APPROXIMATION OF SMALLEST LINEAR TREE GRAMMARS

ARTUR JEŻ AND MARKUS LOHREY

Abstract. A simple linear-time algorithm for constructing a linear context-free tree grammar of size $\mathcal{O}(r^2 g \log n)$ 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 approximation ratio. The analysis of the algorithm uses an extension of the recompression technique (used in the context of grammar-based string compression) from strings to trees.

1. Introduction

Grammar-based compression has emerged to an active field in string compression during the last couple of years. The principle 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 product a string of length $2^{\Omega(n)}$. Hence, an SLP can be seen indeed as a 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 produces s. Unfortunately, finding a minimal (with respect to size) SLP for a given input string is not achievable in polynomial time unless $\mathbf{P} = \mathbf{NP}$ [24]. Therefore, one should concentrate on grammar-based compressors, whose output SLP is not much larger than a size-minimal SLP for the input string. Formally, in [4] 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 [4] 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 [4] prove lower and upper bounds on the approximation ratios of well-known grammar-based string compressors (LZ 78, 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 [4, 10, 20, 21] 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 favor of grammar-based compression:

- The output of a grammar-based compressor (an SLP) 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 [14] for a survey).

Key words and phrases. Grammar-based compression; Tree compression; Tree-grammars.

Supported by NCN grant number 2011/01/D/ST6/07164, 2011–2014.

- There are grammar-based compressors which achieve very good compression ratios. For example RePair [13] performs very well in practice and was for instance used for the compression of web graphs [5].
- The idea of grammar-based string compression can be generalized to other data types as long as suitable grammar formalisms are known for them.

The last point is the most important one for this work. In [3], grammar-based compression was generalized from strings to trees.[1] 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 [7, 15, 18, 23, 22]. In [16], RePair was generalized from strings to trees, and the resulting algorithm TreeRePair achieved excellent results on real XML data trees. Other grammar-based tree compressors were developed in [17]. But none of these compressors has a good approximation ratio. For instance, in [16] 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 [3, 17]).

In this paper, we give the first example of a grammar-based tree compressor with an approximation ratio of $\mathcal{O}(\log n)$ assuming the maximal rank r of symbols is bounded; otherwise the approximation ratio becomes $\mathcal{O}(r^2 \log n)$. Our algorithm is based on the work [10] of the first author, where another grammar-based string compressor with an approximation ratio of $\mathcal{O}(\log n)$ is presented. The remarkable fact about this latter compressor is that in contrast to [4, 20, 21] it does not use the LZ77 factorization of a string (which makes the compressors from [4, 20, 21] 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 [8] and successfully applied for a variety of algorithmic problems for SLP-compressed strings [8, 9] and word equations [12, 11]. The basic idea is to compress a string using two operations:

- Block compressions, which replaces every maximal substring of the form a^ℓ for a letter a by a new symbol a_ℓ.
- Pair compression, which for a given partition $\Sigma_\ell \uplus \Sigma_r$ replaces 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 the single compression steps, one obtains 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 the $\mathbb{G}$. More precisely, one can compute from $\mathbb{G}$ a new grammar $\mathbb{G}'$, 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 [10], 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 generalize 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 node that are leafs 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

[1] 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.

input tree by basically reversing the sequence of compression operations. The recompression approach again yield an approximation ratio of $\mathcal{O}(\log n)$ for our compression algorithm, but the analysis is technically more subtle.

Related work on grammar-based tree compression. We already mentioned that grammar-based tree compressors were developed in [3, 16, 17], but none of these compressors has a good approximation ratio. Another grammar-based tree compressors was presented in [1]. It is based on the BISECTION algorithm for strings and has an approximation ratio of $\mathcal{O}(n^{5/6})$. But this algorithm used a different form of grammars (elementary ordered tree grammars) and it is not clear, whether the results from [1] can be extended to SLCF tree grammars, or whether the good algorithmic results for SLCF-compressed trees [7, 15, 18, 23, 22] can be extended to elementary ordered tree grammars. Let us finally mention the work from [2] where trees are compressed by so called top trees. These are another hierarchical representation of trees. Upper bounds on the size of the minimal top tree are derived in [2] and compared with the size of the minimal dag (directed acyclic graph). More precisley, it is shown in [2] that the size of the minimal top tree 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.

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. It sorts m many k-ary numbers of length ℓ in time $\mathcal{O}(\ell m + \ell k)$, see e.g. [6, Section 8.3]. We need a still standard but slightly more powerful version 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, see the appendix. 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 and context trees.** Let us fix for every $i \geq 0$ a countably infinite set $\mathbb{F}_i$ of letters of rank i 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 shall 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 the left to the right) and each node (except for the root) has a pointer to its parent node.

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 will be patterns, and we will 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 replace these subtrees by pairwise different parameters, then we obtain a pattern $p(y_1, \dots, y_n)$ and we say that (i) the subpattern v is an *occurrence* of the pattern $p(y_1, \dots, y_n)$ in t and (ii) $p(y_1, \dots, y_n)$ 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. 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 $a(b(c))$ and two occurrences of the pattern $a(b(y))$.

A *chain pattern* is a pattern of the form $a_1(a_2(\dots(a_k(y))\dots))$ with $a_1, a_2, \dots, a_k \in \mathbb{F}_1$. 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, will be 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 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 $b(a(a(a(y))))$.

We write $a_1 a_2 \cdots a_k$ for the chain pattern $a_1(a_2(\dots(a_k(y))\dots))$ 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 consideration, 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* or short *linear CF tree grammar*[2] is a triple $\mathbb{G} = (N, F, P, S)$ 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 $t \in \mathcal{T}(F \cup N, \{y_1, \dots, y_{\mathsf{rank}(A)}\})$ 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* of rank 0.

To stress the dependency of A on its parameters we sometimes write $A(y_1, \dots, y_{\mathrm{rank}(A)}) \to t$ 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 $\mathcal{G}$ is *k-bounded* (for a natural number k) if $\mathsf{rank}(A) \leq k$ for every $A \in N$. Moreover, $\mathcal{G}$ 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 $(A(y_1, \dots, y_\ell) \to t) \in P$ such that s' is obtained from s by replacing some subtree $A(t_1, \dots, t_\ell)$ of s by t with each y_i replaced by t_i. Intuitively, we replace an A-labelled node by the pattern $t(y_1 \dots, y_{\mathrm{rank}(A)})$ and thereby identify the j-th child of A with the unique y_j-labelled node of the pattern. 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 appears in t then $B < A$, where $<$ is a linear order on N, and
- S is the maximal nonterminal with respect to $<$.

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

By the first two conditions, every $A \in N$ derives exactly one tree from $\mathcal{T}(F, \{y_1, \dots, y_{\mathsf{rank}(A)}\})$; we denote this tree by $\mathrm{val}(A)$ (like *value*). 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 $y_1, \dots, y_{\mathrm{rank}(A)}$ appear in t in the order $y_1, y_2, \dots, y_{\mathrm{rank}(A)}$ from left to right. This can be ensured by a simple bottom-up rearranging procedure.

2.3. **Grammar size.** There is a subtle point, when defining the *size* $|\mathbb{G}|$ of the SLCF grammar $\mathbb{G}$: One possible definition could be $|\mathbb{G}| = \sum_{(A \to t) \in P} |t|$, i.e., the sum of all sizes of all right-hand sides. However, consider for instance the rule $A(y_1, \dots, y_\ell) \to f(y_1, \dots, y_{i-1}, a, y_i, \dots, y_\ell)$. 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 nonterminals of rank at most $r - 1$, 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.

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 a tree T and applies local compression operations to the tree that shrink the size of the tree. With T we will always denote the current tree stored by TtoG, whereas n denotes the size of the initial input tree. The algorithm TtoG we shall add fresh letters to the tree. With F we will always denote the set of letters appearing in the current tree T. The ranks of the fresh letters do not exceed the maximal rank of the original letters. To be more precise, if we add a letter a to F_i, then $F_{\geq i}$ was non-empty before this addition. By r we denote the maximal rank of the letters appearing in the input tree. By the above remark, TtoG will never introduce letters of rank larger than 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 [10]. To be more precise, (a) and (b) affect only chains, return chains as well, and when a chain is treated as a string the result of (a) and (b), respectively, corresponds to the result of the corresponding operation on strings. On the other hand, the last operations (c) is new and designed specifically to deal with trees. Let us inspect those 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)).

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

Note that each of these operations shrinks 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 $a_1, \dots, a_\ell$.

Despite its rather cumbersome definition, the idea behind leaf compression is easy: For a fixed appearance of f in a tree we ‘absorb’ all leaf-children of f that are constants (and do the same for all other appearances 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 $(f, i_1, a_1 \dots, i_\ell, a_\ell)$-leaf compression the production is

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

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

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 have to 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' above) 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 do not mix the notation, we shall 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 shall say that the sum of the sizes of the right-hand sides of the generated subgrammar is the representation cost for a_ℓ (for this 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 $\mathsf{TreeChainComp}(F_1, t)$: Compression of chains of letters from F_1 in a tree t

```
for $a \in F_1$ do                                                        ▷ chain compression
    for $\ell \leftarrow 1 \mathinner{..} |t|$ do
        replace every $a$-maximal chain of size $\ell$ by a fresh letter $a_\ell$     ▷ $a_\ell \notin F_1$
```

We shall refer to the procedure TreeChainComp simply as *chain compression.* The running time of an appropriate implementation is considered in the next section and the size of the generated grammar 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 $\mathsf{TreeLeafComp}(F_{\geq 1}, F_0, t)$: leaf compression for parent nodes in $F_{\geq 1}$, and leaf-children in F_0 for a tree t

```
for $f \in F_{\geq 1}, 0 < i_1 < i_2 < \cdots < i_\ell \leq \mathrm{rank}(f) =: m, (a_1, a_2, \ldots, a_\ell) \in F_0^\ell$ do
    replace each subtree $f(t_1, \ldots, t_m)$ s.t. $t_{i_j} = a_j$ for $1 \leq j \leq \ell$ and $|t_i| > 1$ for
    $i \notin \{i_1, \ldots, i_\ell\}$ by $f'(t_1, \ldots, t_{i_1-1}, t_{i_1+1}, \ldots, t_{i_\ell-1}, t_{i_\ell+1}, \ldots, t_m)$     ▷ $f' \notin F_{\geq 1} \cup F_0$
```

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 [10], 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 for each unary letter we can tell 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 new letters are outside $F_1^{\mathrm{up}} \cup F_1^{\mathrm{down}}$.

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

```
for $a \in F_1^{\mathrm{up}}$ and $b \in F_1^{\mathrm{down}}$ do
    replace each occurrence of $ab$ with a fresh unary letter $c$          ▷ $c \notin F_1^{\mathrm{up}} \cup F_1^{\mathrm{down}}$
```

The procedure TreeUnaryComp is called $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression in the following.

3. Algorithm

In a single phase of the algorithm TtoG, chain compression, $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression and leaf compression are executed in this order (for an appropriate choice of the partition $F_1^{\mathrm{up}}, F_1^{\mathrm{down}}$). 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 half, as it effectively reduces all constant nodes, i.e. leaves of the tree, and more than half of 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 of the first author on string compression [10], 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 $F_1^{\mathrm{up}}, F_1^{\mathrm{down}}$ of the letters appearing 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 occuring 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 $T$
    $T \leftarrow$ 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 5   ▷ time $\mathcal{O}(|T|)$
    $T \leftarrow$ 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 $T$
    $T \leftarrow$ 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 (polynomial implementation is straightforward), while in Section 4 we analyze the approximation ratio of TtoG.

Before we make any analysis, we note that at the beginning of each phase we can make a linear-time preprocessing that guarantees that the letters in T form an interval of numbers (which makes them more suitable for sorting using RadixSort).

Lemma 2 (cf. [10, Lemma 1]). *At the beginning of each phase of* TtoG*, we can rename in time $\mathcal{O}(|T|)$ the letters used in T so that they form an interval of numbers.*

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 $n' \leq n$.

Suppose that at the beginning of the phase the letters form 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 phase the letters appearing in T are within an interval $[m, \ldots, m+k']$ for some $k' > k$. It is now left to re-number the letters from $[m, \ldots, m+k']$, so that the ones appearing 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 $flag[a] = 0$, we set $flag[a] := 1$; $new[a] := next$, and $next := next + 1$. Moreover, we replace the label of the current node (which is a) by $new[a]$. When we spot a symbol $a \in [m, \ldots, m+k']$ with $flag[a] = 1$, then we replace the label of the current node (which is a) by $new[a]$. Clearly the running time is $\mathcal{O}(|T|)$ and after the algorithm the symbols form a subinterval of $[m+k'+1, \ldots, m+2k'+1]$. □

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 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.

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 will have a representation cost of $\mathcal{O}(k + \sum_{i=1}^{k} \log(\ell_i - \ell_{i-1}))$ (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 $\ell_1, \ell_2, \dots, \ell_k$.

Lemma 3 (cf. [10, Lemma 2])**.** *Given a list $\ell_1 < \ell_2 < \cdots < \ell_k$ we can represent the letters $a_{\ell_1}, a_{\ell_2}, \dots, a_{\ell_k}$ that replace the chain patterns $a^{\ell_1}, a^{\ell_2}, \dots, a^{\ell_k}$ 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 [10], still we supply it for completeness.

Firstly observe that without loss of generality we may assume that the list $\ell_1, \ell_2, \dots, \ell_k$ is given in a sorted way, as it can be easily obtained form the sorted list of appearances 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 shall 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 a new unary letters $a_{\ell_i - \ell_{i-1}}$ for each $1 \leq i \leq k$, which will 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 $a_1, a_2, \dots, a_{2^{\lfloor \log \ell \rfloor}}$. 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 shall 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 4 (cf. [10, 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. 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. 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. 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 (an almost) 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 appearing in T into subsets F_1^{up} and F_1^{down}. Then we replace all 2-chains, whose first (resp., second) node is labelled with a letter from F_1^{up} (resp., 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 5. *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 $F_1 = F_1^{up} \uplus F_1^{down}$ such that the number of occurrences of chain patterns from $F_1^{up} F_1^{down}$ in T is at least $(n_1 - 3c + 2)/4$, 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 $ab \in F_1^{up} F_1^{down}$ appearing 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 was shown in [10] (for completeness, we provide a proof below):

Claim 1 ([10, Lemma 4]). *For a word w that does not contain a factor aa for some symbol a and whose alphabet can by identified with an interval of numbers of size m, one can find in time $\mathcal{O}(|w| + m)$ a partition of the letters appearing in w into sets F_1^{up} and F_1^{down} such that the number of occurrences of chain patterns from $F_1^{up} F_1^{down}$ in w is at least $(|w| - 1)/4$. In the same running time we can provide for each $ab \in F_1^{up} F_1^{down}$ appearing in w a lists of pointers to all occurrences of ab in w.*

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 $w_1, w_2, \ldots, w_c$. Take a new letter # (which is identified with a number so that the alphabet of w is an interval of numbers) and consider the string $w = w_1\#w_2\# \ldots \#w_c$. Its length is $n_1 + c - 1$ which is at most $2|T|$. Observe that no two consecutive letters in w are equal: No two #'s are next to each other and by the assumption from the lemma no two consecutive letters in a string w_i are identical. Moreover, the alphabet of w is an interval of size $\mathcal{O}(|T|)$. By Claim 1 one can compute in time $\mathcal{O}(|w|) \leq \mathcal{O}(|T|)$ a partition $F_1 \cup \{\#\} = F_1^{\mathrm{up}} \uplus F_1^{\mathrm{down}}$ such that $\frac{|w|-1}{4}$ many 2-chains from w are covered by this partition. Consider the partition of F_1 obtained from $F_1 \cup \{\#\} = F_1^{\mathrm{up}} \uplus F_1^{\mathrm{down}}$ by removing #. It covers at least

$$\frac{|w| - 1}{4} - (c - 1) = \frac{n_1 + (c - 1) - 1}{4} - (c - 1) = \frac{n_1 - 3c + 2}{4}$$

many 2-chains in $w_1, w_2, \ldots, w_c$ (and hence in T), because we loose at most $c - 1$ many 2-chains from w (namely those 2-chains that contain the letter #). Moreover, by Claim 1 one can also compute in time $\mathcal{O}(|w|) \leq \mathcal{O}(|T|)$ for every $ab \in F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ appearing in w a lists of pointers to all appearances of ab in w. It is straightforward to compute from this list a lists of pointers to all appearances of ab in T.

Let us now provide a proof of Claim 1:

Proof. The existence of a partition covering at least one fourth of the appearances 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. Then $a \in F_1^{\mathrm{up}}$ and $b \in F_1^{\mathrm{down}}$ with probability 1/4. There are $|w| - 1$ such 2-chains in w, so the expected number of occurrences of patterns from $F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ in w is $(|w| - 1)/4$. Hence, there exists a partition that covers at least $(|w| - 1)/4$ many occurrences of 2-chains. Observe, that the expected number of occurrences of patterns from $F_1^{\mathrm{up}} F_1^{\mathrm{down}} \cup F_1^{\mathrm{down}} F_1^{\mathrm{up}}$ is $(|w| - 1)/2$.

The deterministic construction of a partition covering at least $(|w| - 1)/4$ appearances follows by a simple derandomisation, using an expected value approach. It is easier to first find a partition $F_1 = F_1' \uplus F_1''$ such that at least $(|w| - 1)/2$ many occurrences of 2-chains in w are covered by $F_1^{\mathrm{up}} F_1^{\mathrm{down}} \cup F_1^{\mathrm{down}} F_1^{\mathrm{up}}$. We then choose $F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ or $F_1^{\mathrm{down}} F_1^{\mathrm{up}}$, depending on which of them covers more appearances.

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 appearances of patterns from $aF_1^{\mathrm{up}} \cup F_1^{\mathrm{up}} a$ are not going to be covered, while appearances of patterns from $aF_1^{\mathrm{down}} \cup F_1^{\mathrm{down}} a$ 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 will cover at least half of all appearances of 2-chains that contain the letter a. Finally, as each appearance of a pattern ab from w 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 are covered.

In order to make the selection efficient, the algorithm Greedy2Chains below keeps for every letter a counters $\mathit{count}'[a]$ and $\mathit{count}''[a]$, storing the number of appearances of patterns from $aF_1^{\mathrm{up}} \cup F_1^{\mathrm{up}} a$ and $aF_1^{\mathrm{down}} \cup F_1^{\mathrm{down}} a$, respectively, in w. 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 $w$
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:     $count'[a] \leftarrow count''[a] \leftarrow 0$                  ▷ initialisation
5: for $a \in F_1$ do
6:     if $count''[a] \geq count'[a]$ then        ▷ choose the one that guarantees larger cover
7:         $F_1^{\text{up}} \leftarrow F_1^{\text{up}} \cup \{a\}$
8:         for each occurrence of $ab$ or $ba$ in $w$ do
9:             $count'[b] \leftarrow count'[b] + 1$
10:    else
11:        $F_1^{\text{down}} \leftarrow F_1^{\text{down}} \cup \{a\}$
12:        for each occurrence of $ab$ or $ba$ in $w$ do
13:            $count''[b] \leftarrow count''[b] + 1$
14: if # occurrences of patterns from $F_1^{\text{down}}F_1^{\text{up}}$ in $w$ > # occurrences of patterns from $F_1^{\text{up}}F_1^{\text{down}}$
    in $w$ then
15:     switch $F_1^{\text{down}}$ and $F_1^{\text{up}}$
16: 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 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

It is left to give an efficient variant of Greedy2Chains. The non-obvious operations are the updating of $count'[b]$ (resp., $count''[b]$) in line 9 (resp., line 13) and the choice of the actual partition in line 15. All other operation clearly take at most time $\mathcal{O}(|w|)$. The latter is simple: since we organize F_1^{up} and F_1^{down} as bit vectors, we can read w from left to right 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 $\mathcal{O}(|w|)$ (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.

To implement $count'$ and $count''$, for each letter a in w we store a *right list* $right(a) = \{b \mid ab \text{ appears in } w\}$, represented as a list. Furthermore, the element b on the right list points to a list of all appearances of the pattern ab in w. There is a similar *left list* $left(a) = \{b \mid ba \text{ appears in } w\}$. We comment on how to create the left lists and right lists in linear time later.

Given *right* and *left*, performing the updates in line 9 and line 13 is easy: Suppose that we update $count'$. We go through $right(a)$ (resp., $left(a)$) and increment $count'[b]$ for every occurrence of ab (resp., ba). Note that in this way each of the lists $right(a)$ ($left(a)$) is read once during Greedy2Chains. Therefore, all updates of $count'$ and $count''$ only need time $\mathcal{O}(|w|)$.

It remains to show how to initially create $right(a)$ ($left(a)$ is created similarly). We read w. When reading a pattern ab we create a record (a, b, p), where p is a pointer to this appearance. We then sort these record lexicographically using RadixSort, ignoring the last component. There are only $\mathcal{O}(|w|)$ records and the alphabet is an interval of size m, so RadixSortneeds time $\mathcal{O}(|w|+m)$. Now, for a fixed letter a, the consecutive tuples with the first component a can be turned into $right(a)$: For $b \in right(a)$ we want to store a list I of pointers to appearances of ab, and on a sorted list of tuples the entries (a, b, p) for $p \in I$ form consecutive elements. 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 $ab \in F_1^{\text{up}}F_1^{\text{down}}$, the lists of pointers to appearances of ab in w, it is enough to read *right* and filter the patterns ab such that $a \in F_1^{\text{up}}$ and $b \in F_1^{\text{down}}$; the filtering can be done in $\mathcal{O}(1)$ per appearance as F_1^{up} and F_1^{down} are represented as bitvectors. The total needed time is $\mathcal{O}(|w|)$. □

This concludes the proof of Lemma 5. □

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.

3.3. **Leaf compression.** Leaf compression is done in a similar way as 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 $2|T|+1$, furthermore each position index is at most $r \leq |T|$ and by Lemma 2 also each letter is a number from a range 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 pointer component 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 $\mathcal{O}(|T|)$.

3.4. **Size and running time.** It remains to estimate the total running time of algorithm TtoG, summed over all phases. As each subprocedure in a phase has running time $\mathcal{O}(|T|)$, 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 6. *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 similarly for $n_{\geq 2}'$, $n_{\geq 2}''$, and $n_{\geq 2}'''$. Clearly

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

We show that

$$n_0''' + n_1''' + n_{\geq 2}''' \leq \frac{7}{8}(n_0 + n_1 + n_{\geq 2}) \quad ,$$

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

$$c \leq n_{\geq 2} + \frac{n_0}{2} + \frac{1}{2} \quad . \tag{5}$$

Indeed, consider a maximal chain. Then the node above has a label from $F_{\geq 2}$ (unless the maximal chain includes the root) while the node below it has a label from $F_{\geq 2} \cup F_0$. Summing this up by all chains we get $2c \leq 2n_{\geq 2} + n_0 + 1$ (the '+1' is for the possibility that the root has a unary label), which yields (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 5 we choose a partition such that at least $\frac{n_1' - 3c + 2}{4}$ many 2-chains are compressed (note that the assumption of Lemma 5 that no parent node and its child are labelled with the same unary letter is satisfied by Lemma 4), so the size of the tree is reduced by at least $\frac{n_1' - 3c + 2}{8}$. Hence, the size of the tree after unary pair compression is at most

$$\begin{aligned} n_0'' + n_1'' + n_{\geq 2}'' &\leq n_0' + n_1' + n_{\geq 2}' - \frac{n_1' - 3c + 2}{8} \\ &= n_0 + \frac{7n_1'}{8} + n_{\geq 2} + \frac{3c}{8} - \frac{1}{4} \\ &\leq n_0 + \frac{7n_1}{8} + n_{\geq 2} + \frac{3c}{8} - \frac{1}{4} \ . \end{aligned} \tag{6}$$

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

$$\begin{aligned} n_0''' + n_1''' + n_{\geq 2}''' &= n_0'' + n_1'' + n_{\geq 2}'' - n_0 && \text{leaf compression} \\ &\leq n_0 + \frac{7n_1}{8} + n_{\geq 2} + \frac{3c}{8} - n_0 - \frac{1}{4} && \text{from (6)} \\ &= \frac{7n_1}{8} + n_{\geq 2} + \frac{3c}{8} - \frac{1}{4} && \text{simplification} \\ &\leq \frac{7n_1}{8} + n_{\geq 2} + \frac{3n_{\geq 2}}{8} + \frac{3n_0}{16} + \frac{3}{16} - \frac{1}{4} && \text{from (5)} \\ &< \frac{3n_0}{16} + \frac{7n_1}{8} + \frac{11n_{\geq 2}}{8} && \text{simplification} \\ &= \frac{7}{8}\Big(n_0 + n_1 + n_{\geq 2}\Big) + \Big(-\frac{11n_0}{16} + \frac{4n_{\geq 2}}{8}\Big) && \text{simplification} \\ &< \frac{7}{8}\Big(n_0 + n_1 + n_{\geq 2}\Big), && \text{from (4)} \end{aligned}$$

as claimed. □

Theorem 1. TtoG *runs in linear time.*

Proof. Each phase clearly takes $\mathcal{O}(|T|)$ time and by Lemma 6 the $|T|$ 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 grammar, we start with a smallest 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 a grammar $\mathbb{G}$ (also generating T) of a special form and of size $\mathcal{O}(r|\mathbb{G}_{\text{opt}}|)$, where r is the maximal rank of symbols in F (the set of letters appearing in $\mathbb{G}_{\text{opt}}$). The transformation is based on known results on normal forms for SLCF grammars [18], see Section 4.1.

During the run of TtoG we modify $\mathbb{G}$, preserving its special form, so that it generates T (i.e. the current tree kept by TtoG) after each of the compression steps of TtoG. Then the cost of representing the letters introduced by TtoG is paid by various credits assigned to $\mathbb{G}$. 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.

4.1. Normal form.

It is known that every SLCF grammar $\mathbb{G}$ can be transformed into a monadic SLCF grammar $\mathbb{G}'$ (thus, every nonterminal of $\mathbb{G}'$ has rank 0 or 1) generating the same tree. Moreover, $\mathbb{G}'$ is only $\mathcal{O}(r)$ times larger than $\mathbb{G}$, where r is the maximal rank of letters from $\mathbb{F}$ used in $\mathbb{G}$:

Lemma 7 ([18, Theorem 10])**.** *From a given SLCF grammar $\mathbb{G}$ one can construct in polynomial time a monadic SLCF grammar $\mathbb{G}'$ of size $\mathcal{O}(r \cdot |\mathbb{G}|)$ such that* $\text{val}(\mathbb{G}) = \text{val}(\mathbb{G}')$*, where r is the maximal rank of a letter from $\mathbb{F}$ used in $\mathbb{G}$.*

The assumption that a grammar is monadic is still too weak for us. We need a particular, slightly stricter form of a monadic grammar. This special form is introduced in this subsection.

We say that a pattern $t(y)$ is a *handle* if it is of the form

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

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. Note that $a(y)$ for a unary letter a is a handle. Since handles have one parameter only, for handles $h_1, h_2, \dots, h_\ell$ we write $h_1 h_2 \cdots h_\ell$ for the tree $h_1(h_2(\dots(h_\ell(y))))$ and treat it as a string, similarly to chains patterns.

We say that an SLCF grammar $\mathbb{G}$ 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.

(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, $B \in N_1$, $C \in N_0$, and $j, k < i$. 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 $A \to a(C)$ for the handle $a(y) = f(B, y)$ or as $A \to b(B)$ for the handle $b = f(y, C)$. 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.

For each monadic SLCF grammar we can find an equivalent handle grammar of similar size:

Lemma 8. *Given a monadic SLCF grammar $\mathbb{G}$ one can in polynomial time construct a handle $\mathbb{G}'$ of size $\mathcal{O}(|\mathbb{G}|)$ that derives the same tree.*

Proof. For the purpose of the proof it is convenient to define a *pre-handle* as a tree of the form $f(t_1, t_2, \dots, t_{i-1}, y, t_{i+1}, t_{i+1}, \dots, t_\ell)$, where each t_j is a tree from $\mathcal{T}(\mathbb{F} \cup \mathbb{N}_0 \cup \mathbb{N}_1)$, i.e. it has no parameters. We write compositions of pre-handles and nonterminals from N_1 in string notation, i.e. using concatenation, in the same way as for handles and chains. Each pre-handle can be turned into a handle. It is enough to replace each t_i that is not a nonterminal from N_0 by a new nonterminal A_i with the rule $A_i \to t_i$.

We follow a natural strategy: We rebuild $\mathbb{G}$ in the way that all original nonterminals derive exactly the same tree. Consider any rule $A \to s$ for $A \in N_1$, where $s \neq y$ without loss of generality (we can remove such nonterminals without increasing the size of the grammar). Then there is a unique appearance of the parameter y in s. Consider the path from the root of s to y. Since the grammar is monadic, we can represent s as a concatenation (along this path) of pre-handles and nonterminals from N_1. We turn each such pre-handle into a handle. This doubles the size of the grammar in the worst case. Then, for each $A \in N_1$ the rule has the form $A \to w_1 A_1 w_2 A_2 w_3 \dots w_k A_k w_{k+1}$, where each w_j is a (perhaps empty) sequence of handles. We introduce new nonterminals $B_2, \dots, B_k$ of rank 1 with rules $B_j \to w_j A_j B_{j+1}$ for $j < k$ and $B_k \to w_k A_k$. Finally, replace the rule for A by $A \to w_1 A_{i_1} B_2 w_{k+1}$. Clearly this again at most doubles the size of the grammar. In this way all rules for nonterminals from N_1 are transformed into the desired form.

It remains to deal with rules of the form $A \to s$ for $A \in N_0$. If s is a constant symbol, then nothing is to do. If $s \in N_0$, then we can eliminate A. If $s = B(s')$ for some $B \in N_1$ and tree s', then we introduce a new nonterminal B' of rank 0 with the rule $B' \to s'$ and replace the rule for A by $A \to BB'$ (which is allowed in a handle grammar). We then continue with the smaller tree s'. Finally, if $s = f(s_1, \ldots, s_n)$ then we introduce new nonterminals $A_1, \ldots, A_n$ of rank 0 with the rule $A_i \to s_i$ and replace the rule for A by $A \to f(A_1, \ldots, A_n)$ (which is allowed in a handle grammar). We then continue with the smaller trees $s_1, \ldots, s_n$.

Iteration of above two procedures leads to a handle grammar. It is easy to check that they at most double the size of the grammar. □

Corollary 1. *Let $\mathbb{G}$ be an SLCF grammar. Then there exists a handle grammar $\mathbb{G}'$ such that* $\mathrm{val}(\mathbb{G}) = \mathrm{val}(\mathbb{G}')$ *and* $|\mathbb{G}'| = \mathcal{O}(r|\mathbb{G}|)$, *where r is the maximal rank of the letters used in $\mathbb{G}$.*

When considering handle grammars it is useful to have some intuition about the trees they derive. Let a *context* be a pattern $t(y) \in \mathcal{T}(F, \{y\})$ 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 the string notation. Also, when we attach a tree from $\mathcal{T}(F)$ to a context, we obtain another tree from $\mathcal{T}(F)$. 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. 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 handle grammar $\mathbb{G}$ generating T. In the following, by g we always denote the size of this initial minimal handle grammar. For analyzing the size of the grammar produced by TtoG applied to T, we use the accounting method, see e.g. [6, Section 17.2]. With each appearance of a letter from $\mathbb{F}$ in $\mathbb{G}$'s rules we associate 2 units of *credit.* During the run of TtoG we want to appropriately modify $\mathbb{G}$, so that $\mathrm{val}(\mathbb{G}) = T$ (where T always denote the current tree in TtoG). In other words, we want to perform the compression steps of TtoG also on $\mathbb{G}$. We always maintain the invariant that every appearance 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 will not suffice. Here we need some extra amount that will be estimate separately later on in Section 4.6. At the end, we can bound the size of the grammar produced by TtoG as the sum of the initial credit assigned to $\mathbb{G}$ (which is at most $2g$) 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 the earlier paper of the first author [10] 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 denotes its nonterminals with $A_1, \ldots, A_m$. We assume that $i < j$ if A_i appears 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 analyzing 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 appearances of ab in $\mathrm{val}(\mathbb{G})$ will be 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 intuitions are made precise in Section 4.3. To deal with this problem, we modify the grammar, so that it disappears. Similar problems occur also when we want to replace an a-maximal chain or perform leaf compression. Solutions to those problems are similar and are given in Section 4.4 and Section 4.5, respectively.

To ensure that $\mathbb{G}$ is handle and to estimate the amount of issued credit, we show that the grammar preserves the following invariants, where n_0 (resp. n_1) is the initial number of nonterminals from N_0 (resp., N_1) in $\mathbb{G}$ and g is the initial size of $\mathbb{G}$.

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

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

(GR3) There are at most g appearances of nonterminals from N_0 in $\mathbb{G}$ and at most $n_0 + 2n_1$ appearances of nonterminals from N_1.

(GR4) The number of appearances of nonterminals from $\widetilde{N_0}$ in $\mathbb{G}$ is at most $(n_0 + 2n_1)(r - 1)$.

(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.

It is easy to show that (GR1)–(GR5) hold for the initial handle grammar $\mathbb{G}$: We assign each nonterminal of rank 0 (respectively 1) to N_0 (respectively N_1); thus $\widetilde{N_0}$ is empty. The only nontrivial condition is that the number of appearances of nonterminals from N_1 is at most $n_0 + 2n_1$. However, observe that in a rule for $A_i \in N_0$ there is at most one appearance of a nonterminal from N_1, namely the first nonterminal in this rule (all other nonterminals are parts of handles and so they are from N_0). Similarly in a rule for $A_i \in N_1$ there are at most two appearances of nonterminals from N_1, namely the first and the second nonterminal in this rule.

4.3. **$(F_1^{\mathbf{up}}, F_1^{\mathbf{down}})$-compression.** We begin with some necessary definition 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. A chain pattern ab has a *crossing appearance* 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 $\mathrm{val}(A_j)$ is b

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

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

A chain pattern ab is *crossing* if it has a crossing appearance 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 $\mathsf{TreeUnaryComp}$ (Algorithm 3) to each right-hand side of $\mathbb{G}$. We use the shortname $\mathsf{TreeUnaryComp}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathbb{G})$ for the resulting output grammar.

In order to distinguish between the nonterminals, grammar, etc. before and after the application of $\mathsf{TreeUnaryComp}$ (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 9. *Let $\mathbb{G}$ satisfy (GR1)–(GR5) and $\mathbb{G}' = \mathsf{TreeUnaryComp}(F_1^{up}, F_1^{down}, \mathbb{G})$. Then $\mathbb{G}'$ satisfies (GR1)–(GR5) as well. If there is no crossing chain pattern from $F_1^{up} F_1^{down}$ in $\mathbb{G}$, then $\mathrm{val}(\mathbb{G}') = \mathsf{TreeUnaryComp}(F_1^{up}, F_1^{down}, \mathrm{val}(\mathbb{G}))$. The credit for new letters in $\mathbb{G}'$ and the cost of representing these new letters is 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 is replaced by $\mathsf{TreeUnaryComp}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathbb{G})$. If it 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 appearance of ab in the rule for A_i'. However note that if a is the first (or is b the last) letter of $\mathrm{val}(A_j)$, then it was also the first (last, respectively) letter of $\mathrm{val}(A_j)$ in the input instance, as we neither introduce any new letters from F_1^{up} nor F_1^{down}. So 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 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 appearance of ab, 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 appearances of the nonterminals is unaffected, so also (GR2)–(GR4) are preserved. □

By Lemma 9 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 appearance. To deal with crossing appearances we change the grammar. In our example, we replace A_1 with a, leaving only $A_2 \to ab(c)$, which does not contain a crossing appearance 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). It is shown in the lemma below that popping up and popping down removes all crossing appearances of ab. Note that the operations of popping up and popping down can be performed for many 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).

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

```
for $i \leftarrow 1 .. m-1$ do
    if the first symbol of $\alpha_i$ is $b \in F_1^{\mathrm{down}}$ then          ▷ popping up $b$
        if $\alpha_i = b$ then
            replace $A_i$ in all right-hand sides of $\mathbb{G}$ by $b$
        else
            remove this leading $b$ from $\alpha_i$
            replace $A_i$ in all right-hand sides of $\mathbb{G}$ by $bA_i$
    if $A_i \in N_1$ and the last symbol of $\alpha_i$ is $a \in F_1^{\mathrm{up}}$ then   ▷ popping down $a$
        if $\alpha_i = a$ then
            replace $A_i$ in all right-hand sides of $\mathbb{G}$ by $a$
        else
            remove this final $a$ from $\alpha_i$
            replace $A_i$ in all right-hand sides of $\mathbb{G}$ by $A_i a$
```

Lemma 10. *Assume that $F_1^{up} \cap F_1^{down} = \emptyset$. All chain patterns from $F_1^{up} F_1^{down}$ are non-crossing in $\mathbb{G}' = \mathsf{Pop}(F_1^{up}, F_1^{down}, \mathbb{G})$. Furthermore, $\mathrm{val}(A'_m) = \mathrm{val}(A_m)$. At most $2g + 2(n_0 + 2n_1)(r+1)$ units of credit are issued in the computation of $\mathbb{G}'$. If $\mathbb{G}$ satisfies (GR1)–(GR5) then so does $\mathbb{G}'$.*

Proof. Observe first that whenever we pop-up b from some A_i, 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. 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, a contradiction.

Suppose 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) in case aA'_i is a subpattern in a right-hand side of $\mathbb{G}'$ and the first letter of $\mathrm{val}(A'_i)$ is b; other cases are dealt in a similar fashion. Note that as $a \notin F_1^{\mathrm{down}}$ is labelling the parent node of an occurence 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 it should have (by our earlier claim), which is a contradiction.

Note that Pop introduces at most 2 new letters for each occurrence of a nonterminal from N_1 (one popped up and one popped down) and at most 1 new letter for each occurrence of a nonterminal from $N_0 \cup \widetilde{N}_0$ (as nonterminals of rank 0 cannot pop down a letter). Thus, by (GR3) and (GR4) at most $2g + 2(n_0 + 2n_1)(r + 1)$ units of credit are issued:

- Every nonterminal from $N_0 \cup \widetilde{N}_0$ pops at most 1 letter and there are at most $g + (n_0 + 2n_1)(r - 1)$ many occurrences of them. Hence $g + (n_0 + 2n_1)(r - 1)$ many occurrences of new letters are introduced. Since each occurrence of a new letter gets two units of credit, we have to issue in total $2g + 2(n_0 + 2n_1)(r - 1)$ units of credit.
- Every nonterminal from N_1 pops at most 2 letter and there are at most $(n_0 + 2n_1)$ occurrences of them. Hence $2(n_0 + 2n_1)$ many occurrences of new letters are introduced. Since each occurrence of a new letter gets two units of credit, we have to issue in total $4(n_0 + 2n_1)$ units of credit.

Summing up both contributions gives $2g+2(n_0+2n_1)(r-1)+4(n_0+2n_1) = 2g+2(n_0+2n_1)(r+1)$ units of credit.

Concerning the preservation of the invariants, note that we only remove and add appearances of unary letters, which cannot affect the set of nonterminals or their number of appearances. Therefore, (GR2)–(GR4) are preserved. Moreover, also the shape of the productions cannot be spoiled. So (GR1) and (GR5) are preserved as well, as claimed. □

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 11. *Let $\mathbb{G}$ satisfy (GR1)–(GR5) and let*

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

Then $\mathrm{val}(\mathbb{G}') = \mathsf{TreeUnaryComp}(F_1^{up}, F_1^{down}, \mathrm{val}(\mathbb{G}))$ *and $\mathbb{G}'$ satisfies (GR1)–(GR5) as well. Moreover, at most $2g + 2(n_0 + 2n_1)(r + 1)$ units of credit are issued in the computation of $\mathbb{G}'$. The issued credit and the credit released by* $\mathsf{TreeUnaryComp}$ *cover the representation cost of fresh letters as well as their credit.*

Proof. By Lemma 10, every chain pattern from $F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ is non-crossing in $\mathsf{Pop}(F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathbb{G})$ and at most $2g+2(n_0+2n_1)(r+1)$ units of credit are issued. By Lemma 9 the cost of representing new letters is covered by the released credit. Finally, both $\mathsf{TreeUnaryComp}$ and Pop preserve the invariants (GR1)–(GR5). □

Since by Lemma 6 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}) to $\mathbb{G}$, we get:

Corollary 2. *(F_1^{up}, F_1^{down})-compression issues in total $\mathcal{O}((g + (n_0 + n_1)r) \log n)$ 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 $\mathsf{TreeChainComp}$ (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 appearance in α_i, otherwise it has no crossing chain. As for $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression, when there are no crossing chains, we apply $\mathsf{TreeChainComp}$ to the right-hand sides of $\mathbb{G}$. We denote with $\mathsf{TreeChainComp}(F_1, \mathbb{G})$ the grammar obtained by applying $\mathsf{TreeChainComp}$ to all right-hand sides of $\mathbb{G}$.

Lemma 12. *Let* $\mathbb{G}' = \mathsf{TreeChainComp}(F_1, \mathbb{G})$. *If no unary letter has a crossing chain in a rule of* $\mathbb{G}$*, then* $\mathrm{val}(\mathbb{G}') = \mathsf{TreeChainComp}(F_1, \mathrm{val}(\mathbb{G}))$. *If* $\mathbb{G}$ *satisfies (GR1)–(GR5), then so does* $\mathbb{G}'$.

The proof is similar to the proof of Lemma 9 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. 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 because 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 beginn with a. Thus, we keep on popping up until the first letter of $\mathrm{val}(A_i)$ is not a. In order to do this in one step we need some notation: We say that a^ℓ is an *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). In this terminology, we remove the a-prefix of $\mathrm{val}(A_i)$. Similarly, we say that a^ℓ is an *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). The following algorithm $\mathsf{RemCrChains}$ eliminates crossing chains from $\mathbb{G}$.

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

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

Lemma 13. *After* $\mathsf{RemCrChains}$ *no unary letter has a crossing chain and* $\mathrm{val}(A_m) = \mathrm{val}(A'_m)$. *Furthermore, if* $\mathbb{G}$ *satisfies (GR1)–(GR5), then so does* $\mathbb{G}' = \mathsf{RemCrChains}(\mathbb{G})$.

Proof. We first show that indeed when $\mathsf{RemCrChains}$ 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 appears 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 $\mathsf{RemCrChains}$. Since whenever we removed the a-prefix a^{ℓ_i} from the rule for A_i we replace each occurrence of A_i by $a^{\ell_i} A_i$ and similarly for b-suffixes, we get $\mathrm{val}(A'_m) = \mathrm{val}(A_m)$ (note that we do not pop prefixes and suffixes from A_m).

Concerning the preservation of (GR1)–(GR5), the argument is the same as in the proof of Lemma 10 and so it is omitted. □

So chain compression is done by first running $\mathsf{RemCrChains}$ and then $\mathsf{TreeChainComp}$ on the right-hand sides of $\mathbb{G}$. Concerning the amount of issued credit, note that the arbitrarily long chains popped by $\mathsf{RemCrChains}$ are compressed into single letters by $\mathsf{TreeChainComp}$. Hence, only 4 units of credit per occurrence of a nonterminal from N_1 and 2 units of credit per occurrence of a nonterminal from $N_0 \cup \widetilde{N}_0$ have to be issued (as for $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression). We get:

Lemma 14. *Let* $\mathbb{G}$ *satisfy (GR1)–(GR5) and* $\mathbb{G}' = \mathsf{TreeChainComp}(F_1, \mathsf{RemCrChains}(\mathbb{G}))$*. Then* $\mathrm{val}(\mathbb{G}') = \mathsf{TreeChainComp}(F_1, \mathrm{val}(\mathbb{G}))$ *and* $\mathbb{G}'$ *satisfies (GR1)–(GR5) as well. Moreover, at most* $2g + 2(n_0 + 2n_1)(r+1)$ *units of credit are issued in the computation of* $\mathbb{G}'$ *and this credit is used to pay the credit for the fresh letters introduced by* $\mathsf{TreeChainComp}$ *(but not their representation cost).*

Since by Lemma 6 we apply at most $\mathcal{O}(\log n)$ many chain compressions to $\mathbb{G}$, we get:

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

The representation cost for the new letters a_ℓ introduced by chain compression will be 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 $\mathsf{TreeLeafComp}$ 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 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 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 15. *Let* $\mathbb{G}' = \mathsf{TreeLeafComp}(F_{\geq 1}, F_0, \mathbb{G})$*. 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* $\mathbb{G}'$*.*

Proof. Most of the proof follows similar lines as the proof of Lemma 9, 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 9 it suffices to show that $\mathrm{val}(\mathbb{G}')$ does not contain a subtree of the form $f(t_1, \ldots, t_\ell)$ with $f \in F_{\geq 1}$ such that there exist positions $1 \leq i_1 < i_2 < \cdots < i_k \leq \ell$ and constants $a_1, \ldots, a_k \in F_0$ with $t_{i_j} = a_j$ for $1 \leq j \leq k$ and t_i is a non-constant for $i \notin \{i_1, \ldots, i_k\}$ (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 will deduce a contradiction. If the root f together with its children at positions $i_1, \ldots, i_k$ are generated by some other nonterminal A_j appearing in α_i, then these nodes are compressed by the induction assumption. If they all appear explicitly in α_i 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 appearance 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 2 units are needed for the new node. The other 2ℓ 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, 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, the first case arises). 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). This ends the proof of the lemma. Note that we possibly add a second nonterminal to the right-hand side of $A_i \in N_0$ in the final case. □

If there is a crossing parent-leaf pair, 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, as when we pop a letter, it may have a 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 the issued credit is reduced and no new appearance 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 generalized popping procedure is called GenPop (Algorithm 8).

Lemma 16. *Let $\mathbb{G}$ satisfy (GR1)–(GR5) and let $\mathbb{G}' = \mathsf{GenPop}(F_{\geq 1}, F_0, \mathbb{G})$. Then $\mathbb{G}'$ has no crossing parent-leaf pair and satisfies (GR1)–(GR5). Moreover, at most $2g + 2(n_0 + 2n_1)(2r - 1)$ units of credit are issued during the run of* GenPop.

Proof. First, we show that (GR1)–(GR5) are preserved: Replacing nonterminals by constants and popping 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

Algorithm 8 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 a rule then
 6:         mark $A_i$                                    ▷ we need to pop down a handle from $A_i$
 7:     if $A_i$ is marked and $\alpha_i$ 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 $\alpha_i$ end with handle $f(t_1, \ldots, t_k, y, t_{k+1}, \ldots, t_\ell)$      ▷ $\alpha_i$ must end with a handle
12:         remove this handle from $\alpha_i$
13:         for $j \leftarrow 1 \,..\, \ell$ do
14:             if $t_j$ is not a constant then
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 $A_i(f(\gamma_1, \ldots, \gamma_k, t, \gamma_{k+1}, \ldots, \gamma_\ell))$
```

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.

So it is left to deal with (GR4), i.e. the bound on the number of appearances of nonterminals from $\widetilde{N}_0$, which is the only non-trivial task. When we remove the handle $f(t_1, \ldots, t_k, y, t_{k+1}, \ldots, t_\ell)$ 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 $A_i \in N_1$). Furthermore, when we replace each appearance of A_i with $A_i(f(A_{i_1}, \ldots, A_{i_k}, t, A_{i_{k+1}}, \ldots, A_{i_\ell}))$, those appearances of $A_{i_1}, \ldots, A_{i_\ell}$ are *owned* by this particular appearance of A_i. Since one appearance of a nonterminal cannot split into several copies during the modifications of $\mathbb{G}$ (though, in line 15 of GenPop we can move an appearance from one place to another), an appearance of A_{i_j} ceases to exist only when A_{i_j} is replaced by a constant (in line 3 of GenPop). Since by (GR3) there are at most $n_0 + 2n_1$ appearances of nonterminals from N_1, it is thus enough to show that each appearance of $A_i \in N_1$ owns at most $r-1$ appearances of nonterminals from $\widetilde{N}_0$. In fact, we show a stronger claim: Whenever we create new occurrences of the nonterminals $A_{i_1}, \ldots, A_{i_\ell}$ in line 20 (there are at most $r-1$ many) then just before line 20 A_i does not own any nonterminals. This is shown in a series of claims.

Claim 2. Each appearance of a nonterminal A_i owns the appearances of the same nonterminals $A_{i_1}, \ldots, A_{i_\ell}$.

This is obvious: We assign appearances of the same nonterminals $A_{i_1}, \ldots, A_{i_\ell}$ to each appearance of A_i in line 20 and the only way that such an appearance ceases to exist is when A_{i_j} is replaced with a constant, but this happens for all appearances 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 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.

Claim 3. 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 appearance 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 appearance of A_i owns the appearance 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 appearance 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 4. 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 $A_i(a)$ appears 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. By a simple 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 main proof for (GR4), 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 4 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 2 this occurrence of A_i owns an occurrence of A_q. Then by Claim 3 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.

Hence, at any time, every nonterminal from N_1 owns at most $r-1$ many nonterminals from $\widetilde{N}_0$. Since the number of occurrences of nonterminals from N_1 is bounded by $n_0 + 2n_1$ and every occurrence of a nonterminal from $\widetilde{N}_0$ is owned by a unique occurrence of a nonterminal from N_1, the bound in (GR4) follows.

Concerning the credit: We issue at most 2 units of credit per occurrence of a nonterminal from N_0 and $\widetilde{N}_0$ during popping up and $2r$ units per occurrence of a nonterminal from N_1 during popping down. So by (GR3) and (GR4) at most $2g + 2(n_0 + 2n_1)(r-1) + 2(n_0 + 2n_1)r = 2g + 2(n_0 + 2n_1)(2r-1)$ units of credit have to be issued.

Finally, we now 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 because of (FC1) and (FC3).

In order to rule out (FC2), we proceed with a series of claims: 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 $A_j(a)$ appears 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 if 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 $A_i(a)$ appears in the right-hand already after the loop in line 1, because a cannot appear afterwards and we do not pop-down a letter from A_i. 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 17. *Let* $\mathbb{G}$ *satisfy (GR1)–(GR5) and* $\mathbb{G}' = \mathsf{TreeLeafComp}(F_{\geq 1}, F_0, \mathsf{GenPop}(F_{\geq 1}, F_0, \mathbb{G}))$*. Then* $\mathrm{val}(\mathbb{G}') = \mathsf{TreeLeafComp}(F_{\geq 1}, F_0, \mathrm{val}(\mathbb{G}))$ *and* $\mathbb{G}'$ *satisfies (GR1)–(GR5) as well. At most* $2g + 2(n_0 + 2n_1)(2r-1)$ *units of credit are issued in the computation of* $\mathbb{G}'$*. The issued credit and the credit released by* TreeLeafComp *cover the representation cost of fresh letters as well as their credit.*

Proof. This is a combination of Lemma 15 and Lemma 16: GenPop makes sure that there are no crossing parent-leaf pairs and issues at most $2g+2(n_0+2n_1)(2r-1)$ units of credit by Lemma 16. Then by Lemma 15, 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). □

Since by Lemma 6 we apply at most $\mathcal{O}(\log n)$ many leaf compressions to $\mathbb{G}$, we get:

Corollary 4. *Leaf compression issues in total at most* $\mathcal{O}(((n_0 + n_1)r + g)\log n)$ *units of credit during all modifications of* $\mathbb{G}$.

From Corollaries 2, 3, and 4 we get:

Corollary 5. *The whole credit issued during all modifications of* $\mathbb{G}$ *is bounded by* $\mathcal{O}(((n_0 + n_1)r + g)\log n)$.

4.6. **Calculating the cost of representing letters in chain compression.** The issued credit is enough to pay the 2 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 ChainComp (the $\mathbb{G}$*-based representation*). Then, we show that for this scheme the representation cost is bounded by $\mathcal{O}((g + (n_0 + n_1)r)\log n)$. Lastly, it is proved that the actual cost of the representation of letters introduced by chain compression during the run of TtoG (the TtoG*-based representation*, which was defined in Section 3.1) is smaller than the $\mathbb{G}$-based one. Hence, it is bounded by $\mathcal{O}((g + (n_0 + n_1)r)\log n)$ as well.

4.6.1. $\mathbb{G}$*-based representation.* 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 to a shorter pattern a^k explicit occurrences of a in a rule to the left and right. 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. However, this cannot happen too often: When patterns of length $p_1, p_2, \ldots, p_\ell$ are compressed (at the cost of $\mathcal{O}(\sum_{i=1}^{\ell}(1 + \log p_i)) = \mathcal{O}(\log \prod_{i=1}^{\ell} p_i)$, as $p_i \geq 2$ for each $1 \leq i \leq \ell$), 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.

We create a new letter for each chain pattern from a^+ (where a is a unary symbol) that is either popped from a right-hand side or is in a rule at the end of the RemCrChains (i.e., after popping but before the actual replacement in TreeChainComp). Such a chain pattern is a *power* if it is obtained by concatenation of a popped suffix and a popped prefix inside a right-hand side (and perhaps some other letters that were in the rule before). For this to happen, in the rule $A_i \to uA_jvA_kw$ (or $A_i \to uA_jvA_k$) the popped suffix of A_j and popped prefix of A_k belong to a^+ for some unary letter a, and furthermore $v \in a^*$. Note that it might be that one (or both) of A_j and A_k are removed in the process and in such a case this power may be popped (up or down) from A_i. 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 a chain pattern a^ℓ may be created in several different ways. For instance, a power a^ℓ may be also created in a right-hand side by concatenating explicit occurrences of a to a shorter represented patter a^k. It is an arbitrary choice to declare a^ℓ a power in this case. Similarly, a non-power a^ℓ may be created in several right-hand sides by concatenating occurrences of a to shorter patterns. In this case, we arbitrarily fix one choice.

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 $\ell - k$ explicit occurrences of a from a right-hand side to a^k (recall that we fixed some choice in this case). In this case we represent a_ℓ as $a_k a^{\ell-k}$. The representation cost is $\ell - k + 1$; it is covered by the $2(\ell - k) \geq \ell - k + 1$ units of credit released from the $\ell - k > 0$ many explicit occurrences of a. (Recall that the credit for appearances of a fresh letter a_ℓ is covered by the issued credit, see Lemma 14, 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. 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 $\mathcal{O}((n_0 + n_1)\log n)$.

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 a nonterminal A_i produces powers of length $p_1, p_2, \ldots, p_\ell$ (which have total representation cost $\mathcal{O}(\sum_{i=1}^{\ell}(1 + \log p_i)) = \mathcal{O}(\log(\prod_{i=1}^{\ell} p_i)))$ 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 18. *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 time, see the last sentence in the proof of Lemma 15.

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 $\mathrm{val}(A_j)$ (resp., $\mathrm{val}(A_k)$) has a suffix (resp., prefix) from a^+.

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_ℓ. Lastly, the p_ℓ repetitions of $a^{(\ell)}$ are replaced by $a^{(\ell+1)}$. Observe, that $a^{(s+1)}$ does not need to be the letter that replaced $(a^{(s)})^{p_s}$, as there might have been some other compressions performed on that letter. 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 $s = 1, \dots, \ell$.

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

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 $a^{(s)}_{p_s}$ which has weight $p_s \cdot \mathrm{w}(a^{(s)})$ 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 symbol between A_j and A_k cannot decrease, we must have

$$\mathrm{w}(a^{(s+1)}) \geq \mathrm{w}(a^{(s)}_{p_s}) = p_s \, \mathrm{w}(a^{(s)}) \ .$$

Since $\mathrm{w}(a^{(1)}) \geq 1$ it follows that $\mathrm{w}(a^{(\ell+1)}) \geq \prod_{s=1}^{\ell} p_s$. As $\mathrm{w}(a^{(\ell+1)}) \leq n$ 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 similar form. In particular we do not have to consider the case when the second nonterminal A_k is removed from the rule.
- 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, so 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 $a^{(1)}, a^{(2)}, \dots, a^{(\ell)}$ and let their lengths be $p_1, p_2, \dots, p_\ell$. Lastly, the p_ℓ repetitions of $a^{(\ell)}$ are replaced by $a^{(\ell+1)}$. It is enough to show that $\mathrm{w}(a^{(s+1)}) \geq p_s \, \mathrm{w}(a_s)$ as then the rest of the proof follows as in the case of a nonterminal from N_1.

Consider two such consecutive compressions. In both case the second nonterminal (the one from N_0) cannot be removed, so in both cases the rule ends with a nonterminal from N_0. Hence the right-hand side after the first compression is $uA_j a^{(s)}_{p_s} A_k$ and right before the next compression it is $u' A_j (a^{(s+1)})^{p_{s+1}} A_{k'}$. By the earlier observation, the maximal weight of letters below A_j can only increase, hence $\mathrm{w}(a^{(s+1)}) \geq p_s \, \mathrm{w}(a_s)$ as claimed. □

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

Corollary 6. *The cost of the $\mathbb{G}$-based representation is $\mathcal{O}((g + (n_0 + n_1)r) \log n)$.*

Proof. Concerning powers, we redirect to each nonterminal from $N_0 \cup N_1$ a cost of $\mathcal{O}(\log n)$ by Lemma 18. There are at most $n_0 + n_1$ such nonterminals by (GR2). 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 $2g$) plus the total issued credit during all modifications of $\mathbb{G}$ (which is $\mathcal{O}((g + (n_0 + n_1)r) \log n)$ by Corollary 5). We get the statement of the lemma 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 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 represent both representations by edge-weighted graphs such that the total cost of a representation is bounded (up to a constant factor) by the 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 $a_k a^{\ell - k}$, 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 it is obvious that:

Lemma 19. *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^k to a^ℓ if and only if $\ell < k$ and there is no node a^q with $\ell < q < k$ (note that we may have $\ell = 0$).

Lemma 20. *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 3 guarantees that if $a^{\ell_1}, a^{\ell_2}, \ldots, a^{\ell_k}$ 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 $\mathcal{O}(\sum_{i=1}^{k}(1 + \log(\ell_i - \ell_{i-1})))$ (with $\ell_0 = 0$). □

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

Lemma 21. *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 appears in the $\mathbb{G}$-based representation but which does not appear in the TtoG-based representation. On the other hand, every node a^ℓ that appears in $\mathcal{G}_{\mathsf{TtoG}}$ also appears 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 13. Hence, a^ℓ appears 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) Lemmas 20, 21, and 19, followed by Corollary 6, we get:

Corollary 7. *The total cost of the* TtoG*-representation is* $\mathcal{O}((g + (n_0 + n_1)r) \log n)$.

4.7. **Total cost of representation.**

Corollary 8. *Let g be the size of the smallest handle grammar that generates T. The total representation cost of the letters introduced by* TtoG *(and hence the size of the grammar produced by* TtoG*) is* $\mathcal{O}((g + (n_0 + n_1)r) \log n) \leq \mathcal{O}(g \cdot r \cdot \log n)$.

Proof. By Corollary 7 the representation cost of letters introduced by chain compression is $\mathcal{O}((g + (n_0 + n_1)r) \log n)$, while by Lemma 11 and Lemma 17 the representation cost of letters introduced by unary pair compression and leaf compression is covered by the initial credit (which is at most $2g$) plus the total amount of issued credit. By Corollary 5 the latter is $\mathcal{O}((g + (n_0 + n_1)r) \log n)$, which ends the proof. □

Recall that Corollary 1 already showed that the smallest grammar generating T is most $\mathcal{O}(r)$ times smaller than the smallest handle grammar generating T, which yields an estimation on the approximation ratio of TtoG.

Corollary 9. *The approximation ratio of* TtoG *is $\mathcal{O}(r^2 \log n)$, where n is the size of the input tree T and r is the maximal rank of letters used in T.*

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 [19] and discussions, as well as Sebatian Maneth and Stefan Böttcher for the question of applicability of the recompression-based approach to the tree case.

References

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

[2] Philip Bille, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. Tree compression with top trees. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *ICALP (1)*, volume 7965 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 2013.

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

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

[5] Francisco Claude and Gonzalo Navarro. Fast and compact web graph representations. *ACM Transactions on the Web*, 4(4), 2010.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[7] Adria Gascón, Guillem Godoy, and Manfred Schmidt-Schauß. Unification and matching on compressed terms. *ACM Trans. Comput. Log.*, 12(4):26, 2011.

[8] Artur Jeż. Compressed membership for NFA (DFA) with compressed labels is in NP (P). In Christoph Dürr and Thomas Wilke, editors, *STACS*, volume 14 of *LIPIcs*, pages 136–147. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[9] Artur Jeż. Faster fully compressed pattern matching by recompression. In Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer, editors, *ICALP (1)*, volume 7391 of *LNCS*, pages 533–544. Springer, 2012.

[10] Artur Jeż. Approximation of grammar-based compression via recompression. In Johannes Fischer and Peter Sanders, editors, *CPM*, volume 7922 of *LNCS*, pages 165–176. Springer, 2013. full version at http://arxiv.org/abs/1301.5842.

[11] A. Jeż. One-variable word equations in linear time. In *Proceedings of ICALP 2013 (2)*, LNCS 7966, pages 324–335. Springer, 2013.

[12] Artur Jeż. Recompression: a simple and powerful technique for word equations. In Natacha Portier and Thomas Wilke, editors, *STACS*, volume 20 of *LIPIcs*, pages 233–244, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[13] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Proceedings of the 1999 Data Compression Conference (DCC 1999), Snowbird (Utah, USA)*, pages 296–305. IEEE Computer Society Press, 1999.

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

[15] Markus Lohrey and Sebastian Maneth. The complexity of tree automata and XPath on grammar-compressed trees. *Theoretical Computer Science*, 363(2):196–210, 2006.

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

[17] Markus Lohrey, Sebastian Maneth, and Eric Nöth. XML compression via DAGs. In *Proceedings of the 16th International Conference on Database Theory (ICDT 2013)*, 2013.

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

[19] Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

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

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

[22] Manfred Schmidt-Schauß. Matching of compressed patterns with character-variables. In *Proceedings of the 23rd International Conference on Rewriting Techniques and Applications, RTA 2012*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[23] Manfred Schmidt-Schauß, David Sabel, and Altug Anis. Congruence closure of compressed terms in polynomial time. In *Proceedings of the 8th International Symposium on Frontiers of Combining Systems, FroCos 2011*, volume 6989 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2011.

[24] James A. Storer and Thomas G. Szymanski. The macro model for data compression. In Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors, *STOC*, pages 30–39. ACM, 1978.

Appendix

RadixSort for lists of varying lengths. Imagine we want to sort sequences $\overline{a}_1, \overline{a}_2, \dots, \overline{a}_m$ of lengths $\ell_1, \ell_2, \dots, \ell_m$ from an alphabet $[1, \dots, k]$ lexicographically. Let $\ell = \max_{i=1}^m \ell_i$, $L = \sum_{i=1}^m \ell_i$, and $\overline{a}_i = (a_{i,1}, a_{i,2}, \dots a_{i,\ell_i})$. Standard RadixSort uses time $\mathcal{O}((k+m)\ell)$, see e.g. [6, Section 8.3]. We want to reduce it to $\mathcal{O}(k+L)$. The problem with RadixSort is that on each position $j = \ell, \ell-1, \dots, 1$ it checks buckets for each possible value in $[1, \dots, k]$. To avoid this, we first create for each position $j \in [1, \dots, \ell]$ a list $pos[j]$ which is a sorted list of all digits $a \in [1, \dots, k]$ that appear on position j in any of the sequences $\overline{a}_1, \overline{a}_2, \dots, \overline{a}_m$. This is easily done by creating a pair $(a_{i,j}, j)$ for each $1 \leq i \leq m$ and $1 \leq j \leq \ell_i$. Then, we sort these L many pairs using simple RadixSort in time $\mathcal{O}(L + \max\{\ell, k\}) \leq \mathcal{O}(L+k)$. Additionally, we create for each position $1 \leq j \leq \ell$ a list $\mathit{new}[j]$ of last elements of strings that have length exactly j together with the index i, i.e. (as a set) $\mathit{new}[j] = \{(a_{i,j}, i) \mid 1 \leq i \leq m, \ell_i = j\}$. Note that no sorting is needed, it is enough to go trough $\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_m$ in time $\mathcal{O}(L)$. Now in round $j = \ell, \ell-1, \ldots, 1$ we keep a list of letters on position j, i.e. $\{(a_{i,j}, i) \mid 1 \leq i \leq m, j \leq \ell_i\}$ sorted according to the lexicographic order on suffixes of $\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_m$ starting at position j. To update the list when going from $j+1$ to j we need to change each pair $(a_{i,j+1}, i)$ to $(a_{i,j}, i)$ and append the list $\mathit{new}[j]$ at the beginning and then stably sort them using CountingSort, see [6, Section 8.2] (which sorts a list of m number from the interval $[1, .., k]$ in time $\mathcal{O}(k+m)$). However, using the list $\mathit{pos}[j]$ we only need to inspect the digits that actually appear in our list, so the running time of CountingSort in the j-th phase can be reduced to $\mathcal{O}(|\mathit{pos}[j]| + |\{i \mid 1 \leq i \leq m, j \leq \ell_i\}|)$. Summing over all j yields a running time of $\mathcal{O}(L)$ and hence in total $\mathcal{O}(k+L)$.

Max Planck Institute für Informatik,, Campus E1 4, DE-66123 Saarbrücken, Germany, and Institute of Computer Science, University of Wrocław, ul. Joliot-Curie 15, 50-383 Wrocław, Poland, aje@cs.uni.wroc.pl

University of Siegen, Department of Electrical Engineering and Computer Science, 57068 Siegen, Germany, lohrey@eti.uni-siegen.de