Constructing small tree grammars and small circuits for formulas[☆]

Moses Ganardi[b], Danny Hucke[b], Artur Jeż[a], Markus Lohrey[b], Eric Noeth[b]

[a] *University of Wrocław, Poland*
[b] *University of Siegen, Germany*

Abstract

It is shown that every tree of size n over a fixed set of σ different ranked symbols can be decomposed (in linear time as well as in logspace) into $O\big(\frac{n}{\log_\sigma n}\big) = O\big(\frac{n \log \sigma}{\log n}\big)$ many hierarchically defined pieces. Formally, such a hierarchical decomposition has the form of a straight-line linear context-free tree grammar of size $O\big(\frac{n}{\log_\sigma n}\big)$, which can be used as a compressed representation of the input tree. This generalizes an analogous result for strings. Previous grammar-based tree compressors were not analyzed for the worst-case size of the computed grammar, except for the top dag of Bille et al. [6], for which only the weaker upper bound of $O\big(\frac{n}{\log_\sigma^{0.19} n}\big)$ (which was very recently improved to $O\big(\frac{n \cdot \log \log_\sigma n}{\log_\sigma n}\big)$ [22]) for unranked and unlabelled trees has been derived. The main result is used to show that every arithmetical formula of size n, in which only $m \leq n$ different variables occur, can be transformed (in linear time as well as in logspace) into an arithmetical circuit of size $O\big(\frac{n \cdot \log m}{\log n}\big)$ and depth $O(\log n)$. This refines a classical result of Brent from 1974, according to which an arithmetical formula of size n can be transformed into a logarithmic depth circuit of size $O(n)$.

1. Introduction

Grammar-based string compression. *Grammar-based compression* has emerged to an active field in string compression during the past 20 years. The idea is to represent a given string w by a small context-free grammar that generates only s; such a grammar is also called a *straight-line program*, briefly SLP. For instance, the word $(ab)^{1024}$ can be represented by the SLP with the rules $A_0 \to ab$ and $A_i \to A_{i-1}A_{i-1}$ for $1 \leq i \leq 10$ (A_{10} is the start symbol). The size of this grammar is much smaller than the size (length) of the string $(ab)^{1024}$. In general, an SLP of size n (the size of an SLP is usually defined as the total length of all right-hand sides of the rules) can produce a string of length $2^{\Omega(n)}$. Hence, an SLP can be seen indeed as a succinct representation of the generated string. The goal of grammar-based string compression is to construct from a given input string w a small SLP that produces w. Several algorithms for this have been proposed and analyzed. Prominent grammar-based string compressors are for instance LZ78, RePair, and BiSection, see [12] for more details.

To evaluate the compression performance of a grammar-based compressor $\mathcal{C}$, two different approaches can be found in the literature:

(a) One can analyze the maximal size of SLPs produced by $\mathcal{C}$ on strings of length n over the alphabet Σ (the size of Σ is considered to be a constant larger than one in the further discussion). Formally, let

$$g_{\mathcal{C}}(n) = \max_{w \in \Sigma^n} |\mathcal{C}(w)|,$$

[☆] Some of the results in this paper were announced in the short version [23].

Email addresses: ganardi@eti.uni-siegen.de (Moses Ganardi), hucke@eti.uni-siegen.de (Danny Hucke), aje@cs.uni.wroc.pl (Artur Jeż), lohrey@eti.uni-siegen.de (Markus Lohrey), eric.noeth@eti.uni-siegen.de (Eric Noeth)

where $\mathcal{C}(w)$ is the SLP produced by $\mathcal{C}$ on input w, and $|\mathcal{C}(w)|$ is the size of this SLP. An information-theoretic argument shows that for every n there is a binary string of length n, for which a smallest SLP has size $\Omega\big(\frac{n}{\log n}\big)$.[1] Explicit examples of strings for which the smallest SLP has size $\Omega\big(\frac{n}{\log n}\big)$ were constructed in [3, 5, 39]. On the other hand, for many grammar-based compressors $\mathcal{C}$ one gets $g_{\mathcal{C}}(n) \in O\big(\frac{n}{\log n}\big)$ and hence $g_{\mathcal{C}}(n) \in \Theta\big(\frac{n}{\log n}\big)$. This holds for instance for the above mentioned LZ78, RePair, and BiSection, and in fact for all compressors that produce so-called irreducible SLPs [28]. This fact is used in [28] to construct universal string compressors based on grammar-based compressors.

(b) A second approach is to analyze the size of the SLP produced by $\mathcal{C}$ for an input string w compared to the size of a smallest SLP for w. This leads to the approximation ratio for $\mathcal{C}$, which is formally defined as

$$\alpha_{\mathcal{C}}(n) = \max_{w \in \Sigma^n} \frac{|\mathcal{C}(w)|}{g(w)},$$

where $g(w)$ is the size of a smallest SLP for w. It is known that unless $\mathsf{P} = \mathsf{NP}$, there is no polynomial time grammar-based compressor $\mathcal{C}$ such that $\alpha_{\mathcal{C}}(n) < 8569/8568$ for all n [12]. The best known polynomial time grammar-based compressors [12, 25, 26, 41, 42] have an approximation ratio of $O(\log(n/g))$, where g is the size of a smallest SLP for the input string and all of them work in linear time. The algorithms in [12, 26, 41] start with the LZ77-factorization of the input string w and transform it into an SLP of size $O(m \cdot \log(n/g))$, where m is the number of factors of the LZ77-factorization of w, which, by a result of Rytter [41], is bounded by the size of a smallest SLP for w.

Grammar-based tree compression. In this paper, we want to follow approach (a), but for trees instead of strings. A tree in this paper is always a rooted ordered tree over a ranked alphabet, i.e., every node is labelled with a symbol, the rank of this symbol is equal to the number of children of the node and there is an ordering among the children of a node. In [11], grammar-based compression was extended from strings to trees. For this, linear context-free tree grammars were used. Linear context-free tree grammars that produce only a single tree are also known as tree straight-line programs (TSLPs) or straight-line context-free tree grammars (SLCF tree grammars). TSLPs generalize dags (directed acyclic graphs), which are widely used as a compact tree representation. Whereas dags only allow to share repeated subtrees, TSLPs can also share repeated internal tree patterns.

Several grammar-based tree compressors were developed in [2, 8, 11, 27, 34], where the work from [2] is based on another type of tree grammars (elementary ordered tree grammars). The algorithm from [27] achieves an approximation ratio of $O(\log n)$ (for a constant set of node labels). On the other hand, for none of the above mentioned compressors it is known, whether for every input tree with n nodes the size of the output grammar is bounded by $O\big(\frac{n}{\log n}\big)$, as it is the case for many grammar-based string compressors. Recently, it was shown that the so-called *top dag* of an unranked and unlabelled tree of size n has size $O\big(\frac{n}{\log^{0.19} n}\big)$ [6] and this bound has been subsequently improved to $O\big(\frac{n \cdot \log\log_\sigma n}{\log_\sigma n}\big)$ in [22]. The top dag can be seen as a slight variant of a TSLP for an unranked tree.

In this paper, we present a grammar-based tree compressor that transforms a given node-labelled tree of size n with σ different node labels, whose rank is bounded by a constant, into a TSLP of size $O\big(\frac{n}{\log_\sigma n}\big)$ and depth $O(\log n)$, where the depth of a TSLP is the depth of the corresponding derivation tree (we always assume that $\sigma \geq 2$). In particular, for an unlabelled binary tree we get a TSLP of size $O\big(\frac{n}{\log n}\big)$. Our compressor is basically an extension of the BiSection algorithm [29] from strings to trees and is therefore called TreeBiSection. It works in two steps (the following outline works only for binary trees, but it can be easily adapted to trees of higher rank, as long as the rank is bounded by a constant).

[1] If we do not specify the base of a logarithm, we always mean $\log_2(n)$.

In the first step, TreeBiSection hierarchically decomposes in a top-down way the input tree into pieces of roughly equal size. This is a well-known technique that is also known as the $(1/3, 2/3)$-Lemma [31]. But care has to be taken to bound the ranks of the nonterminals of the resulting TSLP. As soon as we get a tree with three holes during the decomposition (which corresponds in the TSLP to a nonterminal of rank three) we have to do an intermediate step that decomposes the tree into two pieces having only two holes each. This may involve an unbalanced decomposition. On the other hand, such an unbalanced decomposition is only necessary in every second step. This trick to bound the number of holes by three was used by Ruzzo [40] in his analysis of space-bounded alternating Turing machines.

The TSLP produced in the first step can be identified with its derivation tree, which has logarithmic depth. Thanks to the fact that all nonterminals have rank at most three, we can encode the derivation tree by a tree with $O(\sigma)$ many labels. Moreover, this derivation tree is weakly balanced in the following sense. For each edge (u, v) in the derivation tree such that both u and v are internal nodes, the derivation tree is balanced at u or v. In a second step, TreeBiSection computes the minimal dag of the derivation tree. Due to its balanced shape, we can show that this minimal dag has size at most $O\big(\frac{n}{\log_\sigma n}\big)$. The nodes of this dag are the nonterminals of our final TSLP.

We prove that the algorithm sketched above can be implemented so that it works in logarithmic space, see Section 5.2. An alternative implementation achieves the running time $O(n \log n)$, but we are not aware of a linear time implementation. Therefore we present in Section 5.6 another algorithm BU-Shrink (for bottom-up shrink) that constructs a TSLP of size $O\big(\frac{n}{\log_\sigma n}\big)$ in linear time (again, we present an outline of the algorithm for the case of binary trees). In a first step, BU-Shrink merges nodes of the input tree in a bottom-up way. Thereby it constructs a partition of the input tree into $O\big(\frac{n}{\log_\sigma n}\big)$ many connected parts of size at most $c \cdot \log_\sigma n$, where c is a suitably chosen constant. Every connected part is a complete subtree, where at most two subtrees were removed (i.e., a subtree with at most two holes) By associating with each such connected part a nonterminal of rank at most two, we obtain a TSLP for the input tree consisting of a start rule $S \to s$, where s consists of $O\big(\frac{n}{\log_\sigma n}\big)$ many nonterminals of rank at most two, each having a right-hand side consisting of $c \cdot \log_\sigma n$ many terminal symbols. By choosing the constant c suitably, we can (using the formula for the number of binary trees of size m, which is given by the Catalan numbers) bound the number of different subtrees of these right-hand sides by $\sqrt{n} \in O\big(\frac{n}{\log_\sigma n}\big)$. This allows to build up the right-hand sides for the non-start nonterminals using $O\big(\frac{n}{\log_\sigma n}\big)$ many nonterminals. A combination of this algorithm with TreeBiSection — we call the resulting algorithm BU-Shrink+TreeBiSection — finally yields a linear time algorithm for constructing a TSLP of size $O\big(\frac{n}{\log_\sigma n}\big)$ and depth $O(\log n)$.

Let us remark that our size bound $O\big(\frac{n}{\log_\sigma n}\big)$ does not contradict any information-theoretic lower bounds (it actually matches the information theoretic limit): Consider for instance unlabelled ordered trees. There are roughly $4^n/\sqrt{\pi n^3}$ such trees with n nodes. Hence under any binary encoding of unlabelled trees, most trees are mapped to bit strings of length at least $2n - o(n)$. But when encoding a TSLP of size m into a bit string, another $\log(m)$-factor arises, because nonterminals have to be encoded by bit strings of length $O(\log m)$. Hence, a TSLP of size $O\big(\frac{n}{\log n}\big)$ is encoded by a bit string of size $O(n)$.

It is also important to note that our size bound $O\big(\frac{n}{\log_\sigma n}\big)$ only holds for trees in which the maximal rank is bounded by a constant. In particular, it does not directly apply to unranked trees (that are, for instance, the standard tree model for XML), which is in contrast to top dags. To overcome this limitation, one can transform an unranked tree into its first-child-next-sibling encoding [30, Paragraph 2.3.2], which is a ranked tree of the same size as the original tree. Then, the first-child-next-sibling encoding can be transformed into a TSLP of size $O\big(\frac{n}{\log_\sigma n}\big)$.

Transforming formulas into circuits. Our main result has an interesting application for the classical problem of transforming formulas into small circuits, which will be presented in Section 6. Spira [43] has shown that for every Boolean formula of size n there exists an equivalent Boolean circuit of depth $O(\log n)$ and size $O(n)$, where the size of a circuit is the number of gates and the

depth of a circuit is the length of a longest path from an input gate to the output gate. Brent [9] extended Spira's theorem to formulas over arbitrary semirings and moreover improved the constant in the $O(\log n)$ bound. Subsequent improvements that mainly concern constant factors can be found in [7, 10]. An easy corollary of our $O\big(\frac{n}{\log_\sigma n}\big)$ bound for TSLPs is that for every (not necessarily commutative) semiring (or field), every formula of size n, in which only $m \leq n$ different variables occur, can be transformed into a circuit of depth $O(\log n)$ and size $O\big(\frac{n \cdot \log m}{\log n}\big)$. Hence, we refine the size bound from $O(n)$ to $O\big(\frac{n \cdot \log m}{\log n}\big)$ (Theorem 23). The transformation can be achieved in logspace and, alternatively, in linear time. Another interesting point of our formula-to-circuit conversion is that most of the construction (namely the construction of a TSLP for the input formula) is purely syntactic. The remaining part (the transformation of the TSLP into a circuit) is straightforward. In contrast, the constructions from [7, 9, 10, 43] construct a log-depth circuit from a formula in one step.

Related work. In view of our logspace implementation of TreeBiSection, it is interesting to remark that Gagie and Gawrychowski considered in [19] the problem of computing in logspace a small SLP for a given string. They present a logspace algorithm that achieves an approximation ratio of $O(\min\{g, \sqrt{n/\log n}\})$, where g is the size of a smallest SLP and n is the length of the input word.

Several papers deal with algorithmic problems on trees that are succinctly represented by TSLPs, see [33] for a survey. Among other problems, equality checking and the evaluation of tree automata can be done in polynomial time for TSLPs.

It is interesting to compare our $O\big(\frac{n}{\log_\sigma n}\big)$ bound with the known bounds for dag compression. A counting argument shows that for all unlabelled binary trees of size n, except for an exponentially small part, the size of a smallest TSLP is $\Omega\big(\frac{n}{\log n}\big)$, and hence (by our main result) $\Theta\big(\frac{n}{\log n}\big)$. This implies that also the average size of the minimal TSLP, where the average is taken for the uniform distribution on unlabelled binary trees of size n, is $\Theta\big(\frac{n}{\log n}\big)$. In contrast, the average size of the minimal dag for trees of size n is $\Theta\big(\frac{n}{\sqrt{\log n}}\big)$ [18], whereas the worst-case size of the dag is n.

In the context of tree compression, succinct data structures for trees are another big topic. There, the goal is to represent a tree in a number of bits that asymptotically matches the information theoretic lower bound, and at the same time allows efficient querying (in the best case in time $O(1)$). For unlabelled unranked trees of size n there exist representations with $2n + o(n)$ bits that support navigation in time $O(1)$ [24, 37]. This result has been extended to labelled trees, where $(\log \sigma) \cdot n + 2n + o(n)$ bits suffice when σ is the number of node labels [16]. See [38] for a survey.

2. Computational models

We will consider time and space bounds for computational problems. For time bounds, we will use the standard RAM model. We make the assumption that for an input tree of size n, arithmetical operations on numbers with $O(\log n)$ bits can be carried out in time $O(1)$. We assume that the reader has some familiarity with logspace computations, see e.g. [4, Chapter 4.1] for more details. A function can be computed in logspace, if it can be computed on a Turing machine with three tapes: a read-only input tape, a write-only output tape, and a read-write working tape of length $O(\log n)$, where n is the length of the input. It is an important fact that if functions f and g can be computed in logspace, then the composition of f and g can be computed in logspace as well. We will use this fact implicitly all over the paper.

3. Strings and Straight-Line Programs

Before we come to grammar-based tree compression, let us briefly discuss grammar-based string compression. A *straight-line program*, briefly SLP, is a context-free grammar that produces a single string. Formally, it is defined as a tuple $\mathcal{G} = (N, \Sigma, S, P)$, where N is a finite set of nonterminals, Σ is a finite set of terminal symbols ($\Sigma \cap N = \emptyset$), $S \in N$ is the start nonterminal, and P is a finite set of rules of the form $A \to w$ for $A \in N$, $w \in (N \cup \Sigma)^*$ such that the following conditions hold:

- There do not exist rules $(A \to u)$ and $(A \to v)$ in P with $u \neq v$.

- The binary relation $\{(A, B) \in N \times N \mid (A \to w) \in P,\ B \text{ occurs in } w\}$ is acyclic.

These conditions ensure that every nonterminal $A \in N$ produces a unique string $\mathrm{val}_{\mathcal{G}}(A) \in \Sigma^*$. The string defined by $\mathcal{G}$ is $\mathrm{val}(\mathcal{G}) = \mathrm{val}_{\mathcal{G}}(S)$. The size of the SLP $\mathcal{G}$ is $|\mathcal{G}| = \sum_{(A \to w) \in P} |w|$, where $|w|$ denotes the length of the string w. SLPs are also known as *word chains* in the area of combinatorics on words [5, 14].

A simple induction shows that for every SLP $\mathcal{G}$ of size m one has $|\mathrm{val}(\mathcal{G})| \leq O(3^{m/3})$ [12, proof of Lemma 1]. On the other hand, it is straightforward to define an SLP $\mathcal{H}$ of size $2n$ such that $|\mathrm{val}(\mathcal{H})| \geq 2^n$. This justifies to see an SLP $\mathcal{G}$ as a compressed representation of the string $\mathrm{val}(\mathcal{G})$, and exponential compression rates can be achieved in this way.

Let $\sigma \geq 2$ be the size of the terminal alphabet Σ.[2] It is well-known that for every string $w \in \Sigma^*$ of length n there exists an SLP $\mathcal{G}$ of size $O\big(\frac{n}{\log_\sigma n}\big)$ such that $\mathrm{val}(\mathcal{G}) = w$, see e.g. [28]. On the other hand, it was shown in [3] that for every n there exists a word w of length n over an alphabet of size σ such that every SLP for w has size at least $\frac{n}{\log_\sigma n}$.

4. Trees and Tree Straight-Line Programs

In this paper, we will deal with *ranked trees*, where nodes are labelled with symbols. Every symbol has an associated rank (a natural number). If a node is labelled with a symbol of rank i, then it is required that the node has i children. Moreover, these children are linearly ordered.

For the purpose of defining tree grammars we need two types of ranked symbols: Terminals and nonterminals. For each of these two types and every rank i we assume to have a pool of countably many symbols. Moreover, we need a third type of symbols, so called parameters, which are treated as symbols of rank i. Intuitively, these parameters specify positions in a tree, which can be substituted by other trees.

Formally, for every $i \geq 0$ we fix a countably infinite set $\mathcal{F}_i$ of *terminals of rank* i and a countably infinite set $\mathcal{N}_i$ of *nonterminals of rank* i. Let $\mathcal{F} = \bigcup_{i \geq 0} \mathcal{F}_i$ and $\mathcal{N} = \bigcup_{i \geq 0} \mathcal{N}_i$. Moreover, let $\mathcal{X} = \{x_1, x_2, \dots\}$ be a countably infinite set of *parameters*. We assume that the three sets $\mathcal{F}$, $\mathcal{N}$, and $\mathcal{X}$ are pairwise disjoint. A *labelled tree* $t = (V, \lambda)$ is a finite, rooted and ordered tree t with node set V, whose nodes are labelled by elements from $\mathcal{F} \cup \mathcal{N} \cup \mathcal{X}$. The function $\lambda : V \to \mathcal{F} \cup \mathcal{N} \cup \mathcal{X}$ denotes the labelling function. We require that a node $v \in V$ with $\lambda(v) \in \mathcal{F}_k \cup \mathcal{N}_k$ has exactly k children, which are ordered from left to right. We also require that every node v with $\lambda(v) \in \mathcal{X}$ is a leaf of t. The size of t is $|t| = |\{v \in V \mid \lambda(v) \in \mathcal{F} \cup \mathcal{N}\}|$, i.e., we do not count parameters.

We denote trees in their usual term notation, e.g. $b(a, a)$ denotes the tree with root node labelled by b and two children, both labelled by a. We define $\mathcal{T}$ as the set of all labelled trees. The *depth* of a tree $t = (V, \lambda)$ is the maximal length (number of edges) of a path from the root to a leaf. Let $\mathrm{labels}(t) = \{\lambda(v) \mid v \in V\}$. For $\mathcal{L} \subseteq \mathcal{F} \cup \mathcal{N} \cup \mathcal{X}$ we let $\mathcal{T}(\mathcal{L}) = \{t \mid \mathrm{labels}(t) \subseteq \mathcal{L}\}$. We write $<_t$ for the depth-first-order on V. Formally, $u <_t v$ if u is an ancestor of v or if there exists a node w and $i < j$ such that the i^{th} child of w is an ancestor of u and the j^{th} child of w is an ancestor of v. The tree $t \in \mathcal{T}$ is a *pattern* if there do not exist different nodes that are labelled with the same parameter. For example, $f(x_1, x_1, x_3)$ is not a pattern, whereas $f(x_1, x_{21}, x_{99})$ and $f(x_1, x_2, x_3)$ are patterns. A pattern $t = (V, \lambda) \in \mathcal{T}$ is *valid* if (i) $\mathrm{labels}(t) \cap \mathcal{X} = \{x_1, \dots, x_n\}$ for some $n \geq 0$ and (ii) for all $u, v \in V$ with $\lambda(u) = x_i$, $\lambda(v) = x_j$ and $u <_t v$ we have $i < j$. For example the pattern $f(x_1, x_{21}, x_{99})$ is not valid, whereas $f(x_1, x_2, x_3)$ is valid. For a pattern t we define $\mathrm{valid}(t)$ as the unique valid pattern which is obtained from t by renaming the parameters. For instance, $\mathrm{valid}(f(x_{21}, x_2, x_{99})) = f(x_1, x_2, x_3)$. A valid pattern t in which the parameters $x_1, \dots, x_n$ occur is also written as $t(x_1, \dots, x_n)$ and we write $\mathrm{rank}(t) = n$ (the *rank* of the pattern). We say that a valid pattern p of rank n *occurs* in a tree t if there exist n trees $t_1, \dots, t_n$ such that $p(t_1, \dots, t_n)$ (the tree obtained from p by replacing the parameter x_i by t_i for $1 \leq i \leq n$) is a subtree of t.

The following counting lemma will be needed several times:

[2] The case $\sigma = 1$ is not interesting, since every string of length n over a unary alphabet can be produced by an SLP of size $O(\log n)$.

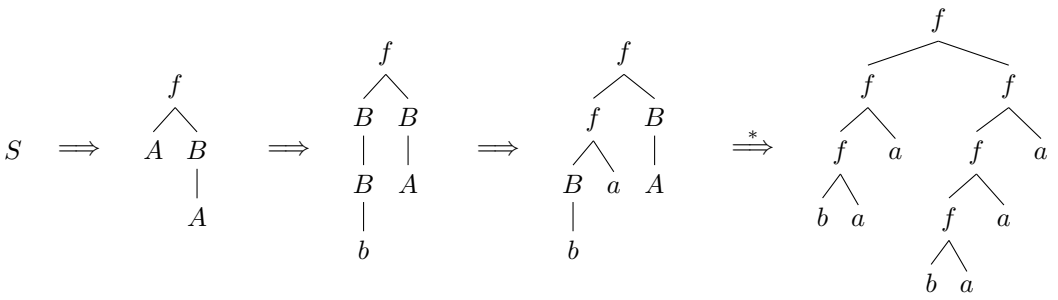

Figure 1: A derivation of the TSLP from Example 2.

Lemma 1. *The number of trees $t \in \mathcal{T}(\mathcal{F})$ with $1 \leq |t| \leq n$ and $|\text{labels}(t)| \leq \sigma$ is bounded by $\frac{4}{3}(4\sigma)^n$.*

Proof. The number of different rooted ordered (but unranked) trees with k nodes is $\frac{1}{k+1}\binom{2k}{k} \leq 4^k$ (the k^{th} Catalan number, see e.g. [44]). Hence, the number of trees in the lemma can be bounded by

$$\sigma^n \cdot \sum_{k=1}^{n} 4^k \leq \sigma^n \cdot \frac{4^{n+1}-1}{3} \leq \frac{4}{3}(4\sigma)^n.$$

□

We now define a particular form of context-free tree grammars (see [13] for more details on context-free tree grammars) with the property that exactly one tree is derived. A *tree straight-line program (TSLP)* is a tuple $\mathcal{G} = (N, \Sigma, S, P)$, where $N \subseteq \mathcal{N}$ is a finite set of nonterminals, $\Sigma \subseteq \mathcal{F}$ is a finite set of terminals, $S \in N \cap \mathcal{N}_0$ is the start nonterminal, and P is a finite set of rules (or productions) of the form $A(x_1, \ldots, x_n) \to t(x_1, \ldots, x_n)$ (which is also briefly written as $A \to t$), where $n \geq 0$, $A \in N \cap \mathcal{N}_n$ and $t(x_1, \ldots, x_n) \in \mathcal{T}(N \cup \Sigma \cup \{x_1, \ldots, x_n\})$ is a valid pattern such that the following conditions hold:

- For every $A \in N$ there is exactly one tree t such that $(A \to t) \in P$.
- The binary relation $\{(A, B) \in \mathcal{N} \times \mathcal{N} \mid (A \to t) \in P, B \in \text{labels}(t)\}$ is acyclic.

Note that N and Σ are implicitly defined by the rules from P. Therefore, we can (and always will) write a TSLP as a pair $\mathcal{G} = (S, P)$ consisting of rules and a start nonterminal.

The above conditions ensure that from every nonterminal $A \in N \cap \mathcal{N}_n$ exactly one valid pattern $\text{val}_{\mathcal{G}}(A) \in \mathcal{T}(\mathcal{F} \cup \{x_1, \ldots, x_n\})$ is derived by using the rules as rewrite rules in the usual sense. The tree defined by $\mathcal{G}$ is $\text{val}(\mathcal{G}) = \text{val}_{\mathcal{G}}(S)$. Instead of giving a formal definition, we show a derivation of $\text{val}(\mathcal{G})$ from S in an example:

Example 2. *Let $\mathcal{G} = (S, P)$, $S, A \in \mathcal{N}_0, B \in \mathcal{N}_1$, $a, b \in \mathcal{F}_0$, $f \in \mathcal{F}_2$ and*

$$P = \{S \to f(A, B(A)),\ A \to B(B(b)),\ B(x_1) \to f(x_1, a)\}.$$

A possible derivation of $\text{val}(\mathcal{G})$ *from S is depicted in Figure 1.*

The size $|\mathcal{G}|$ of a TSLP $\mathcal{G} = (S, P)$ is the total size of all trees on the right-hand sides of P, i.e. $|\mathcal{G}| = \sum_{(A \to t) \in P} |t|$. For instance, the TSLP from Example 2 has size 9.

Note that for the size of a TSLP we do not count nodes of right-hand sides that are labelled with a parameter. To justify this, we use the following internal representation of valid patterns (which is also used in [27]): For every non-parameter node v of a tree, with children $v_1, \ldots, v_n$ we store in a list all pairs (i, v_i) such that v_i is a non-parameter node. Moreover, we store for every symbol (node label) its rank. This allows to reconstruct the valid pattern, since we know the positions where parameters have to be inserted.

A TSLP is in *Chomsky normal form* if for every rule $A(x_1, \ldots, x_n) \to t(x_1, \ldots, x_n)$ one of the following two cases holds:

$$t(x_1, \ldots, x_n) = B(x_1, \ldots, x_{i-1}, C(x_i, \ldots, x_k), x_{k+1}, \ldots, x_n) \text{ for } B, C \in \mathcal{N} \quad (1)$$
$$t(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \text{ for } f \in \mathcal{F}_n. \quad (2)$$

If the tree t in the corresponding rule $A \to t$ is of type (1), we write $\text{index}(A) = i$. If otherwise t is of type (2), we write $\text{index}(A) = 0$. One can transform every TSLP efficiently into an equivalent TSLP in Chomsky normal form with a small size increase. More precisely, it is shown in [36] that from a TSLP $\mathcal{G}$, where all terminals and nonterminals have rank at most r, one can construct in time $O(r \cdot |\mathcal{G}|)$ an equivalent TSLP in Chomsky normal form. We mainly consider TSLPs in Chomsky normal form in the following.

We define the rooted, ordered derivation tree $\mathcal{D}_\mathcal{G}$ of a TSLP $\mathcal{G} = (S, P)$ in Chomsky normal form as for string grammars: The inner nodes of the derivation tree are labelled by nonterminals and the leaves are labelled by terminal symbols. Formally, we start with the root node of $\mathcal{D}_\mathcal{G}$ and assign it to the label S. Then, for every nonterminal A whose corresponding right-hand side is of type (1) and every node in $D_\mathcal{G}$ labelled by A, we attach a left child labelled by B and a right child labelled by C. If the right-hand side of the rule for A is of type (2), we attach a single child labelled by f to A. Note that these nodes are the leaves of $\mathcal{D}_\mathcal{G}$ and they represent the nodes of the tree $\text{val}(\mathcal{G})$. We denote by $\text{depth}(\mathcal{G})$ the depth of the derivation tree $\mathcal{D}_\mathcal{G}$.

A TSLP is called *monadic* if every nonterminal has rank at most one. The following result was shown in [36]:

Theorem 3. *From a given TSLP $\mathcal{G}$ in Chomsky normal form such that every nonterminal has rank at most k and every terminal symbol has rank at most r, one can compute in time $O(|\mathcal{G}| \cdot k \cdot r)$ a monadic TSLP $\mathcal{H}$ with the following properties:*

- $\text{val}(\mathcal{G}) = \text{val}(\mathcal{H})$,
- $|\mathcal{H}| \in O(|\mathcal{G}| \cdot r)$,
- $\text{depth}(\mathcal{H}) \in O(\text{depth}(\mathcal{G}))$.

Moreover, one can assume that every production of $\mathcal{H}$ has one of the following four forms

- $A \to B(C)$,
- $A(x_1) \to B(C(x_1))$,
- $A \to f(A_1, \ldots, A_n)$,
- $A(x_1) \to f(A_1, \ldots, A_{i-1}, x_1, A_{i+1}, \ldots, A_n)$,

where $A, B, C, A_1, \ldots, A_n$ are nonterminals of rank 0 or 1, and f is a terminal symbol.

A commonly used tree compression scheme is obtained by writing down repeated subtrees only once. In that case all occurrences except for the first are replaced by a pointer to the first one. This leads to a node-labelled *directed acyclic graph* (dag). It is known that every tree has a unique minimal dag, which is called *the dag* of the initial tree. An example can be found in Figure 4, where the right graph is the dag of the middle tree. A dag can be seen as a TSLP in which every nonterminal has rank zero: The nonterminals are the nodes of the dag. A node v with label f and n children $v_1, \ldots, v_n$ corresponds to the rule $v \to f(v_1, \ldots, v_n)$. The root of the dag is the start variable. Vice versa, it is straightforward to transform a TSLP, in which every variable has rank zero, into an equivalent dag.

The dag of a tree t can be constructed in time $O(|t|)$ [15]. The following lemma shows that the dag of a tree can be also constructed in logspace.

Lemma 4. *The dag of a given tree can be computed in logspace.*

Proof. Assume that the node set of the input tree t is $\{1, \dots, n\}$. We denote by $t[i]$ the subtree of t rooted at node i. Given two nodes i, j of t one can verify in logspace whether the subtrees $t[i]$ and $t[j]$ are isomorphic (we write $t[i] \cong t[j]$ for this) by performing a preorder traversal over both trees and thereby comparing the two trees symbol by symbol.

The nodes and edges of the dag of t can be enumerated in logspace as follows. A node i of t is a node of the dag if there is no $j < i$ with $t[i] \cong t[j]$. By the above remark, this can be checked in logspace. Let i be a node of the dag and let j be the k^{th} child of i in t. Then j' is the k^{th} child of i in the dag where j' is the smallest number such that $t[j'] \cong t[j]$. Again by the above remark this j' can be found in logspace. □

5. Constructing a small TSLP for a tree

Let t be a tree of size n with σ many different node labels. In this section we present two algorithms. Each construct a TSLP for t of size $O\big(\frac{n}{\log_\sigma n}\big)$ and depth $O(\log n)$, assuming the maximal rank of symbols is bounded by a constant. Our first algorithm TreeBiSection achieves this while only using logarithmic space, but needs time $O(n \cdot \log n)$. The second algorithm first reduces the size of the input tree and then performs TreeBiSection on the resulting tree, which yields a linear running time.

5.1. TreeBiSection

TreeBiSection uses the well-known idea of splitting a tree recursively into smaller parts of roughly equal size, see e.g. [9, 43]. It is a generalization of the grammar-based string compressor BiSection [29]. The latter algorithm first splits an input word w with $|w| \geq 2$ as $w = w_1 w_2$ such that $|w_1| = 2^j$ for the unique number $j \geq 0$ with $2^j < |w| \leq 2^{j+1}$. This process is recursively repeated with w_1 and w_2 until we obtain words of length 1. During the process, BiSection introduces a nonterminal for each distinct factor of length at least two and creates a rule with two symbols on the right-hand side corresponding to the split.

For a valid pattern $t = (V, \lambda) \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$ and a node $v \in V$ we denote by $t[v]$ the tree valid(s), where s is the subtree rooted at v in t. We further write $t \setminus v$ for the tree valid(r), where r is obtained from t by replacing the subtree rooted at v by a new parameter. If for instance $t = h(g(x_1, f(x_2, x_3)), x_4)$ and v is the f-labelled node, then $t[v] = f(x_1, x_2)$ and $t \setminus v = h(g(x_1, x_2), x_3)$. The following lemma is well-known, at least for binary trees; see e.g. [31].

Lemma 5. *Let $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$ be a tree with $|t| \geq 2$ such that every node has at most r children (where $r \geq 1$). Then there is a node v such that*

$$\frac{1}{2(r+2)} \cdot |t| \;\leq\; |t[v]| \;\leq\; \frac{r+1}{r+2} \cdot |t|.$$

Proof. We start a search at the root node, checking at each node v whether $|t[v]| \leq \frac{d+1}{d+2} \cdot |t|$, where d is the number of children of v. If the property does not hold, we continue the search at a child that spawns a largest subtree (using an arbitrary deterministic tie-breaking rule that is fixed for the further discussion). Note that we eventually reach a node such that $|t[v]| \leq \frac{d+1}{d+2} \cdot |t|$: If $|t[v]| = 1$, then $|t[v]| \leq \frac{1}{2}|t|$ since $|t| \geq 2$.

So, let v be the first node with $|t[v]| \leq \frac{d+1}{d+2} \cdot |t|$, where d is the number of children of v. We get

$$|t[v]| \leq \frac{d+1}{d+2} \cdot |t| \leq \frac{r+1}{r+2} \cdot |t|.$$

Moreover v cannot be the root node. Let u be its parent node and let e be the number of children of u. Since v spans a largest subtree among the children of u, we get $e \cdot |t[v]| + 1 \geq |t[u]| \geq \frac{e+1}{e+2} \cdot |t|$,

i.e.,

$$\begin{aligned} |t[v]| &\geq \frac{e+1}{e(e+2)} \cdot |t| - \frac{1}{e} \\ &= \left(\frac{e+1}{e(e+2)} - \frac{1}{|t| \cdot e} \right) \cdot |t| \\ &\geq \left(\frac{e+1}{e(e+2)} - \frac{1}{2e} \right) \cdot |t| \\ &= \frac{1}{2(e+2)} \cdot |t| \\ &\geq \frac{1}{2(r+2)} \cdot |t|. \end{aligned}$$

This proves the lemma. □

For the remainder of this section we refer with split(t) to the unique node in a tree t which is obtained by the procedure from the proof above. Based on Lemma 5 we now construct a TSLP $\mathcal{G}_t = (S, P)$ with $\mathrm{val}(\mathcal{G}_t) = t$ for a given tree t. It is *not* the final TSLP produced by TreeBiSection. For our later analysis, it is important to bound the number of parameters in the TSLP $\mathcal{G}_t$ by a constant. To achieve this, we use an idea from Ruzzo's paper [40].

We will first present the construction and analysis of $\mathcal{G}_t$ only for trees in which every node has at most two children, i.e., we consider trees from $\mathcal{T}(\mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2)$. Let us write $\mathcal{F}_{\leq 2}$ for $\mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2$. In Section 5.5, we will consider trees of larger branching degree. For the case that $r = 2$, Lemma 5 yields for every tree $s \in \mathcal{T}(\mathcal{F}_{\leq 2} \cup \mathcal{X})$ with $|s| \geq 2$ a node $v = \mathrm{split}(s)$ such that

$$\frac{1}{8} \cdot |s| \;\leq\; |s[v]| \;\leq\; \frac{3}{4} \cdot |s|. \tag{3}$$

Consider an input tree $t \in \mathcal{T}(\mathcal{F}_{\leq 2})$ (we assume that $|t| \geq 2$). The TSLP $\mathcal{G}_t$ we are going to construct has the property that every nonterminal has rank at most three. Our algorithm stores two sets of rules, P_{temp} and P_{final}. The set P_{final} will contain the rules of the TSLP $\mathcal{G}_t$ and the rules from P_{temp} ensure that the TSLP $(S, P_{\mathrm{temp}} \cup P_{\mathrm{final}})$ produces t at any given time of the procedure. We start with the initial setting $P_{\mathrm{temp}} = \{S \to t\}$ and $P_{\mathrm{final}} = \emptyset$. While P_{temp} is non-empty we proceed for each rule $(A \to s) \in P_{\mathrm{temp}}$ as follows:

Remove the rule from P_{temp}. Let $A \in \mathcal{N}_r$. If $r \leq 2$ we determine the node $v = \mathrm{split}(s)$ in s. Then we split the tree s into the two trees $s[v]$ and $s \setminus v$. Let $r_1 = \mathrm{rank}(s[v])$, $r_2 = \mathrm{rank}(s \setminus v)$ and let $A_1 \in \mathcal{N}_{r_1}$ and $A_2 \in \mathcal{N}_{r_2}$ be fresh nonterminals. Note that $r = r_1 + r_2 - 1$. If the size of $s[v]$ ($s \setminus v$, respectively) is larger than 1 we add the rule $A_1 \to s[v]$ ($A_2 \to s \setminus v$, respectively) to P_{temp}. Otherwise we add it to P_{final} as a final rule. Let k be the number of nodes of s that are labelled by a parameter and that are smaller (w.r.t. $<_s$) than v. To link the nonterminal A to the fresh nonterminals A_1 and A_2 we add the rule

$$A(x_1, \ldots, x_r) \to A_1(x_1, \ldots, x_k, A_2(x_{k+1}, \ldots, x_{k+r_2}), x_{k+r_2+1}, \ldots, x_r)$$

to the set of final rules P_{final}.

To bound the rank of the introduced nonterminals by three we handle rules $A \to s$ with $A \in \mathcal{N}_3$ as follows. Let v_1, v_2, and v_3 be the nodes of s labelled by the parameters x_1, x_2, and x_3, respectively. Instead of choosing the node v by split(s) we set v to the lowest common ancestor of (v_1, v_2) or (v_2, v_3), depending on which one has the greater distance from the root node (see Figure 2). This step ensures that the two trees $s[v]$ and $s \setminus v$ have rank 2, so in the next step each of these two trees will be split in a balanced way according to (3). As a consequence, the resulting TSLP $\mathcal{G}_t$ has depth $O(\log |t|)$ but size $O(|t|)$. Also note that $\mathcal{G}_t$ is in Chomsky normal form.

Example 6. *Let t_n be the complete binary tree of height n, i.e. $t_0 = a$ and $t_{n+1} = f(t_n, t_n)$ where $f \in \mathcal{F}_2$ and $a \in \mathcal{F}_0$. Figure 3 illustrates how the tree t_3 is decomposed hierarchically during the algorithm. We only explain the first steps of the algorithm. Final rules are framed.*

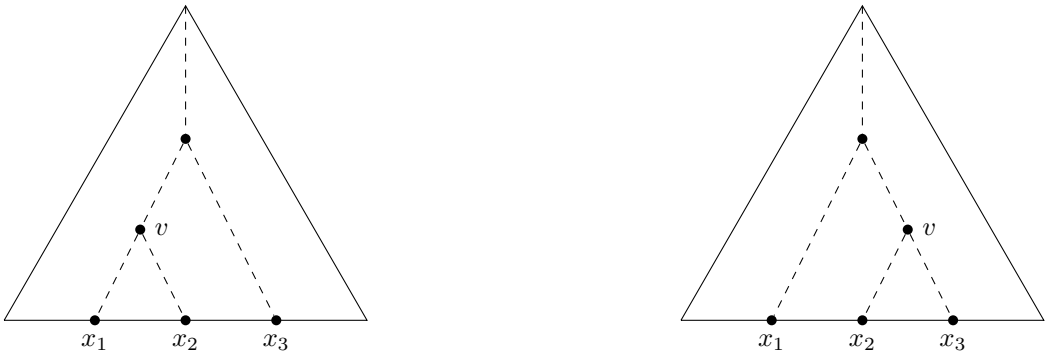

Figure 2: Splitting a tree with three parameters

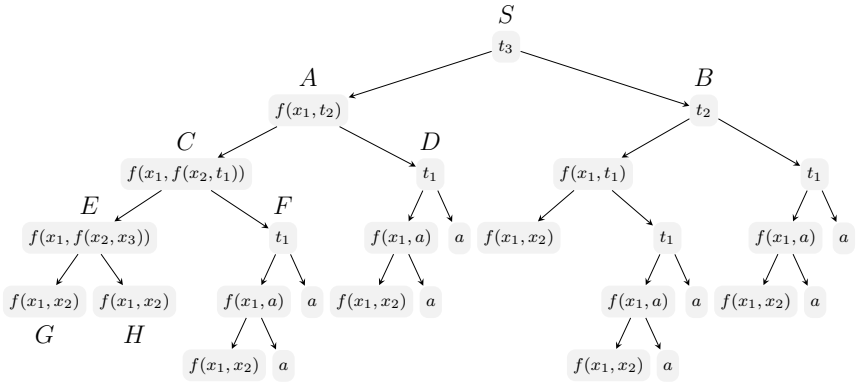

Figure 3: The hierarchical decomposition of a tree produced by TreeBiSection

.

1. *We start with the rule* $S \to t_3$.
2. *Possible split nodes of* t_3 *are the children of its root. We split* $S \to t_3$ *into the rules* $\boxed{S \to A(B),}$ $A(x_1) \to f(x_1, t_2)$ *and* $B \to t_2$.
3. *We continue to split the rule* $A(x_1) \to f(x_1, t_2)$. *After two more steps we obtain the rules* $\boxed{A(x_1) \to C(x_1, D),}$ $D \to t_1$, $\boxed{C(x_1, x_2) \to E(x_1, x_2, F),}$ $E(x_1, x_2, x_3) \to f(x_1, f(x_2, x_3))$ *and* $F \to t_1$.
4. *Since the nonterminal* E *has rank 3, we have to decompose* $f(x_1, f(x_2, x_3))$ *along the lowest common ancestor of two parameters as described above, in this case* x_2 *and* x_3.[3] *Hence, the rule for the nonterminal* E *is split into the rules* $\boxed{E(x_1, x_2, x_3) \to G(x_1, H(x_2, x_3))}$, $\boxed{G(x_1, x_2) \to f(x_1, x_2)}$ *and* $\boxed{H(x_1, x_2) \to f(x_1, x_2).}$

In the next step we want to compact the TSLP by considering the dag of the derivation tree. For this we first build the derivation tree $\mathcal{D}_t := \mathcal{D}_{\mathcal{G}_t}$ from the TSLP $\mathcal{G}_t$ as described above.

We now want to identify nonterminals that produce the same tree. Note that if we just omit the nonterminal labels from the derivation tree, then there might exist isomorphic subtrees of the derivation tree whose root nonterminals produce different trees. This is due to the fact that we lost for an A-labelled node of the derivation tree with a left (right, respectively) child that is labelled with B (C, respectively) the information at which argument position of B the nonterminal

[3]Here the least common ancestor of x_2 and x_3 happens to coincide with the node split$(f(x_1, f(x_2, x_3)))$.

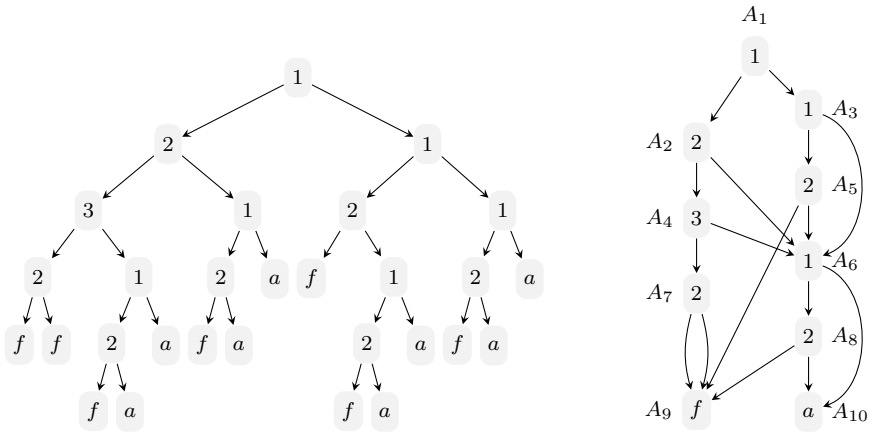

Figure 4: Modified derivation tree of the TSLP from Example 6 and its minimal dag.

C is substituted. This information is exactly given by $\text{index}(A) \in \{0, 1, 2, 3\}$, which was defined in Section 4. Moreover, we remove every leaf v and write its label into its parent node. We call the resulting tree the *modified derivation tree* and denote it by $\mathcal{D}_t^*$. Note that $\mathcal{D}_t^*$ is a full binary tree with node labels from $\{1, 2, 3\} \cup \text{labels}(t)$. The modified derivation tree for Example 6 is shown on the left of Figure 4.

The following lemma shows how to compact our grammar by considering the dag of $\mathcal{D}_t^*$.

Lemma 7. *Let u and v be nodes of $\mathcal{D}_t$ labelled by A and B, respectively. Moreover, let u' and v' be the corresponding nodes in $\mathcal{D}_t^*$. Then, the subtrees $\mathcal{D}_t^*[u']$ and $\mathcal{D}_t^*[v']$ are isomorphic (as labelled ordered trees) if and only if $\text{val}_{\mathcal{G}_t}(A) = \text{val}_{\mathcal{G}_t}(B)$.*

Proof. For the if-direction assume that $\text{val}_{\mathcal{G}_t}(A) = \text{val}_{\mathcal{G}_t}(B)$. Hence, when producing the TSLP $\mathcal{G}_t$, the intermediate productions $A \to \text{val}_{\mathcal{G}_t}(A)$ and $B \to \text{val}_{\mathcal{G}_t}(B)$ are split in exactly the same way (since the splitting process is deterministic). This implies that $\mathcal{D}_t^*[u']$ and $\mathcal{D}_t^*[v']$ are isomorphic.

We prove the other direction by induction over the size of the trees $\mathcal{D}_t^*[u']$ and $\mathcal{D}_t^*[v']$. Consider u and v labelled by A and B, respectively. We have $\text{index}(A) = \text{index}(B) = i$. For the induction base assume that $i = 0$. Then u' and v' are both leaves labelled by the same terminal. Hence, $\text{val}_{\mathcal{G}_t}(A) = \text{val}_{\mathcal{G}_t}(B)$ holds. For the induction step assume that $i > 0$. Let A_1 (B_1, respectively) be the label of the left child of u (v, respectively) and let A_2 (B_2, respectively) be the label of the right child of u (v, respectively). By induction, we get $\text{val}_{\mathcal{G}_t}(A_1) = \text{val}_{\mathcal{G}_t}(B_1) = s(x_1, \ldots, x_m)$ and $\text{val}_{\mathcal{G}_t}(A_2) = \text{val}_{\mathcal{G}_t}(B_2) = t(x_1, \ldots, x_n)$. Therefore, $\text{rank}(A) = \text{rank}(B) = m + n - 1$ and $\text{val}_{\mathcal{G}_t}(A) = s(x_1, ..., x_{i-1}, t(x_i, ..., x_{i+n-1}), x_{i+n}, ..., x_{n+m-1}) = \text{val}_{\mathcal{G}_t}(B)$. □

By Lemma 7, if two subtrees of $\mathcal{D}_t^*$ are isomorphic we can eliminate the nonterminal of the root node of one subtree. Hence, we can compact our TSLP by constructing the minimal dag d of $\mathcal{D}_t^*$. The minimal dag of the TSLP of Example 6 is shown on the right of Figure 4. The nodes of d are the nonterminals of the final TSLP produced by TreeBiSection. For a nonterminal that corresponds to an inner node of d (a leaf of d, respectively), we obtain a rule whose right-hand side has the form (1) ((2), respectively). Let n_1 be the number of inner nodes of d and n_2 be the number of leaves. Then the size of our final TSLP is $2n_1 + n_2$, which is bounded by twice the number of nodes of d.

Example 8. *We continue Example 6 and obtain the final TSLP from the minimal dag d of $\mathcal{D}_t^*$ shown in Figure 4 on the right. We assign to each node of the minimal dag a fresh nonterminal*

and define the rules according to the labels as follows. Here the start nonterminal is A_1.

$$\begin{aligned} A_1 &\to A_2(A_3) & A_2(x_1) &\to A_4(x_1, A_6) \\ A_3 &\to A_5(A_6) & A_4(x_1, x_2) &\to A_7(x_1, x_2, A_6) \\ A_5(x_1) &\to A_9(x_1, A_6) & A_6 &\to A_8(A_{10}) \\ A_7(x_1, x_2, x_3) &\to A_9(x_1, A_9(x_2, x_3)) & A_8(x_1) &\to A_9(x_1, A_{10}) \\ A_9(x_1, x_2) &\to f(x_1, x_2) & A_{10} &\to a \end{aligned}$$

Algorithm 1 shows the pseudocode of TreeBiSection. There, we denote for a valid pattern $s(x_1, \ldots, x_k)$ by $\mathrm{lca}_s(x_i, x_j)$ the lowest common ancestor of the unique leaves that are labelled with x_i and x_j.

In Section 5.2 we will analyze the running time of TreeBiSection, and we will present a logspace implementation. In Sections 5.3 and 5.4 we will analyze the size of the produced TSLP.

5.2. Running time and space consumption of TreeBiSection

In this section, we show that TreeBiSection can be implemented so that it works in logspace, and alternatively in time $O(n \cdot \log n)$. Note that these are two different implementations.

Lemma 9. *Given a tree $t \in \mathcal{T}(\mathcal{F}_{\leq 2})$ of size n one can compute (i) in time $O(n \log n)$ and (using a different algorithm) (ii) in logspace the TSLP produced by* TreeBiSection *on input t.*

Proof. Let t be the input tree of size n. The dag of a tree can be computed in (i) linear time [15] and (ii) in logspace by Lemma 4. Hence, it suffices to show that the modified derivation tree $\mathcal{D}_t^*$ for t can be computed in time $O(n \cdot \log n)$ as well as in logspace.

For the running time let us denote with $P_{\mathrm{temp},i}$ the set of productions P_{temp} after i iterations of the while loop. Moreover, let n_i be the sum of the sizes of all right-hand sides in $P_{\mathrm{temp},i}$. Then, we have $n_{i+1} \leq n_i$: When a single rule $A \to s$ is replaced with $A_1 \to t_1$ and $A_2 \to t_2$ then each non-parameter node in t_1 or t_2 is one of the nodes of s. Hence, we have $|s| = |t_1| + |t_2|$ (recall that we do not count parameters for the size of a tree). We might have $n_{i+1} < n_i$ since rules with a single terminal symbol on the right-hand side are put into P_{final}. We obtain $n_i \leq n$ for all i. Hence, splitting all rules in $P_{\mathrm{temp},i}$ takes time $O(n)$, and a single iteration of the while loop takes time $O(n)$ as well. On the other hand, since every second split reduces the size of the tree to which the split is applied by a constant factor (see (3)). Hence, the while loop is iterated at most $O(\log n)$ times. This gives the time bound.

The inquisitive reader may wonder whether our convention of neglecting parameter nodes for the size of a tree affects the linear running time. This is not the case: Every right-hand side s in $P_{\mathrm{temp},i}$ has at most three parameters, i.e., the total number of nodes in s is at most $|s| + 3 \leq 4|s|$. This implies that the split node can be computed in time $O(|s|)$. Doing this for all right-hand sides in $P_{\mathrm{temp},i}$ yields the time bound $O(n)$ as above.

For the logspace version, we first describe how to represent a single valid pattern occurring in t in logspace and how to compute its split node. Let $s(x_1, \ldots, x_k)$ be a valid pattern which occurs in t and has k parameters where $0 \leq k \leq 3$, i.e., $s(t_1, \ldots, t_k)$ is a subtree of t for some subtrees $t_1, \ldots, t_k$ of t. We represent the tree $s(x_1, \ldots, x_k)$ by the tuple $\mathrm{rep}(s) = (v_0, v_1 \ldots, v_k)$ where $v_0, v_1, \ldots, v_k$ are the nodes in t corresponding to the roots of $s, t_1, \ldots, t_k$, respectively. Note that $\mathrm{rep}(s)$ can be stored using $O((k+1) \cdot \log(n))$ many bits. Given such a tuple $\mathrm{rep}(s) = (v_0, \ldots, v_k)$, we can compute in logspace the size $|s|$ by a preorder traversal of t, starting from v_0 and skipping subtrees rooted in the nodes $v_1, \ldots, v_k$. We can also compute in logspace the split node v of s: If s has at most two parameters, then $v = \mathrm{split}(s)$. Note that the procedure from Lemma 5 can be implemented in logspace since the size of a subtree of s can be computed as described before. If s has three parameters, then v is the lowest common ancestor of either v_1 and v_2, or of v_2 and v_3, depending on which node has the larger distance from v_0. The lowest common ancestor of two nodes can also be computed in logspace by traversing the paths from the two nodes to the

Algorithm 1: TreeBiSection(t, k)

input : binary tree t
$P_{\text{temp}} := \{S \to t\}$
$P_{\text{final}} := \emptyset$
while $P_{\text{temp}} \neq \emptyset$ **do**
  foreach $(A \to s) \in P_{\text{temp}}$ **do**
    $P_{\text{temp}} := P_{\text{temp}} \setminus \{A \to s\}$
    if $\text{rank}(s) = 3$ **then**
      $v :=$ the lower of nodes $\text{lca}_s(x_1, x_2), \text{lca}_s(x_2, x_3)$
    else
      $v := \text{split}(s)$
    end
    $t_1 := s[v]$; $t_2 := s \setminus v$
    $r_1 := \text{rank}(t_1)$; $r_2 := \text{rank}(t_2)$
    Let A_1 and A_2 be fresh nonterminals.
    foreach $i = 1$ **to** 2 **do**
      if $|t_i| > 1$ **then**
        $P_{\text{temp}} := P_{\text{temp}} \cup \{A_i(x_1, \ldots, x_{r_i}) \to t_i\}$
      else
        $P_{\text{final}} := P_{\text{final}} \cup \{A_i(x_1, \ldots, x_{r_i}) \to t_i\}$
      end
    end
    $r := r_1 + r_2 - 1$
    Let k be the number of nodes in s labelled by parameters that are smaller than v w.r.t. $<_s$.
    $P_{\text{final}} := P_{\text{final}} \cup \{A(x_1, \ldots, x_r) \to A_1(x_1, \ldots, x_k, A_2(x_{k+1}, \ldots, x_{k+r_2}), x_{k+r_2+1}, \ldots, x_r)\}$
  end
end
Let $\mathcal{G}$ be the TSLP (S, P_{final}).
Construct the modified derivation tree $\mathcal{D}_t^*$ of $\mathcal{G}$.
Compute the minimal dag of $\mathcal{D}_t^*$ and let $\mathcal{H}$ be the corresponding TSLP.
return TSLP $\mathcal{H}$

root upwards. From $\mathrm{rep}(s) = (v_0, \ldots, v_k)$ and a split node v we can easily determine $\mathrm{rep}(s[v])$ and $\mathrm{rep}(s \setminus v)$ in logspace.

Using the previous remarks we are ready to present the logspace algorithm to compute $\mathcal{D}_t^*$. Since $\mathcal{D}_t^*$ is a binary tree of depth $O(\log n)$ we can identify a node of $\mathcal{D}_t^*$ with the string $u \in \{0,1\}^*$ of length at most $c \cdot \lfloor \log n \rfloor$ that stores the path from the root to the node, where $c > 0$ is a suitable constant. We denote by s_u the tree (with at most three parameters) described by a node u of $\mathcal{D}_t^*$ in the sense of Lemma 7. That is, if u' is the corresponding node of the derivation tree $\mathcal{D}_t$ and u' is labelled with the nonterminal A, then $s_u = \mathrm{val}_{\mathcal{G}_t}(A)$.

To compute $\mathcal{D}_t^*$, it suffices for each string $w \in \{0,1\}^*$ of length $c \cdot \lfloor \log n \rfloor$ to check in logspace whether it is a node of $\mathcal{D}_t^*$ and in case it is a node, to determine the label of w in $\mathcal{D}_t^*$. For this, we compute for each prefix u of w, starting with the empty word, the tuple $\mathrm{rep}(s_u)$ and the label of u in $\mathcal{D}_t^*$, or a bit indicating that u is not a node of $\mathcal{D}_t^*$ (in which case also w is not a node of $\mathcal{D}_t^*$). Thereby we only store the current bit strings w, u and the value of $\mathrm{rep}(s_u)$, which fit into logspace. If $u = \varepsilon$, then $\mathrm{rep}(s_u)$ consists only of the root of t. Otherwise, we first compute in logspace the size $|s_u|$ from $\mathrm{rep}(s_u)$. If $|s_u| = 1$, then u is a leaf in $\mathcal{D}_t^*$ with label $\lambda(u)$ and no longer prefixes represent nodes in $\mathcal{D}_t^*$. If $|s_u| > 1$, then u is an inner node in $\mathcal{D}_t^*$ and, as described above, we can compute in logspace from $\mathrm{rep}(s_u)$ the tuples $\mathrm{rep}(s_{u0})$ and $\mathrm{rep}(s_{u1})$, from which we can easily read off the label of u from $\{1,2,3\}$. If $u = w$, then we stop, otherwise we continue with ui and $\mathrm{rep}(s_{ui})$, where $i \in \{0,1\}$ is such that ui is a prefix of w. □

5.3. Size of the minimal dag

In order to bound the size of the TSLP produced by TreeBiSection we have to bound the number of nodes in the dag of the modified derivation tree. To this end, we prove in this section a general result about the size of dags of certain weakly balanced binary trees, which might be of independent interest.

Let t be a binary tree and let $0 < \beta \leq 1$. The *leaf size* of a node v is the number of leaves of the subtree rooted at v. We say that an inner node v with children v_1 and v_2 is *β-balanced* if the following holds: If n_i is the leaf size of v_i, then $n_1 \geq \beta n_2$ and $n_2 \geq \beta n_1$. We say that t is *β-balanced* if the following holds: For all inner nodes u and v such that v is a child of u, we have that u is β-balanced or v is β-balanced.

Theorem 10. *If t is a β-balanced binary tree having σ different node labels and n leaves (and hence $|t|, \sigma \leq 2n-1$), then the size of the dag of t is bounded by $O\big(\frac{\alpha \cdot n}{\log_\sigma n}\big)$, where $\alpha = 1 + \log_{1+\beta}(\beta^{-1})$ only depends on β.*[4]

Proof. Let us fix a tree $t = (V, \lambda)$ as in the theorem with n leaves. Moreover, let us fix a number k that will be defined later. We first bound the number of different subtrees with at most k leaves in t. Afterwards we will estimate the size of the remaining *top tree*. The same strategy is used for instance in [21, 32] to derive a worst-case upper bound on the size of binary decision diagrams.

Claim 1. The number of different subtrees of t with at most k leaves is bounded by d^k with $d = 16\sigma^2$.

A subtree of t with at most k leaves has at most $2k-1$ nodes, each of which is labelled with one of σ many labels. Hence, by Lemma 1 we can bound the number of subtrees of t with at most k leaves by $\frac{4}{3}(4\sigma)^{2k-1} = \frac{1}{3\sigma}(4\sigma)^{2k} \leq (16\sigma^2)^k$.

Let $\mathrm{top}(t,k)$ be the tree obtained from t by removing all nodes with leaf size at most k. Recall that $\alpha = 1 + \log_{1+\beta}(\beta^{-1})$.

Claim 2. The number of nodes of $\mathrm{top}(t,k)$ is bounded by $4\alpha \cdot \frac{n}{k}$.

The tree $\mathrm{top}(t,k)$ has at most n/k leaves since it is obtained from t by removing all nodes with leaf size at most k. Each node in $\mathrm{top}(t,k)$ has at most two children. We show that the length of

[4] Since $0 < \beta \leq 1$, we have $\alpha \geq 1$.

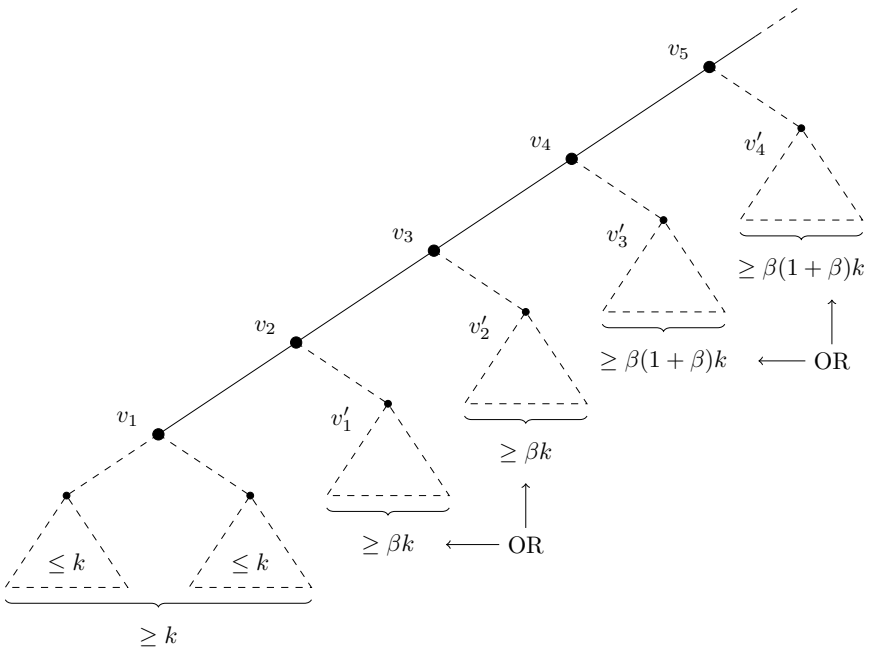

Figure 5: A chain within a top tree. The subtree rooted at v_1 has more than k leaves.

every *unary chain* in $\mathrm{top}(t,k)$ is bounded by 2α. This implies that $\mathrm{top}(t,k)$ has at most $4\alpha \cdot n/k$ many nodes.

Let $v_1, \ldots, v_m$ be a unary chain in $\mathrm{top}(t,k)$ where v_i is the single child node of v_{i+1}. Moreover, let v'_i be the removed sibling of v_i in t, see Figure 5. Note that each node v'_i has leaf size at most k.

We claim that the leaf size of v_{2i+1} is larger than $(1+\beta)^i k$ for all i with $2i+1 \leq m$. For $i=0$ note that v_1 has leaf size more than k since otherwise it would have been removed in $\mathrm{top}(t,k)$. For the induction step, assume that the leaf size of v_{2i-1} is larger than $(1+\beta)^{i-1}k$. One of the nodes v_{2i} and v_{2i+1} must be β-balanced. Hence, v'_{2i-1} or v'_{2i} must have leaf size at least $\beta(1+\beta)^{i-1}k$. Hence, v_{2i+1} has leaf size more than $(1+\beta)^{i-1}k + \beta(1+\beta)^{i-1}k = (1+\beta)^i k$.

If $m \geq 2\alpha+1$, then $v_{2\alpha-1}$ exists and has leaf size at least $(1+\beta)^{\alpha-1}k = k/\beta$, which implies that the leaf size of $v'_{2\alpha-1}$ or $v'_{2\alpha}$ (both nodes exist) is more than k, which is a contradiction. Hence, we must have $m \leq 2\alpha$. Figure 5 shows an illustration of our argument.

Using Claim 1 and 2 we can now prove the theorem: The number of nodes of the dag of t is bounded by the number of different subtrees with at most k leaves (Claim 1) plus the number of nodes of the remaining tree $\mathrm{top}(t,k)$ (Claim 2). Let $k = \frac{1}{2}\log_d n$. Recall that $d = 16\sigma^2$ and hence $\log d = 4 + 2\log\sigma$, which implies that $\log_d n \in \Theta(\log_\sigma n)$. With Claim 1 and 2 we get the following bound on the size of the dag:

$$d^k + 4\alpha \cdot \frac{n}{k} = d^{(\log_d n)/2} + \frac{8\alpha \cdot n}{\log_d n} = \sqrt{n} + \frac{8\alpha \cdot n}{\log_d n} \in O\left(\frac{\alpha \cdot n}{\log_d n}\right) = O\left(\frac{\alpha \cdot n}{\log_\sigma n}\right)$$

This proves the theorem. □

Obviously, one could relax the definition of β-balanced by only requiring that if $(v_1, v_2, \ldots, v_\delta)$ is a path down in the tree, where δ is a constant, then one of the nodes $v_1, v_2, \ldots, v_\delta$ must be β-balanced. Theorem 10 would still hold with this definition (with α depending linearly on δ).

Before we apply Theorem 10 to TreeBiSection let us present a few other results on the size of dags that are of independent interest. If β is a constant, then a β-balanced binary tree t has depth $O(\log|t|)$. One might think that this logarithmic depth is responsible for the small dag size in Theorem 10. But this intuition is wrong:

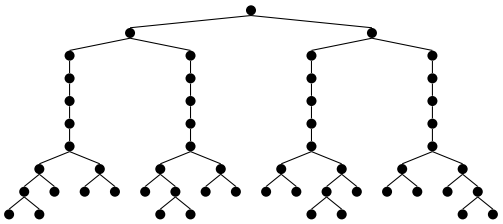

Figure 6: Tree t_{16} from the proof of Theorem 11.

Theorem 11. *There is a family of trees $t_n \in \mathcal{T}(\{a, b, c\})$ with $a \in \mathcal{F}_0$, $b \in \mathcal{F}_1$, and $c \in \mathcal{F}_2$ $(n \geq 1)$ with the following properties:*[5]

- $|t_n| \in \Theta(n)$
- *The depth of t_n is $\Theta(\log n)$.*
- *The size of the minimal dag of t_n is at least n.*

Proof. Let $k = \frac{n}{\log n}$ (we ignore rounding problems with $\log n$, which only affect multiplicative factors). Choose k different binary trees $s_1, \ldots, s_k \in \mathcal{T}(\{a, c\})$, each having $\log n$ many internal nodes. This is possible: By the asymptotic formula for the Catalan numbers (see e.g. [17, p. 38]) the number of different binary trees with $\log n$ many internal nodes is asymptotically equal to

$$\frac{4^{\log n}}{\sqrt{\pi \cdot \log^3 n}} = \frac{n^2}{\sqrt{\pi \cdot \log^3 n}} > n.$$

Then consider the trees $s'_i = b^{\log n}(s_i)$. Each of these trees has size $\Theta(\log n)$ and depth $\Theta(\log n)$. Next, let $u_n(x_1, \ldots, x_k) \in \mathcal{T}(\{c, x_1, \ldots, x_k\})$ be a balanced binary valid pattern (all non-parameter nodes are labelled with c) of depth $\Theta(\log k) = \Theta(\log n)$ and size $\Theta(k) = \Theta(\frac{n}{\log n})$. We finally take $t_n = u_n(s'_1, \ldots, s'_k)$. Figure 6 shows the tree t_{16}. We obtain $|t_n| = \Theta(\frac{n}{\log n}) + \Theta(k \cdot \log n) = \Theta(n)$. The depth of t_n is $\Theta(\log n)$. Finally, in the minimal dag for t_n the unary b-labelled nodes cannot be shared. Basically, the pairwise different trees $t_1, \ldots, t_n$ work as different constants that are attached to the b-chains. But the number of b-labelled nodes in t_n is $k \cdot \log n = n$. □

Note that the trees from Theorem 11 are not β-balanced for any constant $0 < \beta < 1$, and by Theorem 10 this is necessarily the case. Interestingly, if we assume that every subtree s of a binary tree t has depth at most $O(\log |s|)$, then Hübschle-Schneider and Raman [22] have implicitly shown the bound $O\big(\frac{n \cdot \log \log_\sigma n}{\log_\sigma n}\big)$ for the size of the minimal dag.

Theorem 12 ([22])**.** *Let α be a constant. Then there is a constant β that only depends on α such that the following holds: If t is a binary tree of size n with σ many node labels such that every subtree s of t has depth at most $\alpha \log n + \alpha$, then the size of the dag of t is at most $\frac{\beta \cdot n \cdot \log \log_\sigma n}{\log_\sigma n} + \beta$.*

Interestingly, we can show that the bound in this result is sharp:

Theorem 13. *There is a family of trees $t_n \in \mathcal{T}(\{a, b, c\})$ with $a \in \mathcal{F}_0$, $b \in \mathcal{F}_1$, and $c \in \mathcal{F}_2$ $(n \geq 1)$ with the following properties:*[6]

- $|t_n| \in \Theta(n)$
- *Every subtree s of a tree t_n has depth $O(\log |s|)$.*

[5] The unary node label b can be replaced by the pattern $c(d, x)$, where $d \in \mathcal{F}_0 \setminus \{a\}$, to obtain a binary tree.
[6] Again, the unary node label b can replaced by the pattern $c(d, x)$, where $d \in \mathcal{F}_0 \setminus \{a\}$ to obtain a binary tree.

- *The size of the minimal dag of t_n is $\Omega(\frac{n \cdot \log\log n}{\log n})$.*

Proof. The tree t_n is similar to the one from the proof of Theorem 11. Again, let $k = \frac{n}{\log n}$. Fix a balanced binary tree $v_n \in \mathcal{T}(\{a, c\})$ with $\log k \in \Theta(\log n)$ many leaves. From v_n we construct k many different trees $s_1, \ldots, s_k \in \mathcal{T}(\{a, b, c\})$ by choosing in v_n an arbitrary subset of leaves (there are k such subsets) and replacing all leaves in that subset by $b(a)$. Note that $|s_i| \in \Theta(\log n)$. Moreover, every subtree s of a tree s_i has depth $O(\log |s|)$ (since v_n is balanced). Then consider the trees $s'_i = b^{\log\log n}(s_i)$ (so, in contrast to the proof of Theorem 11, the length of the unary chains is $\log\log n$). Clearly, $|s'_i| \in \Theta(\log n)$. Moreover, we still have the property that every subtree s of a tree s'_i has depth $O(\log |s|)$: This is clear, if that subtree is rooted in a node from s_i. Otherwise, the subtree s has the form $b^h(s_i)$ for some $h \leq \log\log n$. This tree has depth $h + \Theta(\log\log n) = \Theta(\log\log n)$ and size $\Theta(\log n)$. Finally we combine the trees $s'_1, \ldots, s'_k \in \mathcal{T}(\{a, b, c\})$ in a balanced way to a single tree using the binary valid pattern $u_n(x_1, \ldots, x_k)$ from the proof of Theorem 11, i.e., $t_n = u_n(s'_1, \ldots, s'_k)$. Then, $|t_n| = \Theta(n)$. Moreover, by the same argument as in the proof of Theorem 11, the dag for t_n has size $\Omega(\frac{n \cdot \log\log n}{\log n})$ since the nodes in the $k = \frac{n}{\log n}$ many unary chains of length $\log\log n$ cannot be shared with other nodes. It remains to show that every subtree s of t_n has depth $O(\log |s|)$. For the case that s is a subtree of one of the trees s'_i, this has been already shown above. But the case that s is rooted in a node from $u_n(x_1, \ldots, x_k)$ is also clear: In that case, s is of the form $s = u'(s'_i, \ldots, s'_j)$, where $u'(x_i, \ldots, x_j)$ is a subtree of $u_n(x_1, \ldots, x_k)$. Assume that d is the depth of $u'(x_i, \ldots, x_j)$. Since $u_n(x_1, \ldots, x_k)$ is a balanced binary tree, we have $|u'(x_i, \ldots, x_j)| \in \Omega(2^d)$ and $j - i + 1 \in \Omega(2^d)$. Hence, $s = u'(s'_i, \ldots, s'_j)$ has size $\Omega(2^d + 2^d \cdot \log n) = \Omega(2^d \cdot \log n)$ and depth $d + \Theta(\log\log n)$. This shows the desired property since $\log(2^d \cdot \log n) = d + \log\log n$. □

Hübschle-Schneider and Raman [22] used Theorem 12 to prove the upper bound $O\big(\frac{n \cdot \log\log_\sigma n}{\log_\sigma n}\big)$ for the size of the top dag of an unranked tree of size n with σ many node labels. It is not clear whether this bound can be improved to $O\big(\frac{n}{\log_\sigma n}\big)$. The trees used in Theorem 13 do not seem to arise as the top trees of unranked trees.

5.4. Size of the TSLP produced by TreeBiSection

Let us fix the TSLP $\mathcal{G}_t$ for a tree $t \in \mathcal{T}(\mathcal{F}_{\leq 2})$ that has been produced by the first part of TreeBiSection. Let $n = |t|$ and σ be the number of the different node labels that appear in t. For the modified derivation tree $\mathcal{D}^*_t$ we have the following:

- $\mathcal{D}^*_t$ is a strictly binary tree with n leaves and hence has $2n - 1$ nodes.
- There are $\sigma + 3$ possible node labels, namely $1, 2, 3$ and those appearing in t.
- $\mathcal{D}^*_t$ is $(1/7)$-balanced by (3). If we have two successive nodes in $\mathcal{D}^*_t$, then we split at one of the two nodes according to (3). Now, assume that we split at node v according to (3). Let v_1 and v_2 be the children of v, let n_i be the leaf size of v_i, and let $n = n_1 + n_2$ be the leaf size of v. We get $\frac{1}{8}n \leq n_1 \leq \frac{3}{4}n$ and $\frac{1}{4}n \leq n_2 \leq \frac{7}{8}n$ (or vice versa). Hence, $n_1 \geq \frac{1}{8}n \geq \frac{1}{7}n_2$ and $n_2 \geq \frac{1}{4}n \geq \frac{1}{3}n_1$.

Recall that the nodes of the dag of $\mathcal{D}^*_t$ are the nonterminals of the TSLP produced by TreeBiSection and that this TSLP is in Chomsky normal form. Moreover, recall that the depth of $\mathcal{D}^*_t$ is in $O(\log n)$. Hence, with Lemma 9 we get:

Corollary 14. *For a tree from $\mathcal{T}(\mathcal{F}_{\leq 2})$ of size n with σ different node labels,* TreeBiSection *produces a TSLP in Chomsky normal form of size $O\big(\frac{n}{\log_\sigma n}\big)$ and depth $O(\log n)$. Every nonterminal of that TSLP has rank at most 3, and the algorithm can be implemented in logspace and, alternatively, in time $O(n \cdot \log n)$.*

In particular, for an unlabelled tree of size n we obtain a TSLP of size $O\big(\frac{n}{\log n}\big)$.

5.5. Extension to trees of larger degree

If the input tree t has nodes with many children, then we cannot expect good compression by TSLPs. The extreme case is $t_n = f_n(a, \dots, a)$ where f_n is a symbol of rank n. Hence, $|t_n| = n + 1$ and every TSLP for t_n has size $\Omega(n)$. On the other hand, for trees in which the maximal rank is bounded by a constant $r \geq 1$, we can easily generalize TreeBiSection. Lemma 5 allows to find a splitting node v satisfying

$$\frac{1}{2(r+2)} \cdot |t| \;\leq\; |t[v]| \;\leq\; \frac{r+1}{r+2} \cdot |t|. \tag{4}$$

The maximal arity of nodes also affects the arity of patterns: we allow patterns of rank up to r. Now assume that $t(x_1, \dots, x_{r+1})$ is a valid pattern of rank $r + 1$, where r is again the maximal number of children of a node. Then we find a subtree containing k parameters, where $2 \leq k \leq r$: Take a smallest subtree that contains at least two parameters. Since the root node of that subtree has at most r children, and every proper subtree contains at most one parameter (due to the minimality of the subtree), this subtree contains at most r parameters. By taking the root of that subtree as the splitting node, we obtain two valid patterns with at most r parameters each. Hence, we have to change TreeBiSection in the following way:

- As long as the number of parameters of the tree is at most r, we choose the splitting node according to Lemma 5.
- If the number of parameters is $r + 1$ (note that in each splitting step, the number of parameters increases by at most 1), then we choose the splitting node such that the two resulting fragments have rank at most r.

As before, this guarantees that in every second splitting step we split in a balanced way. But the balance factor β from Section 5.3 now depends on r. More precisely, if in the modified derivation tree $\mathcal{D}_t^*$ we have a node v with children v_1 and v_2 of leaf size n_1 and n_2, respectively, and this node corresponds to a splitting satisfying (4), then we get

$$\begin{aligned}
\frac{1}{2(r+2)} \cdot n \;&\leq\; n_1 \;\leq\; \frac{r+1}{r+2} \cdot n, \\
\frac{1}{r+2} \cdot n \;=\; \left(1 - \frac{r+1}{r+2}\right) \cdot n \;&\leq\; n_2 \;\leq\; \left(1 - \frac{1}{2(r+2)}\right) \cdot n \;=\; \frac{2r+3}{2(r+2)} \cdot n
\end{aligned}$$

or vice versa. This implies

$$\begin{aligned}
n_1 \;&\geq\; \frac{1}{2(r+2)} \cdot n \;\geq\; \frac{1}{2r+3} \cdot n_2, \\
n_2 \;&\geq\; \frac{1}{r+2} \cdot n \;\geq\; \frac{1}{r+1} \cdot n_1 \;\geq\; \frac{1}{2r+3} \cdot n_1.
\end{aligned}$$

Hence, the modified derivation tree becomes β-balanced for $\beta = 1/(2r + 3)$. Moreover, the label alphabet of the modified derivation tree now has size $\sigma + r + 1$ (since the TSLP produced in the first step has nonterminals of rank at most $r + 1$). Theorem 10 yields the following bound on the dag of the modified derivation tree and hence the size of the final TSLP:

$$O\left(\frac{n}{\log_{\sigma+r}(n)} \cdot \log_{1+\frac{1}{2r+3}}(2r+3)\right) = O\left(\frac{n}{\log_{\sigma+r}(n)} \cdot \frac{\log(2r+3)}{\log(1+\frac{1}{2r+3})}\right)$$

Note that $\log(1 + x) \geq x$ for $0 \leq x \leq 1$. Hence, we can simplify the bound to

$$O\left(\frac{n \cdot \log(\sigma + r) \cdot r \cdot \log r}{\log n}\right).$$

By Lemma 5, the depth of the produced TSLP can be bounded by $2 \cdot d$, where d is any number that satisfies

$$n \cdot \left(\frac{r+1}{r+2}\right)^d \leq 1.$$

Hence, we can bound the depth by

$$2 \cdot \left\lceil \frac{\log n}{\log(1 + \frac{1}{r+1})} \right\rceil \leq 2 \cdot \lceil (r+1) \cdot \log n \rceil \in O(r \cdot \log n).$$

Theorem 15. *For a tree of size n with σ different node labels, each of rank at most r,* TreeBiSection *produces a TSLP in Chomsky normal form of size $O\big(\frac{n \cdot \log(\sigma+r) \cdot r \cdot \log r}{\log n}\big)$ and depth $O(r \cdot \log n)$. Every nonterminal of that TSLP has rank at most $r+1$.*

For the running time we obtain the following bound:

Theorem 16. TreeBiSection *can be implemented such that it works in time $O(r \cdot n \cdot \log n)$ for a tree of size n in which each symbol has rank at most r.*

Proof. TreeBiSection makes $O(r \cdot \log n)$ iterations of the while loop (this is the same bound as for the depth of the TSLP) and each iteration takes time $O(n)$. To see the latter, our internal representation of trees with parameters from Section 4 is important. Using this representation, we can still compute the split node in a right-hand side s from P_{temp} in time $O(|s|)$: We first compute for every non-parameter node v of s (i) the size of the subtree rooted at v (as usual, excluding parameters) and (ii) the number of parameters below v. This is possible in time $O(|s|)$ using a straightforward bottom-up computation. Using this information, we can compute the split node in s in time $O(|s|)$ for both cases (number of parameters in s is $r+1$ or smaller than $r+1$) by searching from the root downwards. □

In particular, if r is bounded by a constant, TreeBiSection computes a TSLP of size $O\big(\frac{n}{\log_\sigma n}\big)$ and depth $O(\log n)$ in time $O(n \cdot \log n)$. Moreover, our logspace implementation of TreeBiSection (see Lemma 9) directly generalizes to the case of a constant rank.

On the other hand, for unranked trees in which the number of children of a node is arbitrary and not determined by the node label (which is the standard tree model in XML) all this fails: TreeBiSection only yields TSLPs of size $\Theta(n)$ and this is unavoidable as shown by the example from the beginning of this subsection. Moreover, the logspace implementation from Section 5.2 no longer works since nonterminals have rank at most $r+1$ and we cannot store anymore the pattern derived from a nonterminal in space $O(\log n)$ (we have to store $r+1$ many nodes in the tree).

Fortunately there is a simple workaround for all these problems: An unranked tree can be transformed into a binary tree of the same size using the well known first-child next-sibling encoding [8, 30]. Then, one can simply apply TreeBiSection to this encoding to get in logspace and time $O(n \cdot \log n)$ a TSLP of size $O\big(\frac{n}{\log_\sigma n}\big)$.

For the problem of traversing a compressed unranked tree t (which is addressed in [6] for top dags) another (equally well known) encoding is more favorable. Let $c(t)$ be a compressed representation (e.g., a TSLP or a top dag) of t. The goal is to represent t in space $O(|c(t)|)$ such that one can efficiently navigate from a node to (i) its parent node, (ii) its first child, (iii) its next sibling, and (iv) its previous sibling (if they exist). For top dags [6], it was shown that a single navigation step can be done in time $O(\log |t|)$. Using a suitable binary encoding, we can prove the same result for TSLPs: Let r be the maximal rank of a node of the unranked tree t. We define the binary encoding $\text{bin}(t)$ by adding for every node v of rank $s \leq r$ a binary tree of depth $\lceil \log s \rceil$ with s many leaves, whose root is v and whose leaves are the children of v. This introduces at most $2s$ many new binary nodes, which are labelled by a new symbol. We get $|\text{bin}(t)| \leq 3|t|$. In particular, we obtain a TSLP of size $O\big(\frac{n}{\log_\sigma n}\big)$ for $\text{bin}(t)$, where $n = |t|$ and σ is the number of different node labels. Note that a traversal step in the initial tree t (going to the parent node, first child, next sibling, or previous sibling) can be simulated by $O(\log r)$ many traversal steps in $\text{bin}(t)$ (going to the parent node, left child, or right child). But for a binary tree s, it was recently shown that a TSLP $\mathcal{G}$ for s can be represented in space $O(|\mathcal{G}|)$ such that a single traversal step takes time $O(1)$ [35].[7] Hence, we can navigate in t in time $O(\log r) \leq O(\log |t|)$.

[7] This generalizes a corresponding result for strings [20].

5.6. Linear Time TSLP Construction

We showed that TreeBiSection can be implemented such that it works time $O(n \cdot \log n)$ for a tree of size n. In this section we present a linear time algorithm BU-Shrink (for bottom-up shrink) that also constructs a TSLP of size $O\big(\frac{n}{\log_\sigma n}\big)$ for a given tree of size n with σ many node labels of constant rank. The basic idea of BU-Shrink is to merge in a bottom-up way nodes of the tree to *patterns* of size roughly k, where k is defined later. This is a bottom-up computation in the sense that we begin with individual nodes and gradually merge them into larger fragments (the term "bottom-up" should not be understood in the sense that the computation is done from the leaves of the tree towards the root). The dag of the small trees represented by the patterns then yields the compression.

For a valid pattern p of rank r we define the weight of p as $|p| + r$. This is the total number of nodes in p including those nodes that are labelled with a parameter (which are not counted in the size $|p|$ of p). A *pattern tree* is a tree in which the labels of the tree are valid patterns. If a node v is labelled with the valid pattern p and $\mathrm{rank}(p) = d$, then we require that v has d children in the pattern tree. For convenience, BU-Shrink also stores in every node the weight of the corresponding pattern. For a node v, we denote by p_v its pattern and by $w(v)$ the weight of p_v.

Let us fix a number $k \geq 1$ that will be specified later. Given a tree $t = (V, \lambda)$ of size n such that all node labels in t are of rank at most r, BU-Shrink first creates a pattern tree t' by replacing every label $f \in \mathcal{F}$ by the valid pattern $f(x_1, \ldots, x_d)$ of weight $d+1$, where d is the rank of f. Note that the parameters in these patterns correspond to the edges of the tree t. We will keep this invariant during the algorithm, which will shrink the pattern tree t'. Hence, the total number of all parameter occurrences in the patterns that appear as labels in the current pattern tree t' will be always the number of nodes of the current tree t' minus 1. This allows us to ignore the cost of handling parameters for the running time of the algorithm.

After generating the initial pattern tree t', BU-Shrink creates a queue Q that contains references to all nodes of t' having at most one child (excluding the root node) in an arbitrary ordering. During the run of the algorithm, the queue Q will only contain references to non-root nodes of the current tree t' that have at most one child (but Q may not contain references to all such nodes). For each node v of the queue we proceed as follows. Let v be the i^{th} child of its parent node u. If $w(v) > k$ or $w(u) > k$, we simply remove v from Q and proceed. Otherwise we merge the node v into the node u. More precisely, we delete the node v, and set the i^{th} child of u to the unique child of v if it exists (otherwise, u loses its i^{th} child). The pattern p_u is modified by replacing the parameter at the position of the i^{th} child by the pattern p_v and re-enumerating all parameters to get a valid pattern. We also set the weight $w(u)$ to $w(v) + w(u) - 1$ (which is the weight of the new pattern p_u). Note that in this way both the number of edges of t' and the total number of parameter occurrences in all patterns decreases by 1 and so these two sizes stay equal. For example, let u be a node with $p_u = f(x_1, x_2)$ and let v be its second child with $p_v = g(x_1)$. Then the merged pattern becomes $f(x_1, g(x_2))$, and its weight is 4. If the node u has at most one child after the merging and its weight is at most k, then we add u to Q (if it is not already in the queue). We do this until the queue is empty. Note that every pattern appearing in the final pattern tree has rank at most r (the maximal rank in the initial tree).

Now consider the forest consisting of all patterns appearing in the resulting final pattern tree. We construct the dag of this forest, which yields grammar rules for each pattern with shared nonterminals. The dag of a forest (i.e., a disjoint union of trees) is constructed in the same way as for a single tree. This dag has for every subtree appearing in the forest exactly one node. The parameters $x_1, x_2, \ldots, x_r$ that appear in the patterns are treated as ordinary constants when constructing the dag. As usual, the dag can be viewed as a TSLP, where the nodes of the dag are the nonterminals. In this way we obtain a TSLP in which each pattern is derived by a nonterminal of the same rank. Finally, we add to the TSLP the start rule $S \to s$, where s is obtained from the pattern tree by labelling each node v with the unique nonterminal A such that A derives $p(v)$. Algorithm 2 shows the pseudocode for BU-Shrink.

Example 17. *Consider the pattern tree depicted in Figure 7. Assuming no further mergings are done, the final TSLP is* $S \to A(B(C), B(B(C)))$, $A(x_1, x_2) \to f(g(x_1), x_2)$, $B(x_1) \to$

Algorithm 2: BU-Shrink(t, k)

input : tree $t = (V, \lambda)$, number $k \leq |t|$
$Q := \emptyset$
foreach $v \in V$ **do**
  let $f = \lambda(v) \in \mathcal{F}_d$ be the label of node v
  $w(v) := 1 + d$ `(the weight of node v)`
  $p_v := f(x_1, \ldots, x_d)$ `(the pattern stored in node v)`
  if $d \leq 1$ *and* v *is not the root* **then**
    $Q := Q \cup \{v\}$
  end
end
while $Q \neq \emptyset$ **do**
  choose arbitrary node $v \in Q$ and set $Q := Q \setminus \{v\}$
  let u be the parent node of v
  if $w(v) \leq k$ **and** $w(u) \leq k$ **then**
    $d := \mathrm{rank}(p_v)$; $e := \mathrm{rank}(p_u)$
    let v be the i^{th} child of u
    $w(u) := w(u) + w(v) - 1$
    $p_u := p_u(x_1, \ldots, x_{i-1}, p_v(x_i, \ldots, x_{i+d-1}), x_{i+d}, \ldots, x_{d+e-1})$
    if v *has a (necessarily unique) child* v' **then**
      set v' to the i^{th} child of u
    end
    delete node v
    if $d + e - 1 \leq 1$ *and* $w(u) \leq k$ **then**
      $Q := Q \cup \{u\}$
    end
  end
end
$P := \emptyset$
compute the minimal dag for the forest consisting of all patterns p_v, where v is a node of t
foreach *node* v *of* t **do**
  create a fresh nonterminal A_v of rank $d := \mathrm{rank}(p_v)$
  $P := P \cup \{A_v(x_1, \ldots, x_d) \to p_v(x_1, \ldots, x_d)\}$
  $\lambda(v) := A_v$ `(the new label of node v)`
end
return TSLP $(S, P \cup \{S \to t\})$

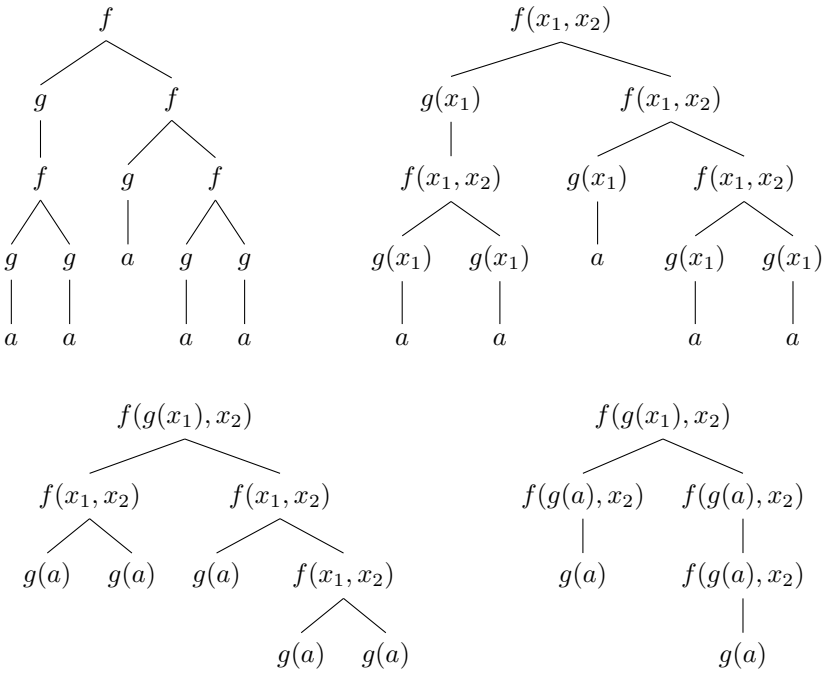

Figure 7: BU-Shrink first transforms the input tree on the top left into the pattern tree on the top right (the weights are omitted to improve readability). Then it starts to shrink this pattern tree. The two trees at the bottom depict possible intermediate trees during the shrinking process.

$f(C, x_1)$, $C \to g(a)$.

It is easy to see that BU-Shrink runs in time $O(n)$ for a tree of size n. First of all, the number of mergings is bounded by n, since each merging reduces the number of nodes of the pattern tree by one. Moreover, if a node is removed from Q (because its weight or the weight of its parent node is larger than k) then it will never be added to Q again (since weights are never reduced). A single merging step needs only a constant number of pointer operations and a single addition (for the weights). For this, it is important that we do not copy patterns, when the new pattern (for the node u in the above description) is constructed. The forest, for which we construct the dag, has size $O(n)$: The number of non-parameter nodes is exactly n, and the number of parameters is at most $n-1$: initially the forest has $n-1$ parameters (as there is a parameter for each node except the root) and during BU-Shrink we can only decrease the total amount of parameters.

Let us now analyze the size of the constructed TSLP. In the following, let t be the input tree of size n and let r be the maximal rank of a label in t. Let $\Sigma = \text{labels}(t)$ and $\sigma = |\Sigma|$.

Lemma 18. *Let t_p be the pattern tree resulting from* BU-Shrink. *Then $|t_p| \leq \frac{4 \cdot r \cdot n}{k} + 2$.*

Proof. Let us first assume that $r \geq 2$. The number of non-root nodes in t_p of arity at most one is at least $|t_p|/2 - 1$. For each of those nodes, either the node itself or the parent node has weight at least k. We now map in t_p each non-root node of arity at most one to a node of weight at least k: Let u be a node of t_p (which is not the root) having arity at most one. If the weight of u is at least k, we map u to itself, otherwise we map u to its parent node, which then must be of weight at least k. Note that at most r nodes are mapped to a fixed node v: If v has arity at most one, then v and its child (if it exists) can be mapped to v; if v has arity greater than one then only its children can be mapped to v, and v has at most r children. Therefore there must exist at least $\frac{|t_p|-2}{2r}$ many nodes of weight at least k. Because the sum of all weights in t_p is at most $2n$, this yields

$$\frac{|t_p| - 2}{2r} \cdot k \leq 2n,$$

which proves the lemma for the case $r \geq 2$. The case $r = 1$ can be proved in the same way: Clearly, the number of non-root nodes of arity at most one is $|t_p| - 1$ and at most 2 nodes are mapped to a fixed node of weight at least k (by the above argument). Hence, there exist at least $\frac{|t_p|-1}{2} = \frac{|t_p|-1}{2r}$ many nodes of weight at least k. □

Note that each node in the final pattern tree has weight at most $2k$ since BU-Shrink only merges nodes of weight at most k. By Lemma 1 the number of different patterns in $\mathcal{T}(\Sigma \cup \{x_1, \ldots, x_r\})$ of weight at most $2k$ is bounded by $\frac{4}{3}(4(\sigma + r))^{2k} \leq d^k$ for $d = (6(\sigma + r))^2$. Hence, the size of the dag constructed from the patterns is bounded by d^k. Adding the size of the start rule, i.e. the size of the resulting pattern tree (Lemma 18) we get the following bound for the constructed TSLP:

$$d^k + \frac{4 \cdot r \cdot n}{k} + 2.$$

Let us now set $k = \frac{1}{2} \cdot \log_d n$. We get the following bound for the constructed TSLP:

$$\begin{aligned} d^{\frac{1}{2} \cdot \log_d n} + \frac{8 \cdot n \cdot r}{\log_d n} + 2 = \sqrt{n} + O\left(\frac{n \cdot r}{\log_d n}\right) = O\left(\frac{n \cdot r}{\log_{\sigma+r} n}\right) \\ = O\left(\frac{n \cdot \log(\sigma + r) \cdot r}{\log n}\right). \end{aligned}$$

Theorem 19. BU-Shrink *computes for a given tree of size n with σ many node labels of rank at most r in time $O(n)$ a TSLP of size $O\big(\frac{n \cdot \log(\sigma+r) \cdot r}{\log n}\big)$. Every nonterminal of that TSLP has rank at most r.*

Clearly, if r is bounded by a constant, we obtain the bound $O\big(\frac{n}{\log_\sigma n}\big)$. On the other hand, already for $r \in \Omega(\log n)$ the bound $O\big(\frac{n \cdot r}{\log_{r+\sigma} n}\big)$ is in $\omega(n)$. But note that the size of the TSLP produced by BU-Shrink can never exceed n.

Combining TreeBiSection and BU-Shrink to achieve logarithmic grammar depth. Recall that TreeBiSection produces TSLPs in Chomsky normal form of logarithmic depth, which will be important in the next section. Clearly, the TSLPs produced by BU-Shrink are not in Chomsky normal form. To get a TSLP in Chomsky normal form we have to further partition the right-hand sides of the TSLP. Let us assume in this section that the maximal rank of symbols appearing in the input tree is bounded by a constant. Hence, BU-Shrink produces for an input tree t of size n with σ many node labels a TSLP of size $O\big(\frac{n}{\log_\sigma n}\big)$. The weight and hence also the depth of the patterns that appear in t_p is $O(\log_d n) = O(\log_\sigma n) \leq O(\log n)$. The productions that arise from the dag of the forest of all patterns have the form $A \to f(A_1, \ldots, A_r)$ and $A \to x$ where f is a node label of the input tree, r is bounded by a constant, and x is one of the parameters (recall that in the dag construction, we consider the parameters appearing in the patterns as ordinary constants). Productions $A \to x$ are eliminated by replacing all occurrences of A in a right-hand side by the parameter x. All resulting productions (except the start rule $S \to s$) have the form $A \to f(\alpha_1, \ldots, \alpha_r)$ where r is a constant and every α_i is either a nonterminal or a parameter. These productions are then split such that all resulting productions (except the start rule $S \to s$) are in Chomsky normal form. This is straightforward. For instance the production $A \to f(A_1, A_2, A_3)$ is split into $A \to B(A_3)$, $B(x) \to C(A_2, x)$, $C(x, y) \to D(A_1, x, y)$ and $D(x, y, z) \to f(x, y, z)$. Recall that we assume that the maximal rank of terminal symbols is bounded by a constant. Therefore, the above splitting increases the size and depth only by a constant.

Recall that for the start rule $S \to s$, where s is the tree returned by BU-Shrink, the tree s has size $O\big(\frac{n}{\log_\sigma n}\big) = O\big(\frac{n \cdot \log \sigma}{\log n}\big)$. We want to apply TreeBiSection to balance the tree s. But we cannot use it directly because the resulting running time would not be linear if σ is not a constant: Since TreeBiSection needs time $O(|s| \log |s|)$ on trees of constant rank (see the paragraph after the proof

of Theorem 16), this yields the time bound

$$O(|s| \log |s|) = O\left(\frac{n \cdot \log \sigma}{\log n} \cdot \log\left(\frac{n \cdot \log \sigma}{\log n}\right)\right)$$
$$= O\left(\frac{n \cdot \log \sigma}{\log n} \cdot (\log n + \log\log \sigma - \log\log n)\right) = O(n \log \sigma).$$

To eliminate the factor $\log \sigma$, we apply BU-Shrink again to s with $k = \log \sigma \leq \log n$. Note that the maximal rank in s is still bounded by a constant (the same constant as for the input tree). By Lemma 18 this yields in time $O(|s|) \leq O(n)$ a tree s' of size $O\big(\frac{n}{\log n}\big)$ on which we may now use TreeBiSection to get a TSLP for s' in Chomsky normal form of size $O(|s'|) = O\big(\frac{n}{\log n}\big)$ (note that every node of s' may be labelled with a different symbol, in which case TreeBiSection cannot achieve any compression for s', when we count the size in bits) and depth $O(\log |s'|) = O(\log n)$. Moreover, the running time of TreeBiSection on s' is

$$O(|s'| \cdot \log |s'|) = O\left(\frac{n}{\log n} \log\left(\frac{n}{\log n}\right)\right)$$
$$= O\left(\frac{n}{\log n} \cdot (\log n - \log\log n)\right) = O(n).$$

Let us call this combined algorithm BU-Shrink+TreeBiSection.

Theorem 20. BU-Shrink+TreeBiSection *computes for a given tree t of size n with σ many node labels of constant rank each in time $O(n)$ a TSLP in Chomsky normal form of size $O\big(\frac{n}{\log_\sigma n}\big)$ and depth $O(\log n)$. The rank of every nonterminal of that TSLP is bounded by the maximal rank of a node label in t (a constant).*

6. Arithmetical Circuits

In this section, we present our main application of Corollary 14 and Theorem 20. Let $\mathcal{S} = (S, +, \cdot)$ be a (not necessarily commutative) semiring. Thus, $(S, +)$ is a commutative monoid with identity element 0, $(S, \cdot)$ is a monoid with identity element 1, and $\cdot$ left and right distributes over $+$.

We use the standard notation of arithmetical formulas and circuits over $\mathcal{S}$: An *arithmetical formula* is just a labelled binary tree in which internal nodes are labelled with the semiring operations $+$ and $\cdot$, and leaf nodes are labelled with variables $y_1, y_2, \ldots$ or the constants 0 and 1. An *arithmetical circuit* is a (not necessarily minimal) dag whose internal nodes are labelled with $+$ and $\cdot$ and whose leaf nodes are labelled with variables or the constants 0 and 1. The *depth* of a circuit is the length of a longest path from the root node to a leaf. An arithmetical circuit evaluates to a multivariate noncommutative polynomial $p(y_1, \ldots, y_n)$ over $\mathcal{S}$, where $y_1, \ldots, y_n$ are the variables occurring at the leaf nodes. Two arithmetical circuits are equivalent if they evaluate to the same polynomial.

Brent [9] has shown that every arithmetical formula of size n over a commutative ring can be transformed into an equivalent circuit of depth $O(\log n)$ and size $O(n)$ (the proof easily generalizes to semirings). By first constructing a TSLP of size $O\big(\frac{n \cdot \log m}{\log n}\big)$, where m is the number of different variables in the formula, and then transforming this TSLP into a circuit, we will refine the size bound to $O\big(\frac{n \cdot \log m}{\log n}\big)$. Moreover, by Corollary 14 (Theorem 20, respectively) this conversion can be done in logspace (linear time, respectively).

In the following, we consider TSLPs, whose terminal alphabet consists of the binary symbols $+$ and $\cdot$ and the constant symbols $0, 1, y_1, \ldots, y_m$ for some m. Let us denote this terminal alphabet with Σ_m. For our formula-to-circuit conversion, it will be important to work with monadic TSLPs, i.e., TSLPs in which every nonterminal has rank at most one.

Lemma 21. *From a given tree $t \in \mathcal{T}(\Sigma_m)$ of size n one can construct in logspace (linear time, respectively) a monadic TSLP $\mathcal{H}$ of size $O\big(\frac{n \cdot \log m}{\log n}\big)$ and depth $O(\log n)$ with $\mathrm{val}(\mathcal{H}) = t$ and such that all productions are of the following forms:*

- $A \to B(C)$ *for* $A, C \in \mathcal{N}_0$, $B \in \mathcal{N}_1$,
- $A(x) \to B(C(x))$ *for* $A, B, C \in \mathcal{N}_1$,
- $A \to f(B, C)$ *for* $f \in \{+, \cdot\}$, $A, B, C \in \mathcal{N}_0$,
- $A(x) \to f(x, B)$, $A(x) \to f(B, x)$ *for* $f \in \{+, \cdot\}$, $A \in \mathcal{N}_1$, $B \in \mathcal{N}_0$,
- $A \to a$ *for* $a \in \{0, 1, y_1, \ldots, y_m\}$, $A \in \mathcal{N}_0$,
- $A(x) \to B(x)$ *for* $A, B \in \mathcal{N}_1$,
- $A(x) \to x$ *for* $A \in \mathcal{N}_1$.

Proof. The linear time version is an immediate consequence of Theorem 3 and Theorem 20.[8] It remains to show the logspace version. We first apply TreeBiSection (Corollary 14) to get in logspace a TSLP $\mathcal{G}$ in Chomsky normal form of size $O\big(\frac{n \cdot \log m}{\log n}\big)$ and depth $O(\log n)$ with $\mathrm{val}(\mathcal{G}) = t$. Note that every nonterminal of $\mathcal{G}$ has rank 3. Moreover, for every nonterminal A of rank $k \leq 3$, TreeBiSection computes $k+1$ nodes $v_0, v_1, \ldots, v_k$ of t that represent the pattern $\mathrm{val}_{\mathcal{G}}(A)$: v_0 is the root node of an occurrence of $\mathrm{val}_{\mathcal{G}}(A)$ in t and v_i $(1 \leq i \leq k)$ is the node of the occurrence to which the parameter x_i is mapped, see also the proof of Lemma 9. We can assume that for every nonterminal A of rank k this tuple s_A has been computed.

We basically show that the construction from [36], which makes a TSLP monadic, works in logspace if all nonterminals and terminals of the input TSLP have constant rank.[9] For a nonterminal A of rank 3 with $s_A = (v_0, v_1, v_2, v_3)$, the pattern $\mathrm{val}_{\mathcal{G}}(A)$ has two possible branching structures, which are the branching structures shown in Figure 2. By computing the paths from the three nodes v_1, v_2, v_3 up to v_0, we can compute in logspace, which of the two branching structures $\mathrm{val}_{\mathcal{G}}(A)$ has. Moreover, we can compute the two binary symbols $f_1, f_2 \in \{+, \cdot\}$ at which the three paths that go from v_1, v_2, and v_3, respectively, up to v_0 meet. We finally associate with each of the five dashed edges in Figure 2 a fresh unary nonterminal A_i $(0 \leq i \leq 4)$ of the TSLP $\mathcal{H}$. In this way we can built up in logspace what is called the skeleton tree for A. It is one of the following two trees, depending on the branching structure of $\mathrm{val}_{\mathcal{G}}(A)$, see also Figure 8:

1. $A_0(f_1(A_1(f_2(A_2(x_1), A_3(x_1))), A_4(x_3)))$
2. $A_0(f_1(A_1(x_1), A_2(f_2(A_3(x_2), A_4(x_3)))))$

For a nonterminal A of rank two there is only a single branching structure and hence a single skeleton tree $A_0(f(A_1(x_1), A_2(x_2)))$ for $f \in \{+, \cdot\}$. Finally, for a nonterminal A of rank at most one, the skeleton tree is A itself (this is in particular the case for the start nonterminal S, which will be also the start nonterminal of $\mathcal{H}$), and this nonterminal then belongs to $\mathcal{H}$ (nonterminals of $\mathcal{G}$ that have rank larger than one do not belong to $\mathcal{H}$). What remains is to construct in logspace productions for the nonterminals of $\mathcal{H}$ that allow to rewrite the skeleton tree of A to $\mathrm{val}_{\mathcal{G}}(A)$. For this, let us consider the productions of $\mathcal{G}$, whose right-hand sides have the form (1) and (2). A production $A(x_1, \ldots, x_k) \to f(x_1, \ldots, x_k)$ with $k \leq 1$ is copied to $\mathcal{H}$. On the other hand, if $k = 2$, then A does not belong to $\mathcal{H}$ and hence, we do not copy the production to $\mathcal{H}$. Instead, we introduce the productions $A_i(x_1) \to x_1$ $(0 \leq i \leq 2)$ for the three nonterminals A_0, A_1, A_2 that appear in the skeleton tree of A. Now consider a production

$$A(x_1, \ldots, x_k) \to B(x_1, \ldots, x_{i-1}, C(x_i, \ldots, x_{i+l-1}), x_{i+l}, \ldots, x_k),$$

[8] Note that productions of the form $A(x) \to B(x)$ and $A(x) \to x$ do not appear in Theorem 3. We allow them in the lemma, since they make the logspace part of the lemma easier to show and do not pose a problem in the remaining part of this section.

[9] We only consider the case that nonterminals have rank at most three and terminals have rank zero or two, which is the case we need, but the general case, where all nonterminals and all terminals of the input TSLP have constant rank could be handled in a similar way in logspace.

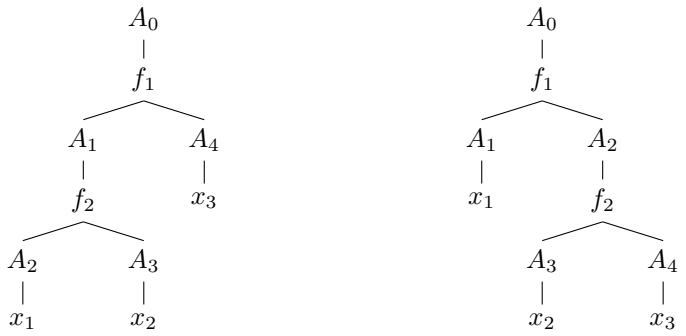

Figure 8: The two possible skeleton trees for a nonterminal A of rank three

where $k, l, k-l+1 \leq 3$ (note that l is the rank of C and $k-l+1$ is the rank of B). We have constructed the skeleton trees t_A, t_B, t_C for A, B, and C, respectively. Consider the tree $t_B(x_1, \ldots, x_{i-1}, t_C(x_i, \ldots, x_{i+l-1}), x_{i+l}, \ldots, x_k)$. We now introduce the productions for the nonterminals that appear in t_A in such a way that $t_A(x_1, \ldots, x_k)$ can be rewritten to the tree

$$t_B(x_1, \ldots, x_{i-1}, t_C(x_i, \ldots, x_{i+l-1}), x_{i+l}, \ldots, x_k).$$

There are several cases depending on k, l, and i. Let us only consider two typical cases (all other cases can be dealt in a similar way): The trees $t_A(x_1, x_2, x_3)$ and $t_B(x_1, t_C(x_2, x_3))$ for a production $A(x_1, x_2, x_3) \to B(x_1, C(x_2, x_3))$ are shown in Figure 9. Note that the skeleton tree $t_A(x_1, x_2, x_3)$ is the right tree from Figure 8. We add the following productions to $\mathcal{H}$:

$$A_0(x) \to B_0(x) \qquad A_1(x) \to B_1(x) \qquad A_2(x) \to B_2(C_0(x))$$
$$A_3(x) \to C_1(x) \qquad A_4(x) \to C_2(x)$$

Let us also consider the case $A(x_1, x_2) \to B(x_1, x_2, C)$. The trees $t_A(x_1, x_2)$ and $t_B(x_1, x_2, t_C) = t_B(x_1, x_2, C)$ are shown in Figure 10 (we assume that the skeleton tree for B is the left one from Figure 8). We add the following productions to $\mathcal{H}$:

$$A_0(x) \to B_0(f_1(B_1(x), B_4(C))), \quad A_1(x) \to B_2(x), \quad A_2(x) \to B_3(x). \tag{5}$$

Other cases can be dealt with similarly. In each case we write out a constant number of productions that clearly can be produced by a logspace machine using the shape of the skeleton trees. Correctness of the construction (i.e., $\mathrm{val}(\mathcal{G}) = \mathrm{val}(\mathcal{H})$) follows from $\mathrm{val}_{\mathcal{H}}(t_A) = \mathrm{val}_{\mathcal{G}}(A)$, which can be shown by a straightforward induction, see [36]. Clearly, the size and depth of $\mathcal{H}$ is linearly related to the size and depth, respectively, of $\mathcal{G}$. Finally, productions of the form $A(x) \to B(f(C(x), D(E)))$ (or similar forms) as in (5) can be easily split in logspace into productions of the forms shown in the lemma. For instance, $A(x) \to B(f(C(x), D(E)))$ is split into $A(x) \to B(F(x))$, $F(x) \to G(C(x))$, $G(x) \to f(x, H)$, $H \to D(E)$. Again, the size and depth of the TSLP increases only by a linear factor. □

Going from a monadic TSLP to a circuit (or dag) that evaluates over every semiring to the same noncommutative polynomial is easy:

Lemma 22. *From a given monadic TSLP $\mathcal{G}$ over the terminal alphabet Σ_m such that all productions are of the form shown in Lemma 21, one can construct in logspace (linear time, respectively) an arithmetical circuit C of depth $O(\mathrm{depth}(\mathcal{G}))$ and size $O(|\mathcal{G}|)$ such that over every semiring, C and $\mathrm{val}(\mathcal{G})$ evaluate to the same noncommutative polynomial in m variables.*

Proof. Fix an arbitrary semiring $\mathcal{S}$ and let $\mathcal{R}$ be the polynomial semiring $\mathcal{R} = \mathcal{S}[y_1, \ldots, y_m]$. Clearly, for a nonterminal A of rank 0, $\mathrm{val}_{\mathcal{G}}(A)$ is a tree without parameters that evaluates to an element p_A of the semiring $\mathcal{R}$. For a nonterminal $A(x)$ of rank 1, $\mathrm{val}_{\mathcal{G}}(A)$ is a tree, in which the

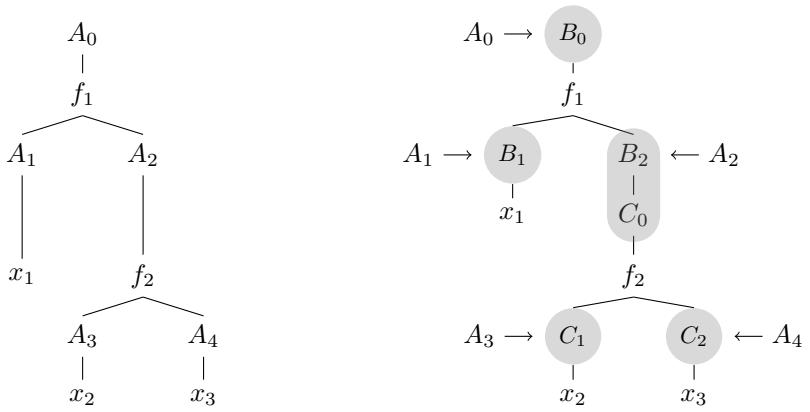

Figure 9: The skeleton tree $t_A(x_1, x_2, x_3)$ and the tree $t_B(x_1, t_C(x_2, x_3))$

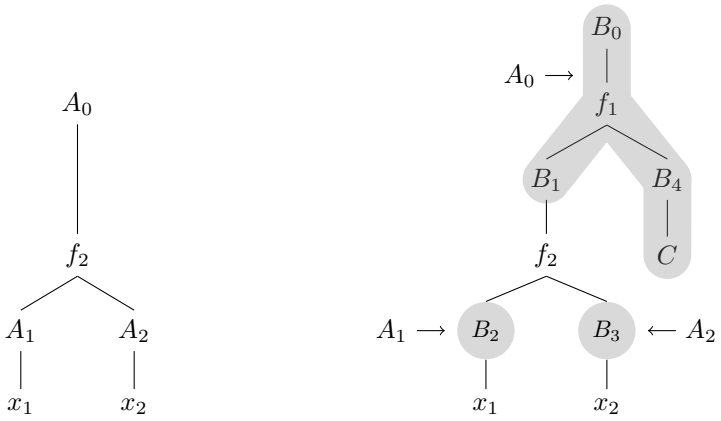

Figure 10: The skeleton tree $t_A(x_1, x_2)$ and the tree $t_B(x_1, x_2, t_C)$

only parameter x occurs exactly once. Such a tree evaluates to a noncommutative polynomial $p_A(x) \in \mathcal{R}[x]$. Since the parameter x occurs exactly once in the tree $\mathrm{val}(A)$, it turns out that $p_A(x)$ is linear and contains exactly one occurrence of x. More precisely, by induction on the structure of the TSLP $\mathcal{G}$ we show that for every nonterminal $A(x)$ of rank 1, the tree $\mathrm{val}_{\mathcal{G}}(A)$ evaluates in $\mathcal{R}[x]$ to a noncommutative polynomial of the form

$$p_A(x) = A_0 + A_1 x A_2,$$

where $A_0, A_1, A_2 \in \mathcal{R} = \mathcal{S}[y_1, \ldots, y_m]$. Using the same induction, one can build up a circuit of size $O(|\mathcal{G}|)$ and depth $O(\mathrm{depth}(\mathcal{G}))$ that contains gates evaluating to A_0, A_1, A_2. For a nonterminal A of rank zero, the circuit contains a gate that evaluates to the semiring element $p_A \in \mathcal{R}$, and we denote this gate with A as well.

The induction uses a straightforward case distinction on the rule for $A(x)$. The cases that the unique rule for A has the form $A(x) \to x$, $A(x) \to B(x)$, $A(x) \to f(x, B)$, $A(x) \to f(B, x)$, $A \to f(B, C)$, or $A \to a$ is clear. For instance, for a rule $A(x) \to +(B, x)$, we have $p_A(x) = B{+}1{\cdot}x{\cdot}1$, i.e., we set $A_0 := B$, $A_1 := 1$, $A_2 := 1$. Now consider a rule $A(x) \to B(C(x))$ (for $A \to B(C)$ the argument is similar). We have already built up a circuit containing gates that evaluate to $B_0, B_1, B_2, C_0, C_1, C_2$, where

$$p_B(x) = B_0 + B_1 x_1 B_2, \qquad p_C(x) = C_0 + C_1 x C_2.$$

We get

$$\begin{aligned} p_A(x) &= p_B(p_C(x)) \\ &= B_0 + B_1(C_0 + C_1 x C_2)B_2 \\ &= (B_0 + B_1 C_0 B_2) + B_1 C_1 x_1 C_2 B_2 \end{aligned}$$

and therefore set

$$A_0 := B_0 + B_1 C_0 B_2,\ A_1 := B_1 C_1,\ A_2 := C_2 B_2.$$

So we can define the polynomials A_0, A_1, A_2 using the gates B_0, B_1, B_2, C_0, C_1, C_2 with only 5 additional gates. Note that also the depth only increases by a constant factor (in fact, 2).

The output gate of the circuit is the start nonterminal of the TSLP $\mathcal{G}$. The above construction can be carried out in linear time as well as in logspace. □

Now we can show the main result of this section:

Theorem 23. *A given arithmetical formula F of size n having m different variables can be transformed in logspace (linear time, respectively) into an arithmetical circuit C of depth $O(\log n)$ and size $O\big(\frac{n \cdot \log m}{\log n}\big)$ such that over every semiring, C and F evaluate to the same noncommutative polynomial in m variables.*

Proof. Let F be an arithmetical formula of size n and let $y_1, \ldots, y_m$ be the variables occurring in F. Fix an arbitrary semiring $\mathcal{S}$ and let $\mathcal{R}$ be the polynomial semiring $\mathcal{R} = \mathcal{S}[y_1, \ldots, y_m]$. Using Lemma 21 we can construct in logspace (linear time, respectively) a monadic TSLP $\mathcal{G}$ of size $O\big(\frac{n \cdot \log m}{\log n}\big)$ and depth $O(\log n)$ such that $\mathrm{val}(\mathcal{G}) = F$. Finally, we apply Lemma 22 in order to transform $\mathcal{G}$ in logspace (linear time, respectively) into an equivalent circuit of size $O\big(\frac{n \cdot \log m}{\log n}\big)$ and depth $O(\log n)$. □

Theorem 23 can also be shown for fields instead of semirings. In this case, the expression is built up using variables, the constants -1, 0, 1, and the field operations $+, \cdot$ and $/$. The proof is similar to the semiring case. Again, we start with a monadic TSLP of size $O\big(\frac{n \cdot \log m}{\log n}\big)$ and depth $O(\log n)$ for the arithmetical expression. Again, one can assume that all rules have the form $A(x) \to B(C(x))$, $A \to B(C)$, $A(x) \to f(x, B)$, $A(x) \to f(B, x)$, $A \to f(B, C)$, $A(x) \to B(x)$, $A(x) \to x$, or $A \to a$, where f is one of the binary field operations and a is either -1, 0, 1, or a variable. Using this particular rule format, one can show that every nonterminal $A(x)$ evaluates to a rational function $(A_0 + A_1 x)/(A_2 + A_3 x)$ for polynomials A_0, A_1, A_2, A_3 in the circuit variables, whereas a nonterminal of rank 0 evaluates to a fraction of two polynomials. Finally, these polynomials can be computed by a single circuit of size $O\big(\frac{n \cdot \log m}{\log n}\big)$ and depth $O(\log n)$.

Lemma 22 has an interesting application to the problem of checking whether the polynomial represented by a TSLP over a ring is the zero polynomial. The question, whether the polynomial computed by a given circuit is the zero polynomial is known as *polynomial identity testing* (PIT). This is a famous problem in algebraic complexity theory. For the case that the underlying ring is $\mathbb{Z}$ or $\mathbb{Z}_n$ ($n \geq 2$) polynomial identity testing belongs to the complexity class **coRP** (the complement of randomized polynomial time), see [1]. PIT can be generalized to arithmetic expressions that are given by a TSLP. Using Lemma 22 and Theorem 3 we obtain:

Theorem 24. *Over any semiring, the question, whether the polynomial computed by a given TSLP is the zero polynomial, is equivalent with respect to polynomial time reductions to PIT. In particular, if the underlying semiring is $\mathbb{Z}$ or $\mathbb{Z}_n$, then the question, whether the polynomial computed by a given TSLP is the zero polynomial, belongs to* **coRP**.

7. Future work

In [45] a universal (in the information-theoretic sense) code for binary trees is developed. This code is computed in two phases: In a first step, the minimal dag for the input tree is constructed.

Then, a particular binary encoding is applied to the dag. It is shown that the *average redundancy* of the resulting code converges to zero (see [45] for definitions) for every probability distribution on binary trees that satisfies the so-called domination property and the representation ratio negligibility property. Whereas the domination property is somewhat technical and easily established for many distributions, the representation ratio negligibility property means that the average size of the dag divided by the tree size converges to zero for the underlying probability distribution. This is, for instance, the case for the uniform distribution, since the average size of the dag is $\Theta\big(\frac{n}{\sqrt{\log n}}\big)$ [18].

We construct TSLPs that have *worst-case* size $O\big(\frac{n}{\log n}\big)$ assuming a constant number of node labels. We are confident that replacing the minimal dag by a TSLP of worst-case size $O\big(\frac{n}{\log n}\big)$ in the universal encoder from [45] leads to stronger results. In particular, we hope that for the resulting encoder the *maximal pointwise redundancy* converges to zero for certain probability distributions. For strings, such a result was obtained in [28] using the fact that every string of length n over a constant size alphabet has an SLP of size $O\big(\frac{n}{\log n}\big)$.

It would be interesting to know the worst-case output size of the grammar-based tree compressor from [27]. It works in linear time and produces a TSLP of size $O(rg \log(n/rg))$ for a tree of size n and maximal rank r, where g is the size of a smallest TSLP. Since this function is monotonically increasing with g and $g \in O(n/\log_\sigma n)$ for trees of constant rank, the algorithm in [27] yields for a tree of constant rank a TSLP of size $O(n \log\log_\sigma n/\log_\sigma n)$. It remains open, whether the additional factor $\log\log_\sigma n$ is necessary.

In [6] the authors proved that the top dag of a given tree t of size n is at most by a factor $\log n$ larger than the minimal dag of t. It is not clear, whether the TSLP constructed by TreeBiSection has this property too. The construction of the top dag is done in a bottom-up way, and as a consequence identical subtrees are compressed in the same way. This property is crucial for the comparison with the minimal dag. TreeBiSection works in a top-down way. Hence, it is not guaranteed that identical subtrees are compressed in the same way. In contrast, BU-Shrink works bottom-up (although in a somehow different way as for the dag), and one might compare the size of the produced TSLP with the size of the minimal dag.

Acknowledgment. We thank Anna Gál and the anonymous referees for helpful comments. The third, fourth and fifth author acknowledge support from the DFG research project QUANT-KOMP (Lo 748/10-1). The third author was additionally supported by Humboldt Foundation return fellowship.

[1] M. Agrawal and R. Saptharishi. Classifying polynomials and identity testing. *Current Trends in Science — Platinum Jubilee Special*, pages 149–162, 2009

[2] T. Akutsu. A bisection algorithm for grammar-based compression of ordered trees. *Inf. Process. Lett.*, 110(18–19):815–820, 2010.

[3] I. Althöfer. Tight lower bounds on the length of word chains. *Inf. Process. Lett.*, 34(5): 275–276, 1990.

[4] S. Arora and B. Barak. *Computational Complexity — A Modern Approach*. Cambridge University Press, 2009.

[5] J. Berstel and S. Brlek. On the length of word chains. *Inf. Process. Lett.*, 26(1): 23–28, 1987.

[6] P. Bille, I. L. Gørtz, G. M. Landau, and O. Weimann. Tree compression with top trees. *Inf. Comput.*, 243: 166–177, 2015.

[7] M. L. Bonet and S. R. Buss. Size-depth tradeoffs for Boolean fomulae. *Inf. Process. Lett.*, 49(3):151–155, 1994.

[8] M. Bousquet-Mélou, M. Lohrey, S. Maneth, and E. Noeth. XML compression via DAGs. *Theor. Comput. Syst.*, 57(4):1322–1371, 2015.

[9] R. P. Brent. The parallel evaluation of general arithmetical expressions. *J. ACM*, 21(2):201–206, 1974.

[10] N. H. Bshouty, R. Cleve, and W. Eberly. Size-depth tradeoffs for algebraic formulas. *SIAM J. Comput.*, 24(4):682–705, 1995.

[11] G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML document trees. *Inf. Syst.*, 33(4–5):456–474, 2008.

[12] M. Charikar, E. Lehman, A. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005.

[13] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://tata.gforge.inria.fr/`, 2007.

[14] A. A Diwan. A new combinatorial complexity measure for languages. Tata Institute, Bombay, India, 1986.

[15] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980.

[16] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), 2009.

[17] P. Flajolet and R. Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

[18] P. Flajolet, P. Sipala, and J.-M. Steyaert. Analytic variations on the common subexpression problem. In *Proc. ICALP 1990*, LNCS 443, pages 220–234. Springer, 1990.

[19] T. Gagie and P. Gawrychowski. Grammar-based compression in a streaming model. In *Proc. LATA 2010*, LNCS 6031, pages 273–284. Springer, 2010.

[20] L. Gasieniec, R. M. Kolpakov, I. Potapov, and P. Sant. Real-time traversal in grammar-based compressed files. In *Proc. DCC 2005*, page 458. IEEE Computer Society, 2005.

[21] M. A. Heap and M. R. Mercer. Least upper bounds on OBDD sizes. *IEEE Trans. Computers*, 43(6):764–767, 1994.

[22] L. Hübschle-Schneider and Rajeev Raman. Tree compression with top trees revisited. In *Proc. SEA 2015*, LNCS 9125, pages 15–27. Springer, 2015. Long version at arXiv.org, 2015., `http://arxiv.org/abs/1506.04499`.

[23] D. Hucke, M. Lohrey, and E. Noeth. Constructing small tree grammars and small circuits for formulas. In *Proc. FSTTCS 2014*, vol. 29 of *LIPIcs*, pages 457–468. Leibniz-Zentrum für Informatik, 2014.

[24] G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS 1989*, pages 549–554. IEEE Computer Society, 1989.

[25] A. Jeż. Approximation of grammar-based compression via recompression. *Theor. Comput. Sci.*, 592:115–134, 2015.

[26] A. Jeż. A really simple approximation of smallest grammar. *Theor. Comput. Sci.*, 616:141–150, 2016.

[27] A. Jeż and M. Lohrey. Approximation of smallest linear tree grammars. In *Proc. STACS 2014*, vol. 25 of *LIPIcs*, pages 445–457. Leibniz-Zentrum für Informatik, 2014.

[28] J. C. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory*, 46(3):737–754, 2000.

[29] J. C. Kieffer, E.-H. Yang, G. J. Nelson, and P. C. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Trans. Inf. Theory*, 46(4):1227–1245, 2000.

[30] D. E. Knuth. *The Art of Computer Programming, Vol. I: Fundamental Algorithms*. Addison-Wesley, 1968.

[31] P. M. Lewis II, R. E. Stearns, and J. Hartmanis. Memory bounds for recognition of context-free and context-sensitive languages. In *Proc. 6th Annual IEEE Symp. Switching Circuit Theory and Logic Design*, pages 191–202, 1965.

[32] H.-T. Liaw and C.-S. Lin. On the OBDD-representation of general Boolean functions. *IEEE Trans. Computers*, 41(6):661–664, 1992.

[33] M. Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.

[34] M. Lohrey, S. Maneth, and R. Mennicke. XML tree structure compression using RePair. *Inf. Syst.*, 38(8):1150–1167, 2013.

[35] M. Lohrey, S. Maneth, and C. P. Reh. Traversing grammar-compressed trees with constant delay. In *Proc. DCC 2016*, IEEE Computer Society, 2016.

[36] M. Lohrey, S. Maneth, and M. Schmidt-Schauß. Parameter reduction and automata evaluation for grammar-compressed trees. *J. Comput. Syst. Sci.*, 78(5):1651–1669, 2012.

[37] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

[38] R. Raman and S. S. Rao. Succinct Representations of Ordinal Trees. In *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, LNCS 8066, pages 319–332. Springer, 2013.

[39] P. Roth. A note on word chains and regular languages. *Inf. Process. Lett.*, 30(1): 15–18, 1989.

[40] W. L. Ruzzo. Tree-size bounded alternation. *J. Comput. Syst. Sci.*, 21:218–235, 1980.

[41] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1–3):211–222, 2003.

[42] H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discrete Algorithms*, 3(2-4):416–430, 2005.

[43] P. M. Spira. On time-hardware complexity tradeoffs for Boolean functions. In *Proc. 4th Hawaii Symp. on System Sciences*, pages 525–527, 1971.

[44] R. P. Stanley. *Catalan Numbers*. Cambridge University Press, 2015.

[45] J. Zhang, E.-H. Yang, and J. C. Kieffer. A universal grammar-based code for lossless compression of binary trees. *IEEE Trans. Inf. Theory*, 60(3):1373–1386, 2014.