Deciding Context Unification (with Regular Constraints)

Artur Jeż(✉)

University of Wrocław, Wrocław, Poland
aje@cs.uni.wroc.pl

Abstract. Given a ranked alphabet, context are terms with a single occurrence of a special symbol • (outside of the alphabet), which represents a missing subterm. One can naturally build equations over contexts: the context variables are treated as symbols of arity one and a substitution S assigns to each such a variable a context $S(X)$. A substitution S is extended to terms with context variables in a natural way: $S(X(t))$ is a context $S(X)$ in which the unique occurrence of • is replaced with $S(t)$. For historical reasons, the satisfiability of context equations is usually referred to as *context unification.* in PSPACE

Context unification generalizes word equations and first-order term unification (which are decidable) and is subsumed by second order unification (which is undecidable) and its decidability status remained open for almost two decades. In this paper I will sketch a PSPACE algorithm for this problem. The idea is to apply simple compression rules (replacing pairs of neighbouring function symbols) to the solution of the context equation; to this end we appropriately modify the equation (without the knowledge of the actual solution) so that compressing the solution can be simulated by compressing parts of the equation. When the compression operations are appropriately chosen, then the size of the instance is polynomial during the whole algorithm, thus giving a PSPACE-upper bound. The best known lower bounds are as for word equations, i.e. NP. The method can be extended to the scenario in which tree-regular constraints for the variables are present, in which case the problem is EXPTIME-complete.

1 Introduction

Context unification is a generalization of word equations to terms. In the word equations problem we are given an alphabet Σ and a set of variables $\mathcal{X}$. Then, given an equation of the form $u = v$, where both u, v are words over the letters and variables, we ask about the existence of a substitution of variables by words over the alphabet that turns this formal equation into a true equality (of words over the alphabet). The first algorithm for this problem was given by Makanin [18]; the currently best algorithms for this problem utilize different (and simpler) approach and work in PSPACE [11,22], the best known lower bound is NP, and it follows easily from, say, the NP-hardness of integer programming.

We view terms as rooted, ordered (meaning that the children of a node are ordered using a usual left-to-right order) trees, usually denoted with letters t or s. Nodes are labelled with elements from a ranked alphabet Σ, i.e. each letter $a \in \Sigma$ has a fixed arity $\mathrm{ar}(f)$; those elements are usually called *letters.* A tree is *well-formed* if a node labelled with f has exactly $\mathrm{ar}(f)$ children; we consider only well-formed trees, which can be equivalently seen as *ground terms* over Σ. We will also use term notation to denote the trees in text, for example, $f(c, c')$ denotes the tree with root labelled with f and two children, first (left) labelled with c and the second (right) with c'; those children are leaves.

When generalizing the word equations from words to terms, one first needs to decide, what a variable can represent. If a variable can only represent a well-formed term, then we arrive at a standard first-order unification problem, which can be solved in linear time; so this does not even generalize the word equations. Thus the variables are allowed to take arguments, i.e. they define trees with missing subtrees. Formally, we extend the alphabet with parameter symbols $\bullet_1, \bullet_2, \ldots$ of arity 0. If a term t uses $\bullet_1, \bullet_2, \ldots, \bullet_i$ then $t(t_1, \ldots, t_i)$, where $t_1, \ldots, t_i$ do not use parameters, is t in which $\bullet_j$ is replaced with t_j. Thus the variables are ranked: X takes $\mathrm{ar}(X)$ arguments and the substitution for it has to use $\bullet_1, \bullet_2, \ldots, \bullet_{ar(X)}$. For instance, an equation $f(X(c), X(c)) = X(f(c, c))$ has a solution $X = \bullet$. Under this substitution both sides evaluate to $f(c, c)$. There are other solutions, for instance $X = f(\bullet, \bullet)$, which evaluates both sides to $f(f(c, c), f(c, c))$; in general, solution that evaluates both sides to full binary tree of arbitrary height is easy to construct. When no further restrictions are given, this problem is the second order unification and is undecidable [9].

In context unification we require that each $\bullet_j$ is used exactly once. For instance, the aforementioned equation $f(X(c), X(c)) = X(f(c, c))$ as an instance of context unification has exactly one solution: $X = \bullet$, other solution are excluded by the restriction that $\bullet$ is used exactly once. It is easy to see that the case of many argument NP-reduces to the case with only one argument, and we deal with only this case further on. Context unification was introduced by Comon [1,2], who also coined the name, and independently by Schmidt-Schauß [24]. It found usage in analysis of rewrite systems with membership constraints [1,2], analysis of natural language [20,21], distributive unification [25] and bi-rewriting systems [13].

Context unification is both subsumed by second order unification (which is undecidable) and subsumes word equations (which are decidable). Furthermore, other natural problems between those two usually trivially reduce to word equations or are undecidable. Thus, in a sense, context unification is the only such problem, whose decidability remains open. This is one of the reasons why it gained considerable attention in the term rewriting community [23] and no wonder that there was a large body of work focused on context unification and several partial results were obtained [2,6–8,12,15,17,26,28,29]. Note that in most cases the corresponding variants of the general second order unification remain undecidable, which supported the conjecture that context unification is decidable.

Context unification was shown to be equivalent to 'equality up to constraint' problem [20] (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 [21]. It is known that the first-order theory of one-step rewriting constraints is undecidable [19,32,33]. For whole context unification, already the $\forall\, \exists^8$-equational theory is Π^0_1-hard [34].

Some fragments of second order unification are known to reduce to context unification: the *bounded second order unification* assumes that the number of occurrences 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; cf. monadic second order unification, which can be seen as a similar variant of word equations, and is known to be NP-complete [14]. This fragment on one hand easily reduces to context unification and on the other hand it is known to be decidable [27]; in fact its generalisation to higher-order unification is decidable as well [30] and it is known that bounded second order unification is NP-complete [15].

The context unification can be also extended by allowing some additional constraints on variables, a natural one allows the usage of the *tree-regular constraints*, i.e. for any variable we require that its substitution comes from a certain regular set of trees. It is known that such an extension is equivalent to the linear second order unification [16], defined by Levy [12]: 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 expressive power to encode such more complex bounding. Note, that the reductions are not polynomial and the equivalence is stated only on the decidability level.

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 essentially all known algorithms for word equations problem can be generalised to word equations with regular constraints [3,4,31].

$\rightarrow$ the actual complexity of linear 2nd order unification is open?

Results

The decidability status of context unification remained unknown for almost two decades. In this paper I present a proof that context unification can be solved in PSPACE, using a generalization of an algorithm for word equations; see [10] for a full version.

The idea is to apply simple compression rules (replacing pairs of neighbouring function symbols) to the solution of the context equation; to this end we appropriately modify the equation (without the knowledge of the actual solution) so that compressing the solution can be simulated by compressing parts of the equation. It is shown that if the compression operations are appropriately chosen, then the size of the instance is polynomial during the whole algorithm, thus giving a PSPACE-upper bound. The best known lower bounds are the same for word equations, i.e. context unification is NP-hard. The method can be extended to the scenario in which tree-regular for the variables are present, in which case the problem is EXPTIME-complete.

This idea, known under the name of recompression, was used before for word equations [11], simplifying the existing proof of containment in PSPACE. Furthermore, applications of compression to fragments of context unification were known before [6,8,15,30] and the presented algorithm extends this method to terms in full generality. In this way solving word equations using recompression [11] generalises to solving context unification. This in some sense fulfils the original plan of extending the algorithms for word equations to context unification.

2 Definitions

2.1 Trees

As said before, we are given a ranked alphabet Σ, i.e. there is an arity function $\mathrm{ar} : \Sigma \to \mathbb{N}$, and we deal with rooted (there is a designated root), ordered (there is a fixed linear order on children of each node) Σ-labelled trees. We say that a tree is well-formed when a node labelled with $a \in \Sigma$ has $\mathrm{ar}(a)$ children. We also view such trees as terms, then a tree is well-built if seen as a term it is well-built.

2.2 Patterns

We want to replace fragments of a tree with new nodes, those fragments are not necessarily well-formed. Thus we want to define 'trees with holes', where holes represent missing arguments. Let $\mathbb{Y} = \{\bullet, \bullet_1, \bullet_2, \dots\}$ be an infinite set of symbols of arity 0, we think of each of them as a place of a missing argument; its elements are collectively called *parameters*. A *pattern* is a tree over an alphabet $\Sigma \cup \mathbb{Y}$, such that each parameter occurs at most once in it. The usual convention is that the used parameters are $\bullet_1, \bullet_2, \dots, \bullet_k$, or $\bullet$, when there is only 1 parameter; moreover, we usually assume that the order (according to preorder traversal of the pattern) of occurrences of the parameters in the pattern is $\bullet_1, \bullet_2, \dots, \bullet_k$. We often refer to *parameter nodes* and *non-parameter nodes* to refer to nodes labelled with parameters and non-parameters, respectively. A pattern using r parameters is called r-pattern. A pattern p *occurs* (at a node v) in a tree t if p can be obtained by taking a subtree t' of t rooted at v and replacing some of subtrees of t' by appropriate parameters. This is also called an *occurrence* of p in t. A pattern p is a subpattern of t if p occurs in t.

In a more classic terminology, 1-patterns are also called contexts, hence the name "context unification".

Given a tree t, its r-subpattern p occurrence and an r-pattern p' we can naturally replace p with p': we delete the part of t corresponding to p with removed parameters and plug p' with removed parameters instead and reattach all the subtrees in the same order; as the number of parameters is the same, this is well-defined. We can perform several replacements at the same time, as long as occurrences of replaced patterns do not share non-parameter nodes. In this terminology, our algorithm will replace occurrences of subpatterns of t in t.

We focus on some specific patterns: A *chain* is a pattern that consists only of unary nodes plus one parameter node. Chains that have two nodes that are labelled with different letters, i.e. of the form $a(b(\bullet))$ for $a \neq b$, are called *pairs*; chains whose all unary nodes are labelled with the same letter a, i.e. of the form $a(a(\ldots(a(\bullet)\ldots)))$, are called *$a$-chains*. A chain t' that is a subpattern of t is a *chain subpattern* of t, an occurrence of an a-chain subpattern $a(a(\ldots(a(\bullet)\ldots)))$ is *maximal* if it cannot be extended (in t) by a nor up nor down. A pattern of a form $f(\bullet_1, \bullet_2, \ldots, \bullet_{i-1}, c, \bullet_i, \ldots, \bullet_{\text{ar}(f)-1})$ is denoted as (f, i, c).

We treat chains as strings and write them in the string notation (in particular, we drop the parameters) and 'concatenate' them, that is, for two chains $s_a = a_1(a_2(\ldots a_k(\bullet)\ldots))$ and $s_b = b_1(b_2(\ldots b_\ell(\bullet)\ldots))$ we write them as $s_a = a_1 a_2 \cdots a_k$ and $s_b = b_1 b_2 \cdots b_\ell$ and their concatenation $s_a s_b = a_1 a_2 \cdots a_k b_1 b_2 \cdots b_\ell$ denotes the chain $a_1(a_2(\ldots a_k(b_1(b_2(\ldots b_\ell(\bullet)\ldots)))\ldots))$. In this convention ab denotes a pair and a^ℓ denotes an a-chain. We use those conventions also for 1-patterns and also for 1-patterns followed by a single term, i.e. for 1-patterns $p_1, \ldots, p_k$ and a term t the $p_1 p_2 \cdots p_k t$ denotes the term $p_1(p_2(\ldots p_k(t)\ldots))$.

2.3 Context Unification: Formal Statement

By $\mathcal{V}$ we denote an infinite set of context variables X, Y, Z, ... We also use individual term variables x, y, z, ... taken from $\mathcal{X}$. When we do not want to distinguish between a context variable or term variable, we call it a *variable* and denote it by a small Greek letter, like $\alpha, \beta, \gamma, \ldots$.

Definition 1. *The* terms *over Σ, $\mathcal{X}$, $\mathcal{V}$ are ground terms with alphabet $\Sigma \cup \mathcal{X} \cup \mathcal{V}$ in which we extend the arity function* ar *to $\mathcal{X} \cup \mathcal{V}$ by* $\text{ar}(X) = 1$ *and* $\text{ar}(x) = 0$ *for each $X \in \mathcal{V}$ and $x \in \mathcal{X}$. A* context equation *is an equation of the form $u = v$ where both u and v are terms over $\Sigma \cup \mathcal{X} \cup \mathcal{V}$.*

We are interested in the solutions of the context equations, i.e. substitutions that replace term variables with ground terms and context variables with 1-patterns, such that a formal equality $u = v$ is turned into a valid equality of ground terms. More formally:

Definition 2. *A* substitution *is a mapping S that assigns a 1-pattern $S(X)$ 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 $a \in \Sigma$;*
- *$S(f(t_1, \ldots, t_n)) := f(S(t_1), \ldots, S(t_m))$ for an m-ary $f \in \Sigma$;*
- *$S(X(t)) := S(X)(S(t))$ for $X \in \mathcal{V}$.*

A substitution S is a solution *of the context equation $u = v$ if $S(u) = S(v)$. The* size *of a solution S of an equation $u = v$ is $|S(u)|$, which is simply the total* *number of nodes in $S(u)$. A solution is* size-minimal*, if for every other solution S' it holds that $|S(u)| \leq |S'(u)|$. A solution S is* non-empty *if $S(X)$ is not a parameter for each $X \in \mathcal{X}$ from the context equation $u = v$.*

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 2 and that it comes from α if it was obtained from $S(\alpha)$ in Definition 2.

Example 1. Consider an alphabet $\Sigma = \{f, c, c'\}$ with $\text{ar}(f) = 2$ and $\text{ar}(c) = \text{ar}(c') = 0$ and an equation $X(c) = Y(c')$ over it. It has a solution (which is easily seen to be size-minimal) $S(X) = f(\bullet, c')$ and $S(Y) = f(c, \bullet)$; under this substitution this equation evaluates to $S(X(c)) = S(Y(c')) = f(c, c')$.

3 Local Compression of Trees

3.1 Compression Operations

We perform three types of *subpattern compression* on a tree t:

a-chain compression. For a unary letter a and every $\ell > 1$ we replace each occurrence of a maximal a-chain subpattern a^ℓ by a new unary letter a_ℓ.

ab compression. For two different unary letters a and b we replace each occurrence of a subpattern ab with a new unary letter c.

(f, i, c) compression. For a constant c and letter f of arity $\text{ar}(f) = m \geq i \geq 1$, we replace each occurrence of subpattern (f, i, c), i.e. $f(\bullet_1, \bullet_2, \ldots, \bullet_{i-1}, c, \bullet_i, \ldots, \bullet_{m-1})$, with subpattern $f'(\bullet_1, \bullet_2, \ldots, \bullet_{i-1}, \bullet_i, \ldots, \bullet_{m-1})$, where f' is a fresh letter of arity $m - 1$ (intuitively: the constant c on i-th place is 'absorbed' by its father labelled with f).

These operations are collectively called *subpattern compressions*. When we want to specify the type but not the actual subpattern compressed, we use the names *chain compression*, *pair compression* and *leaf compression*. These operations are also called $\mathsf{TreePattComp}(ab, t)$, $\mathsf{TreePattComp}(a, t)$ and $\mathsf{TreePattComp}((f, i, c), t)$, or simply $\mathsf{TreePattComp}(p, t)$ for a pattern $p \in \{a, ab, (f, i, c)\}$.

The a-chain compression and ab compression are direct translations of the operations used in the recompression-based algorithm for word equations [11]. On the other hand, the leaf compression is a new operation that is designed specifically to deal with trees.

3.2 Compression of Non-crossing Patterns

Consider a context equation $u = v$ and its solution S. Suppose that we want to perform the ab compression on $S(u)$ and $S(v)$, i.e. we want to replace each occurrence of a subpattern ab 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 $S(\alpha)$ for some variable α: in the former case we modify the equation by replacing the occurrences of ab with c, in the latter the modification is done implicitly (i.e. we replace the occurrences of fab in $S(\alpha)$ with c). The problematic part is with the ab subpattern that is of neither of those forms, as they 'cross' between $S(\alpha)$ and some letter outside $S(\alpha)$. This is formalised in the below definition, the intuition and definition is similar also for a-chains and (f, i, c) patterns.

Definition 3. *For an equation $u = v$ and a substitution S we say that an occurrence of a subpattern p in $S(u)$ (or $S(v)$) is*

explicit with respect to S: *if all non-parameter letters in this occurrence come from explicit letters in $u = v$;*
implicit with respect to S: *if all non-parameter letters in this occurrence come from $S(\alpha)$ for a single occurrence of a variable α;*
crossing with respect to S: *otherwise.*

We say that ab is a crossing pair *(a* has a crossing chain*; (f, i, c) is a* crossing father-leaf subpattern*) with respect to S if it has at least one crossing occurrence (there is a crossing occurrence of an a^ℓ chain; has at least one crossing occurrence) with respect to S. Otherwise ab (a, (f, i, c)) is a* non-crossing pair *(*has no crossing chain*; is a* non-crossing father-leaf subpattern*) with respect to S.*

To make the notions more uniform, we will also say that $p \in \{ab, a, (f, i, c)\}$ is a crossing/non-crossing subpattern, meaning that ab is a crossing/noncrossing pair, a has crossing chain/has no crossing chains and (f, i, c) is a crossing/non-crossing father-leaf subpattern.

When ab is non-crossing with respect to a solution S, we can simulate the $\mathsf{TreePattComp}(ab, S(u))$ on $u = v$ simply by performing the ab compression on the explicit letters in the equation, we refer to this operation as $\mathsf{PattCompNCr}(ab, `u = v\text{'})$. Then occurrences of ab that come from explicit letters are compressed, the ones that come from $S(\alpha)$ are compressed by changing the solution and there are no other possibilities. The same applies also to compression of a-chains and (f, i, c)-compression.

As we discuss correctness of nondeterministic procedures, in the following we will say that a nondeterministic procedure is *sound*, if given a non-satisfiable context equation it cannot transform it to a satisfiable equation, regardless of the nondeterministic choices.

Lemma 1. $\mathsf{PattCompNCr}$ *is sound.*

Let $u = v$ has a solution S and let $p \in \{ab, a, (f, i, c)\}$ be a noncrossing subpattern. Then the returned equation $u' = v'$ has a solution S' (over an alphabet expanded by letters introduced during the subpattern compression) such that $S'(u') = \mathsf{TreePattComp}(p, S(u))$.

Example 2. Consider the following context equation over the alphabet $\{a, b, c, c', f\}$, with $\mathrm{ar}(a) = \mathrm{ar}(b) = 1$, $\mathrm{ar}(c) = \mathrm{ar}(c') = 0$ and $\mathrm{ar}(f) = 2$:

$$bXaXc = baa(f(aYc', bb(f(c, c')))),$$

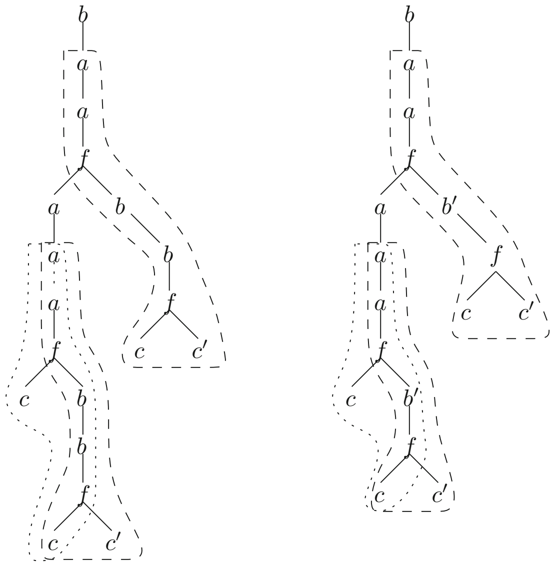

Fig. 1. An illustration to Example 2. The Figure presents the tree obtained at both sides under the substitution, the values substitutions for variables are depicted: in dashed line for X and dotted for Y. On the right the tree after the compression of bb is depicted.

see also Fig. 1. It is easy to see that there is a unique solution $S(X) = aa(f(\bullet, bb(f(c,c'))))$ and $S(Y) = aa(f(c, bb(f(c,\bullet))))$.

The subpattern ba has a crossing occurrence on the left-hand side, as a is the first letter of $S(X)$ and bX is a subpattern. Subpattern bb has only noncrossing occurrences, some of them explicit and some implicit. Subpatterns $(f, 2, c'), (f, 1, c), a^3$ are also crossing.

Compressing b subpattern leads to an equation

$$bXaXc = baa(f(baYc', b'(f(c,c')))).$$

Then the solution is $S'(X) = aa(f(b\bullet, b'(f(c,c'))))$ and $S'(Y) = aa(f(bc, b'(f(c,\bullet))))$.

4 Uncrossing

In general, one cannot assume that an arbitrary pair ab, a-chain or (f, i, c) subpattern is non-crossing. However, for a fixed subpattern p and a solution S we can modify the instance so that this p becomes non-crossing with respect to a solution S' (that corresponds to S of the original equation). This modification is the cornerstone of our main algorithm, as it allows subpattern compression to be performed directly on the equation, regardless of how the solution actually looks like.

4.1 Uncrossing a Pair

We begin with showing how to turn a crossing pair ab into a non-crossing one. As a first step, we characterise crossing pairs in a more operational manner: for a non-empty substitution S, a variable α and a context variable X by a *first letter* of $S(\alpha)$ we denote the topmost-letter in $S(\alpha)$, by the *last letter* of $S(X)$ we denote the function symbol that is the father of '$\bullet$' in $S(X)$. Then ab is crossing with respect to S if and only if one of the following conditions holds for some variable α and context variable X:

(CP1) $a\alpha$ occurs in $u = v$ and b is the first letter of $S(\alpha)$ *or*
(CP2) Xb occurs in $u = v$ and a is the last letter of $S(X)$ *or*
(CP3) $X\alpha$ occurs in $u = v$, a is the last letter of $S(X)$ and b the first letter of $S(\alpha)$.

In each of (CP1–CP3) it is easy to modify the instance so that ab is no longer a crossing pair:

Ad (CP1): We *pop up* the letter b: we replace α with $b\alpha$. In this way we also modify the solution $S(\alpha)$ from bt to t. If the new substitution for α is empty (which can happen only when α is a context variable), we remove α from the equation.
Ad (CP2): We *pop down* the letter a: we replace each occurrence of X with Xa. In this way we implicitly modify the solution $S(X)$ from sa to s. If the new substitution for X is empty, we remove X from the equation.
Ad (CP3): this is a combination of the two cases above, in which we need to pop-down from X and pop-up from α.

The whole uncrossing procedure can be even simplified: for each context variable X we guess its last letter and it if is a, then we pop it down. Similarly, for each variable α we guess its first letter and if it is b then we pop it up. If at any point a context some variable becomes empty then we remove its occurrences. We call the whole procedure Uncross $(ab, `u = v')$ and its details are in the pseudocode in Algorithm 1.

If $u = v$ has a solution S then for appropriate non-deterministic choices Uncross$(ab, `u = v')$ returns an equation $u' = v'$ that has a solution S' such that ab is non-crossing with respect to S' and $S'(u') = S(u)$.

Lemma 2. *Let $a \neq b$ be two different unary letters. Then* Uncross$(ab, `u = v')$ *is sound and if $u = v$ has a non-empty solution S (over an alphabet Σ) then for appropriate non-deterministic choices the returned equation $u' = v'$ has a non-empty solution S' (over the same alphabet Σ) such that $S'(u') = S(u)$ and ab is a non-crossing pair with respect to S'.*

Example 3. Continuing Example 2, recall that ba is a crossing subpattern in

$$bXaXc = baa(f(aYc', bb(f(c, c')))),$$

Algorithm 1. Uncross(ab, '$u = v$')

```
1: for $X \in \mathcal{V}$ do
2:     if the last letter of $S(X)$ is $a$ then                                   ▷ Guess
3:         replace each occurrence of $X$ in $u = v$ by $Xa$
4:                                          ▷ Implicitly change $S(X) = sa$ to $S(X) = s$
5:         if $S(X)$ is empty then                                                 ▷ Guess
6:             remove $X$ from $u = v$: replace each $Xs$ in the equation by $s$
7: for $\alpha \in \mathcal{V} \cup \mathcal{X}$ do
8:     if the first letter of $S(\alpha)$ is $b$ then                             ▷ Guess
9:         replace each occurrence of $\alpha$ in $u = v$ by $b\alpha$
10:                                         ▷ Implicitly change $S(\alpha) = bs$ to $S(\alpha) = s$
11:        if $S(\alpha)$ is empty then          ▷ Guess; applies only to context variables
12:            remove $\alpha$ from $u = v$: replace each $\alpha s$ in the equation by $s$
```

where the alphabet is $\{a, b, c, c', f\}$, with $\text{ar}(a) = \text{ar}(b) = 1$, $\text{ar}(c) = \text{ar}(c') = 0$ and $\text{ar}(f) = 2$ and $S(X) = a(a(f(\bullet, b(b(f(c, c'))))))$ and $S(Y) = a(a(f(c, b(b(f(c, \bullet))))))$. We pop a up from X, obtaining

$$baXbaaXbc = baa(f(aYc', bb(f(c, c')))),$$

the solution is $S'(X) = a(f(\bullet, bb(f(c, c'))))$ and $S'(Y) = S(Y)$. It is easy to verify that ba is no longer crossing.

4.2 Uncrossing Chains

Suppose that some unary letter a has a crossing chain with respect to a non-empty solution S. As in the case of pairs, it is easy to see that a has a crossing chain with respect to S if and only if one of the following holds for variable α and context variable X (note that those conditions are in fact (CP1–CP3) for $a = b$):

(CC1) $a\alpha$ occurs in $u = v$ and the first letter of $S(\alpha)$ is a;
(CC2) Xa occurs in $u = v$ and a is the last letter of $S(X)$;
(CC3) $X\alpha$ occurs in $u = v$ and a is the last letter of $S(X)$ as well as the first letter of $S(\alpha)$.

The cases (CC1) and (CC2) are symmetric while (CC3) is a composition of (CC1) and (CC2). So suppose that (CC2) holds. Then we can replace each occurrence of X in the equation $u = v$ with Xa (implicitly changing the solution $S(X) = ta$ 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 $S(X) = ta^r$ and the last letter of t is not a. Then a and X can no longer satisfy condition (CC2), as the last letter of $S'(X)$ is 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 α. To simplify the arguments, for a ground term or 1-pattern 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 1-pattern t we say that a^r is the *a-suffix* of t if $t = t'a^r$ and the last letter of t' is not a (in particular, t' may be empty).

We call this procedure Uncross $(a, `u = v')$, its formal details are given in Algorithm 2.

Algorithm 2. Uncross $(a, `u = v')$ Uncrossing all a-chains

```
 1: for $\alpha \in \mathcal{V} \cup \mathcal{X}$ do
 2:     if $a$ is the first letter of $S(\alpha)$ then                         ▷ Guess
 3:         guess $\ell \geq 1$                              ▷ $a^\ell$ is the $a$-prefix of $S(\alpha)$
 4:         replace each $\alpha$ in $u = v$ by $a^\ell \alpha$   ▷ implicitly change $S(\alpha) = a^\ell t$ to $S(\alpha) = t$
 5:         if $S(\alpha)$ is empty then              ▷ Guess; applies only to context variables
 6:             remove $\alpha$ from $u = v$: replace each $\alpha(t)$ in the equation by $t$
 7: for $X \in \mathcal{V}$ do
 8:     if $a$ is the last letter of $S(X)$ then                          ▷ Guess
 9:         guess $r \geq 1$                                 ▷ $a^r$ is the $a$-suffix of $S(X)$
10:         replace each $X$ in $u = v$ by $Xa^r$          ▷ implicitly change $S(X) = ta^r$ to
    $S(X) = t$
11:         if $S(X)$ is empty then                                        ▷ Guess
12:             remove $X$ from $u = v$: replace each $X(t)$ by $t$
```

Lemma 3. Uncross$(a, `u = v')$ *is sound and if $u = v$ has a non-empty solution S (over an alphabet Σ) then for appropriate non-deterministic choices the returned equation $u' = v'$ has a non-empty solution S' (over the same alphabet Σ) such that $S'(u') = S(u)$ and a has no crossing chains with respect to S'.*

Example 4. Continuing Example 2:

$$bXaXc = baa(f(aYc', bb(f(c, c')))),$$

where the alphabet is $\{a, b, c, c', f\}$, with $\text{ar}(a) = \text{ar}(b) = 1$, $\text{ar}(c) = \text{ar}(c') = 0$ and $\text{ar}(f) = 2$ and $S(X) = aa(f(\bullet, bb(f(c, c'))))$ and $S(Y) = aa(f(c, bb(f(c, \bullet))))$.

There are crossing a chains, because aY is a subpattern and a is the first letter of $S(Y)$. We pop the a-prefixes from X, Y, there are no a-suffixes. The instance is now

$$baaXaaaXc = baa(f(aaaYc', bb(f(c, c')))),$$

where $S'(X) = f(\bullet, bb(f(c, c')))$ and $S'(Y) = f(c, bb(f(\bullet, c)))$. It is easy to verify that a has no crossing chains.

4.3 Uncrossing Father-Leaf Subpattern

We now show how to uncross a father-leaf subpattern (f, i, c). It is easy to observe that father-leaf subpattern (f, i, c) is crossing (with respect to a non-empty S) if and only if one of the following holds for some context variable X and term variable y:

(CFL 1) f with an i-th son y occurs in $u = v$ and $S(y) = c$;
(CFL 2) Xc occurs in $u = v$ and the last letter of $S(X)$ is f and $\bullet$ is its i-th child;
(CFL 3) Xy occurs in $u = v$, $S(y) = c$ and f is the last letter of $S(X)$ and $\bullet$ is its i-th child.

We want to 'pop-up' c and 'pop-down' f. Popping up c is easy (we replace y with c); popping-down f is more complex. Let us first present the intuition:

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

We call this procedure Uncross$((f, i, c), `u = v')$, its formal description is given in Algorithm 3.

Algorithm 3. Uncross$((f, i, c), `u = v')$

```
1: for $x \in \mathcal{X}$ do
2:     if $S(x) = c$ then                                              ▷ Guess
3:         replace each $x$ in $u = v$ by $c$              ▷ $S$ is no longer defined on $x$
4: let $m \leftarrow \text{ar}(f)$
5: for $X \in \mathcal{V}$ do
6:     if $f$ is the last letter of $S(X)$, $\bullet$ is its $i$-th child and $Xc$ is a subpattern in $u = v$
   then                                                                ▷ Guess
7:         replace each $X$ in $u = v$ by $X(f(x_1, x_2, \ldots, x_{i-1}, \bullet, x_i, \ldots, x_{m1-}))$
             ▷ Implicitly change $S(X) = sf(t_1, t_2, \ldots, t_{i-1}, \bullet, t_i, \ldots, t_{m-1})$ to $S(X) = s$
           ▷ Add new variables $x_1, \ldots, x_{m-1}$ to $\mathcal{X}$ and extend $S$ by setting $S(x_j) = t_j$
8:         if $S(X)$ is empty then                                     ▷ Guess
9:             remove $X$ from the equation: replace each $X(t)$ in the equation by $t$
10: for new variables $x \in \mathcal{X}$ do
11:     if $S(x) = c$ then                                             ▷ Guess
12:         replace each $x$ in $u = v$ by $c$             ▷ $S$ is no longer defined on $x$
```

Lemma 4. *Let* $\text{ar}(f) \geq i \geq 1$ *and* $\text{ar}(c) = 0$, *then* Uncross$((f, i, c), `u = v')$ *is sound and if* $u = v$ *has a non-empty solution* S *(over an alphabet* Σ*) then for* *appropriate non-deterministic choices the returned equation $u' = v'$ has a non-empty solution S' (over the same alphabet Σ) such that $S'(u') = S(u)$ and there is no crossing father-leaf subpattern (f, i, c) with respect to S'.*

Example 5. Continuing Example 2, the equations

$$bXaXc = baa(f(aYc', bb(f(c, c')))),$$

where the alphabet is $\{a, b, c, c', f\}$, with $\mathrm{ar}(a) = \mathrm{ar}(b) = 1$, $\mathrm{ar}(c) = \mathrm{ar}(c') = 0$ and $\mathrm{ar}(f) = 2$, has a solution $S(X) = aa(f(\bullet, bb(f(c, c'))))$ and $S(Y) = aa(f(c, bb(f(c, \bullet))))$. A subpattern $(f, 2, c')$ is crossing, as Yc' is a subpattern, the last letter of Y is f and the hole $\bullet$ is the second child of this f We uncross it by popping down from Y: The instance is now

$$bXaXc = baa(f(aY(f(y, c')), bb(f(c, c'))))$$

where $S'(X) = S(X)$, $S'(Y) = f(c, b(b(\bullet)))$ and $S(y) = c$. It is easy to see that $(f, 2, c')$ is now noncrossing.

5 The Algorithm

In its main part, ContextEqSatSimp iterates the following operation: it nondeterministically chooses to perform one of the compressions: ab compression, a-chain compression or (f, i, c) compression, where a, b, c, f are some letters of appropriate arity. It then nondeterministically choose, whether this pattern is crossing or not. If so, it performs the appropriate uncrossing. Then it performs the subpattern compression for p and adds the new letter (or letters, for chains compression) to Σ. We call one iteration of main loop of ContextEqSatSimp a *phase*.

Algorithm 4. ContextEqSatSimp(‘$u = v$’, Σ) Checking the satisfiability of a context equation $u = v$

```
1: while $|u| > 1$ or $|v| > 1$ do
2:     choose $p$ from $\{a, ab, (f, i, c)\}$ to compress, $a, b, c, f \in \Sigma$
3:     if $p$ is crossing then                                              ▷ Guess
4:         Uncross($p$, ‘$u = v$’)
5:     PattCompNCr($a$, ‘$u = v$’)
6:     add letters representing compressed subpatterns to $\Sigma$
7: Solve the problem naively            ▷ With sides of size 1, the problem is trivial
```

The extended algorithm ContextEqSat works in the same way, except that at the beginning of each iteration it removes from the alphabet the letters that are neither from the original alphabet neither are present in the current context equation. It is easy to show that such removal does not change the satisfiability of the given equation.

Theorem 1. ContextEqSatSimp *and* ContextEqSat *store an equation of length $\mathcal{O}(n^2k^2)$, where n is the size of the input equation and k the maximal arity of symbols from the input alphabet. They non-deterministically solve context equations, in the sense that:*

- *if the input equation is not-satisfiable then they never return 'YES';*
- *if the input equation is satisfiable then for some nondeterministic choices in $\mathcal{O}(n^3k^3 \log N)$ phases they return '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.

Theorem 2. *Context unification is in* PSPACE.

6 Space Bounds

While the soundness of the algorithm follows from soundness of its subprocedures, the space bounds, and so the termination, remains to be shown.

6.1 General Bounds

First, we recall that the following known bound on the size of the a-chains for size-minimal solutions. This ensures that we can compress the chains in polynomial space.

Lemma 5 (Exponent of periodicity bound [28]). *Let S be a size-minimal solution of a context equation $u = v$ (for an alphabet Σ). Suppose that $S(X)$ (or $S(x)$) can be written as $ta^m t'$, where t, t' are 1-patterns (or t' is a ground term, respectively) and a is a unary letter. Then $m = 2^{\mathcal{O}(|u|+|v|)}$.*

Now we bound the number of variables occurrences during the algorithm. Note that this bound works for all nondeterministic choices.

Lemma 6. *The number of occurrences of context variables during* ContextEqSat *is at most n. The number of occurrences of term variables is at most nk.*

The bound for context variables is obvious, as we never introduce new ones. For term variables observe that during the (f, i, c) uncrossing we introduce new term variables, by popping them from context variables. However, it can be shown that when we pop new term variables from X, all term variables previously introduced by X have been removed. This yields the bound.

As a next step, we estimate the number of different crossing subpatterns. This follows by a simple argument that such a pattern can be associated with a top or bottom letter in a variable.

Lemma 7. *For an equation $u = v$ during* ContextEqSat *and its solution S the number of different crossing subpatterns of the form $a, ab, (f, i, c)$ is at most $n(k + 1)$.*

We can also limit the number of new letters introduced during the uncrossing. Again, this follows a simple calculation.

Lemma 8. *Uncrossing and compression of a subpattern introduces at most* $n(k+1)$ *letters to the equation.*

6.2 Strategy

The strategy of choosing nondeterministic choices is easy: if there is a noncrossing pattern, then we compress it, as this decreases both the size of the equation and of the minimal solution.

If there is none, then we choose a pattern, whose compression makes equation smallest possible (after this one compression). As there are only $n(k+1)$ such candidates, see Lemma 7, one of them will appear roughly $(|u|+|v|)/n(k+1)$ many times. Its compression removes $(|u|+|v|)/2n(k+1)$ letters and introduces at most $n(k+1)$ many letters, see Lemma 8. This shows that we never exceed the quadratic bound on $|u|+|v|$ given in Theorem 1.

If we additionally make choices so as to minimize the size of the solution, then we can guarantee to terminate after the number of steps depending on $\log N$ (and not N), so as claimed in Theorem 1.

7 Detailed Example

We now run the algorithm on Example 2, see also the Fig. 2. Recall the equation:

$$bXaXc = ba(a(f(aYc', b(b(f(c,c')))))),$$

where the alphabet is $\Sigma = \{a, b, c, c', f\}$, with $\text{ar}(a) = \text{ar}(b) = 1$, $\text{ar}(c) = \text{ar}(c') = 0$ and $\text{ar}(f) = 2$ and $S(X) = a(a(f(\bullet, b(b(f(c,c'))))))$ and $S(Y) = a(a(f(c, b(b(f(c,\bullet))))))$. There is no need to preprocess the alphabet.

As b has no crossing chains, so we compress it, obtaining

$$bXaXc = baa(f(aYc', b'(f(c,c')))),$$

where the alphabet is $\Sigma \cup \{b'\}$, with $\text{ar}(b') = 1$ and the solution and $S(X) = aa(f(\bullet, b'(f(c,c'))))$ and $S(Y) = aa(f(c, b'(f(c,\bullet))))$. Now every potential subpattern for compression is crossing. We choose $(f,1,c)$ for compression and uncross it. To this end we pop f down from X. Note, that according to the algorithm, f is not popped down from Y, even though f is its last letter, it does not take part in any crossing occurrence of a subpattern $(f,1,c)$. The instance is now

$$bX(f(aXf(c,x),x)) = baa(f(aYc', b'(f(c,c')))),$$

with a solution $S(X) = aa$, $S(Y) = aa(f(c, b'(f(c,\bullet))))$ and $S(x) = b'(f(c,c'))$. We compress $(f,1,c)$, obtaining

$$bX(f(aXf'x,x)) = baa(f(aYc', b'f'c')),$$

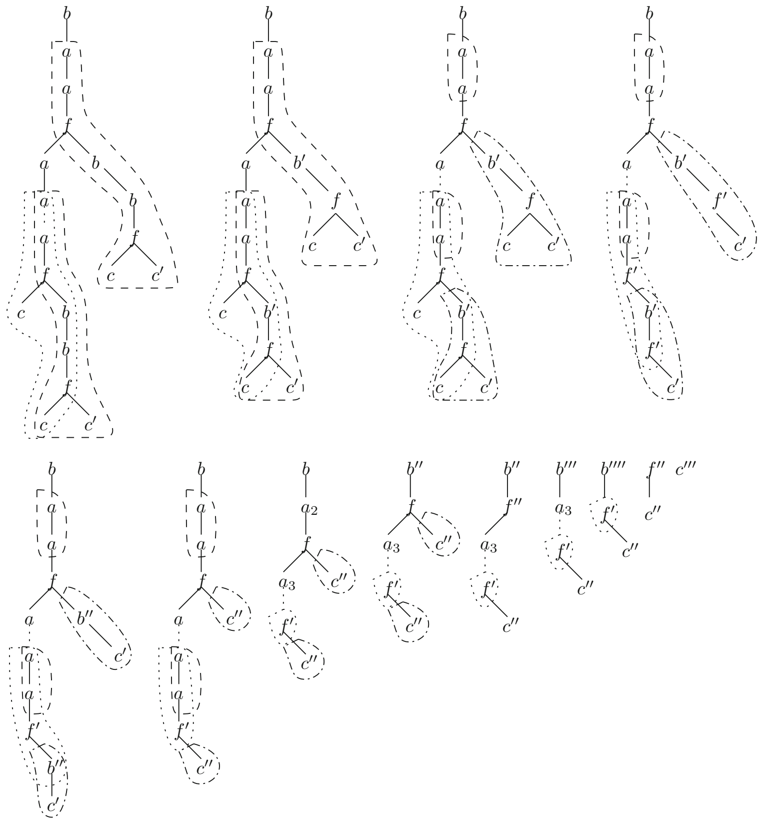

Fig. 2. An illustration of running the algorithm on the instance from Example 2. The Figure presents the term obtained at both sides under the substitution, the values substitutions for variables are depicted: in dashed line for X and dotted for Y, later also dash-dotted for x. The changes are according to the consecutive actions described in the section: b chains compression, uncrossing $(f, 1, c)$, compression of $(f, 1, c)$, compression of $b'f'$, uncrossing and compression of $(b'', 1, c')$, uncrossing a-chains and compression of a chains, compression of ba_2, uncrossing and compression of $(f, 2, c'')$, compression of $b''f''$, compression of $b'''a_3$, uncrossing and compression of $b''''f'$ and finally compression of $(f'', 1, c'')$.

with a solution $S(X) = aa$, $S(Y) = aaf'b'f'$, $S(x) = b'f'c'$, here f' is of arity $\text{ar}(f') = 1$ is added to the alphabet, which is now $\Sigma \cup \{b', f'\}$. Note that there is a new term variable x. Now $b'f'$ is noncrossing, so we compress it (into b'' of arity 1), obtaining

$$bX(f(aXf'x, x)) = baa(f(aYc', b''c')),$$

with a solution $S(X) = aa$, $S(Y) = aaf'b''$ and $S(x) = b''c'$. The b' is no longer used, so it is removed from the alphabet, which is now $\Sigma \cup \{b'', f'\}$ We choose to compress $b''c'$, as a $(b, 1, c')$ subpattern, we first uncross it by popping f down from Y and then we compress into a new constant c'', the b'' can be removed. So the alphabet is now $\Sigma \cup \{f', c''\}$ and the equation is

$$bX(f(aXf'x, x)) = baa(f(aYc'', c'')),$$

with a solution $S(X) = aa$, $S(Y) = aaf'$ and $S(x) = c''$.

Now we uncross and compress a-blocks: in uncrossing we replace X with a^2 and pop-up a^2 from Y. Afterwards we replace a^2 with a_2 and a^3 with a_3. The alphabet is now $\Sigma \cup \{f', c'', a_2, a_3\}$ and the equation is

$$ba_2(f(a_3f'x, x)) = ba_2(f(a_3Yc'', c'')),$$

with a solution $S(Y) = f'$ and $S(x) = c''$.

Now we compress the noncrossing ba_2 into b'' (note that we reuse the letter b''), a_2 is removed from the alphabet, which is now $\Sigma \cup \{b'', f', c'', a_3\}$ and the equation is

$$b''(f(a_3f'x, x)) = b''(f(a_3Yc'', c'')),$$

with a solution $S(Y) = f'$ and $S(x) = c''$. Now we choose to compress $(f, 2, c'')$, so we pop up c'' from x (thus removing it). After the compression, the alphabet is $\Sigma \cup \{b'', f', f'', c'', a_3\}$ and the equation is $b''f''a_3f'c'' = b''f''a_3Yc''$, with a solution $S(Y) = f'$.

We compress noncrossing $b''f''$, obtaining the alphabet $\Sigma \cup \{b''', f', c'', a_3\}$ and the equation is $b'''a_3f'c'' = b'''a_3Yc''$, with a solution $S(Y) = f'$. We now compress $b''a_3$, which is noncrossing, yielding an equation $b''''f'c'' = b''''Yc''$, with a solution $S(Y) = f'$ over an alphabet $\Sigma \cup \{b'''', f', c''\}$.

Now we uncross $b''''f'$, by replacing Y with f' and compress it, obtaining a trivial equation $f''c'' = f''c''$ over an alphabet $\Sigma \cup \{f'', c''\}$, for which we perform the final compression $(f'', 1, c'')$, yielding an equation $c''' = c'''$.

8 Regular Constraints

We now consider the problem of context unification with regular constraints. In this setting, the input contains (some description of) regular tree languages $L_1, \ldots, L_\ell$ of ground terms and/or 1-patterns over the input alphabet Σ. Those automata are used for enriching the equation with additional constraints of the form $\alpha \in L_i$, meaning that the substitution for the variable α should be from language L_i. Naturally, those languages have to be specified in some way, we choose one, see Sect. 8.2, but other natural descriptions are equivalent, see discussion at the end of that section.

Context unification with regular constraints was investigated mostly because *linear second order unification* [12] and context unification with regular constraints reduce to each other [16]; note that those reductions are not polynomial-time, so cannot be used directly to claim the computational complexity of linear second order unification. On the other hand, adding constraints to unification is interesting and important on its own.

To generalise ContextEqSat to this setting, we assign to each letter in the alphabet its transition function; such transition functions can be generalised to patterns, so in particular to substitutions for variables; transition vectors and vectors of such transitions are defined in Sect. 8.1 and we explain how to use them to define constraints for context unification in Sect. 8.2. When we compress a certain subpattern into one letter, we compose those transition functions. When we pop letters from variables we assign to the variable a new transition function, so that the composition of transition function of popped letters and the new transition function for a variable is equal to the old transition function for a variable.

However, several simplifications do not work in this setting. This is for a reason: it is known that the non-emptiness problem for intersection of (deterministic) finite tree automata is EXPTIME-complete [5] and we can easily encode this problem within context unification with regular constraints, so we cannot hope to extend our algorithm to this setting without affecting the computational complexity.

To resolve this problem, we *extend* the input alphabet by adding to it one letter f_Δ for every possible vector of transition functions Δ (we limit the allowed arities, though, to the maximal arity of letters in the input alphabet). We do not store this alphabet explicitly, instead we use an (EXPTIME) oracle to decide, whether a letter belongs to the alphabet or not. It is easy to see that a context equation is equisatisfiable over its input alphabet and over such an extended alphabet. Later on, for any equation, we consider its solution over an alphabet consisting of the extended alphabet and letters present in the equation.

Ultimately, for algorithms ContextEqSatRegSimp and ContextEqSatReg which are generalisations of ContextEqSatSimp and ContextEqSat to scenarios with regular constraints, respectively, we want to show the following theorem.

Theorem 3. ContextEqSatRegSimp *and* ContextEqSatReg *keep an equation of size $\mathcal{O}(n^2k^2)$. Given an unsatisfiable context equation with regular constraints they never return 'YES'. Given a satisfiable one with a minimal-size solution of size N they return 'YES' after $\mathcal{O}(n^2k^2 \log N)$ compression steps, for appropriate non-deterministic choices.*

ContextEqSatReg *uses an oracle for the intersection of tree-regular languages, which can always be implemented in* EXPTIME*. Except for that, it runs in* PSPACE*.*

As a corollary, we get an EXPTIME bound on the satisfiability of context unification with regular constraints.

Corollary 1. *Context unification with regular constraints is* EXPTIME-*complete.*

8.1 Tree Automata

A tree automaton is defined as a triple $(\Sigma, Q, \delta_{f\,f\in\Sigma})$, where Σ is a ranked alphabet, Q is a set of states and each δ_f is a transition relation (sometimes called *transition function* for historical reasons) of a letter from Σ.

To be more precise, when $\mathrm{ar}(f) = r$ then $\delta_f \subseteq Q^r \times Q$. The meaning of the transition functions is that we consider labelling of ground terms with states (such labellings are usually called *runs*) such that a node labelled with f whose children are assigned states $(q_1, q_2, \ldots, q_r)$ can be assigned any state q such that $(q_1, q_2, \ldots, q_r, q) \in \delta_f$. Thus we think of δ_f as a nondeterministic transition function, and by $\delta_f(q_1, q_2, \ldots, q_{\mathrm{ar}(f)})$ we denote the set of states $\{q : (q_1, q_2, \ldots, q_{\mathrm{ar}(f)}, q) \in \delta_f\}$.

We can treat a letter f as a pattern with a unique non-parameter node f; in this way we can define δ_p for arbitrary patterns: given an r-pattern p the tuple $(q_1, q_2, \ldots, q_{\mathrm{ar}(f)}, q)$ is in δ_p if and only if there is a run for p in which nodes $\bullet_1, \bullet_2, \ldots, \bullet_{\mathrm{ar}(f)}$ are labelled with $q_1, q_2, \ldots, q_{\mathrm{ar}(f)}$ and the root of p is labelled with q. Note that this implicitly gives a rule of composing transition functions; such composition is associative in case of 1-patterns.

Concerning the notation, we will explicitly compose only transition functions for patterns that occur during the subpattern compression. Thus for unary patterns $p_1, p_2, \ldots, p_\ell$ by $\delta_{p_1}\delta_{p_2}\ldots\delta_{p_\ell}$ we denote the transition function of the pattern $p_1p_2\ldots p_\ell$ and when $p_1 = p_2 = \cdots = p_\ell$ then we denote this function by $(\delta_{p_1})^\ell$. Similarly, for an r-ary pattern f and ground terms $t_1, t_2, \ldots, t_m$ by $\delta_f[\bullet_{i_1}/\delta_{t_1}, \bullet_{i_2}/\delta_{t_2}, \ldots, \bullet_{i_m}/\delta_{t_m}]$ we denote the transition function of a pattern obtained by replacing $\bullet_{i_1}, \bullet_{i_2}, \ldots, \bullet_{i_m}$ by $t_1, t_2, \ldots, t_m$, respectively. Note, that it could be that $m < r$, i.e. not all parameters of a pattern f are substituted by ground terms.

For a fixed sequence of automata $A_1, A_2, \ldots, A_\ell$ (and it is fixed for a fixed instance of context unification with regular constraints) with transition functions $\{\delta^1_f\}_{f\in\Sigma}, \{\delta^2_f\}_{f\in\Sigma}, \ldots, \{\delta^\ell_f\}_{f\in\Sigma}$ by Δ_p we denote the tuple of transition functions $(\delta^1_p, \delta^2_p, \ldots, \delta^\ell_p)$ for a pattern p; this is a *vector of transitions* of this pattern. Note that this is a vector of sets. We denote vectors of transitions by $\Delta, \Delta_1, \ldots$ and consider them even without underlying patterns, and refer to r-vector of transitions, when this is a vector of transitions of an r-ary pattern.

We extend the composition of transition functions and its notation to vectors of transitions in a natural way, i.e. we perform the appropriate operation coordinate-wise on each transition function.

8.2 Context Unification with Regular Constraints

Now we are ready to define the problem of context unification with regular constraints. As an input we are given a finite alphabet Σ, finite automata $A_1, A_2, \ldots, A_\ell$ over Σ (with state sets $Q_1, Q_2, \ldots, Q_\ell$ and transition functions $\{\delta^1_f\}_{f\in\Sigma}, \{\delta^2_f\}_{f\in\Sigma}, \ldots, \{\delta^\ell_f\}_{f\in\Sigma}$), a context equation $u = v$ and a set of constraints on the vectors of transitions for variables u in total. To be more precise, those constraints are:

term variable constraints: we are given 0-vectors of transitions Δ_x for some variables $x \in \mathcal{X}$;

equations constraints: similarly, we are given 0-vector of transition Δ_u;

context variable constraints: we are given 1-vectors of transitions Δ_X for some context variables $X \in \mathcal{V}$.

The meaning of the constraints is clear: we ask, whether there is a substitution S, such that $S(u) = S(v)$, $\Delta_{S(u)} = \Delta_u$ and $\Delta_{S(\alpha)} = \Delta_\alpha$ for each variable α.

8.3 Modifications of ContextEqSat

We now explain the modifications of ContextEqSatSimp to ContextEqSatRegSimp, i.e. consider the algorithm that enriches the alphabet with every letter that it created.

Compression. When a subpattern p is compressed into f, we calculate its vector of transitions and set $\Delta_f \leftarrow \Delta_p$.

Popping Letters. When popping letters, we guess the new vectors of transitions for the variable, so that the composition of vectors of transitions (in the appropriate order) of the popped letter and variable is the same as it used to be; this applies also to popping of term variables during the uncrossing of leaf-pair. For instance, when we replace X with $X(f(x_1, \bullet_2, x_3))$ then we guess new transitions $\Delta'_X, \Delta_{x_1}, \Delta_{x_3}$, such that $\Delta_X = \Delta'_X(\Delta_f[\bullet_1/\Delta_{x_1}, \bullet_3/\Delta_{x_3}])$; we add $\Delta_{x_1}, \Delta_{x_3}$ to sets constraints and update Δ_X to Δ'_X. When we remove a context variable X, we need to ensure that its transition function Δ_X is the same as $\Delta(\bullet)$, i.e. it is an identity; similarly, when we replace x with c we need to validate that $\Delta_x = \Delta_c$.

Ending. When the whole equation is reduced to a single equation $c = c$, we check whether the transition function for c is the same as for the whole equation, i.e. $\Delta_u = \Delta_c$. If so, we accept, if not, we reject.

Satisfiability. Whenever we claim that an equation is satisfiable (so, it has some solution S), we need to additionally assert that the transition for a variable (and the whole equation) is as in the constraints kept by the instance, that is, $\Delta_{S(\alpha)} = \Delta_\alpha$ for each variable α and $\Delta_{S(u)} = \Delta_u$.

Subprocedures. Lemma 1 holds in the new setting, to this end it is enough to recall that during compression the new letter has the same transition function as the pattern it replaced and for popping, we always guess the popped letters and the new constraints of variables so that the composition of their vectors of transitions is equal to the vector of transitions of the variable before the popping.

The discussion above shows the proof of Theorem 3 in case of ContextEqSatRegSimp. The only remaining problem is that the alphabet used by ContextEqSatRegSimp grows and the size of transition vectors of the involved letters can be even exponential. However, careful inspection shows that one can define appropriate subclass of all vectors of transitions, called *reachable*. They are of polynomial size and can be composed in polynomial time; moreover, each letter that occurs during ContextEqSatReg has a reachable vector of transition and vice versa—each reachable vector of transitions can be realised by tree or a pattern over the input alphabet. Lastly, one can check in EXPTIME, whether a vector of transitions is reachable. This ends the analysis for Theorem 3.

9 Open Questions

Computational complexity. The exact computational complexity of context unification remains unknown: the presented algorithm shows containment in PSPACE and the best known lower bound is NP, by a trivial reduction of Integer Programming. Perhaps the additional structure of terms allows showing a stronger lower bound?

Size of minimal solutions. Extension of the given proof shows that the size of the smallest solution of a context unification is of at most doubly exponential size. At the same time, we know no solution which is super-exponential, so the same as in the case of word equations. An exponential upper bound would imply containment in NP, a counterexample would somehow suggest that PSPACE *is* the computational complexity of the problem.

Unary second order unification. The decidability status of subproblem of second order unification, in which each second order has arity 1, remains unknown. The presented approach does not generalize to this case and at the same time the existing proof of undecidability essentially requires second-order variables of rank 2.

One context variable. Context unification with one context variable is known to be in NP [6] and some of its fragments are in P [7,8]. It remains an open question, whether the whole problem is in P.

References

1. Comon, H.: Completion of rewrite systems with membership constraints. Part I: deduction rules. J. Symb. Comput. **25**(4), 397–419 (1998). https://doi.org/10.1006/jsco.1997.0185
2. Comon, H.: Completion of rewrite systems with membership constraints. Part II: constraint solving. J. Symb. Comput. **25**(4), 421–453 (1998). https://doi.org/10.1006/jsco.1997.0186
3. Diekert, V., Gutiérrez, C., Hagenah, C.: The existential theory of equations with rational constraints in free groups is PSPACE-complete. Inf. Comput. **202**(2), 105–140 (2005). https://doi.org/10.1016/j.ic.2005.04.002
4. Diekert, V., Jeż, A., Plandowski, W.: Finding all solutions of equations in free groups and monoids with involution. Inf. Comput. **251**, 263–286 (2016). https://doi.org/10.1016/j.ic.2016.09.009
5. Frühwirth, T.W., Shapiro, E.Y., Vardi, M.Y., Yardeni, E.: Logic programs as types for logic programs. In: LICS, pp. 300–309. IEEE Computer Society (1991). https://doi.org/10.1109/LICS.1991.151654
6. Gascón, A., Godoy, G., Schmidt-Schauß, M., Tiwari, A.: Context unification with one context variable. J. Symb. Comput. **45**(2), 173–193 (2010). https://doi.org/10.1016/j.jsc.2008.10.005
7. Gascón, A., Schmidt-Schauß, M., Tiwari, A.: Two-restricted one context unification is in polynomial time. In: Kreutzer, S. (ed.) CSL. LIPIcs, vol. 41, pp. 405–422. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2015). https://doi.org/10.4230/LIPIcs.CSL.2015.405

8. Gascón, A., Tiwari, A., Schmidt-Schauß, M.: One context unification problems solvable in polynomial time. In: LICS, pp. 499–510. IEEE (2015). https://doi.org/10.1109/LICS.2015.53
9. Goldfarb, W.D.: The undecidability of the second-order unification problem. Theor. Comput. Sci. **13**, 225–230 (1981). https://doi.org/10.1016/0304-3975(81)90040-2
10. Jeż, A.: Context unification is in PSPACE. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014. LNCS, vol. 8573, pp. 244–255. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43951-7_21. Full version http://arxiv.org/abs/1310.4367
11. Jeż, A.: Recompression: a simple and powerful technique for word equations. J. ACM **63**(1), 4:1 (2016). https://doi.org/10.1145/2743014
12. Levy, J.: Linear second-order unification. In: Ganzinger, H. (ed.) RTA 1996. LNCS, vol. 1103, pp. 332–346. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61464-8_63
13. Levy, J., Agustí-Cullell, J.: Bi-rewrite systems. J. Symb. Comput. **22**(3), 279–314 (1996). https://doi.org/10.1006/jsco.1996.0053
14. Levy, J., Schmidt-Schauß, M., Villaret, M.: The complexity of monadic second-order unification. SIAM J. Comput. **38**(3), 1113–1140 (2008). https://doi.org/10.1137/050645403
15. Levy, J., Schmidt-Schauß, M., Villaret, M.: On the complexity of bounded second-order unification and stratified context unification. Log. J. IGPL **19**(6), 763–789 (2011). https://doi.org/10.1093/jigpal/jzq010
16. Levy, J., Villaret, M.: Linear second-order unification and context unification with tree-regular constraints. In: Bachmair, L. (ed.) RTA 2000. LNCS, vol. 1833, pp. 156–171. Springer, Heidelberg (2000). https://doi.org/10.1007/10721975_11
17. Levy, J., Villaret, M.: Currying second-order unification problems. In: Tison, S. (ed.) RTA 2002. LNCS, vol. 2378, pp. 326–339. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45610-4_23
18. Makanin, G.: The problem of solvability of equations in a free semigroup. Matematicheskii Sbornik **2**(103), 147–236 (1977). (in Russian)
19. Marcinkowski, J.: Undecidability of the first order theory of one-step right ground rewriting. In: Comon, H. (ed.) RTA 1997. LNCS, vol. 1232, pp. 241–253. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-62950-5_75
20. Niehren, J., Pinkal, M., Ruhrberg, P.: On equality up-to constraints over finite trees, context unification, and one-step rewriting. In: McCune, W. (ed.) CADE 1997. LNCS, vol. 1249, pp. 34–48. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63104-6_4
21. Niehren, J., Pinkal, M., Ruhrberg, P.: A uniform approach to underspecification and parallelism. In: Cohen, P.R., Wahlster, W. (eds.) ACL, pp. 410–417. Morgan Kaufmann Publishers/ACL (1997). https://doi.org/10.3115/979617.979670
22. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. J. ACM **51**(3), 483–496 (2004). https://doi.org/10.1145/990308.990312
23. RTA Problem List: Problem 90 (1990). http://rtaloop.mancoosi.univ-paris-diderot.fr/problems/90.html
24. Schmidt-Schauß, M.: Unification of stratified second-order terms. Internal Report 12/94, Johann-Wolfgang-Goethe-Universität (1994)
25. Schmidt-Schauß, M.: A decision algorithm for distributive unification. Theor. Comput. Sci. **208**(1–2), 111–148 (1998). https://doi.org/10.1016/S0304-3975(98)00081-4
26. Schmidt-Schauß, M.: A decision algorithm for stratified context unification. J. Log. Comput. **12**(6), 929–953 (2002). https://doi.org/10.1093/logcom/12.6.929

27. Schmidt-Schauß, M.: Decidability of bounded second order unification. Inf. Comput. **188**(2), 143–178 (2004). https://doi.org/10.1016/j.ic.2003.08.002
28. Schmidt-Schauß, M., Schulz, K.U.: On the exponent of periodicity of minimal solutions of context equations. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 61–75. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0052361
29. Schmidt-Schauß, M., Schulz, K.U.: Solvability of context equations with two context variables is decidable. J. Symb. Comput. **33**(1), 77–122 (2002). https://doi.org/10.1006/jsco.2001.0438
30. Schmidt-Schauß, M., Schulz, K.U.: ~~Decidability~~ of bounded higher-order unification. J. Symb. Comput. **40**(2), 905–954 (2005). https://doi.org/10.1016/j.jsc.2005.01.005
31. Schulz, K.U.: Makanin's algorithm for word equations-two improvements and a generalization. In: Schulz, K.U. (ed.) IWWERT 1990. LNCS, vol. 572, pp. 85–150. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55124-7_4
32. Treinen, R.: The first-order theory of linear one-step rewriting is undecidable. Theor. Comput. Sci. **208**(1–2), 179–190 (1998). https://doi.org/10.1016/S0304-3975(98)00083-8
33. Vorobyov, S.: The first-order theory of one step rewriting in linear Noetherian systems is undecidable. In: Comon, H. (ed.) RTA 1997. LNCS, vol. 1232, pp. 254–268. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-62950-5_76
34. Vorobyov, S.: ∀∃*-Equational theory of context unification is Π^0_1-hard. In: Brim, L., Gruska, J., Zlatuška, J. (eds.) MFCS 1998. LNCS, vol. 1450, pp. 597–606. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055810