Balancing Straight-Line Programs*

MOSES GANARDI, Max Planck Institute for Software Systems (MPI-SWS), Germany
ARTUR JEŻ, University of Wrocław, Poland
MARKUS LOHREY, University of Siegen, Germany

We show that a context-free grammar of size m that produces a single string w of length n (such a grammar is also called a string straight-line program) can be transformed in linear time into a context-free grammar for w of size $O(m)$, whose unique derivation tree has depth $O(\log n)$. This solves an open problem in the area of grammar-based compression, improves many results in this area and greatly simplifies many existing constructions. Similar results are shown for two formalisms for grammar-based tree compression: top dags and forest straight-line programs. These balancing results can be all deduced from a single meta-theorem stating that the depth of an algebraic circuit over an algebra with a certain finite base property can be reduced to $O(\log n)$ with the cost of a constant multiplicative size increase. Here, n refers to the size of the unfolding (or unravelling) of the circuit. In particular, this results applies to standard arithmetic circuits over (noncommutative) semirings.

ACM Reference Format:
Moses Ganardi, Artur Jeż, and Markus Lohrey. 2021. Balancing Straight-Line Programs. 1, 1 (March 2021), 39 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Grammar-based string compression. In *grammar-based compression* a combinatorial object is compactly represented using a grammar of an appropriate type. Such a grammar can be up to exponentially smaller than the object itself. A well-studied example of this general idea is grammar-based string compression using context-free grammars that produce only one string, which are also known as *straight-line programs*. Since the term "straight-line programs" is used in the literature for different kinds of objects (e.g. arithmetic straight-line programs) and we will also deal with different types of straight-line programs, we use the term *string straight-line program*, SSLP for short. Grammar-based string compression is tightly related to dictionary-based compression: the famous LZ78 algorithm can be viewed as a particular grammar-based compressor, the number of phrases in the LZ77-factorization is a lower bound for the smallest SSLP for a string [36], and an LZ77-factorization of length m can be converted to an SSLP of size $O(m \cdot \log n)$ where n is the length of the string [11, 24, 26, 36]. For various other aspects of grammar-based string compression see [11, 29].

*This article is an extended version of the conference paper [18]

Artur Jeż was supported under National Science Centre, Poland project number 2017/26/E/ST6/00191.
Markus Lohrey has been supported by the DFG research project LO 748/10-1.
Authors' addresses: Moses Ganardi, ganardi@mpi-sws.org, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße G 26, 67663, Kaiserslautern, Germany; Artur Jeż, aje@cs.uni.wroc.pl, University of Wrocław, Poland; Markus Lohrey, lohrey@eti.uni-siegen.de, University of Siegen, Germany.

XXXX-XXXX/2021/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Balancing string straight-line programs. The two important measures for an SSLP are size and depth. To define these measures, it is convenient to assume that all right-hand sides of the grammar have length two (as in Chomsky normal form). Then, the size $|\mathcal{G}|$ of an SSLP $\mathcal{G}$ is the number of variables (nonterminals) of $\mathcal{G}$ and the depth of $\mathcal{G}$ ($\mathrm{depth}(\mathcal{G})$ for short) is the depth of the unique derivation tree of $\mathcal{G}$. It is straightforward to show that any string s of length n can be produced by an SSLP of size $O(n)$ and depth $O(\log n)$. A more difficult problem is to balance a given SSLP: Assume that the SSLP $\mathcal{G}$ produces a string of length n. Several authors have shown that one can restructure $\mathcal{G}$ in time $O(|\mathcal{G}| \cdot \log n)$ into an equivalent SSLP $\mathcal{H}$ of size $O(|\mathcal{G}| \cdot \log n)$ and depth $O(\log n)$ [11, 26, 36].

Finding SSLPs of small size and small depth is important in many algorithmic applications. A prominent example is the *random access problem for grammar-compressed strings*: For a given SSLP $\mathcal{G}$ that produces the string s of length n and a given position $p \in [1, n]$ one wants to access the p-th symbol in s. As observed in [8] one can solve this problem in time $O(\mathrm{depth}(\mathcal{G}))$ (assuming arithmetic operations on numbers from the interval $[0, n]$ use constant time). Combined with one of the known SSLP balancing procedures [11, 36] one obtains access time $O(\log n)$, but one has to pay with an increased SSLP size of $O(|\mathcal{G}| \cdot \log n)$. Using sophisticated data structures, the following result was shown in [8]:

Theorem 1.1 (random access to grammar-compressed strings, cf. [8])**.** *From a given SSLP $\mathcal{G}$ of size m that generates the string s of length n, one can construct in time $O(m)$ a data structure of size $O(m)$ (measured in words of bit length $\log n$) that allows to answer random access queries in time $O(\log n)$.*

Our main result for string straight-line programs states that SSLP balancing is in fact possible with a constant blow-up in size.

Theorem 1.2. *Given an SSLP $\mathcal{G}$ producing a string of length n one can construct in linear time an equivalent SSLP $\mathcal{H}$ of size $O(|\mathcal{G}|)$ and depth $O(\log n)$.*

As a corollary we obtain a very simple and clean proof of Theorem 1.1. We can also obtain an algorithm for the random access problem with running time $O(\log n / \log\log n)$ using $O(m \cdot \log^{\epsilon} n)$ words of bit length $\log n$; previously this bound was only shown for balanced SSLPs [2]. Section 2.4 contains a list of further applications of Theorem 1.2, which include the following problems on SSLP-compressed strings: rank and select queries [2], subsequence matching [3], computing Karp-Rabin fingerprints [6], computing runs, squares, and palindromes [23], real-time traversal [19, 33] and range-minimum queries [20]. In all these applications we either improve existing results or significantly simplify existing proofs by replacing $\mathrm{depth}(\mathcal{G})$ by $O(\log n)$ in time/space bounds.

Let us say a few words over the underlying computational model in Theorem 1.2. Our balancing procedure involves (simple) arithmetic on lengths, i.e., numbers of order n. Thus the linear running time can be achieved assuming that machine words have $\Omega(\log n)$ bits. Otherwise the running time increases by a multiplicative $\log n$ factor. Note that such an assumption is realistic and standard in the field since machine words of bit length $\Omega(\log n)$ are needed, say, for indexing positions in the represented string. Furthermore, our procedure works in the pointer model regime.

Balancing forest straight-line programs and top dags. Grammar-based compression has been generalized from strings to ordered ranked node-labelled trees. In fact, the representation of a tree t by its smallest directed acyclic graph (DAG) is a form of grammar-based tree compression. This DAG is obtained by merging nodes where the same subtree of t is rooted. It can be seen as a regular tree grammar that produces only t. A drawback of DAG-compression is that the size of the DAG is lower-bounded by the height of the tree t. Hence, for deep narrow trees (like for instance caterpillar trees), the DAG-representation cannot achieve good compression. This can be overcome by representing a tree t by a linear context-free tree grammar that produces only t. Such grammars are also known as *tree straight-line programs* in the case of ranked trees [10, 31, 32] and *forest straight-line programs* in the case of unranked trees [16]. The latter are tightly related to *top dags* [4, 7, 13, 16, 22], which are another tree compression formalism, also akin to grammars. Our balancing technique works similarly for those types of compression:

Theorem 1.3. *Given a top dag / forest straight-line program / tree straight-line program $\mathcal{G}$ producing the tree t one can compute in time $O(|\mathcal{G}|)$ a top dag / forest straight-line program / tree straight-line program $\mathcal{H}$ for t of size $O(|\mathcal{G}|)$ and depth $O(\log |t|)$.*

For top dags, this solves an open problem from [7], where it was proved that from a tree t of size n, whose minimal DAG has size m (measured in number of edges in the DAG), one can construct in linear time a top dag for t of size $O(m \cdot \log n)$ and depth $O(\log n)$. It remained open whether one can get rid of the factor $\log n$ in the size bound. For the specific top dag constructed in [7], it was shown in [4] that the factor $\log n$ in the size bound $O(m \cdot \log n)$ cannot be avoided. On the other hand, our results yield another top dag of size $O(m)$ and depth $O(\log n)$. To see this note that one can easily convert the minimal DAG of t into a top dag of roughly the same size, which can then be balanced. This also gives an alternative proof of a result from [13], according to which one can construct in linear time a top dag of size $O(n/\log_\sigma n)$ and depth $O(\log n)$ for a given tree of size n containing σ many different node labels.

Balancing circuits over algebras. Our balancing results for SSLPs, top dags, forests straight-line programs and tree straight-line programs are all instances of a general balancing result that applies to a large class of circuits over algebraic structures. To see the connection between circuits and straight-line programs, consider SSLPs as an example. An SSLP is the same thing as a bounded fan-in circuit over a free monoid. The circuit gates compute the concatenation of their inputs and correspond to the variables of the SSLP. In general, for any algebra one can define straight-line programs, which coincide with the classic notion of circuits.

The definition of a class of algebras, to which our general balancing technique applies, uses *unary linear term functions*, which were also used for instance in the context of efficient parallel evaluation of expression trees [34]. Fix an algebra $\mathcal{A}$ (a set together with finitely many operations of possibly different arities). For some of our applications we have to allow multi-sorted algebras that have several carrier sets (think for instance of a vector space, where the two carrier sets are an abelian group and a field of scalars). A unary linear term function is a unary function on $\mathcal{A}$ that is computed by a term (or algebraic expression) that contains a single variable x (which stands for the function argument) and, moreover, x occurs exactly once in the term. For instance, a unary linear term function over a commutative ring is of the form $x \mapsto ax + b$ for ring elements a, b. A *subsumption base* for an algebra $\mathcal{A}$ is, roughly speaking, a finite set $C(\mathcal{A})$ of unary linear term functions that are described by terms with parameters such that every unary linear term function can be obtained from one of the terms in $C(\mathcal{A})$ by instantiating the parameters. In the above example for a commutative ring the set $C(\mathcal{A})$ consists of the single term $ax + b$, where a and b are the parameters.

Our general balancing result needs one more concept, namely the *unfolded size* of a circuit $\mathcal{G}$. It can be conveniently defined as follows: we replace in $\mathcal{G}$ every input gate by the number 1, and we replace every internal gate by an addition gate. The unfolded size of $\mathcal{G}$ is the value of this additive circuit. In other words, this is the size of the tree obtained by unravelling $\mathcal{G}$ into a tree. Note that the size of this unfolding can be exponential in the circuit size. Now we can state the general balancing result in a slightly informal way (the precise statement can be found in Theorem 3.20):

Theorem 1.4 (informal statement)**.** *Let $\mathcal{A}$ be a multi-sorted algebra with a finite number of operations (of arbitrary arity) such that $\mathcal{A}$ has a finite subsumption base. Given a circuit $\mathcal{G}$ over $\mathcal{A}$ whose unfolded size is n, one can compute in time $O(|\mathcal{G}|)$ a circuit $\mathcal{H}$ evaluating to the same element of $\mathcal{A}$ such that $|\mathcal{H}| \in O(|\mathcal{G}|)$ and depth($\mathcal{H}$) $\in O(\log n)$.*

Theorems 1.2 and 1.3 are immediate corollaries of Theorem 1.4. Theorem 1.4 can be also applied to not necessarily commutative semirings, as every semiring has a finite subsumption base. Hence, for every semiring circuit one can reduce with a linear size blow-up the depth to $O(\log n)$, where n is the size of the circuit unfolding.

Note that in the depth bound $O(\log n)$ in our balancing result for string straight-line programs (Theorem 1.2), n refers to the length of the produced string. A string straight-line program can be viewed as a circuit for a non-commutative semiring circuit that produces a single monomial (the symbols in the string correspond to the non-commuting variables). If one considers arbitrary circuits over non-commutative semirings (that produce a sum of more than one monomial), depth reduction is not possible in general by a result of Kosaraju [27]. For circuits over commutative semirings depth reduction is possible by a seminal result of Valiant, Skyum, Berkowitz and Rackoff [38]: for any commutative semiring, every circuit of size m and formal degree d can be transformed into an equivalent circuit of depth $O(\log m \log d)$ and size polynomial in m and d. This result led to many further investigations on depth reduction for bounded degree circuits over various classes of commutative as well as non-commutative semirings [1]. If one drops the restriction to bounded degree circuits, then depth reduction gets even harder. For general Boolean circuits, the best known result states that every Boolean circuit of size m is equivalent to a Boolean circuit of depth $O(m/\log m)$ [35].

Proof strategy. The proof of Theorem 1.2 consists of two main steps (the general result Theorem 1.4 is shown similarly). Take an SSLP $\mathcal{G}$ for the string s of length n and let m be the size of $\mathcal{G}$. We consider the derivation tree t for $\mathcal{G}$; it has size $O(n)$. The SSLP $\mathcal{G}$ can be viewed as a DAG for t of size m. We decompose this DAG into node-disjoint paths such that each path from the root to a leaf intersects $O(\log n)$ paths from the decomposition (Section 2.1). Each path from the decomposition is then viewed as a string of integer-weighted symbols, where the weights are the lengths of the strings derived from nodes that branch off from the path. For this weighted string we construct an SSLP of linear size that produces all suffixes of the path in a weight-balanced way (Section 2.2). Plugging these SSLPs together yields the final balanced SSLP.

Some of the concepts of our construction can be traced back to the area of parallel algorithms: the path decomposition for DAGs from Section 2.1 is related to the centroid decomposition of trees [12], where it is the key technique in several parallel algorithms on trees. Moreover, the SSLP of linear size that produces all suffixes of a weighted string with (Section 2.2) can be seen as a weight-balanced version of the optimal prefix sum algorithm.

For the general result Theorem 1.4 we need another ingredient: when the above construction is used for circuits over algebras, the corresponding procedure produces a tree straight-line program for the unfolding of the circuit. We show that if the underlying algebra $\mathcal{A}$ has a finite subsumption base, then one can compute from a tree straight-line program an equivalent circuit over $\mathcal{A}$. Moreover, the size and depth of this circuit are linearly bounded in the size and depth of the tree straight-line program. This construction was used before for the special cases of semirings and regular expressions [15, 17].

2 PART I: BALANCING OF STRING STRAIGHT-LINE PROGRAMS

The goal of the first part of the paper is to prove Theorem 1.2. This result can be also derived from our general balancing theorem (Theorem 1.4), which will be shown in the second part of the paper (Section 3). The techniques that we introduce in part I will be also needed in Section 3. Moreover, we believe that it helps the reader to first see the simpler balancing procedure for string straight-line programs before going into the details of the general balancing result. Finally, the reader who is only interested in SSLP balancing can ignore part II completely.

We start with the afore-mentioned new decomposition technique for DAGs that we call symmetric centroid decomposition. The technical heart of our string balancing procedure is the linear-size SSLP that produces all suffixes of the path in a weight-balanced way (Section 2.2). Section 2.2 concludes the proof of Theorem 1.2, and Section 2.4 presents applications.

2.1 The symmetric centroid decomposition of a DAG

We start with a new decomposition of a DAG (directed acyclic graph) into disjoint paths. We believe that this decomposition might have further applications. For trees, there exist several decompositions into disjoint paths with the additional property that every path from the root to a leaf only intersects a logarithmic number of paths from the decomposition; note that in general such decompositions allow empty paths, i.e., consisting of a single node only. Examples are the heavy path decomposition [21] and centroid decomposition [12]. These decompositions can be also defined for DAGs but a technical problem is that the resulting paths are no longer disjoint and form, in general, a subforest of the DAG, see e.g. [8].

Our new decomposition can be seen as a symmetric form of the centroid decomposition of [12]. Consider a DAG $\mathcal{D} = (V, E)$ with node set V and the set of multi-edges E, i.e., E is a finite subset of $V \times \mathbb{N} \times V$ such that $(u, d, v) \in E$ implies that for every $1 \leq i < d$ there exists $v' \in V$ with $(u, i, v') \in E$. Intuitively, (u, d, v) is the d-th outgoing edge of u. We assume that there is a single root node $r \in V$, i.e., r is the unique node with no incoming edges. Hence, all nodes are reachable from r. A path from $u \in V$ to $v \in V$ is a sequence of edges $(v_0, d_1, v_1), (v_1, d_2, v_2), \ldots, (v_{p-1}, d_p, v_p)$ where $u = v_0$ and $v = v_p$. We also allow the empty path from u to u. With $\pi(u, v)$ we denote the number of paths from u to v, and for $V' \subseteq V$ let $\pi(u, V') = \sum_{v \in V'} \pi(u, v)$. Let $W \subseteq V$ be the set of sink nodes of $\mathcal{D}$, i.e., those nodes without outgoing edges, and let $n(\mathcal{D}) = \pi(r, W)$. This is the number of leaves in the tree obtained by unfolding $\mathcal{D}$ into a tree. For a node $v \in V$ consider $(\pi(r, v), \pi(v, W))$: the number of paths leading from the root to v and the number of paths leading from v to all leaves. We assign to v the pair consisting of rounded (down) logarithms of those two numbers: $\lambda_{\mathcal{D}}(v) = (\lfloor \log_2 \pi(r, v) \rfloor, \lfloor \log_2 \pi(v, W) \rfloor)$. If $\lambda_{\mathcal{D}}(v) = (k, \ell)$, then $k, \ell \leq \lfloor \log_2 n(\mathcal{D}) \rfloor$ because $\pi(r, v)$ and $\pi(v, W)$ are both bounded by $n(\mathcal{D})$. Let us now define the edge set $E_{\text{scd}}(\mathcal{D})$ ("scd" stands for symmetric centroid decomposition) as $E_{\text{scd}}(\mathcal{D}) = \{(u, i, v) \in E \mid \lambda_{\mathcal{D}}(u) = \lambda_{\mathcal{D}}(v)\}$. Figure 1 gives a detailed example of a symmetric centroid decomposition of a DAG.

Lemma 2.1. *Let $\mathcal{D} = (V, E)$ be a DAG with $n = n(\mathcal{D})$. Then every node has at most one outgoing and at most one incoming edge from $E_{scd}(\mathcal{D})$. Furthermore, every path from the root r to a sink node contains at most $2 \log_2 n$ edges that do not belong to $E_{scd}(\mathcal{D})$.*

Proof. Consider a node $v \in V$ with two different outgoing edges $(u, i, v), (u, j, w) \in E_{\text{scd}}(\mathcal{D})$. Hence, $\lambda_{\mathcal{D}}(u) = \lambda_{\mathcal{D}}(v) = \lambda_{\mathcal{D}}(w)$. Let $\lambda_{\mathcal{D}}(u) = (k, \ell)$. If W is the set of sinks, we get $\pi(u, W) \geq \pi(v, W) + \pi(w, W)$ (since we consider paths of multi-edges, this inequality also holds for $v = w$). W.l.o.g. assume that $\pi(w, W) \geq \pi(v, W)$ and thus $\pi(u, W) \geq 2\pi(v, W)$. We get

$$\lfloor \log_2 \pi(u, W) \rfloor \geq 1 + \lfloor \log_2 \pi(v, W) \rfloor = 1 + \lfloor \log_2 \pi(u, W) \rfloor,$$

where the last equality follows from $\lambda_{\mathcal{D}}(u) = \lambda_{\mathcal{D}}(v)$. This is a contradiction and proves the claim for outgoing edges. Incoming edges are treated similarly, this time using $\pi(r, v)$.

For the second claim of the Lemma, consider a path $(v_0, d_1, v_1), (v_1, d_2, v_2), \ldots, (v_{p-1}, d_p, v_p)$, where v_0 is the root and v_p is a sink. Let $\lambda_{\mathcal{D}}(v_i) = (k_i, \ell_i)$, then $k_i \leq k_{i+1}$ and $\ell_i \geq \ell_{i+1}$ for all

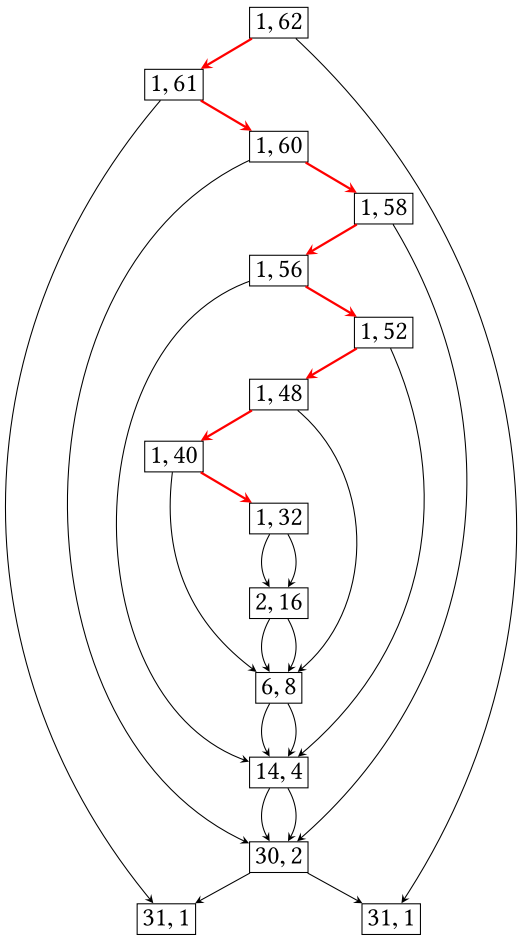

Fig. 1. A DAG and its symmetric centroid path decomposition. The numbers in a node v are the values $\pi(r, v)$, where r is the root, i.e., the number of incoming paths from r, and $\pi(v, W)$, where W consists of the two sink nodes, i.e., the number of paths from v to sinks. Edges that belong to a symmetric centroid path are drawn in red. Note that the 9 topmost nodes form a symmetric centroid path since $\lfloor \log_2 \pi(r, v) \rfloor = 0$ and $\lfloor \log_2 \pi(v, W) \rfloor = 5$ for each of these nodes. In this example the symmetric centroid path decomposition consists of one path of length 8 (number of edges); all other nodes form symmetric centroid paths of length zero, i.e., they are trivial.

$0 \le i \le p-1$. Moreover, $k_0 = \ell_p = 0$ and $\ell_0, k_p \le \lfloor \log_2 n \rfloor$. Consider now an edge $(v_i, d_i, v_{i+1}) \in E \setminus E_{\mathrm{scd}}(\mathcal{D})$. Since $\lambda_{\mathcal{D}}(v_i) \neq \lambda_{\mathcal{D}}(v_{i+1})$, we have $k_i < k_{+1}$ or $\ell_i > \ell_{i+1}$. Hence, there can be at most $2\lfloor \log_2 n \rfloor \le 2 \log_2 n$ edges from $E \setminus E_{\mathrm{scd}}(\mathcal{D})$ on the path. □

Lemma 2.1 implies that the subgraph $(V, E_{\mathrm{scd}}(\mathcal{D}))$ is a disjoint union of possibly empty paths, called *symmetric centroid paths* of $\mathcal{D}$. It is straight-forward to compute the edge set $E_{\mathrm{scd}}(\mathcal{D})$ in time $O(|\mathcal{D}|)$, where $|\mathcal{D}|$ is defined as the number of edges of the DAG: By traversing $\mathcal{D}$ in both directions (from the root to the sinks and from the sinks to the root) one can compute all pairs $\lambda_{\mathcal{D}}(v)$ for $v \in V$ in linear time.

One can use Lemma 2.1 in order to simplify the original proof of Theorem 1.1 from [8]: in [8], the authors use the heavy-path decomposition of the derivation tree of an SSLP. In the SSLP (viewed as a DAG that defines the derivation tree), these heavy paths lead to a forest, called the heavy path forest [8]. The important property used in [8] is the fact that any path from the root of the DAG to a sink node contains only $O(\log n)$ edges that do not belong to a heavy path, where n is the length of string produced by the SSLP. Using Lemma 2.1, one can replace this heavy path forest by the decomposition into symmetric centroid paths. The fact that the latter is a disjoint union of paths in the DAG simplifies the technical details in [8] a lot. On the other hand, Theorem 1.1 follows directly from Theorem 1.2, see Section 2.4.

2.2 Straight-line programs and suffixes of weighted strings

Given an alphabet of symbols Σ, Σ^* denotes the set of all finite words over the alphabet Σ, including the empty word ε. The set of non-empty words is denoted by $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. The length of a word w is denoted with $|w|$.

Let Σ be a finite alphabet of terminal symbols. A string straight-line program (SSLP for short) over the alphabet Σ is a triple $\mathcal{G} = (\mathcal{V}, \rho, S)$, where $\mathcal{V}$ is a finite set of variables, $S \in \mathcal{V}$ is the start variable, and $\rho \colon \mathcal{V} \to (\Sigma \cup \mathcal{V})^*$ (the right-hand side mapping) has the property that the binary relation $E(\mathcal{G}) = \{(X, Y) \in \mathcal{V} \times \mathcal{V} \colon Y \text{ occurs in } \rho(X)\}$ is acyclic. This allows to define for every variable $X \in \mathcal{V}$ a string $[\![X]\!]_{\mathcal{G}}$ as follows: if $\rho(X) = u_0 X_1 u_1 X_2 \cdots u_{n-1} X_n u_n$ with $u_0, u_1, \ldots, u_n \in \Sigma^*$ and $X_1, \ldots, X_n \in \mathcal{V}$ then $[\![X]\!]_{\mathcal{G}} = u_0 [\![X_1]\!]_{\mathcal{G}} u_1 [\![X_2]\!]_{\mathcal{G}} \cdots u_{n-1} [\![X_n]\!]_{\mathcal{G}} u_n$. We omit the subscript $\mathcal{G}$ if $\mathcal{G}$ is clear from the context. Finally, we define $[\![\mathcal{G}]\!] = [\![S]\!]$.

An SSLP $\mathcal{G}$ can be seen as a context-free grammar that produces the single string $[\![\mathcal{G}]\!]$. Quite often, one assumes that all right-hand sides $\rho(X)$ are from $\Sigma \cup \mathcal{V}\mathcal{V}$. This corresponds to the Chomsky normal form. For a given SSLP $\mathcal{G}$ with $[\![\mathcal{G}]\!] \neq \varepsilon$ one can construct in linear time an equivalent SSLP in Chomsky normal form [30, Proposition 3.8].

Fix an SSLP $\mathcal{G} = (\mathcal{V}, \rho, S)$. We define the size $|\mathcal{G}|$ of $\mathcal{G}$ as $\sum_{X \in \mathcal{V}} |\rho(X)|$. Let d be the length of a longest path in the DAG $(\mathcal{V}, E(\mathcal{G}))$ and $r = \max\{|\rho(X)| \colon X \in \mathcal{V}\}$. We define the depth of $\mathcal{G}$ as $\mathrm{depth}(\mathcal{G}) = d \cdot \lceil \log_2 r \rceil$. Note that for an SSLP in Chomsky normal form, the definition of the depth simplifies to $\mathrm{depth}(\mathcal{G}) = d$ and that these definitions ensure that depth and size only increase by fixed (multiplicative) constants when an SSLP is transformed into Chomsky normal form. This is exactly the purpose of this definition: transforming into Chomsky normal form is a standard preprocessing and/or normalization procedure and we want size and height to be roughly preserved during such a procedure. Lastly, observe that more refined notion of height, which takes into the account the rule sizes on particular paths, and not a bound on all of them, or the height of the actual Chomsky normal form (which is not unique) could be devised, but it is not needed for our purposes.

A *weighted string* is a string $s \in \Sigma^*$ equipped with a weight function $\|\cdot\| \colon \Sigma \to \mathbb{N} \setminus \{0\}$, which is extended to a homomorphism $\|\cdot\| \colon \Sigma^* \to \mathbb{N}$ by $\|a_1 a_2 \cdots a_n\| = \sum_{i=1}^{n} \|a_i\|$. If X is a variable in an SSLP $\mathcal{G}$, we also write $\|X\|$ for the weight of the string $[\![X]\!]_{\mathcal{G}}$ derived from X. Moreover, when we speak of suffixes of a string, we always mean non-empty suffixes.

Proposition 2.2. *For every non-empty weighted string s of length n one can construct in linear time an SSLP $\mathcal{G}$ with the following properties:*

- *$\mathcal{G}$ contains at most $3n$ variables,*
- *all right-hand sides of $\mathcal{G}$ have length at most 4,*
- *$\mathcal{G}$ contains* suffix variables *$S_1, \ldots, S_n$ producing all suffixes of s, and*
- *every path from S_i to some terminal symbol a in the derivation tree of $\mathcal{G}$ has length at most $3 + 2(\log_2 \|S_i\| - \log_2 \|a\|)$.*

Before moving to the proof, let us explain the intuition behind the definition and statement: The intended usage of the weight is that a letter a with weight $\|a\| > 1$ is in fact a variable that has its separate SSLP $\mathcal{A}$, in which it derives a string of length $\|a\|$, i.e., $\|[\![a]\!]_{\mathcal{A}}\| = \|a\|$: in our constructions we will compose several SLPs and symbols treated as letters in one SSLP are the starting variables in the other. The last condition of Proposition 2.2 fits well into this understanding: when we compose several SSLPs, then a becomes a variable deriving string of length $\|a\|$. Then several estimation of the form $3 + 2(\log_2 \|S_i\| - \log_2 \|a\|)$, $3 + 2(\log_2 \|a\| - \log_2 \|b\|), \ldots$ are added together and they form a telescopic sum, so in the end we are left with $2(\log \|S_i\| - 0)$, which is logarithmic in the length of the word derived by S_i, as desired (there are also the sums of 3s, but those are estimated separately).

Proof. First, the presented algorithm never uses the fact that some letters of s may be equal. Thus it is more convenient to assume that letters in s are pairwise different—in this way the path from a variable S_i to a terminal symbol a in the last condition is defined uniquely.

For the sake of an inductive proof, the constructed SSLP will satisfy a slightly stronger and more technical variant of the last condition: every path from S_i to some terminal symbol a in the derivation tree of $\mathcal{G}$ has length at most $1 + 2(\lceil\log_2 \|S_i\|\rceil - \log_2 \|a\|)$. The trivial estimation $\lceil\log_2 \|S_i\|\rceil \le 1 + \log_2 \|S_i\|$ then yields the announced variant.

We first show how to construct $\mathcal{G}$ with the desired properties and then prove that the construction can be done in linear time.

The case $n = 1$ is trivial. Now assume that $n \ge 2$ and let

$$s = a_1 \cdots a_k \, c \, b_1 \cdots b_m$$

where $cb_1 \cdots b_m$ is the shortest suffix of s such that $\lceil\log_2 \|cb_1 \cdots b_m\|\rceil = \lceil\log_2 \|s\|\rceil$. Clearly such a suffix exists (in the extreme cases it is the entire string s or a single letter). Note that

$$\lceil\log_2 \|cb_1 \cdots b_m\|\rceil = \lceil\log_2 \|a_i \cdots a_k cb_1 \cdots b_m\|\rceil \tag{1}$$

for $1 \le i \le k + 1$. Moreover, the following inequalities hold:

$$\lceil\log_2 \|cb_1 \cdots b_m\|\rceil \ge \lceil\log_2 \|b_1 \cdots b_m\|\rceil + 1 \tag{2}$$

$$\lceil\log_2 \|cb_1 \cdots b_m\|\rceil \ge \lceil\log_2 \|a_1 \cdots a_k\|\rceil + 1 \tag{3}$$

(here, we define $\log_2(0) = -\infty$). The former is clear from the definition of $cb_1 \cdots b_m$, as $b_1 \cdots b_m$ satisfies $\lceil\log_2 \|b_1 \cdots b_m\|\rceil < \lceil\log_2 \|s\|\rceil = \lceil\log_2 \|cb_1 \cdots b_m\|\rceil$. If (3) does not hold then both $a_1 \cdots a_k$ and $cb_1 \cdots b_m$ have weights strictly more than $2^{\lceil\log_2 \|s\|\rceil - 1}$ and so their concatenation s has weight strictly more than $2^{\lceil\log_2 \|s\|\rceil} \ge \|s\|$, which is a contradiction.

Recall that the symbols $a_1, \ldots, a_k, c, b_1, \ldots, b_m$ are pairwise different by the convention from the first paragraph of the proof.

For $b_1 \cdots b_m$ we make a recursive call (if $m = 0$ we do nothing at this step) and include the produced SSLP in the output SSLP $\mathcal{G}$. Let $V_1, V_2, \ldots, V_m$ be the variables such that

$$[\![V_i]\!]_{\mathcal{G}} = b_i \cdots b_m.$$

By the inductive assumption, every path $V_i \xrightarrow{*} b_j$ in the derivation tree has length at most

$$1 + 2\lceil\log_2 \|V_i\|\rceil - 2\log_2 \|a_j\|.$$

Add a variable V_0 with right-hand side cV_1 (or c if $m = 0$), which derives the suffix $cb_1 \cdots b_m$. The path from V_0 to c in the derivation tree has length 1, which is fine, and the path $V_0 \xrightarrow{*} a_j$ is one larger than the path $V_1 \xrightarrow{*} a_j$ and hence has length at most

$$1 + 1 + 2\lceil\log_2 \|V_1\|\rceil - 2\log_2 \|a_j\| \le 2\lceil\log_2 \|V_0\|\rceil - 2\log_2 \|a_j\|,$$

as $1 + \lceil \log_2 \|V_1\| \rceil \leq \lceil \log_2 \|V_0\| \rceil$ by (2).

Next we decompose the prefix $a_1 \cdots a_k$ into $\lfloor k/2 \rfloor$ many blocks of length two and, when k is odd, one block of length 1, consisting of a_k. We add to the output SSLP $\mathcal{G}$ new variables $X_1, \ldots, X_{\lfloor k/2 \rfloor}$ and define their right-hand sides by

$$\rho(X_i) = a_{2i-1} a_{2i}.$$

The number of variables in $\mathcal{G}$ is $\lfloor k/2 \rfloor$. For ease of presentation, when k is odd, define $X_{\lceil k/2 \rceil} = a_k$, this is not a new variable, rather just a notational convention to streamline the presentation. Note that for even k we have $\lceil k/2 \rceil = \lfloor k/2 \rfloor$ and in this case $X_{\lceil k/2 \rceil}$ is already defined. Viewing $X_1 \cdots X_{\lceil k/2 \rceil}$ as a weighted string of length $\lceil k/2 \rceil$ over the alphabet $\{X_1, \ldots, X_{\lceil k/2 \rceil}\}$, we obtain inductively an SSLP $\mathcal{G}_X$ with at most $3\lceil k/2 \rceil$ variables and right-hand sides of length at most 4 (if $k = 0$ we do nothing at this step). Moreover, $\mathcal{G}_X$ contains variables $U_1, U_2, \ldots, U_{\lceil k/2 \rceil}$ with

$$[\![U_i]\!]_{\mathcal{G}_X} = X_i X_{i+1} \cdots X_{\lceil k/2 \rceil}$$

such that any path of the form $U_i \xrightarrow{*} X_j$ in the derivation tree of $\mathcal{G}_X$ has length at most

$$1 + 2\lceil \log_2 \|U_i\| \rceil - 2 \log_2 \|X_j\|.$$

By adding all variables and right-hand side definitions from $\mathcal{G}_X$ to $\mathcal{G}$ (where all symbols X_i are variables, except $X_{\lceil k/2 \rceil}$ when k is odd, in which case $X_{\lceil k/2 \rceil} = a_k$) we obtain

$$[\![U_i]\!]_{\mathcal{G}} = a_{2i-1} a_{2i} \cdots a_k$$

for all $1 \leq i \leq \lceil k/2 \rceil$. Any path $U_i \xrightarrow{*} a_j$ in the derivation tree of $\mathcal{G}$ has length at most

$$2 + 2\lceil \log_2 \|U_i\| \rceil - 2 \log_2 \|a_j\|. \tag{4}$$

Now, every suffix of s that includes some letter of $a_1 \cdots a_k$ (note that we already have variables for all other suffixes) can be defined by a right-hand side of the form $U_i c V_1$ or $a_{2i-2} U_i c V_1$ ($U_i c$ or $a_{2i-2} U_i c$ if $m = 0$). As in the statement of the lemma, denote those variables by $S_1, \ldots, S_k$. Let us next verify the condition on the path lengths for derivations from those variables. All paths $S_i \xrightarrow{*} c$ have length one. Now consider a path $S_i \xrightarrow{*} a_j$. If the path has length one then we are done. Otherwise, the path must be of the form $S_i \to U_l \xrightarrow{*} a_j$. Therefore, by (4) the path length is at most

$$\begin{aligned} 3 + 2\lceil \log_2 \|U_l\| \rceil - 2 \log_2 \|a_j\| &\leq 3 + 2\lceil \log_2 \|U_1\| \rceil - 2 \log_2 \|a_j\| \\ &= 3 + 2\lceil \log_2 \|a_1 \cdots a_k\| \rceil - 2 \log_2 \|a_j\| \\ &\leq 1 + 2\lceil \log_2 \|c b_1 \cdots b_m\| \rceil - 2 \log_2 \|a_j\| \\ &= 1 + 2\lceil \log_2 \|S_i\| \rceil - 2 \log_2 \|a_j\|, \end{aligned}$$

where the second inequality follows from (3) and the equality at the end follows from (1).

Paths of the form $S_i \xrightarrow{*} b_j$ can be treated similarly: they are of the form $S_i \to V_1 \xrightarrow{*} b_j$, where the path $V_1 \xrightarrow{*} b_j$ is of length at most $1 + 2\lceil \log_2 \|V_1\| \rceil - 2 \log_2 \|a_j\|$ by the inductive assumption. Thus, the whole path is of length at most

$$\begin{aligned} 2 + 2\lceil \log_2 \|V_1\| \rceil - 2 \log_2 \|b_j\| &\leq 2\lceil \log_2 \|c b_1 \cdots b_m\| \rceil - 2 \log_2 \|b_j\| \\ &= 2\lceil \log_2 \|S_i\| \rceil - 2 \log_2 \|b_j\|, \end{aligned}$$

which follows from (2) and (1).

The SSLP $\mathcal{G}$ consists of $\lfloor k/2 \rfloor$ variables X_i, $3(\lceil k/2 \rceil)$ variables from the recursive call for the weighted string $X_1 \cdots X_{\lceil k/2 \rceil}$, $3m = 3(n - k - 1)$ variables from the recursive call for $b_1 \cdots b_m$, and

$1+k$ new suffix variables for suffixes beginning at $a_1 \cdots a_k c$ (note that those beginning at $b_1 \cdots b_m$ are taken care of by the recursive call). Therefore $\mathcal{G}$ contains at most

$$\lfloor k/2 \rfloor + 3\lceil k/2 \rceil + 3(n-k-1) + 1 + k = 3n + 2\lceil k/2 \rceil - k - 2 < 3n$$

variables. Also note that all right-hand sides of $\mathcal{G}$ have length at most four.

It remains to show that the construction works in linear time. To this end we need a small trick: we assume that when the algorithm is called on s, we supply the algorithm with the value $\|s\|$. More formally, the main algorithm applied to a string s computes $\|s\|$ in linear time by going through s and adding weights. Then it calls a subprocedure main$'(s, \|s\|)$, which performs the actions described above. To find the appropriate symbol c, main$'$ computes the weights of consecutive prefixes $s_1 s_2 \cdots s_i$, until it finds the first such that $\lceil \log_2 \|s\| \rceil > \lceil \log_2(\|s\| - \|s_1 \cdots s_i\|) \rceil$. Then $k = i-1$ and so $a_1 \cdots a_k = s_1 \cdots s_{i-1}$, $c = s_i$, $b_1 \cdots b_m = s_{i+1} \cdots s_{|s|}$. Moreover, we can compute $\|a_1 \cdots a_k\|$ and $\|b_1 \cdots b_m\|$ for the recursive calls of main$'$ in constant time.

Let $T(n)$ be the running time of main$'$ on a word of length n. Then all operations of main$'$, except the recursive calls, take at most $\alpha(k+1)$ time for some constant $\alpha \geq 1$, where s is represented as $a_1 \cdots a_k c b_1 \cdots b_m$. Thus $T(n)$ satisfies $T(1) = 1$ and

$$T(n) = T(\lceil k/2 \rceil) + T(n-k-1) + \alpha(k+1).$$

We claim that $T(n) \leq 2\alpha n$. This is true for $n = 1$ and inductively for $n \geq 2$ we get

$$\begin{aligned} T(n) &\leq 2\alpha(\lceil k/2 \rceil) + 2\alpha(n-k-1) + \alpha(k+1) \\ &\leq 2\alpha\frac{k+1}{2} + 2\alpha n - \alpha(k+1) \\ &= 2\alpha n. \end{aligned}$$

This concludes the proof of the lemma. □

2.3 Proof of Theorem 1.2

We now prove Theorem 1.2. Let $\mathcal{G} = (\mathcal{V}, \rho_{\mathcal{G}}, S)$. W.l.o.g. we can assume that $\mathcal{G}$ is in Chomsky normal form (the case that $[\![G]\!] = \varepsilon$ is trivial). Note that the graph $(\mathcal{V}, E(\mathcal{G}))$ is a directed acyclic graph (DAG). We can assume that every variable is reachable from the start variable S. Consider a variable X with $\rho_{\mathcal{G}}(X) = YZ$. Then X has the two outgoing edges (X, Y) and (X, Z) in $(\mathcal{V}, E(\mathcal{G}))$. We replace these two edges by the triples $(X, 1, Y)$ and $(X, 2, Z)$. Hence, $\mathcal{D} := (\mathcal{V}, E(\mathcal{G}))$ becomes a DAG with multi-edges (triples from $\mathcal{V} \times \{1,2\} \times \mathcal{V}$). Figure 2 shows the DAG $\mathcal{D}$ for an example SSLP; it is the same DAG as in Figure 1. The right-hand sides for the two sink variables X_{13} and X_{14} are terminal symbols (the concrete terminals are not relevant for us). The start variable S is X_0.

We define for every $X \in \mathcal{V}$ the weight $\|X\|$ as the length of the string $[\![X]\!]_{\mathcal{G}}$. Moreover, for a string $w = X_1 X_2 \cdots X_n$ we define the weight $\|w\| = \sum_{i=1}^{n} \|X_i\|$. Note that $\|S\| = n$ is the length of the derived string $[\![\mathcal{G}]\!]$ and that this also the value $n(\mathcal{D})$ defined in Section 2.1.

We compute in linear time the edges from symmetric centroid decomposition of the DAG $\mathcal{D}$, see Lemma 2.1. In Figure 2 these are the red edges. The weights $\|X_i\|$ of the variables are written next to the corresponding nodes; these weights can be found as the second components in Figure 1. Hence, we have $\|X_0\| = 62$, $\|X_1\| = 61$, $\|X_2\| = 60$, $\|X_3\| = 58$, etc.

Consider a symmetric centroid path

$$(X_0, d_0, X_1), (X_1, d_1, X_2), \ldots, (X_{p-1}, d_{p-1}, X_p) \qquad (5)$$

in $\mathcal{D}$, where all X_i belong to $\mathcal{V}$ and $d_i \in \{1,2\}$. Thus, for all $0 \leq i \leq p-1$, the right-hand side of X_i in $\mathcal{G}$ has the form $\rho_{\mathcal{G}}(X_i) = X_{i+1} X'_{i+1}$ (if $d_i = 1$) or $\rho_{\mathcal{G}}(X_i) = X'_{i+1} X_{i+1}$ (if $d_i = 2$) for some $X'_{i+1} \in \mathcal{V}$. Note that we can have $X'_i = X'_j$ for $i \neq j$, this happens for instance on Figure 2, it can

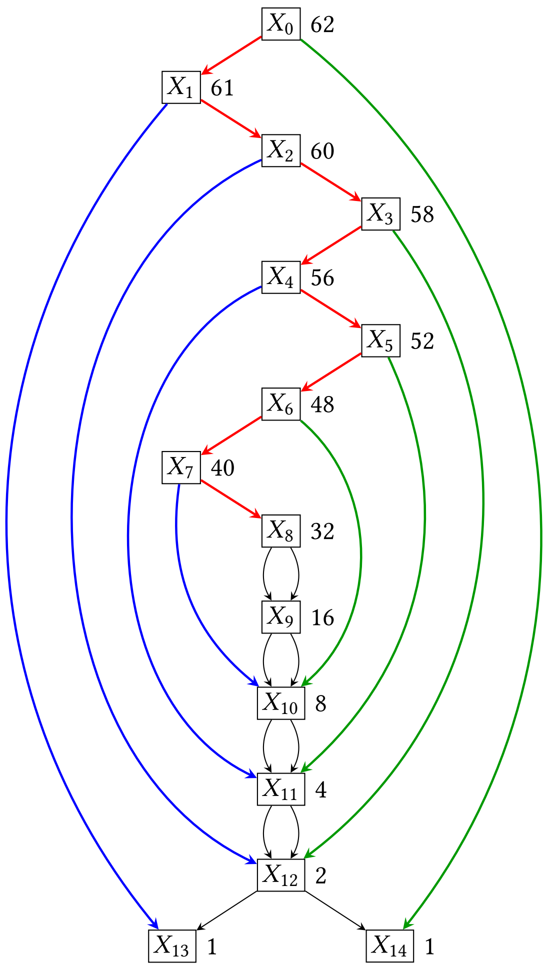

Fig. 2. The DAG for an SSLP (it is the the same DAG as in Figure 1). The only non-trivial symmetric centroid path is drawn in red, as in Figure 1. Each node is annotated with the length of the derived string. The blue edges are the left edges of nodes whose right edges are on the symmetric centroid path (i.e., are red), similarly green edges are the right edges such that the corresponding left edges are on the symmetric centroid path. All other edges are black. If a node has an outgoing black edge then both its outgoing edges are black and this node is the last node on some symmetric centroid path (this includes the case, when it is the only node on the path, i.e., when the path is trivial). Using the notation from the proof of Theorem 1.2, the node X_{10} is both X'_7 and X'_8, as there are edges outside the centroid path from X_6 and X_7 to X_{10}.

also happen that $X_{i'} = X_j$ for $j > i$. The right-hand side $\rho_{\mathcal{G}}(X_p)$ belongs to $\Sigma \cup \mathcal{V}\mathcal{V}$. Note that the variables X'_i $(1 \le i \le p)$ and the variables in $\rho_{\mathcal{G}}(X_p)$ (if they exist) belong to other symmetric centroid paths. We will introduce $O(p)$ many variables in the SSLP $\mathcal{H}$ to be constructed. Moreover, all right-hand sides of $\mathcal{H}$ have length at most four. By summing over all symmetric centroid paths, this yields the size bound $O(|\mathcal{G}|)$ for $\mathcal{H}$.

We now define the right-hand sides of the variables $X_0, \ldots, X_p$ in $\mathcal{H}$. We write $\rho_{\mathcal{H}}$ for the right-hand side mapping of $\mathcal{H}$. For X_p we set $\rho_{\mathcal{H}}(X_p) = \rho_{\mathcal{G}}(X_p)$. For the variables $X_0, \ldots, X_{p-1}$ we have to "accelerate" the derivation somehow in order to get the depth bound $O(\log n)$ at the end.

For this, we apply Proposition 2.2. Let $L_1 \cdots L_s$ be the subsequence obtained from $X'_1 X'_2 \cdots X'_p$ by keeping only those X'_i with $d_{i-1} = 2$ for $i = 1, \dots, p$ and let $R_1 \cdots R_t$ be the subsequence obtained from the reversed sequence $X'_p X'_{p-1} \cdots X'_1$ by keeping only those X'_i with $d_{i-1} = 1$ for $i = p, \dots, 1$. Take for instance the red symmetric centroid path consisting of the nodes $X_0, X_1, \dots, X_8$ (hence, $p = 8$) from our running example in Figure 2. We have $L_1 \cdots L_s = X_{13} X_{12} X_{11} X_{10}$ (the target nodes of the blue edges) and $R_1 \cdots R_t = X_{10} X_{11} X_{12} X_{14}$ (the target nodes of the green edges).

Note that every string $[\![X_i]\!]$ ($0 \leq i \leq p-1$) can be derived in $\mathcal{G}$ from a word $w_\ell X_p w_r$, where w_ℓ is a suffix of $L_1 \cdots L_s$ and w_r is a prefix of $R_1 \cdots R_t$. For instance, $[\![X_2]\!]$ can be derived from $(X_{12} X_{11} X_{10}) X_8 (X_{10} X_{11} X_{12})$ in our running example, so $w_\ell = X_{12} X_{11} X_{10}$ and $w_r = X_{10} X_{11} X_{12}$. We now apply Proposition 2.2 to the sequence $L_1 \cdots L_s$ in order to get an SSLP $\mathcal{G}_\ell$ of size $O(s) \leq O(p)$ that contains variables $S_1 \dots, S_s$ for the non-empty suffixes of $L_1 \cdots L_s$. Moreover, every path from a variable S_i to some L_j in the derivation tree has length at most $3 + 2\log_2 \|S_i\| - 2\log_2 \|L_j\|$, where $\|S_i\|$ is the weight of $[\![S_i]\!]_{\mathcal{G}_\ell}$. Analogously, we obtain an SSLP $\mathcal{G}_r$ of size $O(t) \leq O(p)$ that contains variables $P_1 \dots, P_t$ for the non-empty prefixes of $R_1 \cdots R_t$. Moreover, every path from a variable P_i to some R_j in the derivation tree has length at most $3 + 2\log_2 \|P_i\| - 2\log_2 \|R_j\|$. We can then define every right-hand side $\rho_{\mathcal{H}}(X_i)$ as $S_j X_p P_k$, $X_p P_k$, $S_j X_p$, or X_p for suitable j and k. Moreover, we add all variables and right-hand side definitions of $\mathcal{G}_\ell$ and $\mathcal{G}_r$ to $\mathcal{H}$.

We make the above construction for all symmetric centroid paths of the DAG $\mathcal{D}$. This concludes the construction of $\mathcal{H}$. In our running example we set $\rho_{\mathcal{H}}(X_i) = \rho_{\mathcal{G}}(X_i)$ for $8 \leq i \leq 14$. Since we introduce $O(p)$ many variables for every symmetric centroid path of length p and all right-hand sides of $\mathcal{H}$ have length at most four, we obtain the size bound $O(|\mathcal{G}|)$ for $\mathcal{H}$.

It remains to show that the depth of the SSLP $\mathcal{H}$ is $O(\log n)$. Let us first consider the symmetric centroid path (5) and a path in the derivation tree of $\mathcal{H}$ from a variable X_i ($0 \leq i \leq p$) to a variable Y, where Y is

(a) a variable in $\rho_{\mathcal{G}}(X_p) = \rho_{\mathcal{H}}(X_p)$ or
(b) a variable X'_j for some $i < j \leq p$.

In case (a), the path $X_i \overset{*}{\to} Y$ has length at most two. In case (b) the path $X_i \overset{*}{\to} Y$ is of the form $X_i \to S_k \overset{*}{\to} X'_j = Y$ or $X_i \to P_k \overset{*}{\to} X'_j = Y$. Here, $S_k \overset{*}{\to} X'_j$ (resp., $P_k \overset{*}{\to} X'_j$) is a path in $\mathcal{G}_\ell$ (resp., $\mathcal{G}_r$) and therefore has length at most $3 + 2\log_2 \|S_k\| - 2\log_2 \|Y\|$ (resp., $3 + 2\log_2 \|P_k\| - 2\log_2 \|Y\|$). In both cases, we can bound the length of the path $X_i \overset{*}{\to} Y$ by $4 + 2\log_2 \|X_i\| - 2\log_2 \|Y\|$.

Consider a maximal path in the derivation tree of $\mathcal{H}$ that starts in the root S and ends in a leaf. We can factorize this path as

$$S = X_0 \overset{*}{\to} X_1 \overset{*}{\to} X_2 \overset{*}{\to} \cdots \overset{*}{\to} X_k \tag{6}$$

where all variables X_i belong to the original SSLP and every subpath $X_i \overset{*}{\to} X_{i+1}$ is of the form $X_i \overset{*}{\to} Y$ considered in the previous paragraph; note that X_i in general is not a parent of X_{i+1}, which is the case in the running example of Figure 2. The right-hand side of X_k is a single symbol from Σ. In the DAG $\mathcal{D}$ we have a corresponding path $X_i \overset{*}{\to} X_{i+1}$, which is contained in a single symmetric centroid path except for the last edge leading to X_{i+1}. By the above consideration, the length of the path (6) is bounded by

$$\sum_{i=0}^{k-1} (4 + 2\log_2 \|X_i\| - 2\log_2 \|X_{i+1}\|) \leq 4k + 2\log_2 \|S\| = 4k + 2\log_2 n.$$

By the second claim of Lemma 2.1 we have $k \leq 2\log_2 n$ which shows that the length of the path (6) is bounded by $10\log_2 n$. □

2.4 Applications of Theorem 1.2

There are several algorithmic applications of Theorem 1.2 with always the same idea: let $\mathcal{G}$ be an SSLP of size m for a string s of length n. In many algorithms for SSLP-compressed strings the running time or space consumption depends on $\text{depth}(\mathcal{G})$, which can be m in the worst case. Theorem 1.2 shows that we can replace $\text{depth}(\mathcal{G})$ by $O(\log n)$. This is the best we can hope for since $\text{depth}(\mathcal{G}) \geq \Omega(\log n)$ for every SSLP $\mathcal{G}$. Moreover, SSLPs that are produced by practical grammar-based compressors (e.g., LZ78 or RePair) are in general unbalanced in the sense that $\text{depth}(\mathcal{G}) \geq \omega(\log n)$.

The time bounds in the following results refer to the RAM model, where arithmetic operations on numbers from the interval $[0, n]$ need time $O(1)$. The size of a data structure is measured in the number of words of bit length $\log_2 n$.

As a first application of Theorem 1.2 we can present a very simple new proof of Theorem 1.1 (random access for grammar-compressed strings) based on the folklore random access algorithm that works in time $O(\text{depth}(\mathcal{G}))$.

Proof of Theorem 1.1. Using Theorem 1.2 we compute in time $O(m)$ an equivalent SSLP $\mathcal{H}$ for s of size $O(m)$ and depth $O(\log n)$. By a single pass over $\mathcal{H}$ we compute for every variable X of $\mathcal{H}$ the length of the word $[\![X]\!]$. Using these lengths one can descend in the derivation tree $[\![\mathcal{H}]\!]$ from the root to the i-th leaf node (which is labelled with the i-th symbol of s) in time $O(\text{depth}(\mathcal{H})) \leq O(\log n)$. □

Remark 2.3. It is easy to see that the balancing algorithm from Theorem 1.2 can be implemented on a pointer machine, see [37] for a discussion of the pointer machine model. This yields a pointer machine implementation of the random access data structure from Theorem 1.1. In contrast, the random access data structure from [8] needs the RAM model (for the pointer machine model only preprocessing time and size $O(m \cdot \alpha_k(m))$ for any fixed k, where α_k is the k-th inverse Ackermann function, is shown in [8]). On the other hand, recently, in [5], the $O(m)$-space data structure from [8] has been modified so that it can be implemented on a pointer machine as well.

Using fusion trees [14] one can improve the time bound in Theorem 1.1 to $O(\log n/ \log\log n)$ at the cost of an additional factor of $O(\log^{\epsilon} n)$ in the size bound. The following result has been shown in [2, Theorem 2] under the assumption that the input SSLP has depth $O(\log n)$. We can enforce this bound with Theorem 1.2.

Corollary 2.4. *Fix an arbitrary constant $\epsilon > 0$. From a given SSLP $\mathcal{G}$ of size m such that the string $s = [\![\mathcal{G}]\!]$ has length n, one can construct in time $O(m \cdot \log^{\epsilon} n)$ a data structure of size $O(m \cdot \log^{\epsilon} n)$ that allows to answer random access queries in time $O(\log n/ \log\log n)$.*

Proof. The proof is exactly the same as for [2, Theorem 2]. There, the authors have to assume that the input SSLP has depth $O(\log n)$, which we can enforce by Theorem 1.2. Roughly speaking, the idea in [2] is to reduce the depth of the SSLP to $O(\log n/ \log\log n)$ by expanding right-hand sides to length $O(\log^{\epsilon} n)$. Then for each right-hand side a fusion tree is constructed, which allows to spend constant time at each variable during the navigation to the i-th symbol.

Let us also remark that the size bound for the computed data structure in [2] is given in bits, which yields $O(m \cdot \log^{1+\epsilon} n)$ bits since numbers from $[0, n]$ have to be encoded with $\log_2 n$ bits. □

In the general case of SSLPs, the balancing in [2, Theorem 3] was ensured using known algorithm [36], at the cost of increasing the SSLP size by $\log(n/m)$ and so the resulting data structure had size $O(m \cdot \log^{\epsilon} n \cdot \log(n/m))$.

Given a string $s \in \Sigma^*$, a rank query gets a position $1 \leq i \leq |s|$ and a symbol $a \in \Sigma$ and returns the number of a's in the prefix of s of length i. A select query gets a symbol $a \in \Sigma$ and returns the position of the i-th a in s (if it exists). Similarly to random access queries, the following result was shown [2, Theorem 2] assuming that the input SSLP is balanced, which we can ensure using Theorem 1.2.

Corollary 2.5. *Fix an arbitrary constant $\epsilon > 0$. From a given SSLP $\mathcal{G}$ of size m such that the string $s = [\![\mathcal{G}]\!]$ has length n, one can construct in time $O(m \cdot |\Sigma| \cdot \log^{\epsilon} n)$ a data structure of size $O(m \cdot |\Sigma| \cdot \log^{\epsilon} n)$ that allows to answer rank and select queries in time $O(\log n / \log\log n)$.*

Proof. Again we follow the proof [2, Theorem 2] but first apply Theorem 1.2 in order to reduce the depth of the SSLP to $O(\log n)$. □

For arbitrary grammars in [2, Theorem 3] the balancing increased the SSLP size by a factor of $\log(n/m)$, and so the resulting data structure answering rank and select queries in time $O(\log n / \log\log n)$ had size $O(m \cdot |\Sigma| \cdot \log^{\epsilon} n \cdot \log(n/m))$.

Our balancing result also yields an improvement for the *compressed subsequence problem* [3]. Bille et al. [3] present an algorithm based on a *labelled successor* data structure. Given a string $s = a_1 \cdots a_n \in \Sigma^*$, a labelled successor query gets a position $1 \leq i \leq n$ and a symbol $a \in \Sigma$ and returns the minimal position $j > i$ with $a_j = a$ (or rejects if it does not exist). The following result is an improvement over [3], where the authors present two algorithms for the compressed subsequence problem: one with $O(m + m \cdot |\Sigma|/w)$ preprocessing time and $O(\log n \cdot \log w)$ query time, and another algorithm with $O(m + m \cdot |\Sigma| \cdot \log w / w)$ preprocessing time and $O(\log n)$ query time.

Corollary 2.6. *There is a data structure supporting labelled successor (and predecessor) queries on a string $s \in \Sigma^*$ of length n compressed by an SSLP of size m in the word RAM model with word size $w \geq \log_2 n$ using $O(m + m \cdot |\Sigma|/w)$ space, $O(m + m \cdot |\Sigma|/w)$ preprocessing time, and $O(\log n)$ query time.*

Proof. In the preprocessing phase we first reduce the depth of the given SSLP to $O(\log n)$ using Theorem 1.2. We compute for every variable X the length of $[\![X]\!]$ in time and space $O(m)$ as in the proof of Theorem 1.1. Additionally for every variable X we compute a bitvector of length $|\Sigma|$ which encodes the set of symbols $a \in \Sigma$ that occur in $[\![X]\!]$. Notice that this information takes $O(m \cdot |\Sigma|)$ bits and fits into $O(m \cdot |\Sigma|/w)$ memory words. If $\rho(X) = YZ$ then the bitvector of X can be computed from the bitvectors of Y and Z by $O(|\Sigma|/w)$ many bitwise OR operations. Hence in total all bitvectors can be computed in time $O(m \cdot |\Sigma|/w)$.

A labelled successor query (for position i and symbol a) can now be answered in $O(\log n)$ time in a straightforward way: First we compute the path $(X_0, X_1, \ldots, X_\ell)$ in the derivation tree from the root X_0 to the symbol at the i-th position. Then we follow the path starting from the leaf upwards to find the maximal k such that $\rho(X_k) = X_{k+1}Y$ and $[\![Y]\!]$ contains the symbol a, or reject if no such k exists. Finally, starting from Y we navigate in time $O(\log n)$ to the leftmost leaf in the derivation tree which produces the symbol a. □

A *minimal subsequence occurrence* of a string $p = a_1 a_2 \cdots a_k$ in a string $s = b_1 b_2 \cdots b_l$ is given by two positions i, j with $1 \leq i \leq j \leq l$ such that p is a subsequence of $b_i b_{i+1} \cdots b_j$ (i.e., $b_i b_{i+1} \cdots b_j$ belongs to the language $\Sigma^* a_1 \Sigma^* a_2 \cdots \Sigma^* a_k \Sigma^*$) but p is neither a subsequence of $b_{i+1} \cdots b_j$ nor of $b_i \cdots b_{j-1}$. Following the proof of [3, Theorem 1] we obtain:

Corollary 2.7. *Given an SSLP $\mathcal{G}$ of size m producing a string $s \in \Sigma^*$ of length n and a pattern $p \in \Sigma^*$ one can compute all minimal subsequence occurrences of p in s in space $O(m + m \cdot |\Sigma|/w)$ and time $O(m + m \cdot |\Sigma|/w + |p| \cdot \log n \cdot occ)$ where $w \geq \log n$ is the word size and occ is the number of minimal subsequence occurrences of p in s.*

Corollary 2.7 improves [3, Theorem 1], which states the existence of two algorithms for the computation of all minimal subsequence occurrences with the following running times (the space bounds are the same as in Corollary 2.7):

- $O(m + m \cdot |\Sigma|/w + |p| \cdot \log n \cdot \log w \cdot \mathrm{occ})$,
- $O(m + m \cdot |\Sigma| \cdot \log w/w + |p| \cdot \log n \cdot \mathrm{occ})$.

Let us briefly mention some other application of Theorem 1.2. As before let $\mathcal{G}$ be an SSLP of size m for a string s of length n.

Computing Karp-Rabin fingerprints for compressed strings. This problem has been studied in [6], where the reader can also finde the definition of finger prints. Given two positions $i \leq j$ in s one wants to compute the Karp-Rabin fingerprint of the factor of s that starts at position i and ends at position j. In [6] it was shown that one can compute from $\mathcal{G}$ a data structure of size $O(m)$ that allows to compute fingerprints in time $O(\log n)$. First, the authors of [6] present a very simple data structure of size $O(m)$ that allows to compute fingerprints in time $O(\mathrm{depth}(\mathcal{G}))$. With Theorem 1.2, we can use this data structure to obtain a $O(\log n)$-time solution. This simplifies the proof in [6] considerably.

Computing runs, squares, and palindromes in SSLP-compressed strings. It is shown in [23] that certain compact representations of the set of all runs, squares and palindromes in s (see [23] for precise definitions) can be computed in time $O(m^3 \cdot \mathrm{depth}(\mathcal{G}))$. With Theorem 1.2 we can improve the time bound to $O(m^3 \cdot \log n)$.

Real time traversal for SSLP-compressed strings. One wants to output the symbols of s from left to right and thereby spend constant time per symbol. A solution can be found in [19]; a two-way version (where one can navigate in each step to the left or right neighboring position in s) can be found in [33]. The drawback of these solutions is that they need workspace $O(\mathrm{depth}(\mathcal{G}))$. With Theorem 1.2 we can reduce this to workspace $O(\log n)$.

Compressed range minimum queries. Range minimum data structure preprocesses a given string s of integers so that the following queries can be efficiently answered: given $i \leq j$, what is the minimum element in $s_i, \ldots, s_j$ (the substring of s from position i to j). We are interested in the variant of the problem, in which the input is given as an SSLP $\mathcal{G}$. It is known, that after a preprocessing taking $O(|\mathcal{G}|)$ time, one can answer range minimum queries in time $O(\log n)$ [20, Theorem 1.1]. This implementation extends the data structure for random access for SSLP [8] with some additional information, which includes in particular adding standard range minimum data structures for subtrees leaving the heavy path and extending the original analysis. Using the balanced SSLP the same running time can be easily obtained, treating the balanced SSLP construction as a black-box, without the need of enhancing it. To this end for each variable X we store the length ℓ_X of the derived word $[\![X]\!]$ as well the minimum value in $[\![X]\!]$. In the following, let $\mathrm{RMQ}(X, i, j)$ be the range minimum query called on $[\![X]\!]$ for interval $[i, j]$. Given $\mathrm{RMQ}(X, i, j)$, with the rule for X being $X \to YZ$ we proceed as follows:

- If the query asks about the minimum in the whole $[\![X]\!]$, i.e., $i = 1$ and $j = \ell_X$, then we return the minimum of $[\![X]\!]$; we call this case trivial in the following.
- If the whole range is within the substring generated by the first variable in the rule, i.e., $j \leq \ell_Y$, then we call $\mathrm{RMQ}(Y, i, j)$.
- If the whole range is within the substring generated by the second nonterminal in a rule, i.e., $i > \ell_Y$, then we call $\mathrm{RMQ}(Z, i - \ell_Y, j - \ell_Y)$.

- Otherwise, i.e., when $i \leq \ell_Y$ and $j > \ell_Y$ and $(i,j) \neq (1,\ell_X)$, the range spans over the substrings generated by both nonterminals. Thus we compute the queries for two substrings and take their minimum, i.e., we return the minimum of $\mathrm{RMQ}(Y,i,\ell_Y)$ and $\mathrm{RMQ}(Z,1,j-\ell_Y)$.

To see that the running time is $O(\mathrm{depth}(\mathcal{G})) = O(\log n)$ observe first that the cost of trivial cases can be charged to the function that called them. Thus it is enough to estimate the number of nontrivial recursive calls. In the second and third case there is only one recursive call for a variable that is deeper in the derivation tree of the SSLP. In the fourth case there are two calls, but two nontrivial calls are made at most once during the whole computation: if two nontrivial calls are made in the fourth case then one of them asks for the RMQ of a suffix of $[\![Y]\!]$ and the other call asks for the RMQ of a prefix of $[\![Z]\!]$. Moreover, every recursive call on a prefix of some string $[\![X']\!]$ leads to at most one nontrivial call, which is again on a prefix of some string $[\![X'']\!]$; and analogously for suffixes.

Lifshits' algorithm for compressed pattern matching [28]. The input consists of an SSLP $\mathcal{P}$ for a pattern p and an SSLP $\mathcal{T}$ for a text t and the question is whether p occurs in t. Lifshits' algorithm has a running time of $O(|\mathcal{P}| \cdot |\mathcal{T}|^2)$. It was conjectured by the author that the running time could be improved to $O(|\mathcal{P}| \cdot |\mathcal{T}| \cdot \log |t|)$. This follows easily from Theorem 1.2: the algorithm fills a table of size $|\mathcal{P}| \cdot |\mathcal{T}|$ and on each entry it calls a recursive subprocedure, whose running time is at most $\mathrm{depth}(\mathcal{T})$. By Theorem 1.2 we can bound the running time by $O(\log |t|)$, which proves Lifshits' conjecture. Note, that in the meantime a faster algorithm with running time $O(|\mathcal{T}| \cdot \log |p|)$ [25] was found.

Smallest grammar problem. We conclude Part I of the paper with a remark on the so-called *smallest grammar problem* for strings. In this problem one wants to compute for a given string w a smallest SSLP defining w. The decision variant of this problem is NP-hard, the best known approximation lower bound is $\frac{8569}{8568}$ [11], and the best known approximation algorithms have an approximation ratio of $O(\log(n/m_{\mathrm{opt}}))$, where n is the length of the input string and m_{opt} the size of the smallest SSLP for the input string [11, 24, 26, 36]. Except for [24], all these algorithms produce SSLPs of depth $O(\log n)$. It was discussed in [24] that the reason for the lack of constant-factor approximation algorithms might be the fact that smallest SSLPs can have larger than logarithmic depth. Theorem 1.2 refutes this conjecture.

3 PART II: BALANCING CIRCUITS OVER ALGEBRAS

In this second part of the paper we prove our general balancing result Theorem 1.4.

Example of free monoids. For a gentle introduction into this more algebraic part of the paper, let us illustrate the main ideas using the example of a free monoid $\mathcal{A} = \Sigma^*$, which is the underlying algebraic structure of SSLPs. Recall that in the proof of Theorem 1.2 we treated every symmetric centroid path $X_0, X_1, \ldots, X_p$ individually. Let $L_1, \ldots, L_s$ be the variables branching off the path to the left and $R_1, \ldots, R_t$ be the variables branching off the path to the right. We were able to produce all suffixes of $L_1 \cdots L_s$ and all prefixes of $R_1 \cdots R_t$ in "small depth" by the suffix lemma on weighted strings (Proposition 2.2). Then every variable X_i can be written as a $u_i X_p v_i$ where u_i is a suffix of $L_1 \ldots L_s$ and v_i is a prefix of $R_1 \ldots R_t$.

A different approach would be to introduce special variables Z_i which produce strings with "holes" $u_i * v_i$, i.e., we replace in the above right-hand side $u_i X_p v_i$ the variable X_p (the last last variable on the symmetric centroid path) by the hole $*$. For the moment, we allow such variables Z_i, which will be eliminated in a second step. Observe that the variable Z_i is obtained from Z_{i+1} by inserting either L_j* or $*R_j$ for some j into the hole $*$. In the end X_i is obtained from Z_i by inserting

X_p, which closes the hole. Again, using the suffix lemma on weighted strings, we can balance the computation of the variables Z_i to achieve logarithmic depth of the SLP.

What we obtain in this way is not a string SLP anymore, but an SLP (or a circuit) over a two-sorted algebra $\hat{\mathcal{A}}$: besides the sort of strings from Σ^*, it contains a second sort, namely so called *contexts* (strings with holes), which can be formalized as functions $x \mapsto uxv$ on Σ^*. We write $\text{lin}(\Sigma^*)$ for the set of all such functions (the function $x \mapsto uxv$ is a linear function on Σ^*). The above variables Z_i would be of sort $\text{lin}(\Sigma^*)$; the hole $*$ in $u_i * v_i$ markes the position of the variable x.

Besides the usual concatenation on strings (which is an operation from $\Sigma^* \times \Sigma^*$ to Σ^*) the algebra $\hat{\mathcal{A}}$ contains an operation $\alpha \colon \Sigma^* \times \text{lin}(\Sigma^*) \to \Sigma^*$ to insert a string into a context (which corresponds to function application) and an operation $\gamma \colon \text{lin}(\Sigma^*) \times \text{lin}(\Sigma^*) \to \text{lin}(\Sigma^*)$ to insert a context into another context (which corresponds to the composition of functions). Finally, it has two operations that take a string w as input and return the context $w*$, respectively $*w$. An SLP over $\hat{\mathcal{A}}$ contains variables of two sorts, which either produce strings or contexts. Every variable is computed from other variables by applying the operations from the algebra $\hat{\mathcal{A}}$.

In a second step, the SLP over $\hat{\mathcal{A}}$ of logarithmic depth (whose construction we have sketched above) can be turned into an ordinary SSLP again. The reason is that every variable X which computes a context $u * v$ can be split into two component variables X_1 computing u and X_2 computing v. Furthermore, all operations on context variables can be simulated in an SSLP on these component variables. For example to translate the insertion of a context into another one, say $X = \alpha(Y, Z)$, we define $X_1 = Y_1 Z_1$ and $X_2 = Z_2 Y_2$.

At this point the reader might ask for the purpose of this detour via the two-sorted algebra $\hat{\mathcal{A}}$. Indeed, if we would only be interested in the free monoid $\mathcal{A} = \Sigma^*$ and SSLPs there would be no reason to do this detour. The main advantage of our detour is that it can be generalized to a large class of algebraic structures $\mathcal{A}$. It enables us to balance SLPs (or, equivalently, circuits) over $\mathcal{A}$ using a general construction and thereby avoiding an adhoc approach for each individual structure $\mathcal{A}$. As for the case of the free monoid, our general construction proceeds in two steps.

(1) First we constructed from the input SLP over $\mathcal{A}$ an equivalent logarithmic depth linear sized SLP over an extended algebra $\hat{\mathcal{A}}$. It turns out that this step can be carried out for every (multi-sorted) algebra $\mathcal{A}$. For this we define $\hat{\mathcal{A}}$ as the extension of $\mathcal{A}$ by so called *unary linear term functions* (unary linear term function over the free monoid Σ^* are exactly functions of the form $x \mapsto uxv$).

(2) In the second step we have to transform the SLP over $\hat{\mathcal{A}}$ from the first step back into an SLP over $\mathcal{A}$. Thereby the size and depth of the SLP should only increase by a constant factor. This second step only works for algebras with a *finite subsumption base*. Roughly speaking, this means that all unary linear term functions over $\mathcal{A}$ can be described by a finite number of parameterized term functions. In the example of the free monoid, this parameterized form is $x \mapsto y_1 x y_2$ where y_1, y_2 are the parameters that can be substituted by strings.

Outline. Part II of this paper is structured as follows: In Sections 3.1.1–3.1.6 we start with definitions on (multi-sorted) algebras, terms, and straight-line programs over algebras. In Sections 3.1.7 and 3.2 we prove (a reformulation of) Theorem 1.4. Finally in Section 3.3 and 3.4 we apply Theorem 1.4 to forest straight-line programs and top dags, which yields Theorem 1.3 from the introduction.

3.1 Algebras and their straight line programs

3.1.1 Ranked trees. Let us fix a finite set $\mathcal{S}$ of *sorts*. Later, we will assign to each sort $i \in \mathcal{S}$ a set A_i (of elements of sort i). Formally, an *$\mathcal{S}$-sorted signature* is a set of symbols Γ and a mapping $\text{type} \colon \Gamma \to \mathcal{S}^+$ that assigns to each symbol from Γ a non-empty word over the alphabet $\mathcal{S}$. For each

$f \in \Gamma$ the number $|\mathrm{type}(f)| - 1 \geq 0$ is also called the *rank* of f. Let $\Gamma_i \subseteq \Gamma$ ($i \geq 0$) be the set of all symbols in Γ of rank i.

In most cases, i.e., in proofs and definitions, the type mapping is either clear from the context or unimportant, therefore it is suppressed and the sorted signature (Γ, type) is simply denoted by Γ, which is called a sorted signature.

Let us also fix a second (infinite) $\mathcal{S}$-sorted signature $\mathcal{X}$, where every $x \in \mathcal{X}$ has rank zero; so formally: the sorted signature is $(\mathcal{X}, \mathrm{type})$ and $\mathrm{type}(x)$ is an element of $\mathcal{S}$ for each $x \in \mathcal{X}$. Elements of $\mathcal{X}$ are called *variables*. For $p \in \mathcal{S}$ let $\mathcal{X}_p = \{x \mid \mathrm{type}(x) = p\}$. We assume that every set $\mathcal{X}_p$ is infinite. We will always work with a finite subset $\mathcal{Y}$ of $\mathcal{X}$. Take such a set $\mathcal{Y}$. For each sort $p \in \mathcal{S}$ we define the set of *terms* $\mathcal{T}_p(\Gamma, \mathcal{Y})$ of sort p by simultaneous induction as the smallest set such that the following holds:

- Every $x \in \mathcal{X}_p \cap \mathcal{Y}$ belongs to $\mathcal{T}_p(\Gamma, \mathcal{Y})$.
- If $f \in \Gamma_n$ with $\mathrm{type}(f) = p_1 \cdots p_n q$ and $t_i \in \mathcal{T}_{p_i}(\Gamma, \mathcal{Y})$ for $1 \leq i \leq n$, then $f(t_1, t_2, \ldots, t_n) \in \mathcal{T}_q(\Gamma, \mathcal{Y})$.

We write $\mathcal{T}_p(\Gamma)$ for $\mathcal{T}_p(\Gamma, \emptyset)$, and call its elements *ground terms* (of sort p). Note that if $a \in \Gamma_0$ and $\mathrm{type}(a) = p \in \mathcal{S}$ then $a() \in \mathcal{T}_p(\Gamma)$. In this case, we write a for $a()$ and call a a constant of sort p. Let $\mathcal{T}(\Gamma, \mathcal{Y}) = \bigcup_{p \in \mathcal{S}} \mathcal{T}_p(\Gamma, \mathcal{Y})$.

Elements of $\mathcal{T}(\Gamma, \mathcal{Y})$ can be viewed as node labeled trees, where leaves are labeled with symbols from $\Gamma_0 \cup \mathcal{Y}$ and every internal node is labeled with a symbol from some Γ_n with $n \geq 1$: The root of the tree corresponding to the term $f(t_1, t_2, \ldots, t_n)$ is labeled with f and its direct subtrees are the trees corresponding to $t_1, \ldots, t_n$. Note that the composition of two functions $f \colon A \to B$ and $g \colon B \to C$ is denoted by $g \circ f$, in particular we first apply f followed by g.

For a term t we define the size $|t|$ of t as the number of edges of the corresponding tree. Equivalently, $|t|$ is inductively defined as follows: If $t = x$ is a variable, then $|t| = 0$. If $t = f(t_1, t_2, \ldots, t_n)$ for $f \in \Gamma$, then $|t| = n + \sum_{i=1}^{n} |t_i|$. The *depth* of a term t is denoted by $\mathrm{depth}(t)$ and defined inductively as usual: If $t = x$ is a variable, then $\mathrm{depth}(t) = 0$. If $t = f(t_1, t_2, \ldots, t_n)$ for $f \in \Gamma$, then $\mathrm{depth}(t) = \max\{1 + \mathrm{depth}(t_i) \mid 1 \leq i \leq n\}$ with $\max \emptyset = 0$.

Definition 3.1 (substitutions). A *substitution* is a mapping $\eta \colon \mathcal{Y} \to \mathcal{T}(\Gamma, \mathcal{Z})$ for finite (not necessarily disjoint) subsets $\mathcal{Y}, \mathcal{Z} \subseteq \mathcal{X}$ such that $y \in \mathcal{Y} \cap \mathcal{X}_p$ implies $\eta(y) \in \mathcal{T}_p(\Gamma, \mathcal{Z})$. If $\mathcal{Z} = \emptyset$, we speak of a *ground substitution*. For $t \in \mathcal{T}(\Gamma, \mathcal{Y})$ we define the term $\eta(t)$ by replacing simultaneously all occurrences of variables in t by their images under η. Formally we extend $\eta \colon \mathcal{Y} \to \mathcal{T}(\Gamma, \mathcal{Z})$ to a mapping $\eta \colon \mathcal{T}(\Gamma, \mathcal{Y}) \to \mathcal{T}(\Gamma, \mathcal{Z})$ by $\eta(f(t_1, \ldots, t_n)) = f(\eta(t_1), \ldots, \eta(t_n))$ (in particular, $\eta(a) = a$ for $a \in \Gamma_0$). A *variable renaming* is a bijective substitution $\eta \colon \mathcal{Y} \to \mathcal{Z}$ for finite variable sets $\mathcal{Y}$ and $\mathcal{Z}$ of the same size.

Definition 3.2 (contexts). Let $p, q \in \mathcal{S}$. We define the set of *contexts* $C_{pq}(\Gamma, \mathcal{Y})$ as the set of all terms $t \in \mathcal{T}_q(\Gamma, \mathcal{Y} \cup \{x\})$, where $x \in \mathcal{X}_p \setminus \mathcal{Y}$ is a fresh variable such that (i) $t \neq x$, (ii) and x occurs exactly once in t. We call x the *main variable* of t and $\mathcal{Y}$ the set of auxiliary variables of t.[1] We write $C_{pq}(\Gamma)$ for $C_{pq}(\Gamma, \emptyset)$. Elements of $C_{pq}(\Gamma)$ are called *ground contexts*. Let $C(\Gamma, \mathcal{Y}) = \bigcup_{p,q \in \mathcal{S}} C_{pq}(\Gamma, \mathcal{Y})$ and $C(\Gamma) = C(\Gamma, \emptyset)$. For $s \in C_{qr}(\Gamma, \mathcal{Y})$ and $t \in \mathcal{T}_q(\Gamma, \mathcal{Z})$ (or $t \in C_{pq}(\Gamma, \mathcal{Z})$) we define $s[t] \in \mathcal{T}_r(\Gamma, \mathcal{Y} \cup \mathcal{Z})$ ($s[t] \in C_{pr}(\Gamma, \mathcal{Y} \cup \mathcal{Z})$) as the result of replacing the unique occurrence of the main variable in s by t. Formally, we can define $s[t]$ as $\eta(s)$ where η is the substitution with domain $\{x\}$ and $\eta(x) = t$, where x is the main variable of s. An *atomic context* is a context of the form $f(y_1, \ldots, y_{k-1}, x, y_{k+1}, \ldots, y_k)$ where x is the main variable and the y_i are the auxiliary variables (we can have $y_i = y_j$ for $i \neq j$). Note that there are only finitely many atomic contexts up to renaming of variables.

[1]Since also $\mathcal{Y}$ may contain a variable y that occurs exactly once in t, we explicitly have to declare a variable as the main variable. Most of the times, the main variable will be denoted with x.

3.1.2 Algebras. We will produce strings, trees and forests by ground terms (also called algebraic expressions in this context) over certain (multi-sorted) algebras. These expressions will be compressed by directed acyclic graphs. In this section, we introduce the generic framework, which will be reinstantiated several times later on.

Fix a finite $\mathcal{S}$-sorted signature Γ. A Γ-*algebra* is a tuple $\mathcal{A} = ((A_p)_{p\in\mathcal{S}}, (f^{\mathcal{A}})_{f\in\Gamma})$ where every A_p is a non-empty set (the universe of sort p or the set of elements of sort p) and for every $f \in \Gamma_n$ with $\mathrm{type}(f) = p_1 p_2 \cdots p_n q$, $f^{\mathcal{A}} \colon \prod_{1\le j\le n} A_{p_j} \to A_q$ is an n-ary function. We also say that Γ is the *signature* of $\mathcal{A}$. In our settings, the sets A_p will be always pairwise disjoint, but formally we do not need this. Quite often, we will identify the function $f^{\mathcal{A}}$ with the symbol f. Functions of arity zero are elements of some A_p. A ground term $t \in \mathcal{T}_p(\Gamma)$ can be viewed as an algebraic expression over $\mathcal{A}$ that evaluates to an element $t^{\mathcal{A}} \in A_p$ in the natural way. For $x \in \bigcup_{p\in\mathcal{S}} A_p$ we also write $x \in \mathcal{A}$ and for A_p we also write $\mathcal{A}_p$.

When we define a Γ-algebra, we usually will not specify the types of the symbols in Γ. Instead, we just list the sets A_p ($p \in \mathcal{S}$) and the functions $f^{\mathcal{A}}$ ($f \in \Gamma$) including their domains. The latter implicitly determine the types of the symbols in Γ.

Example 3.3. A well known example of a multi-sorted algebra is a vector space. More precisely, it can be formalized as a Γ-algebra, where $\Gamma = \{\overline{0}, 0, 1, \oplus, \odot, +, \cdot\}$ is a $\mathcal{S}$-sorted signature for $\mathcal{S} = \{v, s\}$. Here v stands for "vectors" and s stands for "scalars". The types of the symbols in Γ are defined as follows:

- $\mathrm{type}(\overline{0}) = v$ (the zero vector),
- $\mathrm{type}(0) = s$ (the 0-element of the scalar field),
- $\mathrm{type}(1) = s$ (the 1-element of the scalar field),
- $\mathrm{type}(\oplus) = vvv$ (vector addition),
- $\mathrm{type}(\odot) = svv$ (multiplication of a scalar by a vector),
- $\mathrm{type}(+) = sss$ (addition in the field of scalars),
- $\mathrm{type}(\cdot) = sss$ (multiplication in the field of scalars).

Note that we cannot define non-trivial vectors by ground terms. For this, we should add some constants of type v to the signature. For the vector space F^n for a field F we might for instance add the constants $e_1, \dots, e_n$, where e_i denotes the i-th unit vector in the standard basis.

From the sets $\mathcal{T}_p(\Gamma)$ one can construct the *free term algebra*

$$\mathcal{T}(\Gamma) = ((\mathcal{T}_p(\Gamma))_{p\in\mathcal{S}}, (f)_{f\in\Gamma}),$$

where every ground term evaluates to itself. For every Γ-algebra $\mathcal{A}$, the mapping $t \mapsto t^{\mathcal{A}}$ ($t \in \mathcal{T}(\Gamma)$) is a homomorphism from the free term algebra to $\mathcal{A}$. We need the technical assumption that this homomorphism is surjective, i.e., for every $a \in \mathcal{A}$ there exists a ground term $t \in \mathcal{T}(\Gamma)$ with $a = t^{\mathcal{A}}$. In our concrete applications this assumption will be satisfied. Moreover, one can always replace $\mathcal{A}$ by the subalgebra induced by the elements $t^{\mathcal{A}}$ (we will say more about this later).

For a Γ-algebra $\mathcal{A} = ((A_p)_{p\in\mathcal{S}}, (f^{\mathcal{A}})_{f\in\Gamma})$, a variable $x \in \mathcal{X}_p$ and $a \in A_p$, we define the $(\Gamma \cup \{x\})$-algebra $\mathcal{A}[x/a] = ((A_p)_{p\in\mathcal{S}}, (f^{\mathcal{A}[x/a]})_{f\in\Gamma\cup\{x\}})$ by $f^{\mathcal{A}[x/a]} = f^{\mathcal{A}}$ for $f \in \Gamma$ and $x^{\mathcal{A}[x/a]} = a$.

Definition 3.4 (unary linear term functions)**.** Given a Γ-algebra $\mathcal{A}$ and a ground context $t \in \mathcal{C}_{pq}(\Gamma)$ with main variable x, we define the function $t^{\mathcal{A}} \colon A_p \to A_q$ by $t^{\mathcal{A}}(a) = t^{\mathcal{A}[x/a]}$ for all $a \in A_p$. We call $t^{\mathcal{A}}$ a *unary linear term function*, ULTF for short. We write $\mathrm{lin}_{pq}(\mathcal{A})$ for the set of all ULTFs $t^{\mathcal{A}}$ with $t \in \mathcal{C}_{pq}(\Gamma)$.

Example 3.5. Consider the the vector space $\mathbb{R}^2$ in the context of Example 3.3 and let

$$t = e_1 \oplus ((1 + 1) \odot (x_v \oplus e_2)) \in \mathcal{C}_{vv}$$

(recall that v is the sort of vectors). The corresponding ULTF is the affine mapping $x \mapsto 2x \oplus (1,2)^{\mathrm{T}}$ on $\mathbb{R}^2$ ($(1,2)^{\mathrm{T}}$ is the column vector with entries 1 and 2).

As another example note that a ULTF, where the underlying algebra is a ring $\mathcal{R}$ (this is a one-sorted algebra), is nothing else than a linear polynomial over $\mathcal{R}$ in a single variable x.

3.1.3 Straight-line programs. Let Γ be any $\mathcal{S}$-sorted signature. A *straight-line program* over Γ (Γ-SLP for short) is a tuple $\mathcal{G} = (\mathcal{V}, \rho, S)$, where $\mathcal{V} \subseteq \mathcal{X}$ is a finite set of variables, $S \in \mathcal{V}$ is the *start variable* and $\rho \colon \mathcal{V} \to \mathcal{T}(\Gamma, \mathcal{V})$ is a substitution (the so called *right-hand side mapping*) such that the edge relation $E(\mathcal{G}) = \{(y,z) \in \mathcal{V} \times \mathcal{V} \mid z \text{ occurs in } \rho(y)\}$ is acyclic. This implies that there exists an $n \geq 1$ such that $\rho^n \colon \mathcal{T}(\Gamma, \mathcal{V}) \to \mathcal{T}(\Gamma)$ (the n-fold composition of ρ) is a ground substitution (we can choose $n = |\mathcal{V}|$). For this n, we write ρ^* for ρ^n. Note that $\rho^* \circ \rho = \rho \circ \rho^* = \rho^*$. The term defined by $\mathcal{G}$ is $[\![\mathcal{G}]\!] := \rho^*(S)$; it is also called the *derivation tree* of $\mathcal{G}$.

In many papers on straight-line programs, the variables of a Γ-SLP are denoted by capital letters X, Y, Z, X', etc. We follow this tradition. For a variable $X \in \mathcal{V}$ we also write $[\![X]\!]_{\mathcal{G}}$ (or $[\![X]\!]$ if $\mathcal{G}$ is clear from the context) for the ground term $\rho^*(X)$.

Let $\mathcal{A}$ be a Γ-algebra. A Γ-SLP $\mathcal{G} = (\mathcal{V}, \rho, S)$ is also called an SLP over the algebra $\mathcal{A}$. We can evaluate every variable $X \in \mathcal{V}$ to its value $\rho^*(X)^{\mathcal{A}} = [\![X]\!]^{\mathcal{A}} \in \mathcal{A}$ in $\mathcal{A}$. It is important to distinguish this value from the syntactically computed ground term $\rho^*(X)$ (which is the evaluation of X in the free term algebra). Also note that in part I of the paper, we used the notation $[\![X]\!]$ for variables of string straight-line programs, which are obtained from the above general definition by taking a free monoid Σ^* for the structure $\mathcal{A}$. In other words: a string $[\![X]\!]$ from the first part of the paper would be denoted with $[\![X]\!]^{\Sigma^*}$ in the second part of the paper. The reason for this change in notation is two-fold. First, we did not want to overload the notation in Part I (especially for readers that are only interested in the balancing result for strings); hence we decided to omit the superscripts Σ^* there. Second, in the following sections the ground terms $[\![X]\!]$ are the important objects, which justifies a short notation for them.

The term $\rho(X)$ is also called the *right-hand side* of the variable $X \in \mathcal{V}$. By adding fresh variables, we can transform every Γ-SLP in linear time into a so-called *standard* Γ*-SLP*, where all right-hand sides have the form $f(X_1, \ldots, X_n)$ for variables $X_1, \ldots, X_n$ (we can have $X_i = X_j$ for $i \neq j$). A standard Γ-SLP $\mathcal{G}$ is the same object as a DAG (directed acyclic graph) with Γ-labelled nodes: the DAG is $(\mathcal{V}, E(\mathcal{G}))$ and if $\rho(X) = f(X_1, \ldots, X_n)$ then node X is labelled with f. Since the order of the edges (X, X_i) ($1 \leq i \leq n$) is important and we may have $X_i = X_j$ for $i \neq j$ we formally replace the edge (X, X_i) by the triple (X, i, X_i). A Γ-SLP interpreted over a Γ-algebra $\mathcal{A}$ is also called an *algebraic circuit* over $\mathcal{A}$.

Consider a (possibly non-standard) Γ-SLP $\mathcal{G} = (\mathcal{V}, \rho, S)$. We define the *size* $|\mathcal{G}|$ of $\mathcal{G}$ as $\sum_{X \in \mathcal{V}} |\rho(X)|$. For a standard Γ-SLP this is the number of edges of the corresponding DAG $(\mathcal{V}, E(\mathcal{G}))$. The *depth* of $\mathcal{G}$ is defined as $\mathrm{depth}(\mathcal{G}) = \mathrm{depth}([\![\mathcal{G}]\!])$, i.e. the depth of the derivation tree of $\mathcal{G}$. For a standard Γ-SLP $\mathcal{G}$ this is the maximum length of a directed path in the DAG $(\mathcal{V}, E(\mathcal{G}))$. Our definitions of size and depth ensure that both measures do not increase when one transforms a given Γ-SLP into a standard Γ-SLP. In this paper, the sizes of the right-hand sides will be always bounded by a constant that only depends on the underlying algebra $\mathcal{A}$.

Example 3.6. Consider the following standard Γ-SLP $\mathcal{G}$, where Γ is the $\{v, s\}$-sorted signature for vector spaces from Example 3.3. The variables are $X_1, \ldots, X_7$ (variables of sort v) and $Y_1, \ldots, Y_4$ (variables of sort s), the start variable is X_1, and the mapping ρ is defined by

$$\rho(X_1) = X_2 \oplus X_6,\ \rho(X_2) = X_3 \oplus X_4,\ \rho(X_3) = Y_1 \odot X_4,\ \rho(X_4) = X_5 \oplus X_6,\ \rho(X_5) = X_7 \oplus X_7,$$

$$\rho(X_6) = e_2,\ \rho(X_7) = e_1$$

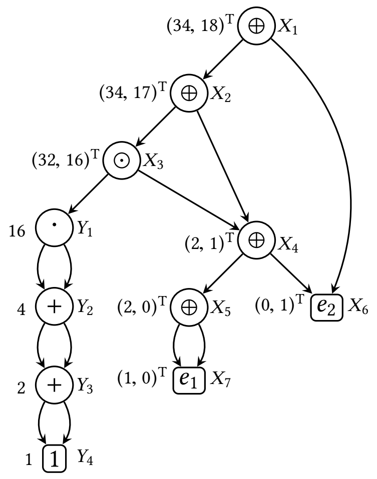

Fig. 3. A circuit over the vector space $\mathbb{R}^2$, which evaluates to the vector $(34, 18)^\mathrm{T}$.

$$\rho(Y_1) = Y_2 \cdot Y_2,\ \rho(Y_2) = Y_3 + Y_3,\ \rho(Y_3) = Y_4 + Y_4,\ \rho(Y_4) = 1.$$

Figure 3 shows the algebraic circuit that corresponds to $\mathcal{G}$. We can evaluate the SLP over the vector space $\mathcal{A} = \mathbb{R}^2$ (a particular Γ-algebra, see also Examples 3.5). The resulting values (either vectors or scalars) of the variables are written next to the corresponding gates in Figure 3. We have $|\mathcal{G}| = 16$ (there are 16 edges in Figure 3) and $\mathrm{depth}(\mathcal{G}) = 6$.

3.1.4 *Functional extensions.*

An important concept in this paper is a functional extension $\hat{\mathcal{T}}(\Gamma)$ of the free term algebra $\mathcal{T}(\Gamma)$. We define an algebra $\hat{\mathcal{T}}(\Gamma)$ over an $\mathcal{S} \cup \mathcal{S}^2$-sorted signature $\hat{\Gamma}$.

Definition 3.7 (Signature $\hat{\Gamma}$)**.** Let Γ be a $\mathcal{S}$-sorted signature. The $\mathcal{S} \cup \mathcal{S}^2$-sorted signature $\hat{\Gamma}$ is

$$\hat{\Gamma} = \Gamma \uplus \bigcup_{n \geq 1} \{\hat{f}_i \mid f \in \Gamma_n, 1 \leq i \leq n\} \uplus \{\gamma_{pqr} \mid p, q, r \in \mathcal{S}\} \uplus \{\alpha_{pq} \mid p, q \in \mathcal{S}\} \tag{7}$$

where the type function is defined as follows:

- Symbols from Γ have the same types in $\hat{\Gamma}$.
- If $\mathrm{type}(f) = p_1 \cdots p_n q$ then $\mathrm{type}(\hat{f}_i) = p_1 \cdots p_{i-1} p_{i+1} \cdots p_n q$.
- For all $p, q, r \in \mathcal{S}$ we set $\mathrm{type}(\gamma_{pqr}) = (p, q)(q, r)(p, r)$.
- For all $p, q \in \mathcal{S}$ we set $\mathrm{type}(\alpha_{pq}) = p(p, q)q$.

Definition 3.8 ($\hat{\Gamma}$-algebra $\hat{\mathcal{T}}(\Gamma)$)**.** The $\hat{\Gamma}$-algebra $\hat{\mathcal{T}}(\Gamma) = ((A_s)_{s \in \mathcal{S} \cup \mathcal{S}^2}, (f^{\hat{\mathcal{T}}(\Gamma)})_{f \in \hat{\Gamma}})$ is defined as follows: the sets A_p and A_{pq} for $p, q \in \mathcal{S}$ are defined as

- $A_p = \mathcal{T}_p(\Gamma)$ and
- $A_{pq} = \mathcal{C}_{pq}(\Gamma)$.

The operations $g^{\hat{\mathcal{T}}(\Gamma)}$ ($g \in \hat{\Gamma}$) are defined as follows, where we write g instead of $g^{\hat{\mathcal{T}}(\Gamma)}$:

- For every symbol $f \in \Gamma_n$ the algebra $\hat{\mathcal{T}}(\Gamma)$ inherits the function $f^{\mathcal{T}(\Gamma)}$ from $\mathcal{T}(\Gamma)$.
- For every symbol $f \in \Gamma_n$ with $\mathrm{type}(f) = p_1 \cdots p_n q$ ($n \geq 1$) and every $1 \leq k \leq n$ we define the $(n-1)$-ary operation

$$\hat{f}_k \colon \prod_{\substack{1 \leq i \leq n \\ i \neq k}} \mathcal{T}_{p_i}(\Gamma) \to \mathcal{C}_{p_k q}(\Gamma)$$

by $\hat{f}_k(t_1, \dots, t_{k-1}, t_{k+1}, \dots, t_n) = f(t_1, \dots, t_{k-1}, x, t_{k+1}, \dots, t_n)$ for all $t_i \in \mathcal{T}_{p_i}(\Gamma)$ $(1 \le i \le n, i \ne k)$.

- For all $p, q, r \in \mathcal{S}$ the binary operation $\gamma_{pqr} \colon C_{pq}(\Gamma) \times C_{qr}(\Gamma) \to C_{pr}(\Gamma)$ is defined by $\gamma_{pqr}(t, s) = s[t]$.
- For all $p, q \in \mathcal{S}$ the binary operation $\alpha_{pq} \colon \mathcal{T}_p(\Gamma) \times C_{pq}(\Gamma) \to \mathcal{T}_q(\Gamma)$ is defined by $\alpha_{pq}(t, s) = s[t]$.

The definition of the operations α_{pq} and γ_{pqr} suggests to write $s[t]$ instead of $\alpha_{pq}(t, s)$ or $\gamma_{pq}(t, s)$, which we will do most of the times.

Recall the definition of unary linear term functions (ULTFs) from Definition 3.4. An *atomic ULTF* is of the form $z \mapsto f^{\mathcal{A}}(a_1, \dots, a_{k-1}, z, a_{k+1}, \dots, a_n)$ for $f \in \Gamma_n$ with $\mathrm{type}(f) = p_1 \cdots p_n q$ and $a_i \in A_{p_i}$ for $(1 \le i \le n, i \ne k)$. We denote this function with $f^{\mathcal{A}}(a_1, \dots, a_{k-1}, \cdot, a_{k+1}, \dots, a_n)$ in the following. At this point, we use the assumption that every element of $\mathcal{A}$ can be written as $t^{\mathcal{A}}$ for a ground term t. Hence, the elements a_i are defined by terms, which ensures that $f^{\mathcal{A}}(a_1, \dots, a_{k-1}, \cdot, a_{k+1}, \dots, a_n)$ is indeed a ULTF. It is easy to see that every ULTF is the composition of finitely many atomic ULTFs.

Definition 3.9 ($\hat{\Gamma}$-algebra $\hat{\mathcal{A}}$). Given a Γ-algebra $\mathcal{A} = ((A_p)_{p \in \mathcal{S}}, (f^{\mathcal{A}})_{f \in \Gamma})$ we define the $\hat{\Gamma}$-algebra $\hat{\mathcal{A}} = ((B_s)_{s \in \mathcal{S} \cup \mathcal{S}^2}, (f^{\hat{\mathcal{A}}})_{f \in \hat{\Gamma}})$ as follows: The sets B_p and B_{pq} for $p, q \in \mathcal{S}$ are defined as:

- $B_p = A_p$ and
- $B_{pq} = \mathrm{lin}_{pq}(\mathcal{A})$.

The operations $g^{\hat{\mathcal{A}}}$ $(g \in \hat{\Gamma})$ are defined as follows, where we write g instead of $g^{\hat{\mathcal{A}}}$.

- Every $f \in \Gamma$ is interpreted as $f^{\hat{\mathcal{A}}} = f^{\mathcal{A}}$.
- For every symbol $f \in \Gamma_n$ with $\mathrm{type}(f) = p_1 \cdots p_n q$ $(n \ge 1)$ and every $1 \le k \le n$ we define the $(n-1)$-ary operation
$$\hat{f}_k : \prod_{\substack{1 \le i \le n \\ i \ne k}} A_{p_i} \to \mathrm{lin}_{p_k q}(\mathcal{A})$$
by $\hat{f}_k(a_1, \dots, a_{k-1}, a_{k+1}, \dots, a_n) = f^{\mathcal{A}}(a_1, \dots, a_{k-1}, \cdot, a_{k+1}, \dots, a_n)$ for all $a_i \in A_{p_i}$ $(1 \le i \le n, i \ne k)$.
- For all $p, q, r \in \mathcal{S}$ the binary operation $\gamma_{pqr} \colon \mathrm{lin}_{pq}(\mathcal{A}) \times \mathrm{lin}_{qr}(\mathcal{A}) \to \mathrm{lin}_{pr}(\mathcal{A})$ is defined as function composition: $\gamma_{pqr}(g, h) = h \circ g$.
- For all $p, q \in \mathcal{S}$ the binary operation $\alpha_{pq} \colon A_p \times \mathrm{lin}_{pq}(\mathcal{A}) \to A_q$ is defined as function application: $\alpha_{pq}(a, g) = g(a)$.

Note that Definitions 3.8 and 3.9 are consistent in the following sense: If we apply the construction from Definition 3.9 for $\mathcal{A} = \mathcal{T}(\Gamma)$ (the free term algebra) then we obtain an isomorphic copy of the algebra $\hat{\mathcal{T}}(\Gamma)$ from Definition 3.8, i.e., $\widehat{\mathcal{T}(\Gamma)} \cong \hat{\mathcal{T}}(\Gamma)$. Moreover, the mappings $t \mapsto t^{\mathcal{A}}$ (for ground terms t) and $c \mapsto c^{\mathcal{A}}$ (for ground contexts c) yield a canonical surjective morphism from $\hat{\mathcal{T}}(\Gamma)$ to $\hat{\mathcal{A}}$ that extends the canonical morphism from the free term algebra $\mathcal{T}(\Gamma)$ to $\mathcal{A}$.

Example 3.10. If $\mathcal{A} = (\Sigma^*, \cdot)$ is the free monoid over the finite alphabet Σ (i.e., $\cdot$ is the concatenation function: $u \cdot v = uv$), then $\hat{\mathcal{A}}$ is the extended algebra explained in the example at the beginning of Section 3: The unary function $\hat{\cdot}_1$ (respectively, $\hat{\cdot}_2$) turns a string $w \in \Sigma^*$ into the unary function $x \mapsto xw$ (respectively, $x \mapsto wx$). The function γ inserts into a unary function another unary function, and α inserts into a unary function a string.

3.1.5 Tree straight-line programs. Recall the definition of the $\mathcal{S} \cup \mathcal{S}^2$-sorted signature $\hat{\Gamma}$ in (7). A $\hat{\Gamma}$-SLP $\mathcal{G}$ which evaluates in the $\hat{\Gamma}$-algebra $\hat{\mathcal{T}}(\Gamma)$ to a ground term (i.e., $[\![\mathcal{G}]\!]^{\hat{\mathcal{T}}(\Gamma)} \in \mathcal{T}(\Gamma)$) is also called a *tree straight-line program* over Γ (Γ-TSLP for short) [15, 17, 31].

Recall that $\hat{\Gamma}$ contains for every $f \in \Gamma_n$ with $n \geq 1$ the unary symbols $\hat{f}_k$ ($1 \leq k \leq n$). Right-hand sides of the form $\hat{f}_k(X_1, \ldots, X_{k-1}, X_{k+1}, \ldots, X_n)$ in a Γ-TSLP are written for better readability as $f(X_1, \ldots, X_{k-1}, x, X_{k+1}, \ldots, X_n)$. This is also the notation used in [15, 17, 31]. For right-hand sides of the form $\alpha_{pq}(X, Y)$ or $\gamma_{pqr}(X, Y)$ we write $X[Y]$.

Example 3.11. Let us assume that $\mathcal{S}$ consists of a single sort. Consider the Γ-TSLP

$$\mathcal{G} = (\{S, X_1, \ldots, X_7\}, \rho, S)$$

with $\Gamma_2 = \{f, g\}$, $\Gamma_0 = \{a, b\}$ and $\rho(S) = X_1[X_2]$, $\rho(X_1) = X_3[X_3]$, $\rho(X_2) = X_4[X_5]$, $\rho(X_3) = f(x, X_7)$, $\rho(X_4) = X_6[X_6]$, $\rho(X_5) = a$, $\rho(X_6) = g(X_7, x)$, $\rho(X_7) = b$. We get

- $[\![X_6]\!]^{\hat{\mathcal{T}}(\Gamma)} = \rho^*(X_6)^{\hat{\mathcal{T}}(\Gamma)} = g(b, x)$,
- $[\![X_4]\!]^{\hat{\mathcal{T}}(\Gamma)} = \rho^*(X_4)^{\hat{\mathcal{T}}(\Gamma)} = g(b, x)[g(b, x)] = g(b, g(b, x))$,
- $[\![X_3]\!]^{\hat{\mathcal{T}}(\Gamma)} = \rho^*(X_3)^{\hat{\mathcal{T}}(\Gamma)} = f(x, b)$,
- $[\![X_2]\!]^{\hat{\mathcal{T}}(\Gamma)} = \rho^*(X_2)^{\hat{\mathcal{T}}(\Gamma)} = g(b, g(b, x))[a] = g(b, g(b, a))$,
- $[\![X_1]\!]^{\hat{\mathcal{T}}(\Gamma)} = \rho^*(X_1)^{\hat{\mathcal{T}}(\Gamma)} = f(x, b)[f(x, b)] = f(f(x, b), b)$, and
- $[\![\mathcal{G}]\!]^{\hat{\mathcal{T}}(\Gamma)} = \rho^*(S)^{\hat{\mathcal{T}}(\Gamma)} = f(f(x, b), b)[g(b, g(b, a))] = f(f(g(b, g(b, a)), b), b)$.

3.1.6 From TSLPs to SLPs. Fix a Γ-algebra $\mathcal{A}$. Our first goal is to transform a Γ-TSLP $\mathcal{G}$ into a Γ-SLP $\mathcal{H}$ of size $O(|\mathcal{G}|)$ and depth $O(\text{depth}(\mathcal{G}))$ such that $[\![\mathcal{H}]\!]^{\mathcal{A}} = [\![\mathcal{G}]\!]^{\hat{\mathcal{A}}}$. For this, we have to restrict the class of Γ-algebras. For instance, for the free term algebra the above transformation cannot be achieved in general: the chain tree $t_n = f(f(f(\cdots f(a) \cdots)))$ with 2^n occurrences of f can be easily produced by a $\{a, f\}$-TSLP of size $O(n)$ but the only DAG (= SLP over the free term algebra $\mathcal{T}(\{a, f\})$) for t_n is t_n itself. We restrict ourselves to algebras with a finite subsumption base, as defined below. Such algebras have been implicitly used in our recent papers [15, 17].

Definition 3.12 (equivalence and subsumption preorder in $\mathcal{A}$)**.** For contexts $s, t \in C_{pq}(\Gamma, \mathcal{Y})$ we say that s and t are *equivalent* in $\mathcal{A}$ if for every ground substitution $\eta \colon \mathcal{Y} \to \mathcal{T}(\Gamma)$ we have $\eta(s)^{\mathcal{A}} = \eta(t)^{\mathcal{A}}$ (which is an ULTF).

For contexts $s \in C_{pq}(\Gamma, \mathcal{Y})$ and $t \in C_{pq}(\Gamma, \mathcal{Z})$ we say that t *subsumes* s in $\mathcal{A}$ or that s is subsumed by t in $\mathcal{A}$ ($t \leq^{\mathcal{A}} s$ for short) if there exists a substitution $\zeta \colon \mathcal{Z} \to \mathcal{T}(\Gamma, \mathcal{Y})$ such that s and $\zeta(t)$ are equivalent in $\mathcal{A}$.

A *subsumption base* of $\mathcal{A}$ is a set of (not necessarily ground) contexts C such that for every context s there exists a context $t \in C$ with $t \leq^{\mathcal{A}} s$.

It is easy to see that $\leq^{\mathcal{A}}$ is reflexive and transitive but in general not antisymmetric. Moreover, the relation $\leq^{\mathcal{A}}$ satisfies the following monotonicity property:

Lemma 3.13. *Let $s \in C_{qr}(\Gamma, \mathcal{Y})$, $t_1 \in C_{pq}(\Gamma, \mathcal{Z}_1)$ and $t_2 \in C_{pq}(\Gamma, \mathcal{Z}_2)$ be contexts such that $\mathcal{Y} \cap \mathcal{Z}_1 = \emptyset$ and $\mathcal{Y} \cup \mathcal{Z}_1 \cup \mathcal{Z}_2$ contains none of the main variables of s, t_1, t_2. If $t_1 \leq^{\mathcal{A}} t_2$ then $s[t_1] \leq^{\mathcal{A}} s[t_2]$.*

Proof. Since t_1 subsumes t_2 in $\mathcal{A}$ there exists a substitution $\zeta \colon \mathcal{Z}_1 \to \mathcal{T}(\Gamma, \mathcal{Z}_2)$ such that for every ground substitution $\eta \colon \mathcal{Z}_2 \to \mathcal{T}(\Gamma)$ we have

$$\eta(t_2)^{\mathcal{A}} = \eta(\zeta(t_1))^{\mathcal{A}}.$$

Define the substitution $\zeta' \colon \mathcal{Y} \cup \mathcal{Z}_1 \to \mathcal{T}(\Gamma, \mathcal{Y} \cup \mathcal{Z}_2)$ by

$$\zeta'(y) = \begin{cases} \zeta(y) & \text{if } y \in \mathcal{Z}_1, \\ y & \text{if } y \in \mathcal{Y}. \end{cases}$$

As $\mathcal{Z}_1 \cap \mathcal{Y} = \emptyset$ by the assumption, ζ' is well defined. It satisfies $\zeta'(t_1) = \zeta(t_1)$ and $\zeta'(s) = s$. For any ground substitution $\eta \colon \mathcal{Y} \cup \mathcal{Z}_2 \to \mathcal{T}(\Gamma)$ we have:

$$\begin{aligned} \eta(s[t_2])^{\mathcal{A}} &= (\eta(s)[\eta(t_2)])^{\mathcal{A}} \\ &= \eta(s)^{\mathcal{A}} \circ \eta(t_2)^{\mathcal{A}} \\ &= \eta(s)^{\mathcal{A}} \circ \eta(\zeta(t_1))^{\mathcal{A}} \\ &= \eta(\zeta'(s))^{\mathcal{A}} \circ \eta(\zeta'(t_1))^{\mathcal{A}} \\ &= (\eta(\zeta'(s))[\eta(\zeta'(t_1))])^{\mathcal{A}} \\ &= \eta(\zeta'(s[t_1]))^{\mathcal{A}}. \end{aligned}$$

This implies $s[t_1] \leq^{\mathcal{A}} s[t_2]$. □

We will be interested in algebras that have a finite subsumption base. In order to show that a set C is a finite subsumption base we will use the following lemma.

Lemma 3.14. *Let $\mathcal{A}$ be a Γ-algebra and let C be a finite set of contexts with the following properties:*

- *For every atomic context s there exists $t \in C$ with $t \leq^{\mathcal{A}} s$.*
- *For every atomic context s and every $t \in C$ such that $s[t]$ is defined and s and t do not share auxiliary variables, there exists $t' \in C$ with $t' \leq^{\mathcal{A}} s[t]$.*

Then C is a subsumption base.

Proof. Assume that the two conditions from the lemma hold. We show by induction on s that for every context s there exists a context $t \in C$ with $t \leq^{\mathcal{A}} s$.

If $s = f(s_1, \dots, s_{i-1}, x, s_{i+1}, \dots, s_n)$ for some terms $s_1, \dots, s_{i-1}, s_{i+1}, \dots, s_n$ then s is subsumed in $\mathcal{A}$ by the atomic context $f(y_1, \dots, y_{i-1}, x, y_{i+1}, \dots, y_n)$, which in turn is subsumed in $\mathcal{A}$ by some $t \in C$. If $s = f(s_1, \dots, s_{i-1}, s', s_{i+1}, \dots, s_n)$ for some terms $s_1, \dots, s_n$ and some context s' then $f(y_1, \dots, y_{i-1}, s', y_{i+1}, \dots, y_n) \leq^{\mathcal{A}} s$ for fresh auxiliary variables $y_1, \dots, y_{i-1}, y_{i+1}, \dots, y_n$ (that neither occur in s' nor any context from C). By induction there exists $t' \in C$ with $t' \leq^{\mathcal{A}} s'$. By Lemma 3.13 we have $f(y_1, \dots, y_{i-1}, t', y_{i+1}, \dots, y_n) \leq^{\mathcal{A}} f(y_1, \dots, y_{i-1}, s', y_{i+1}, \dots, y_n)$. By the second assumption from the lemma, we have that $t'' \leq^{\mathcal{A}} f(y_1, \dots, y_{i-1}, t', y_{i+1}, \dots, y_n)$ for some $t'' \in C$. We get $t'' \leq^{\mathcal{A}} s$ by transitivity of $\leq^{\mathcal{A}}$. □

Remark 3.15. Recall that we made the technical assumption that every element a of $\mathcal{A}$ can be written as $t^{\mathcal{A}}$ for a ground term $\mathcal{A}$. Let $\mathcal{B}$ be the subalgebra of $\mathcal{A}$ that is induced by all elements $t^{\mathcal{A}}$ for $t \in \mathcal{T}(\mathcal{A})$. It is obvious that every subsumption base of $\mathcal{A}$ is also a subsumption base of $\mathcal{B}$.

Example 3.16. Every semiring $\mathcal{A} = (A, +, \times, a_1, \dots, a_n)$, where $a_1, \dots, a_n \in A$ are arbitrary constants, has a finite subsumption base. Here we do not assume that $\times$ is commutative, nor do we assume that identity elements with respect to $+$ or $\times$ exist. In other words: $(A, +)$ is a commutative semigroup, $(A, \times)$ is a semigroup and the left and right distributive law holds. The finite subsumption base $C(\mathcal{A})$ consists of the following contexts $axb + c, ax + c, xb + c, x + c, axb, ax, xb$, and x, where x is the main variable and a, b, c are auxiliary variables. We write ab instead of $a \times b$ and omit in axb brackets that are not needed due to the associativity of multiplication. To see that every context s is subsumed in $\mathcal{A}$ by one of the contexts from $C(\mathcal{A})$, observe that a context defines a linear polynomial in the main variable x. Hence, every context is equivalent in $\mathcal{A}$ to a context of the form $sxt + u, sx + u, xt + u, x + u, sxt, sx, xt$ or x, where s, t, u are terms that contain the auxiliary parameters. Each of these contexts is subsumed by a context from $C(\mathcal{A})$ by the substitution ζ with $\zeta(a) = s$, $\zeta(b) = t$, and $\zeta(c) = u$.

Let us remark that the above proof can be adapted to the situation that also $+$ is not commutative. In that case, we have include the terms $c' + axb + c$, $c' + ax + c$, $c' + xb + c$, $c' + x + c$, $c' + axb$, $c' + ax$, $c' + xb$ and $c' + x$ to the set $C(\mathcal{A})$.

On the other hand, if $(A, +)$ has a neutral element, called 0 in the following, then $axb + c, ax + c, xb + c, x + c$ is s subsumption base. To see this observe that, for instance, axb is subsumed by $axb + c$, which is shown by the substitution $c \mapsto 0$.

Example 3.17. If Γ contains a symbol of rank at least one, then the free term algebra $\mathcal{T}(\Gamma)$ has no finite subsumption base: If C were a finite subsumption base of $\mathcal{T}(\Gamma)$, then every ground context could be obtained from some $t \in C$ by replacing the auxiliary parameters in t by ground terms. But this replacement does not change the length of the path from the root of the context to its main variable. Hence, we would obtain a bound for the length of the path from the root to the main variable in a ground context, which clearly does not exist.

Lemma 3.18. *Assume that the Γ-algebra $\mathcal{A}$ has a finite subsumption base. Then from a given Γ-TSLP $\mathcal{G}$ one can compute in time $O(|\mathcal{G}|)$ a Γ-SLP $\mathcal{H}$ of size $O(|\mathcal{G}|)$ and depth $O(depth(\mathcal{G}))$ such that $[\![\mathcal{G}]\!]^{\hat{\mathcal{A}}} = [\![\mathcal{H}]\!]^{\mathcal{A}}$.*

Proof. Let $C(\mathcal{A})$ be a finite subsumption base for $\mathcal{A}$. We say that a context $s \in \mathcal{C}(\Gamma, \mathcal{Y})$ belongs to $C(\mathcal{A})$ *up to variable renaming* if there is a variable renaming $\theta : \mathcal{Y} \to \mathcal{Z}$ such that $\theta(s) \in C(\Gamma)$. Since the algebra $\mathcal{A}$ is fixed, the set $C(\mathcal{A})$ has size $O(1)$. Assume that s and t are contexts with the following properties: (i) $s[t]$ is defined, (ii) s and t have no common auxiliary variable, and (iii) s and t belong to $C(\mathcal{A})$ up to variable renaming. We denote with $s \cdot t$ a context from $C(\mathcal{A})$ with $s \cdot t \leq^{\mathcal{A}} s[t]$. Since s and t have size $O(1)$ ($C(\mathcal{A})$ is a fixed set of contexts), we can compute from s, t in constant time the context $s \cdot t$ and a substitution ζ such that $s[t]$ and $\zeta(s \cdot t)$ are equivalent in $\mathcal{A}$. Similarly, one can compute from a given atomic context s in constant time a context $t \in C(\mathcal{A})$ and a ground substitution ζ such that s and $\zeta(t)$ are equivalent in $\mathcal{A}$.

Let $\mathcal{G} = (\mathcal{V}, \rho, S)$. We define $\mathcal{V}_0 = \{X \in \mathcal{V} \mid \rho^*(X)^{\hat{\mathcal{T}}(\Gamma)} \in \mathcal{T}(\Gamma)\}$ and $\mathcal{V}_1 = \{X \in \mathcal{V} \mid \rho^*(X)^{\hat{\mathcal{T}}(\Gamma)} \in \mathcal{C}(\Gamma)\} = \mathcal{V} \setminus \mathcal{V}_0$. The Γ-SLP $\mathcal{H}$ to be constructed will be denoted with $\mathcal{H} = (\mathcal{V}', \tau, S)$. We will have $\mathcal{V}_0 \subseteq \mathcal{V}'$. A variable $X \in \mathcal{V}_1$ is replaced in $\mathcal{H}$ by a finite set $\mathcal{Y}_X$ of variables. Moreover, we will compute a context $t_X \in \mathcal{C}(\Gamma, \mathcal{Y}_X)$ that belongs to $C(\mathcal{A})$ up to variable renaming. We can assume that $\mathcal{Y}_X \cap \mathcal{Y}_{X'} = \emptyset = \mathcal{Y}_X \cap \mathcal{V}_0$ for all $X, X' \in \mathcal{V}_1$ with $X \neq X'$. The set of variables of $\mathcal{H}$ is then $\mathcal{V}' = \mathcal{V}_0 \cup \bigcup_{X \in \mathcal{V}_1} \mathcal{Y}_X$. Moreover, $\mathcal{H}$ will satisfy the following conditions:

(a) If $X \in \mathcal{V}_0$ then $\rho^*(X)^{\hat{\mathcal{A}}} = \tau^*(X)^{\mathcal{A}}$ (which is an element of $\mathcal{A}$).
(b) If $X \in \mathcal{V}_1$ then $\rho^*(X)^{\hat{\mathcal{A}}} = \tau^*(t_X)^{\mathcal{A}}$ (which is a ULTF on $\mathcal{A}$).

We construct $\mathcal{H}$ bottom-up. That means that we process all variables in $\mathcal{V}$ in a single pass over $\mathcal{G}$. When we process a variable $X \in \mathcal{V}$ we have already processed all variables X' that appear in $\rho(X)$. In particular, the set $\mathcal{Y}_{X'}$ and the context $t_{X'} \in \mathcal{C}(\Gamma, \mathcal{Y}_{X'})$ (in case $X' \in \mathcal{V}_1$) are defined. In addition, X' satisfies the above conditions (a) and (b).

We proceed by a case distinction according to the right-hand side $\rho(X)$ of $X \in \mathcal{V}$. This right-hand side has one of the following four forms:

Case 1. $X \in \mathcal{V}_0$ and $\rho(X) = f(X_1, \ldots, X_n)$ for $f \in \Gamma_n$ ($n \geq 0$) and $X_1, \ldots, X_n \in \mathcal{V}_0$. Then we set $\tau(X) = \rho(X)$. Clearly, the above condition (a) holds.

Case 2. $X \in \mathcal{V}_0$ and $\rho(X) = X'[X'']$ with $X' \in \mathcal{V}_1$, $X'' \in \mathcal{V}_0$. By induction we have $\rho^*(X'')^{\hat{\mathcal{A}}} = \tau^*(X'')^{\mathcal{A}}$. Moreover, we have computed a context $t_{X'} \in \mathcal{C}(\Gamma, \mathcal{Y}_{X'})$ that belongs to $C(\mathcal{A})$ up to variable renaming and such that $\rho^*(X')^{\hat{\mathcal{A}}} = \tau^*(t_{X'})^{\mathcal{A}}$. We define $\tau(X) = t_{X'}[X''] \in \mathcal{T}(\Gamma, \mathcal{Y}_{X'} \cup \{X''\})$ (that is, we replace the main variable in $t_{X'}$ by X'') and get

$$\begin{aligned}\rho^*(X)^{\hat{\mathcal{A}}} = \rho^*(X')^{\hat{\mathcal{A}}}(\rho^*(X'')^{\hat{\mathcal{A}}}) &= \tau^*(t_{X'})^{\mathcal{A}}(\tau^*(X'')^{\mathcal{A}}) \\ &= \tau^*(t_{X'}[X''])^{\mathcal{A}} = \tau^*(\tau(X))^{\mathcal{A}} = \tau^*(X)^{\mathcal{A}}.\end{aligned}$$

Case 3. $X \in \mathcal{V}_1$ and $\rho(X) = f(X_1, \dots, X_{k-1}, x, X_{k+1}, \dots, X_n)$ for $f \in \Gamma_n$ ($n \geq 1$) and $X_1, \dots, X_{k-1}, X_{k+1}, \dots, X_n \in \mathcal{V}_0$. By induction we have $\rho^*(X_i)^{\hat{\mathcal{A}}} = \tau^*(X_i)^{\mathcal{A}}$ for $1 \leq i \leq n$, $i \neq k$. We can view $\rho(X)$ as an atomic context with main variable x and auxiliary variables $X_1, \dots, X_{k-1}, X_{k+1}, \dots, X_n$. Hence, we can compute $t_X \in C(\mathcal{A})$ with $t_X \leq^{\mathcal{A}} \rho(X)$. We rename the auxiliary variables of t_X such that they do not already belong to $\mathcal{H}$. Let $\mathcal{Y}_X$ be the set of auxiliary variables of t_X. We then add all variables in $\mathcal{Y}_X$ to $\mathcal{H}$. By the definition of $\leq^{\mathcal{A}}$ there is a substitution $\zeta\colon \mathcal{Y}_X \to \mathcal{T}(\Gamma, \{X_1, \dots, X_{k-1}, X_{k+1}, \dots, X_n\})$ such that

$$\rho^*(X)^{\hat{\mathcal{A}}} = \rho^*(\rho(X))^{\hat{\mathcal{A}}} = \tau^*(\rho(X))^{\mathcal{A}} = \tau^*(\zeta(t_X))^{\mathcal{A}}.$$

We define the right-hand side for every new variable $Y \in \mathcal{Y}_X$ by $\tau(Y) = \zeta(Y)$ and get $\rho^*(X)^{\hat{\mathcal{A}}} = \tau^*(\zeta(t_X))^{\mathcal{A}} = \tau^*(\tau(t_X))^{\mathcal{A}} = \tau^*(t_X)^{\mathcal{A}}$, which is point (b).

Case 4. $X \in \mathcal{V}_1$ with $\rho_{\mathcal{G}}(X) = X'[X'']$ and $X', X'' \in \mathcal{V}_1$. We have already defined the terms $t_{X'}, t_{X''}$ that belong to $C(\mathcal{A})$ up to variable renaming. The set of auxiliary variables of $t_{X'}$ (resp., $t_{X''}$) is $\mathcal{Y}_{X'}$ (resp., $\mathcal{Y}_{X''}$) and we have $\mathcal{Y}_{X'} \cap \mathcal{Y}_{X''} = \emptyset$. Moreover, by the induction hypothesis for X' and X'' we have $\rho^*(X')^{\hat{\mathcal{A}}} = \tau^*(t_{X'})^{\mathcal{A}}$ and $\rho^*(X'')^{\hat{\mathcal{A}}} = \tau^*(t_{X''})^{\mathcal{A}}$. We set $t_X := t_{X'} \cdot t_{X''} \in C(\mathcal{A})$. We rename the auxiliary variables of t_X such that they do not already belong to $\mathcal{H}$. Let $\mathcal{Y}_X$ be the set of auxiliary variables of t_X. We then add every $Y \in \mathcal{Y}_X$ to $\mathcal{H}$. By definition of t_X we have $t_X \leq^{\mathcal{A}} t_{X'}[t_{X''}]$, which implies that there is a substitution $\zeta\colon \mathcal{Y}_X \to \mathcal{T}(\Gamma, \mathcal{Y}_{X'} \cup \mathcal{Y}_{X''})$ with

$$\begin{aligned}\rho^*(X)^{\hat{\mathcal{A}}} = \rho^*(X')^{\hat{\mathcal{A}}} \circ \rho^*(X'')^{\hat{\mathcal{A}}} &= \tau^*(t_{X'})^{\mathcal{A}} \circ \tau^*(t_{X''})^{\mathcal{A}} \\ &= \tau^*(t_{X'}[t_{X''}])^{\mathcal{A}} = \tau^*(\zeta(t_X))^{\mathcal{A}}.\end{aligned}$$

We define the right-hand side for every new variable $Y \in \mathcal{Y}_X$ by $\tau(Y) = \zeta(Y)$ and get $\rho^*(X)^{\hat{\mathcal{A}}} = \tau^*(\zeta(t_X))^{\mathcal{A}} = \tau^*(\tau(t_X))^{\mathcal{A}} = \tau^*(t_X)^{\mathcal{A}}$, which is point (b).

The running time for the construction of $\mathcal{H}$ is $O(|\mathcal{G}|)$, since for each variable $X \in \mathcal{V}$ we only spend constant time (see the remark from the first paragraph of the proof). In each step we have to take a constant number of fresh auxiliary variables. We can take them from a list $Y_1, Y_1, Y_3, \dots$ and store a pointer to the next free variable. □

It is known [15, 17] that a ranked tree t of size n can be transformed in linear time into a tree straight-line program of size $O(n/\log_\sigma n)$ and depth $O(\log n)$, where σ is the number of different node labels that appear in t. With Lemma 3.18 it follows that for every algebra $\mathcal{A}$ having a finite subsumption base one can compute in linear time from a given expression tree of size n an equivalent circuit of size $O(n/\log_\sigma n)$ and depth $O(\log n)$ (σ is a constant here, namely the number of operations of the algebra $\mathcal{A}$).

3.1.7 *Main result for* Γ*-straight line programs.*

We now state the main technical result for Γ-straight line programs. Note that for some applications we need a signature Γ that is part of the input.

Theorem 3.19. *From a given signature* Γ *and a* Γ*-SLP* $\mathcal{G}$*, which defines the tree* $t = [\![\mathcal{G}]\!] \in \mathcal{T}_0(\Gamma)$*, one can compute in time* $O(|\mathcal{G}|)$ *a* Γ*-TSLP* $\mathcal{H}$ *such that* $[\![\mathcal{H}]\!]^{\hat{\mathcal{T}}(\Gamma)} = t$*,* $|\mathcal{H}| \in O(|\mathcal{G}|)$ *and depth*$(\mathcal{H}) \in O(\log |t|)$*.*

We will prove Theorem 3.19 in Section 3.2. Together with Lemma 3.18, Theorem 3.19 yields the following result:

Theorem 3.20. *Take a fixed signature Γ and a fixed Γ-algebra $\mathcal{A}$ that has a finite subsumption base. From a given Γ-SLP $\mathcal{G}$, which defines the derivation tree $t = [\![\mathcal{G}]\!] \in \mathcal{T}_0(\Gamma)$, one can compute in time $O(|\mathcal{G}|)$ a Γ-SLP $\mathcal{H}$ such that $[\![\mathcal{H}]\!]^{\mathcal{A}} = [\![\mathcal{G}]\!]^{\mathcal{A}}$, $|\mathcal{H}| \in O(|\mathcal{G}|)$ and depth$(\mathcal{H}) \in O(\log|t|)$.*

Proof. Using Theorem 3.19 we obtain from $\mathcal{G}$ in time $O(|\mathcal{G}|)$ a Γ-TSLP $\mathcal{G}'$ such that $[\![\mathcal{G}']\!]^{\hat{\mathcal{T}}(\Gamma)} = t$, $|\mathcal{G}'| \in O(|\mathcal{G}|)$ and depth$(\mathcal{G}') \in O(\log|t|)$. From $[\![\mathcal{G}']\!]^{\hat{\mathcal{T}}(\Gamma)} = t = [\![\mathcal{G}]\!]$ we get $[\![\mathcal{G}']\!]^{\hat{\mathcal{A}}} = [\![\mathcal{G}]\!]^{\mathcal{A}}$. By Lemma 3.18 we can compute from $\mathcal{G}'$ in time $O(|\mathcal{G}'|) = O(|\mathcal{G}|)$ a Γ-SLP $\mathcal{H}$ of size $O(|\mathcal{G}'|) = O(|\mathcal{G}|)$ and depth $O(\text{depth}(\mathcal{G}')) = O(\log|t|)$ such that $[\![\mathcal{H}]\!]^{\mathcal{A}} = [\![\mathcal{G}']\!]^{\hat{\mathcal{A}}} = [\![\mathcal{G}]\!]^{\mathcal{A}}$. □

Note that Theorem 3.20 is exactly the same statement as Theorem 1.4 from the introduction (which is formulated via circuits instead of straight-line programs).

Remark 3.21. Recall that we made the technical assumption that every element a of $\mathcal{A}$ can be written as $t^{\mathcal{A}}$ for a ground term $\mathcal{A}$. We can still prove Corollary 3.20 in case $\mathcal{A}$ does not satisfy this assumption: let $\mathcal{B}$ be the subalgebra of $\mathcal{A}$ that is induced by all elements $t^{\mathcal{A}}$ for $t \in \mathcal{T}(\mathcal{A})$. By Remark 3.15, $\mathcal{B}$ has a finite subsumption base as well. Moreover, for every Γ-SLP $\mathcal{G}$ we obviously have $[\![\mathcal{G}]\!]^{\mathcal{A}} = [\![\mathcal{G}]\!]^{\mathcal{B}}$. Hence, Corollary 3.20 applied to the algebra $\mathcal{B}$ yields the statement for $\mathcal{A}$.

Remark 3.22. Theorem 3.20 only holds for a fixed Γ-algebra because Lemma 3.18 assumes a fixed Γ-algebra. Nevertheless there are settings, where we consider a family $\{\mathcal{A}_i \mid i \in I\}$ with the $\mathcal{A}_i$ being Γ_i-algebras. An example is the family of all free monoids Σ^* for a finite alphabet Σ that is part of the input. Under certain assumptions, the statement of Theorem 3.20 can be extended to the uniform setting, where the signature Γ_i ($i \in I$) is part of the input and SLPs are evaluated in the algebra $\mathcal{A}_i$. First of all we have to assume that every symbol $f \in \Gamma_i$ fits into a machine word of the underlying RAM model, which is a natural assumption if the signature Γ_i is part of the input. For the Γ_i-algebras $\mathcal{A}_i$ we need the following assumptions:

(i) There is a constant r such that the rank of every symbol $f \in \bigcup_{i\in I}\Gamma_i$ is bounded by r.
(ii) There is a constant c and a finite subsumption base $C(\mathcal{A}_i)$ for every $i \in I$ such that the size of every context $s \in \bigcup_{i\in I} C(\mathcal{A}_i)$ is bounded by c. With the above assumption on the word size of the RAM this ensures that a context $s \in \bigcup_{i\in I} C(\mathcal{A}_i)$ fits into $O(1)$ many machine words.
(iii) There is a constant time algorithm that computes from a given atomic context s over the signature Γ_i a context $t \in C(\mathcal{A}_i)$ and a substitution ζ such that $\zeta(t)$ and s are equivalent in $\mathcal{A}_i$.
(iv) There is a constant time algorithm that takes two contexts s and t over the signature Γ_i such that $s[t]$ is defined, s and t have no common auxiliary variable, and s and t belong to $C(\mathcal{A}_i)$ up to variable renaming, and computes the context $s \cdot t$ (see the first paragraph in the proof of Lemma 3.18) and a substitution ζ such that $\zeta(s \cdot t)$ and $s[t]$ are equivalent in $\mathcal{A}_i$.

Under these assumptions the construction from the proof of Lemma 3.18 can still be carried out in linear time. Since the statement of Theorem 3.19 holds for a signature Γ that is part of the input, this allows to extend Theorem 3.20 to the setting where the signature Γ_i ($i \in I$) is part of the input. This situation will be encountered for forest algebras (Section 3.3.1) and top dags (Section 3.4).

Before we go into the proof of Theorem 3.19, we first discuss a simple applications of Theorem 3.20 (further applications for certain tree algebras that yield Theorem 1.3 can be found in Sections 3.3 and 3.4). Consider straight-line programs over a semiring $\mathcal{A}$. Such straight-line programs are also known as arithmetic circuits in the literature. We view addition and multiplication in $\mathcal{A}$ as binary operations. In other words, we consider bounded fan-in arithmetic circuits. We also include arbitrary constants in the algebra $\mathcal{A}$ (this is necessary in order to build expressions). The following result follows directly from Theorem 3.19 and the fact that every semiring has a finite subsumption base; see Example 3.16.

Corollary 3.23. *Let $\mathcal{A}$ be an arbitrary semiring with constants (we neither assume that $\mathcal{A}$ is commutative nor that identity elements with respect to $+$ or $\times$ exist). Given an arithmetic circuit $\mathcal{G}$ over $\mathcal{A}$ such that the corresponding derivation tree t has n nodes, one can compute in time $O(|\mathcal{G}|)$ an arithmetic circuit $\mathcal{H}$ over $\mathcal{A}$ such that $[\![\mathcal{H}]\!]^{\mathcal{A}} = [\![\mathcal{G}]\!]^{\mathcal{A}}$, $|\mathcal{H}| \in O(|\mathcal{G}|)$ and depth$(\mathcal{H}) \in O(\log n)$.*

Theorem 1.2 (balancing of string straight-line programs) can be deduced in the same way from Theorem 3.19 by noting that every free monoid has a finite subsumption base.

3.2 Proof of Theorem 3.19

Those readers that worked through part I of the paper (Section 2) will notice that our proof of Theorem 3.19 is very similar to the proof of Theorem 1.2 in Section 2.3. For the following proof we will use the part I results from Sections 2.1 and 2.2.

Let us fix a signature Γ and a standard Γ-SLP $\mathcal{G} = (\mathcal{V}, \rho, S)$. Let $t = [\![\mathcal{G}]\!]$ be its derivation tree and $n = |t|$. We view $\mathcal{G}$ also as a DAG $\mathcal{D} := (\mathcal{V}, E)$ with node labels from Γ. The edge relation E contains all edges (X, i, X_i) where $\rho(X)$ is of the form $f(X_1, \ldots, X_n)$ and $1 \le i \le n$. We can assume that all nodes of the DAG are reachable from the start variable S. All variables from $\mathcal{V}$ also belong to the TSLP $\mathcal{H}$ and produce the same trees in $\mathcal{G}$ and $\mathcal{H}$. The right-hand side mapping of $\mathcal{H}$ will be denoted by τ.

We start with the symmetric centroid decomposition of the DAG $\mathcal{D}$, which can be computed in linear time as remarked in Section 2.1. Note that the number $n(\mathcal{D})$ defined in Section 2.1 is the number of leaves of t. Hence, we have $n(\mathcal{D}) \le n$. Consider a symmetric centroid path

$$(X_0, d_0, X_1), (X_1, d_1, X_2), \ldots, (X_{p-1}, d_{p-1}, X_p) \tag{8}$$

in $\mathcal{D}$, where all X_i belong to $\mathcal{V}$ and $d_i \ge 1$. Thus, for all $0 \le i \le p-1$, the right-hand side of X_i in $\mathcal{G}$ has the form

$$\rho(X_i) = f_i(X_{i,1}, \ldots, X_{i,d_i-1}, X_{i+1}, X_{i,d_i+1}, \ldots, X_{i,n_i}) \tag{9}$$

for $f_i \in \Gamma_{n_i}$, $X_{i,j} \in \mathcal{V}$ for $1 \le j \le n_i$, $j \ne d_i$. Figure 4 shows such a path. Note that the variables $X_{i,j}$ do not have to be pairwise different (as Figure 4 might suggest). Also note that the variables $X_{i,j}$ from (9) and all variables in $\rho(X_p)$ belong to other symmetric centroid paths.

We will introduce $O(p)$ many variables in the TSLP $\mathcal{H}$ to be constructed and the sizes of the corresponding right-hand sides will sum up to $\sum_{i=0}^{p} |\rho(X_i)| + O(p)$. By summing over all symmetric centroid paths of $\mathcal{D}$, this yields the size bound $O(|\mathcal{G}|)$ for $\mathcal{H}$.

Define the ground terms $t_i = [\![X_i]\!]_{\mathcal{G}}$ for $0 \le i \le p$ and $t_{i,j} = [\![X_{i,j}]\!]_{\mathcal{G}}$ for $0 \le i \le p-1$ and $1 \le j \le n_i$, $j \ne d_i$. Recall that every variable X_i $(0 \le i \le p)$ of $\mathcal{G}$ also belongs to $\mathcal{H}$. For every $0 \le i \le p-1$ we introduce a fresh variable Y_i which will evaluate in $\mathcal{H}$ to the context obtained by taking the tree t_i and cutting out the occurrence of the subtree t_p that is reached via the directions $d_i, d_{i+1}, \ldots, d_{p-1}$ from the root of t_i. In Figure 4 this context is visualized for $i = 4$ by the red part. Hence, we set

$$\tau(X_i) = Y_i[X_p] \tag{10}$$

for $0 \le i \le p$. For X_p we define

$$\tau(X_p) = \rho(X_p). \tag{11}$$

It remains to come up with right-hand sides such that every Y_i derives to the intended context. For this, we introduce variables Z_i $(0 \le i \le p-1)$ and define

$$\tau(Z_i) = f_i(X_{i,1}, \ldots, X_{i,d_i-1}, x, X_{i,d_i+1}, \ldots, X_{i,n_i}) \tag{12}$$

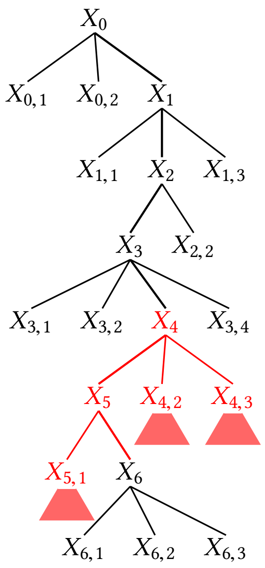

Fig. 4. A symmetric centroid path in the proof of Theorem 3.19.

for $0 \le i \le p-1$. It remains to add variables and right-hand sides such that every Y_i derives in $\mathcal{H}$ to $Z_i[Z_{i+1}[\cdots[Z_{p-1}]\cdots]]$. This is basically a string problem: we want to produce an SSLP for all suffixes of $Z_0 Z_1 \cdots Z_{p-1}$. This SSLP should have small depth in order to keep the total depth of the final TSLP bounded by $O(\log n)$. Here we use Proposition 2.2. For this we have to define the weights of the variables Z_i. We set $\|Z_i\| = |t_i| - |t_{i+1}|$. We additively extend the weight function to strings over the symbols $Z_0, \ldots, Z_{p-1}$.

Using Proposition 2.2 we can construct in time $O(p)$ a single SSLP $\mathcal{I}$ with the following properties:

- $\mathcal{I}$ has $O(p)$ many variables and all right-hand sides have length at most four,
- $\mathcal{I}$ contains the variables $Y_0, \ldots, Y_{p-1}$, where Y_i produces $Z_i Z_{i+1} \cdots Z_{p-1}$ for $0 \le i \le p-1$ and
- every path from a variable Y_i to a variable Z_k in the derivation tree of $\mathcal{I}$ has length at most $3 + 2\log_2 \|Y_i\| - 2\log_2 \|Z_k\|$ for $i \le k \le p-1$.

Note that $\|Y_i\| = |t_i| - |t_p|$. We finally add to the TSLP $\mathcal{H}$ all right-hand side definitions (10), (11), (12), and all right-hand side definitions from the SSLP $\mathcal{I}$. Here, we have to replace a concatenation YZ in a right-hand side of $\mathcal{I}$ by $Y[Z]$.

Concerning the number of introduced variables: for each X_i we introduce Y_i, Z_i, so $2p$ in total, and the $\mathcal{I}$ is guaranteed to have $O(p)$ variables as well. Summed over all paths this yields $O(n)$. For the size of the rules, each rule introduced in (11) is exactly the rule for X_p (i.e., in (9)) and similarly a rule for Z_i, where $0 \le i < p$, corresponds to a rule for X_i, in particular, $|\tau(Z_i)| = |\rho(X_i)|$. And so the sum of those productions' sizes is $\sum_{i=0}^{p} |\rho(X_i)|$. Rules in (10) have size 2 and there are p of them, so their productions' size is $2p$. Lastly, rules introduced as a translation of rules from $\mathcal{I}$ have the same size as those in $\mathcal{I}$, which is guaranteed to be $O(p)$. Thus the sum of rules' sizes is at most $\sum_{i=0}^{p} |\rho(X_i)| + O(p)$. We make the above construction for every symmetric centroid path of $\mathcal{G}$. Hence, the total size of the TSLP $\mathcal{H}$ is indeed $O(|\mathcal{G}|)$. Moreover, the construction of $\mathcal{H}$ needs linear time. It remains to show that the depth of $\mathcal{H}$ is $O(\log n)$.

First, we consider the symmetric centroid path (8) and a path in $\mathcal{H}$ from a variable X_i ($0 \le i \le p$) to a variable $X_{j,k}$ ($i \le j \le p-1$, $1 \le k \le n_j$, $k \ne d_j$) or a variable from $\rho(X_p)$. Let us define the weight $\|X\|$ for a variable $X \in \mathcal{V}$ of $\mathcal{G}$ as the size of the tree $[\![X]\!]_{\mathcal{G}}$. A path from X_i to a variable Y in $\rho(X_p)$ has the form $X_i \to Y$ or $X_i \to X_p \to Y$ (since $\tau(X_p) = \rho(X_p)$) and hence has length at most two. Now consider a path from X_i to a variable $X_{j,k}$ with $i \le j \le p-1$. We claim that the length of this path is bounded by $5 + 2\log_2 \|X_i\| - 2\log_2 \|X_{j,k}\|$. The path $X_i \xrightarrow{*} X_{j,k}$ has the form

$$X_i \to Y_i \xrightarrow{*} Z_j \to X_{j,k},$$

where $Y_i \xrightarrow{*} Z_j$ is a path in $\mathcal{I}$ and hence has length at most $3 + 2\log_2 \|Y_i\| - 2\log_2 \|Z_j\|$. Hence, the length of the path is bounded by

$$5 + 2\log_2 \|Y_i\| - 2\log_2 \|Z_j\| \le 5 + 2\log_2 \|X_i\| - 2\log_2 \|X_{j,k}\|$$

since $\|Y_i\| = |t_i| - |t_p| \le |t_i| = \|X_i\|$ and $\|Z_j\| = |t_j| - |t_{j+1}| \ge |t_{j,k}| = \|X_{j,k}\|$.

Finally, we consider a maximal path in the derivation tree of $\mathcal{H}$ that starts in the root S and ends in a leaf. We can factorize this path as

$$S = X_0 \xrightarrow{*} X_1 \xrightarrow{*} X_2 \xrightarrow{*} \cdots \xrightarrow{*} X_k \tag{13}$$

where all variables X_i belong to the original Γ-SLP $\mathcal{G}$, and every subpath $X_i \xrightarrow{*} X_{i+1}$ has the form considered in the last paragraph. The right-hand side of X_k is a single symbol from Γ_0 (such a right-hand side can appear in (11)). In the Γ-SLP $\mathcal{G}$ we have a corresponding path $X_i \xrightarrow{*} X_{i+1}$ that is contained in a single symmetric centroid path except for the last edge leading to X_{i+1}. By the above consideration, the length of the path (13) is bounded by

$$\sum_{i=0}^{k-1} (5 + 2\log_2 \|X_i\| - 2\log_2 \|X_{i+1}\|) \le 5k + 2\log_2 \|S\| = 5k + 2\log_2 n.$$

By the second claim of Lemma 2.1 we have $k \le 2\log_2 n$ which shows that the length of the path (13) is bounded by $7\log_2 n$. This concludes the proof of Theorem 3.19. □

3.3 Forest algebras and forest straight-line programs

3.3.1 Forest algebra. Let us fix a finite set Σ of node labels. In this section, we consider Σ-labelled rooted ordered trees, where "ordered" means that the children of a node are totally ordered. Every node has a label from Σ. In contrast to the trees from Section 3.1.1 we make no rank assumption: the number of children of a node (also called its degree) *is not* determined by its node label. A *forest* is a (possibly empty) sequence of such trees. The size $|v|$ of a forest is the total number of nodes in v. The set of all Σ-labelled forests is denoted by $\mathcal{F}_0(\Sigma)$. Formally, $\mathcal{F}_0(\Sigma)$ can be inductively defined as the smallest set of strings over the alphabet $\Sigma \cup \{(,)\}$ such that

- $\varepsilon \in \mathcal{F}_0(\Sigma)$ (the empty forest),
- if $u, v \in \mathcal{F}_0(\Sigma)$ then $uv \in \mathcal{F}_0(\Sigma)$, and
- if $u \in \mathcal{F}_0(\Sigma)$ then $a(u) \in \mathcal{F}_0(\Sigma)$ (this is the forest consisting of a single tree whose root is labelled with a).

Let us fix a distinguished symbol $* \notin \Sigma$. The set of forests $u \in \mathcal{F}_0(\Sigma \cup \{*\})$ such that $*$ has a unique occurrence in u and this occurrence is at a leaf node is denoted by $\mathcal{F}_1(\Sigma)$. Elements of $\mathcal{F}_1(\Sigma)$ are called *forest contexts*. Following [9], we define the *forest algebra* as the 2-sorted algebra

$$\mathsf{F}(\Sigma) = (\mathcal{F}_0(\Sigma), \mathcal{F}_1(\Sigma), \ominus_{00}, \ominus_{01}, \ominus_{10}, \oslash_0, \oslash_1, (a(*))_{a\in\Sigma}, \varepsilon, *)$$

as follows:

- $\ominus_{ij} : \mathcal{F}_i(\Sigma) \times \mathcal{F}_j(\Sigma) \to \mathcal{F}_{i+j}(\Sigma)$ ($ij \in \{00, 01, 10\}$) is a horizontal concatenation operator: for $u \in \mathcal{F}_i(\Sigma)$, $v \in \mathcal{F}_j(\Sigma)$ we set $u \ominus_{ij} v = uv$ (i.e., we concatenate the corresponding sequences of trees).

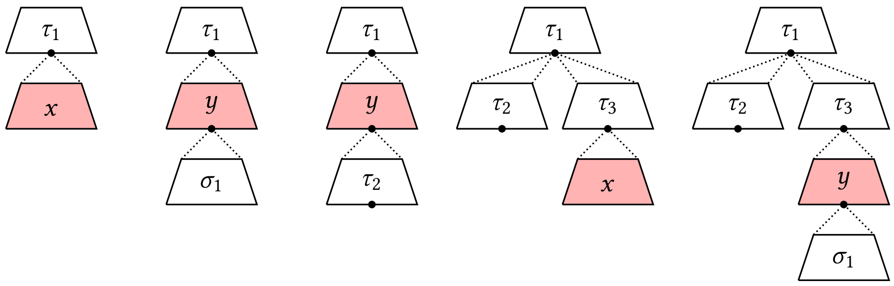

Fig. 5. The shapes of the contexts in C (proof of Lemma 3.24). Forests and forest contexts are represented by trapezoids. The roots of the forests/forest contexts are located on the top horizontal lines of the trapezoids. Bullet nodes represent occurrences of $*$. Symmetric shapes where the roles of τ_2 and τ_3 exchanged are omitted.

- $\oplus_i : \mathcal{F}_1(\Sigma) \times \mathcal{F}_i(\Sigma) \to \mathcal{F}_i(\Sigma)$ is a vertical concatenation operator: for $u \in \mathcal{F}_1(\Sigma)$ and $v \in \mathcal{F}_i(\Sigma)$, $u \oplus_i v$ is obtained by replacing in u the unique occurrence of $*$ by v.
- $\varepsilon \in \mathcal{F}_0(\Sigma)$ and $*, a(*) \in \mathcal{F}_1(\Sigma)$ ($a \in \Sigma$) are constants of the forest algebra.

Note that $(\mathcal{F}_0(\Sigma), \ominus_{00}, \varepsilon)$ and $(\mathcal{F}_1(\Sigma), \oplus_1, *)$ are monoids. In the following we will omit the subscripts i, j in $\ominus_{ij}$ and $\oplus_i$, since they will be always clear from the context. Most of the time, we simply write uv instead of $u \ominus v$, $a(u)$ instead of $a(*) \oplus u$, and a instead of $a(\varepsilon)$. With these abbreviations, a forest $u \in \mathcal{F}(\Sigma)$ can be also viewed as an algebraic expression over the algebra $\mathsf{F}(\Sigma)$, which evaluates to u itself (analogously to the free term algebra).

Lemma 3.24. *Every forest algebra* $\mathsf{F}(\Sigma)$ *has a finite subsumption base.*

Proof. In the following we denote by x and y the main variables of sorts $\mathcal{F}_0(\Sigma)$ and $\mathcal{F}_1(\Sigma)$, respectively, and by $\sigma, \sigma_1, \sigma_2, \dots$ (resp., $\tau, \tau_1, \tau_2, \dots$) auxiliary variables of sorts $\mathcal{F}_0(\Sigma)$ (resp., $\mathcal{F}_1(\Sigma)$). In the following, subsumption and equivalence of contexts are always meant with respect to the forest algebra $\mathsf{F}(\Sigma)$.

Let C be the set of containing the following contexts (see also Figure 5):

(a) $\tau_1 \oplus x$,
(b) $\tau_1 \oplus y \oplus \sigma_1$ and $\tau_1 \oplus y \oplus \tau_2$,
(c) $\tau_1 \oplus (\tau_2 \ominus (\tau_3 \oplus x))$ and $\tau_1 \oplus ((\tau_2 \oplus x) \ominus \tau_3)$,
(d) $\tau_1 \oplus (\tau_2 \ominus (\tau_3 \oplus y \oplus \sigma_1))$ and $\tau_1 \oplus ((\tau_2 \oplus y \oplus \sigma_1) \ominus \tau_3)$.

A context from point (x) (for x = a,b,c,d) will be also called a (x)-context below. First notice that every atomic context is of the form $\tau \oplus x$, $\tau \oplus y$, $y \oplus \sigma$, $y \oplus \tau$, $\sigma \ominus x$, $x \ominus \sigma$, $\sigma \ominus y$, $y \ominus \sigma$, $\tau \ominus x$, or $x \ominus \tau$ (up to variable renaming). Each of these contexts is subsumed by a context in C. For the atomic contexts $\tau \oplus x$, $\tau \oplus y$, $y \oplus \sigma$, $y \oplus \tau$, $\tau \ominus x$, and $x \ominus \tau$ this is obvious. For $\sigma \ominus x$ note that $\sigma \ominus x$ is equivalent to the context $(\sigma \ominus *) \oplus x$, which is subsumed by $\tau_1 \oplus x$. A similar argument also applies to $x \ominus \sigma$, $\sigma \ominus y$ and $y \ominus \sigma$.

Now consider any context $s \in C$. We prove that for any atomic context s' from above, $s'[s]$ is subsumed by some context from C.

Case $\tau \oplus s$ Since s is of the form $s = \tau_1 \oplus s'$ for some s', the context $\tau \oplus s = \tau \oplus (\tau_1 \oplus s')$ is subsumed by $s \in C$ itself.

Case $s \oplus \sigma$ **and** $s \oplus \tau$ In this case s must be either the (b)-context $s = \tau_1 \oplus y \oplus \tau_2$, a (c)-context or a (d)-context.

(1) If $s = \tau_1 \oplus y \oplus \tau_2$, then $s \oplus \sigma$ and $s \oplus \tau$ are subsumed by a (b)-context.

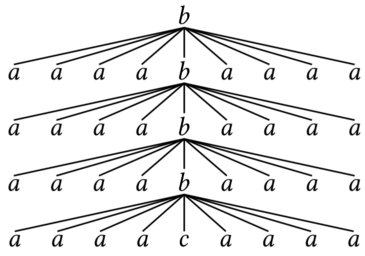

Fig. 6. Forest $[\![\mathcal{G}]\!]^{\mathsf{F}(\Sigma)}$ for $n = 2$ from Example 3.26.

(2) Assume that s is a (c)-context, say $s = \tau_1 \oplus (\tau_2 \ominus (\tau_3 \oplus x))$. Then $s \oplus \sigma$ is equivalent to $(\tau_1 \oplus ((\tau_2 \oplus \sigma) \ominus \tau_3)) \oplus x$ which is subsumed by $\tau_1 \oplus x$. Moreover, $s \oplus \tau$ is equivalent to $\tau_1 \oplus ((\tau_2 \oplus \tau) \ominus (\tau_3 \oplus x))$, which is subsumed by s itself.

(3) Assume that s is a (d)-context, say $s = \tau_1 \oplus (\tau_2 \ominus (\tau_3 \oplus y \oplus \sigma_1))$. Firstly, $s \oplus \sigma$ is equivalent to $(\tau_1 \oplus ((\tau_2 \oplus \sigma) \ominus \tau_3)) \oplus y \oplus \sigma_1$, which is subsumed by the context $\tau_1 \oplus x \oplus \sigma_1$. Secondly, $s \oplus \tau$ is equivalent to $\tau_1 \oplus ((\tau_2 \oplus \tau) \ominus (\tau_3 \oplus y \oplus \sigma_1))$, which is subsumed by s itself.

Case $\sigma \ominus s$ and $s \ominus \sigma$ Since s is of the form $s = \tau_1 \oplus s'$ for some s' the context $\sigma \ominus s$ is equivalent to $(\sigma \ominus \tau_1) \oplus s'$, which is subsumed by s itself. The case $s \ominus \sigma$ is similar.

Case $\tau \ominus s$ and $s \ominus \tau$ In this case s must be either the (a)-context or the (b)-context $\tau_1 \oplus y \oplus \sigma_1$. If s is the (a)-context $\tau_1 \oplus x$ then $\tau \ominus s$ is subsumed by the (c)-context $\tau_1 \oplus (\tau_2 \ominus (\tau_3 \oplus x))$. If s is the (b)-context $\tau_1 \oplus y \oplus \sigma_1$ then $\tau \ominus s$ is subsumed by the (d)-context $\tau_1 \oplus (\tau_2 \ominus (\tau_3 \oplus y \oplus \sigma_1))$. The case $s \ominus \tau$ is similar.

By Lemma 3.14, C is a finite subsumption base. □

Remark 3.25. Similarly to the proof of Lemma 3.24 one can show that for every signature Γ the functional extension $\hat{\mathcal{T}}(\Gamma)$ of the free term algebra $\mathcal{T}(\Gamma)$ has a finite subsumption base as well. Recall from Example 3.17 that the free term algebra $\mathcal{T}(\Gamma)$ has no finite subsumption base if Γ contains a symbol of rank at least one.

3.3.2 *Forest straight-line programs.*

A *forest straight-line program* over Σ, FSLP for short, is a straight-line program $\mathcal{G}$ over the algebra $\mathsf{F}(\Sigma)$ such that $[\![\mathcal{G}]\!]^{\mathsf{F}(\Sigma)} \in \mathcal{F}_0(\Sigma)$. Iterated vertical and horizontal concatenations allow to generate forests, whose depth and width is exponential in the size of the FSLP. For an FSLP $\mathcal{G} = (\mathcal{V}, \rho, S)$ and $i \in \{0,1\}$ we define $\mathcal{V}_i = \{X \in \mathcal{V} \mid [\![X]\!]^{\mathsf{F}(\Sigma)} \in \mathcal{F}_i(\Sigma)\}$. Every right-hand side of a standard FSLP $\mathcal{G}$ must have one of the following forms: (i) ε (the empty forest), (ii) $*$, (iii) $a(*)$ for $a \in \Sigma$, (iv) $X \ominus Y$ (for which we write XY) for $X, Y \in \mathcal{V}$ with $X \in \mathcal{V}_0$ or $Y \in \mathcal{V}_0$, or (v) $X \oplus Y$ for $X \in \mathcal{V}_1$ and $Y \in \mathcal{V}$.

Example 3.26. Let $n \in \mathbb{N}$. Consider the (non-standard) FSLP

$$\mathcal{G} = (\{S, X_0, \ldots, X_n, Y_0, \ldots, Y_n\}, \rho, S)$$

over $\{a, b, c\}$ with ρ defined by $\rho(X_0) = a$, $\rho(X_i) = X_{i-1}X_{i-1}$ for $1 \le i \le n$, $\rho(Y_0) = b(X_n * X_n)$, $\rho(Y_i) = Y_{i-1} \oplus Y_{i-1}$ for $1 \le i \le n$, and $\rho(S) = Y_n \oplus c$. We have

$$[\![\mathcal{G}]\!]^{\mathsf{F}(\Sigma)} = b(a^{2^n} b(a^{2^n} \cdots b(a^{2^n} c\, a^{2^n}) \cdots a^{2^n}) a^{2^n}),$$

where b occurs 2^n many times, see Figure 6 for $n = 2$.

Let us first show that most occurrences of ε and $*$ can be eliminated in an FSLP.

Lemma 3.27. *From a given FSLP $\mathcal{G}$ with $[\![\mathcal{G}]\!]^{\mathsf{F}(\Sigma)} \neq \varepsilon$ one can compute in linear time an FSLP $\mathcal{H}$ such that $[\![\mathcal{G}]\!]^{\mathsf{F}(\Sigma)} = [\![\mathcal{H}]\!]^{\mathsf{F}(\Sigma)}$, $|\mathcal{H}| \in O(|\mathcal{G}|)$, depth$(\mathcal{H}) \in O($depth$(\mathcal{G}))$, and $\mathcal{H}$ does not contain occurrences of the constants ε and $*$, except for right-hand sides of the form $a(\varepsilon)$.*[2]

Proof. Let $\mathcal{G} = (\mathcal{V}, \rho, S)$. We first construct an equivalent FSLP which does not contain the constant $*$. Let us denote with $\mathcal{V}_* \subseteq \mathcal{V}_1$ the set of all variables $X \in \mathcal{V}_1$ such that $[\![X]\!]^{\mathsf{F}(\Sigma)}$ is of the form $u_\ell * u_r$ for forests $u_\ell, u_r \in \mathcal{F}_0(\Sigma)$. In other words: $*$ occurs at a root position in the forest $[\![X]\!]^{\mathsf{F}(\Sigma)}$. The set $\mathcal{V}_*$ can be easily computed in linear time by a single pass over $\mathcal{G}$. Every variable $X \in \mathcal{V}_*$ with $[\![X]\!]^{\mathsf{F}(\Sigma)} = u_\ell * u_r$ is replaced in $\mathcal{H}$ by two variables X_ℓ and X_r that produce in $\mathcal{H}$ the forests u_ℓ and u_r, respectively. Every variable $X \in \mathcal{V}_1 \setminus \mathcal{V}_*$ is replaced in $\mathcal{H}$ by three variables X_t, X_ℓ, X_r. Since $X \in \mathcal{V}_1 \setminus \mathcal{V}_*$, $[\![X]\!]^{\mathsf{F}(\Sigma)}$ contains a unique subtree of the form $a(u_\ell * u_r)$. Let us denote with u_t (the top part of u) the forest that is obtained from u by replacing the subtree $a(u_\ell * u_r)$ by $a(*)$. We then will have $[\![X_t]\!]^{\mathsf{F}(\Sigma)} = u_t$, $[\![X_\ell]\!]^{\mathsf{F}(\Sigma)} = u_\ell$, and $[\![X_r]\!]^{\mathsf{F}(\Sigma)} = u_r$. Finally, all variables from $\mathcal{V}_0$ also belong to $\mathcal{H}$ and produce in $\mathcal{H}$ the same forests as in $\mathcal{G}$.

It is straight-forward to define the right-hand sides of $\mathcal{H}$ such that the variables indeed produce the desired forests (τ denotes the right-hand side mapping of $\mathcal{H}$):

- If $\rho(X) = *$ then $\tau(X_\ell) = \tau(X_r) = \varepsilon$.
- If $\rho(X) = a(*)$ then $\tau(X_t) = a(*)$ and $\tau(X_\ell) = \tau(X_r) = \varepsilon$.
- If $\rho(X) = \varepsilon$ or $\rho(X) = YZ$ with $X, Y, Z \in \mathcal{V}_0$ then $\tau(X) = \rho(X)$.
- If $\rho(X) = YZ$ with $X, Y \in \mathcal{V}_*$ and $Z \in \mathcal{V}_0$ then $\tau(X_\ell) = Y_\ell$ and $\tau(X_r) = Y_r Z$, and analogously for $X, Z \in \mathcal{V}_*$ and $Y \in \mathcal{V}_0$.
- If $\rho(X) = YZ$ with $X, Y \in \mathcal{V}_1 \setminus \mathcal{V}_*$ and $Z \in \mathcal{V}_0$ then $\tau(X_\ell) = Y_\ell$, $\tau(X_r) = Y_r$, and $\tau(X_t) = Y_t Z$, and analogously for $X, Z \in \mathcal{V}_1 \setminus \mathcal{V}_*$ and $Y \in \mathcal{V}_0$.
- If $\rho(X) = Y \oplus Z$ with $X, Z \in \mathcal{V}_0$ and $Y \in \mathcal{V}_*$ then $\tau(X) = Y_\ell Z Y_r$.
- If $\rho(X) = Y \oplus Z$ with $X, Z \in \mathcal{V}_0$ and $Y \in \mathcal{V}_1 \setminus \mathcal{V}_*$ then $\tau(X) = Y_t \oplus (Y_\ell Z Y_r)$.
- If $\rho(X) = Y \oplus Z$ with $X, Y, Z \in \mathcal{V}_*$ then $\tau(X_\ell) = Y_\ell Z_\ell$ and $\tau(X_r) = Z_r Y_r$.
- If $\rho(X) = Y \oplus Z$ with $Y \in \mathcal{V}_*$ and $X, Z \in \mathcal{V}_1 \setminus \mathcal{V}_*$ then $\tau(X_t) = Y_\ell Z_t Y_r$, $\tau(X_\ell) = Z_\ell$, and $\tau(X_r) = Z_r$.
- If $\rho(X) = Y \oplus Z$ with $Z \in \mathcal{V}_*$ and $X, Y \in \mathcal{V}_1 \setminus \mathcal{V}_*$ then $\tau(X_t) = Y_t$, $\tau(X_\ell) = Y_\ell Z_\ell$, and $\tau(X_r) = Z_r Y_r$.
- If $\rho(X) = Y \oplus Z$ with $X, Y, Z \in \mathcal{V}_1 \setminus \mathcal{V}_*$ then $\tau(X_t) = Y_t \oplus (Y_\ell Z_t Y_r)$, $\tau(X_\ell) = Z_\ell$, and $\tau(X_r) = Z_r$.

Note that all right-hand sides of the new FSLP have constant length. Variables X such that $\tau(X)$ is a variable can be eliminated.

Let us finally eliminate occurrences of the constant ε, except for right-hand sides of the form $a(\varepsilon)$. Let us take an FSLP $\mathcal{G} = (\mathcal{V}, \rho, S)$ with $[\![\mathcal{G}]\!]^{\mathsf{F}(\Sigma)} \neq \varepsilon$ and which does not contain occurrences of the constant $*$. Let $\mathcal{V}_\varepsilon = \{X \in \mathcal{V}_0 \mid [\![X]\!]^{\mathsf{F}(\Sigma)} = \varepsilon\}$. Note that $S \notin \mathcal{V}_\varepsilon$. The set $\mathcal{V}_\varepsilon$ can be easily computed in linear time by a single pass over $\mathcal{G}$. We construct an equivalent FSLP $\mathcal{H}$ which neither contains $*$ nor ε, except for right-hand sides of the form $a(*)$ and $a(\varepsilon)$. All variables from $\mathcal{G}$ are also contained in $\mathcal{H}$, except for variables in $\mathcal{V}_\varepsilon$. For every variable $X \in \mathcal{V}_1$, $\mathcal{H}$ also contains a copy X_ε that produces $[\![X]\!]^{\mathsf{F}(\Sigma)} \oplus \varepsilon$. The right-hand side mapping τ of $\mathcal{H}$ is defined as follows:

- If $\rho(X) = a(*)$ then $\tau(X) = a(*)$ and $\tau(X_\varepsilon) = a(\varepsilon)$.
- If $\rho(X) = \varepsilon$ then X does not belong to $\mathcal{H}$.
- If $\rho(X) = YZ$ with $Y, Z \in \mathcal{V}_\varepsilon$ then $X \in \mathcal{V}_\varepsilon$ does not belong to $\mathcal{H}$.
- If $\rho(X) = YZ$ or $\rho(X) = ZY$ with $Y \in \mathcal{V}_\varepsilon$ and $Z \in \mathcal{V}_0 \setminus \mathcal{V}_\varepsilon$ then $\tau(X) = Z$.
- If $\rho(X) = YZ$ with $Y, Z \in \mathcal{V}_0 \setminus \mathcal{V}_\varepsilon$ then $\tau(X) = YZ$.

[2]Constants $a(*)$ are allowed as well. Formally, $a(*)$ is a constant symbol that is interpreted by the forest context $a(*)$.

- If $\rho(X) = YZ$ or $\rho(X) = ZY$ with $Y \in \mathcal{V}_\varepsilon$ and $X, Z \in \mathcal{V}_1$ then $\tau(X) = Z$ and $\tau(X_\varepsilon) = Z_\varepsilon$.
- If $\rho(X) = YZ$ with $Y \in \mathcal{V}_0 \setminus \mathcal{V}_\varepsilon$ and $X, Z \in \mathcal{V}_1$ then $\tau(X) = YZ$ and $\tau(X_\varepsilon) = YZ_\varepsilon$, and similarly if $Z \in \mathcal{V}_0 \setminus \mathcal{V}_\varepsilon$ and $X, Y \in \mathcal{V}_1$.
- If $\rho(X) = Y \oplus Z$ with $X, Y, Z \in \mathcal{V}_1$ then $\tau(X) = Y \oplus Z$ and $\tau(X_\varepsilon) = Y \oplus Z_\varepsilon$.
- If $\rho(X) = Y \oplus Z$ with $Y \in \mathcal{V}_1$ and $Z \in \mathcal{V}_\varepsilon$ then $\tau(X) = Y_\varepsilon$.
- If $\rho(X) = Y \oplus Z$ with $Y \in \mathcal{V}_1$ and $Z \in \mathcal{V}_0 \setminus \mathcal{V}_\varepsilon$ then $\tau(X) = Y \oplus Z$.

As in the previous case, variables X such that $\tau(X)$ is a variable, can be eliminated. Note that the construction does not introduce new occurrences of $*$. All variables from $\mathcal{V} \setminus \mathcal{V}_\varepsilon$ produce the same forest in $\mathcal{G}$ and $\mathcal{H}$, which implies $[\![\mathcal{G}]\!]^{\mathsf{F}(\Sigma)} = [\![\mathcal{H}]\!]^{\mathsf{F}(\Sigma)}$. Finally note that both constructions increase the size and depth of the FSLP only by a constant factor. □

Corollary 3.28. *Given a finite alphabet Σ and an FSLP $\mathcal{G}$ over the forest algebra $\mathsf{F}(\Sigma)$ defining the forest $u = [\![\mathcal{G}]\!]^{\mathsf{F}(\Sigma)}$, one can compute in time $O(|\mathcal{G}|)$ an FSLP $\mathcal{H}$ such that $[\![\mathcal{H}]\!]^{\mathsf{F}(\Sigma)} = u$, $|\mathcal{H}| \in O(|\mathcal{G}|)$ and depth$(\mathcal{H}) \in O(\log |u|)$.*

Proof. The case $u = \varepsilon$ is trivial. Let us now assume that $u \neq \varepsilon$. We first apply Lemma 3.27 and construct from $\mathcal{G}$ in linear time an equivalent FSLP $\mathcal{G}'$ which does not contain occurrences of the constants $*$ and ε, except for right-hand sides of the form $a(\varepsilon)$. This ensures that the derivation tree $t = [\![\mathcal{G}']\!]$ has size $O(|u|)$. The size and depth of $\mathcal{G}'$ are linearly bounded in the size and depth, respectively, of $\mathcal{G}$. By Lemma 3.24 we can apply Theorem 3.20 in order to get the FSLP $\mathcal{H}$ with the desired properties for the situation where the alphabet Σ is fixed. For the situation where Σ is part of the input one has to use Remark 3.22. The arguments are analogous to the proof of Theorem 1.2. Note in particular that the subsumption base from the proof of Lemma 3.24 does not depend on the alphabet Σ of the forest algebra $\mathsf{F}(\Sigma)$. □

Remark 3.29. Using Remark 3.25 one can show the following variant of Corollary 3.28: Take a fixed signature Γ. From a given Γ-TSLP $\mathcal{G}$ defining the tree $t = [\![\mathcal{G}]\!]^{\hat{\mathcal{T}}(\Gamma)}$, one can compute in time $O(|\mathcal{G}|)$ a Γ-TSLP $\mathcal{H}$ such that $[\![\mathcal{H}]\!]^{\hat{\mathcal{T}}(\Gamma)} = t$, $|\mathcal{H}| \in O(|\mathcal{G}|)$ and depth$(\mathcal{H}) \in O(\log |t|)$. In other words, Γ-TSLPs can be balanced with a linear size increase. Note that this is a much stronger statement than Theorem 3.19, which states that a Γ-SLP can be balanced into an equivalent Γ-TSLP with a linear size increase. On the other hand, the above balancing result for Γ-TSLPs finally uses the weaker Theorem 3.19 in its proof. We have to assume a fixed signature Γ in the above argument since the size of the contexts in a finite subsumption for $\hat{\mathcal{T}}(\Gamma)$ depends on the maximal rank of the symbols in Γ.

Alternatively, the balancing result for Γ-TSLPs can be deduced from the corresponding balancing result for FSLPs (Corollary 3.28): A given Γ-TSLP $\mathcal{G}$ can be directly translated into an FSLP $\mathcal{G}_1$ for the tree $[\![\mathcal{G}]\!]^{\hat{\mathcal{T}}(\Gamma)}$. The size of $\mathcal{G}_1$ is $O(|\mathcal{G}|)$. Using Corollary 3.28 one can compute from $\mathcal{G}_1$ a balanced FSLP $\mathcal{G}_2$ of size $O(|\mathcal{G}_1|)$. Finally, the FSLP $\mathcal{G}_2$ can be easily transformed back into a Γ-TSLP of size $O(r \cdot |\mathcal{G}_2|)$, where r is the maximal rank of a symbol in Γ. For this one has to eliminate horizontal concatenations in the FSLP. Since we assumed Γ to be a fixed signature, r is a constant.

3.4 Cluster algebras and top dags

FSLPs are very similar to *top dags* that were introduced in [7] and further studied in [4, 13, 22]. In fact, top dags can be defined in the same way as FSLPs, one only has to slightly change the two concatenation operations $\ominus$ and $\oplus$, which yields the so called cluster algebra defined below.

Let us fix an alphabet Σ of node labels and define for $a \in \Sigma$ the set $\mathcal{K}_a(\Sigma) = \{a(u) \mid u \in \mathcal{F}_0(\Sigma) \setminus \{\varepsilon\}\}$. Note that $\mathcal{K}_a(\Sigma)$ consists of unranked Σ-labelled trees of size at least two, where the root is labeled with a. Elements of $\mathcal{K}_a(\Sigma)$ (for any a) are also called *clusters of rank* 0. For $a, b \in \Sigma$ let $\mathcal{K}_{ab}(\Sigma)$ be the set of all trees $t \in \mathcal{K}_a(\Sigma)$ together with a distinguished b-labelled leaf of t, which is called the *bottom boundary node* of t. Elements of $\mathcal{K}_{ab}(\Sigma)$ (for any a, b) are called *clusters of rank one*. The root node of a cluster t (of rank zero or one) is called the *top boundary node* of t. When writing a cluster of rank one, we underline the bottom boundary node. For instance $a(bc(\underline{b}a))$ is an element of $\mathcal{K}_{ab}(\Sigma)$. An *atomic cluster* is of the form $a(b)$ or $a(\underline{b})$ for $a, b \in \Sigma$.

We define the *cluster algebra* $\mathsf{K}(\Sigma)$ as an algebra over a $(\Sigma \cup \Sigma^2)$-sorted signature. The universe of sort $a \in \Sigma$ is $\mathcal{K}_a(\Sigma)$ and the universe of sort $ab \in \Sigma^2$ is $\mathcal{K}_{ab}(\Sigma)$. The operations of $\mathsf{K}(\Sigma)$ are the following:

- There are $|\Sigma| + 2|\Sigma|^2$ many horizontal merge operators; we denote all of them with the same symbol $\ominus$. Their domains and ranges are specified by: $\ominus \colon \mathcal{K}_a(\Sigma) \times \mathcal{K}_a(\Sigma) \to \mathcal{K}_a(\Sigma)$, $\ominus \colon \mathcal{K}_a(\Sigma) \times \mathcal{K}_{ab}(\Sigma) \to \mathcal{K}_{ab}(\Sigma)$, and $\ominus \colon \mathcal{K}_{ab}(\Sigma) \times \mathcal{K}_a(\Sigma) \to \mathcal{K}_{ab}(\Sigma)$, where $a, b \in \Sigma$. All of these merge operators are defined by $a(u) \ominus a(v) = a(uv)$, where sorts of the clusters u, v must match the input sorts for one of the merge operators.
- There are $|\Sigma|^2 + |\Sigma|^3$ many vertical merge operators; we denote all of them with the same symbol $\varobar$. Their domains and ranges are specified by: $\varobar \colon \mathcal{K}_{ab}(\Sigma) \times \mathcal{K}_b(\Sigma) \to \mathcal{K}_a(\Sigma)$ and $\varobar \colon \mathcal{K}_{ab}(\Sigma) \times \mathcal{K}_{bc}(\Sigma) \to \mathcal{K}_{ac}(\Sigma)$ for $a, b, c \in \Sigma$. For clusters $s \in \mathcal{K}_{ab}(\Sigma)$ and $t \in \mathcal{K}_b(\Sigma) \cup \mathcal{K}_{bc}(\Sigma)$ we obtain $s \varobar t$ by replacing in s the bottom boundary node by t. For instance,
$$a(bc(\underline{b}a)) \varobar b(ac) = a(bc(b(ac)a)).$$
- The atomic clusters $a(b)$ and $a(\underline{b})$ are constants of the cluster algebra.

In the following, we just write $\mathcal{K}_a$ and $\mathcal{K}_{ab}$ for $\mathcal{K}_a(\Sigma)$ and $\mathcal{K}_{ab}(\Sigma)$, respectively. A *top dag* over Σ is an SLP $\mathcal{G}$ over the algebra $\mathsf{K}(\Sigma)$ such that $[\![\mathcal{G}]\!]^{\mathsf{K}(\Sigma)}$ is a cluster of rank zero.[3] In our terminology, cluster straight-line program would be a more appropriate name, but we prefer to use the original term "top dag".

Example 3.30. Consider the top dag $\mathcal{G} = (\{S, X_0, \dots, X_n, Y_0, \dots, Y_n\}, \rho, S)$ with $\rho(X_0) = b(a)$, $\rho(X_i) = X_{i-1} \ominus X_{i-1}$ for $1 \le i \le n$, $\rho(Y_0) = X_n \ominus b(\underline{b}) \ominus X_n$, $\rho(Y_i) = Y_{i-1} \varobar Y_{i-1}$ for $1 \le i \le n$, and $\rho(S) = Y_n \varobar b(c)$. We have
$$[\![\mathcal{G}]\!]^{\mathsf{K}(\Sigma)} = b(a^{2^n} b(a^{2^n} \cdots b(a^{2^n} b(c) a^{2^n}) \cdots a^{2^n}) a^{2^n}),$$
where b occurs $2^n + 1$ many times.

In [16] it was shown that from a top dag $\mathcal{G}$ one can compute in linear time an equivalent FSLP of size $O(|\mathcal{G}|)$. Vice versa, from an FSLP $\mathcal{H}$ for a tree $t \in C_a$ (for some $a \in \Sigma$) one can compute in time $O(|\Sigma| \cdot |\mathcal{H}|)$ an equivalent top dag of size $O(|\Sigma| \cdot |\mathcal{H}|)$. The additional factor $|\Sigma|$ in the transformation from FSLPs to top dags is unavoidable; see [16] for an example.

Lemma 3.31. *Every cluster algebra* $\mathsf{K}(\Sigma)$ *has a finite subsumption base.*

Proof. The proof is similar to the proof of Lemma 3.24. Let the set C contain the following contexts, where in each context, each of the auxiliary variables $\sigma_1, \sigma_2, \sigma_3, \tau_1, \tau_2, \tau_3, \tau_4$ can be also missing (this is necessary since in the cluster algebra, the merge operations have no neutral elements). The main variable x and the auxiliary variables $\sigma_1, \sigma_2, \sigma_3$ must have sorts from Σ (rank zero), whereas the main variable y and the auxiliary variables $\tau_1, \tau_2, \tau_3, \tau_4$ must have sorts from $\Sigma\Sigma$ (rank one). The concrete sorts must be chosen such that all horizontal and vertical merge operations are defined.

(a) $\tau_1 \varobar (\sigma_1 \ominus x \ominus \sigma_2)$

[3]Note that the definition of a top dag in [7] refers to the outcome of a particular top dag construction. In other words: for every tree t a very specific SLP over the cluster algebra is constructed and this SLP is called the top dag of t. Here, as in [16], we call any SLP over the cluster algebra a top dag.

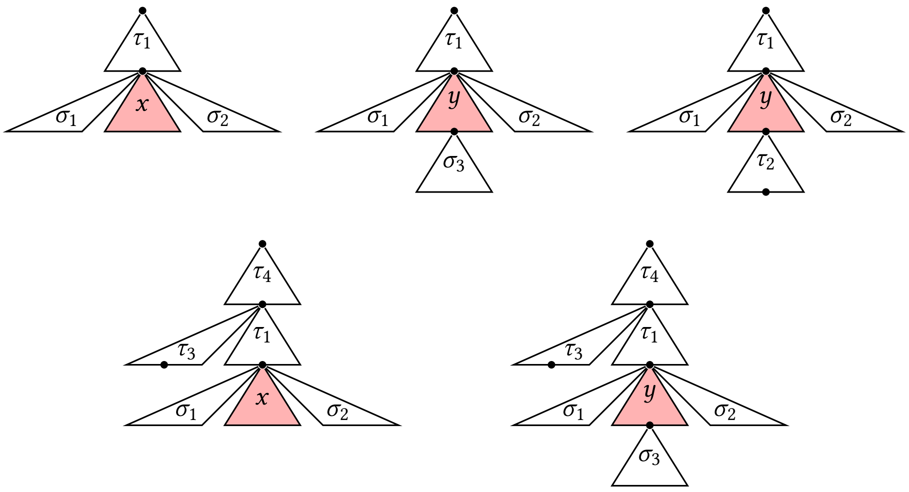

Fig. 7. The shapes of the contexts in C (proof of Lemma 3.31). Bullet nodes represent boundary nodes. Symmetric shapes where τ_3 is to the right of τ_1 are omitted.

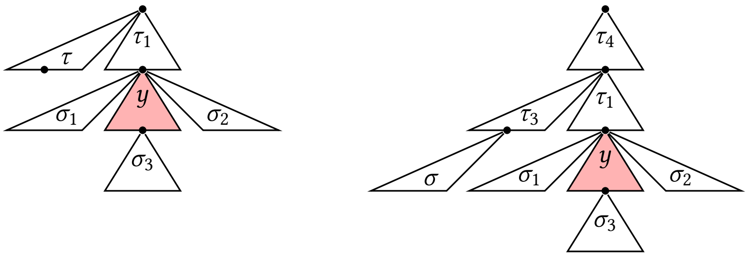

Fig. 8. Two example cases from the proof of Lemma 3.31.

(b) $\tau_1 \oplus (\sigma_1 \ominus y \ominus \sigma_2) \oplus \sigma_3$
(c) $\tau_1 \oplus (\sigma_1 \ominus y \ominus \sigma_2) \oplus \tau_2$
(d) $\tau_4 \oplus (\tau_3 \ominus (\tau_1 \oplus (\sigma_1 \ominus x \ominus \sigma_2)))$
(e) $\tau_4 \oplus ((\tau_1 \oplus (\sigma_1 \ominus x \ominus \sigma_2)) \ominus \tau_3)$
(f) $\tau_4 \oplus (\tau_3 \ominus (\tau_1 \oplus (\sigma_1 \ominus y \ominus \sigma_2) \oplus \sigma_3))$
(g) $\tau_4 \oplus ((\tau_1 \oplus (\sigma_1 \ominus y \ominus \sigma_2) \oplus \sigma_3) \ominus \tau_3)$

Note that these forms are very similar to the forms (a)–(g) for forest algebras from the proof of Lemma 3.24. Only the variables σ_1 and σ_2 that are horizontally merged with x (resp., y) are new.

Figure 7 shows the shapes of the above contexts. Let us explain the intuition behind these shapes. Take a cluster s (of rank zero or one) and cut out from s a subcluster x of rank zero or a subcluster y of rank one. We do not give a formal definition of subclusters (see [7]), but roughly speaking this means that x (resp., y) is a cluster that occurs somewhere in s. In Figure 7, these subclusters are the red triangles. The part of s that does not belong to the subcluster x (resp., y) can be partitioned into finitely many subclusters, and these are the white triangles in Figure 7.

Using Lemma 3.14 we can show that C is a finite subsumption base for the cluster algebra $\mathsf{K}(\Sigma)$. The atomic clusters are $\tau \ominus x$, $\sigma \ominus x$, $\sigma \ominus y$, $x \ominus \tau$, $x \ominus \sigma$, $y \ominus \sigma$, $\tau \oplus x$, $\tau \oplus y$, $x \oplus \tau$, $x \oplus \sigma$ (where x and σ have sorts from Σ and y and τ have sorts from $\Sigma\Sigma$). Each of these atomic contexts belongs to

C up to renaming of auxiliary variables. For this it is important that every context from the above list (a)–(g), where some of the auxiliary variables are omitted, belongs to C as well.

Let us now consider a context $s'[s]$, where $s \in C$ and s' is atomic. We have to show that $s'[s]$ is subsumed in $\mathsf{K}(\Sigma)$ by a context from C. The case distinction is very similar to the proof of Lemma 3.24. Two examples are shown in Figure 8. The left figure shows the case $s = \tau_1 \oplus (\sigma_1 \ominus y \ominus \sigma_2) \oplus \sigma_3$ and $s' = \tau \ominus x$. In this case $s'[s] = \tau \ominus (\tau_1 \oplus (\sigma_1 \ominus y \ominus \sigma_2) \oplus \sigma_3)$ is subsumed in $\mathsf{K}(\Sigma)$ by $\tau_3 \ominus (\tau_1 \oplus (\sigma_1 \ominus y \ominus \sigma_2) \oplus \sigma_3)$ (the latter is obtained from the context in (f) by removing τ_4).

Figure 8 on the right shows the case $s = \tau_4 \oplus (\tau_3 \ominus (\tau_1 \oplus (\sigma_1 \ominus y \ominus \sigma_2) \oplus \sigma_3))$ and $s' = y \oplus \sigma$. We have $s'[s] = \tau_4 \oplus ((\tau_3 \oplus \sigma) \ominus (\tau_1 \oplus (\sigma_1 \ominus y \ominus \sigma_2) \oplus \sigma_3))$, which is equivalent in $\mathsf{K}(\Sigma)$ to $(\tau_4 \ominus ((\tau_3 \oplus \sigma) \ominus \tau_1)) \oplus (\sigma_1 \ominus y \ominus \sigma_2) \oplus \sigma_3$. The latter context is subsumed in $\mathsf{K}(\Sigma)$ by $\tau_1 \oplus (\sigma_1 \ominus y \ominus \sigma_2) \oplus \sigma_3 \in C$. ◻

We can now show the main result for top dags:

Corollary 3.32. *Given a finite alphabet Σ and a top dag $\mathcal{G}$ over the cluster algebra $\mathsf{K}(\Sigma)$ producing the tree $t = [\![\mathcal{G}]\!]^{\mathsf{K}(\Sigma)}$, one can compute in time $O(|\mathcal{G}|)$ a top dag $\mathcal{H}$ for t of size $O(|\mathcal{G}|)$ and depth $O(\log |t|)$.*

Proof. Note that in the derivation tree $[\![\mathcal{G}]\!]$ of a top dag $\mathcal{G}$, all leaves are labelled with atomic clusters and all internal nodes have rank two. Hence, the size of the derivation tree $[\![\mathcal{G}]\!]$ is linearly bounded in the size of the generated tree $[\![\mathcal{G}]\!]^{\mathsf{K}(\Sigma)}$ (in the forest algebra, we needed Lemma 3.27 to enforce this property). For the case of a fixed alphabet Σ, the statement of the corollary follows from Lemma 3.31 and Theorem 3.20 analogously to Corollary 3.28 for FSLPs. For the general case of a variable-size alphabet Σ we have to use again Remark 3.22. As for SSLPs and FSLPs we need the natural assumption that symbols from the input alphabet fit into a single machine word of the RAM. All operations from a cluster algebra have rank zero and two, and the subsumption base C from the proof of Lemma 3.31 has the property that every context $s \in \Sigma$ has constant size. In contrast to free monoids and forest algebras, the subsumption base depends on the alphabet Σ. Basically, we need to choose the sorts of the variables $x, y, \tau_1, \tau_2, \tau_3, \sigma_1, \sigma_2, \sigma_3, \sigma_4$ in each of the contexts from C. This implies that every context $s \in C$ can be represented by a constant number of symbols from Σ and hence can be stored in a constant number of machine words. The constant time algorithms from point (iii) and (iv) from Remark 3.22 make a constant number of comparisons between the Σ-symbols representing the input contexts. ◻

In [20] top dags have been used for compressed range minimum queries (RMQs). It is well known that for a string s of integers one can reduce RMQs to lowest common ancestor queries on the Cartesian tree corresponding to s. Two compressed data structures for answering RMQs for s are proposed in [20]: one is based on an SSLP for s, we commented on it already in Section 2.4, the other one uses a top dag for the Cartesian tree corresponding to s. The following result has been shown, see [20, Corollary 1.4]:
Given a string s of length n over an alphabet of σ many integers, let m_{opt} denote the size of a smallest SSLP for s. There is a top dag $\mathcal{G}$ for the Cartesian tree corresponding to s of size $|\mathcal{G}| \leq \min(O(n/\log n), O(m_{\text{opt}} \cdot \log n \cdot \sigma))$, and there is a data structure of size $O(|\mathcal{G}|)$ that answers range minimum queries on s in time $O(\log \sigma \cdot \log n)$.

As the time bound $O(\log \sigma \cdot \log n)$ comes from the height of the constructed top dag, using Corollary 3.32 we can enforce the bound $O(\log n)$ on the height of the constructed top dag and ensure that the transformation can be applied to any input SSLP. This yields the following improvement of the result of [20]:

Theorem 3.33. *Given an SSLP of size m generating a string s of length n over an alphabet of σ many integers one can compute a top dag $\mathcal{G}$ for the Cartesian tree corresponding to s of size $|\mathcal{G}| \leq \min(O(n/\log n), O(m \cdot \sigma))$ and depth $O(\log n)$, and there is a data structure of size $O(|\mathcal{G}|)$ that answers RMQs on s in time $O(\log n)$. If m_{opt} denotes the size of a smallest SSLP generating s then, using Rytter's algorithm, we can assume that $m \leq O(m_{opt} \cdot \log n)$.*

4 OPEN PROBLEMS

For SSLPs one may require a strong notion of balancing. Let us say that an SSLP $\mathcal{G}$ is c-balanced if (i) the length of every right-hand side is at most c and (ii) if a variable Y occurs in $\rho(X)$ then $|[\![Y]\!]_{\mathcal{G}}| \leq |[\![X]\!]_{\mathcal{G}}|/2$. It is open, whether there is a constant c such that for every SSLP of size m there exists an equivalent c-balanced SSLP of size $O(m)$.

Another important open problem is whether the query time bound in Theorem 1.1 (random access to grammar-compressed strings) can be improved from $O(\log n)$ to $O(\log n/\log\log n)$. If we allow space $O(m \cdot \log^{\epsilon} n)$ (for any small $\epsilon > 0$) then such an improvement is possible by Corollary 2.4, but it is open whether query time $O(\log n/\log\log n)$ can be achieved with space $O(m)$. By the lower bound from [39] this would be an optimal random-access data structure for grammar-compressed strings.

REFERENCES

[1] Eric Allender, Jia Jiao, Meena Mahajan, and V. Vinay. Non-commutative arithmetic circuits: Depth reduction and size lower bounds. *Theoretical Computer Science*, 209(1-2):47–86, 1998.

[2] Djamal Belazzougui, Patrick Hagge Cording, Simon J. Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In *Proceedings of the 23rd Annual European Symposium on Algorithms, ESA 2015*, volume 9294 of *Lecture Notes in Computer Science*, pages 142–154. Springer, 2015.

[3] Philip Bille, Patrick Hagge Cording, and Inge Li Gørtz. Compressed subsequence matching and packed tree coloring. *Algorithmica*, 77(2):336–348, 2017. URL: https://doi.org/10.1007/s00453-015-0068-9, doi:10.1007/s00453-015-0068-9.

[4] Philip Bille, Finn Fernstrøm, and Inge Li Gørtz. Tight bounds for top tree compression. In *Proceedings of the 24th International Symposium on String Processing and Information Retrieval, SPIRE 2017*, volume 10508 of *Lecture Notes in Computer Science*, pages 97–102. Springer, 2017.

[5] Philip Bille, Paweł Gawrychowski, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. Top tree compression of tries. In Pinyan Lu and Guochuan Zhang, editors, *Proceedings of the 30th International Symposium on Algorithms and Computation, ISAAC 2019*, volume 149 of *LIPIcs*, pages 4:1–4:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. URL: https://doi.org/10.4230/LIPIcs.ISAAC.2019.4, doi:10.4230/LIPIcs.ISAAC.2019.4.

[6] Philip Bille, Inge Li Gørtz, Patrick Hagge Cording, Benjamin Sach, Hjalte Wedel Vildhøj, and Søren Vind. Fingerprints in compressed strings. *Journal of Computer and System Sciences*, 86:171–180, 2017.

[7] Philip Bille, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. Tree compression with top trees. *Information and Computation*, 243:166–177, 2015.

[8] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.

[9] Mikołaj Bojańczyk and Igor Walukiewicz. Forest algebras. In *Proceedings of Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas].*, volume 2 of *Texts in Logic and Games*, pages 107–132. Amsterdam University Press, 2008.

[10] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Efficient memory representation of XML document trees. *Information Systems*, 33(4–5):456–474, 2008.

[11] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[12] Richard Cole and Uzi Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.

[13] Bartłomiej Dudek and Paweł Gawrychowski. Slowing down top trees for better worst-case compression. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching, CPM 2018*, volume 105 of *LIPIcs*, pages 16:1–16:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[14] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[15] Moses Ganardi, Danny Hucke, Artur Jeż, Markus Lohrey, and Eric Noeth. Constructing small tree grammars and small circuits for formulas. *Journal of Computer and System Sciences*, 86:136–158, 2017. URL: http://dx.doi.org/10.1016/j.jcss.2016.12.007.

[16] Adrià Gascón, Markus Lohrey, Sebastian Maneth, Carl Philipp Reh, and Kurt Sieber. Grammar-based compression of unranked trees. In *Proceedings of 13th International Computer Science Symposium in Russia, CSR 2018*, volume 10846 of *Lecture Notes in Computer Science*, pages 118–131. Springer, 2018.

[17] Moses Ganardi and Markus Lohrey. A universal tree balancing theorem. *ACM Transaction on Computation Theory*, 11(1):1:1–1:25, October 2018.

[18] Moses Ganardi, Markus Lohrey, and Artur Jeż. Balancing Straight-Line Programs. In *Proceedings of 60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019*, Baltimore, Maryland, USA, November 9-12, 2019, pages 1169–1183. IEEE Computer Society, 2019.

[19] Leszek Gasieniec, Roman M. Kolpakov, Igor Potapov, and Paul Sant. Real-time traversal in grammar-based compressed files. In *Proceedings of the 2005 Data Compression Conference, DCC 2005*, page 458. IEEE Computer Society, 2005.

[20] Paweł Gawrychowski, Seungbum Jo, Shay Mozes, and Oren Weimann. Compressed range minimum queries. *Theoretical Computer Science*, 812:39–48, 2020. URL: https://doi.org/10.1016/j.tcs.2019.07.002, `doi:10.1016/j.tcs.2019.07.002`.

[21] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[22] Lorenz Hübschle-Schneider and Rajeev Raman. Tree compression with top trees revisited. In *Proceedings of the 14th International Symposium on Experimental Algorithms, SEA 2015*, volume 9125 of *Lecture Notes in Computer Science*, pages 15–27. Springer, 2015.

[23] Tomohiro I, Wataru Matsubara, Kouji Shimohira, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, Kazuyuki Narisawa, and Ayumi Shinohara. Detecting regularities on grammar-compressed strings. *Information and Computation*, 240:74–89, 2015.

[24] Artur Jeż. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015. URL: https://doi.org/10.1016/j.tcs.2015.05.027, `doi:10.1016/j.tcs.2015.05.027`.

[25] Artur Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11(3):20:1–20:43, Jan 2015. URL: http://doi.acm.org/10.1145/2631920, `doi:10.1145/2631920`.

[26] Artur Jeż. A *really* simple approximation of smallest grammar. *Theoretical Computer Science*, 616:141–150, 2016. URL: http://dx.doi.org/10.1016/j.tcs.2015.12.032, `doi:10.1016/j.tcs.2015.12.032`.

[27] S. Rao Kosaraju. On parallel evaluation of classes of circuits. In *Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 1990*, volume 472 of *Lecture Notes in Computer Science*, pages 232–237. Springer, 1990.

[28] Yury Lifshits. Processing compressed texts: A tractability border. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching, CPM 2007*, volume 4580 of *Lecture Notes in Computer Science*, pages 228–240. Springer, 2007. URL: http://dx.doi.org/10.1007/978-3-540-73437-6_24, `doi:10.1007/978-3-540-73437-6_24`.

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

[30] Markus Lohrey. *The Compressed Word Problem for Groups*. SpringerBriefs in Mathematics. Springer, 2014.

[31] Markus Lohrey. Grammar-based tree compression. In *Proceedings of the 19th International Conference on Developments in Language Theory, DLT 2015*, volume 9168 of *Lecture Notes in Computer Science*, pages 46–57. Springer, 2015.

[32] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. XML tree structure compression using RePair. *Information Systems*, 38(8):1150–1167, 2013.

[33] Markus Lohrey, Sebastian Maneth, and Carl Philipp Reh. Constant-time tree traversal and subtree equality check for grammar-compressed trees. *Algorithmica*, 80(7):2082–2105, 2018.

[34] Gary L. Miller and Shang-Hua Teng. Tree-based parallel algorithm design. *Algorithmica*, 19(4):369–389, 1997.

[35] Mike Paterson and Leslie G. Valiant. Circuit size is nonlinear in depth. *Theoretical Computer Science*, 2(3):397–400, 1976.

[36] Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1–3):211–222, 2003.

[37] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[38] Leslie G. Valiant, Sven Skyum, S. Berkowitz, and Charles Rackoff. Fast parallel computation of polynomials using few processors. *SIAM Journal on Computing*, 12(4):641–644, 1983.

[39] Elad Verbin and Wei Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching, CPM 2013*, volume 7922 of *Lecture Notes in Computer Science*, pages 247–258. Springer, 2013.