

CONTEXT UNIFICATION IS IN PSPACE

ARTUR JEŽ

ABSTRACT. Contexts are terms with one ‘hole’, i.e. a place in which we can substitute an argument. In context unification we are given an equation over terms with variables representing contexts and ask about the satisfiability of this equation. Context unification is a natural subvariant of second-order unification, which is undecidable, and a generalization of word equations, which are decidable, at the same time. It is the unique problem between those two whose decidability is uncertain (for already almost two decades). In this paper we show that the context unification is in PSPACE. The result holds under a (usual) assumption that the first-order signature is finite.

This result is obtained by an extension of the recompression technique, recently developed by the author and used in particular to obtain a new PSPACE algorithm for satisfiability of word equations, to context unification. The recompression is based on performing simple compression rules (replacing pairs of neighbouring function symbols), which are (conceptually) applied on the solution of the context equation and modifying the equation in a way so that such compression steps can be in fact performed directly on the equation, without the knowledge of the actual solution.

1. INTRODUCTION

1.1. Context unification. Solving equations, whether they are over groups, fields, semigroups, terms or any other objects, was always a central point in mathematics and the corresponding decision problems received a lot of attention in the theoretical computer science community. Solving equations can be equally seen as unification problem, as we are to unify two objects (with some variables).

Context unification is one of prominent problems of this kind, let us first introduce the objects we will work on. A ground context is a ground term with exactly one occurrence of a special constant that represents a missing argument. Ground contexts can be applied to ground terms, which results in a replacement of the special constant by the given ground term; similarly we can define a composition of two ground contexts, which is again a ground context. Hence we can build terms using ground contexts, treating them as function symbols of arity 1.

In context unification we are given a finite signature, a set of variables (which shall denote ground terms) and a set of so-called context variables (which shall denote ground contexts). Using those variables we can build terms: we simply treat each context variable as a function symbol of arity one and each variable as a constant. A context equation is an equation between two such terms and a solution of a context equation assigns to each context variable a ground context (over the given input signature) and to each variable a ground term (over the same signature) such that both sides of the equation evaluate to the same (ground) term. The context unification is the decision problem, whether a context equation has a solution (as in some sense we unify the two contexts on the sides of the equation).

Context unification was introduced by Comon [1, 2] (who also coined the name) and independently by Schmidt-Schauß [27]. It found usage in analysis of rewrite systems with membership constraints [1, 2], analysis of natural language [22, 23], distributive unification [28], bi-rewriting systems [15].

In a broader sense, context unification is a special case of second-order unification, in which the argument of the second-order variable X can be used unbounded number of times in the substitution term for X (also, there may be many parameters for a second order variable, this is however not an essential difference). On the other hand, when the underlying signature is

Key words and phrases. context unification, second order unification, term rewriting.

Research supported by Humboldt Foundation Research Fellowship for Postdoctoral Researchers, 2013–14.

restricted to the case when only unary function symbols and constants are allowed, the context equation is in fact a word equation (in this well-known problem we are given an equation $u = v$, where u and v are strings of letters and variables and we are to substitute the variables with strings so that this formal equation is turned into a true equality of strings). The second order unification is known to be undecidable [6], (even in very restricted cases [4, 14, 17]) however, the proofs do not apply to the case of context unification as they essentially use the fact that the argument may be used many times in the substitution term. On the other hand, the satisfiability of word equations is known to be decidable (in PSPACE [24]) and up to recently there were essentially only three different algorithms for this problem [20, 25, 24]; whether these algorithms generalise to context unification remains an open question. Hence context unification is both upper and lower-bounded by two well-studied problems.

The problem gained considerable attention in the term rewriting community [26], mainly for two reasons: on one hand it is the only known natural problem which is subsumed by second order unification (which is undecidable) and subsumes word equations (which are decidable) and on the other hand it has several ties to other problems, see Section 1.2.

There was a large body of work focused on context unification and several partial results were obtained:

- a fragment in which any occurrence of the same context variable is always applied to the same term is decidable [2];
- stratified context unification, in which for any occurrence of a fixed second-order variable X the string of second-order variables from this occurrence to the root of the containing term is the same is decidable [29] (this problem is even known to be NP-complete [16] and in fact the result holds even for infinite signatures);
- a fragment in which every variable and context variable occurs at most twice (such equations are usually called *quadratic*) is decidable [14];
- a fragment when there are only two context variables is decidable [32];
- the notion of exponent of periodicity, which is crucial in algorithms for solving word equations, is generalised to context unification and so is the exponential bound on it [31];
- context unification reduces to the fragment in which the signature contains only one binary symbol and constants [19];
- context unification with one context variable is known to be in NP [5].

Note that in most cases the corresponding variants of the general second order unification remain undecidable, which gave hope that context unification is indeed decidable.

In this paper we show that context unification can be nondeterministically decided in space polynomial in k and n , where n is the size of the context equation and k is the maximal arity of function symbols in the signature (this means that we can consider infinite signatures as long as the maximal arity is bounded; however, this case in general reduces to the case of finite signatures).

1.2. Extensions and connections to other problems. The context unification was shown to be equivalent to ‘equality up to constraint’ problem [22] (which is a common generalisation of equality constraints, subtree constraints and one-step rewriting constraints). In fact one-step rewriting constraints, which is a problem extensively studied on its own, are equivalent to stratified context unification [23]. It is known that the existential theory of one-step rewriting constraints is undecidable [35, 21, 36]. The case of general context unification was improved by Vorobyov, who showed that its $\forall \exists^8$ -equational theory is Π_1^0 -hard [37].

Some fragments of second order unification are known to reduce to context unification: the *bounded second order unification* we assume that the number of appearances of the argument of the second-order variable in the substitution term is bounded by a constant (note that it *can be zero* and this is the crucial difference with context unification). This fragment on one hand easily reduces to context unification and on the other hand it is known to be decidable [30] (in fact its generalisation to higher-order unification is decidable as well [33] and it is known that bounded second order unification is NP-complete [16]). In particular, the work presented here imply the results on bounded second order unification.

The context unification can be also extended by allowing some additional constraints on the (context) variables, a natural one allows the usage of the tree-regular constraints (i.e. we assume that the substitution for the (context) variable comes from a certain regular set of trees). It is known that such an extension is equivalent to the linear second order unification [18], defined by Levy [14]: in essence, the linear second order unification allows bounding variables on different levels of the function, which makes direct translations to context unification infeasible, however, usage of regular constraints gives enough additional power to actually encode such more complicated bounding.

Notice the usage of regular constraints is very popular in case of word equations, in particular it is used in generalisations of the algorithm for word equation to the group case and both Makanin's and Plandowski's algorithms can be generalised to word equations with regular constraints [34, 3].

1.3. Recompression. The connection between compression and word equations was first observed and used by Plandowski and Rytter [25], who showed that each length-minimal solution (of size N) of the word equation (of size n) has $\text{poly}(n, \log N)$ description (in terms of LZ77). This connection was exploited more efficiently by Plandowski, whose PSPACE algorithm works on compressed representation of the word equation (and uses some finely tuned word factorisations to process this equation).

The recompression method, introduced recently by the author, is based solely of compression: it performs two simple compression steps (replacing pairs of letters ab by a new letter c , replacing maximal blocks a^ℓ by a new letter a_ℓ) on a word represented in some implicit way, say as a grammar, compression scheme or even as a solution of a word equation. In order to make such compression steps applicable, the implicit representation is modified a bit, for instance in case of word equations a variable x is replaced with ax or xb where a, b are letters. The intuition behind such a modification is apparent: when we want to replace each ab by c then some of those substrings appear explicitly in the equation (which are hence easy to replace), some in the substitution for the variables (which are thus replaced 'implicitly' by changing the solution) and some are 'crossing' between variables and letters in the equation (or two variables). The last case is the only problematic one, and it occurs (for a pair ab) when ax occurs in the equation and the solution for x begins with b (there is also the symmetric case). In such a case replacing ab is impossible. To fix this problem, we modify the equation, by replacing x with bx , thus 'left-popping' the letter out of the variable. It is easy to show that after left-popping b and 'right-popping' a the pair ab no longer has the problematic crossing occurrences and so it can be replaced with c in the equation. The crucial observation is that when the compression are done in a proper way, we can bound the size of the equation, as the number of letters popped into the equation is linear in the number of variables, while the compression steps guarantee shortening of the equation by a constant factor. Those two effects cancel each other out and so the equation has linear size.

This method turned out to be applicable to several problems for implicit representations of words [9, 7, 11, 8, 10] in particular it is applicable to the word equations problems in which case it yields a much simpler PSPACE algorithm that checks the satisfiability of the word equation (and returns a finite representation of all solutions) [11]. Retracing the compression steps yields an SLP (so a context free grammar generating a unique string) for the size-minimal solution of the word equation and a simple analysis show that the size of this SLP is $\text{poly}(n, \log N)$, which yields an alternative proof of the result by Plandowski and Rytter [25] (with slightly better bounds). Quite surprisingly, this algorithm, when restricted to the case of word equations with only one variable yields a linear time algorithm [10], improving the previously known algorithms (of course some further analysis and usage of tailored data structures is needed).

Applications of compression to fragments of context unification are known [33] and this paper extends the recompression method to terms in full generality. In this way solving word equations using recompression [11] generalises to solving context unification, which in some sense fulfils the plan of extending the algorithms for word equations to context unification. In particular, it provides a tree-grammar of size $\text{poly}(n, \log N)$ generating a solution of a context equation.

A word can be seen as a term over signature containing only unary symbols (plus some constant at the bottom) and vice versa. Thus the two compression operations for word equations generalise naturally to subterms containing only unary function symbols. Hence the recompression for terms uses the two already mentioned operations (which are applicable only to function symbols of arity one ¹) but it also introduces another local compression rule, designed specifically for terms: we replace a term $f(t_1, \dots, t_{i-1}, c, t_{i+1}, \dots, t_m)$ (where c is a constant) with $f'(t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_m)$, where f' is a fresh function symbol (i.e. not used the context equation, it can however be in Σ). While such a compression introduces new function symbols, it does not increase the maximal arity of functions in the signature, which proves to be important (as the space consumption depends on this maximal arity). This new rule requires also a generalisation of the variable replacements (x by ax or xb): when X denotes a context, we sometimes replace it with $a(X)$, where a is a unary letter, or $X(f(x_1, x_2, \dots, x_{i-1}, \Omega, x_i, \dots, x_m))$, where x_1, x_2, \dots, x_m are new variables denoting full terms and Ω denotes the place in which we apply the argument.

As in the case of word equations, the key observation is that while the variable replacements increase the size of the context equation (proportionally to the number of occurrences of variables in the context equation), the replacement rules guarantee that the size of the context equation is decreased by a constant factor (for proper nondeterministic choices). Those two effects cancel each out and the size of the context equation remains linear.

Note that the generalisation of the recompression to terms is independently considered also in the joint work of the author and with Lohrey [12] on tree grammars and the presentation of the recompression for terms there is similar to the one given here.

1.4. Outline. First, in Section 2, we explain in detail how to generalise the recompression from strings to trees. Then, in Section 3, we define formally the context unification and state some of its basic properties. As a next step we identify the easy cases (so called *non-crossing*), in which the compression rules can be applied directly to the context equations, see Section 4. The main technical part of this paper is the explanation how to modify the context equations so that each context equation is reduced to such an easy case mentioned above, so that the compression schema can be applied directly to context equations, which is done in Section 5; this technique is called *uncrossing* (and, despite its significance, is very easy). The last Section 6 wraps everything up, by presenting the full statement of the algorithm for context unification as well as a (relatively simple) analysis of it as well as the polynomial bound on the space consumption.

2. COMPRESSION OF TREES

In this section we generalise the technique of local compression of trees. It is independently used in work of Jež and Lohrey on tree-grammar compressions and the presentation there is similar [12].

2.1. Labelled trees and their compression. We deal with rooted, ordered trees, usually denoted with letters t or s . Nodes are labelled with elements from a ranked alphabet Γ , i.e. each letter $a \in \Gamma$ has a fixed arity $\text{ar}(a)$. A tree is *well-formed* if a node labelled with f has exactly $\text{ar}(f)$ children. Unless explicitly written, we consider only well-formed trees, which can be equivalently seen as *ground terms* over Γ .

In the following we usually consider a set of labels Σ which is finite but growing during the run of our algorithm. We call the labels from Σ *letters* and pay particular attention to letters of arity 1 (*unary letters*) and to letters of arity 0 (*constants*). On the other hand, Γ (perhaps with some subscripts) is used for some subalphabet of Σ , say, letters used in some particular tree or letters of arity at least 1.

¹Note that by work of Levy [19] it is enough to consider context unification with constants and a single binary symbol. However, our algorithm will transform the input instance and it can introduce unary symbols. So even if the input satisfies such a condition, we cannot guarantee that the current context equation stored by the algorithm also satisfies it.

We want to replace fragments of a tree with new letters. Those fragments are not necessarily well-formed. Thus we define a *subtree* in a natural way, in general not necessarily well-formed, but in such a case we explicitly mention it. A *pattern* is a tree (perhaps not well-formed) in which a node labelled with f has *at most* $\text{ar}(f)$ children; since we imagine a pattern as a part of a term with some of the subterm removed, the $0 \leq m \leq \text{ar}(f)$ children of f in the pattern are numbered $1 \leq i_1 < i_2 < \dots < i_m \leq \text{ar}(f)$ to denote which children of f are those in a ‘real term’. A subpattern of a tree t is any subtree which is a pattern; we often consider individual *occurrences* of subpatterns of a tree t . In this terminology, our algorithm will replace occurrences of subpatterns of t in t (the subtree rooted in children which are omitted in the subpattern are attached in the same order, details are given later).

A *chain* is a pattern that consists only of unary letters. We consider 2-chains, so consisting only of two unary letters (usually different) and a -chains, which consists solely of letters a . We treat chains as strings and write them in the string notation and ‘concatenate’ them, i.e. for two chains s and s' the ss' denotes the chain obtained by attaching the top-most node in s to the bottom node in s' . A chain t' that is a subpattern of t is a *chain subpattern* of t , an occurrence of a chain subpattern a^ℓ is *a-maximal* if it cannot be extended by a nor up nor down.

2.2. Local compression of trees. We perform three types of compressions on a tree t , all of them replace subpatterns by a single letter:

a -maximal chain compression: For a unary letter a we replace each a -maximal chain subpattern a^ℓ by a new unary letter a_ℓ (making the father of a^ℓ the father of a_ℓ and the unique child of a^ℓ the unique child of a_ℓ).

a, b pair compression: For two unary letters a and b we replace each 2-chain subpattern ab with a new unary letter c .

(f, i, c) leaf compression: For a constant c and letter f of arity $\text{ar}(f) = m \geq i \geq 1$, we replace each subtree $f(t_1, t_2, \dots, t_{i-1}, c, t_{i+1}, \dots, t_m)$ with $f'(t_1, t_2, \dots, t_{i-1}, t_{i+1}, \dots, t_m)$ where f' is a fresh letter of arity $m-1$ added to Σ (intuitively: the constant c is ‘absorbed’ by its father).

Those operations are applied (in some specific order) on a tree t until it is reduced to a single leaf.

Observe that the a -maximal chain compression and a, b pair compression are direct translations of the operations used in the recompression-based algorithm for word equations [11]. To be more precise, both those compressions affect only chains, return chains as well, and when a chain is treated as a string the result of those compressions corresponds to the result of the corresponding operation on strings. On the other hand, the leaf compression is a new operation that is designed specifically to deal with trees.

2.2.1. Parallel compressions. To make the compression more effective, we apply several compression steps in parallel: consider the a -maximal chain compression. As a -maximal and b -maximal chain subpatterns do not overlap (it does not matter whether $a = b$ or not), we can perform a -maximal chain compression for all $a \in \Gamma_1$ in parallel (as long as the letters that are used to replace the chains are not taken from Γ_1). We call the resulting procedure **TreeChainComp**(Γ_1, t) or simply *chain compression*, when Γ_1 and t are clear from the context.

Algorithm 1 **TreeChainComp**(Γ_1, t): Compression of chains of letters from Γ_1 in a tree t

Require: Γ_1 contains only unary letters

- 1: **for** each $a \in \Gamma_1$ **do** ▷ Chain compression
 - 2: **for** each $\ell \in \mathbb{N}$ **do**
 - 3: replace each a -maximal occurrence of chain subpattern a^ℓ in t by a_ℓ
-

An important property of the chain compression is that afterwards a father and son cannot be labelled with the same unary letter.

Lemma 1. *After the chain compression performed for all unary letters in the obtained tree there is no node labelled with the same unary letter as its father.*

Proof. Suppose that in t' there is a father and son labelled with the same unary letter a . If a was not introduced by the chain compression, then we arrive at a contradiction, as those two letters should have been replaced with one unary letter. If a replaced some chain b^ℓ then we again obtain a contradiction, as aa represents a chain $b^{2\ell}$, so none of the two replaced b^ℓ was maximal. \square

We would like to perform also several (f, i, a) compressions, for $f \in \Gamma_{\geq 1}$ and $a \in \Gamma_0$ in parallel. Clearly for a fixed node labelled with $f \in \Gamma_{\geq 1}$ we can perform several different leaf compressions with its children (that are from Γ_0) at the same time, in this way we could define $(f, i_1, a_1, i_2, a_2, \dots, i_\ell, a_\ell)$ leaf compression, which replaces a subpattern f with children a_j at position i_j for $j = 1 \dots, \ell$ with a new letter f' with those children removed. However, in this way two different $(f, i_1, a_1, i_2, a_2, \dots, i_\ell, a_\ell)$ and $(f, i'_1, a'_1, i'_2, a'_2, \dots, i'_{\ell'}, a'_{\ell'})$ leaf compressions could be applied to the same node labelled with f and the result depends on the order of those two compressions, in particular they cannot be applied in parallel. To remedy this, we apply the $(f, i_1, a_1, i_2, a_2, \dots, i_\ell, a_\ell)$ leaf compression only to nodes that do not have children labelled with letters from Γ_0 on positions other than $\{i_1, \dots, i_\ell\}$. More formally, the $(f, i_1, a_1, i_2, a_2, \dots, i_\ell, a_\ell)$ leaf compression replaces each subtree $f(t_1, t_2, \dots, t_{i_1-1}, a_1, t_{i_1+1}, \dots, t_{i_\ell-1}, a_\ell, t_{i_\ell+1}, \dots, t_k)$ with $f'(t_1, \dots, t_k)$ when $t_i \notin \Gamma_0$ for each $i \notin \{i_1, \dots, i_\ell\}$. Clearly such a compression can be also performed for different labels f and different tuples $(i_1, a_1, i_2, a_2, \dots, i_\ell, a_\ell)$ in parallel, as long as we do not try to compress also the letters introduced during the compression. This is formalised in the following algorithm $\text{TreeLeafComp}(\Gamma_{\geq 1}, \Gamma_0, t)$, when $\Gamma_{\geq 1}$, Γ_0 , and t are clear from the context, we simply talk about *leaf compression*.

Algorithm 2 $\text{TreeLeafComp}(\Gamma_{\geq 1}, \Gamma_0, t)$: leaf compression for parent nodes in $\Gamma_{\geq 1}$, and leaf-children in Γ for a tree t

Require: $\Gamma_{\geq 1}$ contains no constant, Γ_0 contains only constants

- 1: **for** $f \in \Gamma_{\geq 1}, 0 < i_1 < i_2 < \dots < i_\ell \leq \text{ar}(f) =: m, (a_1, a_2, \dots, a_\ell) \in \Gamma_0^\ell$ **do**
 - 2: replace each subtree $f(t_1, \dots, t_m)$ s.t. $t_{i_j} = a_j$ for $1 \leq j \leq \ell$ and $t_i \notin \Gamma_0$ for $i \notin \{i_1, \dots, i_\ell\}$
 by $f'(t_1, \dots, t_{i_1-1}, t_{i_1+1}, \dots, t_{i_\ell-1}, t_{i_\ell+1}, \dots, t_m)$ $\triangleright f' \notin \Gamma_{\geq 1} \cup \Gamma_0$
-

In case of the pair compression the situation is a bit more difficult: observe that in a chain subpattern 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 word equations [11], parallel a, b pair compressions are possible when we take a and b from disjoint subalphabets Γ_1 and Γ_2 , respectively. Those subalphabets are usually a partition of letters present in some tree and so we call them a partition, even if we do not explicitly say of what. 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 pairs, as long as new letters are outside $\Gamma_1 \cup \Gamma_2$. This is formalised in the below algorithm $\text{TreePartitionComp}(\Gamma_1, \Gamma_2, t)$, when t is clear from the context (or unimportant) we refer to it simply as Γ_1, Γ_2 compression (we list the Γ_1 and Γ_2 to stress the dependency of the procedure on them).

Algorithm 3 $\text{TreePartitionComp}(\Gamma_1, \Gamma_2, t)$: Γ_1, Γ_2 -compression for a tree t

Require: Γ_1, Γ_2 contains only unary letters and are disjoint

- 1: **for** $a \in \Gamma_1$ and $b \in \Gamma_2$ **do**
 - 2: replace each occurrence of a chain subpattern ab with a fresh letter c $\triangleright c \notin \Gamma_1 \cup \Gamma_2$
-

The main property of the the listed procedures is that they shrink the tree by a constant factor; to be more precise: the chain compression followed by a Γ_1, Γ_2 child compression (for a proper choice of partition (Γ_1, Γ_2)) followed by a leaf compression applied to a tree t results in a tree t' which is smaller by a constant factor than t . This should be intuitively clear: the leaf compression removes all leaves; this would halve the size of the the tree if there were no nodes with unary labels. But such nodes are compressed on their own: first by chain compression and

then by the Γ_1, Γ_2 compression and it is known from the earlier work on word equations that for appropriate partition of unary letters a word is shortened by a constant factor by those two operations [11]. The details are presented in the algorithm below and the following theorem.

Algorithm 4 TreeComp(t): Compression of a tree t

- 1: $\Gamma \leftarrow$ unary letters in t
 - 2: $t \leftarrow$ TreeChainComp(Γ, t)
 - 3: $\Gamma \leftarrow$ unary letters in t
 - 4: guess partition of Γ into Γ_1 and Γ_2
 - 5: $t \leftarrow$ TreePartitionComp(Γ_1, Γ_2, t)
 - 6: $\Gamma_0 \leftarrow$ constants in t , $\Gamma_{\geq 1} \leftarrow$ other letters in t
 - 7: $t \leftarrow$ TreeLeafComp($\Gamma_{\geq 1}, \Gamma_0, t$)
-

Theorem 1. Consider tree t on which we run TreeComp and the obtained tree t' . For some partition Γ_1, Γ_2 it holds that $|t'| < \frac{3|t|}{4}$.

As a first technical step we show that when no node is labelled with the same unary letter as its father, the claim of the theorem holds (note that then the chain compression does not change the tree). Then it is enough to see that the chain compression cannot increase the size of the tree and that after it there are no two such nodes, by Lemma 1.

Lemma 2. Consider a tree t in which no node is labelled with the same unary letter as its father and a run of TreeComp(t). Let t' the tree obtained after the Γ_1, Γ_2 compression and t'' the tree obtained after the leaf compression. For some partition Γ_1, Γ_2 it holds that $|t''| < \frac{3|t|}{4}$.

Proof. Note that by the assumption that no node is labelled with the same unary label as its father, the TreeChainComp returns the same tree t .

Let n_0, n_1 and $n_{\geq 2}$ denote, respectively, the number of leaves, nodes with only one child and other nodes in t , n'_0, n'_1 and $n'_{\geq 2}$ the number of such nodes in t' , n''_0, n''_1 and $n''_{\geq 2}$ in t'' . We show that $n''_0 + n''_1 + n''_{\geq 2} < \frac{3}{4}(n_0 + n_1 + n_{\geq 2})$, which shows the claim.

Clearly

$$(1) \quad n_{\geq 2} \leq n_0 - 1 .$$

This is easy to show: except for a root, each vertex has a father, i.e. there are $n_0 + n_1 + n_{\geq 2} - 1$ sons, on the other hand, we can estimate the number of sons by calculating the number of children, which yields that there are at least $n_1 + 2n_{\geq 2}$ sons, hence $2n_{\geq 2} + n_1 \leq n_0 + n_1 + n_{\geq 2} - 1$, yielding (1).

Concerning the Γ_1, Γ_2 compression, we first need some notions: we say that an occurrence of a chain subpattern ab in t is *covered* by Γ_1, Γ_2 if $a \in \Gamma_1$ and $b \in \Gamma_2$. We claim that there is a partition of unary letters in t (i.e. Γ) into Γ_1, Γ_2 such that at least $\frac{n_1 - c}{4}$ occurrences of two letter chain subpatterns in t are covered by Γ_1, Γ_2 , where c is the number of maximal chains in t .

Claim 1. Let Γ be the set of unary letters in t' . There is a partition Γ_1, Γ_2 of Γ such that at least $\frac{n'_1 - c}{4}$ occurrences of 2-chains subpatterns in t are covered by Γ_1, Γ_2 , where c is the number of maximal chains in t .

Proof. Consider a random partition of Γ into Γ_1 and Γ_2 , which assigns each letter from Γ to Γ_1 or Γ_2 with equal probability. Then for a fixed occurrence of a two letter chain subpattern ab in t (note that by the assumption $a \neq b$) with probability $1/4$ this occurrence is covered by the partition: the probability that $a \in \Gamma_1$ is $1/2$, probability that $b \in \Gamma_2$ is $1/2$ as well and as $a \neq b$ those two events are independent. Since there are $n_1 - c$ occurrences of 2-chain subpatterns in total, expected number of occurrences covered by a partition is $\frac{n_1 - c}{4}$, so for some partition at least $\frac{n_1 - c}{4}$ occurrences are covered. \square

We should now estimate c —the number of maximal chains in t , it is at most

$$(2) \quad c \leq n_{\geq 2} + \frac{n_0}{2} + \frac{1}{2} .$$

Indeed, consider an occurrence of a maximal chain subpattern in t . Then the node above has a label of arity at least 2 (unless the occurrence of a maximal chain subpattern includes the root) while the node below has a label of arity other than 1. 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 (2).

Thus for the choice of Γ_1, Γ_2 from Claim 1 there are at least $\frac{n_1-c}{4}$ 2-chains compressed, so the tree is smaller by at least $\frac{n_1-c}{4}$ nodes, hence the value of $n'_0 + n'_1 + n'_{\geq 2}$ is at most

$$\begin{aligned}
 n'_0 + n'_1 + n'_{\geq 2} &\leq n_0 + n_1 + n_{\geq 2} - \frac{n_1 - c}{4} && \text{from Claim 1} \\
 &= n_0 + \frac{3n_1}{4} + n_{\geq 2} + \frac{c}{4} && \text{simplification} \\
 &\leq n_0 + \frac{3n_1}{4} + n_{\geq 2} + \underbrace{\frac{n_{\geq 2}}{4} + \frac{n_0}{8} + \frac{1}{8}}_{c/4} && \text{from (2)} \\
 (3) \quad &= \frac{9n_0}{8} + \frac{3n_1}{4} + \frac{5n_{\geq 2}}{4} + \frac{1}{8} && \text{simplification .}
 \end{aligned}$$

Lastly, the leaf compression simply removes all $n'_{\geq 2}$ leaves, which is exactly $n_{\geq 2}$ (no leaves are created or removed during the previous compression steps). Hence

$$\begin{aligned}
 n''_0 + n''_1 + n''_{\geq 2} &\leq n'_0 + n'_1 + n'_{\geq 2} - n_0 && \text{leaf compression} \\
 &\leq \frac{9n_0}{8} + \frac{3n_1}{4} + \frac{5n_{\geq 2}}{4} + \frac{1}{8} - n_0 && \text{from (3)} \\
 &\leq \frac{n_0}{8} + \frac{3n_1}{4} + \frac{5n_{\geq 2}}{4} + \frac{1}{8} && \text{simplification} \\
 &= \frac{3}{4} \left(n_0 + n_1 + n_{\geq 2} \right) + \left(-\frac{5n_0}{8} + \frac{n_{\geq 2}}{2} + \frac{1}{8} \right) && \text{desimplification} \\
 &< \frac{3}{4} \left(n_0 + n_1 + n_{\geq 2} \right) && \text{from (1) .}
 \end{aligned}$$

□

Now the proof of Theorem 1 follows.

proof of Theorem 1. The tree obtained from t after the chain compression is clearly at most as large as before, so it is enough to show that the application of Γ_1, Γ_2 compression followed by the leaf compression reduces the size of tree by at least one fourth. This is shown in Lemma 2, ut with the additional assumption that there is no node labelled with the same unary letter as its father. But by Lemma 1 we know that this assumption holds after the chain compression. Which ends the proof. □

The essential part of the paper is showing, how to modify the equation so that the compression steps can be performed directly on the context equation.

In the next sections the following observation, which bounds the maximal arity of the letters introduced during the compression steps, proves useful

Lemma 3. *If the maximal degree of nodes in t is k then in $t' = \text{TreeComp}(t)$ the maximal degree of a node is also at most k .*

Proof. Observe that the chain compression replaces chain of unary nodes with a single unary node. Similarly, Γ_1, Γ_2 compression replaces chains of length two with single unary letters. Lastly, leaf compression can only reduce the arity of a node (or keep it the same). □

3. CONTEXT UNIFICATION

In this section we (more formally) define the problem of context unification and the notions necessary to state the problem. The presentation here is based on [32].

Recall that Σ is the set of letters used as labels for nodes in trees. By Ω we denote a special constant outside Σ (and no letter added to Σ may be equal to Ω). \mathcal{V} denotes an infinite set of context variables X, Y, Z, \dots . We also use individual variables x, y, z, \dots taken from \mathcal{X} .

Definition 1 (cf. [32, Definition 2.1]). A *ground context* is a ground $(\Sigma \cup \{\Omega\})$ -term t , where $\text{ar}(\Omega) = 0$, that has exactly one occurrence of the constant Ω . The ground context Ω is the *empty ground context*.

The intuition of the symbol Ω is that it is a ‘hole’ and that one should replace this hole with a ground term to obtain a proper ground term.

Given a ground context s and a ground term/context t we write st for the ground term/context that is obtained from s when we replace the occurrence of Ω in s by t . (This form of composition is associative.) In the same spirit, when a is a unary letter, we usually write at to denote $a(t)$.

Definition 2 (cf. [32, Definition 2.2]). The *terms* over $\Sigma, \mathcal{X}, \mathcal{V}$ are ground terms with alphabet $\Sigma \cup \mathcal{X} \cup \mathcal{V}$ in which we extend ar to $\mathcal{X} \cup \mathcal{V}$ by $\text{ar}(X) = 1$ and $\text{ar}(x) = 0$ for each $x \in \mathcal{X}$ and $X \in \mathcal{V}$.

The *context terms* are ground context terms over $\Sigma \cup \mathcal{X} \cup \mathcal{V} \cup \{\Omega\}$ with exactly one occurrence of Ω and ar extended to $\mathcal{X} \cup \mathcal{V}$ as above and $\text{ar}(\Omega) = 0$.

A *context equation* is an equation of the form $u = v$ where both u and v are terms.

We call the letters from Σ that occur in a context equation the *explicit letters* and talk about explicit occurrences of letters in a context equation.

3.1. Solutions. We are interested in the solutions of the context equations, i.e. substitutions that replace variables with ground terms and context variables with ground contexts, such that a formal equality $u = v$ is turned into a true equality of ground terms. More formally:

Definition 3 (cf. [32, Definition 2.3]). A *substitution* is a mapping S that assigns a ground context $S(X)$ to each context variable $X \in \mathcal{V}$ and a ground term $S(x)$ to each variable $x \in \mathcal{X}$. The mapping S is naturally extended to arbitrary terms as follows:

- $S(a) := a$ for each constant $a \in \Sigma$;
- $S(f(t_1, \dots, t_n)) := f(S(t_1), \dots, S(t_n))$ for an n -ary $f \in \Sigma$;
- $S(Xt) := S(X)S(t)$ for $X \in \mathcal{V}$.

A substitution S is a *solution* of the context equation $u = v$ if $S(u) = S(v)$. A solution S is *size-minimal*, if for every other solution S' it holds that $|S(u)| \leq |S'(u)|$. A solution S is non-empty if $S(X) \neq \Omega$ for each $X \in \mathcal{V}$ from the context equation $u = v$.

In the following, we are interested only in non-empty solutions. Notice that this is not restricting, as for the input instance we can guess, which context variables have empty substitution in the solution and remove them.

For a ground term $S(u)$ and an occurrence of a letter a in it we say that this occurrence *comes from* u if it was obtained as $S(a)$ in Definition 3 and that it comes from X (or x) if it was obtained from $S(X)$ (or $S(x)$, respectively) in Definition 3.

Let the maximal arity of letters in Σ be k . We claim that without loss of generality we may assume that for each $k' \leq k$ the Σ contains a letter of arity k' ; this is formalised in the following lemma.

Lemma 4. *Let Σ be a signature (that contains a constant) and $f \in \Sigma$ be the letter of maximal arity (k) in Σ and $u = v$ be a context equation. If $u = v$ has a solution S' over a signature Σ' such that each letter in $\Sigma' \setminus \Sigma$ is not in $u = v$ and has arity at most k then $u = v$ also has a solution, which is at most k times larger.*

Note that if a signature does not contain a constant then no terms can be build using it.

Proof. Let f be a letter of arity k in Σ and a any constant in Σ . Let h be any letter used by S' which is neither in $u = v$ nor in Σ , let $m = \text{ar}(h)$. We change S' by replacing each $h(t_1, t_2, \dots, t_m)$ by $f(t_1, t_2, \dots, t_m, \underbrace{a, a, \dots, a}_{k-m \text{ times}})$. It is easy to see that the new substitution is also

a solution: as h is not used in the equation, each of its occurrences in $S(u)$ or $S(v)$ comes from $S(X)$ or $S(x)$ and we replace each such occurrence with $f(\underbrace{\dots a, a, \dots a}_{k-m \text{ times}})$. Iterating over all h

yields the claim. \square

In the following we use the Lemma 4 implicitly — we always assume that Σ contains letters of arity $0, 1, \dots, k$. Note that this assumption may influence the size of the size-minimal solution (as it may decrease sizes of some solutions), we disregard this small effect and assume that the signature preprocessing is done before any algorithm is run.

3.2. Properties of solutions. In case of word equations, size-minimal solutions are considered mainly because one can bound the *exponent of periodicity* for them, below we recall a known similar fact for context equations.

Lemma 5 (Exponent of periodicity bound [31]). *Let S be a size-minimal solution of a context equation $u = v$. Suppose that $S(X)$ (or $S(x)$) can be written as ts^mt' , where t, s, t' are ground context terms (or t' is a ground term, respectively). Then $m = 2^{\mathcal{O}(|u|+|v|)}$.*

We use Lemma 5 only for the case when s is a unary letter, for which the proof simplifies significantly and is essentially the same as in the case of word equations [11] (which is a simplification of the general bound on the exponent of periodicity by Kościelski and Pacholski [13]).

Furthermore in case of word equation the minimality of solution is used also for another purpose: whenever a letter a occurs in the minimal solution, it needs to occur also in the equation [25].

Lemma 6 ([25]). *Let S be a length minimal solution of a word equation $u = v$. If a letter a occurs in $S(u)$ then it occurs also in u or v .*

The property from Lemma 6 is very useful, as we can still deduce the set of letters used by minimal solutions simply by looking at the equation, in particular we can restrict ourselves to Σ such that $|\Sigma| \leq |u| + |v|$.

However, this is *not* the case of the context-equations: an equation $X(a) = Y(b)$ over a signature $\{f, a, b\}$, with f being binary and a, b being constants, has a solution (which is easily seen to be size-minimal) $S(X) = f(\Omega, b)$ and $S(Y) = f(a, \Omega)$ and in fact each solution needs to use f , which does not occur in the context equation. Still, a very similar property holds for context equations: We say that for solutions S and S' of $u = v$ the S' is a *simpler equivalent* of S if S' is obtained by applying a letter homomorphism on top of S (i.e. we exchange each letter a in $S(X)$ and $S(x)$ by some fixed $h(a)$ of the same arity). Then following simple lemma gives the a relatively close approximation of Lemma 6 for the case of context equations:

Lemma 7. *Consider a context equation $u = v$ over a signature Σ , such that the maximal arity of letters in Σ is k . Then for every solution S there is a simpler equivalent S' such that for each $k' = 0, 1, 2, \dots, k$ it uses at most one letter of arity k' that is not used in $u = v$.*

Furthermore, for each solution S there exists a solution S' that uses only unary letters that are present in $u = v$ and $|S'(u)| \leq |S(u)|$.

Note that the usage of Lemma 7 is similar to the one of Lemma 6: whenever we have an equation $u = v$ over some large signature Σ (with maximal arity k) we can remove from Σ all (except for $k + 1$) letters that do not occur in the equation $u = v$ while preserving satisfiability.

Note also that Lemma 7 that among the size-minimal solutions there exist one that does not any unary letters not present in the equation $u = v$.

Proof. Let S be a solution of $u = v$. Fix some $0 \leq k' \leq k$ and all letters f, g, h, \dots that occur in $S(u)$ but not in $u = v$. Replace each occurrence of g, h, \dots in $S(X)$ and $S(x)$ (for each context variable X and each variable x) with f . We claim that the new substitution S' is also a solution: since the letters g, h, \dots did not occur in the equation $u = v$ then the only way that they could occur in $S(u)$ and $S(v)$ is from $S(X)$ or $S(x)$. But in all of those variables and context variables we uniformly replaced g, h, \dots with f . So the equality $S'(u) = S'(v)$ still holds. Iterating this argument for $k' = 0, 1, 2, \dots, k$ yields the first claim.

To show the second claim consider a solution S and a unary letter a it uses which is not in $u = v$. Consider a new solution S' which is obtained from S by deleting each a , i.e. replacing each subterm $a^\ell t$ where a^ℓ is a -maximal with t (since a is a unary letter, this is a valid operation on terms). Since all a in $S(u)$ and $S(v)$ came from the S , the $S'(u)$ is obtained from $S(u)$ by deleting all as , similarly $S'(v)$ is obtained from $S(v)$ by deleting all as . Hence $S'(u) = S'(v)$, which shows that indeed S' is a solution of $u = v$. Iterating over all unary letters a in S that are not in $u = v$ yields the second claim. \square

The usage of Lemma 7 is as follows: when we want to perform the compression steps we need to know what are the letters used by (some) solution. Lemma 7 states that without loss of generality we can restrict ourselves to letters present in the equation and one letter for each arity. Furthermore, when we are concerned only with unary letters, we can consider only letters that are present in the equation.

4. COMPRESSION OF NON-CROSSING SUBPATTERNS

In this section we adapt the compressions from Section 2 to the case when the terms are given implicitly, i.e. as a solution of a context equation. To this end we identify cases, in which performing such a compression is easy and those in which it is hard and show how to make the compression in the easy cases. In the next section we present how to transform the difficult cases to the easy ones. Finally, in Section 6 we wrap everything up and present the algorithm for context-unification together with the space-usage analysis.

However, before stating the procedures that transform the equations, we formalise the notions about their correctness. This might be a little non-obvious, as our procedures in general make non-deterministic choices.

4.1. Soundness and Completeness. The intuition of the correctness of a non-deterministic procedure is clear: if the context equation is satisfiable then for some non-deterministic choices we should transform it to a (simpler) satisfiable instance. If it is unsatisfiable, we can transform it only to a non-satisfiable instance, regardless of the non-deterministic choices.

Definition 4. A (nondeterministic) procedure is *sound*, when given a unsatisfiable word equation $u = v$ it cannot transform it to a satisfiable one, regardless of the nondeterministic choices; such a procedure is *complete*, if given a satisfiable equation $u = v$ for some nondeterministic choices it returns a satisfiable equation $u' = v'$.

Observe, that a composition of sound (complete) procedures is also sound (complete, respectively)

A very general class of operations is sound:

Lemma 8. *The following operations are sound:*

- (1) Replacing all occurrences of a context variable X (variable x) with tX (tx , respectively) throughout the $u = v$, where t is a context term.
- (2) Replacing all occurrences of a context variable X with Xt throughout the $u = v$ where t is a context term.
- (3) Replacing all occurrences of a variable x with a ground term t .
- (4) (f, i, a) leaf compression performed on $u = v$.
- (5) a, b pair compression performed on $u = v$.
- (6) a -maximal chain compression performed on $u = v$.

Note that a context term may include variables, letters of large arity etc. However, we use Lemma 8 in a very restricted scenario, in which we use only constants unary letters as context terms (except case 2, in which we replace X with $X(f(x_1, x_2, \dots, x_{i-1}, \Omega, x_{i+1}, \dots, x_m))$).

Proof. The proof follows a simple principle: if the obtained equation $u' = v'$ has a solution S' then we can define a solution S of the original context equation by reversing the performed operation.

In 1, if S' is a solution of the new equation then $S(X) = S'(t)S'(X)$ is a solution (the same holds for x).

Similarly, in 2, if S' is a solution of the new equation then $S(X) = S'(X)S'(t)$ is a solution of the original equation.

In 3 if S' is a solution of the new equation, we define S in the same way, but set $S(x) = t$.

In 4, let f' denote the letter that replaced f with child a at positions i during the (f, i, a) leaf compression. Let S' be a solution of the new equation, we define a solution S : if $S'(X)$ contains the occurrences of a letter f' , then we replace the whole subterm $f'(t_1, t_2, \dots, t_{i-1}, t_{i+1}, \dots, t_k)$ in $S'(X)$ with $f(t_1, \dots, t_{i-1}, a, t_{i+1}, \dots, t_k)$, the same is done for $S(x)$.

In case 5, if the letter c occurs in $S(X)$ or $S(x)$ then we replace it with a chain pattern ab .

Similarly, in the last case S is obtained from S' by replacing each occurrence of a letter a_ℓ with a chain a^ℓ (for all ℓ).

It is easy to see that all those operations define a valid solution of the original equation. \square

4.2. Non-crossing partitions and their compression. We begin the considerations with the Γ_1, Γ_2 compression as it is the easiest to explain and the intuition behind is most apparent.

Consider a context equation $u = v$ and a solution S . Suppose that we want to perform the Γ_1, Γ_2 compression on $S(u)$ and $S(v)$, i.e. we want to replace each occurrence of a chain subpattern $ab \in \Gamma_1 \Gamma_2$ with a fresh unary letter c . Such replacement is easy, when the occurrence of ab subpattern comes from the letters in the equation or from $S(X)$ (or $S(x)$) for some context variable X (or a variable x , respectively): in the former case we modify the equation by replacing the subpattern ab with c , in the latter the modification is done implicitly (i.e. we replace the subpattern ab in $S(X)$ or $S(x)$ with c). The problematic part is with the ab chain subpattern that is of neither of those forms, as they ‘cross’ between $S(X)$ (or $S(x)$) and some letter outside this $S(X)$ (or $S(x)$). This is formalised in the below definition.

Definition 5. For an equation $u = v$ and a non-empty substitution S we say that an occurrence of a chain subpattern ab in $S(u)$ (or $S(v)$) is

- explicit with respect to S :** the occurrences of both a and b come from explicit letters a and b in $u = v$;
- implicit with respect to S :** the occurrences of both a and b come from $S(x)$ (or $S(X)$);
- crossing with respect to S :** otherwise.

We say that ab is a *crossing pair* with respect to S if they have at least one crossing occurrence with respect to S . Otherwise ab is a *non-crossing pair* (with respect to S).

Unless explicitly written, we consider only crossing/noncrossing pairs ab in which $a \neq b$.

The notions of a crossing chain subpattern can be defined in a more operational manner: for a non-empty substitution S by *first letter* of $S(X)$ ($S(x)$) we denote the topmost-letter in $S(X)$ ($S(x)$, respectively), by the *last letter* of $S(X)$ we denote the function symbol that is the father of Ω in $S(X)$. Then it is easy to see that ab is crossing with respect to S if and only if one of the following conditions hold for some context variables X, Y (or variable y):

- aX (or ax) is a chain subpattern in $u = v$ and b is the first letter of $S(X)$ (or $S(x)$, respectively) *or*
- Xb is a chain subpattern in $u = v$ and a is the last letter of $S(X)$ *or*
- XY (or Xy) is a chain subpattern in $u = v$, a is the last letter of $S(X)$ and b the first letter of $S(Y)$ ($S(y)$, respectively).

These conditions prove to be useful afterwards.

Since we perform several pair compression in one go, we generalise a definition of a crossing pairs to partitions:

Definition 6. A partition Γ_1, Γ_2 of Γ is *non-crossing* (with respect to a solution S) if there is no pair ab with $a \in \Gamma_1$ and $b \in \Gamma_2$ such that ab is a crossing pair (with respect to S); otherwise it is *non-crossing* with respect to S .

When a partition Γ_1, Γ_2 is non-crossing with respect to a solution S , we can simulate the $\text{TreePartitionComp}(\Gamma_1, \Gamma_2, S(u))$ on $u = v$ simply by performing the Γ_1, Γ_2 compression on

the explicit letters in the equation: then occurrences of ab that come from explicit letters are compressed, the ones that come from $S(X)$ and $S(x)$ are compressed by changing the solution and there are no other possibilities. To be more precise we treat the equation $u = v$ as a term over $\Sigma \cup \mathcal{X} \cup \mathcal{V} \cup \{=\}$ (imagine u and v as children of the root labelled with '=', which has arity 2) and apply the Γ_1, Γ_2 pair compression on this tree, we refer to this operation as $\text{TreePartitionComp}(\Gamma_1, \Gamma_2, 'u = v')$ (note that context variables are not in Γ_1 , nor in Γ_2 while variables as well as '=' have arity other than 1 so they cannot be compressed either).

Algorithm 5 $\text{PartitionComp}(\Gamma_1, \Gamma_2, 'u = v')$ Γ_1, Γ_2 compression for a non-crossing partition Γ_1, Γ_2

Require: Γ_1, Γ_2 contain only unary letters and are a non-crossing partition

- 1: run $\text{TreePartitionComp}(\Gamma_1, \Gamma_2, 'u = v')$ ▷ Treat, $u = v$ as a tree
 - ▷ Variables and context variables are not compressed
-

Lemma 9. $\text{PartitionComp}(\Gamma_1, \Gamma_2, 'u = v')$ is sound.

If $u = v$ has a solution S such that Γ_1, Γ_2 is a non-crossing partition with respect to S then $\text{PartitionComp}(\Gamma_1, \Gamma_2, 'u = v')$ is complete, to be more precise, the returned equation $u' = v'$ has a solution S' such that $S'(u') = \text{TreePartitionComp}(\Gamma_1, \Gamma_2, S(u))$.

Proof. Note that while $\text{PartitionComp}(\Gamma_1, \Gamma_2, 'u = v')$ applies several ab pair compression for $ab \in \Gamma_1 \Gamma_2$ in parallel, since Γ_1 and Γ_2 are disjoint, we can in fact think that they are done sequentially. Furthermore, when done 'sequentially', the pairs in $\Gamma_1 \Gamma_2$ do not become non-crossing, as we do not introduce any new letters from Γ_1 , nor Γ_2 to the equation, nor to the solution. For each such compression we use Lemma 8 to show that it is sound and so also $\text{PartitionComp}(\Gamma_1, \Gamma_2, 'u = v')$ is.

No, concerning the completeness. Suppose that $u = v$ has a solution S such that Γ_1, Γ_2 is a non-crossing partition with respect to S . We define a substitution S' for the obtained equation $u' = v'$ such that $S'(u') = \text{TreePartitionComp}(\Gamma_1, \Gamma_2, S(u))$ and symmetrically $S'(v') = \text{TreePartitionComp}(\Gamma_1, \Gamma_2, S(v))$. Since $S(u) = S(v)$ this shows that S' is indeed a solution of $u' = v'$ and so the second claim of the lemma holds.

The definition is straightforward: $S'(X)$ is obtained by performing the Γ_1, Γ_2 compression on $S(X)$ (the $S'(x)$ is defined in the same way) formally $S'(X) = \text{TreePartitionComp}(\Gamma_1, \Gamma_2, S(X))$ (note that Ω is not in Γ_1 , nor in Γ_2 and so it is not replaced).

Consider $a \in \Gamma_1$ with child labelled with $b \in \Gamma_2$ in $S(u)$. Consider where this chain subpattern ab comes from:

they both come from explicit letters: Then $\text{PartitionComp}(\Gamma_1, \Gamma_2, 'u = v')$ will perform the Γ_1, Γ_2 compression on them, i.e. replace them with a letter c .

they both come from $S(X)$ or $S(x)$: Then this occurrence of ab is replaced by the definition of S' .

one of them comes from an explicit letter and one from $S(X)$ or $S(x)$: But then Γ_1, Γ_2 is a crossing partition with respect to S , contradicting the assumption.

As the argument applies to every occurrence of chain subpattern $ab \in \Gamma_1 \Gamma_2$, this shows that $S'(u') = \text{TreePartitionComp}(\Gamma_1, \Gamma_2, S(u))$, which ends the proof of the lemma. \square

4.3. Non-crossing a -maximal chains and their compression. Similarly, consider a context equation $u = v$ and a solution S . Suppose that we want to perform the a -maximal chain compression on $S(u)$ and $S(v)$. Then all occurrences of a -maximal chains are to be replaced with new unary letters. Such replacement is easy, when the chain is a chain subpattern of the equation or is a chain subpattern of $S(X)$ (or $S(x)$) for some context variable X (or a variable x , respectively). The problematic part is with the occurrences that are of neither of those forms, as they 'cross' between $S(X)$ (or $S(x)$) and another subtree. This is formalised in the below definition.

Definition 7. For an equation $u = v$ and a substitution S we say that an occurrence of an a -maximal chain subpattern a^ℓ in $S(u)$ (or $S(v)$) is

- explicit with respect to S :** if this occurrence comes wholly from u (or v), i.e. it is a chain subpattern of u (or v);
- implicit with respect to S :** if this occurrence comes wholly from $S(X)$ or $S(x)$, i.e. it is a chain subpattern of $S(X)$ or $S(x)$;
- crossing with respect to S :** otherwise.

We say that a has a *crossing chain* if there is at least one occurrence of a crossing a -maximal chain subpattern. Otherwise, a has no crossing chain.

As in the case of Γ_1, Γ_2 , it is easy to see that a has a crossing chain with respect to a non-empty solution S if and only if one of the following holds for some context variables X, Y (or variable y):

- ax (or aX) is a chain subpattern in $u = v$ and the first letter of $S(x)$ ($S(X)$, respectively) is a ;
- Xa is a chain subpattern in $u = v$ and a is the last letter of $S(X)$;
- XY (or Xy) is a chain subpattern in $u = v$ and a is the last letter of $S(X)$ as well as the first letter of $S(Y)$ (or $S(y)$).

When no unary letter (from Γ) has a crossing chain to simulate the chain compression on the context equation we perform the **TreeChainComp** on the explicit letters, treating the context equation as a tree, similarly as in the case of the Γ_1, Γ_2 compression.

Algorithm 6 ChainComp($\Gamma, 'u = v'$): Compressing chains when there is no crossing chain

Require: Γ contains only unary letters, there are no crossing chains for letters in Γ

- 1: run TreeChainComp($\Gamma, 'u = v'$) \triangleright Treat, $u = v$ as a tree
 \triangleright Context variables are not compressed
-

Lemma 10. ChainComp is sound.

If $u = v$ has a solution S such that no letter in Γ has a crossing chain then it is complete, to be more precise, the returned equation $u' = v'$ has a solution S' such that $S'(u') = \text{TreeChainComp}(\Gamma, S(u))$.

The proof is essentially the same as in Lemma 12 and so it is omitted.

4.4. Non-crossing father-leaf pairs and their compression. Suppose now that given a context equation $u = v$ with a solution S we would like to perform leaf compression on $S(u)$ and $S(v)$. To this end we need to identify each $f \in \Gamma_{\geq 1}$ and its children in Γ_0 and replace them accordingly. Again, this is easy if each occurrence of such a subpattern comes either from explicit letters in $u = v$ or wholly from $S(X)$ (or $S(x)$). In such a case we proceed similarly as in the case of Γ_1, Γ_2 -compression and chain compression and treat the $u = v$ as a tree and perform the leaf compression on it. We are left to identify the cases in which this indeed properly simulates the leaf compression, which are similar to those in Γ_1, Γ_2 compression.

Definition 8. Let $\text{ar}(f) \geq 1$ and $\text{ar}(a) = 0$ and consider a subpattern consisting of f with a child a (on some position $i \leq \text{ar}(f)$). For an equation $u = v$ and a substitution S we say we say that an occurrence of such a subpattern is

- explicit with respect to S :** if both the occurrence of f and a come from explicit letters in u (or v);
- implicit with respect to S :** if both the occurrence of f and a come from some $S(X)$ or $S(x)$;
- crossing with respect to S :** otherwise.

Then (f, a) is a *crossing parent-leaf pair* in $u = v$ with respect to S if it has at least one crossing occurrence in $u = v$ with respect to S . Otherwise it is *noncrossing* with respect to S .

It is easy to observe that there is such a crossing father-leaf pair f, a (with respect to a non-empty S) if and only if one of the following holds for some context variables X and y

- f with a son y is a subpattern in $u = v$ and $S(y) = a$ or

- Xa is a subpattern in $u = v$ and the last letter of $S(X)$ is f or
- Xy is a subpattern in $u = v$, $S(y) = a$ and f is the last letter of $S(X)$.

When there is no crossing father-leaf pair (f, a) for $f \in \Gamma_{\geq 1}$ and $a \in \Gamma_0$ then to simulate leaf compression on $S(u)$ and $S(v)$ it is enough to perform it on the equation, treating it as a tree.

Algorithm 7 LeafComp($\Gamma_{\geq 1}, \Gamma_0, 'u = v'$): Leaf compression when there is no crossing father-leaf pair

Require: $\Gamma_{\geq 1}$ contains no constant, Γ_0 contains only constants,

there is no crossing father-leaf pair (f, a) with $f \in \Gamma_{\geq 1}$ and $a \in \Gamma_0$

- 1: run TreeLeafComp($\Gamma_{\geq 1}, \Gamma_0, 'u = v'$) \triangleright Treat, $u = v$ as a tree
 - \triangleright Context variables and variables are not compressed
-

Lemma 11. LeafComp is sound.

If $u = v$ has a solution S such that there is no crossing father-leaf pair (f, a) with $f \in \Gamma_{\geq 1}$ and $a \in \Gamma_0$ in $u = v$ with respect to S then it is complete, more precisely, the returned equation $u' = v'$ has a solution S' such that $S'(u') = \text{TreeLeafComp}(\Gamma_{\geq 1}, \Gamma_0, S(u))$.

The proof is essentially the same as in Lemma 12 and so it is omitted.

5. UNCROSSING

In general, one cannot assume that an arbitrary partition Γ_1, Γ_2 is noncrossing, similarly we cannot assume that there are no crossing chains nor crossing father-leaf pairs. However, for a fixed partition Γ_1, Γ_2 and a solution S we can modify the instance so that this fixed partition becomes non-crossing with respect to a solution S' (that corresponds to S of the original equation); similarly, given an equation $u = v$ we can turn it into an equation that has no letters with a crossing chain with respect to a solution S' of the new equation; lastly, for $\Gamma_{\geq 1}$ and Γ_0 we can modify the instance so that no father-leaf pair (f, a) with $f \in \Gamma_{\geq 1}$ and Γ_0 is crossing with respect to S' . Those modifications are the cornerstone of our main algorithm, as they allow compression to be performed directly on the equation, regardless of how the solution actually look like.

5.1. Uncrossing partitions. We begin with showing how to turn a partition into a non-crossing one. Recall that Γ_1, Γ_2 is a crossing partition (with respect to a non-empty S) if and only if for some $ab \in \Gamma_1 \Gamma_2$ one of the following holds for some context variables X, Y (or variable y), we assume here that S is non-empty

- (CP1) aX (or ax) is a chain subpattern in $u = v$ and b is the first letter of $S(X)$ (or $S(x)$, respectively) or
- (CP2) Xb is a chain subpattern in $u = v$ and a is the last letter of $S(X)$ or
- (CP3) XY (or Xy) is a chain subpattern in $u = v$, a is the last letter of $S(X)$ and b the first letter of $S(Y)$ ($S(y)$, respectively).

In each of those cases it is easy to modify the instance so that ab is no longer a crossing pair:

- In (CP1) we *pop up* the letter b : we replace X (x) with bX (bx , respectively). In this way we also modify the solution $S(X)$ ($S(x)$) from $S(X) = bt$ ($S(x) = bt$, respectively) to $S'(X) = t$ ($S'(x) = t$, respectively). If $S'(X)$ is empty, we remove X from the equation.
- In (CP2) we *pop down* the letter a : we replace each occurrence of X with Xa . In this way we implicitly modify $S(X) = sa\Omega$ to $S'(X) = s$. If $S'(X)$ is empty, we remove X from the equation.
- The case (CP3) is a combination of the two cases above, in which we need to pop-down from X and pop-up from Y (or y).

It is easy to observe that this procedure can be performed on all $ab \in \Gamma_1 \Gamma_2$ in parallel, as presented in the algorithm below.

Algorithm 8 $\text{Pop}(\Gamma_1, \Gamma_2, 'u = v')$

```

1: for  $X \in \mathcal{V}$  do
2:   let  $a$  be the last letter of  $S(X)$  ▷ Guess
3:   if  $a \in \Gamma_1$  then
4:     replace each occurrence of  $X$  in  $u = v$  by  $Xa$ 
5:     ▷ Implicitly change  $S(X) = sa\Omega$  to  $S(X) = s$ 
6:     if  $S(X)$  is empty then ▷ Guess
7:       remove  $X$  from  $u = v$ : replace each  $X(s)$  in by  $s$ 
8: for  $X \in \mathcal{V}$  or  $x \in \mathcal{X}$  do
9:   let  $b$  be the first letter of  $S(X)$  (or  $S(x)$ ) ▷ Guess
10:  if  $b \in \Gamma_2$  then
11:    replace each occurrence of  $X$  in  $u = v$  by  $bX$  (or  $x$  with  $bx$ )
12:    ▷ Implicitly change  $S(X) = bs$  to  $S(X) = s$  or  $S(x) = bt$  to  $S(x) = t$ 
13:    if  $S(X)$  is empty then ▷ Guess
14:      remove  $X$  from  $u = v$ : replace each  $X(s)$  in by  $s$ 

```

We show that if $u = v$ has a solution S then for appropriate non-deterministic choices $\text{Pop}(\Gamma_1, \Gamma_2, 'u = v')$ returns an equation $u' = v'$ that has a solution S' such that Γ_1, Γ_2 is non-crossing with respect to S' , furthermore S' somehow corresponds to S .

Lemma 12. *Suppose that Γ_1, Γ_2 are disjoint. Then $\text{Pop}(\Gamma_1, \Gamma_2, 'u = v')$ is sound and complete. To be more precise, if $u = v$ has a non-empty solution S then for appropriate non-deterministic choices the returned equation $u' = v'$ has a non-empty solution S' such that $S'(u') = S(u)$ and Γ_1, Γ_2 is a non-crossing partition with respect to S' .*

Proof. By iterative application of Lemma 8 we obtain that $\text{Pop}(\Gamma_1, \Gamma_2, 'u = v')$ is sound.

Concerning the second part of the lemma, for simplicity of presentation we deal only with the first part of Pop , i.e. the one in which the letters are popped-down, the second part is dealt with similarly.

Suppose that $\text{Pop}(\Gamma_1, \Gamma_2, 'u = v')$ always makes the non-deterministic choices according to S (i.e. whenever we make a guess about $S(X)$ or $S(x)$ we guess correctly). Let us define a new substitution S' , the value of $S'(X)$ depends on actions performed on X by Pop :

- if X popped up b and $S(X) = bt$ (which holds, as $\text{Pop}(\Gamma_1, \Gamma_2, 'u = v')$ chose according to S and so the first letter of $S(X)$ is b) then $S'(X) = t$;
- if X did not pop any letter up then $S'(X) = S(X)$.

Note that X is removed from the equation if and only if $S'(X) = \Omega$.

It is easy to verify that indeed in each case the defined S' is a solution of the obtained equation $u' = v'$ and $S'(u') = S(u)$, as claimed: when X is not modified, its substitution is the same, if the pops up b , then its solution loses this b . Furthermore, S' is non-empty (as if it is empty then we remove the empty context variable).

So suppose that the partition Γ_1, Γ_2 is crossing with respect to S' , i.e. there exists $a \in \Gamma_1$ and $b \in \Gamma_2$ such that one of (CP1)–(CP3) holds. We consider only the case (CP1), in which aX is a chain subpattern in $u = v$ and b is the first letter of $S(X)$, other cases are shown in a similar way.

Consider, whether X popped up a letter.

- X popped a letter up:** In such a case the father of X is labelled with $b' \in \Gamma_2$, a contradiction, as the case assumption is that the father is labelled with $a \in \Gamma_1$ and $\Gamma_1 \cap \Gamma_2 = \emptyset$.
- X did not pop a letter up:** Since we consider the non-deterministic choices made according to S , we know that the first letter of $S(X)$ is outside Γ_2 . And by definition of S' we know that $S'(X)$ has the same first letter as $S(X)$, i.e. outside Γ_2 . A contradiction with the case assumption.

Analysis of cases (CP2)–(CP3) leads to a contradiction in a similar way, so the argument is skipped, which ends the proof. \square

5.2. Uncrossing chains. Suppose that some unary letter a has a crossing chain with respect to a non-empty solution S . Recall that a has a crossing chain if and only if one of the following holds for some context variables X, Y (or variable y)

- (CC1) ax (or aX) is a chain subpattern in $u = v$ and the first letter of $S(x)$ ($S(X)$, respectively) is a ;
- (CC2) Xa is a chain subpattern in $u = v$ and a is the last letter of $S(X)$;
- (CC3) XY (or Xy) is a chain subpattern in $u = v$ and a is the last letter of $S(X)$ as well as the first letter of $S(Y)$ (or $S(y)$).

The first two cases are symmetric while the third is a composition of the first two. So suppose that the second case holds. Then we can replace X with Xa throughout the equation $u = v$ (implicitly changing the solution $S(X) = ta\Omega$ to $S(X) = t$) but it can still happen that a is the last letter of $S(X)$. So we keep popping down a until the last letter of $S(X)$ is not a , in other words we replace X with Xa^r , where $S(X) = ta^r\Omega$ and the last letter of t is not a . Then a and X can no longer satisfy condition (CC2), as $S'(X)$ ends with a letter different than a . A symmetric action and analysis apply to (CC1), and (CC3) follows by applying the popping down for X and popping up for Y (or y). To simplify the arguments, for a ground term or context t we say that a^ℓ is the a -prefix of t if $t = a^\ell t'$ and the first letter of t' is not a (t' may be empty). Similarly, for a ground context t we say that b^r is a b -suffix of t if $t = t'b^r\Omega$ and the last letter of t' is not b (in particular, t' may be empty).

Algorithm 9 CutPrefSuff($\Gamma_1, 'u = v'$) Uncrossing all chains

```

1: for  $X \in \mathcal{V}$  or  $x \in \mathcal{X}$  do
2:   let  $a$  be the first of  $S(X)$  (or  $S(x)$ )
3:   if  $a \in \Gamma_1$  then
4:     guess  $\ell \geq 1$  ▷  $a^\ell$  is the  $a$ -prefix of  $S(X)$  or  $S(x)$ 
5:     replace each  $X$  (or  $x$ ) in  $u = v$  by  $a^\ell X$  (or  $a^\ell x$ ) ▷  $\ell$  is stored using  $\mathcal{O}(\ell)$  bits
6:     ▷ implicitly change  $S(X) = a^\ell t$  to  $S(X) = t$  (or  $S(x) = a^\ell t$  to  $S(x) = t$ )
7:     if  $S(X)$  is empty then ▷ Guess
8:       remove  $X$  from  $u = v$ : replace each  $X(t)$  by  $t$ 
9:   for  $X \in \mathcal{V}$  do
10:    let  $b$  be the last letter of  $S(X)$ 
11:    if  $b \in \Gamma_1$  then
12:      guess  $r \geq 1$  ▷  $b^r$  is the  $b$ -suffix of  $S(X)$ 
13:      replace each  $X$  in  $u = v$  by  $Xb^r$  ▷  $b^r$  is stored in a compressed form
14:      ▷ implicitly change  $S(X) = tb^r\Omega$  to  $S(X) = t$ 
15:      if  $S(X)$  is empty then ▷ Guess
16:        remove  $X$  from  $u = v$ : replace each  $X(t)$  by  $t$ 

```

Lemma 13. CutPrefSuff($\Gamma_1, 'u = v'$) is sound and complete; to be more precise, if $u = v$ has a non-empty solution S then for appropriate non-deterministic choices the returned equation $u' = v'$ has a solution S' such that $S'(u') = S(u)$ and there are no crossing chains with respect to S' .

The proof is essentially the same as the proof of Lemma 12 and so it is omitted.

5.3. Uncrossing father-leaf pairs. Now it is left to show how to ensure that there is no crossing father-leaf pair (f, a) with $f \in \Gamma_{\geq 1}$ and $a \in \Gamma_0$. Recall that there is such a pair (f, a) (with respect to a non-empty S) if and only if one of the following holds for some context variable X and variable y :

- (CFL 1) f with a son x is a subpattern in $u = v$ and $S(x) = a$ or
- (CFL 2) Xa is a subpattern in $u = v$ and the last letter of $S(X)$ is f or
- (CFL 3) Xy is a subpattern in $u = v$, $S(y) = a$ and f is the last letter of $S(X)$.

The modifications needed to uncross the father-leaf pair are in fact the only new uncrossing operations, when compared with the recompression technique for strings, however, they are similar to the one in the case of uncrossing partition Γ_1, Γ_2 . Note that in some sense we even have a partition: Γ_0 and $\Gamma_{\geq 1}$ and we want to pop-up from Γ_0 and pop-down from $\Gamma_{\geq 1}$. The former operation is trivial, but the details of the latter are not, let us present the intuition.

- In (CFL1) we *pop up* the letter a from x , which in this case means that we replace each x with $a = S(x)$. Since x is no longer in the context equation, we can restrict the solution so that it does not assign any value to x .
- In (CFL2) we *pop down* the letter f : let $S(X) = sf(t_1, \dots, t_{i-1}, \Omega, t_{i+1}, \dots, t_m)$, where s is a ground context and each t_i is a ground term and $\text{ar}(f) = m$. Then we replace each X with $Xf(x_1, x_2, \dots, x_{i-1}, \Omega, x_{i+1}, \dots, x_m)$, where $x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_m$ are fresh variables. In this way we implicitly modify the solution $S(X) = sf(t_1, t_2, \dots, t_{i-1}, \Omega, t_{i+1}, \dots, t_m)$ to $S'(X) = s$ and add $S'(x_j) = t_j$ for $j = 1 \dots, i-1, i+1, \dots, m$. If $S'(X)$ is empty, we remove X from the equation.
- The third case (CFL3) is a combination of (CFL1)–(CFL2), in which we need to down pop from X and pop up from y .

It is easy to observe that this procedure can be performed on all $f \in \Gamma_{\geq 1}$ and $a \in \Gamma_0$ in parallel, as presented in the algorithm below; this uncrosses all father-leaf pair (f, a) for $f \in \Gamma_{\geq 1}$ and $a \in \Gamma_0$.

Algorithm 10 $\text{GenPop}(\Gamma_{\geq 1}, \Gamma_0, 'u = v')$

```

1: for  $x \in \mathcal{X}$  do
2:   if  $S(x) \in \Gamma_0$  then ▷ Guess
3:     replace each  $x$  in  $u = v$  by  $S(x)$  ▷  $S$  is no longer defined on  $x$ 
4: for  $X \in \mathcal{V}$  do
5:   let  $f$  be the last letter of  $S(X)$  ▷ Guess
6:   if  $f \in \Gamma_{\geq 1}$  and for some  $a \in \Gamma_0$  the  $Xa$  is a subpattern in  $u = v$  then
7:     let  $m = \text{ar}(f)$ 
8:     let  $i$  be such that  $\Omega$  labels the  $i$ -th child of its father in  $S(X)$  ▷ Guess
9:     replace each  $X$  in  $u = v$  by  $Xf(x_1, x_2, \dots, x_{i-1}, \Omega, x_{i+1}, \dots, x_m)$ 
       ▷ Implicitly change  $S(X) = sf(t_1, t_2, \dots, t_{i-1}, \Omega, t_{i+1}, \dots, t_m)$  to  $S(X) = s$ 
       ▷ Add new variables  $x_1, \dots, x_m$  to  $\mathcal{X}$  with  $S(x_j) = t_j$ 
10:    if  $S(X)$  is empty then ▷ Guess
11:      remove  $X$  from the equation: replace each  $X(u)$  by  $u$ 
12: for new variables  $x \in \mathcal{X}$  do
13:   if  $S(x) \in \Gamma_0$  then ▷ Guess
14:     replace each  $x$  in  $u = v$  by  $S(x)$  ▷  $S$  is no longer defined on  $x$ 

```

There is a subtle difference between uncrossing a partition Γ_1, Γ_2 and uncrossing father-leaf pairs: for a partition popping down letters from Γ_1 is unconditional while the corresponding popping down the last letters $f \in \Gamma_{\geq 1}$ from X is done only when it is really needed: i.e. we want to make some (f, i, a) leaf compression, f is the last letter of $S(X)$, its i -th child is Ω and some occurrence of X is applied on a . This assumption turns out to be crucial to bound the number of introduced variables, see Lemma 17.

Lemma 14. *Let $\Gamma_{\geq 1}$ be a set of some letters of arity at least 1 and Γ_0 set of some constants, then $\text{GenPop}(\Gamma_{\geq 1}, \Gamma_0, 'u = v')$ is sound.*

It is complete, to be more precise, if $u = v$ has a non-empty solution S then for appropriate non-deterministic choices the returned equation $u' = v'$ has a non-empty solution S' such that $S'(u') = S(u)$ and there is no crossing father-leaf pair (f, a) with $f \in \Gamma_{\geq 1}$ and $a \in \Gamma_0$ with respect to S' .

Proof. The proof is similar as in the case of Lemma 12, however, some details are different so it is supplied.

By iterative application of Lemma 8 we obtain that $\text{GenPop}(\Gamma_{\geq 1}, \Gamma_0, 'u = v')$ is sound.

Concerning the second part of the lemma, we proceed as in Lemma 8: let $\text{GenPop}(\Gamma_{\geq 1}, \Gamma_0, 'u = v')$ always make the non-deterministic choices according to the S : we replace x with a when $S(x) = a \in \Gamma_0$ and when we pop down $f(x_1, \dots, x_{i-1}, \Omega, x_{i+1}, \dots, x_m)$ from X then indeed f is the last letter of $S(X)$ and Ω labels the i -th child of f . We define a new substitution S' :

- The values on old variables do not change, i.e. $S'(x) = S(x)$ for each variable x present in the context equation both before and after GenPop .
- For a context variable X from which we did not pop a letter we set $S'(X) = S(X)$.
- For X from which GenPop popped down $f(x_1, \dots, x_{i-1}, \Omega, x_{i+1}, \dots, x_m)$ let $S(X) = sf(t_1, \dots, t_{i-1}, \Omega, t_{i+1}, \dots, t_m)$ (such a representation is possible as GenPop guesses according to S). Then we define $S'(X) = s$ and $S'(x_j) = t_j$ for $j = 1, \dots, i-1, i+1, \dots, m$. Note that when $s = \Omega$ then X is removed from the equation.
- For x that popped-up a constant we do not need to define $S(x)$ as it is no longer in the context equation.

It is easy to verify that indeed in each case the defined S' is a solution of the obtained equation $u' = v'$ and $S'(u') = S(u)$, as claimed.

So suppose that there is a crossing father-leaf pair (f, a) with $f \in \Gamma_{\geq 1}$ and $a \in \Gamma_0$ with respect to S' , i.e. one of the (CFL1)–(CFL3) holds. Note that in (CFL1) and (CFL3) there is a variable y such that $S'(y) \in \Gamma_0$, however, by our assumption that GenPop always makes the choice according to the S each such variable y was replaced with $S(y)$ in the context equation in line 3 or line 14. So it remains to consider the (CFL2).

So let X be as in (CFL2), i.e. the last letter of $S'(X)$ is $f \in \Gamma_{\geq 1}$ and Xa is a subpattern in $u = v$ for some $a \in \Gamma_0$. Consider, whether X popped down a letter:

X popped a letter down: Then for each occurrence of subpattern Xt in the context equation, the first letter of t is always some $g \in \Gamma_{\geq 1}$ (as there was no way to change this), since Γ_0 and $\Gamma_{\geq 1}$ are disjoint, this is a contradiction with the assumption that Xa is a subpattern in the equation for some $a \in \Gamma_0$.

X did not pop a letter down: Consider the occurrence of a subpattern Xa . This a letter was there when we decided not to pop down a letter from X in line 6. Then $\text{Pop}(\Gamma_{\geq 1}, \Gamma_0, 'u = v')$ should have popped the last letter of f from X , as in line 6 we were supposed to guess according to S , contradiction. \square

6. MAIN ALGORITHM

Now we are ready to describe the whole algorithm for testing the satisfiability of context equations. It works in *phases*, each of which is divided into two subphases. In each subphase we first perform the chain compression, the Γ_1, Γ_2 compression for appropriate partition Γ_1, Γ_2 and lastly the leaf compression. In order to make the chain compression we first uncross all chains, similarly in order to perform the Γ_1, Γ_2 compression we ensure that Γ_1, Γ_2 is a non-crossing partition and in order to make the leaf compression we make sure that there is no crossing father-leaf pair.

The reason to have two subphases is quite simple: (for appropriate guess of partition) the first subphase ensures that the size of the (size-minimal) solution decreases by a constant factor (cf. Theorem 1), the second phase is used to make sure that the size of the equation is bounded (in some sense the second phase decreases the size of the equation, but as the equation grows in the first subphase, in total we can only guarantee that the equation is of more or less the same size).

Algorithm 11 ContextEqSat($'u = v', \Sigma$) Checking the satisfiability of a context equation $u = v$ over signature Σ

```

1: let  $k \leftarrow$  maximal arity of functions from  $\Sigma$ 
2: while  $|u| > 1$  or  $|v| > 1$  do
3:   for  $i \leftarrow 1..2$  do     $\triangleright$  One iteration to shorten the solution, one to shorten the equation
4:      $\Gamma_1 \leftarrow$  unary letters in  $u = v$                                  $\triangleright$  By Lemma 7
5:     CutPrefSuff( $\Gamma_1, 'u = v'$ )                                 $\triangleright$  No letter has a crossing block
6:     ChainComp( $\Gamma_1, 'u = v'$ )                                 $\triangleright$  Chain compression
7:      $\Gamma \leftarrow$  the set of unary in  $u = v$                                  $\triangleright$  By Lemma 7
8:     guess partition of  $\Gamma$  into  $\Gamma_1$  and  $\Gamma_2$ 
9:     Pop( $\Gamma_1, \Gamma_2, 'u = v'$ )                                 $\triangleright \Gamma_1, \Gamma_2$  is a non-crossing partition
10:    PartitionComp( $\Gamma_1, \Gamma_2, 'u = v'$ )                                 $\triangleright \Gamma_1, \Gamma_2$  compression
11:     $\Gamma_{\geq 1} \leftarrow$  non-constants in  $'u = v'$  plus one fresh letter  $f_i$  of arity  $i$  for each  $1 < i \leq k$ 
                                                     $\triangleright$  By Lemma 7
12:     $\Gamma_0 \leftarrow$  constants in  $'u = v'$  plus one fresh constant  $c$                                  $\triangleright$  By Lemma 7
13:    GenPop( $\Gamma_{\geq 1}, \Gamma_0, 'u = v'$ )                                 $\triangleright$  No crossing father-leaf pairs
14:    LeafComp( $\Gamma_{\geq 1}, \Gamma_0, 'u = v'$ )
15: Solve the problem naively                                 $\triangleright$  With sides of size 1, the problem is trivial

```

The properties of ContextEqSat are summarised in the following lemma

Theorem 2. ContextEqSat stores equation of length $\mathcal{O}(nk)$ and uses additional $\mathcal{O}(n^2k^2)$ memory, where n is the size of the input equation while k is the maximal arity of symbols from Σ . It non-deterministically solves context equation, in the sense that:

- if the input equation is not-satisfiable then it returns ‘NO’;
- if the input equation is satisfiable then for some nondeterministic choices in $\mathcal{O}(\log N)$ phases it returns ‘YES’, where N is the size of size-minimal solution.

As a corollary we get an upper bound on the computational complexity of context unification.

Corollary 1. Context unification is in PSPACE.

Proof. By Theorem 2 the (non-deterministic) algorithm ContextEqSat works in space $\mathcal{O}(n^2k^2)$, which is polynomial in the input size. By Savitch Theorem the non-deterministic polynomial space algorithm can be determinised, using at most quadratically more space. \square

6.1. Analysis. The actual statement needed to show Theorem 2 is given in the below technical lemma.

Lemma 15. ContextEqSat is sound.

It is complete, to be more precise for some nondeterministic choices the following conditions are satisfied:

- (1) the stored context equation has size $\mathcal{O}(nk)$, with at most n context variables and kn variables;
- (2) if N is the size of the size-minimal solution at the beginning of the phase then at the end of the phase the equation has a solution of size at most $\frac{3N}{4}$;
- (3) the additional memory usage is at most $\mathcal{O}(k^2n^2)$ (counted in bits);
- (4) the maximal arity of symbols in Σ does not increase during ContextEqSat.

The rest of this subsection is devoted to the proof of Lemma 15.

6.1.1. Number of phases. We show that the number of phases is logarithmic in N : we show that one subphase of ContextEqSat in some sense can simulate an action of TreeComp on a size-minimal solution of an equation. Thus, by Theorem 1 the size of the length-minimal solution drops by a constant in a phase. Due to the presence of letters that are not in the equation, we cannot guarantee that this solution prevails the compression steps, however, the size of the length-minimal solution does drop by a constant factor.

Lemma 16. *Let the size-minimal solution of $u = v$ has size N . Then for appropriate non-deterministic choices after first subphase of **ContextEqSat** the obtained equation $u' = v'$ has a solution S' of size $N' \leq \frac{3N}{4}$.*

Proof. Consider some size-minimal solution S_{\min} of size N and let Γ_1 be the set of unary letters in $u = v$. By Lemma 7 there is another solution S of the same size N that uses only unary letters from Γ_1 : by the first part of Lemma 7 we can find a solution of the same size with at most one unary letter not used in $u = v$ and then the second part of the Lemma guarantees that we can make the solution even smaller by deleting all occurrences of this letter (which contradicts the size-minimality of the solution).

By Lemma 13 for appropriate non-deterministic choices after the **CutPrefSuff**($\Gamma_1, 'u = v'$) new equation $u_1 = v_1$ has a solution S_1 such that $S_1(u_1) = S(u)$ and there are no crossing chains for $a \in \Gamma_1$ with respect to S_1 . Then by Lemma 10 after the **ChainComp**($\Gamma_1, 'u_1 = v_1'$) the obtained equation $u_2 = v_2$ has a solution S_2 such that $S_2(u_2) = \text{TreeChainComp}(\Gamma_1, S_1(u_1))$. Note that clearly $|S_2(u_2)| \leq N$.

Consider S_2 . In a similar way as for S_{\min} we can show using Lemma 7 that there is a solution S'_2 such that $S'_2(u_2) \leq S_2(u_2)$ such that S'_2 uses only unary letters that are used in $u_2 = v_2$. By Lemma 2 there is some partition of unary letters used in $'u_2 = v_2'$ into Γ_1 and Γ_2 such that the Γ_1, Γ_2 compression followed by the leaf-compression results in a tree of size at most $\frac{3}{4}|S'_2(u_2)|$, which is at most $\frac{3}{4}N$; fix this partition Γ_1, Γ_2 for the remainder of the proof.

We perform **Pop**($\Gamma_1, \Gamma_2, 'u_2 = v_2'$), by Lemma 12 for appropriate non-deterministic choices the returned equation $u_3 = v_3$ has a solution S_3 such that $S_3(u_3) = S'_2(u_2)$ and Γ_1, Γ_2 is a non-crossing partition with respect to S_3 .

We apply **PartitionComp**($\Gamma_1, \Gamma_2, 'u_3 = v_3'$), since the partition Γ_1, Γ_2 is non-crossing for $u_3 = v_3$ with respect to S_3 , by Lemma 9 the obtained equation $u_4 = v_4$ has a solution S_4 such that $S_4(u_4) = \text{TreePartitionComp}(\Gamma_1, \Gamma_2, S_3(u_3))$.

Finally, consider the solution S_4 of $u_4 = v_4$. By Lemma 7 there is a solution S'_4 that is a simpler equivalent (in particular, $S'_4(u_4)$ has the same number of constants as $S_4(u_4)$) and uses only one letter per arity that is not used by $'u_4 = v_4'$. Let $\Gamma'_{\geq 1}$ denote the set of letters of arity greater than 1 used in $S'_4(u_4)$ and $\Gamma_{\geq 1}$ in $S_4(u_4)$ while Γ'_0 be the set of constants used in $S'_4(u_4)$ and Γ_0 in $S_4(u_4)$. Observe that $\text{TreeLeafComp}(\Gamma_{\geq 1}, \Gamma_0, S_4(u_4))$ and $\text{TreeLeafComp}(\Gamma'_{\geq 1}, \Gamma'_0, S'_4(u_4))$ have the same size, as S'_4 is a simpler equivalent of S_4 implies that $S'_4(u_4)$ has the same number of constants as $S_4(u_4)$ and **TreeLeafComp** on both of them simply compresses all leaves to their respective fathers. Hence, by Lemma 2, $\text{TreeLeafComp}(\Gamma'_{\geq 1}, \Gamma'_0, S'_4(u_4))$ has size at most $\frac{3}{4}|S'_2(u_2)| \leq \frac{3}{4}|S(u)| = \frac{3}{4}N$.

So it is left to show that we can simulate $\text{TreeLeafComp}(\Gamma'_{\geq 1}, \Gamma'_0, S'_4(u_4))$ on the equation. So consider $S'_4, \Gamma'_{\geq 1}$ and Γ'_0 . By Lemma 14 for appropriate non-deterministic choices after **GenPop**($\Gamma'_{\geq 1}, \Gamma'_0, 'u_4 = v_4'$) the obtained equation $u_5 = v_5$ has a solution S_5 such that $S_5(u_5) = S'_4(u_4)$ and there is no crossing father-leaf pair (f, a) with $f \in \Gamma'_1$ and $a \in \Gamma'_0$ with respect to S_5 . We now apply **LeafComp**($\Gamma'_{\geq 1}, \Gamma'_0, 'u_5 = v_5'$). By Lemma 11 for appropriate non-deterministic choices the returned equation $u_6 = v_6$ has a solution S_6 such that $S_6(u_6) = \text{TreeLeafComp}(\Gamma'_{\geq 1}, \Gamma'_0, S_5(u_5)) = \text{TreeLeafComp}(\Gamma'_{\geq 1}, \Gamma'_0, S'_4(u_4))$. In particular, this solution is as small as promised in the lemma. \square

6.1.2. Space consumption. Observe that, in contrast to the recompression-based algorithm for word equations, **ContextEqSat** introduces new variables and their occurrences to the equation (when **GenPop** pops down a letter of arity greater than 1). At first it seems like a large problem, as the number of letters introduced to the equation in one phase depends on the number of variables, however, we are able of bounding the number of such variables at any given time of the **ContextEqSat** by kn . To this end, we need some definitions: we say that a variable x_i is *owned* by a context variable X if x_i occurred in the equation when X popped a letter down. A particular occurrence of x_i in the equation is *owned* by the occurrence of the context variable that introduced it. When a context variable X is removed from the equation the variables it owns get *disowned* (and particular occurrences of this variable are also disowned).

We show that each context variable owns at most $k - 1$ variables.

Lemma 17. *Every context variable X present in $u = v$ owns at most $k - 1$ variables. In particular, there are at most kn occurrences of variables in $u = v$.*

Note that the upper bound on the number of variables *does not* depend on the non-deterministic choices of **ContextEqSat**.

Proof. Given an occurrence of a subterm Xt we say that this occurrence of X dominates the occurrences of variables in t .

We show by induction two technical claims:

- (1) For every occurrence of a variable X the multiset of variables, whose occurrences it owns, is the same.
- (2) Each appearance of X dominates its owned occurrences of variables.

The subclaim 1 is trivial: at the beginning, there are no owned variables. When we introduce new X -owned variables, we replace each X with the same $Xf(x_1, \dots, x_{i-1}, \Omega, x_{i+1}, \dots, x_m)$, in particular the set of X -owned variables for each occurrence of X is increased by $\{x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_m\}$. When we remove occurrences of x , we remove them all at the same time. Which ends the induction.

Concerning the subclaim 2, this vacuously holds for the input instance, which yields the induction base. For the induction step, consider now the operation performed by **ContextEqSat** on the context equation. Any compression is performed only on letters, so it cannot affect the domination. When we pop the letters from a variable x , we replace x with ax (or remove x altogether), so this also does not affect the domination. Similarly, when we pop letters from context variables, we either replace X with aX or X with $Xf(x_1, \dots, x_{i-1}, \Omega, x_{i+1}, \dots, x_m)$, in both cases the domination of the old variables is not affected and in the last case the new variables x_1, \dots, x_m owned by this particular occurrence of X are indeed dominated by this occurrence of X .

Using those two subclaims we now show that if during **GenPop** X pops down a letter, then X does not own any variables. Suppose that X pops down a letter. Then in $u = v$ there is a subtree Xc for $c \in \Gamma_0$. Suppose that X owned a variable x before popping down the letter. Then by subclaim 1 the occurrence which is applied on c also owns occurrence of x and by 2 this occurrence is dominated by its owning occurrence of X , which is not possible, as this owning occurrence of X is part of the term Xc . As a consequence, each occurrence of a context variable owns at most $k - 1$ occurrences of variables.

Now, concerning the number of variables: initially there are at most (not owned nor disowned) n variables occurrences and n context variables occurrences. Suppose that at some point there are $m \leq n$ context variables occurrences. Since we never introduce context variables, there are at most $m(k - 1)$ owned variables' occurrences, and at most $(n - m)(k - 1)$ disowned ones and n that are neither owned, nor disowned (those are the occurrences of variables that were present in the input equation), so nk occurrences of variables in total, as claimed. \square

We move to the crucial part of the proof: the space consumption of **ContextEqSat**. The intuition should be clear: in the second subphase we treat the equation as a term and try to ensure that its size drops by one fourth, just as in the case of Theorem 1. However, in the meantime we also increased the size of the equation, as we pop the letters into the context equation (in both subphases). The number of those letters depends linearly on the number of occurrences of variables and context variables in $u = v$, which is known to be $\mathcal{O}(kn)$, see Lemma 17. Hence those two effects (increasing the size and reducing the size) cancel each out and it can be shown that the size of the equation is $\mathcal{O}(kn)$.

Lemma 18. *For appropriate non-deterministic choices in second subphase the context equation at the end of a phase of **ContextEqSat** has size $\mathcal{O}(nk)$. Furthermore, for those choices **ContextEqSat** is complete, to be more precise: If $u = v$ after the first subphase had a solution S of size N then after the second subphase the obtained equation $u' = v'$ has a solution of size at most N .*

Proof. We show some nondeterministic choices for which the run of **ContextEqSat** in the second subphase satisfies the claim of the lemma.

In the following we consider only the number of letters in $u = v$: since no new context variables are introduced, there are at most n such occurrences and by Lemma 17 there are at most kn occurrences of variables in $u = v$.

Consider, how many new letters were introduced during the first subphase into the equation.

- The chains introduced by **CutPrefSuff** are immediately replaced with a single letter, therefore we can think that **CutPrefSuff** introduces 2 letters per context variable and 1 per variable, so at most $2n + kn$ in total. (Note that popping a letter down introduces also variables, but those are counted separately).
- Similarly, each **Pop** introduces at most 2 letters per context variables and 1 per variable, so also at most $2n + kn$ in total.
- Lastly, the popping down in **GenPop** introduces 1 letter per context variable, so n letters while the popping up introduces at most 1 letter per variable, but all those letter are compressed into their parents immediately afterwards (in **LeafComp**: all letters popped up are from Γ_0 and by choice of $\Gamma_{\geq 1}$ their fathers in $S(U)$ or $S(v)$ are from Γ_1 , so they are compressed. Thus we do not need to count them.

Hence, in total, during the first subphase the size of the equation increases by at most $5n + 2kn$ letters.

Concerning the second subphase, the following analysis is similar to the one in Lemma 2 but it takes into the account also the letters introduced due to popping. Furthermore, we need to also guarantee that the equation stays satisfiable and the the size of the size-minimal solution does not increase.

As in Lemma 2, let n_0 , n_1 and $n_{\geq 2}$ denote the number of letters of arity 0, 1 and at least 2, respectively in the equation $u = v$ between the first and second subphase; n'_0 , n'_1 and $n'_{\geq 2}$ the number of letters of arity 0, 1 and at least 2 in $u = v$ after the chain compression, n''_0 , n''_1 and $n''_{\geq 2}$ after the (appropriate) Γ_1 , Γ_2 compression and finally n'''_0 , n'''_1 and $n'''_{\geq 2}$ after the leaf compression. We shall show that

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

where f is some function linear in n and k . Taking into the account that during the first subphase the size of the equation increased by at most $5n + 2kn$, we obtain that the equation at the end of the phase is of size at most (let m be the size of the equation at the beginning of the phase)

$$\frac{3}{4}m + \frac{15}{4}n + \frac{6}{4}kn + f(n, k) .$$

From which by an easy induction it follows that

$$m \leq 15n + 6kn + 4f(n, k) .$$

So it is left to show that estimation (4) indeed holds.

Let $u = v$ be a satisfiable equation after the first subphase, let S be one of its size-minimal equation (note that in this proof in general we do not need to worry about the letters that are not present in the equation, as we focus on compressing letters in the equation). Let Γ_1 be the set of letters present in the equation $u = v$. After the **CutPrefSuff**($\Gamma_1, 'u = v'$) the obtained equation $u_1 = v_1$ has a solution S_1 such that $S_1(u_1) = S(u)$. In particular the size-minimal solution of $u_1 = v_1$ is not larger than the one of $u = v$.

Note that each chain popped from a context variable or variable by **CutPrefSuff**($\Gamma_1, 'u = v'$) is immediately replaced with a single letter during the **ChainComp**($\Gamma_1, 'u_1 = v_1'$), there are at most $2n + kn$ new unary) letters introduced to $u = v$, hence

$$(5a) \quad n'_0 = n_0 \quad n'_1 \leq n_1 + 2n + kn \quad n'_{\geq 2} = n_{\geq 2} .$$

Furthermore, by Lemma 10 the obtained equation $u_2 = v_2$ has a solution S_2 such that $S_2(u_2) = \text{TreeChainComp}(\Gamma_1, S'_1(u_1))$, in particular $|S_2(u_2)| \leq |\text{TreeChainComp}(\Gamma_1, S'_1(u_1))| \leq N$.

Now consider the maximal chains in $u = v$ that are formed only by letters from Σ , i.e. a context variable denotes the beginning or the end of such a chain, let c denotes the number of such chains. Then

$$(5b) \quad c \leq \frac{n_0}{2} + n_{\geq 2} + n + \frac{kn}{2} + \frac{1}{2} .$$

Indeed: consider any such chain and the bottom unary letter in it. Then its child is labelled with either a letter, a context variable or a variable. Similarly, consider the topmost letter in any such chain, then the father is labelled with either a letter of arity at least 2 or by a context variable or nothing at all, when this node is a root. (Note that without loss of generality we may assume that in $u = v$ at most one root is labelled with a unary letter: if both $u = au'$ and $v = av'$ then we can simply replace $u = v$ with ' $u' = v'$ ', if their first letters are different then the equation is trivially not satisfiable.) Summing up those two estimations we get that

$$\begin{aligned} 2c &\leq \underbrace{n'_0 + n'_{\geq 2} + kn + n}_{\text{nodes below}} + \underbrace{n'_{\geq 2} + n}_{\text{nodes above}} + \underbrace{1}_{\text{possible root}} \\ &\leq n_0 + 2n_{\geq 2} + 2n + kn + 1 , \end{aligned}$$

which yields (5b).

Let Γ denote the set of unary letters in $u = v$. Then there is a partition of Γ into Γ_1 and Γ_2 such that at least $\frac{n'_1 - c}{4}$ pairs in $u = v$ are covered by a partition Γ_1, Γ_2 : this follows by a randomised argument similar to the one in Claim 1. Fix this partition for the remainder of the proof.

We first perform the $\text{Pop}(\Gamma_1, \Gamma_2, 'u_2 = v_2')$ (obtaining $u_3 = v_3$) and then $\text{PartitionComp}(\Gamma_1, \Gamma_2, 'u_3 = v_3')$. The former operation introduces at most $2n + kn$ unary letters to the equation, while the latter compresses at least $\frac{n'_1 - c}{4}$ unary letters. Hence

$$(5c) \quad n''_0 = n'_1 = n_0 \quad n''_1 \leq \frac{3}{4}n'_1 + \frac{c}{4} + 2n + kn \quad n''_{\geq 2} = n'_{\geq 2} = n_{\geq 2} .$$

Let us elaborate on the estimation for n''_1 :

$$\begin{aligned} n''_1 &\leq \frac{3}{4}n'_1 + \frac{c}{4} + 2n + kn && \text{from (5c)} \\ &\leq \underbrace{\frac{3n_1}{4} + \frac{3n}{2} + \frac{3kn}{4}}_{3/4n'_1} + \underbrace{\frac{n_0}{8} + \frac{n_{\geq 2}}{4} + \frac{n}{4} + \frac{kn}{8} + \frac{1}{8}}_{c/4} + 2n + kn && \text{from (5a) and (5b)} \\ (5d) \quad &= \frac{n_0}{8} + \frac{3n_1}{4} + \frac{n_{\geq 2}}{4} + \frac{15n}{4} + \frac{15kn}{8} + \frac{1}{8} && \text{simplification} . \end{aligned}$$

Observe that after the $\text{Pop}(\Gamma_1, \Gamma_2, 'u_2 = v_2')$, by Lemma 12 the obtained equation $u_3 = v_3$ has a solution S_3 such that $S_3(u_3) = S_2(u_2)$ (so also $|S_3(u_3)| \leq N$) and Γ_1, Γ_2 is a non-crossing partition with respect to S_3 . Then by Lemma 9 the following $\text{PartitionComp}(\Gamma_1, \Gamma_2, 'u_3 = v_3')$ returns an equation $u_4 = v_4$ which has a solution S_4 such that $S_4(u_4) = \text{TreePartitionComp}(\Gamma_1, \Gamma_2, S_3(u_3))$, in particular $|S_4(u_4)| \leq N$. By Lemma 7, there is also a solution S'_4 of $u_4 = v_4$ that uses at most one letter of each arity that is not in $u_4 = v_4$.

Now, lastly, during the leaf compression we first pop up letters (i.e. we replace some variables by constants) then we pop letters down, introducing one letter per context variables, so at most n letters, that are of arity at least 1 and then again pop up. As S'_4 uses at most one letter of each arity that is not in $u_4 = v_4$ we may assume that ContextEqSat guesses those letters into $\Gamma_{\geq 1}$ and Γ_0 . Hence the letters that are popped-up (i.e. they replace some variables) are from Γ_0 and are immediately compressed to their fathers (who are from Γ_1) during the leaf compression, so we may ignore the letters that are popped up for the purpose of our estimation. On the other

hand, each leaf labelled with a letter is also compressed, i.e. $n''_{\geq 2}$ letters are compressed. Hence

$$\begin{aligned}
& n_0''' + n_1''' + n_{\geq 2}''' \\
& \leq n_0'' + n_1'' + n_{\geq 2}'' + n - n_0'' && \text{popped up and absorbed letters} \\
& = n_1'' + n_{\geq 2}'' + n && \text{simplification} \\
& \leq \underbrace{\frac{n_0}{8} + \frac{3n_1}{4} + \frac{n_{\geq 2}}{4} + \frac{15n}{4} + \frac{15kn}{8} + \frac{1}{8}}_{n_1''} + n_{\geq 2} + n && \text{from (5d)} \\
& = \frac{n_0}{8} + \frac{3n_1}{4} + \frac{5n_{\geq 2}}{4} + \frac{19n}{4} + \frac{15kn}{8} + \frac{1}{8} && \text{simplification} \\
& \leq \frac{n_0}{8} + \frac{3n_1}{4} + \frac{3n_{\geq 2}}{4} + \frac{1}{2} \underbrace{(n_0 + kn - 1)}_{\geq n_{\geq 2}} + \frac{19n}{4} + \frac{15kn}{8} + \frac{1}{8} && \text{from (5a)} \\
& \qquad \qquad \qquad \underbrace{\qquad \qquad \qquad}_{\geq \frac{5}{4}n_{\geq 2}} \\
& < \frac{3}{4}(n_0 + n_1 + \geq 2) + \frac{19n}{4} + \frac{19kn}{8} && \text{simplification} .
\end{aligned}$$

Which shows (4) for $f(n, k) = \frac{19n}{4} + \frac{19kn}{8}$ and so ends the proof. Concerning the satisfiability, by Lemma 14 after the $\text{GenPop}(\Gamma_{\geq 1}, \Gamma_0, 'u_4 = v_4')$ the obtained equation $u_5 = v_5$ has a solution S_5 such that $S_5(u_5) = S_4'(u_4)$, hence also $|S_5(u_5)| \leq N$, and there is no crossing father-leaf pair (f, a) with $f \in \Gamma_{\geq 1}$ and $a \in \Gamma_0$. Then, by Lemma 11, the following $\text{LeafComp}(\Gamma_{\geq 1}, \Gamma_0, 'u_5 = v_5')$ returns an equation $u_6 = v_6$ with a solution S_6 such that $S_6(u_6) = \text{TreeLeafComp}(\Gamma_{\geq 1}, \Gamma_0, S_5(u_5))$, hence $u_6 = v_6$ is satisfiable and it has a solution of size at most N . \square

Now showing Lemma 15 follows naturally.

proof of Lemma 15. • The bound on number of occurrences of variables follows from Lemma 17.

No context variables are introduced, so there are at most n occurrences of context-variables. The bound on the size of the equation follows from Lemma 18.

- The bound on the size of the size-minimal solution after one phase follows from Lemma 16 and Lemma 18: by the former it is reduced by a factor of $1/4$ during the first subphase and by the latter the size of the size-minimal solution does not increase during the second subphase.

Concerning the additional memory usage: storing an equation of length $\mathcal{O}(nk)$ uses $\mathcal{O}(nk \log(nk))$ bits. Additionally, we need to store the lengths of the popped chains of letters (we store a^ℓ as a pair (a, ℓ)). Without loss of generality we can focus on size-minimal solutions, for whose those lengths are of size $2^{c(|u|+|v|)}$ for some constant c , by Lemma 5, so each can be encoded using $\mathcal{O}(|u| + |v|) = \mathcal{O}(kn)$ bits; there are at most n context variables and kn variables (by Lemma 17), so there are $\mathcal{O}(kn)$ such prefixes and suffixes, so in total we need $\mathcal{O}(k^2n^2)$ bits to denote them. All other operations increase the space usage by a constant factor only.

- The bound on the arity of letters is an easy observation, similar to the one in Lemma 3: no operation introduces letters of arity greater than the letters already in the context equation. \square

Acknowledgements. I would like to thank Jan Otop and Manfred Schmidt-Schauß for introducing me to the topic and for the question whether recompression generalises to the context unification.

REFERENCES

- [1] Hubert Comon. Completion of rewrite systems with membership constraints. Part I: Deduction rules. *J. Symb. Comput.*, 25(4):397–419, 1998.
- [2] Hubert Comon. Completion of rewrite systems with membership constraints. Part II: Constraint solving. *J. Symb. Comput.*, 25(4):421–453, 1998.
- [3] Volker Diekert, Claudio Gutiérrez, and Christian Hagenah. The existential theory of equations with rational constraints in free groups is pspace-complete. *Inf. Comput.*, 202(2):105–140, 2005.

- [4] William M. Farmer. Simple second-order languages for which unification is undecidable. *Theor. Comput. Sci.*, 87(1):25–41, 1991.
- [5] Adria Gascón, Guillem Godoy, Manfred Schmidt-Schauß, and Ashish Tiwari. Context unification with one context variable. *J. Symb. Comput.*, 45(2):173–193, 2010.
- [6] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theor. Comput. Sci.*, 13:225–230, 1981.
- [7] Artur Jež. 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. full version available at <http://arxiv.org/abs/1111.3244>.
- [8] Artur Jež. 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 available at <http://arxiv.org/abs/1301.5842>.
- [9] Artur Jež. The complexity of compressed membership problems for finite automata. *Theory of Computing Systems*, 2013. accepted and available online <http://dx.doi.org/10.1007/s00224-013-9443-6>.
- [10] Artur Jež. One-variable word equations in linear time. In Fedor V. Fomin, Rusins Freivalds, Marta Kwiatkowska, and David Peleg, editors, *ICALP (2)*, volume 7966, pages 324–335, 2013. full version at <http://arxiv.org/abs/1302.3481>.
- [11] Artur Jež. 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. full version available at <http://arxiv.org/abs/1203.3705>.
- [12] Artur Jež and Markus Lohrey. Approximation of smallest linear tree grammar. *CoRR*, 1309.4958, 2013. submitted.
- [13] Antoni Kościelski and Leszek Pacholski. Complexity of Makanin’s algorithm. *J. ACM*, 43(4):670–684, 1996.
- [14] Jordi Levy. Linear second-order unification. In Harald Ganzinger, editor, *RTA*, volume 1103 of *LNCS*, pages 332–346. Springer, 1996.
- [15] Jordi Levy and Jaume Agustí-Cullell. Bi-rewrite systems. *J. Symb. Comput.*, 22(3):279–314, 1996.
- [16] Jordi Levy, Manfred Schmidt-Schauß, and Mateu Villaret. On the complexity of bounded second-order unification and stratified context unification. *Logic Journal of the IGPL*, 19(6):763–789, 2011.
- [17] Jordi Levy and Margus Veanes. On the undecidability of second-order unification. *Inf. Comput.*, 159(1–2):125–150, 2000.
- [18] Jordi Levy and Mateu Villaret. Linear second-order unification and context unification with tree-regular constraints. In Leo Bachmair, editor, *RTA*, volume 1833 of *LNCS*, pages 156–171. Springer, 2000.
- [19] Jordi Levy and Mateu Villaret. Currying second-order unification problems. In Sophie Tison, editor, *RTA*, volume 2378 of *LNCS*, pages 326–339. Springer, 2002.
- [20] G. S. Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 2(103):147–236, 1977. (in Russian).
- [21] Jerzy Marcinkowski. Undecidability of the first order theory of one-step right ground rewriting. In Hubert Comon, editor, *RTA*, volume 1232 of *LNCS*, pages 241–253. Springer, 1997.
- [22] Joachim Niehren, Manfred Pinkal, and Peter Ruhrberg. On equality up-to constraints over finite trees, context unification, and one-step rewriting. In William McCune, editor, *CADE*, volume 1249 of *LNCS*, pages 34–48. Springer, 1997.
- [23] Joachim Niehren, Manfred Pinkal, and Peter Ruhrberg. A uniform approach to underspecification and parallelism. In Philip R. Cohen and Wolfgang Wahlster, editors, *ACL*, pages 410–417. Morgan Kaufmann Publishers / ACL, 1997.
- [24] Wojciech Plandowski. Satisfiability of word equations with constants is in NEXPTIME. In *STOC*, pages 721–725, 1999.
- [25] Wojciech Plandowski and Wojciech Rytter. Application of Lempel-Ziv encodings to the solution of word equations. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *LNCS*, pages 731–742. Springer, 1998.
- [26] RTA problem list. Problem 90. <http://rtaloop.mancoosi.univ-paris-diderot.fr/problems/90.html>.
- [27] Manfred Schmidt-Schauß. Unification of stratified second-order terms. Internal Report 12/94, Johann-Wolfgang-Goethe-Universitt, 1994.
- [28] Manfred Schmidt-Schauß. A decision algorithm for distributive unification. *Theor. Comput. Sci.*, 208(1–2):111–148, 1998.
- [29] Manfred Schmidt-Schauß. A decision algorithm for stratified context unification. *J. Log. Comput.*, 12(6):929–953, 2002.
- [30] Manfred Schmidt-Schauß. Decidability of bounded second order unification. *Inf. Comput.*, 188(2):143–178, 2004.
- [31] Manfred Schmidt-Schauß and Klaus U. Schulz. On the exponent of periodicity of minimal solutions of context equation. In *RTA*, volume 1379 of *LNCS*, pages 61–75. Springer, 1998.
- [32] Manfred Schmidt-Schauß and Klaus U. Schulz. Solvability of context equations with two context variables is decidable. *J. Symb. Comput.*, 33(1):77–122, 2002.

- [33] Manfred Schmidt-Schauß and Klaus U. Schulz. Decidability of bounded higher-order unification. *J. Symb. Comput.*, 40(2):905–954, 2005.
- [34] Klaus U. Schulz. Makanin’s algorithm for word equations—two improvements and a generalization. In Klaus U. Schulz, editor, *IWWERT*, volume 572 of *LNCS*, pages 85–150. Springer, 1990.
- [35] Ralf Treinen. The first-order theory of linear one-step rewriting is undecidable. *Theor. Comput. Sci.*, 208(1–2):179–190, 1998.
- [36] Sergei G. Vorobyov. The first-order theory of one step rewriting in linear Noetherian systems is undecidable. In Hubert Comon, editor, *RTA*, volume 1232 of *LNCS*, pages 254–268. Springer, 1997.
- [37] Sergei G. Vorobyov. $\forall\exists^*$ -equational theory of context unification is Π_1^0 -hard. In Lubos Brim, Jozef Gruska, and Jirí Zlatuska, editors, *MFCS*, volume 1450 of *LNCS*, pages 597–606. Springer, 1998.

MAX PLANCK INSTITUTE FÜR INFORMATIK,, CAMPUS E1 4, DE-66123 SAARBRÜCKEN, GERMANY, AND
INSTITUTE OF COMPUTER SCIENCE, UNIVERSITY OF WROCLAW, UL. JOLIOT-CURIE 15, 50-383 WROCLAW,
POLAND, `AJE@CS.UNI.WROC.PL`