Approximation of smallest linear tree grammar

Artur Jeż [1]

Institute of Computer Science, University of Wrocław ul. Joliot-Curie 15, 50-383 Wrocław, Poland

Markus Lohrey

University of Siegen, Department of Electrical Engineering and Computer Science, 57068 Siegen, Germany

Abstract

A simple linear-time algorithm for constructing a linear context-free tree grammar of size $\mathcal{O}(rg + rg\log(n/rg))$ for a given input tree T of size n is presented, where g is the size of a minimal linear context-free tree grammar for T, and r is the maximal rank of symbols in T (which is a constant in many applications). This is the first example of a grammar-based tree compression algorithm with a good, i.e. logarithmic in terms of the size of the input tree, approximation ratio. The analysis of the algorithm uses an extension of the recompression technique from strings to trees.

Keywords: grammar-based compression, tree compression, tree grammars

1. Introduction

Grammar-based compression has emerged to an active field in string compression during the last decade. The idea is to represent a given string s by a small context-free grammar that generates only s; such a grammar is also called a *straight-line program*, briefly SLP. For instance, the word $(ab)^{1024}$ can be represented by the SLP with the productions $A_0 \to ab$ and $A_i \to A_{i-1}A_{i-1}$ for $1 \leq i \leq 10$ (A_{10} is the start symbol). The size of this grammar is much smaller than the size (length) of the string $(ab)^{1024}$. In general, an SLP of size n (the size of an SLP is usually defined as the total length of all right-hand sides of productions) can produce a string of length $2^{\Omega(n)}$. Hence, an SLP can be seen as the succinct representation of the generated word. The principle task of grammar-based string compression is to construct, from a given input string s, a small SLP that generates s. Unfortunately, finding a minimal (with respect to size) SLP for a given input string is not achievable in polynomial time, unless $\mathbf{P} = \mathbf{NP}$ [1] (recently the same result was shown also in case of a constant-size alphabet [2]). Therefore, one can concentrate either on heuristic grammar-based compressors [3, 4, 5], or

[1]The first author was supported by the National Science Centre, Poland project number 2014/15/B/ST6/00615. The second author was supported by the German Research Foundation, project number LO 748/10-1.

Email addresses: aje@cs.uni.wroc.pl (Artur Jeż), lohrey@eti.uni-siegen.de (Markus Lohrey)

compressors whose output SLP is guaranteed to be not much larger than a size-minimal SLP for the input string [6, 7, 8, 9, 10]. In this paper we are interested mostly in the latter approach. Formally, in [6] the approximation ratio for a grammar-based compressor $\mathcal{G}$ is defined as the function $\alpha_{\mathcal{G}}$ with

$$\alpha_{\mathcal{G}}(n) = \max \frac{\text{size of the SLP produced by } \mathcal{G} \text{ with input } x}{\text{size of a minimal SLP for } x},$$

where the maximum is taken over all strings of length n (over an arbitrary alphabet). The above statement means that unless $\mathbf{P} = \mathbf{NP}$ there is no polynomial time grammar-based compressor with the approximation ratio 1. Using approximation lower bounds for computing vertex covers, it is shown in [6] that unless $\mathbf{P} = \mathbf{NP}$ there is no polynomial time grammar-based compressor, whose approximation ratio is less than the constant 8569/8568.

Apart from this complexity theoretic bound, the authors of [6] prove lower and upper bounds on the approximation ratios of well-known grammar-based string compressors (LZ78, BISECTION, SEQUENTIAL, RePair, etc.). The currently best known approximation ratio of a polynomial time grammar-based string compressor is of the form $\mathcal{O}(\log(n/g))$, where g is the size of a smallest SLP for the input string. Actually, there are several compressors achieving this approximation ratio [6, 7, 8, 9, 10] and each of them works in linear time (a property that a reasonable compressor should have).

At this point, the reader might ask, what makes grammar-based compression so attractive. There are actually several arguments in favour of grammar-based compression:

- The output of a grammar-based compressor is a clean and simple object, which may simplify the analysis of a compressor or the analysis of algorithms that work on compressed data; see [11] for a survey.
- There are grammar-based compressors which achieve very good compression ratios. For example RePair [4] performs very well in practice and was for instance used for the compression of web graphs [12].
- The idea of grammar-based string compression can be generalised to other data types as long as suitable grammar formalisms are known for them. See for instance the recent work on grammar-based graph compression [13].

The last point is the most important one for this work. In[14], grammar-based compression was generalised from strings to trees.[3] For this, context-free tree grammars were used. Context free tree grammars that produce only a single tree are also known as straight-line context-free tree grammars (SLCF tree grammars). Several papers deal with algorithmic problems on trees that are succinctly represented by SLCF tree grammars [15, 16, 17, 18, 19, 20]. In [21], RePair was generalised from strings to trees, and the resulting algorithm TreeRePair achieves excellent results on real XML trees. Other grammar-based tree compressors

[3] A tree in this paper is always a rooted ordered tree over a ranked alphabet, i.e., every node is labelled with a symbol and the rank of this symbol is equal to the number of children of the node.

were developed in [22, 23], but none of these compressors has a good approximation ratio. For instance, in [21] a series of trees is constructed, where the n-th tree t_n has size $\Theta(n)$, there exists an SLCF tree grammar for t_n of size $\mathcal{O}(\log n)$, but the grammar produced by TreeRePair for t_n has size $\Omega(n)$ (and similar examples can be constructed for the compressors in [22, 14]).

In this paper, we give the first example of a grammar-based tree compressor TtoG (for "tree to grammar") with an approximation ratio of $\mathcal{O}(\log(n/g))$ assuming the maximal rank r of symbols is bounded and where g denotes the size of the smallest grammar generating the given tree; otherwise the approximation ratio becomes $\mathcal{O}(r + r\log(n/gr))$. Our algorithm TtoG is based on the work [7] of the first author, where another grammar-based string compressor with an approximation ratio of $\mathcal{O}(\log(n/g))$ is presented (here g denotes the size of the smallest grammar for the input string). The remarkable fact about this latter compressor is that in contrast to [6, 8, 9, 10] it does not use the LZ77 factorization of a string (which makes the compressors from [6, 8, 9, 10] not suitable for a generalization to trees, since LZ77 ignores the tree structure and no good analogue of LZ77 for trees is known), but is based on the *recompression technique*. This technique was introduced in [24] and successfully applied for a variety of algorithmic problems for SLP-compressed strings [24, 25] and word equations [26, 27, 28]. The basic idea is to compress a string using two operations:

- block compressions: replace every maximal substring of the form a^ℓ for a letter a by a new symbol a_ℓ;
- pair compression: for a given partition $\Sigma_\ell \uplus \Sigma_r$ replace every substring $ab \in \Sigma_\ell\Sigma_r$ by a new symbol c.

It can be shown that the composition of block compression followed by pair compression (for a suitably chosen partition of the input letters) reduces the length of the string by a constant factor. Hence, the iteration of block compression followed by pair compression yields a string of length one after a logarithmic number of phases. By reversing a single compression step, one obtains a grammar rule for the introduced letter and thus reversing all such steps yields an SLP for the initial string. The term "recompression" refers to the fact, that for a given SLP $\mathbb{G}$, block compression and pair compression can be simulated on $\mathbb{G}$. More precisely, one can compute from $\mathbb{G}$ a new SLP $\mathbb{G}'$, which is not much larger than $\mathbb{G}$ such that $\mathbb{G}'$ produces the result of block compression (respectively, pair compression) applied to the string produced by $\mathbb{G}$. In [7], the recompression technique is used to bound the approximation ratio of the above compression algorithm based on block and pair compression.

In this work we generalise the recompression technique from strings to trees. The operations of block compression and pair compression can be directly applied to chains of unary nodes (nodes having only a single child) in a tree. But clearly, these two operations alone cannot reduce the size of the initial tree by a constant factor. Hence we need a third compression operation that we call *leaf compression*. It merges all children of a node that are leaves into the node. The new label of the node determines the old label, the sequence of labels of the children that are leaves, and their positions in the sequence of all children of the node. Then, one can show that a single phase, consisting of block compression (that we

call chain compression), followed by pair compression (that we call unary pair compression), followed by leaf compression reduces the size of the initial tree by a constant factor. As for strings, we obtain an SLCF tree grammar for the input tree by reversing the sequence of compression operations. The recompression approach again yields an approximation ratio of $\mathcal{O}(\log(n/g))$ (assuming that the maximal rank of symbols is a constant) for our compression algorithm TtoG, but the analysis is technically more subtle.

Theorem 1. *The algorithm* TtoG *runs in linear time, and for a tree T of size n, it returns an SLCF tree grammar of size $\mathcal{O}(gr + gr\log(n/gr))$, where g is the size of a smallest SLCF grammar for T and r is the maximal rank of a symbol in T.*

Note that in some specific cases it could happen that $n < gr$ and so the term $\log(n/gr)$ is in fact negative, we follow the usual practice of bounding the logarithm from below by 0, i.e. in such a case we assign 0 as the value of the logarithm.

Related work on grammar-based tree compression. We already mentioned that grammar-based tree compressors were developed in [14, 21, 22], but none of these compressors has a good approximation ratio. Another grammar-based tree compressors was presented in [29]. It is based on the BISECTION algorithm for strings and has an approximation ratio of $\mathcal{O}(n^{5/6})$. But this algorithm uses a different form of grammars (elementary ordered tree grammars) and it is not clear whether the results from [29] can be extended to SLCF tree grammars, or whether the good algorithmic results for SLCF-compressed trees [16, 17, 18, 19, 20] can be extended to elementary ordered tree grammars. Let us also mention the work from [30] where trees are compressed by so called top dags. These are another hierarchical representation of trees. Upper bounds on the size of the minimal top dag are derived in [30] and compared with the size of the minimal dag (directed acyclic graph). More precisely, it is shown in [30, 31] that the size of the minimal top dag is at most by a factor of $\mathcal{O}(\log n)$ larger than the size of the minimal dag. Since dags can be seen as a special case of SLCF tree grammars, our main result is stronger. In [23], the worst case size of the output grammar of grammar-based tree compressors was investigated and an algorithm that always returns an SLCF tree grammar of size $\mathcal{O}(\frac{n}{\log_\sigma n})$ was given, where σ is the size of the input alphabet. In fact this algorithm can be implemented in linear time or in logarithmic space [32]. Note that (up to constant factors) the upper bound $\mathcal{O}(\frac{n}{\log_\sigma n})$ matches the information-theoretic lower bound. Slightly weaker results were obtained for the already mentioned top dags: it was shown that top dags have size at most $\mathcal{O}(\frac{n}{\log_\sigma n}\log\log n)$ [31]. Finally, the performance of grammar-based string compression of trees that are encoded by preorder traversal string was compared with grammar-based tree compresson [33]: the smallest string SLP for the preorder traversal of a tree can be exponentially smaller than the smallest SLCF tree grammar for the same tree. But on a downside there are queries that can be efficiently (in **P**) computed, when trees are represented by SLCF tree grammars, but become **PSPACE**-complete, when trees are represented by string SLPs.

Other applications of the technique: context unification. The recompression method can be applied to word equations and it is natural to hope that its generalization to trees also applies to appropriate generalizations of word equations. Indeed, the tree recompression approach is used in [34] to show that the context unification problem can be solved in **PSPACE**. It was a long standing open problem whether context unification is decidable [35].

Parallel tree contraction. Our compression algorithm is similar to algorithms for parallel tree evaluation [36, 37]. Here, the problem is to evaluate an algebraic expression of size n in time $\mathcal{O}(\log n)$ on a PRAM. Using parallel tree contraction, this can be achieved on a PRAM with $\mathcal{O}(n/\log n)$ many processors. The rake operation in parallel tree contraction is the same as our leaf compression operation, whereas the *compress operations* contracts chains of unary nodes and hence corresponds to block compression and pair compression. On the other hand, the specific features of block compression and pair compression that yield the approximation ratio of $\mathcal{O}(\log(n/g))$ have no counterpart in parallel tree contraction.

Computational model. To achieve the linear running time we need some assumption on the computational model and form of the input. We assume that numbers of $\mathcal{O}(\log n)$ bits (where n is the size of the input tree) can be manipulated in time $\mathcal{O}(1)$ and that the labels of the input tree come from an interval $[1, \ldots, n^c]$, where c is some constant. Those assumption are needed so that we can employ RadixSort, which sorts m many k-ary numbers of length ℓ in time $\mathcal{O}(\ell m + \ell k)$, see e.g. [38, Section 8.3]. In fact, we need a slightly more powerful version of RadixSort that sorts lexicographically m sequences of digits from $[1, \ldots, k]$ of lengths $\ell_1, \ell_2, \ldots, \ell_m$ in time $\mathcal{O}(k + \sum_{i=1}^{m} \ell_i)$. This is a standard generalisation of RadixSort [39, Theorem 3.2]. If for any reason the labels do not belong to an interval $[1, \ldots, n^c]$, we can sort them in time $\mathcal{O}(n \log n)$ and replace them with numbers from $\{1, 2, \ldots, n\}$.

2. Preliminaries

2.1. Trees

Let us fix for every $i \geq 0$ a countably infinite set $\mathbb{F}_i$ of *letters* (or *symbols*) of rank i, where $\mathbb{F}_i \cap \mathbb{F}_j = \emptyset$ for $i \neq j$, and let $\mathbb{F} = \bigcup_{i \geq 0} \mathbb{F}_i$. Symbols in $\mathbb{F}_0$ are called *constants*, while symbols in $\mathbb{F}_1$ are called *unary letters*. We also write $\mathrm{rank}(a) = i$ if $a \in \mathbb{F}_i$. A *ranked alphabet* is a finite subset of $\mathbb{F}$. Let F be a ranked alphabet. We also write F_i for $F \cap \mathbb{F}_i$ and $F_{\geq i}$ for $\bigcup_{j \geq i} F_i$. An *F-labelled tree* is a rooted, ordered tree whose nodes are labelled with elements from F, satisfying the condition that if a node v is labelled with a then it has exactly $\mathrm{rank}(a)$ children, which are linearly ordered (by the usual left-to-right order). We denote by $\mathcal{T}(F)$ the set of F-labelled trees. In the following we simply speak about trees when the ranked alphabet is clear from the context or unimportant. When useful, we identify an F-labelled tree with a term over F in the usual way. The size of the tree t is its number of nodes and is denoted by $|t|$. We assume that a tree is given using a pointer representation, i.e., each node has a list of its children (ordered from left to right) and each node (except for the root) has a pointer to its parent node.

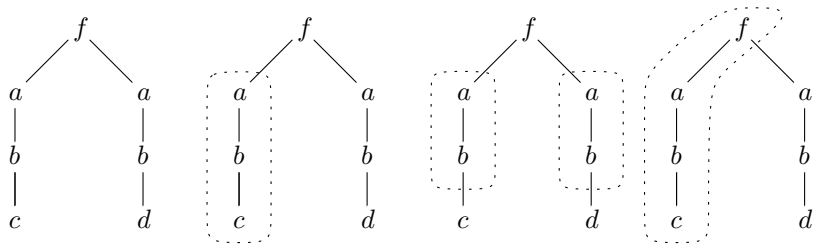

Figure 1: The tree $f(a(b(c)), a(b(d)))$, its subpattern $a(b(c))$, subpatterns $a(b(y))$, and subpattern $f(a(b(c)), y)$ in which a is the first child of f.

Fix a countable set $\mathbb{Y}$ with $\mathbb{Y} \cap \mathbb{F} = \emptyset$ of *(formal) parameters*, which are usually denoted by $y, y_1, y_2, \ldots$. For the purposes of building trees with parameters, we treat all parameters as constants, and so F-labelled trees with parameters from $Y \subseteq \mathbb{Y}$ (where Y is finite) are simply $(F \cup Y)$-labelled trees, where the rank of every $y \in Y$ is 0. However, to stress the special role of parameters we write $\mathcal{T}(F, Y)$ for the set of F-labelled trees with parameters from Y. We identify $\mathcal{T}(F)$ with $\mathcal{T}(F, \emptyset)$. In the following we talk about *trees with parameters* (or even trees) when the ranked alphabet and parameter set is clear from the context or unimportant. The idea of parameters is best understood when we represent trees as terms: For instance $f(y_1, a, y_2, y_1)$ with parameters y_1 and y_2 can be seen as a term with variables y_1, y_2 and we can instantiate those variables later on. A *pattern* (or *linear tree*) is a tree $t \in \mathcal{T}(F, Y)$, that contains for every $y \in Y$ at most one y-labelled node. Clearly, a tree without parameters is a pattern. All trees in this paper are patterns, and we do not mention this assumption explicitly in the following.

When we talk of a *subtree* u of a tree t, we always mean a full subtree in the sense that for every node of u all children of that node in t belong to u as well. In contrast, a *subpattern* v of t is obtained from a subtree u of t by removing some of the subtrees of u. If we replace these subtrees by pairwise different parameters, then we obtain a pattern $p(y_1, \ldots, y_n)$ and we say that *(i)* the subpattern v is an *occurrence* of the pattern $p(y_1, \ldots, y_n)$ in t and *(ii)* $p(y_1, \ldots, y_n)$ is the pattern corresponding to the subpattern v (this pattern is unique up to renaming of parameters). This later terminology applies also to subtrees, since a subtree is a subpattern as well. A *context* $c(y)$ is a pattern with exactly one parameter, and occurrences of a context $c(y)$ in a tree are called *subcontexts*. To make this notions clear, consider for instance the tree $f(a(b(c)), a(b(d)))$ with $f \in \mathbb{F}_2$, $a, b \in \mathbb{F}_1$ and $c, d \in \mathbb{F}_0$. It contains one occurrence of the pattern (in fact, tree) $a(b(c))$, two occurrences of the pattern (in fact, context) $a(b(y))$ and one of the pattern (in fact, context) $f(a(b(c)), y)$, see Figure 1.

A *chain pattern* is a context of the form $a_1(a_2(\ldots(a_k(y))\ldots))$ with $a_1, a_2, \ldots, a_k \in \mathbb{F}_1$. A *chain* in a tree t is an occurrence of a chain pattern in t. A chain s in t is *maximal* if there is no chain s' in t with $s \subsetneq s'$. A 2-chain is a chain consisting of only two nodes (which, most of the time, are labelled with different letters). For $a \in \mathbb{F}_1$, an a-maximal chain is a chain such that (i) all nodes are labelled with a and (ii) there is no chain s' in t such that $s \subsetneq s'$ and all nodes of s' are labelled with a too. Note that an a-maximal chain is not necessarily a maximal chain. Consider for instance the tree $b(a(a(a(c))))$. The unique occurrence of

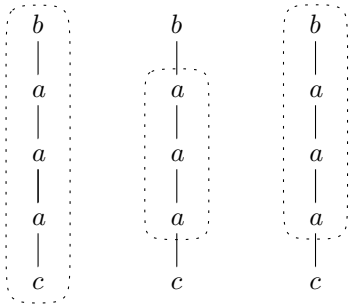

Figure 2: The tree $b(a(a(a(c))))$, its unique a-maximal chain (an occurrence of $a(a(a(y)))$) and its unique maximal chain (an occurrence of $b(a(a(a(y))))$).

the chain pattern $a(a(a(y)))$ is an a-maximal chain, but is not maximal. The only maximal chain is the unique occurrence of the chain pattern $b(a(a(a(y))))$, see Figure 2.

We write $a_1a_2\cdots a_k$ for the chain pattern $a_1(a_2(\ldots(a_k(y))\ldots))$ and treat it as a string (even though this 'string' still needs an argument on its right to form a proper term). In particular, we write a^ℓ for the chain pattern consisting of ℓ many a-labelled nodes and we write vw (for chain patterns v and w) for what should be $v(w(y))$.

2.2. SLCF tree grammars

For the further considerations, fix a countable infinite set $\mathbb{N}_i$ of symbols of rank i with $\mathbb{N}_i \cap \mathbb{N}_j = \emptyset$ for $i \neq j$. Let $\mathbb{N} = \bigcup_{i\geq 0} \mathbb{N}_i$. Furthermore, assume that $\mathbb{F} \cap \mathbb{N} = \emptyset$. Hence, every finite subset $N \subseteq \mathbb{N}$ is a ranked alphabet. A *linear context-free tree grammar*, *linear CF tree grammar* for short, [4] is a tuple $\mathbb{G} = (N, F, P, S)$ such that the following conditions hold:

(1) $N \subseteq \mathbb{N}$ is a finite set of *nonterminals*.

(2) $F \subseteq \mathbb{F}$ is a finite set of *terminals*.

(3) P (the set of *productions*) is a finite set of pairs (A, t) (for which we write $A \to t$), where $A \in N$ and $t \in \mathcal{T}(F \cup N, \{y_1, \ldots, y_{\mathsf{rank}(A)}\})$ is a pattern, which contains exactly one y_i-labelled node for each $1 \leq i \leq \mathrm{rank}(A)$.

(4) $S \in N$ is the *start nonterminal*, which is of rank 0.

To stress the dependency of A on its parameters we sometimes write $A(y_1, \ldots, y_{\mathrm{rank}(A)}) \to t$ instead of $A \to t$. Without loss of generality we assume that every nonterminal $B \in N \setminus \{S\}$ occurs in the right-hand side t of some production $(A \to t) \in P$ (a much stronger fact is shown in [18, Theorem 5]).

A linear CF tree grammar $\mathbb{G}$ is *k-bounded* (for a natural number k) if $\mathsf{rank}(A) \leq k$ for every $A \in N$. Moreover, $\mathbb{G}$ is *monadic* if it is 1-bounded. The derivation relation $\Rightarrow_{\mathbb{G}}$ on $\mathcal{T}(F \cup N, Y)$ is defined as follows: $s \Rightarrow_{\mathbb{G}} s'$ if and only if there is a production $(A(y_1, \ldots, y_\ell) \to$

[4] There exist also non-linear CF tree grammars, which we do not need for our purpose.

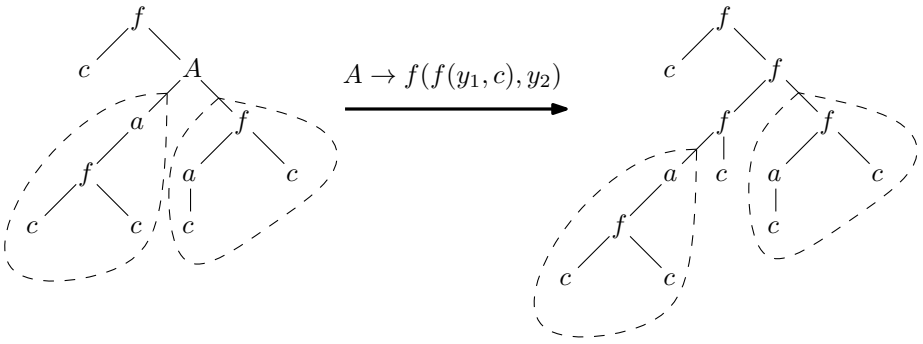

Figure 3: The tree $f(c, A(a(f(c, c)), f(a(c), c)))$ and the result of applying the rule $A(y_1, y_2) \to f(f(y_1, c), y_2)$. The subtrees that are substituted for parameters are within dashed blobs.

$t) \in P$ such that s' is obtained from s by replacing some subtree $A(t_1, \ldots, t_\ell)$ of s by t with each y_i replaced by t_i. Intuitively, we replace an A-labelled node by the pattern $t(y_1 \ldots, y_{\mathrm{rank}(A)})$ and thereby identify the j-th child of A with the unique y_j-labelled node of the pattern, see Figure 3. Then $L(\mathbb{G})$ is the set of all trees from $\mathcal{T}(F)$ (so F-labelled without parameters) that can be derived from S (in arbitrarily many steps).

A *straight-line context-free tree grammar* (or *SLCF grammar* for short) is a linear CF tree grammar $\mathbb{G} = (N, F, P, S)$, where

- for every $A \in N$ there is *exactly one* production $(A \to t) \in P$ with left-hand side A,
- if $(A \to t) \in P$ and B occurs in t then $B < A$, where $<$ is a linear order on N, and
- S is the maximal nonterminal with respect to $<$.

By the first two conditions, every $A \in N$ derives exactly one tree from $\mathcal{T}(F, \{y_1, \ldots, y_{\mathsf{rank}(A)}\})$. We denote this tree by $\mathrm{val}(A)$ (like *valuation*). Moreover, we define $\mathrm{val}(\mathbb{G}) = \mathrm{val}(S)$, which is a tree from $\mathcal{T}(F)$. In fact, every tree from $\mathcal{T}(F \cup N, Y)$ derives a unique tree from $\mathcal{T}(F, Y)$, where Y is an arbitrary finite set of parameters. For an SLCF grammar $\mathbb{G} = (N, F, P, S)$ we can assume without loss of generality that for every production $(A \to t) \in P$ the parameters $y_1, \ldots, y_{\mathrm{rank}(A)}$ occur in t in the order $y_1, y_2, \ldots, y_{\mathrm{rank}(A)}$ from left to right. This can be ensured by a simple bottom-up rearranging procedure, see [18, proof of Theorem 5]. In the rest of the paper, when we speak of grammars, we always mean SLCF grammars.

2.3. Grammar size

When defining the *size* $|\mathbb{G}|$ of the SLCF grammar $\mathbb{G}$, one possibility is $|\mathbb{G}| = \sum_{(A \to t) \in P} |t|$, i.e., the sum of all sizes of all right-hand sides. However, consider for instance the rule $A(y_1, \ldots, y_\ell) \to f(y_1, \ldots, y_{i-1}, a, y_i, \ldots, y_\ell)$. It is in fact enough to describe the right-hand side as $(f, (i, a))$, as we have a as the i-th child of f. On the remaining positions we just list the parameters, whose order is known to us (see the remark in the previous paragraph). In general, each right-hand side of $\mathbb{G}$ can be specified by listing for each node its children that are *not* parameters together with their positions in the list of all children. These positions are numbers between 1 and r (it is easy to show that our algorithm TtoG creates only

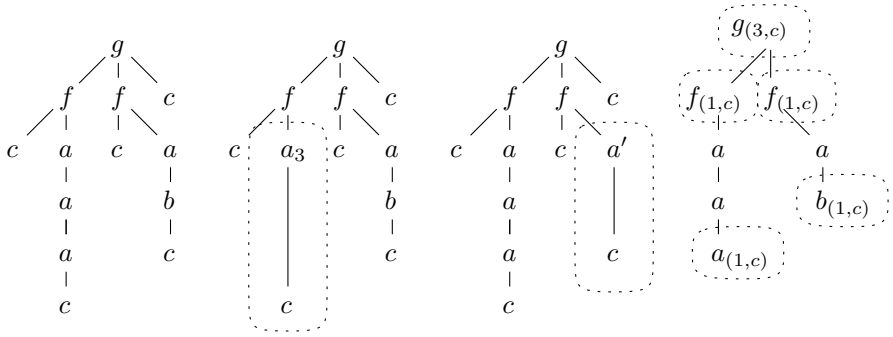

Figure 4: A tree and the results of compression operations: a-chain compression, ab-compression, and (parallel) leaf compression.

nonterminals of rank at most r, see Lemma 1, and hence every node in a right-hand side has at most r children) and therefore fit into $\mathcal{O}(1)$ machine words. For this reason we define the size $|\mathbb{G}|$ as the total number of non-parameter nodes in all right-hand sides. Note that such an approach is well-established; see for instance [14].

Should the reader prefer to define the size of a grammar as the total number of all nodes (including parameters) in all right-hand sides, then the approximation ratio of our algorithm TtoG has to be multiplied with the additional factor r.

2.4. Notational conventions

Our compression algorithm TtoG takes the input tree and applies to it local compression operations, each such operation decreases the size of the tree. With T we always denote the current tree stored by TtoG, whereas n denotes the size of the initial input tree. The algorithm TtoG relabels the nodes of the tree with fresh letters. With F we always denote the set of letters occurring in the current tree T. By r we denote the maximal rank of the letters occurring in the initial input tree. The ranks of the fresh letters do not exceed r.

2.5. Compression operations

Our compression algorithm TtoG is based on three local replacement rules applied to trees:

(a) a-maximal chain compression (for a unary symbol a),

(b) unary pair compression,

(c) and leaf compression.

Operations (a) and (b) apply only to unary letters and are direct translations of the operations used in the recompression-based algorithm for constructing a grammar for a given string [7]. To be more precise, (a) and (b) affect only chains, return chains as well, and when a chain is treated as a string the results of (a) and (b), respectively, correspond to the results of the corresponding operations on strings. On the other hand, the last operations (c) is new and designed specifically to deal with trees. Let us inspect these operations:

a-maximal chain compression. For a unary letter a replace every a-maximal chain consisting of $\ell > 1$ nodes with a fresh unary letter a_ℓ (for all $\ell > 1$).

(a, b)-pair compression. For two unary letters $a \neq b$ replace every occurrence of ab by a single node labelled with a fresh unary letter c (which identifies the pair (a, b)).

$(f, i_1, a_1, \ldots, i_\ell, a_\ell)$-leaf compression:. For $f \in F_{\geq 1}$, $\ell \geq 1$, $a_1, \ldots, a_\ell \in F_0$ and $0 < i_1 < i_2 < \cdots < i_\ell \leq \mathrm{rank}(f) =: m$ replace every occurrence of $f(t_1, \ldots, t_m)$, where $t_{i_j} = a_j$ for $1 \leq j \leq \ell$ and t_i is a non-constant for $i \notin \{i_1, \ldots, i_\ell\}$, with $f'(t_1, \ldots, t_{i_1-1}, t_{i_1+1}, \ldots, t_{i_\ell-1}, t_{i_\ell+1}, \ldots, t_m)$, where f' is a fresh letter of rank $\mathrm{rank}(f) - \ell$ (which identifies $(f, i_1, a_1, \ldots, i_\ell, a_\ell)$).

Note that each of these operations decreases the size of the current tree. Also note that for each of these compression operations one has to specify some arguments: for chain compression the unary letter a, for unary pair compression the unary letters a and b, and for leaf compression the letter f (of rank at least 1) as well as the list of positions $i_1 < i_2 < \cdots < i_\ell$ and the constants $a_1, \ldots, a_\ell$.

Despite its rather cumbersome definition, the idea behind leaf compression is easy: For a fixed occurrence of f in a tree we 'absorb' all leaf-children of f that are constants (and do the same for all other occurrences of f that have the same set of leaf-children on the same positions).

Every application of one of our compression operations can be seen as the 'backtracking' of a production of the grammar that we construct: When we replace a^ℓ by a_ℓ, we in fact introduce the new nonterminal $a_\ell(y)$ with the production

$$a_\ell(y) \to a^\ell(y). \tag{1}$$

When we replace all occurrences of the chain ab by c, the new production is

$$c(y) \to a(b(y)). \tag{2}$$

Finally, for a $(f, i_1, a_1 \ldots, i_\ell, a_\ell)$-leaf compression the production is

$$f'(y_1, \ldots, y_{\mathrm{rank}(f)-\ell}) \to f(t_1, \ldots, t_{\mathrm{rank}(f)}), \tag{3}$$

where $t_{i_j} = a_j$ for $1 \leq j \leq \ell$ and every t_i with $i \notin \{i_1, \ldots, i_\ell\}$ is a parameter (and the left-to-right order of the parameters in the right-hand side is $y_1, \ldots, y_{\mathrm{rank}(f)-\ell}$).

Observe that all productions introduced in (1)–(3) are for nonterminals of rank at most r.

Lemma 1. *The rank of nonterminals defined by* TtoG *is at most r.*

During the analysis of the approximation ratio of TtoG we also consider the nonterminals of a smallest grammar generating the given input tree. To avoid confusion between these nonterminals and the nonterminals of the grammar produced by TtoG, we insist on calling the fresh symbols introduced by TtoG (a_ℓ, c, and f' in (1)–(3)) *letters* and add them to the

set F of current letters, so that F always denotes the set of letters in the current tree. In particular, whenever we talk about nonterminals, productions, etc. we mean the ones of the smallest grammar we consider.

Still, the above rules (1), (2), and (3) form the grammar returned by our algorithm TtoG and we need to estimate their size. In order to not mix the notation, we call the size of the rule for a new letter a the *representation cost* for a and say that a *represents* the subpattern it replaces in T. For instance, the representation cost of a_ℓ in (1) is ℓ, the representation cost of c in (2) is 2, and the representation cost of f' in (3) is $\ell + 1$. A crucial part of the analysis of TtoG is the reduction of the representation cost for a_ℓ. Note that instead of representing $a^\ell(y)$ directly via the rule (1), we can introduce new unary letters representing some shorter chains in a^ℓ and build a longer chains using the smaller ones as building blocks. For instance, the rule $a_8(y) \to a^8(y)$ can be replaced by the rules $a_8(y) \to a_4(a_4(y))$, $a_4(y) \to a_2(a_2(y))$ and $a_2(y) \to a(a(y))$. This yields a total representation cost of 6 instead of 8. Our algorithm employs a particular strategy for representing a-maximal chains. Slightly abusing the notation we say that the sum of the sizes of the right-hand sides of the generated subgrammar is the representation cost for a_ℓ (for our strategy).

2.6. Parallel compression

The important property of the compression operations is that we can perform many of them in parallel: Since different a-maximal chains and b-maximal chains do not overlap (regardless of whether $a = b$ or not) we can perform a-maximal chain compression for all $a \in F_1$ in parallel (assuming that the new letters do not belong to F_1). This justifies the following compression procedure for compression of all a-maximal chains (for all $a \in F_1$) in a tree t:

Algorithm 1 TreeChainComp(F_1, t): Compression of chains of letters from F_1 in a tree t

```
for a ∈ F_1 do                                                  ▷ chain compression
    for ℓ ← 1 .. |t| do
        replace every a-maximal chain of size ℓ by a fresh letter a_ℓ      ▷ a_ℓ ∉ F_1
```

We refer to the procedure TreeChainComp simply as *chain compression*. The running time of an appropriate implementation is considered in the next section and the corresponding representation cost is addressed in Section 4.

A similar observation applies to leaf compressions: we can perform several different leaf compressions as long as we do not try to compress the letters introduced by these leaf compressions.

Algorithm 2 TreeLeafComp$(F_{\geq 1}, F_0, t)$: leaf compression for parent nodes in $F_{\geq 1}$, and leaf-children in F_0 for a tree t

```
for f ∈ F_{≥1}, 0 < i_1 < i_2 < ··· < i_ℓ ≤ rank(f) =: m, (a_1, a_2, ..., a_ℓ) ∈ F_0^ℓ do
    replace each subtree f(t_1, ..., t_m) s.t. t_{i_j} = a_j for 1 ≤ j ≤ ℓ and t_i ∉ F_0 for
    i ∉ {i_1, ..., i_ℓ} by f'(t_1, ..., t_{i_1−1}, t_{i_1+1}, ..., t_{i_ℓ−1}, t_{i_ℓ+1}, ..., t_m)
                                                                    ▷ f' ∉ F_{≥1} ∪ F_0
```

We refer to the procedure TreeLeafComp as *leaf compression*. An efficient implementation is given in the next section, while the analysis of the number of introduced letters is done in Section 4.

The situation is more subtle for unary pair compression: observe that in a chain abc we can compress ab or bc but we cannot do both in parallel (and the outcome depends on the order of the operations). However, as in the case of string compression [7], parallel (a, b)-pair compressions are possible when we take a and b from disjoint subalphabets F_1^{up} and F_1^{down}, respectively. In this case we can tell for each unary letter whether it should be the parent node or the child node in the compression step and the result does not depend on the order of the considered 2-chains, as long as the new letters do not belong to $F_1^{\text{up}} \cup F_1^{\text{down}}$.

Algorithm 3 TreeUnaryComp$(F_1^{\text{up}}, F_1^{\text{down}}, t)$: $(F_1^{\text{up}}, F_1^{\text{down}})$-compression for a tree t

1: **for** $a \in F_1^{\text{up}}$ and $b \in F_1^{\text{down}}$ **do**
2: replace each occurrence of ab with a fresh unary letter c ▷ $c \notin F_1^{\text{up}} \cup F_1^{\text{down}}$

The procedure TreeUnaryComp is called $(F_1^{\text{up}}, F_1^{\text{down}})$-compression in the following. Again, its efficient implementation is given in the next section and the analysis of the number of introduced letters is done in Section 4.

3. Algorithm

In a single phase of the algorithm TtoG, chain compression, $(F_1^{\text{up}}, F_1^{\text{down}})$-compression and leaf compression are executed in this order (for an appropriate choice of the partition $F_1^{\text{up}}, F_1^{\text{down}}$). The intuition behind this approach is as follows: If the tree t in question does not have any unary letters, then leaf compression on its own reduces the size of t by at least half, as it effectively reduces all constant nodes, i.e., leaves of the tree, and more than half of the nodes are leaves. On the other end of the spectrum is the situation in which all nodes (except for the unique leaf) are labelled with unary letters. In this case our instance is in fact a string. Chain compression and unary pair compression correspond to the operations of block compression and pair compression, respectively, from the earlier work on string compression [7], where it is shown that block compression followed by pair compression reduces the size of the string by a constant factor (for an appropriate choice of the partition $F_1^{\text{up}}, F_1^{\text{down}}$ of the letters occurring in the string). The in-between cases are a mix of those two extreme scenarios and it can be shown that for them the size of the instance drops by a constant factor in one phase as well.

Recall from Section 2.4 that T always denotes the current tree kept by TtoG and that F is the set of letters occurring in T. Moreover, n denotes the size of the input tree.

Algorithm 4 TtoG: Creating an SLCF grammar for the input tree T

while $|T| > 1$ **do**
  $F_1 \leftarrow$ list of unary letters in T
  $T \leftarrow \mathsf{TreeChainComp}(F_1, T)$ ▷ time $\mathcal{O}(|T|)$
  $F_1 \leftarrow$ list of unary letters in T
  compute partition $F_1 = F_1^{\text{up}} \uplus F_1^{\text{down}}$ using the algorithm from Lemma 6 ▷ time $\mathcal{O}(|T|)$
  $T \leftarrow \mathsf{TreeUnaryComp}(F_1^{\text{up}}, F_1^{\text{down}}, T)$ ▷ time $\mathcal{O}(|T|)$
  $F_0 \leftarrow$ list of constants in T, $F_{\geq 1} \leftarrow$ list of other letters in T
  $T \leftarrow \mathsf{TreeLeafComp}(F_{\geq 1}, F_0, T)$ ▷ time $\mathcal{O}(|T|)$
return constructed grammar

A single iteration of the main loop of TtoG is called a *phase*. In the rest of this section we show how to implement TtoG in linear time (a polynomial implementation is straightforward), while in Section 4 we analyse the approximation ratio of TtoG.

Since the compression operations use RadixSort for grouping, it is important that right before such a compression the letters in T form an interval of numbers. As no letters are replaced in the listing of letters preceding such a compression, it is enough to guarantee that after each compression, as a post-processing, letters are replaced so that they form an interval of numbers. Such a post-processing takes linear time.

Lemma 2 (cf. [7, Lemma 1])**.** *After each compression operation performed by* TtoG *we can rename in time $\mathcal{O}(|T'|)$ the letters used in T so that they form an interval of numbers, where T' denotes the tree before the compression step. Furthermore, in the preprocessing step we can, in linear time, ensure the same property for the input tree.*

Proof. Recall that we assume that the input alphabet consists of letters that can be identified with elements from an interval $\{1, \ldots, n^c\}$ for a constant c, see the discussion in the introduction. Treating them as n-ary numbers of length c, we we can sort them using RadixSort in $\mathcal{O}(cn)$ time, i.e., in linear time. Then we can renumber the letters to $1, 2, \ldots, n'$ for some $n' \leq n$. This preprocessing is done once at the beginning.

Fix the compression step and suppose that before the listing preceding this compression the letters formed an interval $[m, \ldots, m + k]$. Each new letter, introduced in place of a compressed subpattern (i.e., a chain a^ℓ, a chain ab or a node f together with some leaf-children) is assigned a consecutive value, and so after the compression the letters occurring in T are within an interval $[m, \ldots, m + k']$ for some $k' > k$, note also that $k' - k \leq \mathcal{O}(|T|) \leq \mathcal{O}(|T'|)$, as each new letter labels a node in T. It is now left to re-number the letters from $[m, \ldots, m + k']$, so that the ones occurring in T indeed form an interval. For each symbol in the interval $[m, \ldots, m + k']$ we set a flag to 0. Moreover, we set a variable *next* to $m + k' + 1$. Then we traverse T (in an arbitrary way). Whenever we spot a letter $a \in [m, \ldots, m + k']$ with $\mathit{flag}[a] = 0$, we set $\mathit{flag}[a] := 1$; $\mathit{new}[a] := \mathit{next}$, and $\mathit{next} := \mathit{next} + 1$. Moreover, we replace the label of the current node (which is a) by $\mathit{new}[a]$. When we spot a symbol $a \in [m, \ldots, m + k']$ with $\mathit{flag}[a] = 1$, then we replace the label of the current node (which is

a) by $new[a]$. Clearly the running time is $\mathcal{O}(\mathcal{O}(|T|) + \mathcal{O}(|T'|))$ and after the algorithm the symbols form a subinterval of $[m + k' + 1, .., m + 2k' + 1]$. □

The reader might ask, why we do not assume in Lemma 2 that the letters used in T form an initial interval of numbers (starting with 1). The above proof can be easily modified so that it ensures this property. But then, we would assign new names to letters, which makes it difficult to produce the final output grammar at the end.

3.1. Chain compression

The efficient implementation of $\mathsf{TreeChainComp}(F_1, T)$ is very simple: We traverse T. For an a-maximal chain of size $1 < \ell \leq |T|$ we create a record (a, ℓ, p), where p is the pointer to the top-most node in this chain. We then sort these records lexicographically using RadixSort (ignoring the last component and viewing (a, ℓ) as a number of length 2). There are at most $|T|$ records and we assume that F can be identified with an interval, see Lemma 2. Hence, RadixSort needs time $\mathcal{O}(|T|)$ to sort the records. Now, for a fixed unary letter a, the consecutive tuples with the first component a correspond to all a-maximal chains, ordered by size. It is easy to replace them in time $\mathcal{O}(|T|)$ with new letters.

Lemma 3. $\mathsf{TreeChainComp}(F_1, T)$ *can be implemented in* $\mathcal{O}(|T|)$ *time.*

Note that so far we did not care about the representation cost for the new letters that replace a-maximal chains. We use a particular scheme to represent $a_{\ell_1}, a_{\ell_2}, \ldots, a_{\ell_k}$, which has a representation cost of $\mathcal{O}(k+\sum_{i=1}^{k} \log(\ell_i - \ell_{i-1}))$, where we take $\ell_0 = 0$ for convenience. This is an easy, but important improvement over $\mathcal{O}(k+\sum_{i=1}^{k} \log \ell_i)$ obtained using the binary expansion of the numbers $\ell_1, \ell_2, \ldots, \ell_k$.

Lemma 4 (cf. [7, Lemma 2])**.** *Given a list* $\ell_1 < \ell_2 < \cdots < \ell_k$ *we can represent the letters* $a_{\ell_1}, a_{\ell_2}, \ldots, a_{\ell_k}$ *that replace the chain patterns* $a^{\ell_1}, a^{\ell_2}, \ldots, a^{\ell_k}$ *with a total cost of* $\mathcal{O}(k + \sum_{i=1}^{k} \log(\ell_i - \ell_{i-1}))$*, where* $\ell_0 = 0$*.*

Proof. The proof is identical, up to change of names, to the proof of Lemma 2 in [7], still we supply it for completeness.

Firstly observe that without loss of generality we may assume that the list $\ell_1, \ell_2, \ldots, \ell_k$ is given in a sorted way, as it can be easily obtained form the sorted list of occurrences of a-maximal chains. For simplicity define $\ell_0 = 0$ and let $\ell = \max_{i=1}^{k}(\ell_i - \ell_{i-1})$.

In the following, we define rules for certain new unary letters a_m, each of them derives a^m (in other words, a_m represents a^m). For each $1 \leq i \leq \lfloor \log \ell \rfloor$ introduce a new letter a_{2^i} with the rule $a_{2^i}(y_1) \to a_{2^{i-1}}(a_{2^{i-1}}(y_1))$, where a_1 simply denotes a. Clearly a_{2^i} represents a^{2^i} and the representation cost summed over all $1 \leq i \leq \lfloor \log \ell \rfloor$ is $2\lfloor \log \ell \rfloor$.

Now introduce new unary letters $a_{\ell_i - \ell_{i-1}}$ for each $1 \leq i \leq k$, which represent $a^{\ell_i - \ell_{i-1}}$. These letters are represented using the binary expansions of the numbers $\ell_i - \ell_{i-1}$, i.e., by concatenation of $\lfloor \log(\ell_i - \ell_{i-1}) \rfloor + 1$ many letters from $a_1, a_2, \ldots, a_{2^{\lfloor \log \ell \rfloor}}$. This introduces an additional representation cost of $\sum_{i=1}^{k}(1 + \lfloor \log(\ell_i - \ell_{i-1}) \rfloor) \leq k + \sum_{i=1}^{k} \log(\ell_i - \ell_{i-1})$.

Finally, each a_{ℓ_i} is represented as $a_{\ell_i}(y_1) \to a_{\ell_i - \ell_{i-1}}(a_{\ell_{i-1}}(y_1))$, which adds $2k$ to the representation cost. Summing all contributions yields the promised value $\mathcal{O}(k+\sum_{i=1}^{k} \log(\ell_i - \ell_{i-1}))$. □

In the following we also use a simple property of chain compression: Since no two a-maximal chains can be next to each other, there are no b-maximal chains (for any unary letter b) of length greater than 1 in T after chain compression.

Lemma 5 (cf. [7, Lemma 3])**.** *In line 4 of algorithm* TtoG *there is no node in T such that this node and its child are labelled with the same unary letter.*

Proof. The proof is straightforward: suppose for the sake of contradiction that there is a node u that is labelled with the unary letter a and u's unique child v is labelled with a, too. There are two cases:

Case 1. The letter a was present in T in line 2: But then a was listed in F_1 in line 2 and u and v are part of an a-maximal chain that was replaced by a single node during $\mathsf{TreeChainComp}(F_1, T)$.

Case 2. The letter a was introduced during $\mathsf{TreeChainComp}(F_1, T)$: Assume that a represents b^ℓ. Hence u and v both replaced b-maximal chains. But this is not possible since the definition of a b-maximal chain implies that two b-maximal chains are not adjacent. □

3.2. Unary pair compression

The operation of unary pair compression is implemented similarly as chain compression. As already noticed, since 2-chains can overlap, compressing all 2-chains at the same time is not possible. Still, we can find a subset of non-overlapping chain patterns of length 2 in T such that a (roughly) constant fraction of unary letters in T is covered by occurrences of these chain patterns. This subset is defined by a *partition* of the letters from F_1 occurring in T into subsets F_1^{up} and F_1^{down}. Then we replace all 2-chains, whose first (respectively, second) node is labelled with a letter from F_1^{up} (respectively, F_1^{down}). Our first task is to show that indeed such a partition exists and that it can be found in time $\mathcal{O}(|T|)$.

Lemma 6. *Assume that (i) T does not contain an occurrence of a chain pattern aa for some $a \in F_1$ and (ii) that the symbols in T form an interval of numbers. Then, in time $\mathcal{O}(|T|)$ one can find a partition $F_1 = F_1^{\mathrm{up}} \uplus F_1^{\mathrm{down}}$ such that the number of occurrences of chain patterns from $F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ in T is at least $(n_1 - c + 2)/4$, where n_1 is the number of nodes in T with a unary label and c is the number of maximal chains in T. In the same running time we can provide for each $ab \in F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ occurring in T a lists of pointers to all occurrences of ab in T.*

Proof. For a choice of F_1^{up} and F_1^{down} we say that occurrences of $ab \in F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ are *covered* by the partition $F_1 = F_1^{\mathrm{up}} \uplus F_1^{\mathrm{down}}$. We extend this notion also to words: a partition covers also occurrences of a chain pattern ab in a word (or set of words).

The following claim is a slighter stronger version of [7, Lemma 4], the proof is essentially the same, still, for completeness, we provide it below:

Claim 1 ([7, Lemma 4]). *For words $w_1, w_2, \ldots, w_c$ that do not contain a factor aa for any symbol a and whose alphabet can be identified with an interval of numbers of size m, one can in time $\mathcal{O}(\sum_{i=1}^{c} |w_i| + m)$ partition the letters occurring in $w_1, w_2, \ldots, w_c$ into sets F_1^{up}*

and F_1^{down} *such that the number of occurrences of chain patterns from* $F_1^{\text{up}}F_1^{\text{down}}$ *in* w_1, w_2, …, w_c *is at least* $(\sum_{i=1}^{c}(|w_i|-1))/4$. *In the same running time we can provide for each* $ab \in F_1^{\text{up}}F_1^{\text{down}}$ *occurring in* w_1, w_2, …, w_c *a lists of pointers to all occurrences of* ab *in* w_1, w_2, …, w_c.

It is easy to derive the statement of the lemma from this claim: Consider all maximal chains in T, and let us treat the corresponding chain patterns as strings $w_1, w_2, \ldots, w_c$. The sum of their lengths is $n_1 \leq |T|$. By the assumption from the lemma no two consecutive letters in strings w_1, w_2, …, w_c are identical. Moreover, the alphabet of w_1, w_2, …, w_c is within an interval of size $\mathcal{O}(|T|)$. By Claim 1 one can compute in time $\mathcal{O}(\sum_{i=1}^{c}|w_i|+|T|) \leq \mathcal{O}(|T|)$ a partition $F_1^{\text{up}} \uplus F_1^{\text{down}}$ of F_1 such that $\frac{\sum_{i=1}^{c}(|w_i|-1)}{4}$ many 2-chains from w_1, w_2, …, w_c are covered by this partition, and hence the same applies to T. Moreover, by Claim 1 one can also compute in time $\mathcal{O}(\sum_{i=1}^{c}|w_i|) \leq \mathcal{O}(|T|)$ for every $ab \in F_1^{\text{up}}F_1^{\text{down}}$ occurring in w_1, w_2, …, w_c a lists of pointers to all occurrences of ab in w_1, w_2, …, w_c. It is straightforward to compute from this list a lists of pointers to all occurrences of ab in T.

Let us now provide a proof of Claim 1:

Proof of Claim 1. Observe that finding a partition reduces to the (well-studied and well-known) problem of finding a cut in a directed and weighted graph: For the reduction, for each letter a we create a node a in a graph and make the weight of the edge (a, b) the number of occurrences of ab in w_1, w_2, …, w_c. A *directed cut* in this graph is a partition $V_1 \uplus V_2$ of the vertices, and the weight of this cut is the sum of all weights of edges in $V_1 \times V_2$. It is easy to see that a directed cut of weight k corresponds to a partition of the letters covering exactly k occurrences of chain patterns (and vice-versa). The rest of the the proof gives the standard construction [40, Section 6.3] in the terminology used in the paper (the running time analysis is not covered in standard sources).

The existence of a partition covering at least one fourth of the occurrences can be shown by a simple probabilistic argument: Divide F_1 into F_1^{up} and F_1^{down} randomly, where each letter goes to each of the parts with probability 1/2. Fix an occurrence of ab, then $a \in F_1^{\text{up}}$ and $b \in F_1^{\text{down}}$ with probability 1/4. There are $\sum_{i=1}^{c}(|w_i|-1)$ such 2-chains in w_1, w_2, …, w_c, so the expected number of occurrences of patterns from $F_1^{\text{up}}F_1^{\text{down}}$ in w_1, w_2, …, w_c is $(\sum_{i=1}^{c}(|w_i|-1))/4$. Hence, there exists a partition that covers at least $(\sum_{i=1}^{c}(|w_i|-1))/4$ many occurrences of 2-chains. Observe, that the expected number of occurrences of patterns from $F_1^{\text{up}}F_1^{\text{down}} \cup F_1^{\text{down}}F_1^{\text{up}}$ is $(\sum_{i=1}^{c}(|w_i|-1))/2$.

The deterministic construction of a partition covering at least $(\sum_{i=1}^{c}(|w_i|-1))/4$ occurrences follows by a simple derandomisation, using the conditional expectation approach. It is easier to first find a partition $F_1^{\text{up}} \uplus F_1^{\text{down}}$ such that at least $(\sum_{i=1}^{c}(|w_i|-1))/2$ many occurrences of 2-chains in w_1, w_2, …, w_c are covered by $F_1^{\text{up}}F_1^{\text{down}} \cup F_1^{\text{down}}F_1^{\text{up}}$. We then choose $F_1^{\text{up}}F_1^{\text{down}}$ or $F_1^{\text{down}}F_1^{\text{up}}$, depending on which of them covers more occurrences.

Suppose that we have already assigned some letters to F_1^{up} and F_1^{down} and we have to decide where the next letter a is assigned to. If it is assigned to F_1^{up}, then all occurrences of patterns from $aF_1^{\text{up}} \cup F_1^{\text{up}}a$ are not going to be covered, while occurrences of patterns from $aF_1^{\text{down}} \cup F_1^{\text{down}}a$ are. A similar observation holds if a is assigned to F_1^{down}. The algorithm Greedy2Chains makes a greedy choice, maximising the number of covered 2-chains in each

step. As there are only two options, the choice covers at least half of all occurrences of 2-chains that contain the letter a and a letter from $F_1^{\text{up}} \uplus F_1^{\text{down}}$. Finally, as each occurrence of a pattern ab from $w_1, w_2, \ldots, w_c$ is considered exactly once (namely when the second letter of a and b is considered in the main loop), this procedure guarantees that at least half of all 2-chains in $w_1, w_2, \ldots, w_c$ are covered.

In order to make the selection efficient, the algorithm Greedy2Chains below keeps for every letter a counters $\mathit{count}^{\text{up}}[a]$ and $\mathit{count}^{\text{down}}[a]$, storing the number of occurrences of patterns from $aF_1^{\text{up}} \cup F_1^{\text{up}}a$ and $aF_1^{\text{down}} \cup F_1^{\text{down}}a$, respectively, in $w_1, w_2, \ldots, w_c$. These counters are updated as soon as a letter is assigned to F_1^{up} or F_1^{down}.

Algorithm 5 Greedy2Chains

1: $F_1 \leftarrow$ set of letters used in $w_1, w_2, \ldots, w_c$
2: $F_1^{\text{up}} \leftarrow F_1^{\text{down}} \leftarrow \emptyset$ ▷ organised as a bit vector
3: **for** $a \in F_1$ **do**
4: $\quad \mathit{count}^{\text{up}}[a] \leftarrow \mathit{count}^{\text{down}}[a] \leftarrow 0$ ▷ initialisation
5: **for** $a \in F_1$ **do**
6: $\quad$ **if** $\mathit{count}^{\text{down}}[a] \geq \mathit{count}^{\text{up}}[a]$ **then** ▷ choose the one that guarantees larger cover
7: $\quad\quad \mathit{choice} \leftarrow$ up
8: $\quad$ **else**
9: $\quad\quad \mathit{choice} \leftarrow$ down
10: $\quad$ **for** $b \in F_1$ and all occurrences of ab or ba in $w_1, w_2, \ldots, w_c$ **do**
11: $\quad\quad \mathit{count}^{\text{choice}}[b] \leftarrow \mathit{count}^{\text{choice}}[b] + 1$
12: $\quad F_1^{\text{choice}} \leftarrow F_1^{\text{choice}} \cup \{a\}$
13: **if** # occurrences of patterns from $F_1^{\text{down}}F_1^{\text{up}}$ in $w_1, w_2, \ldots, w_c >$ # occurrences of patterns from $F_1^{\text{up}}F_1^{\text{down}}$ in $w_1, w_2, \ldots, w_c$ **then**
14: $\quad$ switch F_1^{down} and F_1^{up}
15: **return** $(F_1^{\text{up}}, F_1^{\text{down}})$

By the argument given above, when F_1 is partitioned into F_1^{up} and F_1^{down} by Greedy2Chains, at least half of all 2-chains in $w_1, w_2, \ldots, w_c$ are occurrences of patterns from $F_1^{\text{up}}F_1^{\text{down}} \cup F_1^{\text{down}}F_1^{\text{up}}$. Then one of the choices $(F_1^{\text{up}}, F_1^{\text{down}})$ or $(F_1^{\text{down}}, F_1^{\text{up}})$ covers at least one fourth of all 2-chains in $w_1, w_2, \ldots, w_c$.

It is left to give an efficient variant of Greedy2Chains. The non-obvious operations are the updating of $\mathit{count}^{\text{choice}}[b]$ in line 11 and the choice of the actual partition in line 14. All other operation clearly take at most time $\mathcal{O}(\sum_{i=1}^{c} |w_i|)$. The latter is simple: since we organise F_1^{up} and F_1^{down} as bit vectors, we can read each $w_1, w_2, \ldots, w_c$ from left to right (in any order) and calculate the number of occurrences of patterns from $F_1^{\text{up}}F_1^{\text{down}}$ as well as those from $F_1^{\text{down}}F_1^{\text{up}}$ in time $\mathcal{O}(\sum_{i=1}^{c} |w_i|)$ (when we read a pattern ab we check in $\mathcal{O}(1)$ time whether $ab \in F_1^{\text{up}}F_1^{\text{down}}$ or $ab \in F_1^{\text{down}}F_1^{\text{up}}$). Afterwards we choose the partition that covers more 2-chains in $w_1, w_2, \ldots, w_c$.

To implement $\mathit{count}^{\text{up}}$ and $\mathit{count}^{\text{down}}$, for each letter a in $w_1, w_2, \ldots, w_c$ we store a *right list* $\mathit{right}(a) = \{b \mid ab \text{ occurs in } w\}$, represented as a list. Furthermore, the element b on the

right list points to a list of all occurrences of the pattern ab in w_1, w_2, ..., w_c. There is a similar *left list* $\mathit{left}(a) = \{b \mid ba \text{ occurs in } w\}$. We comment on how to create the left lists and right lists in linear time later.

Given *right* and *left*, performing the update in line 11 is easy: We go through $\mathit{right}(a)$ (respectively, $\mathit{left}(a)$) and increment $\mathit{count}^{\mathrm{up}}[b]$ for every occurrence of ab (respectively, ba). Note that in this way each of the lists $\mathit{right}(a)$ ($\mathit{left}(a)$) is read once during Greedy2Chains, the same applies also to pointers from those lists. Therefore, all updates of $\mathit{count}^{\mathrm{up}}$ and $\mathit{count}^{\mathrm{down}}$ only need time $\mathcal{O}(\sum_{i=1}^{c} |w_i|)$, as the total number of pointers on those lists is $\mathcal{O}(|T|)$.

It remains to show how to initially create $\mathit{right}(a)$ ($\mathit{left}(a)$ is created similarly). We read w_1, w_2, ..., w_c. When reading a pattern ab we create a record (a, b, p), where p is a pointer to this occurrence. We then sort these records lexicographically using RadixSort, ignoring the last component. There are $\sum_{i=1}^{c} |w_i|$ records and the alphabet is an interval of size m, so RadixSort needs time $\mathcal{O}(\sum_{i=1}^{c} |w_i| + m)$. Now, for a fixed letter a, the consecutive tuples with the first component a can be turned into $\mathit{right}(a)$: for $b \in \mathit{right}(a)$ we want to store a list I of pointers to occurrences of ab. On a sorted list of records the entries (a, b, p) for $p \in I$ form an interval of consecutive records. This shows the first statement from Claim 1.

In order to show the second statement from Claim 1, i.e., in order to get for each $ab \in F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ the lists of pointers to occurrences of ab in w_1, w_2, ..., w_c, it is enough to read *right* and filter the patterns ab such that $a \in F_1^{\mathrm{up}}$ and $b \in F_1^{\mathrm{down}}$; the filtering can be done in $\mathcal{O}(1)$ per occurrence as F_1^{up} and F_1^{down} are represented as bitvectors. The total needed time is $\mathcal{O}(\sum_{i=1}^{c} |w_i|)$. This concludes the proof of Claim 1 and thus also the proof of Lemma 6. □

When for each pattern $ab \in F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ the list of its occurrences in T is provided, the replacement of these occurrences is done by going through the list and replacing each of the occurrences, which is done in linear time. Note that since F_1^{up} and F_1^{down} are disjoint, the considered occurrences cannot overlap and the order of the replacements is unimportant.

Lemma 7. TreeUnaryComp$(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, T)$ *can be implemented in* $\mathcal{O}(|T|)$ *time.*

3.3. Leaf compression

Leaf compression is done in a way similar to chain compression and $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression: We traverse T. Whenever we reach a node v labelled with a symbol $f \in F_{\geq 1}$, we scan the list of its children. Assume that this list is $v_1, v_2, \ldots, v_m$. When no v_i is a leaf, we do nothing. Otherwise, let $1 \leq i_1 < i_2 < \cdots < i_\ell \leq m$ be a list of those positions such that v_{i_k} is a leaf, say labelled with a constant a_k, for all $1 \leq k \leq \ell$. We create a record $(f, i_1, a_1, i_2, a_2, \ldots, i_\ell, a_\ell, p)$, where p is a pointer to node v, and continue with the traversing of T. Observe that the total number of elements in the created tuples is at most $2|T|$. Furthermore each position index is at most $r \leq |T|$ and by Lemma 2 also each letter is a number from an interval of size at most $|T|$. Hence RadixSort sorts those tuples (ignoring the pointer coordinate) in time $\mathcal{O}(|T|)$ (we use the RadixSort version for lists of varying length). After the sorting the tuples corresponding to nodes with the same label and the same constant-labelled children (at the same positions) are consecutive on the returned list,

so we can easily perform the replacement. Given a tuple $(f, i_1, a_1, i_2, a_2, \ldots, i_\ell, a_\ell, p)$ we use the last component (i.e. pointer) in the created records to localize the node, replace the label f with the fresh label f' and remove the children at positions $i_1, i_2, \ldots, i_\ell$ (note that in the meantime some other children might become leaves, we do not remove them, though). Clearly all of this takes time $\mathcal{O}(|T|)$.

Lemma 8. $\mathsf{TreeLeafComp}(F_{\geq 1}, F_0, T)$ *can be implemented in* $\mathcal{O}(|T|)$ *time.*

3.4. Size and running time

It remains to estimate the total running time of our algorithm TtoG, summed over all phases. As each subprocedure in a phase has running time $\mathcal{O}(|T|)$ and there are constant number of them in a phase, it is enough to show that $|T|$ is reduced by a constant factor per phase (then the sum of the running times over all phases is a geometric sum).

Lemma 9. *In each phase,* $|T|$ *is reduced by a constant factor.*

Proof. For $i \geq 0$ let n_i, n'_i, n''_i and n'''_i be the number of nodes labelled with a letter of rank i in T at the beginning of the phase, after chain compression, unary pair compression, and leaf compression, respectively. Let $n_{\geq 2} = \sum_{i \geq 2} n_i$ and define $n'_{\geq 2}$, $n''_{\geq 2}$, and $n'''_{\geq 2}$ similarly. We have

$$n_0 \geq n_{\geq 2} + 1 \ . \tag{4}$$

To see this, note that there are $n_0 + n_1 + n_{\geq 2} - 1$ nodes that are children ('-1' is for the root). On the other hand, a node of arity i is a parent node for i children. So the number of children is at least $2n_{\geq 2} + n_1$. Comparing those two values yields (4).

We next show that

$$n'''_0 + n'''_1 + n'''_{\geq 2} \leq \frac{3}{4}(n_0 + n_1 + n_{\geq 2}) \ ,$$

which shows the claim of the lemma. Let c denote the number of maximal chains in T at the beginning of the phase, this number does not change during chain compression and unary pair compression. Observe that

$$c \leq n_{\geq 2} + 1 \ . \tag{5}$$

Indeed, consider a maximal chain. Then the node above the chain has a label from $F_{\geq 2}$ or the maximal chain starts in the root. Summing this up over all chains, we get (5).

Clearly after chain compression we have $n'_0 = n_0$, $n'_1 \leq n_1$ and $n'_{\geq 2} = n_{\geq 2}$. Furthermore, the number of maximal chains does not change. During unary pair compression, by Lemma 6, we choose a partition such that at least $\frac{n'_1 - c + 2}{4}$ many 2-chains are compressed (note that the assumption of Lemma 6 that no parent node and its child are labelled with the same unary letter is satisfied by Lemma 5), so the size of the tree is reduced by at least $\frac{n'_1 - c + 2}{4}$. Hence, the size of the tree after unary pair compression is at most

$$\begin{aligned} n''_0 + n''_1 + n''_{\geq 2} &\leq n'_0 + n'_1 + n'_{\geq 2} - \frac{n'_1 - c + 2}{4} \\ &= n'_0 + \frac{3n'_1}{4} + n'_{\geq 2} + \frac{c}{4} - \frac{1}{2} \\ &\leq n_0 + \frac{3n_1}{4} + n_{\geq 2} + \frac{c}{4} - \frac{1}{2} \ . \end{aligned} \tag{6}$$

Lastly, during leaf compression the size is reduced by $n_0'' = n_0$. Hence the size of T after all three compression steps is

$$\begin{aligned}
n_0''' + n_1''' + n_{\geq 2}''' &= n_0'' + n_1'' + n_{\geq 2}'' - n_0 && \text{leaf compression} \\
&\leq n_0 + \frac{3n_1}{4} + n_{\geq 2} + \frac{c}{4} - \frac{1}{2} - n_0 && \text{from (6)} \\
&= \frac{3n_1}{4} + n_{\geq 2} + \frac{c}{4} - \frac{1}{2} && \text{simplification} \\
&\leq \frac{3n_1}{4} + n_{\geq 2} + \underbrace{\frac{n_{\geq 2}}{4} + \frac{1}{4}}_{\geq c/4} - \frac{1}{2} && \text{from (5)} \\
&< \frac{3n_1}{4} + \frac{5n_{\geq 2}}{4} && \text{simplification} \\
&< \frac{3}{4}\Big(n_0 + n_1 + n_{\geq 2}\Big), && \text{from (4)}
\end{aligned}$$

as claimed. □

Theorem 2. TtoG *runs in linear time.*

Proof. By Lemma 2 there is a linear preprocessing. By Lemmata 2, 3, 7, and 8, each phase takes $\mathcal{O}(|T|)$ time and by Lemma 9, $|T|$ drops by a constant factor in each phase. As the initial size of T is n, the total running time is $\mathcal{O}(n)$. □

4. Size of the grammar: recompression

To bound the cost of representing the letters introduced during the construction of the SLCF grammar, we start with a smallest SLCF grammar $\mathbb{G}_{\text{opt}}$ generating the input tree T (note that $\mathbb{G}_{\text{opt}}$ is not necessarily unique) and show that we can transform it into an SLCF grammar $\mathbb{G}$ (also generating T) of a special normal form, called handle grammar. This form is described in detail in Section 4.1. The grammar $\mathbb{G}$ is of size $\mathcal{O}(r|\mathbb{G}_{\text{opt}}|)$, where r is the maximal rank of symbols in F (the set of letters occurring in $\mathbb{G}_{\text{opt}}$). The transformation is based on known results on normal forms for SLCF grammars [18], see Section 4.1.

To bound the size of $\mathbb{G}$, we assign *credits* to $\mathbb{G}$: each occurrence of a letter in a right-hand side of $\mathbb{G}$ has two units of credit. If such a letter is removed from $\mathbb{G}$ for any reason, its credit is *released* and if a new letter is inserted into some right-hand side of a rule, then we *issue* its credit.

During the run of TtoG we modify $\mathbb{G}$, preserving its special handle form, so that it generates T (i.e., the current tree kept by TtoG) after each of the compression steps of TtoG. In essence, if a compression is performed on T then we also apply it on $\mathbb{G}$ and modify $\mathbb{G}$ so that it generates the tree T after the compression step. Then the cost of representing the letters introduced by TtoG is paid by credits released during the compression of letters in TtoG. Therefore, instead of computing the total representation cost of the new letters, it suffices to calculate the total amount of issued credit, which is much easier than calculating

the actual representation cost. Note that this is entirely a mental experiment for the purpose of the analysis, as $\mathbb{G}$ is not stored or even known by TtoG. We just perform some changes on it depending on the actions of TtoG.

The analysis outlined above is not enough to bound the representation cost for chain compression, we need specialised tools for that. They are described in Section 4.6.

In this section we show a slightly weaker bound, the full proof of the bound from Theorem 1 is presented in Section 5.

4.1. Normal form

As explained above, in our mental experiment we modify the grammar $\mathbb{G}$ and perform the compression operations on it. To make the analysis simpler, we want to have a special form in which the compression operation will not interact too much between different parts of the grammar. This idea is formalised using *handles*: We say that a pattern $t(y)$ is a *handle* if it is of the form

$$f(w_1(\gamma_1), w_2(\gamma_2), \ldots, w_{i-1}(\gamma_{i-1}), y, w_{i+1}(\gamma_{i+1}), \ldots, w_\ell(\gamma_\ell)),$$

where $\mathrm{rank}(f) = \ell$, every γ_j is either a constant symbol or a nonterminal of rank 0, every w_j is a chain pattern, and y is a parameter, see Figure 5. Note that $a(y)$ for a unary letter a is a handle. Since handles have one parameter only, for handles $h_1, h_2, \ldots, h_\ell$ we write $h_1 h_2 \cdots h_\ell$ for the tree $h_1(h_2(\ldots(h_\ell(y))))$ and treat it as a string, similarly to chains patterns.

We say that an SLCF grammar $\mathbb{G} = (N, F, P, S)$ is a *handle grammar* (or simply "$\mathbb{G}$ is handle") if the following conditions hold:

(HG1) $N \subseteq \mathbb{N}_0 \cup \mathbb{N}_1$

(HG2) For $A \in N \cap \mathbb{N}_1$ the unique rule for A is of the form

$$A \to uBvCw \quad \text{or} \quad A \to uBv \quad \text{or} \quad A \to u,$$

where u, v, and w are (perhaps empty) sequences of handles and $B, C \in N_1$. We call B the *first* and C the *second* nonterminal in the rule for A, see Figure 6.

(HG3) For $A \in N \cap \mathbb{N}_0$ the rule for A is of the (similar) form

$$A \to uBvC \quad \text{or} \quad A \to uBvc \quad \text{or} \quad A \to uC \quad \text{or} \quad A \to uc,$$

where u and v are (perhaps empty) sequences of handles, c is a constant, $B \in N_1$, and $C \in N_0$, see Figure 6. Again we speak of the *first* and *second* nonterminal in the rule for A.

Note that the representation of the rules for nonterminals from $\mathbb{N}_0$ is not unique. Take for instance the rule $A \to f(B, C)$, which can be written as $A \to h(C)$ for the handle $h(y) = f(B, y)$ or as $A \to h'(B)$ for the handle $h' = f(y, C)$. On the other hand, for nonterminals from $\mathbb{N}_1$ the representation of the rules is unique, since there is a unique occurrence of the parameter y in the right-hand side.

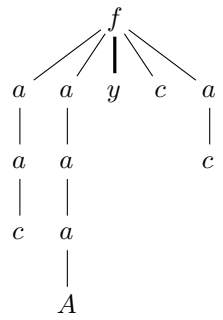

Figure 5: A handle, where A has rank 0, c is a constant and a is a unary letter.

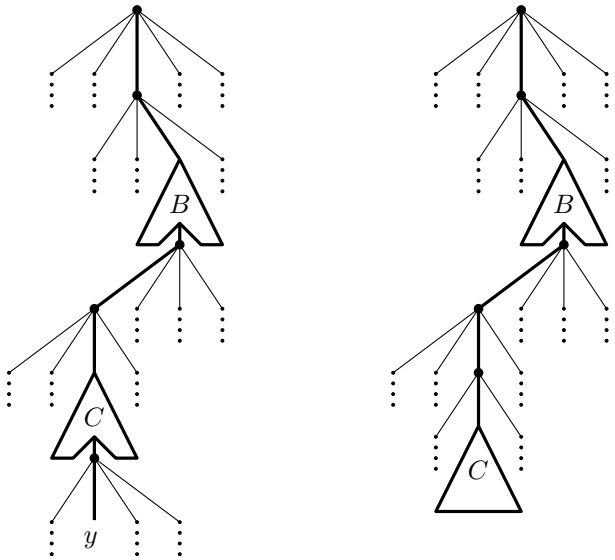

Figure 6: Two possible shapes of right-hand sides in a handle grammar for a nonterminal of rank 1 (left) and rank 0 (right), respectively. The dots symbolise the chains of unary letters ended by a nonterminal of rank 0 or a constant.

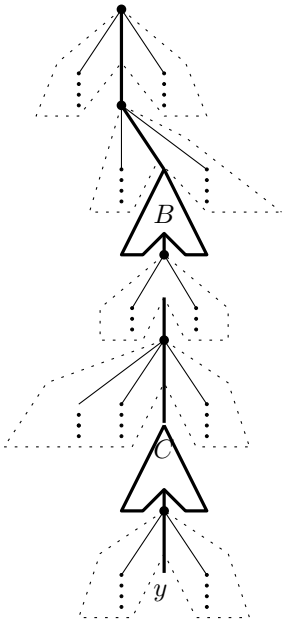

Figure 7: The tree obtained from a rule for nonterminal of rank 1. Each handle and nonterminal represents a context, each such context is enclosed either by a dotted or fat line, respectively.

What is left is to show how to transform an arbitrary SLCF grammar $\mathbb{G}'$ into an equivalent handle grammar $\mathbb{G}$. There is a known construction that transforms an SLCF grammar $\mathbb{G}'$ into an equivalent monadic SLCF grammar $\mathbb{G}$ [18, Theorem 10] (i.e. every nonterminal of $\mathbb{G}'$ has rank 0 or 1). While the original paper [18] contains only weaker statement, in fact this construction returns a handle grammar which has $\mathcal{O}(|\mathbb{G}'|)$ many occurrences of nonterminals of arity 1 in the rules and $\mathcal{O}(r|\mathbb{G}'|)$ occurrences of nonterminals of arity 0 and letters. This stronger result is repeated in the appendix for completeness.

Lemma 10 (cf. [18, Theorem 10])**.** *From a given SLCF grammar $\mathbb{G}'$ of size $g = |\mathbb{G}'|$ one can construct an equivalent handle grammar $\mathbb{G}$ of size $\mathcal{O}(rg)$ with only $\mathcal{O}(g)$ many occurrences of nonterminals of arity 1 in the rules (and $\mathcal{O}(rg)$ occurrences of nonterminals of arity 0).*

The construction and proof of [18, Theorem 10] yield the claim, though the actual statement in [18] is a bit weaker. For completeness, the proof of this stronger statement is given in the appendix.

When considering handle grammars it is useful to have some intuition about the trees they derive. Recall that a *context* is a pattern $t(y) \in \mathcal{T}(\mathbb{F}, \{y\})$ with a unique occurrence of the only parameter y. Observe that each nonterminal $A \in \mathbb{N}_1$ derives a unique context $\mathrm{val}(A)$, the same applies to a handle f and so we write $\mathrm{val}(f)$ as well. Furthermore, we can 'concatenate' contexts, so we write them in string notation. Also, when we attach a tree from $\mathcal{T}(\mathbb{F})$ to a context, we obtain another tree from $\mathcal{T}(\mathbb{F})$. Thus, when we consider a rule $A \to h_1 \cdots h_i B h_{i+1} \cdots h_j C h_{j+1} \cdots h_k$ in a handle grammar (where $h_1, \ldots, h_k$ are handles and A, B, and C are nonterminals of rank 1) then

$$\mathrm{val}(A) = \mathrm{val}(h_1) \cdots \mathrm{val}(h_i)\,\mathrm{val}(B)\,\mathrm{val}(h_{i+1}) \cdots \mathrm{val}(h_j)\,\mathrm{val}(C)\,\mathrm{val}(h_{j+1}) \cdots \mathrm{val}(h_k),$$

i.e., we concatenate the contexts derived by the handles and nonterminals, see Figure 7. Similar considerations apply to other rules of handle grammars as well, also the ones for nonterminals of rank 0.

4.2. Intuition and invariants

For a given input tree T we start (as a mental experiment) with a smallest SLCF grammar generating T. Let g be the size of this grammar. We first transform it to a handle grammar $\mathbb{G}$ of size $\mathcal{O}(gr)$ using Lemma 10. The number of nonterminals of rank 0 (resp., 1) in $\mathbb{G}$ is bounded by $\mathcal{O}(gr)$ (resp., $\mathcal{O}(g)$).

In the following, by T we denote the current tree stored by TtoG. For analysing the size of the grammar produced by TtoG applied to T, we use the accounting method, see e.g. [38, Section 17.2]. With each occurrence of a letter from $\mathbb{F}$ in $\mathbb{G}$'s rules we associate two units of *credit* (no credit is assigned to occurrences of nonterminals in rules). During the run of TtoG we appropriately modify $\mathbb{G}$, so that $\mathrm{val}(\mathbb{G}) = T$ (recall that T always denotes the current tree stored by TtoG). In other words, we perform the compression steps of TtoG also on $\mathbb{G}$. We thereby always maintain the invariant that every occurrence of a letter from $\mathbb{F}$ in $\mathbb{G}$'s rules has two units of credit. In order to do this, we have to *issue* (or pay) some new credits during the modifications, and we have to do a precise bookkeeping on the amount of issued credit. On the other hand, if we do a compression step in $\mathbb{G}$, then we remove some occurrences of letters. The credit associated with these occurrences is then *released* and can be used to pay for the representation cost of the new letters introduced by the compression step. For unary pair compression and leaf compression, the released credit indeed suffices to pay the representation cost for the fresh letters, but for chain compression the released credit does not suffice. Here we need some extra amount that will be estimated separately later on in Section 4.6. At the end, we can bound the size of the grammar produced by TtoG by the sum of the initial credit assigned to $\mathbb{G}$, which is at most $\mathcal{O}(rg)$ by Lemma 10, plus the total amount of issued credit plus the extra cost estimated in Section 4.6.

An important difference between our algorithm and the string compression algorithm from [7], which we generalise, is that we add new nonterminals to $\mathbb{G}$ during its modification. To simplify notation, we denote with m always the number of nonterminals of the current grammar $\mathbb{G}$, and we denote its nonterminals with $A_1, \ldots, A_m$. We assume that $i < j$ if A_i occurs in the right-hand side of A_j, and that A_m is the start nonterminal. With α_i we always denote the current right-hand side of A_i. In other words, the productions of $\mathbb{G}$ are $A_i \to \alpha_i$ for $1 \leq i \leq m$.

Again note that the modification of $\mathbb{G}$ is not really carried out by TtoG, but is only done for the purpose of analysing TtoG.

Suppose a compression step, for simplicity say an (a, b)-pair compression, is applied to T. We should also reflect it in $\mathbb{G}$. The simplest solution would be to perform the same compression on each of the rules of $\mathbb{G}$, hoping that in this way all occurrences of ab in $\mathrm{val}(\mathbb{G})$ are replaced by c. However, this is not always the case. For instance, the 2-chain ab may occur 'between' a nonterminal and a unary letter. This intuition is made precise in Section 4.3. To deal with this problem, we modify the grammar, so that the problem disappears. Similar problems occur also when we want to replace an a-maximal chain or

perform leaf compression. The solutions to those problems are similar and are given in Section 4.4 and Section 4.5, respectively.

To ensure that $\mathbb{G}$ stays handle and to estimate the amount of issued credit, we show that the grammar preserves the following invariants, where $n_0 = \mathcal{O}(gr)$ (respectively, $n_1 = \mathcal{O}(g)$) is the initial number of occurrences of nonterminals from $\mathbb{N}_0$ (respectively, $\mathbb{N}_1$) in $\mathbb{G}$ while g_0 and g_1 are those values at some particular moment. Similarly, $\widetilde{g_0}$ is the number of occurrences of nonterminals from $\widetilde{N_0}$.

(GR1) $\mathbb{G}$ is handle.

(GR2) $\mathbb{G}$ has nonterminals $N_0 \cup N_1 \cup \widetilde{N_0}$, where $\widetilde{N_0} \cup N_0 \subseteq \mathbb{N}_0$, $|N_0| \leq n_0$ and $N_1 \subseteq \mathbb{N}_1$, $|N_1| \leq n_1$.

(GR3) The number g_0 of occurrences nonterminals from N_0 in $\mathbb{G}$ never increases (and is initially n_0), and the number g_1 of occurrences of nonterminals from N_1 also never increases (and is initially n_1).

(GR4) The number $\widetilde{g_0}$ of occurrences of nonterminals from $\widetilde{N_0}$ in $\mathbb{G}$ is at most $n_1(r-1)$.

(GR5) The rules for $A_i \in \widetilde{N_0}$ are of the form $A_i \to wA_j$ or $A_i \to wc$, where w is a string of unary symbols, $A_j \in N_0 \cup \widetilde{N_0}$, and c is a constant.

Intuitively, N_0 and N_1 are subsets of the initial nonterminals of rank 0 and 1, respectively, while $\widetilde{N_0}$ are the nonterminals introduced by TtoG, which are all of rank 0.

Clearly, (GR1)–(GR5) hold for the initial handle grammar $\mathbb{G}$ obtained by Lemma 10.

4.3. ($F_1^{\text{up}}, F_1^{\text{down}}$)-compression

We begin with some necessary definitions that help to classify 2-chains. For a non-empty tree or context t its *first* letter is the letter that labels the root of t. For a context $t(y)$ which is not a parameter its *last* letter is the label of the node above the one labelled with y. For instance, the last letter of the context $a(b(y))$ is b and the last letter of the context $f(a(c), y)$ is f, which is also the first letter.

A chain pattern ab has a *crossing occurrence* in a nonterminal A_i if one of the following holds:

(CR1) aA_j is a subpattern of α_i and the first letter of $\text{val}(A_j)$ is b

(CR2) $A_j(b)$ is a subpattern of α_i and the last letter of $\text{val}(A_j)$ is a

(CR3) $A_j(A_k)$ is a subpattern of α_i, the last letter of $\text{val}(A_j)$ is a and the first letter of $\text{val}(A_k)$ is b.

A chain pattern ab is *crossing* if it has a crossing occurrence in any nonterminal and *non-crossing* otherwise. Unless explicitly written, we use this notion only in case $a \neq b$.

When every chain pattern $ab \in F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ is noncrossing, simulating $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression on $\mathbb{G}$ is easy: It is enough to apply TreeUnaryComp (Algorithm 3) to each right-hand side of $\mathbb{G}$. We denote the resulting grammar with $\mathsf{TreeUnaryComp}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathbb{G})$.

In order to distinguish between the nonterminals, grammar, etc. before and after the application of TreeUnaryComp (or, in general, any procedure) we use 'primed' symbols, i.e., A'_i, $\mathbb{G}'$, T' for the nonterminals, grammar and tree, respectively, after the compression step and 'unprimed' symbols (i.e., A_i, $\mathbb{G}$, T) for the ones before.

Lemma 11. *Let $\mathbb{G}$ be a handle grammar and $\mathbb{G}' = \mathsf{TreeUnaryComp}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathbb{G})$. Then the following hold:*

- *If $\mathbb{G}$ satisfies (GR1)–(GR5) then $\mathbb{G}'$ satisfies (GR1)–(GR5) as well.*
- *If there is no crossing chain pattern from $F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ in $\mathbb{G}$, then*

$$\mathrm{val}(\mathbb{G}') = \mathsf{TreeUnaryComp}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathrm{val}(\mathbb{G})).$$

- *The grammar $\mathbb{G}'$ has the same number of occurrences of nonterminals of each rank as $\mathbb{G}$.*
- *The credit for new letters in $\mathbb{G}'$ and the cost of representing these new letters are paid by the released credit.*

Proof. Clearly, $\mathrm{val}(\mathbb{G}')$ can be obtained from $\mathrm{val}(\mathbb{G})$ by compressing some occurrences of patterns from $F_1^{\mathrm{up}} F_1^{\mathrm{down}}$. Hence, to show that $\mathrm{val}(\mathbb{G}') = \mathsf{TreeUnaryComp}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathrm{val}(\mathbb{G}))$, it suffices to show that $\mathrm{val}(\mathbb{G}')$ does not contain occurrences of patterns from $F_1^{\mathrm{up}} F_1^{\mathrm{down}}$. By induction on i we show that for every $1 \leq i \leq m$, $\mathrm{val}(A'_i)$ does not contain occurrences of patterns from $F_1^{\mathrm{up}} F_1^{\mathrm{down}}$. To get a contradiction, consider an occurrence of $ab \in F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ in $\mathrm{val}(A'_i)$. If it is generated by an explicit occurrence of ab in the right-hand side of A'_i then it was present already in the rule for A_i, since we do not introduce new occurrences of the letters from $\mathbb{G}$. So, the occurrence of ab is replaced by a new letter in $\mathbb{G}'$. If the occurrence is contained within the subtree generated by some A'_j ($j < i$), then the occurrence is compressed by the inductive assumption. The remaining case is that there exists a crossing occurrence of ab in the rule for A'_i. However note that if a is the first (or b is the last) letter of $\mathrm{val}(A'_j)$, then it was also the first (respectively, last) letter of $\mathrm{val}(A_j)$ in the input instance, as we do not introduce new occurrences of the old letters. Hence, the occurrence of ab was crossing already in the input grammar $\mathbb{G}$, which is not possible by the assumption of the lemma.

Each occurrence of $ab \in F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ has 4 units of credit (two for each symbol), which are released in the compression step. Two of the released units are used to pay for the credit of the new occurrence of the symbol c (which replaces the occurrence of ab), while the other two units are used to pay for the representation cost of c (if we replace more than one occurrence of ab in $\mathbb{G}$, some credit is wasted).

Let us finally argue that the invariants (GR1)–(GR5) are preserved: Replacing an occurrence of ab with a single unary letter c cannot make a handle grammar a non-handle

one, so (GR1) is preserved. Similarly, (GR5) is preserved. The set of nonterminals and the number of occurrences of the nonterminals is unaffected, so also (GR2)–(GR4) are preserved. □

By Lemma 11 it is left to assure that indeed all occurrences of chain patterns from $F_1^{\mathrm{up}}F_1^{\mathrm{down}}$ are noncrossing. What can go wrong? Consider for instance the grammar with the rules $A_1(y) \to a(y)$ and $A_2 \to A_1(b(c))$. The pattern ab has a crossing occurrence. To deal with crossing occurrences we change the grammar. In our example, we replace A_1 by a in the right-hand side of A_2, leaving only $A_2 \to ab(c)$, which does not contain a crossing occurrence of ab.

Suppose that some $ab \in F_1^{\mathrm{up}}F_1^{\mathrm{down}}$ is crossing because of (CR1). Let aA_i be a subpattern of some right-hand side and let $\mathrm{val}(A_i) = bt'$. Then it is enough to modify the rule for A_i so that $\mathrm{val}(A_i) = t'$ and replace each occurrence of A_i in a right-hand side by bA_i. We call this action *popping-up b from A_i*. The similar operation of *popping down* a letter a from $A_i \in N \cap \mathbb{N}_1$ is symmetrically defined (note that both pop operations apply only to unary letters). See Figure 8 for an example. A similar operation of popping letters in the context of tree grammars is used also in [41].

The lemma below shows that popping up and popping down removes all crossing occurrences of ab. Note that the operations of popping up and popping down can be performed for several letters in parallel: The procedure $\mathsf{Pop}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathbb{G})$ below 'uncrosses' all occurrences of patterns from the set $F_1^{\mathrm{up}}F_1^{\mathrm{down}}$, assuming that F_1^{up} and F_1^{down} are disjoint subsets of F_1 (and we apply it only in the cases in which they are disjoint).

Recall that for a handle grammar, right-hand sides can be viewed as sequences of nonterminals and handles. Hence, we can speak of the first (respectively, last) symbol of a right-hand side.

Algorithm 6 $\mathsf{Pop}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathbb{G})$: Popping letters from F_1^{up} and F_1^{down}

```
1: for i ← 1..m − 1 do
2:     if the first symbol of α_i is b ∈ F_1^down then                ▷ popping up b
3:         if α_i = b then
4:             replace A_i in all right-hand sides of 𝔾 by b
5:         else
6:             remove this leading b from α_i
7:             replace A_i in all right-hand sides of 𝔾 by bA_i
8:     if A_i ∈ N_1 and the last symbol of α_i is a ∈ F_1^up then      ▷ popping down a
9:         if α_i = a then
10:            replace A_i in all right-hand sides of 𝔾 by a
11:        else
12:            remove this final a from α_i
13:            replace A_i in all right-hand sides of 𝔾 by A_ia
```

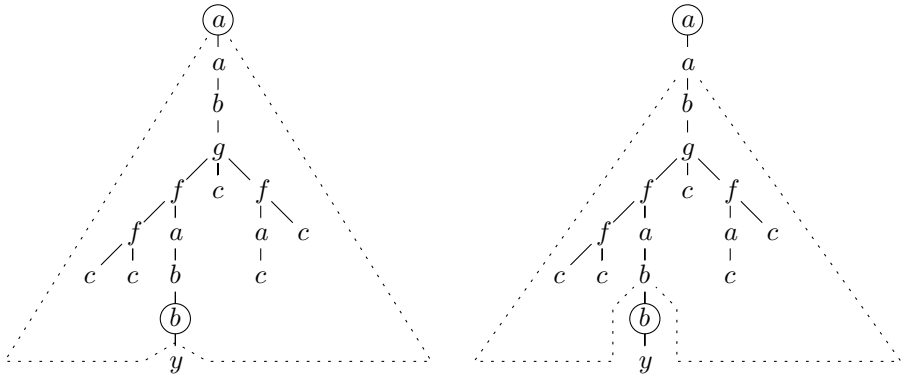

Figure 8: A tree before and after $\mathsf{Pop}(\{a\}, \{b\}, \cdot)$. Affected nodes are encircled, the dotted lines enclose the patterns generated by a nonterminal.

Lemma 12. *Let $\mathbb{G}$ be a handle grammar and $\mathbb{G}' = \mathsf{Pop}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathbb{G})$, where $F_1^{\mathrm{up}} \cap F_1^{\mathrm{down}} = \emptyset$. Then, the following hold:*

- $\mathrm{val}(A'_m) = \mathrm{val}(A_m)$ *and hence* $\mathrm{val}(\mathbb{G}) = \mathrm{val}(\mathbb{G}')$.
- *All chain patterns from $F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ are non-crossing in $\mathbb{G}'$.*
- *If $\mathbb{G}$ satisfies (GR1)–(GR5), then so does $\mathbb{G}'$.*
- *The grammar $\mathbb{G}'$ has at most the same number of occurrences of nonterminals of each rank as $\mathbb{G}$.*
- *During the computation of $\mathbb{G}'$ from $\mathbb{G}$, at most two letters are popped for each occurrence of a nonterminal of rank 1 and at most one letter is popped for each occurrence of a nonterminal of rank 0. In particular, if $\mathbb{G}$ satisfies (GR1)–(GR5) then at most $4g_1 + 2g_0 + 2\widetilde{g_0}$ units of credit are issued during the computation of $\mathbb{G}'$.*

Proof. Observe first that whenever we pop up b from some A_i, then we replace each of A_i's occurrences in $\mathbb{G}$ with bA_i (or with b, when $\mathrm{val}(A_i) = b$), and similarly for the popping down operation, thus the value of $\mathrm{val}(A_j)$ is not changed for $j \neq i$. Hence, in the end we have $\mathrm{val}(A'_m) = \mathrm{val}(A_m) = T$ (note that A_m does not pop letters).

Secondly, we show that if the first letter of $\mathrm{val}(A'_i)$ (where $i < m$) is $b' \in F_1^{\mathrm{down}}$ then we popped-up a letter from A_i (which by the code is some $b \in F_1^{\mathrm{down}}$); a similar claim holds by symmetry for the last letter of $\mathrm{val}(A_i)$. So, suppose that the claim is not true and consider the nonterminal A_i with the smallest i such that the first letter of $\mathrm{val}(A'_i)$ is $b' \in F_1^{\mathrm{down}}$ but we did not pop up a letter from A_i. Consider the first symbol of α_i when Pop considered A_i in line 2. Note, that as Pop did not pop up a letter from A_i, the first letter of $\mathrm{val}(A_i)$ and $\mathrm{val}(A'_i)$ is the same and hence it is $b' \in F_1^{\mathrm{down}}$. So α_i cannot begin with a letter as then it is $b' \in F_1^{\mathrm{down}}$ which should have been popped-up. Hence, the first symbol of α_i is some nonterminal A_j for $j < i$. But then the first letter of $\mathrm{val}(A'_j)$ is $b' \in F_1^{\mathrm{down}}$ and so by the inductive assumption Pop popped-up a letter from A_j. Hence, α_i begins with a letter when A_i is considered in line 2. We obtained a contradiction.

Suppose now that after Pop there is a crossing pattern $ab \in F_1^{\mathrm{up}} F_1^{\mathrm{down}}$. This is due to one of the bad situations (CR1)–(CR3). We consider only (CR1); the other cases are dealt in a similar fashion. Hence, assume that aA_i' is a subpattern in a right-hand side of $\mathbb{G}'$ and the first letter of $\mathrm{val}(A_i')$ is b. Note that as $a \notin F_1^{\mathrm{down}}$ is labelling the parent node of an occurrence of A_i' in $\mathbb{G}'$, A_i did not pop up a letter. But the first letter of $\mathrm{val}(A_i')$ is $b \in F_1^{\mathrm{down}}$. So, A_i should have popped up a letter by our earlier claim, which is a contradiction.

Note that Pop introduces at most two new letters for each occurrence of a nonterminal of rank 1, so from N_1 (one letter popped up and one popped down), and at most one new letter for each occurrence of a nonterminal of rank 0, so from $N_0 \cup \widetilde{N_0}$ (as nonterminals of rank 0 cannot pop down a letter). As each letter has two units of credit, the estimation on the number of issued credit follows from (GR1)–(GR5).

Concerning the preservation of the invariants, note that Pop does not introduce new nonterminals or new occurrences of existing nonterminals (occurrences of nonterminals can be eliminated in line 4 and 10). Therefore, (GR2)–(GR4) are preserved. Moreover, also the form of the productions guaranteed by (GR1) and (GR5) cannot be spoiled, so (GR1) and (GR5) are preserved as well. □

Hence, to simulate $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression on $\mathbb{G}$ it is enough to first uncross all 2-chains from $F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ and then compress them all using $\mathsf{TreeUnaryComp}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathbb{G})$.

Lemma 13. *Let $\mathbb{G}$ be a handle grammar and let*

$$\mathbb{G}' = \mathsf{TreeUnaryComp}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathsf{Pop}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathbb{G})).$$

Then the following hold:

- $\mathrm{val}(\mathbb{G}') = \mathsf{TreeUnaryComp}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathrm{val}(\mathbb{G}))$
- *If $\mathbb{G}$ satisfies (GR1)–(GR5), then so does $\mathbb{G}'$.*
- *The grammar $\mathbb{G}'$ has at most the same number of occurrences of nonterminals of each rank as $\mathbb{G}$.*
- *At most two new occurrences of letters are introduced for each occurrence of a nonterminal of rank 1, and at most one new occurrence of a letter is introduced for each occurrence of a nonterminal of rank 0. In particular, if $\mathbb{G}$ satisfies (GR1)–(GR5) then at most $4g_1 + 2g_0 + 2\widetilde{g_0}$ units of credit are issued during the computation of $\mathbb{G}'$.*
- *The issued credit and the credit released by* TreeUnaryComp *cover the representation cost of fresh letters as well as their credit in $\mathbb{G}'$.*

Proof. By Lemma 12, every chain pattern from $F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ is non-crossing in $\mathsf{Pop}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathbb{G})$. We get

$$\begin{aligned}
\mathrm{val}(\mathbb{G}') &= \mathrm{val}(\mathsf{TreeUnaryComp}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathsf{Pop}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathbb{G}))) \\
&\overset{\text{Lemma 11}}{=} \mathsf{TreeUnaryComp}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathrm{val}(\mathsf{Pop}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathbb{G}))) \\
&\overset{\text{Lemma 12}}{=} \mathsf{TreeUnaryComp}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathrm{val}(\mathbb{G})).
\end{aligned}$$

Moreover, at most $4g_1 + 2g_0 + 2\widetilde{g_0}$ units of credit are issued and this is twice the number of occurrences of new letters in the grammar. By Lemma 11 and 12 no new occurrences of nonterminals are introduced. By Lemma 11 the cost of representing new letters introduced by TreeUnaryComp is covered by the released credit. Finally, both TreeUnaryComp and Pop preserve the invariants (GR1)–(GR5). □

Since by Lemma 9 we apply at most $\mathcal{O}(\log n)$ many $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compressions (for different sets F_1^{up} and F_1^{down}) and by (GR3)–(GR4) we have $g_0 \leq n_0$, $\widetilde{g_0} \leq n_1(r-1)$ and $g_1 \leq n_1$, we obtain.

Corollary 1. *$(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression issues in total $\mathcal{O}((n_0 + n_1 r)\log n)$ units of credit during all modifications of $\mathbb{G}$.*

4.4. Chain compression

Our notations and analysis for chain compression is similar to those for $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression. In order to simulate chain compression on $\mathbb{G}$ we want to apply TreeChainComp (Algorithm 1) to the right-hand sides of $\mathbb{G}$. This works as long as there are no crossing chains: A unary letter *a has a crossing chain* in a rule $A_i \to \alpha_i$ if aa has a crossing occurrence in α_i, otherwise a has no crossing chain. As for $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression, when there are no crossing chains, we apply TreeChainComp to the right-hand sides of $\mathbb{G}$. We denote with $\mathsf{TreeChainComp}(F_1, \mathbb{G})$ the grammar obtained by applying TreeChainComp to all right-hand sides of $\mathbb{G}$.

Lemma 14. *Let $\mathbb{G}$ be a handle grammar and $\mathbb{G}' = \mathsf{TreeChainComp}(F_1, \mathbb{G})$. Then the following hold:*

- *If no unary letter from F_1 has a crossing chain in a rule of $\mathbb{G}$, then*

$$\mathrm{val}(\mathbb{G}') = \mathsf{TreeChainComp}(F_1, \mathrm{val}(\mathbb{G})).$$

- *The grammar $\mathbb{G}'$ has the same number of occurrences of nonterminals of each rank as $\mathbb{G}$.*
- *If $\mathbb{G}$ satisfies (GR1)–(GR5), then so does $\mathbb{G}'$.*

The proof is similar to the proof of Lemma 11 and so it is omitted. Note that so far we have neither given a bound on the amount of issued credit nor on the representation cost for the new letters a_ℓ introduced by TreeChainComp. Let us postpone these points and first show how to ensure that no letter has a crossing chain. The solution is similar to Pop: Suppose for instance that a has a crossing chain due to (CR1), i.e., some aA_i is a subpattern in a right-hand side and $\mathrm{val}(A_i)$ begins with a. Popping up a does not solve the problem, since after popping, $\mathrm{val}(A_i)$ might still begin with a. Thus, we keep on popping up until the first letter of $\mathrm{val}(A_i)$ is not a, see Figure 10. In order to do this in one step we need some notation: We say that a^ℓ is the *a-prefix* of a tree (or context) t if $t = a^\ell t'$ and the first letter of t' is not a (here t' might be the trivial context y), see Figure 9. In this terminology,

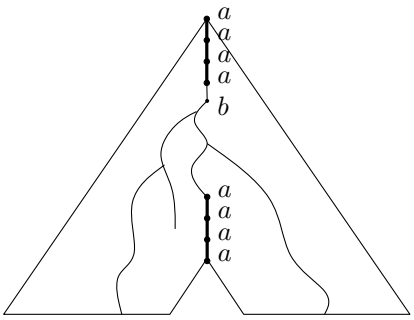

Figure 9: A context with its a-prefix and a-suffix depicted.

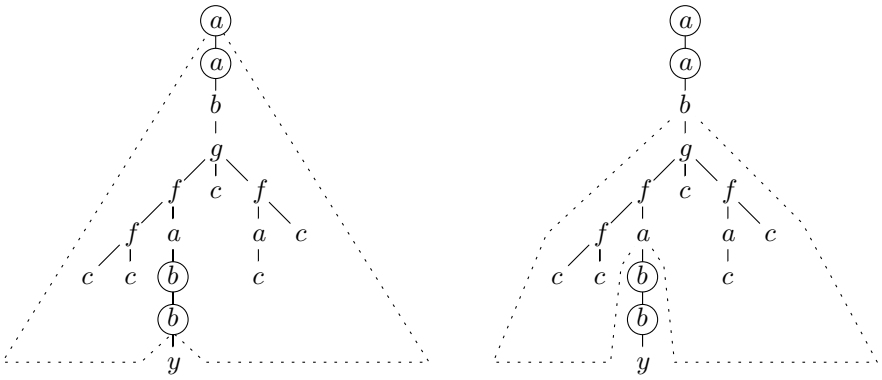

Figure 10: A tree before and after RemCrChains. Affected nodes are encircled, the dotted lines enclose the patterns generated by a nonterminal.

we remove the a-prefix of $\mathrm{val}(A_i)$. Similarly, we say that a^ℓ is the *a-suffix* of a context $t(y)$ if $t = t'(a^\ell(y))$ for a context $t'(y)$ and the last letter of t' is not a (again, t' might be the trivial context y and then a^ℓ is also the a-prefix of t). The following algorithm RemCrChains eliminates crossing chains from $\mathbb{G}$.

Algorithm 7 RemCrChains($\mathbb{G}$): removing crossing chains.

```
1: for i ← 1..m − 1 do
2:     if the first letter a of val(A_i) is unary then
3:         let p be the length of the a-prefix of α_i
4:         if α_i = a^p then
5:             replace A_i in all right-hand sides by a^p
6:         else
7:             remove a^p from the beginning of α_i
8:             replace A_i by a^p A_i in all right-hand sides
9:     if A_i ∈ N_1 and the last letter b of val(A_i) is unary then
10:        let s be the length of the b-suffix of α_i
11:        if α_i = b^s then
12:            replace A_i in all right-hand sides by b^s
13:        else
14:            remove b^s from the end of α_i
15:            replace A_i by A_i b^s in all right-hand sides
```

Lemma 15. *Let $\mathbb{G}$ be a handle grammar and $\mathbb{G}' =$ RemCrChains$(\mathbb{G})$. Then the following hold:*

- $\mathrm{val}(A_m) = \mathrm{val}(A'_m)$ *and hence* $\mathrm{val}(\mathbb{G}) = \mathrm{val}(\mathbb{G}')$.
- *No unary letter has a crossing chain in $\mathbb{G}'$.*
- *If $\mathbb{G}$ satisfies (GR1)–(GR5), then so does $\mathbb{G}'$.*
- *The grammar $\mathbb{G}'$ has at most the same number of occurrences of nonterminals of each rank as $\mathbb{G}$.*

Proof. First observe that whenever we remove the a-prefix a^{p_i} from the rule for A_i we replace each occurrence of A_i by $a^{p_i}A_i$ and similarly for b-suffixes. Hence, as long as A_j is not yet considered, it defines the same tree as in the input tree. In particular, after RemCrChains we have $\mathrm{val}(A'_m) = \mathrm{val}(A_m)$, as we do not pop prefixes and suffixes from A_m.

Next, we show that when RemCrChains considers A_i, then p from line 3 is the length of the a-prefix of $\mathrm{val}(A_i)$ (similarly, s from line 10 is the length of the a-suffix of $\mathrm{val}(A_i)$). Suppose that this is not the case and consider A_i with smallest i which violates the statement. Clearly $i > 1$ since there are no nonterminals in the right-hand side for A_1. Let a^k be the a-prefix of $\mathrm{val}(A_i)$. We have $p < k$. The symbol below a^p in α_i (which must exist because otherwise $\mathrm{val}(A_i) = a^p$) cannot be a letter (as the a-prefix of $\mathrm{val}(A_i)$ is not a^p), so it is a nonterminal A_j with $j < i$. The first letter of $\mathrm{val}(A_j)$ must be a. Let $a^{k'}$ be the a-prefix of $\mathrm{val}(A_j)$. By induction, A_j popped up $a^{k'}$, and at the time when A_i is considered, the first letter of $\mathrm{val}(A_j)$ is different from a. Hence, the a-prefix of $\mathrm{val}(A_i)$ is exactly a^p, a contradiction.

As a consequence of the above statement, if aA'_i occurs in a right-hand side of the output grammar $\mathbb{G}'$, then a is not the first letter of $\mathrm{val}(A'_i)$. This shows that (CR1) cannot hold for

a chain pattern aa. The conditions (CR2) and (CR3) are handled similarly. So there are no crossing chains after RemCrChains.

The arguments for the last two points of the lemma are the same as in the proof of Lemma 12 and therefore omitted. □

So chain compression is done by first running RemCrChains and then TreeChainComp on the right-hand sides of $\mathbb{G}$.

Lemma 16. *Let $\mathbb{G}$ be a handle grammar and $\mathbb{G}' = \mathsf{TreeChainComp}(F_1, \mathsf{RemCrChains}(\mathbb{G}))$. Then, the following hold:*

- *If $\mathbb{G}$ satisfies (GR1)–(GR5), then so does $\mathbb{G}'$.*
- $\mathrm{val}(\mathbb{G}') = \mathsf{TreeChainComp}(F_1, \mathrm{val}(\mathbb{G}))$
- *During the computation of $\mathbb{G}'$ from $\mathbb{G}$ at most two new occurrences of letters are introduced for each occurrence of a nonterminal of rank 1, and at most one occurrence of a letter is introduced for each occurrence of a nonterminal of rank 0. In particular, if $\mathbb{G}$ satisfies (GR3)–(GR4) then at most $4g_1 + 2g_0 + 2\widetilde{g_0}$ units of credit are issued in the computation of $\mathbb{G}'$ and this credit is used to pay the credit for the fresh letters a_ℓ in the grammar introduced by* TreeChainComp *(but not their representation cost).*

Proof. We shall comment only on the amount of new letters and the issued credit, as the rest follows from Lemma 14 and 15. Note that the arbitrarily long chains popped by RemCrChains are compressed into single letters by TreeChainComp. Hence, as for $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression, at most two letters are introduced for each occurrence of a nonterminal of rank 1 (i.e., from N_1) and at most one letter is introduced for each occurrence of a nonterminal of rank 0 (i.e., from $N_0 \cup \widetilde{N_0}$). The (GR3)–(GR4) additionally bound the number of occurrences of nonterminals, so assuming (GR3)–(GR4) yields the bound on the amount of issued credit. □

Since by Lemma 9 we apply at most $\mathcal{O}(\log n)$ many chain compressions to $\mathbb{G}$ and by (GR3)–(GR4) we have $g_0 \leq n_0$, $\widetilde{g_0} \leq n_1(r-1)$ and $g_1 \leq n_1$, we get:

Corollary 2. *Chain compression issues in total $\mathcal{O}((n_0 + n_1 r)\log n)$ units of credit during all modifications of $\mathbb{G}$.*

The total representation cost for the new letters a_ℓ introduced by chain compression is estimated separately in Section 4.6.

4.5. Leaf compression

In order to simulate leaf compression on $\mathbb{G}$ we perform similar operations as in the case of $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression: Ideally we would like to apply TreeLeafComp to each rule of $\mathbb{G}$. However, in some cases this does not return the appropriate result. We say that the pair (f, a) is a *crossing parent-leaf pair* in $\mathbb{G}$, if $f \in F_{\geq 1}$, $a \in F_0$, and one of the following cases holds:

(FC1) $f(t_1, \ldots, t_\ell)$ is a subtree of some right-hand side of $\mathbb{G}$, where for some j we have $t_j = A_k$ and $\mathrm{val}(A_k) = a$.

(FC2) For some $A_i \in N_1$, $A_i(a)$ is a subtree of some right-hand side of $\mathbb{G}$ and the last letter of $\mathrm{val}(A_i)$ is f.

(FC3) For some $A_i \in N_1$ and $A_k \in N_0$, $A_j(A_k)$ is a subtree of some right-hand side of $\mathbb{G}$, the last letter of $\mathrm{val}(A_i)$ is f, and $\mathrm{val}(A_k) = a$.

When there is no crossing parent-leaf pair, we proceed as in the case of any of the two previous compressions: We apply $\mathsf{TreeLeafComp}$ to each right-hand side of a rule. We denote the resulting grammar with $\mathsf{TreeLeafComp}(F_{\geq 1}, F_0, \mathbb{G})$.

Lemma 17. *Let $\mathbb{G}$ be a handle grammar and $\mathbb{G}' = \mathsf{TreeLeafComp}(F_{\geq 1}, F_0, \mathbb{G})$. Then, the following hold:*

- *If there is no crossing parent-leaf pair in $\mathbb{G}$, then $\mathrm{val}(\mathbb{G}') = \mathsf{TreeLeafComp}(F_{\geq 1}, F_0, \mathrm{val}(\mathbb{G}))$.*
- *The cost of representing new letters and the credits for those letters are covered by the released credit.*
- *If $\mathbb{G}$ satisfies (GR1)–(GR5), then so does $\mathbb{G}'$.*
- *The grammar $\mathbb{G}'$ has the same number of occurrences of nonterminals of each rank as $\mathbb{G}$.*

Proof. Most of the proof follows similar lines as the proof of Lemma 11, but there are some small differences.

Let us first prove that $\mathrm{val}(\mathbb{G}') = \mathsf{TreeLeafComp}(F_{\geq 1}, F_0, \mathrm{val}(\mathbb{G}))$ under the assumption that there is no crossing parent-leaf pair in $\mathbb{G}$. As in the proof of Lemma 11 it suffices to show that $\mathrm{val}(\mathbb{G}')$ does not contain a subtree of the form $f(t_1, \ldots, t_k)$ with $f \in F_{\geq 1}$ such that there exist positions $1 \leq i_1 < i_2 < \cdots < i_\ell \leq k$ ($\ell \geq 1$) and constants $a_1, \ldots, a_\ell \in F_0$ with $t_{i_j} = a_j$ for $1 \leq j \leq \ell$ and $t_i \notin F_0$ for $i \notin \{i_1, \ldots, i_\ell\}$ (note that the new letters introduced by $\mathsf{TreeLeafComp}$ do not belong to the alphabet F). Assume that such a subtree exists in $\mathrm{val}(A'_i)$. Using induction, we deduce a contradiction. If the root f together with its children at positions $i_1, \ldots, i_\ell$ are generated by some other nonterminal A'_j occurring in the right-hand side of A'_i, then these nodes are compressed by the induction assumption. If they all occur explicitly in the right-hand side, then they are compressed by $\mathsf{TreeLeafComp}(F_{\geq 1}, F_0, \mathbb{G})$. The only remaining case is that $\mathbb{G}'$ contains a crossing parent-leaf pair. But then, since $f, a_1, \ldots, a_\ell$ are old letters, this crossing parent-leaf pair must be already present in $\mathbb{G}$, which contradicts the assumption from the lemma.

Concerning the representation cost for the new letters, observe that when f and ℓ of its children are compressed, the representation cost for the new letter is $\ell + 1$. There is at least one occurrence of f with those children in a right-hand side of $\mathbb{G}$. Before the compression these nodes held $2(\ell + 1)$ units of credit. After the compression, only two units are needed for the new node. The other $2\ell \geq \ell + 1$ units are enough to pay for the representation cost.

Concerning the preservation of the invariants, observe that no new nonterminals were introduced, so (GR2)–(GR4) are preserved. Also the form of the rules for $A_i \in \widetilde{N}_0$ cannot be altered (the only possible change affecting those rules is a replacement of ac, where $a \in F_1$ and $c \in F_0$, by a new letter $c' \in F_0$).

So it is left to show that the resulting grammar is handle. It is easy to show that after a leaf compression a handle is still a handle, with only one exception: assume we have a handle $h = f(w_1\gamma_1, \ldots, w_{j-1}\gamma_{j-1}, y, w_{j+1}\gamma_{j+1}, \ldots, w_\ell\gamma_\ell)$ followed by a constant c in the right-hand side α_i of A_i. Such a situation can only occur, if $A_i \in N_0$ and α_i is of the form vc or $uBvc$ for sequences of handles u and v, where $v = v'h$ for a possibly empty sequence of handles v' (see (HG3)). Then leaf compression merges the constant c into the f from the handle h. There are two cases: If all w_k (which are chains) are empty and all γ_k are constants from F_0, then the resulting tree after leaf compression is a constant and no problem arises. Otherwise, we obtain a tree of the form $f'(w'_1\gamma'_1, \ldots, w'_{\ell'}\gamma'_{\ell'})$, where every w'_k is a chain, and every γ'_k is either a constant or a nonterminal of rank 0. We must have $\ell' > 0$ (otherwise, this is in fact the first case). Therefore, $f'(w'_1\gamma'_1, \ldots, w'_{\ell'}\gamma'_{\ell'})$ can be written (in several ways) as a handle, followed by (a possibly empty) chain, followed by a constant or a nonterminal of rank 0. For instance, we can write the rule for A_i as $A_i \to v'f'(y, w'_2\gamma'_2, \ldots, w'_{\ell'}\gamma'_{\ell'})w'_1\gamma'_1$ or $A_i \to uBv'f'(y, w'_2\gamma'_2, \ldots, w'_{\ell'}\gamma'_{\ell'})w'_1\gamma'_1$ (depending on the form of the original rule for A_i). This rule has one of the forms from (HG3), which concludes the proof. Note that we possibly add a second nonterminal to the right-hand side of $A_i \in N_0$ in the second case (as γ'_1 can be a non-terminal), which is allowed in (HG3). □

If there are crossing parent-leaf pairs, then we uncross them all by a generalisation of the Pop procedure. Observe that in some sense we already have a partition: We want to pop up letters from F_0 and pop down letters from $F_{\geq 1}$. The latter requires some generalisation, becasue when we pop down a letter, it may have rank greater than 1 and so we need to in fact pop a whole handle. This adds new nonterminals to $\mathbb{G}$ as well as a large number of new letters and hence a large amount of credit, so we need to be careful. There are two crucial details:

- When we pop down a whole handle $h = f(t_1, \ldots, t_k, y, t_{k+1}, \ldots, t_\ell)$, we add to the set $\widetilde{N}_0$ fresh nonterminals for all trees t_i that are non-constants, replace these t_i in h by their corresponding nonterminals and then pop down the resulting handle. In this way on one hand we keep the issued credit small and on the other no new occurrence of nonterminals from $N_0 \cup N_1$ are created.

- We do not pop down a handle from every nonterminal, but do it only when it is needed, i.e., if for $A_i \in N_1$ one of the cases (FC2) or (FC3) holds. This allows preserving (GR5). Note that when the last symbol in the rule for A_i is not a handle but another nonterminal, this might cause a need for recursive popping. So we perform the whole popping down in a depth-first-search style.

Our generalised popping procedure is called GenPop (Algorithm 8) and is shown in Figure 11.

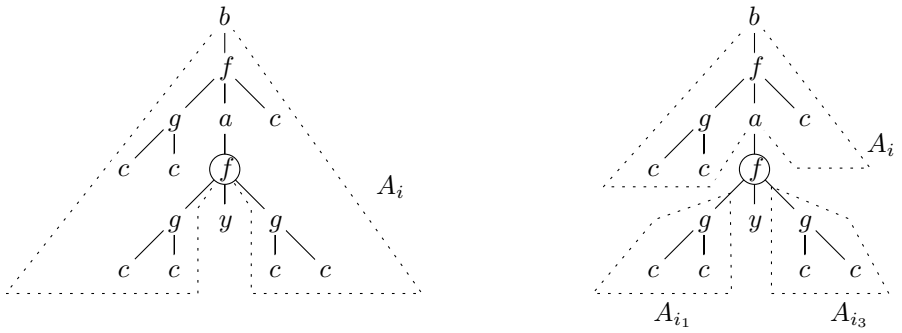

Figure 11: Popping down during uncrossing a crossing parent-leaf pair. The popped node is encircled, and dotted lines enclose patterns generated by nonterminals.

Lemma 18. *Let $\mathbb{G}$ be a handle grammar and $\mathbb{G}' = \mathsf{GenPop}(F_{\geq 1}, F_0, \mathbb{G})$. Then, the following hold:*

- $\mathrm{val}(A_m) = \mathrm{val}(A'_m)$ *and hence* $\mathrm{val}(\mathbb{G}) = \mathrm{val}(\mathbb{G}')$.
- *If $\mathbb{G}$ satisfies (GR1)–(GR5), then so does $\mathbb{G}'$.*
- *The grammar $\mathbb{G}'$ has at most the same number of occurrences of nonterminals of rank 1 as $\mathbb{G}$.*
- *The grammar $\mathbb{G}'$ has no crossing parent-leaf pair.*
- *During the computation of $\mathbb{G}'$ from $\mathbb{G}$ at most one new occurrence of a letter is introduced for each occurrence of a nonterminal of rank 0 and at most r new occurrences of letters are introduced for each occurrence of a nonterminal of rank 1. In particular, if $\mathbb{G}$ satisfies (GR3)–(GR4) then at most $2g_1 r + 2g_0 + 2\widetilde{g_0}$ units of credit are issued.*

Proof. The identity $\mathrm{val}(A_m) = \mathrm{val}(A'_m)$ follows as for Pop. Next, we show that (GR1)–(GR5) are preserved: so, assume that $\mathbb{G}$ satisfies (GR1)–(GR5). Replacing nonterminals by constants and popping down handles cannot turn a handle grammar into one that is not a handle grammar, so (GR1) is preserved. The number of nonterminals in N_0 and N_1 does not increase, so (GR2) also holds. Concerning (GR3), observe that no new occurrences of nonterminals from N_1 are produced and that new occurrences of nonterminals from N_0 can be created only in line 15, when a rule $A_{i_j} \to t_j$ is added to $\mathbb{G}$ (t_j may end with a nonterminal from N_0). However, immediately before, in line 12, we removed one occurrence of t_j from $\mathbb{G}$, so the total count is the same. Hence (GR3) holds.

The rules for the new nonterminals $A_{i_j} \in \widetilde{N_0}$ that are added in line 16 are of the form $A_{i_j} \to t_j$, where $f(t_1, \ldots, t_k, y, t_{k+1}, \ldots, t_\ell)$ was a handle. So, by the definition of a handle, every t_j is either of the form wc or wA_k, where w is a string of unary letters, c a constant, and $A_k \in N_0 \cup \widetilde{N_0}$. Hence, the rule for A_{i_j} is of the form required in (GR5) and thus (GR5) is preserved.

It remains to show (GR4), i.e., the bound on the number of occurrences of nonterminals from $\widetilde{N_0}$, which is the only non-trivial task. When we remove the handle $f(t_1, \ldots, t_k, y, t_{k+1}, \ldots, t_\ell)$

Algorithm 8 $\mathsf{GenPop}(F_{\geq 1}, F_0, \mathbb{G})$: uncrossing parent-leaf pairs

1: **for** $i \leftarrow 1\,..\,m-1$ **do** ▷ popping up letters from F_0
2: **if** $\alpha_i = a \in F_0$ **then**
3: replace each A_i in the right-hand sides by a
4: **for** $i \leftarrow m-1\,..\,1$ **do**
5: **if** $A_i(a)$ with $a \in F_0$ occurs in some rule **then**
6: mark A_i ▷ we need to pop down a handle from A_i
7: **if** A_i is marked and α_i ends with a nonterminal A_j **then**
8: mark A_j ▷ we need to pop down a handle from A_j as well
9: **for** $i \leftarrow 1\,..\,m-1$ **do**
10: **if** A_i is marked **then** ▷ we want to pop down a handle from A_i
11: let α_i end with handle $f(t_1, \ldots, t_k, y, t_{k+1}, \ldots, t_\ell)$ ▷ α_i must end with a handle
12: remove this handle from α_i
13: **for** $j \leftarrow 1\,..\,\ell$ **do**
14: **if** $t_j \notin F_0$ **then** ▷ i.e., it is not a constant
15: create a rule $A_{i_j} \to t_j$ for a fresh nonterminal A_{i_j}
16: add A_{i_j} to $\widetilde{N}_0$
17: $\gamma_j := A_{i_j}$
18: **else**
19: $\gamma_j := t_j$
20: replace each $A_i(t)$ in a right-hand side by $A_i(f(\gamma_1, \ldots, \gamma_k, t, \gamma_{k+1}, \ldots, \gamma_\ell))$

from the rule for A_i and introduce new nonterminals $A_{i_1}, \ldots, A_{i_\ell}$ then we say that A_i *owns* those new nonterminals (note that $A_i \in N_1$).[5] Furthermore, when we replace an occurrence of $A_i(t)$ in a right-hand side by $A_i(f(A_{i_1}, \ldots, A_{i_k}, t, A_{i_{k+1}}, \ldots, A_{i_\ell}))$ in line 20, those new occurrences of $A_{i_1}, \ldots, A_{i_\ell}$ are *owned* by this particular occurrence of A_i. If the owning nonterminal (or its occurrence) is later removed from the grammar, the owned (occurrences of) nonterminals get *disowned* and they remain so till they get removed.

The crucial technical claim is that one occurrence of a nonterminal owns at most $r-1$ occurrences of nonterminals, here stated in a slightly stronger form:

Claim 2. When an occurrence of $A_i \in N_1$ creates new occurrences of nonterminals (in line 20) from $\widetilde{N}_0$, then right before it does not own any occurrences of other nonterminals from $\widetilde{N}_0$.

This is shown in a series of simpler claims.

Claim 3. For a fixed nonterminal $A_i \in N_1$, every occurrence of A_i owns occurrences of the same nonterminals $A_{i_1}, \ldots, A_{i_\ell}$.

This is obvious: We assign occurrences of the same nonterminals $A_{i_1}, \ldots, A_{i_\ell}$ to each occurrence of A_i in line 20 and the only way that such an occurrence ceases to exist is when

[5]Some t_j might be constants and are not replaced by new nonterminals A_{i_j}. For notational simplicity we assume here that no t_j is a constant, which in some sense is the worst case.

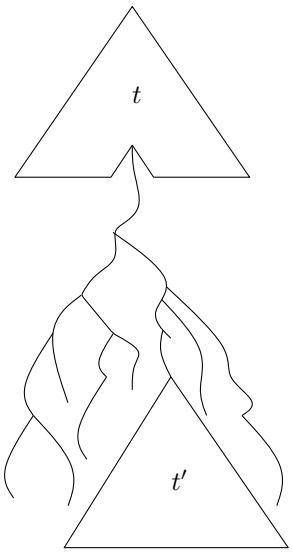

Figure 12: The context t dominates the tree t'.

A_{i_j} is replaced with a constant. But this happens for all occurrences of A_{i_j} at the same time.

In order to formulate the next claim, we need some notation: we say that an occurrence of a subcontext $t(y)$ of T *dominates* an occurrence of the subtree t' of T, if T can be written as $T = C_1(t(C_2(t')))$, where t and t' refer here to the specific occurrences of t and t', respectively, see Figure 12 for an illustration.

Claim 4. When A_i owns A_{i_j} then each subcontext generated by A_i in T dominates a subtree generated by A_{i_j}.

This is true right after the introduction of an owned nonterminal A_{i_j}: Each occurrence of A_i is replaced by $A_i(f(A_{i_1}, \ldots, A_{i_k}, t, A_{i_{k+1}}, \ldots, A_{i_\ell}))$ and this occurrence of A_i owns the occurrence of A_{i_j} in this particular $f(A_{i_1}, \ldots, A_{i_k}, t, A_{i_{k+1}}, \ldots, A_{i_\ell})$. What can change? Compression of letters does not affect dominance, as we always compress subtrees that are either completely within $\mathrm{val}(A_i)$ or completely outside $\mathrm{val}(A_i)$ and the same applies to each A_{i_j}. When popping up from A_{i_j} then the new tree generated by this occurrence of A_{i_j} is a subtree of the previous one, so the dominance is not affected. When popping up or popping down from A_i, then the new context is a subcontext of the previous one, so dominance is also not affected (assuming that A_i exists afterwards). Hence the claim holds.

Claim 5. When A_i is marked by GenPop, then T contains a subtree of the form $t(a)$ for a constant symbol a, where the subcontext t is generated by an occurrence of A_i.

If A_i was marked because $A_i(a)$ occurs in some rule then this is obvious, otherwise it was marked because it is the last nonterminal in the right-hand side of some A_j which is also marked (and $j > i$). By induction we conclude that T contains a subtree of the form $t(a)$ for a constant symbol a, where the subcontext t is generated by an occurrence of A_j. But as A_i is the last symbol in the right-hand side for A_j, the same is true for A_i.

Getting back to the proof of Claim 2, suppose that we create new occurrences of the nonterminals $A_{i_1}, \ldots, A_{i_\ell}$ in line 20 and right before line 20, A_i already owns a nonterminal A_q. Then A_i must be marked and so by Claim 5 we know that T contains a subtree of the form $t(a)$ for a constant a, where the subcontext t is generated by an occurrence of A_i. By Claim 3 this occurrence of A_i owns an occurrence of A_q. Then by Claim 4 A_q must produce the constant a. But this is not possible, since in line 1 we eliminate all nonterminals that generate constants, and there is no way to introduce a nonterminal that produces a constant. So, we derived a contradiction and thus Claim 2 holds.

Using Claim 2 we can show the bound from (GR4) on the number of occurrences of nonterminals from $\widetilde{N_0}$. The bound clearly holds as long as there are no disowned nonterminals: Each occurrence of a nonterminal from $\widetilde{N_0}$ is owned by an occurrence of a nonterminal from N_1. By Claim 2 at most $r-1$ of them are owned by an occurrence of a nonterminal from N_1, so there are at most $g_1(r-1)$ such occurrence, where $g_1 \leq n_1$ by (GR3). As the second subclaim we show that there are at most $(n_1 - g_1)(r-1)$ disowned occurrences of nonterminals from $\widetilde{N_0}$, which finally shows (GR4). This is shown by induction and clearly holds when there are no disowned nonterminals. By (GR3) the number g_1 of occurrences of nonterminals from N_1 never increases. So it is left to consider what happens, when a nonterminal gets disowned. Assume that it was owned by $A_i \in N_1$ and now this A_i is removed from $\mathbb{G}$. Thus g_1 decreases by 1 and we know, from Claim 2 that A_i owns at most $r-1$ nonterminals, which yields the claim.

Concerning the occurrences of new letters and their credit: We introduce at most one letter for each occurrence of a nonterminal of rank 0 (i.e., from $N_0 \cup \widetilde{N_0}$) during popping up and at most r letters for each occurrence of a nonterminal of rank 1 (i.e., from N_1) during popping down. As (GR3)–(GR4) give a bound on number of occurrences of nonterminals, assuming them yields that at most $2rg_1 + 2g_0 + 2\widetilde{g_0}$ units of credit are issued.

Finally, we show that $\mathbb{G}' = \mathsf{GenPop}(F_{\geq 1}, F_0, \mathbb{G})$ does not contain crossing parent-leaf pairs. Observe that after the loop in line 1 there are no nonterminals A_i such that $\mathrm{val}(A_i) \in F_0$. Afterwards, we cannot create a nonterminal that evaluates to a constant in F_0. Hence there can be no crossing parent-leaf pair that satisfies (FC1) or (FC3).

In order to rule out (FC2), we proceed with a series of claims. We first claim that if A_i is marked then in line 11 indeed the last symbol in the rule $A_i \to \alpha_i$ is a handle (so it can be removed in line 12). Suppose this is wrong and let A_i be the nonterminal with the smallest i for which this does not hold. As a first technical step observe that if some A_j is marked then $A_j \in N_1$: Indeed, if $A_j(a)$ occurs in a rule of $\mathbb{G}$ then clearly $A_j \in N_1$ and if A_k is the last nonterminal in the rule for $A_j \in N_1$ then $A_k \in N_1$ as well. Hence $A_i \in N_1$. So the last symbol in the rule for A_i is either a nonterminal $A_j \in N_1$ with $j < i$ or a handle. In the latter case we are done as there is no way to remove this handle from the rule for A_i before A_i is considered in line 11. In the former case observe that A_j is also marked. By the minimality of i, when A_j is considered in line 11, it ends with a handle $f(t_1, \ldots, t_k, y, t_{k+1}, \ldots, t_\ell)$. Hence the terminating $A_j(y)$ in the right-hand side for A_i is replaced by $A_j(f(\gamma_1, \ldots, \gamma_k, y, \gamma_{k+1}, \ldots, \gamma_\ell))$ and there is no way to remove the handle $f(\gamma_1, \ldots, \gamma_k, y, \gamma_{k+1}, \ldots, \gamma_\ell)$ from the end until A_i is considered in line 12.

Finally, suppose that there is a crossing parent-leaf pair because of the situation (FC2) after GenPop, i.e., $A_i(a)$ occurs in some right-hand side and the last letter of $\mathrm{val}(A_i)$ is f. Then in particular we did not pop down a letter from A_i, so by the earlier claim A_i was not marked. But $A_i(a)$ occurs in the right-hand already after the loop in line 1, because a cannot be introduced after the loop. So we should have marked A_i, which is a contradiction. □

So in case of leaf compression we can proceed as in the case of $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression and chain compression: we first uncross all parent-leaf pairs and then compress each right-hand side independently.

Lemma 19. *Let $\mathbb{G}$ be a handle grammar and $\mathbb{G}' = \mathsf{TreeLeafComp}(F_{\geq 1}, F_0, \mathsf{GenPop}(F_{\geq 1}, F_0, \mathbb{G}))$. Then, the following hold:*

- $\mathrm{val}(\mathbb{G}') = \mathsf{TreeLeafComp}(F_{\geq 1}, F_0, \mathrm{val}(\mathbb{G}))$
- *The grammar $\mathbb{G}'$ has at most the same number of occurrences of nonterminals of rank 1 as $\mathbb{G}$.*
- *If $\mathbb{G}$ satisfy (GR1)–(GR5), then so does $\mathbb{G}'$.*
- *During the computation of $\mathbb{G}'$ from $\mathbb{G}$ at most one new occurrence of a letter is introduced for each occurrence of a nonterminal of rank 0 and at most r new occurrences of letters are introduced for each occurrence of a nonterminal of rank 1. In particular, if $\mathbb{G}$ satisfies (GR3)–(GR4) then at most $2g_1r + 2g_0 + 2\widetilde{g_0}$ units of credit are issued.*
- *The issued credit and the credit released by* TreeLeafComp *cover the representation cost of fresh letters as well as their credit in $\mathbb{G}'$.*

Proof. This is a combination of Lemma 17 and 18: By Lemma 18, GenPop eliminates all crossing parent-leaf pairs and introduces at most one letter for each occurrence of nonterminal of rank 0 and at most r letters for each occurrence of a nonterminal of rank 1. The (GR3)–(GR4) give a bound on number of occurrences of nonterminals, which leads to bound $2g_1r + 2g_0 + 2\widetilde{g_0}$ of credit. Then by Lemma 17, TreeLeafComp ensures that $\mathrm{val}(\mathbb{G}') = \mathsf{TreeLeafComp}(F_{\geq 1}, F_0, \mathrm{val}(\mathbb{G}))$. Furthermore the credit of the new letters and the representation cost is covered by the credit released by TreeLeafComp. Finally, both subprocedures preserve (GR1)–(GR5) and do not introduce occurrences of rank 1 nonterminals. □

By Lemma 9 we apply at most $\mathcal{O}(\log n)$ many leaf compressions to $\mathbb{G}$. By (GR3)–(GR4) we have $g_0 \leq n_0$, $\widetilde{g_0} \leq n_1(r-1)$ and $g_1 \leq n_1$. Hence, we get:

Corollary 3. *Leaf compression issues in total at most $\mathcal{O}((n_0 + n_1 r) \log n)$ units of credit during all modifications of $\mathbb{G}$.*

From Corollaries 1, 2, and 3 and the observation that the initial credit is $\mathcal{O}(|\mathbb{G}|) \leq \mathcal{O}(gr)$ we get:

Corollary 4. *The whole credit issued during all modifications of $\mathbb{G}$ is in $\mathcal{O}(gr + (n_0 + n_1 r) \log n)$.*

4.6. Calculating the cost of representing letters in chain compression

The issued credit is enough to pay the two units of credit for every letter introduced during popping, whereas the released credit covers the cost of representing the new letters introduced by $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression and leaf compression. However, the released credit *does not* cover the cost of representation for letters created during chain compression. The appropriate analysis is presented in this section. The overall plan is as follows: Firstly, we define a scheme of representing letters introduced by chain compression based on the grammar $\mathbb{G}$ and the way $\mathbb{G}$ is changed by TreeChainComp (the $\mathbb{G}$-*based representation*). Then, we show that for this scheme the representation cost is bounded by $\mathcal{O}((n_0+n_1 r)\log n)$. Lastly, it is proved that the actual representation cost for the letters introduced by chain compression during the run of TtoG (as defined in Lemma 4 in Section 3.1, called the TtoG-*based representation* later on) is smaller than the $\mathbb{G}$-based one. Hence, it is bounded by $\mathcal{O}((n_0 + n_1 r)\log n)$ as well.

4.6.1. $\mathbb{G}$-based representation

We now define the $\mathbb{G}$-based representation, which is different from the representation actually used by TtoG. As noted, $\mathbb{G}$ is a tree grammar obtained by modifying the optimal grammar for the input tree. At each stage, it produces the tree currently stored by TtoG. The intuition is as follows: While $\mathbb{G}$ can produce patterns of the form a^ℓ, which have exponential length in $|\mathbb{G}|$, most patterns of this form are obtained by concatenating explicit a-chains to a shorter pattern. In such a case, the credit that is released from the explicit occurrences of a can be used to pay for the representation cost. This does not apply when the new pattern is obtained by concatenating two patterns (popped from nonterminals) inside a rule. In such a case we represent the pattern using the binary expansion at the cost of $\mathcal{O}(\log \ell)$. However, this cannot happen too often: When patterns of length $p_1, p_2, \ldots, p_\ell$ are compressed and the obtained letters are represented (at the cost of $\mathcal{O}(\log \prod_{i=1}^{\ell} p_i)$), then it can be shown that the size of the derived context in the input tree is at least $\prod_{i=1}^{\ell} p_i$, which is at most n. Thus $\sum_{i=1}^{\ell} \log p_i = \mathcal{O}(\log \prod_{i=1}^{\ell} p_i) = \mathcal{O}(\log n)$; this is formally shown later on.

Let us fix a unary letter a whose chains are compressed and represented. The $\mathbb{G}$-based representation creates a new letter for each chain pattern from a^+ that is either popped from a right-hand side during RemCrChains or is in a rule at the end of RemCrChains (i.e., after popping but before the actual replacement in TreeChainComp). Some of those chain patterns are designated as powers: fix a rule that is considered by RemCrChains. If the a-suffix popped from the first nonterminal and the a-prefix popped from the second nonterminal are part of one a-pattern (obtained after those poppings), then this a-pattern is a *power*. Note that this power may either stay in this rule or be popped (if one of the nonterminals is removed from the rule). For each chain pattern a^ℓ that is not a power we can identify another represented pattern a^k (where we allow $k = 0$ here) such that a^ℓ is obtained by concatenating explicit occurrences of a from some right-hand side to a^k.

Note that for a fixed length ℓ there may be many different occurrences of the pattern a^ℓ that are represented. In particular, some of them may be powers and some not. We arbitrarily choose one of those occurrences and the way it is created and represent a^ℓ (once) according to this choice.

Example 1. Consider the following grammar, in which only letters of arity 1 are used and they are written in a string notation: $A_0 \to a$, $A_1 \to bcaA_0$, $A_2 \to A_0abc$, $A_3 \to A_0aA_0$, $A_4 \to A_1aA_0$, $A_5 \to A_0aA_2$, $A_6 \to A_1aA_2$, $A_7 \to baaA_3$. Let us consider a-chains. The a-chain a in the right-hand side of A_0 is not a power; it is obtained by concatenating an explicit occurrence of a to ϵ. After replacing A_0 by a in all right-hand sides, we obtain the rules $A_1 \to bca\underline{a}$, $A_2 \to \underline{a}abc$, $A_3 \to \underline{a}a\underline{a}$, $A_4 \to A_1a\underline{a}$, $A_5 \to \underline{a}aA_2$, $A_6 \to A_1aA_2$, $A_7 \to baaA_3$ (new occurrences of a in right-hand sides are underlined). The occurrences of the a-chain a^2 in the right-hand sides for A_1 and A_2 are not powers, since they are obtained by concatenating an explicit a to the a popped from A_0. On the other hand, the occurrence of the a-chain a^3 in the rule for A_3 is a power, since it is obtained by concatenating a popped a, and explicit a, and a popped a. After popping a's from A_1, A_2, and A_3 we obtain the following rules for A_4, A_5, A_6, A_7: $A_4 \to A_1\underline{aa}\underline{a}$, $A_5 \to \underline{a}a\underline{aa}A_2$, $A_6 \to A_1\underline{aa}a\underline{aa}A_2$, $A_7 \to baa\underline{aaa}$. The occurrences of the a-chain a^4 (resp., a^5) in the rules for A_4 and A_5 (resp., A_6) are powers, whereas the occurrence of a^5 in the rule for A_7 is not a power. Note that the a-chain a^5 can be either represented as a power or as a non-power. ◇

We represent chain patterns as follows:

(a) For a chain pattern a^ℓ that is a power we represent a_ℓ using the binary expansion, which costs $\mathcal{O}(1 + \log \ell)$.

(b) A chain pattern a^ℓ that is not a power is obtained by concatenating $\ell - k \geq 1$ explicit occurrences of a from a right-hand side to a^k (recall that we fixed some choice in this case), in particular a_k is represented. In this case we represent a_ℓ as $a_k a^{\ell-k}$. The representation cost is $\ell - k + 1$, which is covered by the $2(\ell - k) \geq \ell - k + 1$ units of credit released from the $\ell - k \geq 1$ many explicit occurrences of a. Recall that the credit for occurrences of a fresh letter a_ℓ is covered by the issued credit, see Lemma 16. Hence the released credit is still available.

We refer to the cost in (a) as the *cost of representing a power*. As remarked above, the cost in (b) is covered by the released credit. The cost in (a) is redirected towards the rule in which this power was created. Note that this needs to be a rule for a nonterminal from $N_0 \cup N_1$, as the right-hand side of the rule needs to have two nonterminals to generate a power and by (GR5) the right-hand sides for nonterminals from $\widetilde{N_0}$ have at most one nonterminal. In Section 4.6.2 we show that the total cost redirected towards a rule during all modifications of $\mathbb{G}$ is at most $\mathcal{O}(\log n)$. Hence, the total cost in (b) is $\mathcal{O}((n_0 + n_1) \log n)$.

Example 2. In Example 1 we can represent a^5 as a power, which yields the rules $a_5 \to a_4 a$, $a_4 \to a_2 a_2$ and $a_2 \to aa$ corresponding to the binary notation of 5, or as a non-power. For the latter choice we get the rule $a_4 \to a_3 aa$, where a_3 is represented elsewhere. ◇

4.6.2. Cost of $\mathbb{G}$-based representation

We now estimate the cost of representing the letters introduced during chain compression described in the previous section. The idea is that if we redirect towards A_i the cost of representing powers of length $p_1, p_2, \ldots, p_\ell$ (which have total representation cost $\mathcal{O}(\sum_{i=1}^{\ell}(1+$

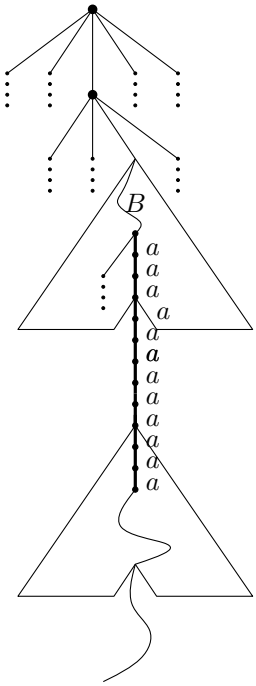

Figure 13: During the creation of a power of a the first nonterminal has an a-suffix, while the second one has an a-prefix, and the letters between them are all a's.

$\log p_i)) = \mathcal{O}(\log(\prod_{i=1}^{\ell} p_i)))$ during all chain compression steps, then in the initial grammar, A_i generates a subpattern of the input tree of size at least $p_1 \cdot p_2 \cdot \cdots \cdot p_\ell \leq n$ and so the total cost of representing powers is at most $\log n$ per nonterminal from $N_0 \cup N_1$. This is formalised in the lemma below.

Lemma 20. *The total cost of representing powers charged towards a single rule for a nonterminal from $N_0 \cup N_1$ is $\mathcal{O}(\log n)$.*

Proof. We first bound the cost redirected towards a rule for $A_i \in N_1$. There are two cases: First, after the creation of a power in the rule $A_i \to uA_jvA_kw$ one of the nonterminals A_j or A_k is removed from the grammar. But this happens at most once for the rule (there is no way to reintroduce a nonterminal from N_1 to a rule) and the cost of $\mathcal{O}(\log n)$ of representing the power can be charged to the rule. Note that here the assumption that we consider $A_i \in N_1$ is important: otherwise it could be that the second nonterminal in a right-hand side is removed and added several times, see the last sentence in the proof of Lemma 17.

The second and crucial case is when after the creation of a power both nonterminals remain in the rule. Fix such a rule $A_i \to uA_jvA_kw$, where u, v, and w are sequences of handles. Since we create a power, there is a unary letter a such that $v \in a^*$ and $\mathrm{val}(A_j)$ (respectively, $\mathrm{val}(A_k)$) has a suffix (respectively, prefix) from a^+, see Figure 13.

Fix this rule and consider all such creations of powers performed in this rule during all modifications of $\mathbb{G}$. Let the consecutive letters, whose chain patterns are compressed, be $a^{(1)}, a^{(2)}, \ldots, a^{(\ell)}$ and their lengths $p_1, p_2, \ldots, p_\ell$. Let also $b^{(s+1)}$ be the letter that replaces

the chain $(a^{(s)})^{p_s}$; note that $b^{(s+1)}$ does not need to be $a^{(s+1)}$, as there might have been some other compressions performed on the letter $b^{(s+1)}$. Then the cost of the representation charged towards this rule is bounded by

$$\mathcal{O}(\sum_{s=1}^{\ell}(1+\log p_s)) = \mathcal{O}(\sum_{s=1}^{\ell}\log p_s) \ , \tag{7}$$

as $p_s \geq 2$ for each $1 \leq s \leq \ell$.

Define the *weight* $\mathsf{w}(a)$ of a letter a as follows: in the input tree each letter has weight 1. When we replace ab by c, set $\mathsf{w}(c) = \mathsf{w}(a) + \mathsf{w}(b)$. Similarly, when a_ℓ represents a^ℓ then set $\mathsf{w}(a_\ell) = \ell \cdot \mathsf{w}(a)$, and when f' represents f with constant-labelled children $a_1, \ldots, a_\ell$, then set $\mathsf{w}(f') = \mathsf{w}(f) + \sum_{i=1}^{\ell} \mathsf{w}(a_i)$. The weight of a tree is defined as the sum of the weights of all node labels. It is easy to see that in this way $\mathsf{w}(\mathrm{val}(A_m)) = n$ is preserved during all modifications of the grammar $\mathbb{G}$.

For a rule $A_i \to uA_jvA_kw$ we say that the letters in handles from v are *between* A_j and A_k. Observe that as long as both A_j and A_k are in the rule, the maximal weight of letters between A_j and A_k cannot decrease: popping letters and handles from A_j and A_k cannot decrease this maximal weight, and the same is true for a compression step. Moreover, there is no way to remove a letter that is between A_j and A_k or to change it into a nonterminal.

Now, directly after the s-th chain compression the only letter between A_j and A_k is $b^{(s+1)}$ which has weight $p_s \cdot \mathsf{w}(a^{(s)})$ since it replaces $(a^{(s)})^{p_s}$. On the other hand, right before the $(s+1)$-th chain compression the sequence between A_j and A_k is $(a^{(s+1)})^{p_{s+1}}$. Since the maximal weight of a letter between A_j and A_k cannot decrease, we have

$$\mathsf{w}(a^{(s+1)}) \geq \mathsf{w}(b^{(s+1)}) = p_s \cdot \mathsf{w}(a^{(s)}) \ .$$

Since $\mathsf{w}(a^{(1)}) \geq 1$ it follows that $\mathsf{w}(b^{(\ell+1)}) \geq \prod_{s=1}^{\ell} p_s$. As $\mathsf{w}(b^{(\ell+1)}) \leq n$ we have

$$n \geq \prod_{s=1}^{\ell} p_s \ ,$$

and so it can be concluded that

$$\begin{aligned} \log n &\geq \log\left(\prod_{s=1}^{\ell} p_s\right) \\ &= \sum_{s=1}^{\ell} \log p_s \ . \end{aligned}$$

Therefore, the total cost $\mathcal{O}(\sum_{s=1}^{\ell} \log p_s)$, as estimated in (7), is $\mathcal{O}(\log n)$.

It is left to describe the differences, when considering nonterminals from N_0. There are two of them:

- When a power is created in a rule for a nonterminal $A_i \in N_0$, then the rule must contain two nonterminals, i.e., it must be of the form $A_i \to uA_ja^kA_k$ for a unary

symbol a, and afterwards it is of the same form. In particular we do not have to consider the case when the second nonterminal A_k is removed from the rule (as A_k is of rank 0, it cannot be replaced with a chain).

- Instead of considering the letters between A_j and A_k, we consider letters that are *below* A_j: In a rule $A_i \to uA_jvA_k$ or $A_i \to uA_jvc$, these are the letters that are in handles in v as well as the ending c.

As before, as long as A_j is in the rule, the maximal weight of letters that are below A_j can only increase (note that the rule for A_i can switch between the forms $A_i \to uA_jvA_k$ and $A_i \to uA_jvc$ many times, but this does not affect the claim).

Considering the cost of creating powers: The representation of the power that is created in the phase when A_j is removed costs at most $\mathcal{O}(\log n)$ and there is no way to bring a nonterminal from N_1 back to this rule. Hence, this cost is paid once. So, it is enough to consider the cost of powers that were created when A_j was still present in the rule. Let as in the previous case the consecutive letters, whose chain patterns are compressed, be $a^{(1)}, a^{(2)}, \ldots, a^{(\ell)}$ and let their lengths be $p_1, p_2, \ldots, p_\ell$. Let also $b^{(s+1)}$ be the letter that replaces the chain $(a^{(s)})^{p_s}$. It is enough to show that $\mathsf{w}(a^{(s+1)}) \geq p_s \cdot \mathsf{w}(a_s)$ as then the rest of the proof follows as in the case of a nonterminal from N_1.

After the s-th compression, the right-hand side of A_i has the form $uA_jb^{(s+1)}A_k$. Before the $(s+1)$-th compression, the right-hand side of A_i has the form $u'A_j(a^{(s+1)})^{p_{s+1}}A_{k'}$. By the earlier observation, the maximal weight of letters below by A_j can only increase, hence $\mathsf{w}(a^{(s+1)}) \geq \mathsf{w}(b^{(s+1)}) = \mathsf{w}((a^{(s)})^{p_s}) = p_s \cdot \mathsf{w}(a_s)$, as claimed. □

Now, the whole cost of the $\mathbb{G}$-based representation can be calculated:

Corollary 5. *The cost of the $\mathbb{G}$-based representation is $\mathcal{O}(gr + (n_0 + n_1 r)\log n)$.*

Proof. Concerning powers, we assign to each nonterminal from $N_0 \cup N_1$ a cost of $\mathcal{O}(\log n)$ by Lemma 20. There are at most $n_0 + n_1$ such nonterminals, as we do not introduce new ones. So, the total representation cost for powers is $\mathcal{O}((n_0 + n_1)\log n)$. For non-powers, the representation cost is paid from the released credit. But the released credit is bounded by the credit assigned to the initial grammar $\mathbb{G}$, which is at most $\mathcal{O}(rg)$ by Lemma 10, plus the total issued credit during all modifications of $\mathbb{G}$, which is $\mathcal{O}((n_0 + n_1 r)\log n)$ by Corollary 4. We get the statement by summing all contributions. □

4.6.3. Comparing the $\mathbb{G}$-based representation cost and the TtoG*-based representation cost*

Recall the TtoG-based representation from Lemma 4 in Section 3.1. We now show that the TtoG-based representation cost is bounded by the $\mathbb{G}$-based representation cost (note that both costs include the credit released by explicit letters). We first bound the costs of both representations by edge-weighted graphs: the total cost of a representation is bounded (up to a constant factor) by the sum of all edge weights of the corresponding graph. Then we show that we can transform the $\mathbb{G}$-based graph into the TtoG-based graph without increasing the sum of the edge weights. For an edge-weighted graph $\mathcal{G}$ let $w(\mathcal{G})$ be the sum of all edge weights.

Let us start with the $\mathbb{G}$-based representation. We define the graph $\mathcal{G}_{\mathbb{G}}$ as follows: Each chain pattern that is represented in the $\mathbb{G}$-based representation is a node of $\mathcal{G}_{\mathbb{G}}$, and edges are defined as follows:

- A power a^ℓ has an edge with weight $1+\log \ell$ to ε. Recall that the cost of representing this power is $\mathcal{O}(1+\log \ell)$.
- When a_ℓ is represented as $a_k a^{\ell-k}$ $(\ell > k)$, then node a^ℓ has an edge to a^k of weight $\ell - k$. The cost of representing a_ℓ is $\ell - k + 1 \leq 2(\ell - k)$.

From the definition of this graph the following statement is obvious:

Lemma 21. *The $\mathbb{G}$-based representation cost is in $\Theta(w(\mathcal{G}_{\mathbb{G}}))$.*

Next let us define the graph $\mathcal{G}_{\mathsf{TtoG}}$ of the TtoG-based representation: The nodes of this graph are all chain patterns that are represented in the TtoG-based representation, and there is an edge of weight $1+\log(\ell-k)$ from a^ℓ to a^k if and only if $\ell > k$ and there is no node a^q with $k < q < \ell$ (note that we may have $k = 0$).

Lemma 22. *The* TtoG*-based cost of representing the letters introduced during chain compression is in $\mathcal{O}(w(\mathcal{G}_{\mathsf{TtoG}}))$.*

Proof. Observe that this is a straightforward consequence of the way chain patterns are represented in Section 3.1: Lemma 4 guarantees that if $a^{\ell_1}, a^{\ell_2}, \ldots, a^{\ell_k}$ $(\ell_1 < \ell_2 < \cdots < \ell_k)$ are all chain patterns of the form a^+ (for a fixed unary letter a) that are represented by TtoG, then the TtoG-based representation cost for these patterns is $\mathcal{O}(\sum_{i=1}^{k}(1+\log(\ell_i - \ell_{i-1})))$, where $\ell_0 = 0$. □

We now show that $\mathcal{G}_{\mathbb{G}}$ can be transformed into $\mathcal{G}_{\mathsf{TtoG}}$ without increasing the sum of edge weights:

Lemma 23. *We have $w(\mathcal{G}_{\mathbb{G}}) \geq w(\mathcal{G}_{\mathsf{TtoG}})$.*

Proof. We transform the graph $\mathcal{G}_{\mathbb{G}}$ into the graph $\mathcal{G}_{\mathsf{TtoG}}$ without increasing the sum of edge weights. Thereby we can fix a letter a and consider only nodes of the form a^k in $\mathcal{G}_{\mathbb{G}}$ and $\mathcal{G}_{\mathsf{TtoG}}$. We start with $\mathcal{G}_{\mathbb{G}}$. Firstly, let us sort the nodes from a^* according to the increasing length. For each node a^ℓ with $\ell > 0$, we redirect its unique outgoing edge to its unique predecessor a^k (i.e., $k < \ell$ and there is no node a^q with $k < q < \ell$), and assign the weight $1+\log(\ell-k)$ to this new edge. This cannot increase the sum of edge weights:

- If a^ℓ has an edge of weight $1+\log \ell$ to ϵ in $\mathcal{G}_{\mathbb{G}}$, then $1+\log \ell \geq 1+\log(\ell-k)$.
- Otherwise it has an edge to some $a^{k'}$ $(k' \leq k)$ with weight $\ell - k'$. Then $\ell - k' \geq \ell - k \geq 1+\log(\ell-k)$, as claimed (note that $1+\log x \leq x$ for $x \geq 1$).

Let $\mathcal{G}'$ be the graph obtained from $\mathcal{G}_{\mathbb{G}}$ by this redirecting. Note that $\mathcal{G}'$ is not necessarily $\mathcal{G}_{\mathsf{TtoG}}$, because $\mathcal{G}_{\mathbb{G}}$ may contain nodes that are not present in $\mathcal{G}_{\mathsf{TtoG}}$. In other words: there might exist a chain a^ℓ which occurs in the $\mathbb{G}$-based representation but which does not occur in the TtoG-based representation. On the other hand, every node a^ℓ that occurs in $\mathcal{G}_{\mathsf{TtoG}}$ also occurs in $\mathcal{G}_{\mathbb{G}}$: if a^ℓ is represented by the TtoG-based representation, then it occurs as an a-maximal chain in T. But right before chain compression, there are no crossing chains in $\mathbb{G}$, see Lemma 15. Hence, a^ℓ occurs in some right-hand side of $\mathbb{G}$ and is therefore represented by the $\mathbb{G}$-based representation as well.

So, assume that (a^{ℓ_0}, a^{ℓ_k}) is an edge in $\mathcal{G}_{\mathsf{TtoG}}$ but in $\mathcal{G}'$ we have edges $(a^{\ell_0}, a^{\ell_1}), (a^{\ell_1}, a^{\ell_2}), \ldots, (a^{\ell_{k-1}}, a^{\ell_k})$, where $k > 1$. But the sum of the weights of these edges in $\mathcal{G}'$ (which is $\sum_{i=1}^{k} 1 + \log(\ell_{i-1} - \ell_i)$) is larger or equal than the weight of (a^{ℓ_0}, a^{ℓ_k}) in $\mathcal{G}_{\mathsf{TtoG}}$ (which is $1 + \log(\ell_0 - \ell_k)$). This follows from $1 + \log(x) + 1 + \log(y) \geq 1 + \log(x + y)$ when $x, y \geq 1$. □

Using (in this order) Lemmata 22, 23, and 21, followed by Corollary 5, we get:

Corollary 6. *The total cost of the* TtoG*-representation is* $\mathcal{O}(gr + (n_0 + n_1 r) \log n)$.

4.7. Total cost of representation

Corollary 7. *The total representation cost of the letters introduced by* TtoG *(and hence the size of the grammar produced by* TtoG*) is* $\mathcal{O}(gr + (n_0 + n_1 r) \log n) \leq \mathcal{O}(gr + gr \log n)$.

Proof. By Corollary 6 the representation cost of letters introduced by chain compression is $\mathcal{O}(gr + (n_0 + n_1 r) \log n)$, while by Lemmata 13 and 19 the representation cost of letters introduced by unary pair compression and leaf compression is covered by the initial credit (which is $\mathcal{O}(gr)$ by Lemma 10) plus the total amount of issued credit. By Corollary 4 the latter is $\mathcal{O}(gr + (n_0 + n_1 r) \log n)$. Recalling that $n_0 = \mathcal{O}(gr)$ and $n_1 = \mathcal{O}(g)$ by Lemma 10 ends the proof. □

5. Improved analysis

The naive algorithm, which simply represents the input tree T as $A_1 \to T$ results in a grammar of size n. In some extreme cases this might be better than $\mathcal{O}(gr + gr \log n)$ as guaranteed by TtoG. In fact, even a stronger fact holds: *any* 'reasonable' grammar for a tree t has size at most $2|t| - 1$, where a grammar (for t) is reasonable if

- it has no production of the form $A \to \alpha$, where $|\alpha| = 1$ and
- all its nonterminals are used in the derivation of t

(recall that the size of α does not include the parameters in it).

Lemma 24. *Let* $\mathbb{G}$ *contain no production* $A \to \alpha$ *with* $|\alpha| = 1$ *and assume that every production is used in the derivation of the tree* t *defined by* $\mathbb{G}$. *Then* $|\mathbb{G}| \leq 2|t| - 1$. *In particular, if at any point* TtoG *already paid* k *units of credit for the representation of the letters and the remaining tree is* T *then the final grammar for the input tree has size at most* $k + 2|T| - 1$

Proof. Assume that $\mathbb{G}$ has the properties from the lemma. An application of a rule $A_i \to \alpha_i$ to the current tree increases its size by $|\alpha_i|-1 \geq 1$ for each occurrence of A_i in the tree derived so far. As we assume that each production is used in the derivation, each of $|\alpha_i| - 1 \geq 1$ is added at least once and so we get $\sum_{i=1}^{m}(|\alpha_i| - 1) \leq |t|$. Thus $\sum_{i=1}^{m} |\alpha_i| \leq |t| + m$ and so it is left to estimate m. As there are m productions and each application increases the size by at least 1, and we start the derivation with a tree of size one, we get $m \leq |t| - 1$. Thus $\sum_{i=1}^{m} |\alpha_i| \leq |t| + m \leq 2|t| - 1$. □

We show that when $|T| \approx gr$ at a certain point of TtoG, then up to this point $\mathcal{O}(gr + gr\log(n/gr))$ units of credit are issued so far, where g is the size of an optimal SLCF grammar for the input tree. It follows that the size of the SLCF grammar returned by TtoG is $\mathcal{O}(gr + gr\log(n/gr))$, as claimed in Theorem 1.

Let T_i be the tree at the beginning of phase i and choose phase i such that $|T_i| \geq gr > |T_{i+1}|$ (for an input tree with at least gr symbols such an i exists, as $|T_1| = n \geq gr$ and for the 'last' i we have $|T_i| = 1$; the easy special case in which $n < gr$ follows directly from Lemma 24). We estimate the representation cost (i.e., the issued credit and the cost of the TtoG-based representation) up to phase i (inclusively). We show that this cost is bounded by $\mathcal{O}(gr + gr\log(n/gr))$, which shows the full claim of Theorem 1.

Lemma 25. *If $|T| \geq gr$ at the beginning of a phase, then till the end of this phase, the representation cost of the fresh letters introduced by* TtoG *as well as the credit of the letters in the current SLCF grammar $\mathbb{G}$ is $\mathcal{O}(gr + gr\log(n/gr))$.*

Proof. We estimate separately the amount of issued credit and the representation cost for letters replacing chains. This covers the whole representation cost for fresh letters (see Lemmata 13, 16 and 19) as well as the credit on the letters in the current SLCF grammar.

Credit

Observe first that the initial grammar $\mathbb{G}$ has at most gr credit, see Lemma 10. The input tree has size n and the one at the beginning of the phase is of size $s = |T|$. Hence, there were $\mathcal{O}(\log(n/s))$ phases before, as in each phase the size of T drops by a constant factor, see Lemma 9. Adding one phase for the current phase still yields $\mathcal{O}(\log(n/s))$ phases. As $s \geq gr$, we obtain the upper bound $\mathcal{O}(\log(n/gr))$ on the number of phases. Due to Lemmata 13, 16 and 19, during unary pair compression, chain compression and leaf compression at most $\mathcal{O}(n_0 + n_1 r)$ units of credit per phase are issued, and by Lemma 10 this is at most $\mathcal{O}(gr)$. So in total $\mathcal{O}(gr + gr\log(n/gr))$ units of credit are issued. From Lemmata 13, 16 and 19 we conclude that this credit is enough to cover the credit of all letters in $\mathbb{G}$'s right-hand sides as well as the representation cost of letters introduced during unary pair compression and leaf compression. So it is left to calculate the cost of representing chains.

Representing chains

Observe that the analysis in Section 4.6 did not assume anywhere that TtoG was carried out completely, i.e., the final grammar was returned. So we can consider the cost of the $\mathbb{G}$-based representation, the TtoG-based representation, and the corresponding graphs.

Lemma 21 still applies and the cost of the $\mathbb{G}$-based representation is $\Theta(w(\mathcal{G}_{\mathbb{G}}))$. By Lemma 22 the cost of the TtoG-based representation is $\mathcal{O}(w(\mathcal{G}_{\mathsf{TtoG}}))$. Lemma 23 shows that we can transform $\mathcal{G}_{\mathbb{G}}$ to $\mathcal{G}_{\mathsf{TtoG}}$ without increasing the sum of weights. Hence it is enough to show that the $\mathbb{G}$-based representation cost is at most $\mathcal{O}(gr + gr\log(n/gr))$.

The $\mathbb{G}$-based representation cost consists of some released credit and the cost of representing powers, see its definition. The former was already addressed (the whole issued credit is $\mathcal{O}(gr + gr\log(n/gr))$) and so it is enough to estimate the latter, i.e., the cost of representing powers.

The outline of the analysis is as follows: When a new power a^ℓ is represented, we mark some letters of the input tree (and perhaps modify some other markings). Those markings are associated with nonterminals. Formally, for a nonterminal $A_i \in N_0 \cup N_1$ we introduce the notions of an A_i-pre-power marking and A_i-in marking. Such a marking is a subset of the node set of the initial tree (note that we do not define such markings for a nonterminal $A_i \in \widetilde{N_0}$). These marking satisfy the following conditions:

(M1) Each marking contains at least two nodes and two different markings are disjoint.

(M2) For every nonterminal A_i and every $X \in \{\text{pre-power}, \text{in}\}$ there is at most on A_i-X marking.

(M3) If $p_1, p_2, \ldots, p_k \geq 2$ are the sizes of the markings (i.e., the cardinalities of the node sets), then the cost of representing powers (created up to the current phase) by the $\mathbb{G}$-based representation is $c\sum_{i=1}^{k} \log p_i$ (for some fixed constant c).

Note that in (M3) we must have $k \leq drg$ for some constant d, because $k \leq 2(|N_0| + |N_1|) \leq \mathcal{O}(gr)$ by Lemma 10.

Using (M1)–(M3) the total cost of representing powers (in the $\mathbb{G}$-based representation) can be upper-bounded by (a constant times)

$$\sum_{i=1}^{k} \log p_i \quad \text{under the constraints } k \leq drg \text{ and } \sum_{i=1}^{k} p_i \leq n \ , \tag{8a}$$

where d is some constant. Let us bound the sum $\sum_{i=1}^{k} \log p_i$ under the above constraints: Clearly, the sum is maximised for $\sum_{i=1}^{k} p_i = n$. For a fixed k and $\sum_{i=1}^{k} p_i = n$ we have $\sum_{i=1}^{k} \log p_i = \log(\prod_{i=1}^{k} p_i)$. By the inequality of arithmetic and geometric means we conclude that $\log(\prod_{i=1}^{k} p_i) \leq \log((\sum_{i=1}^{k} p_i/k)^k) = k\log(n/k)$, where the maximum $k\log(n/k)$ is achieved if each p_i is equal to n/k. Now, the term $k\log(n/k)$ is maximised for $k = n/e$ (independently of the base of the logarithm). Moreover, in the range $[0, n/e)$ the function $f(k) = k\log(n/k)$ is monotonically increasing. Hence, if $drg \leq n/e$, then, indeed, the maximal value of $\sum_{i=1}^{k} \log p_i$ under the constraints in (8a) is in

$$\mathcal{O}\left(gr + gr\log\left(\frac{n}{gr}\right)\right) \ . \tag{8b}$$

On the other hand, if $drg > n/e$, then $n \leq \mathcal{O}(rg)$ and the bound in the statement of the Lemma trivially holds.

The idea of preserving (M1)–(M3) is as follows: If a new power of length ℓ is represented, this yields a cost of $\mathcal{O}(1 + \log \ell)$. Since $\ell \geq 2$, we can treat this cost as $\mathcal{O}(\log \ell)$ and choose c in (M3) so that this is at most $c \log \ell$. Then either we add a new marking of size ℓ or we remove some marking of size ℓ' and add a new marking of size $\ell \cdot \ell'$. It is easy to see that in this way (M1)–(M3) are preserved (still, those details are repeated later in the proof).

Whenever we have to represent powers $a^{\ell_1}, a^{\ell_2}, \ldots$, for each power a^ℓ, where $\ell > 1$, we find the last (according to preorder) maximal chain pattern a^ℓ in the current tree T. It is possible that this particular a^ℓ was obtained as a concatenation of $\ell - k$ explicit letters to a^k (so, not as a power). In such a case we are lucky, as the representation of this a_ℓ is paid by the credit and we do not need to separately consider the cost of representing the power a^ℓ. Otherwise a^ℓ is a power and we add a new marking which is contained in the subcontext of the input tree that is derived from the last occurrence of a^ℓ in the current tree. Let A_i be the smallest nonterminal that derives (before RemCrChains) this last occurrence of the maximal chain pattern a^ℓ (clearly there is such non-terminal, as A_m derives it). Note that $A_i \in N_0 \cup N_1$ as otherwise this a^ℓ is not a power, since powers cannot be created inside nonterminals from $\widetilde{N}_0$. The new marking is either an A_i-pre-power marking or an A_i-in marking: If one of the nonterminals in A_i's right-hand side was removed during RemCrChains, then we add an A_i-pre-power marking (note that such a removed nonterminal is necessarily from N_1, as no nonterminal from $N_0 \cup \widetilde{N}_0$ is removed during RemCrChains). Otherwise, we add an A_i-in marking.

Claim 6. At any time, there is at most one A_i-pre-power marking in the input tree.

When an A_i-in marking is added because of a power a^ℓ, then after chain compression A_i has a rule of the form

- $A_i \to wA_j a_\ell A_k v$, where w and v are (perhaps empty) sequences of handles and $A_j, A_k \in N_1$, if $A_i \in N_1$, or
- $A_i \to wA_j a_\ell A_k$ where w is a (perhaps empty) sequence of handles, $A_j \in N_1$, and $A_k \in N_0$, if $A_i \in N_0$.

Proof. Concerning A_i-pre-power markings, let a^ℓ be the first power that causes the creation of an A_i-pre-power marking. So one nonterminal from N_1 was removed from the right-hand side for A_i and there is no way to reintroduce such a nonterminal. Hence, A_i's rule has at most one nonterminal from N_1 (when $A_i \in N_1$) or none at all (when $A_i \in N_0$). Thus, no more powers can be created in A_i's right-hand side. In particular, neither A_i-pre-power markings nor A_i-in markings will be added in the future.

Next, suppose that an A_i-in marking is added to the input tree because a new power a^ℓ is created. Thus, the last occurrence of the maximal chain pattern a^ℓ is generated by A_i but not by the nonterminals in the rule for A_i (as then, a different marking would be introduced). Since a^ℓ is a power it is obtained in the rule as the concatenation of an a-prefix and an a-suffix popped from nonterminals in the rule for A_i. The suffix needs to come from a nonterminal of rank 1. In particular this means that those two nonterminals in the rule for A_i generate parts of this last occurrence of a^ℓ and in between them only the letter a occurs. If any of those nonterminals would be removed during the chain compression for a^ℓ, then

an A_i-pre-power marking would be introduced, which is not the case. So both nonterminals remain in the rule for A_i. Hence after popping prefixes and suffixes, between those two nonterminals there is exactly a chain pattern a^ℓ, which is then replaced by a_ℓ. This yields the desired form of the rule, both in case $A_i \in N_0$ or $A_i \in N_1$. □

Consider the occurrence of a^ℓ and the 'derived' subcontext w^ℓ of the *input tree*. We show that if there exists a marking inside w^ℓ, then this marking is contained in the last occurrence of w inside the occurrence of w^ℓ.

Claim 7. Let a^ℓ be an occurrence of a maximal chain pattern, which is replaced by a_ℓ. Assume that a_ℓ derives the subcontext w^ℓ of the input tree, where w is a context. If this occurrence of w^ℓ contains a marking, then this marking is contained in the last occurrence of w inside the occurrence of w^ℓ.

Proof. Consider a marking M within w^ℓ. Assume it was created, when some b^k was replaced by b_k. As b_k is a single letter and a^ℓ derives it, each a derives at least one b_k. Then, the marking M must be contained in the subcontext derived from the last b_k (as we always create markings within the last occurrence of the chain pattern to be replaced). Clearly the last b_k can be only derived from the last a within a^ℓ. So in particular, the marking M is contained in the last w inside w^ℓ. So all markings within w^ℓ are in fact within the last w. □

We now demonstrate how to add markings to the input tree. Suppose that we replace a power a^ℓ. Note that we must have $\ell \geq 2$. Let us consider the last occurrence of this a^ℓ in the current tree T and the smallest A_i that generates this occurrence. This a^ℓ generates some occurrence of w^ℓ (for some context w) in the input tree. If this occurrence of w^ℓ contains no marking, then we simply add a marking (either an A_i-pre-power or an A_i-in marking according to the above rule) consisting of $\ell \geq 2$ arbitrarily chosen nodes within w^ℓ. In the other case, by Claim 7, we know that all markings within the occurrence of w^ℓ are contained in the last w. If one of them is the (unique, by (M2)) A_i-in marking, let us choose it. Otherwise choose any other marking in the last w. Let M be the chosen marking and let $\ell' = |M| \geq 2$. We proceed, depending on whether M is the only marking in the last w:

- M is the unique marking in the last w: Then we remove it and mark arbitrarily chosen $\ell \cdot \ell'$ nodes in w^ℓ. This is possible, as $|w| \geq \ell'$ and so $|w^\ell| \geq \ell \cdot \ell'$. Since $\log(\ell \cdot \ell') = \log \ell + \log \ell'$, (M3) is preserved, as it is enough to account for the $1 + \log \ell \leq c \log \ell$ representation cost for a^ℓ as well as the $c \log \ell'$ cost associated with the previous marking of size ℓ'.

- M is not the unique marking in the last w: Then $|w| \geq \ell' + 2$ (the '+2' comes from the other markings, which are of size at least 2, see (M1)). We first remove the chosen marking of size ℓ'. Let us calculate how many unmarked nodes are in w^ℓ afterwards: In $w^{\ell-1}$ there are at least $(\ell - 1) \cdot (\ell' + 2)$ nodes and by Claim 7 none of them belongs to a marking. In the last w there are at least ℓ' unmarked nodes (from the marking

that we removed). Hence, in total we have

$$\begin{aligned}(\ell-1)\cdot(\ell'+2)+\ell' &= \ell\ell'+2\ell-\ell'-2+\ell' \\ &= \ell\ell'+2\ell-2 \\ &> \ell\ell'\end{aligned}$$

many unmarked nodes (recall that $\ell \geq 2$). We arbitrarily choose $\ell \cdot \ell'$ many unmarked nodes and add them as a new marking. By the same argument as in the previous case, (M3) is preserved.

By the above construction, (M1) is preserved. There is one remaining issue concerning (M2): It might be that we create an A_i-in marking while there already was one, violating (M2). However, we show that if there already exists an A_i-in marking, then it is within w^ℓ (and so within the last w, by Claim 7). Hence, we could choose this A_i-in marking as the one that is removed when the new one is created. Consider the previous A_i-in marking. It was introduced when some power b^k was replaced by b_k, which, by Claim 6, became the unique letter between the first and second nonterminal in the right-hand side for A_i. Consider the last (as usual, with respect to preorder) subpattern of the input tree that is either generated by the explicit letters between nonterminals of rank 1 in the rule for A_i (when $A_i \in N_1$) or is generated by the explicit letters below the nonterminals of rank 1 (when $A_i \in N_0$) (recall from the proof of Lemma 20 that in a rule $A_i \to wA_jvA_k$ for a nonterminal of rank 0, where w and v are sequences of handles, all letters occurring in handles in v are classified as being *below* A_j). The operations performed on $\mathbb{G}$ cannot make this subpattern smaller, in fact popping letters expands it. When b_k is created, then this subpattern is generated by b_k, as by Claim 6 this is the unique letter between the nonterminals (resp., below the nonterminal). When a_ℓ is created, it is generated by a_ℓ, again by Claim 6, i.e., it is exactly w^ℓ. So in particular w^ℓ includes the A_i-in marking that was added when the power b^k was replaced by b_k.

This shows that (M1)–(M3) hold and so also the calculations in (8) hold, in particular, the representation cost of powers is $\mathcal{O}(gr + gr\log(n/gr))$. □

Now the estimations from Lemma 25 allow to prove Theorem 1.

Proof of the full version of Theorem 1. Suppose first that the input tree T has size smaller than gr. Then by Lemma 24, TtoG returns a tree of size at most $2gr-3 = \mathcal{O}(gr)$. Otherwise, consider the phase, such that before it T has size s_1 and right after it has size is s_2, where $s_1 \geq gr > s_2$. There is such a phase as in the end T has size 1 and initially it has size at least gr. Then by Lemma 25 the cost of representing letters introduced till the end of this phase is $\mathcal{O}\left(gr + gr\log\left(\frac{n}{gr}\right)\right)$. By Lemma 24 the cost of representing the remaining tree is at most $2gr - 3 = \mathcal{O}(gr)$. Hence, the size of the grammar that is returned by TtoG is at most $\mathcal{O}\left(gr + gr\log\left(\frac{n}{gr}\right)\right)$. □

6. Conclusions

We presented a linear-time grammar-based tree compressor with an approximation ratio of $\mathcal{O}(r + r \log(n/rg))$, where n is the size of the input tree T, g is the size of a minimal linear context-free tree grammar for T, and r is the maximal rank of symbols in T.

Possible future work is listed and discussed below.

Non-linear grammars

One possible direction for future research are non-linear context-free tree grammars. They are defined in the same way as linear context-free tree grammars with the only exception that parameters may occur more than once in right-hand sides. With non-linear context-free tree grammars one can achieve double exponential compression. For instance, the non-linear grammar with the productions $S \to A_1(a)$, $A_i(x) \to A_{i+1}(A_{i+1}(x))$ for $1 \leq i \leq n-1$ and $A_n(x) \to f(x,x)$ produces a perfect binary tree of height 2^n. The authors are not aware of any grammar-based tree compressor that exploits this additional succinctness of non-linear context-free tree grammars.

Graph grammars

A non-linear context-free tree grammar can be viewed as a context-free graph grammar that produces a directed acyclic graph. This graph grammar is obtained by merging all occurrences of the same parameter in a right-hand side. Recently, grammar-based graph compression via context-free graph grammars was considered in [13]. But no quantitative results, e.g., concerning the approximation ratio, have been shown so far. Perhaps techniques used here can help in developing such results.

XML trees

In contrast to trees as considered in this paper, XML trees are usually modelled using unranked (but ordered) trees, i.e. the rank of a node is not determined by its label. SLCF grammars can be used to generate such trees: we drop the assumption of the ranked alphabet, but keep the ranks for nonterminals. In this way, letters in SLCF grammars are *de facto* ranked, as each occurrence in the SLCF grammar has a fixed arity. Thus, when computing such an SLCF grammar for an unranked tree, we can artificially rank all letters and proceed as in the case of a ranked alphabet. The approximation guarantee is the same.

There are also more powerful constructs that are intended to capture XML trees, for instance forest algebras [42], which work on forests instead of trees and allow also horizontal "concatenation" of trees, and this operation yields a forest. A corresponding grammar model can, for instance, represent the tree $f(\underbrace{c,\ldots,c}_{n})$ by a grammar of size $\mathcal{O}(\log n)$, whereas this tree is incompressible by SLCF tree grammars. Approximation algorithms for such grammars have not been investigated so far. Whether the methods proposed here apply in this model remains an open question.

Unordered trees

Our method depends very little on the fact that the considered trees are ordered and it should work also in the unordered (but ranked) case: it is enough that leaf compression does not take the positions of leaves into the account.

Practical applications

While TtoG achieves the best known approximation bound, the authors doubt that it beats in practice the known heuristics, especially TreeRePair [21]. These doubts are based on the fact that in the string case, RePair by far beats the compression algorithms with the best known approximation bound.

Explicit grammar

In many problems the SLCF grammar is given explicitly and we are interested in processing it. The presented "recompression" approach can be naturally applied in this setting, but there is no bound on the size of the SLCF obtained in this way. However, we can use a similar trick as in the case of strings [25]: we have two alternating compression phases and in one of them we proceed as described in Section 3 while in the other we try to make the SLCF small. The only difference is that during pair compression we choose the partition of letters so that many occurrences of pairs in the SLCF are covered; see [25] for details.

Acknowledgements

The first author would like to thank Paweł Gawrychowski for introducing him to the topic of compressed data, for pointing out the relevant literature [43] and discussions, as well as Sebastian Maneth and Stefan Böttcher for the question of applicability of the recompression-based approach to the tree case.

[1] J. A. Storer, T. G. Szymanski, The macro model for data compression, in: R. J. Lipton, W. A. Burkhard, W. J. Savitch, E. P. Friedman, A. V. Aho (Eds.), STOC, ACM, 1978, pp. 30–39.

[2] K. Casel, H. Fernau, S. Gaspers, B. Gras, M. L. Schmid, On the complexity of grammar-based compression over fixed alphabets, in: I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, D. Sangiorgi (Eds.), ICALP, Vol. 55 of LIPIcs, Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2016, pp. 122:1–122:14. doi:10.4230/LIPIcs.ICALP.2016.122.
URL http://dx.doi.org/10.4230/LIPIcs.ICALP.2016.122

[3] J. C. Kieffer, E.-H. Yang, Sequential codes, lossless compression of individual sequences, and Kolmogorov complexity, IEEE Transactions on Information Theory 42 (1) (1996) 29–39. doi:10.1109/18.481775.

[4] N. J. Larsson, A. Moffat, Offline dictionary-based compression, in: J. A. Storer, M. Cohn (Eds.), Data Compression Conference, IEEE Computer Society, 1999, pp. 296–305. doi:10.1109/DCC.1999.755679.

[5] C. G. Nevill-Manning, I. H. Witten, Identifying hierarchical structure in sequences: A linear-time algorithm, J. Artif. Intell. Res. (JAIR) 7 (1997) 67–82. doi:10.1613/jair.374.

[6] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, A. Shelat, The smallest grammar problem, IEEE Transactions on Information Theory 51 (7) (2005) 2554–2576. doi:10.1109/TIT.2005.850116.

[7] A. Jeż, Approximation of grammar-based compression via recompression, Theoretical Computer Science 592 (2015) 115–134. doi:10.1016/j.tcs.2015.05.027.
URL http://dx.doi.org/10.1016/j.tcs.2015.05.027

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

[9] W. Rytter, Application of Lempel-Ziv factorization to the approximation of grammar-based compression, Theor. Comput. Sci. 302 (1-3) (2003) 211–222. doi:10.1016/S0304-3975(02)00777-6.

[10] H. Sakamoto, A fully linear-time approximation algorithm for grammar-based compression, J. Discrete Algorithms 3 (2-4) (2005) 416–430. doi:10.1016/j.jda.2004.08.016.

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

[12] F. Claude, G. Navarro, Fast and compact web graph representations, ACM Transactions on the Web 4 (4).

[13] S. Maneth, F. Peternek, Compressing graphs by grammars, in: ICDE, IEEE Computer Society, 2016, pp. 109–120. doi:10.1109/ICDE.2016.7498233.
URL http://dx.doi.org/10.1109/ICDE.2016.7498233

[14] G. Busatto, M. Lohrey, S. Maneth, Efficient memory representation of XML document trees, Information Systems 33 (4–5) (2008) 456–474.

[15] C. Creus, A. Gascon, G. Godoy, One-context unification with STG-compressed terms is in NP, in: A. Tiwari (Ed.), 23rd International Conference on Rewriting Techniques and Applications (RTA'12), Vol. 15 of LIPIcs, Schloss Dagstuhl — Leibniz Zentrum für Informatik, Dagstuhl, Germany, 2012, pp. 149–164. doi:http://dx.doi.org/10.4230/LIPIcs.RTA.2012.149.
URL http://drops.dagstuhl.de/opus/volltexte/2012/3490

[16] A. Gascón, G. Godoy, M. Schmidt-Schauß, Unification and matching on compressed terms, ACM Trans. Comput. Log. 12 (4) (2011) 26.

[17] M. Lohrey, S. Maneth, The complexity of tree automata and XPath on grammar-compressed trees, Theoretical Computer Science 363 (2) (2006) 196–210.

[18] M. Lohrey, S. Maneth, M. Schmidt-Schauß, Parameter reduction and automata evaluation for grammar-compressed trees, J. Comput. Syst. Sci. 78 (5) (2012) 1651–1669. doi:10.1016/j.jcss.2012.03.003.

[19] M. Schmidt-Schauß, Matching of compressed patterns with character-variables, in: A. Tiwari (Ed.), RTA, Vol. 15 of LIPIcs, Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2012, pp. 272–287.

[20] M. Schmidt-Schauß, D. Sabel, A. Anis, Congruence closure of compressed terms in polynomial time, in: FroCos, Vol. 6989 of LNCS, Springer, 2011, pp. 227–242.

[21] M. Lohrey, S. Maneth, R. Mennicke, XML tree structure compression using RePair, Inf. Syst. 38 (8) (2013) 1150–1167. doi:10.1016/j.is.2013.06.006.
URL http://dx.doi.org/10.1016/j.is.2013.06.006

[22] M. Bousquet-Mélou, M. Lohrey, S. Maneth, E. Nöth, XML compression via directed acyclic graphs, Theory Comput. Syst. 57 (4) (2015) 1322–1371. doi:10.1007/s00224-014-9544-x.
URL http://dx.doi.org/10.1007/s00224-014-9544-x

[23] D. Hucke, M. Lohrey, E. Noeth, Constructing small tree grammars and small circuits for formulas, in: V. Raman, S. P. Suresh (Eds.), FSTTCS, Vol. 29 of LIPIcs, Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2014, pp. 457–468. doi:10.4230/LIPIcs.FSTTCS.2014.457.
URL http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2014.457

[24] A. Jeż, The complexity of compressed membership problems for finite automata, Theory of Computing Systems 55 (2014) 685–718. doi:10.1007/s00224-013-9443-6.
URL http://dx.doi.org/10.1007/s00224-013-9443-6

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

[26] V. Diekert, A. Jeż, W. Plandowski, Finding all solutions of equations in free groups and monoids with involution, in: E. A. Hirsch, S. O. Kuznetsov, J. Pin, N. K. Vereshchagin (Eds.), CSR, Vol. 8476 of LNCS, Springer, 2014, pp. 1–15. doi:10.1007/978-3-319-06686-8_1.
URL http://dx.doi.org/10.1007/978-3-319-06686-8_1

[27] A. Jeż, One-variable word equations in linear time, Algorithmica 74 (2016) 1–48. doi:10.1007/s00453-014-9931-3.
[28] A. Jeż, Recompression: a simple and powerful technique for word equations, Journal of the ACM 63 (1) (2016) 4:1–4:51. doi:10.1145/2743014.
URL http://dx.doi.org/10.1145/2743014
[29] T. Akutsu, A bisection algorithm for grammar-based compression of ordered trees, Inf. Process. Lett. 110 (18–19) (2010) 815–820.
[30] P. Bille, I. L. Gørtz, G. M. Landau, O. Weimann, Tree compression with top trees, Infmormation and Computation 243 (2015) 166–177. doi:10.1016/j.ic.2014.12.012.
URL http://dx.doi.org/10.1016/j.ic.2014.12.012
[31] L. Hübschle-Schneider, R. Raman, Tree compression with top trees revisited, in: E. Bampis (Ed.), ESA, Vol. 9125 of LNCS, Springer, 2015, pp. 15–27. doi:10.1007/978-3-319-20086-6_2.
URL http://dx.doi.org/10.1007/978-3-319-20086-6_2
[32] M. Ganardi, D. Hucke, A. J. M. Lohrey, E. Noeth, Constructing small tree grammars and small circuits for formulas, CoRR 1407.4286.
URL http://arxiv.org/abs/1407.4286
[33] M. Ganardi, D. Hucke, M. Lohrey, E. Noeth, Tree compression using string grammars, in: E. Kranakis, G. Navarro, E. Chávez (Eds.), LATIN, Vol. 9644 of LNCS, Springer, 2016, pp. 590–604. doi:10.1007/978-3-662-49529-2_44.
URL http://dx.doi.org/10.1007/978-3-662-49529-2_44
[34] A. Jeż, Context unification is in PSPACE, in: E. Koutsoupias, J. Esparza, P. Fraigniaud (Eds.), ICALP, Vol. 8573 of LNCS, Springer, 2014, pp. 244–255. doi:10.1007/978-3-662-43951-7_21.
URL http://dx.doi.org/10.1007/978-3-662-43951-7_21
[35] RTA problem list, Problem 90, http://rtaloop.mancoosi.univ-paris-diderot.fr/problems/90.html (1990).
[36] G. L. Miller, J. H. Reif, Parallel tree contraction part 1: Fundamentals, in: S. Micali (Ed.), Randomness and Computation, Vol. 5, JAI Press, Greenwich, Connecticut, 1989, pp. 47–72.
[37] G. L. Miller, J. H. Reif, Parallel tree contraction, part 2: Further applications, SIAM J. Comput. 20 (6) (1991) 1128–1147. doi:10.1137/0220070.
[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms (3. ed.), MIT Press, 2009.
[39] A. V. Aho, J. E. Hopcroft, J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.
[40] M. Mitzenmacher, E. Upfal, Probability and Computing: Randomized Algorithms and Probabilistic Analysis, Cambridge University Press, 2005.
[41] S. Böttcher, R. Hartel, T. Jacobs, S. Maneth, Incremental updates on compressed XML, in: ICDE, IEEE Computer Society, 2016, pp. 1026–1037. doi:10.1109/ICDE.2016.7498310.
URL http://dx.doi.org/10.1109/ICDE.2016.7498310
[42] M. Bojańczyk, I. Walukiewicz, Forest algebras, in: J. Flum, E. Grädel, T. Wilke (Eds.), Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]., Vol. 2 of Texts in Logic and Games, Amsterdam University Press, 2008, pp. 107–132.
[43] K. Mehlhorn, R. Sundar, C. Uhrig, Maintaining dynamic sequences under equality tests in polylogarithmic time, Algorithmica 17 (2) (1997) 183–198. doi:10.1007/BF02522825.

Appendix

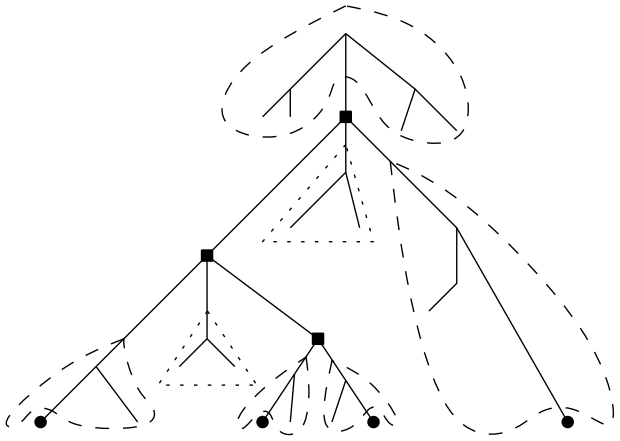

Figure .14: The partitioning of a tree depending on its parameter nodes. The parameters are the round nodes, the square nodes are the branching nodes of the spanning tree. Contexts and trees obtained after the removal of those nodes are enclosed by dashed and dotted lines, respectively.

Proof of Lemma 10 (transforming an SLCF grammar into a handle grammar).

The proof is a slight modification of the original proof of [7, Theorem 10] and it follows exactly the same lines. Let $\mathbb{G}$ be an SLCF grammar of size g.

The idea is as follows, see Figure .14. Consider a nonterminal $A(y_1, \ldots, y_k)$ of $\mathbb{G}$ and the tree $\mathrm{val}(A)$ that it generates. Within $\mathrm{val}(A)$ take the nodes representing the parameters $y_1, \ldots, y_k$ and the spanning tree (within $\mathrm{val}(A)$) for those nodes. Consider the nodes of degree at least 2 within this spanning tree, delete those nodes and delete the parameters. What is left is a collection of subtrees and subcontexts. We want to construct a grammar that has for each such subtree and subcontext a nonterminal generating it. This is done inductively on the structure of $\mathbb{G}$. As the starting nonterminal of $\mathbb{G}$ has rank 0, such a decomposition for $\mathrm{val}(\mathbb{G})$ is in fact trivial. So, in particular the constructed grammar generates $\mathrm{val}(\mathbb{G})$. Lastly, the construction guarantees that the introduced nonterminals, which are of rank 0 and 1, are expressed through each other (plus some rules introduced on the way). So the new grammar generates the same tree and it is monadic. Moreover, the rules for those nonterminals are in the form required for a handle grammar, see (HG2) and (HG3).

Formalising this approach, we say that a *skeleton tree*[6] is a pattern from $\mathcal{T}(N_0 \cup N_1 \cup F, \mathbb{Y})$, satisfying the following conditions:

(SK1) The child of a node of degree 1 can be labelled only with a letter of arity at least 2 or with a parameter.

(SK2) If f of arity at least 2 labels a node with children $v_1, \ldots v_k$, then there are $i \neq j$ such that the subtrees rooted in v_i and v_j both contain parameters.

[6]Note that our definition of a skeleton slightly differs from the one of [7, Lemma 1], but the differences are inessential.

Intuitively, the skeleton tree is what one obtains after replacing each context and tree in the tree constructed above with a nonterminal: (SK1) says that whole context is replaced with a nonterminal, while (SK2) says that only branching nodes of the spanning tree of parameters are be labelled with letters.

Our first goal is to construct for each nonterminal A of the input grammar $\mathbb{G}$ a skeleton tree sk_A together with rules for the nonterminals appearing in sk_A. These rules allow to rewrite sk_A into $\mathrm{val}(A)$; note that we do not assume that these introduced rules satisfy (SK1)–(SK2) and on the other hand, the skeletons do not satisfy (HG2)–(HG3) and they are not part of the constructed grammar, they are just means of construction. Instead, we show that in the introduced rules the nonterminals of arity 1 occur at most $\mathcal{O}(g)$ times, while nonterminals of arity 0 and letters occur at most $\mathcal{O}(rg)$ times.

As a first step, we transform the grammar into *Chomsky normal form* (CNF), which is obtained by a straightforward decomposition of rules. The rules in a CNF grammar are of two possible forms, where $A, B, C \in \mathbb{N}$ and $f \in \mathbb{F}$:

- $A(y_1, \ldots, y_k) \to f(y_1, \ldots, y_k)$ or
- $A(y_1, \ldots, y_k) \to B(y_1, \ldots, y_\ell, C(y_{\ell+1}, \ldots, y_{\ell'}), y_{\ell'+1}, \ldots, y_k)$

Note that the number of parameters can be 0. It is routine to check that any SLCF grammar $\mathbb{G}$ of size g can be transformed into an equivalent CNF grammar of size $\mathcal{O}(g)$ and with $\mathcal{O}(g)$ nonterminals [7, Theorem 5].

Given an SLCF grammar in CNF, we build bottom-up skeleton trees for its nonterminals, During this we introduce $\mathcal{O}(1)$ nonterminals per considered nonterminal, their rules have $\mathcal{O}(1)$ occurrences of nonterminals of arity 1 and at most $\mathcal{O}(r)$ occurrences of nonterminals of arity 0 and constants. Moreover, the rules for those nonterminals are in the form required by the definition of a handle grammar, see (HG2) and (HG3). All nonterminals occurring in the constructed skeletons use only those introduced nonterminals.

Consider some nonterminal A of the CNF grammar. If its rule has the form $A(y_1, \ldots, y_k) \to f(y_1, \ldots, y_k)$, then $\mathrm{sk}_A = f(y_1, \ldots, y_k)$ and if the arity of A is at most 1 then we add A and its rule to the set of constructed rules as well (if the rank of A is at least 2, then we do not add A and its rule). This rule has the desired form (HG2) or (HG3), there is no nonterminal on the right-hand side and at most 1 letter on the right-hand side. If the rule for A has the form $A(y_1, \ldots, y_k) \to B(y_1, \ldots, y_\ell, C(y_{\ell+1}, \ldots, y_{\ell'}), y_{\ell'+1}, \ldots, y_k)$, then we take sk_B and sk_C and replace in sk_B the parameter $y_{\ell+1}$ by sk_C, see Figures .15 and .16 for two different cases. Let us denote the resulting tree with sk'_A; it is transformed into a proper skeleton tree sk_A in the following. Let $y = y_{\ell+1}$.

Let us inspect what changes are needed, so that sk'_A satisfies (SK1)–(SK2). Suppose first that C is of arity at least 1, see Figure .15. It might be that the root node of sk_C and the node above the leaf y in sk_B are both of arity 1, without loss of generality assume that their labels are nonterminals C' and B' (the case of letters follows in the same way). We then introduce a new nonterminal A' of rank 1 and replace the subpattern $B'(C')$ in sk'_A with A' and add a rule $A' \to B'C'$. Note that it is in a form required by (HG2). We claim that the resulting tree satisfies (SK1) and (SK2) and hence can be taken for sk_A.

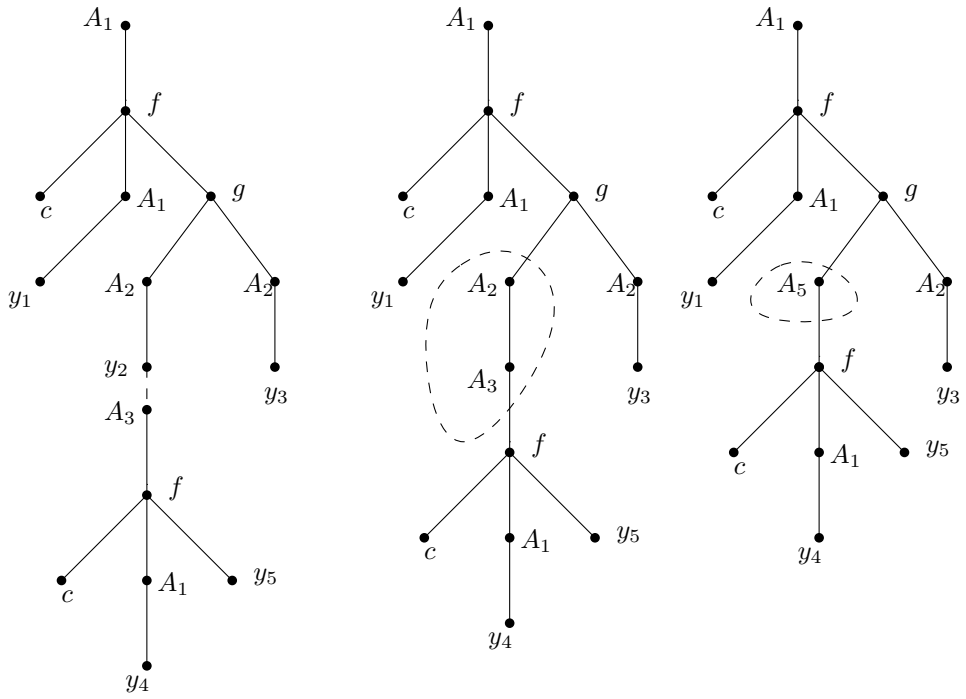

Figure .15: Calculating the skeleton when the lower skeleton is of rank at least 1. On the left-most figure we see the two skeletons. After substitution, there are two neighbouring rank-1 nonterminals: A_2 and A_3. They are replaced with the nonterminal A_5.

Since the node above B' and the node below C' are not of degree 1 (by induction assumption on (SK1)), (SK1) is satisfied. Concerning (SK2), take any node of arity at most 2 in sk'_A. Any node labelled with a letter in sk_C has the same subtrees in sk_C and in sk'_A, so (SK2) holds for them. For the nodes in sk_B the only problem can arise for nodes that had the replaced y in some of their subtrees. However, as y is replaced with sk_C, which has a parameter, the condition is preserved for them.

Suppose now that C is of arity 0, see Figure .16. Then the skeleton sk_C has no parameters, which implies that it is either a constant or a nonterminal of arity 0 (sk_C cannot use letters of arity larger than 1 by (SK2), and cannot use nonterminals and letters of arity 1, as their children need to be labelled with letters of arity at least 2). We only consider the former case (the same argument hold for the latter case). Let sk_C be the constant c. Firstly, the node above y (the parameter which is replaced by $\mathrm{sk}_C = c$) in sk_B can be a node of arity 1. Without loss of generality suppose that it is a nonterminal B' (the case of unary letter follows in the same way). We introduce a fresh nonterminal A' of arity 0, replace the subtree $B'(c)$ by A' and introduce the rule $A' \to B'c$. The rule is of the form required by (HG3). For uniformity, if the node above y is not of arity 1, introduce A' with the rule $A' \to c$ and replace c by A'. Condition (SK1) now holds.

Concerning (SK2), consider the parent node v of A'. Either it does not exist, in which case we are done (as $\mathrm{sk}_A = A'$) or it is labelled with a letter f of arity at least 2. All other nodes in sk'_A labelled with letters of arity at least 2 satisfy (SK2), as the subtree rooted at

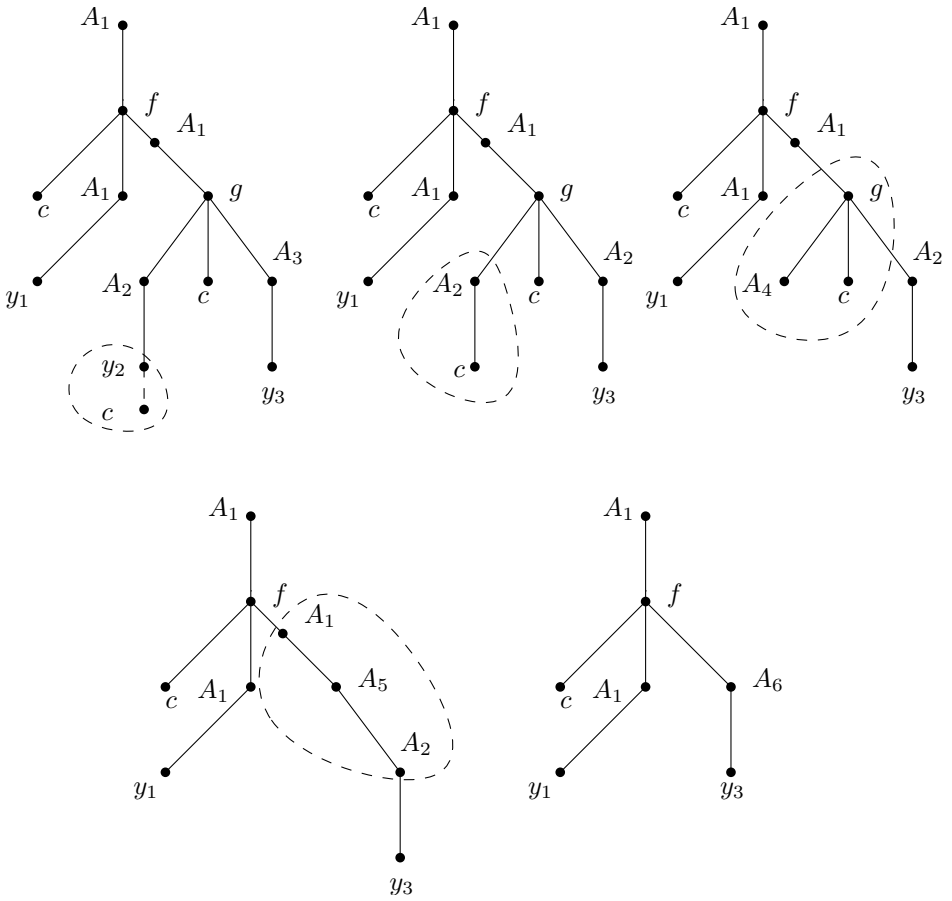

Figure .16: Calculating the skeleton, when the lower skeleton is of rank 0, going from the left-top to bottom-right. In the first picture there are two skeletons, in the second they are substituted into each other. In the third we replace A_2c by A_4 with the rule $A_4 \to A_2c$. In the fourth we replace $g(A_4, c, \cdot)$ by A_5 with the rule $A_5(y) \to g(A_4, c, y)$ and finally we replace $A_1A_5A_2$ by A_6 with the rule $A_6 \to A_1A_5A_2$, which can be split into two rules.

v still contains at least one parameter. Focusing on the f-labelled node v, if it still has at least two children with parameters in their subtrees, then we are done, as (SK2) is satisfied for v. If not, then exactly one of v's children is a subtree with a parameter. Without loss of generality let it be v's first child, all other children are constants or nonterminals of arity 0. So let the children of v (except for the first one) be labelled with $\gamma_2, \gamma_3, \ldots, \gamma_\ell$, where each γ_i is either a constant or a nonterminal of rank 0. Introduce a new nonterminal A' of rank 1 with the rule $A' \to f(y_1, \gamma_2, \gamma_3, \ldots, \gamma_\ell)$ and replace the subpattern $f(t_1, \gamma_2, \gamma_3, \ldots, \gamma_\ell)$ with $A'(t_1)$ (where, t_1 is the subtree rooted at the first child of v). Observe that the rule for A' is of size $\ell \leq r$ and is of the form (HG2). Lastly, now again (SK1) can be violated, because the parent node or the child (or both) of the A'-labelled node can be of degree 1. This can be fixed by replacing those 2 or 3 nodes of degree 1 by one nonterminal of rank 1. This requires adding at most 2 rules for nonterminals of arity 1 of the required form (HG2).

We constructed $\mathcal{O}(1)$ rules of the form (HG2) or (HG3) per nonterminal of the CNF grammar (there are $\mathcal{O}(g)$ of them), each of them has at most r occurrences of letters and nonterminals of arity 0 and at most 2 of nonterminals of arity 1. By a routine calculation it can be shown that $\mathrm{val}(\mathrm{sk}_A) = \mathrm{val}(A)$. If S is the start nonterminal of the CNF grammar, then sk_S has no parameters and hence is either a constant (this case is of course trivial) or a nonterminal of rank 0, which is the start nonterminal of our output grammar.

Concerning the efficiency of the construction, the proof follows in the same way as in [7, Theorem 10]: It is enough to observe that sk_A, where A has rank k, has at most $2r(k-1)+2$ nodes: By (SK1) nodes of arity 1 constitute at most half of all nodes. Secondly, as it has k parameters, it has at most $k-1$ nodes of arity larger than 1, so at most $(k-1)(r-1)+1$ leaves. Summing up yields the claim. □