdown Languages". In: *STACS 2006, 23rd Annual Symposium on Theoretical Aspects of Computer Science, Marseille, France, February 23-25, 2006, Proceedings*. Ed. by Bruno Durand and Wolfgang Thomas. Vol. 3884. Lecture Notes in Computer Science. Springer, 2006, pp. 420–431. DOI: 10.1007/11672142_34. URL: https://doi.org/10.1007/11672142%5C_34.

[3] Michael A. Bender and Martin Farach-Colton. "The LCA Problem Revisited". In: *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*. Ed. by Gaston H. Gonnet, Daniel Panario, and Alfredo Viola. Vol. 1776. Lecture Notes in Computer Science. Springer, 2000, pp. 88–94. DOI: 10.1007/10719839_9. URL: https://doi.org/10.1007/10719839%5C_9.

[4] Philip Bille, Finn Fernstrøm, and Inge Li Gørtz. "Tight Bounds for Top Tree Compression". In: *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings*. Ed. by Gabriele Fici, Marinella Sciortino, and Rossano Venturini. Vol. 10508. Lecture Notes in Computer Science. Springer, 2017, pp. 97–102. DOI: 10.1007/978-3-319-67428-5_9. URL: https://doi.org/10.1007/978-3-319-67428-5%5C_9.

[5] Philip Bille, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. "Tree compression with top trees". In: *Inf. Comput.* 243 (2015), pp. 166–177. DOI: 10.1016/j.ic.2014.12.012. URL: https://doi.org/10.1016/j.ic.2014.12.012.

[6] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. "Random Access to Grammar-Compressed Strings and Trees". In: *SIAM J. Comput.* 44.3 (2015), pp. 513–539. DOI: 10.1137/130936889. URL: https://doi.org/10.1137/130936889.

[7] Adrien Boiret, Vincent Hugot, Joachim Niehren, and Ralf Treinen. "Logics for Unordered Trees with Data Constraints on Siblings". In: *Language and Automata Theory and Applications - 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*. Ed. by Adrian-Horia Dediu, Enrico Formenti, Carlos Martín-Vide, and Bianca Truthe. Vol. 8977. Lecture Notes in Computer Science. Springer, 2015, pp. 175–187. DOI: 10.1007/978-3-319-15579-1_13. URL: https://doi.org/10.1007/978-3-319-15579-1%5C_13.

[8] Mikolaj Boja'nczyk and Igor Walukiewicz. "Forest algebras". In: *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas].* Ed. by Jörg Flum, Erich Grädel, and Thomas Wilke. Vol. 2. Texts in Logic and Games. Amsterdam University Press, 2008, pp. 107–132.

[9] Iovka Boneva, Radu Ciucanu, and Slawek Staworko. "Schemas for Unordered XML on a DIME". In: *Theory Comput. Syst.* 57.2 (2015), pp. 337–376. DOI: 10.1007/s00224-014-9593-1. URL: https://doi.org/10.1007/s00224-014-9593-1.

[10] Mireille Bousquet-Mélou, Markus Lohrey, Sebastian Maneth, and Eric Nöth. "XML Compression via Directed Acyclic Graphs". In: *Theory Comput. Syst.* 57.4 (2015), pp. 1322–1371. DOI: 10.1007/s00224-014-9544-x. URL: https://doi.org/10.1007/s00224-014-9544-x.

[11] Peter Buneman. Private Communication. 2005.

[12] Peter Buneman, Martin Grohe, and Christoph Koch. "Path Queries on Compressed XML". In: *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*. Ed. by Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer. Morgan Kaufmann, 2003, pp. 141–152. URL: http://www.vldb.org/conf/2003/papers/S06P01.pdf.

[13] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. "Efficient memory representation of XML document trees". In: *Inf. Syst.* 33.4-5 (2008), pp. 456–474. DOI: 10.1016/j.is.2008.01.004. URL: https://doi.org/10.1016/j.is.2008.01.004.

[14] Samuel R. Buss. "Alogtime Algorithms for Tree Isomorphism, Comparison, and Canonization". In: *Computational Logic and Proof Theory, 5th Kurt Gödel Colloquium, KGC'97, Vienna, Austria, August 25-29, 1997, Proceedings*. Ed. by Georg Gottlob, Alexander Leitsch, and Daniele Mundici. Vol. 1289. Lecture Notes in Computer Science. Springer, 1997, pp. 18–33. DOI: 10.1007/3-540-63385-5_30. URL: https://doi.org/10.1007/3-540-63385-5%5C_30.

[15] Jiazhen Cai and Robert Paige. "Using Multiset Discrimination to Solve Language Processing Problems Without Hashing". In: *Theor. Comput. Sci.* 145.1&2 (1995), pp. 189–228. DOI: 10.1016/0304-3975(94)00183-J. URL: https://doi.org/10.1016/0304-3975(94)00183-J.

[16] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. "The smallest grammar problem". In: *IEEE Trans. Information Theory* 51.7 (2005), pp. 2554–2576. DOI: 10.1109/TIT.2005.850116. URL: https://doi.org/10.1109/TIT.2005.850116.

[17] Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. Available on: http://www.grappa.univ-lille3.fr/tata. release October, 12th 2007. 2007.

[18] Carles Creus, Adrià Gascón, and Guillem Godoy. "One-context Unification with STG-Compressed Terms is in NP". In: *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*. Ed. by Ashish Tiwari. Vol. 15. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012, pp. 149–164. DOI: 10.4230/LIPIcs.RTA.2012.149. URL: https://doi.org/10.4230/LIPIcs.RTA.2012.149.

[19] O'Neil Delpratt, Rajeev Raman, and Naila Rahman. "Engineering succinct DOM". In: *EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings*. Ed. by Alfons Kemper, Patrick Valduriez, Noureddine Mouaddib, Jens Teubner, Mokrane Bouzeghoub, Volker Markl, Laurent Amsaleg, and Ioana Manolescu. Vol. 261. ACM International Conference Proceeding Series. ACM, 2008, pp. 49–60. DOI: 10.1145/1353343.1353354. URL: https://doi.org/10.1145/1353343.1353354.

[20] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. "Variations on the Common Subexpression Problem". In: *J. ACM* 27.4 (1980), pp. 758–771. DOI: 10.1145/322217.322228. URL: https://doi.org/10.1145/322217.322228.

[21] Bartlomiej Dudek and Pawel Gawrychowski. "Slowing Down Top Trees for Better Worst-Case Compression". In: *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*. Ed. by Gonzalo Navarro, David Sankoff, and Binhai Zhu. Vol. 105. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, 16:1–16:8. DOI: 10.4230/LIPIcs.CPM.2018.16. URL: https://doi.org/10.4230/LIPIcs.CPM.2018.16.

[22] Moses Ganardi, Danny Hucke, Artur Jez, Markus Lohrey, and Eric Noeth. "Constructing small tree grammars and small circuits for formulas". In: *J. Comput. Syst. Sci.* 86 (2017), pp. 136–158. DOI: 10.1016/j.jcss.2016.12.007. URL: https://doi.org/10.1016/j.jcss.2016.12.007.

[23] Moses Ganardi, Artur Jez, and Markus Lohrey. "Balancing Straight-Line Programs". In: *CoRR* abs/1902.03568 (2019). arXiv: 1902.03568. URL: http://arxiv.org/abs/1902.03568.

[24] D. J. H. Garling. "The Cauchy-Schwarz Master Class: An Introduction to the Art of Mathematical Inequalities by J. Michael Steele". In: *The American Mathematical Monthly* 112.6 (2005), pp. 575–579. URL: http://www.jstor.org/stable/30037539.

[25] Adrià Gascón, Guillem Godoy, and Manfred Schmidt-Schauß. "Unification and matching on compressed terms". In: *ACM Trans. Comput. Log.* 12.4 (2011), 26:1–26:37. DOI: 10.1145/1970398.1970402. URL: https://doi.org/10.1145/1970398.1970402.

[26] Leszek Gasieniec, Roman M. Kolpakov, Igor Potapov, and Paul Sant. "Real-Time Traversal in Grammar-Based Compressed Files". In: *2005 Data Compression Conference (DCC 2005), 29-31 March 2005, Snowbird, UT, USA*. IEEE Computer Society, 2005, p. 458. DOI: 10.1109/DCC.2005.78. URL: https://doi.org/10.1109/DCC.2005.78.

[27] Lorenz Hübschle-Schneider and Rajeev Raman. "Tree Compression with Top Trees Revisited". In: *Experimental Algorithms - 14th International Symposium, SEA 2015, Paris, France, June 29 - July 1, 2015, Proceedings*. Ed. by Evripidis Bampis. Vol. 9125. Lecture Notes in Computer Science. Springer, 2015, pp. 15–27. DOI: 10.1007/978-3-319-20086-6_2. URL: https://doi.org/10.1007/978-3-319-20086-6%5C_2.

[28] Artur Jez. "Context Unification is in PSPACE". In: *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*. Ed. by Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias. Vol. 8573. Lecture Notes in Computer Science. Springer, 2014, pp. 244–255. DOI:

10.1007/978-3-662-43951-7_21. URL: https://doi.org/10.1007/978-3-662-43951-7%5C_21.

[29] Artur Jez. "Faster Fully Compressed Pattern Matching by Recompression". In: *ACM Trans. Algorithms* 11.3 (2015), 20:1–20:43. DOI: 10.1145/2631920. URL: https://doi.org/10.1145/2631920.

[30] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[31] Andreas Krebs, Klaus-Jörn Lange, and Michael Ludwig. "Visibly Counter Languages and Constant Depth Circuits". In: *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany*. Ed. by Ernst W. Mayr and Nicolas Ollinger. Vol. 30. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 594–607. DOI: 10.4230/LIPIcs.STACS.2015.594. URL: https://doi.org/10.4230/LIPIcs.STACS.2015.594.

[32] N. Jesper Larsson and Alistair Moffat. "Offline Dictionary-Based Compression". In: *Data Compression Conference, DCC 1999, Snowbird, Utah, USA, March 29-31, 1999.* IEEE Computer Society, 1999, pp. 296–305. DOI: 10.1109/DCC.1999.755679. URL: https://doi.org/10.1109/DCC.1999.755679.

[33] Hartmut Liefke and Dan Suciu. "XMILL: An Efficient Compressor for XML Data". In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.* Ed. by Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein. ACM, 2000, pp. 153–164. DOI: 10.1145/342009.335405. URL: https://doi.org/10.1145/342009.335405.

[34] Markus Lohrey. "Algorithmics on SLP-compressed strings: A survey". In: *Groups Complexity Cryptology* 4.2 (2012), pp. 241–299. DOI: 10.1515/gcc-2012-0016. URL: https://doi.org/10.1515/gcc-2012-0016.

[35] Markus Lohrey. "Equality Testing of Compressed Strings". In: *Combinatorics on Words - 10th International Conference, WORDS 2015, Kiel, Germany, September 14-17, 2015, Proceedings*. Ed. by Florin Manea and Dirk Nowotka. Vol. 9304. Lecture Notes in Computer Science. Springer, 2015, pp. 14–26. DOI: 10.1007/978-3-319-23660-5_2. URL: https://doi.org/10.1007/978-3-319-23660-5%5C_2.

[36] Markus Lohrey. "Grammar-Based Tree Compression". In: *Developments in Language Theory - 19th International Conference, DLT 2015, Liverpool, UK, July 27-30, 2015, Proceedings.* Ed. by Igor Potapov. Vol. 9168. Lecture Notes in Computer Science. Springer, 2015, pp. 46–57. DOI: 10.1007/978-3-319-21500-6_3. URL: https://doi.org/10.1007/978-3-319-21500-6%5C_3.

[37] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. "XML tree structure compression using RePair". In: *Inf. Syst.* 38.8 (2013), pp. 1150–1167. DOI: 10.1016/j.is.2013.06.006. URL: https://doi.org/10.1016/j.is.2013.06.006.

[38] Markus Lohrey, Sebastian Maneth, and Fabian Peternek. "Compressed Tree Canonization". In: *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*. Ed. by Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann. Vol. 9135. Lecture Notes in Computer Science. Springer, 2015, pp. 337–349. DOI: 10.1007/978-3-662-

47666-6_27. URL: https://doi.org/10.1007/978-3-662-47666-6%5C_27.

[39] Markus Lohrey, Sebastian Maneth, and Manfred Schmidt-Schauß. "Parameter reduction and automata evaluation for grammar-compressed trees". In: *J. Comput. Syst. Sci.* 78.5 (2012), pp. 1651–1669. DOI: 10.1016/j.jcss.2012.03.003. URL: https://doi.org/10.1016/j.jcss.2012.03.003.

[40] Sebastian Maneth and Tom Sebastian. "Fast and Tiny Structural Self-Indexes for XML". In: *CoRR* abs/1012.5696 (2010). arXiv: 1012.5696. URL: http://arxiv.org/abs/1012.5696.

[41] John C. Mitchell. *Foundations for programming languages*. Foundation of computing series. MIT Press, 1996. ISBN: 978-0-262-13321-0.

[42] Gonzalo Navarro and Kunihiko Sadakane. "Fully Functional Static and Dynamic Succinct Trees". In: *ACM Trans. Algorithms* 10.3 (2014), 16:1–16:39. DOI: 10.1145/2601073. URL: https://doi.org/10.1145/2601073.

[43] Frank Neven and Thomas Schwentick. "XML schemas without order". Unpublished manuscript. 1999.

[44] Wojciech Plandowski. "Testing Equivalence of Morphisms on Context-Free Languages". In: *Algorithms - ESA '94, Second Annual European Symposium, Utrecht, The Netherlands, September 26-28, 1994, Proceedings*. Ed. by Jan van Leeuwen. Vol. 855. Lecture Notes in Computer Science. Springer, 1994, pp. 460–470. DOI: 10.1007/BFb0049431. URL: https://doi.org/10.1007/BFb0049431.

[45] Baruch Schieber and Uzi Vishkin. "On Finding Lowest Common Ancestors: Simplification and Parallelization". In: *SIAM J. Comput.* 17.6 (1988), pp. 1253–1262. DOI: 10.1137/0217079. URL: https://doi.org/10.1137/0217079.

[46] Thomas Schwentick. "Automata for XML - A survey". In: *J. Comput. Syst. Sci.* 73.3 (2007), pp. 289–315. DOI: 10.1016/j.jcss.2006.10.003. URL: https://doi.org/10.1016/j.jcss.2006.10.003.

[47] Sathya Sundaram and Sanjay Kumar Madria. "A change detection system for unordered XML data using a relational model". In: *Data Knowl. Eng.* 72 (2012), pp. 257–284. DOI: 10.1016/j.datak.2011.11.003. URL: https://doi.org/10.1016/j.datak.2011.11.003.

[48] Sen Zhang, Zhihui Du, and Jason Tsong-Li Wang. "New Techniques for Mining Frequent Patterns in Unordered Trees". In: *IEEE Trans. Cybernetics* 45.6 (2015), pp. 1113–1125. DOI: 10.1109/TCYB.2014.2345579. URL: https://doi.org/10.1109/TCYB.2014.2345579.

PUBLICATIONS

[1] Adrià Gascón, Markus Lohrey, Sebastian Maneth, Carl Philipp Reh, and Kurt Sieber. "Grammar-Based Compression of Unranked Trees". In: *Computer Science - Theory and Applications - 13th International Computer Science Symposium in Russia, CSR 2018, Moscow, Russia, June 6-10, 2018, Proceedings*. Ed. by Fedor V. Fomin and Vladimir V. Podolskii. Vol. 10846. Lecture Notes in Computer Science. Springer, 2018, pp. 118–131. DOI: 10.1007/978-3-319-90530-3_11. URL: https://doi.org/10.1007/978-3-319-90530-3%5C_11.

[2] Danny Hucke, Markus Lohrey, and Carl Philipp Reh. "The Smallest Grammar Problem Revisited". In: *String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings*. Ed. by Shunsuke Inenaga, Kunihiko Sadakane, and Tetsuya Sakai. Vol. 9954. Lecture Notes in Computer Science. 2016, pp. 35–49. DOI: 10.1007/978-3-319-46049-9_4. URL: https://doi.org/10.1007/978-3-319-46049-9%5C_4.

[3] Markus Lohrey, Sebastian Maneth, and Carl Philipp Reh. "Compression of Unordered XML Trees". In: *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy*. Ed. by Michael Benedikt and Giorgio Orsi. Vol. 68. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 18:1–18:17. DOI: 10.4230/LIPIcs.ICDT.2017.18. URL: https://doi.org/10.4230/LIPIcs.ICDT.2017.18.

[4] Markus Lohrey, Sebastian Maneth, and Carl Philipp Reh. "Constant-Time Tree Traversal and Subtree Equality Check for Grammar-Compressed Trees". In: *Algorithmica* 80.7 (2018), pp. 2082–2105. DOI: 10.1007/s00453-017-0331-3. URL: https://doi.org/10.1007/s00453-017-0331-3.

[5] Markus Lohrey, Sebastian Maneth, and Carl Philipp Reh. "Traversing Grammar-Compressed Trees with Constant Delay". In: *2016 Data Compression Conference, DCC 2016, Snowbird, UT, USA, March 30 - April 1, 2016*. Ed. by Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer. IEEE, 2016, pp. 546–555. DOI: 10.1109/DCC.2016.13. URL: https://doi.org/10.1109/DCC.2016.13.

[6] Markus Lohrey, Carl Philipp Reh, and Kurt Sieber. "Size-optimal top dag compression". In: *Inf. Process. Lett.* 147 (2019), pp. 27–31. DOI: 10.1016/j.ipl.2019.03.001. URL: https://doi.org/10.1016/j.ipl.2019.03.001.