


Solving Word Equations (And Other Unification Problems) by Recompression

Artur Jeż 

University of Wrocław, Poland

<http://www.ii.uni.wroc.pl/~aje>

aje@cs.uni.wroc.pl

Abstract

In word equation problem we are given an equation $u = v$, where both u and v are words of letters and variables, and ask for a substitution of variables by words that equalizes the sides of the equation. This problem was first solved by Makanin and a different solution was proposed by Plandowski only 20 years later, his solution works in PSPACE, which is the best computational complexity bound known for this problem; on the other hand, the only known lower-bound is NP-hardness. In both cases the algorithms (and proofs) employed nontrivial facts on word combinatorics.

In the paper I will present an application of a recent technique of recompression, which simplifies the known proofs and (slightly) lowers the complexity to linear nondeterministic space. The technique is based on employing simple compression rules (replacement of two letters ab by a new letter c , replacement of maximal repetitions of a by a new letter), and modifying the equations (replacing a variable X by bX or Xa) so that those operations are sound and complete. In particular, no combinatorial properties of strings are used.

The approach turns out to be quite robust and can be applied to various generalizations and related scenarios (context unification, i.e. equations over terms; equations over traces, i.e. partially ordered words; ...).

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Formalisms; Theory of computation → Grammars and context-free languages; Theory of computation → Design and analysis of algorithms; Theory of computation → Tree languages

Keywords and phrases word equation, context unification, equations in groups, compression

Digital Object Identifier 10.4230/LIPIcs.CSL.2020.3

Category Invited Talk

Funding Work supported under National Science Centre, Poland project number 2017/26/E/ST6/00191.

1 Introduction

1.1 Word equations

The word equation problem, i.e. solving equations in the algebra of words, was first investigated by Markov in the fifties. In this problem we get as an input an equation of the form

$$u = v$$

where u and v are strings of letters (from a fixed alphabet) as well as variables and a *solution* is a substitution of words for variables that turns this formal equation into a true equality of strings of letters (over the same fixed alphabet). It is relatively easy to show a reduction of this problem to the Hilbert's 10-th problem, i.e. the question of solving systems of Diophantine equations. Already then it was generally accepted that Hilbert's 10-th problem is undecidable and Markov wanted to show this by proving the undecidability of word equations.



© Artur Jeż;

licensed under Creative Commons License CC-BY

28th EACSL Annual Conference on Computer Science Logic (CSL 2020).

Editors: Maribel Fernández and Anca Muscholl; Article No. 3; pp. 3:1–3:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

3:2 Solving Word Equations (And Other Unification Problems) by Recompression

Alas, while Hilbert's 10-th problem is undecidable, the word equation problem is *decidable*, which was shown by Makanin [36]. The termination proof of his algorithm is very complex and yields a relatively weak bound on the computational complexity, thus over the years several improvements and simplifications over the original algorithm were proposed [21, 56, 27, 19]. Simplifications have many potential advantages: it seems natural that simpler algorithm can be generalised or extended more easily (for instance, to the case of equations in groups) than a complex one. Moreover, simpler algorithm should be more effective in practical applications and should have a lower complexity bounds.

Subcases. It is easy to show NP-hardness for word equations, so far no better computational complexity lower bound is known. Such hardness stimulated a search for a restricted subclasses of the problem for which efficient (i.e. polynomial) algorithms can be given [2]. One of such subclasses is defined by restricting the amount of different variables that can be used in an equation: it is known that equations with one [13, 29] and two [2, 20, 12] variables can be solved in polynomial time. Already for three variables it is not known, whether they are in NP or not [50] and partial results require nontrivial analysis [50].

Generalisations. Since Makanin's original solution much effort was put into extending his algorithm to other structures. Three directions seemed most natural:

- adding constraints to word equations;
- equations in free groups;
- partial commutation;
- equations in terms.

Constraints From the application point of view, it is advantageous to consider word equations that can also use some additional constraints, i.e. we require that the solution for X has some additional properties. This was first done for regular constraints [56], on the other hand, for several types of constraints, for instance length-constraints, it is still open, whether the resulting problem is decidable or not (it becomes undecidable, if we allow counting occurrences of particular letter in the substitutions and arithmetic operations on such counts [1]).

Free groups From the algebraic point of view, the word equation problem is solving equations in a free semigroup. It is natural to try to extend an algorithm from the free semigroup also to the case of free groups and then perhaps even to a larger class of groups (observe, that there are groups and semigroups for which the word problem is undecidable). The first algorithm for the group case was given by Makanin [37, 38], his algorithm was not primitively-recursive [28]. Furthermore, Razborov showed that this algorithm can be used to give a description of all solutions of an equation [48] (more readable description of the Razborov's construction is available in [25]). As a final comment, note that such a description was the first step in proving the Tarski's Conjecture for free groups (that the theory of free groups is decidable) [26].

Partial commutation Another natural generalization is to allow partial commutation between the letters, i.e. for each pair of letters we specify, whether $ab = ba$ or not. Such partially commutative words are usually called traces, after Mazurkiewicz, and the corresponding groups are usually known as Right-Angled Artin Groups, RAAGs for short. Decidability for trace equations was shown by Matiyasevich [39] and for RAAGs by Diekert and Muscholl [11]. In both cases the main step in the proof was a reduction from a partially commutative case to a no-commutative one.

Terms We can view words as very simple terms: each letter is a function symbol of arity 1. In this way word equations are equations over (very simple) terms. It is known, that term unification can be decided in polynomial time, assuming that variables represent closed (full) terms [49]; thus such a problem is unlikely to generalise word equations.

A natural generalisation of term unification and word equations is a *second-order unification*, in which we allow variables to represent functions that take arguments (which need to be closed terms). However, it is known that this problem is undecidable, even in many restricted subcases [18, 14, 30, 32]. *Context unification* [4, 5, 51] is a natural problem “in between”: we allow variables representing functions, but we insist that they use their argument *exactly once*. It is easy to show that such defined problem generalises word equations, on the other hand, the undecidability proofs for second-order unification do not transfer directly to this model.

Being a natural generalisation is not enough to explain the interest in this problem, more importantly, context unification has natural connections with other, well-studied problems (equality up to constraints [40], linear second-order unification [33, 30], one-step term rewriting [41], bounded second order unification [53], ...). Unfortunately, for over two decades the question of decidability of context unification remained open. Despite intensive research, not much is known about the decidability of this problem: only results for some restricted subcases are known: [5, 52, 31, 30, 55, 54, 34, 17].

1.2 Compression and word equations

For more than 20 years since Makanin’s original solution there was very small progress in algorithms for word equations: the algorithm was improved in many places, in particular this lead to a better estimation of the running time; however, the main idea (and the general complexity of the proof) was essentially the same.

The breakthrough was done by Plandowski and Rytter [47], who, for the first time, used the compression to solve word equations. They showed, that the shortest solution (of size N) of the word equation (of size n) has an SLP¹ representation of size $\text{poly}(n, \log N)$; using the algorithm for testing the equality of two SLPs [43] this easily yields a (non-deterministic) algorithm running in time $\text{poly}(n, \log N)$. Unfortunately, this work did not provide any bound on N and the only known bound (4 times exponential in n) came directly from Makanin’s algorithm, together those two results yielded a 3NEXPTIME algorithm. Soon after the bound on the size of the shortest solution was improved to triply exponential [19], which immediately yielded an algorithm from class 2NEXPTIME, however, the same paper [19] improved Makanin’s algorithm, so that it worked in EXPSPACE.

Next, Plandowski gave a better (doubly exponential) bound on the size of the shortest solution [44] and thus obtained a NEXPTIME algorithm, in particular, at that time this was the best known algorithm for this problem. The proof was based on novel factorisations of words. By better exploiting the interplay between factorisations and compression, he improved the algorithm so that it worked in PSPACE [45].

It is worth mentioning, that the solution proposed by Plandowski is essentially different than the one given by Makanin. In particular, it allowed generalisations more easily: Diekert, Gutiérrez and Hagenah [8] showed, that Plandowski’s algorithm can be extended to the case

¹ A *Straight Line Programme* (SLP for short), is simply a context free grammar generating exactly one word.

in which we allow regular constraints in the equation (i.e. we want that the word substituted for X is from a regular language, whose description by a finite automaton is part of the input) and inversion; such an extended algorithm still works in polynomial space. It is easy to show that solving equations in free groups reduces to the above-mentioned problem of word equations with regular constraints and inversion [8] (it is worth mentioning, that in general we do not know whether solving equations in free groups is easier or harder than solving the ones in a free semigroup).

On the other hand, Plandowski showed, that his algorithm can be used to generate a finite representation of all solutions of a word equation [46], which allows solving several decision problems concerning the set of all solutions (finiteness, boundedness, boundedness of the exponent of periodicity etc.). It is not known, whether this algorithm can be generalised so that it generates all solutions also in the case of regular constraints and inversion (or in a free group).

The new, simpler algorithm for word equations and demonstration of connections between compression and word equations gave a new hope for solving the context unification problem. The first results were very promising: by using “tree” equivalents of SLPs computational complexity of some problems related to context unification was established [17, 31, 6]. Unfortunately, this approach failed to fully generalise Plandowski’s algorithm for words: the equivalent of factorisations that were used in the algorithm were not found for trees.

It is worth mentioning, that the approach proposed by Rytter and Plandowski, in which we compress a solution using SLPs (or in the non-deterministic case – we guess the compressed representation of the solution) and then perform the computation directly on the SLP-compressed representations using known algorithm that work in polynomial time, turned out to be extremely fruitful in many branches of computer science. The recent survey by Lohrey gives several such successful applications [35].

► **Remark.** As this is an informal survey presentations, most of the proofs are only sketched or omitted.

2 Recompression for word equations

We begin with a formal definition of the word equations problem: Consider a finite alphabet Σ and set of variables \mathcal{X} ; during the algorithm Σ will be extended by new letters, but it will always remain finite. Word equation is of a form “ $u = v$ ”, where $u, v \in (\Sigma \cup \mathcal{X})^*$ and its solution is a homomorphism $S : \Sigma \cup \mathcal{X} \mapsto \Sigma^*$, which is constant on Σ , that is $S(a) = a$, and satisfies the equation, i.e. words $S(u)$ and $S(v)$ are equal. By n we denote the size of the equation, i.e. $|u| + |v|$. The algorithm requires only small improvements so that it applies also to systems of equations, to streamline the presentation we will not consider this case.

Fix any solution S of the equation $u = v$, without loss of generality we can assume that this is the *shortest solution*, i.e. the one minimising $|S(u)|$; let N denote the *length of the solution*, that is $|S(u)|$. By the earlier work of Plandowski and Rytter [47] we know that $S(u)$ (and also $S(X)$ for each variable X) has an SLP (of size $\text{poly}(n, \log N)$), in fact the same conclusion can be drawn from the later works of Plandowski [44, 45, 46]. Regardless of the form of S and SLP, we know, that at least one of the productions in this SLP is of the form $c \rightarrow ab$, where c is a nonterminal of the SLP while $a, b \in \Sigma$ are letters. Let us “reverse” this production, i.e. replace in $S(u)$ all pairs of letters ab by c . It is relatively easy to formalise this operation for words, it is not so clear, what should be done in case of equations, so let us inspect the easier fragment first.

Algorithm 1 PairComp(ab, w) Compression of pair ab .

- 1: let $c \in \Sigma$ be an unused letter
 - 2: replace all occurrences of ab in w by c
-

Consider an explicitly given word w . Performing the “ ab -pair compression” on it is easy (we replace each pair ab by c), as long as $a \neq b$: replacing pairs aa is ambiguous, as such pairs can “overlap”. Instead, we replace *maximal blocks* of a letter a : block a^ℓ is *maximal*, when there is no letter a to left and to the right of it (in particular, there could be no letter at all).

Formally, the operations are defined as follows:

- *ab pair compression* For a given word w replace all occurrences of ab in w by a fresh letter c .
- *a block compression* For a given word w replace all occurrences of maximal blocks a^ℓ for $\ell > 1$ in w by fresh letters a_ℓ .

We always assume, that in the ab -pair compression the letters a and b are different.

Observe, that those operations are indeed “inverses” of SLP productions: replacing ab with c corresponds to a production $c \rightarrow ab$, similarly replacing a^ℓ with a_ℓ corresponds to a production $a_\ell \rightarrow a^\ell$.

Algorithm 2 BlockComp(a, w) Block compression for a .

- 1: **for** $\ell > 1$ **do**
 - 2: let $a_\ell \in \Sigma$ be an unused letter
 - 3: replace all maximal blocks a^ℓ in w by a_ℓ
-

Iterating the pair and blocks compression results in a compression of word w , assuming that we treat the introduced symbols as normal letters. There are several possible ways to implement such iteration, different results are obtained by altering the order of the compressions, exact treatment of new letters and so on. Still, essentially each “reasonable” variant works.

Observe, that if we compress two words, say w_1 and w_2 , in parallel then the resulting words w'_1 and w'_2 are equal if and only if w_1 and w_2 are. This justifies the usage of compression operations to both sides of the word equation in parallel, it remains to show, how to do that.

Let us fix a solution S , a pair ab (where $a \neq b$); consider how does a particular occurrence of ab got into $S(u)$.

► **Definition 1.** For an equation $u = v$, solution S and pair ab an occurrence of ab in $S(u)$ (or $S(v)$) is

- explicit, if it consists solely of letters coming from u (or v);
- implicit, if it consists solely of letters coming from a substitution $S(X)$ for a fixed occurrence of some variable X ;
- crossing, otherwise.

A pair ab is crossing (for a solution S) if it has at least one crossing occurrence and non-crossing (for a solution S) otherwise.

We similarly define explicit, implicit and crossing occurrences for blocks of letter a ; a is crossing, if at least one of its blocks has a crossing occurrence. (In other words: aa is crossing).

3:6 Solving Word Equations (And Other Unification Problems) by Recompression

► **Example 2.** Equation

$$aaXbbabababa = XaabbYabX$$

has a unique solution $S(X) = a$, $S(Y) = abab$, under which sides evaluate to

$$aaabbabababa = aaabbabababa .$$

Pair ba is crossing (as the first letter of $S(Y)$ is a and first Y is preceded by a letter b , moreover, the last letter of $S(Y)$ is b and the second Y is succeeded by a letter a), pair ab is non-crossing. Letter b is non-crossing, letter a is crossing (as X is preceded by a letter a on the left-hand side of the equation and on the right-hand side of the equation X is succeeded by a letter a).

■ **Algorithm 3** $\text{PairComp}(ab, 'u = v')$ Pair compression for ab in an equation $u = v$.

-
- 1: let $c \in \Sigma$ be a fresh letter
 - 2: replace all occurrences of ab in ' $u = v$ ' by c
-

■ **Algorithm 4** $\text{BlockComp}(a, 'u = v')$ Block compression for a letter a in an equation ' $u = v$ '.

-
- 1: **for** $\ell > 1$ **do**
 - 2: let $a_\ell \in \Sigma$ be a fresh letter
 - 3: replace all occurrences of maximal blocks a^ℓ in ' $u = v$ ' by a_ℓ
-

Fix a pair ab and a solution S of the equation $u = v$. If ab is non-crossing, performing $\text{PairComp}(ab, S(u))$ is easy: we need to replace every explicit occurrence (which we do directly on the equation) as well as each implicit occurrence, which is done “implicitly”, as the solution is not stored, nor written anywhere. Due to the similarities to PairComp we will simply use the name $\text{PairComp}(ab, 'u = v')$, when we make the pair compressions on the equation. The argument above shows, that if the equation had a solution for which ab is non-crossing then also the obtained equation has a solution. The same applies to the block compression, called $\text{BlockComp}(a, 'u = v')$ for simplicity. On the other hand, if the obtained equation has a solution, then also the original equation had one (this solution is obtained by replacing each letter c by ab , the argument for the block compressions the same).

► **Lemma 3.** *Let the equation $u = v$ have a solution S , such that ab is non-crossing for S . Then $u' = v'$ obtained by $\text{PairComp}(ab, 'u = v')$ is satisfiable.*

If the obtained equation $u' = v'$ is satisfiable, then also the original equation $u = v$ is.

The same applies to $\text{BlockComp}(a, 'u = v')$.

Unfortunately Lemma 3 is not enough to simulate $\text{Compression}(w)$ directly on the equation: In general there is no guarantee that the pair ab (letter a) is non-crossing, moreover, we do not know what are the pairs that have only implicit occurrences. It turns out, that the second problem is trivial: if we restrict ourselves to the shortest solutions then every pair that has an implicit occurrence has also a crossing or explicit one, a similar statement holds also for blocks of letters.

► **Lemma 4** ([47]). *Let S be a shortest solution of an equation ' $u = v$ '. Then:*

- *If ab is a substring of $S(u)$, where $a \neq b$, then a , b have explicit occurrences in the equation and ab has an explicit or crossing occurrence.*
- *If a^k is a maximal block in $S(u)$ then a has an explicit occurrence in the equation and a^k has an explicit or crossing occurrence.*

The proof is simple: suppose that a pair (block) has only implicit occurrences. Then we could remove them and the obtained solution is shorter, contradicting the assumption.

Getting back to the crossing pairs (and blocks), if we fix a pair ab (letter a), then it is easy to “uncross” it: by Definition 1 we can conclude that the pair ab is crossing if and only if for some variables X and Y (not necessarily different) one of the following conditions holds (we assume that the solution does not assign an empty word to any variable – otherwise we could simply remove such a variable from the equation):

- (CP1) aX occurs in the equation and $S(X)$ begins with b ;
- (CP2) Yb occurs in the equation and $S(Y)$ ends with a ;
- (CP3) YX occurs in the equation, $S(X)$ begins with b while $bS(Y)$ ends with a .

In each of these cases the “uncrossing” is natural: in (CP1) we “pop” from X a letter b to the left, in (CP2) we pop a to the right from Y , in (CP3) we perform both operations. It turns out that in fact we can be even more systematic: we do not have to look at the occurrences of variables, it is enough to consider the first and last letter of $S(X)$ for each variable X :

- If $S(X)$ begins with b then we replace X with bX (changing implicitly the solution $S(X) = bw$ to $S'(X) = w$), if in the new solution $S(X) = \epsilon$, i.e. it is empty, then we remove X from the equation;
- if $S(X)$ ends with a then we apply a symmetric procedure.

Such an algorithm is called **Pop**.

■ **Algorithm 5** $\text{Pop}(a, b, 'u = v')$.

```

1: for  $X$ : variable do
2:   if the first letter of  $S(X)$  is  $b$  then ▷ Guess
3:     replace every  $X$  w  $'u = v'$  by  $bX$ 
▷ Implicitly change solution  $S(X) = bw$  to  $S(X) = w$ 
4:     if  $S(X) = \epsilon$  then ▷ Guess
5:       remove  $X$  from  $u$  and  $v$ 
6:     ... ▷ Perform a symmetric operation for the last letter and  $a$ 

```

It is easy to see, that for appropriate non-deterministic choices the obtained equation has a solution for which ab is non-crossing: for instance, if aX occurs in the equation and $S(X)$ begins with b then we make the corresponding non-deterministic choices, popping b to the left and obtaining abX ; a simple proof requires a precise statement of the claim as well as some case analysis.

► **Lemma 5.** *If the equation $'u = v'$ has a solution S then for an appropriate run of $\text{Pop}(a, b, 'u = v')$ (for appropriate non-deterministic choices) the obtained equation $u' = v'$ has a corresponding solution S' , i.e. $S(u) = S'(u')$, for which ab is a non-crossing pair.*

If the obtained equation has a solution then also the original equation had one.

Thus, we know how to proceed with a crossing ab -pair compression: we first turn ab into a non-crossing pair (**Pop**) and then compress it as a non-crossing pair (**PairComp**).

We would like to perform similar operations also for block compression. For non-crossing blocks we can naturally define a similar algorithm $\text{BlockComp}(a, 'u = v')$. It remains to show how to “uncross” a letter a . Unfortunately, if aX occurs in the equation and $S(X)$ begins with a then replacing X with aX is not enough, as $S(X)$ may still begin with a . In such a case we iterate the procedure until the first letter of X is not a (this includes the case in

3:8 Solving Word Equations (And Other Unification Problems) by Recompression

which we remove the whole variable X). Observe, that instead of doing this letter by letter, we can uncross a in one step: it is enough to remove from variable X its whole a -prefix and a -suffix of $S(X)$ (if $w = a^\ell w' a^r$, where w' does not begin nor end with a , a -prefix w is a^ℓ and a -suffix is a^r ; if $w = a^\ell$ then a -suffix is empty). Such an algorithm is called **CutPrefSuff**.

■ **Algorithm 6** **CutPrefSuff**($a, 'u = v'$) Popping prefixes and suffixes.

```

1: for  $X$ : variable do
2:   guess the lengths  $\ell, r$  of  $a$ -prefix and suffix of  $S(X)$            ▷  $S(X) = a^\ell w a^r$ 
                                                                    ▷ If  $S(X) = a^\ell$  then  $r = 0$ 
3:   replace occurrences of  $X$  in  $u$  and  $v$  by  $a^\ell X a^r$    ▷  $a^\ell, a^r$  are stored in a compressed
   way
4:                                     ▷ Implicitly change the solution  $S(X) = a^\ell w b^r$  to  $S(X) = w$ 
5:   if  $S(X) = \epsilon$  then                                           ▷ Guess
6:     remove  $X$  from  $u$  and  $v$ 

```

Similarly as in **Pop** we can show that after an appropriate run of **CutPrefSuff** the obtained equation has a (corresponding) solution for which a is non-crossing. Unfortunately, there is another problem: we need to write down the lengths ℓ and r of a -prefixes and suffixes. We can write them as binary numbers, in which case they use $\mathcal{O}(\log \ell + \log r)$ bits of memory. However in general those still can be arbitrarily large numbers. Fortunately, we can show that in *some* solution those values are at most exponential (and so their description is polynomial-size). This easily follows from the exponential bound on exponent of periodicity [27]. For the moment it is enough that we know that:

► **Lemma 6** ([27]). *In the shortest solution of the equation ' $u = v$ ' each a -prefix and a -suffix has at most exponential length (in terms of $|u| + |v|$).*

Thus in **Pop** we can restrict ourselves to a -prefixes and suffixes of at most exponential length.

► **Lemma 7.** *Let S be a shortest solution of ' $u = v$ '. After some run of **CutPrefSuff**($a, 'u = v'$) (for appropriate non-deterministic choices) the obtained equation ' $u' = v'$ ' has a corresponding solution S' , such that $S'(u') = S(u)$, and a is a non-crossing letter for S' , moreover, the explicit a blocks in ' $u' = v'$ ' have at most exponential length.*

If the obtained equation has a solution then also the original equation had one.

After **Pop** we can compress a -blocks using **BlockComp**($a, 'u = v'$), observe that afterwards long a -blocks are replaced with single letters.

We are now ready to simulate **Compression** directly on the equation. The question is, in which order we should compress pairs and blocks? We make the choice nondeterministically: if there are any non-crossing pairs or letters, we compress them. This is natural, as such compression decreases both the size of the equation and the size of the length-minimal solution of the equation. If all pairs and letters are crossing, we choose greedily, i.e. the one that leads to the smallest equation (in one step). It is easy to show that such a strategy keeps the equation quadratic, more involved strategy, in which we compress many pairs/blocks in parallel, leads to a linear-length equation.

Call one iteration of the main loop a *phase*.

The correctness of the algorithm follows from the earlier discussion on the correctness of **BlockComp**, **CutPrefSuff**, **PairComp** and **Pop**. In particular, the length of the length-minimal solution drops by at least 1 in each iteration, thus the algorithm terminates.

■ **Algorithm 7** WordEqSAT Deciding the satisfiability of word equations.

```

1: while  $|u| > 1$  or  $|v| > 1$  do                                ▷ The equation is nontrivial
2:    $L \leftarrow$  list of letters in  $u, v$                             ▷ Occurring in the equation
3:   Choose a pair  $ab \in P^2$  or a letter  $a \in P$ 
4:   if it is crossing then
5:     uncross it
6:     compress it
7: Solve the problem naively                                ▷ The problem is simple when both sides have length 1

```

► **Lemma 8.** *Algorithm WordEqSAT has $\mathcal{O}(N)$ phases, where N is the length of the shortest solution of the input equation.*

Let us try to bound the space needed by the algorithm: we claim that for appropriate nondeterministic choices the stored equation has at most $8n^2$ letters (and n variables). To see this, observe first that each **Pop** introduces at most $2n$ letters, one at each side of the variable. The same applies to **CutPrefSuff** (formally, **CutPrefSuff** introduces long blocks but they are immediately replaced with single letters, and so we can think that in fact we introduce only $2n$ letters). By (CP1)–(CP3) we know that there are at most $2n$ crossing pairs and crossing letters (as each crossing pair / each crossing letter corresponds to one occurrence of a variable and one “side” of such an occurrence). If the equation has m letters (and at most n occurrences of variables) and there is an occurrence of a non-crossing pair or block then we choose it for compression. Otherwise, there are m letters in the equation and each is covered by at least one pair/block, so for one of $2n$ choice at least $\frac{m}{2n}$ letters is covered, so at least $\frac{m}{4n}$ letters are removed. Thus the new equation has at most

$$\underbrace{m}_{\text{previous}} - \underbrace{\frac{m}{4n}}_{\text{removed}} + \underbrace{2n}_{\text{popped}} \leq 8n^2 - 2n + 2n = 8n^2,$$

where the inequality follows by the inductive assumption that $m \leq 8n^2$. Going for the bit-size, each symbol requires at most logarithmic number of bits, and so

► **Lemma 9.** *WordEqSAT runs in $\mathcal{O}(n^2 \log n)$ space.*

With some effort we can make the above if analysis much tighter:

► **Theorem 10** ([24]). *The recompression based algorithm (nondeterministically) decides word equations problem in $\mathcal{O}(n \log n)$ bit-space; moreover, the stored equation has linear length.*

As a reminder: a PSPACE algorithm for this problem is already known [45]. Its memory consumption is not stated explicitly in that work, however, it is much larger than $\mathcal{O}(n \log n)$: the stored equations are of length $\mathcal{O}(n^3)$ and during the transformations the algorithm uses essentially more memory.

3 Similar applications

Generating a representation of all solutions

So far we have only considered the satisfiability of word equations. In general, there can be many solutions of such an equation and it is desirable to have a (finite) representation of

3:10 Solving Word Equations (And Other Unification Problems) by Recompression

all of them. The first such description was given by Plandowski [46], his algorithm works in PSPACE and generates an exponential representation of all solutions. We show that a similar description can be created using the recompression approach. It is easy to believe that the compression of pairs and blocks “preserves” the set of solutions: if S is a solution of an equation $u = v$ then we can compress the pairs (or blocks) in the word $S(u)$ and simulate such a compression directly on the equation $u = v$ obtaining $u' = v'$ with a “corresponding” solution $S'(u')$. In this way we naturally obtain a graph: its nodes are labelled with equations and an edge between equations $u = v$ and $u' = v'$ will describe how to transform the solutions of $u' = v'$ into solutions of $u = v$ (note that a node labelled with $u = v$ can have several edges to many other nodes). The mentioned description is fairly natural: we replace a letter c by a pair ab or replace a_ℓ with a^ℓ or prepend or append some letters to $S(X)$. It remains to guarantee that both the nodes and the edges have reasonable size description (in our case: polynomial).

Unfortunately, there is a problem: consider an equation $aXXXX = XaYY$, it has solutions of the form $S(X) = a^{\ell_X}$, $S(Y) = a^{\ell_Y}$, where additionally $4\ell_X + 1 = \ell_X + 1 + 2\ell_Y$. There are infinitely many such solutions and replacing X by a^{ℓ_X} during the CutPrefSuff can use arbitrarily large memory and transform this equation into an infinite number of other equations. On the other hand, as a next step we replace a -blocks of length $4\ell_X + 1 = \ell_X + 1 + 2\ell_Y$ by a new letter and the precise length of those blocks is unimportant, what matters is that they are of the same length. In general, we can improve the block compression so that it uses numerical parameters (for lengths) instead of concrete values of prefixes and suffixes. As a first step, in CutPrefSuff when we pop an a -prefix of length a^{ℓ_X} from X , the ℓ_X is not a number, but rather a numerical parameter, the same applies to the a -suffix r_X . Next we (non-deterministically) identify maximal blocks of the same length and verify, whether indeed such blocks can be of equal length. The guessed equalities correspond to a system of linear Diophantine equations. Moreover, each solution of such a system corresponds to a solution of the word equation and vice-versa. In this way we no longer need to consider large numbers and long equations and can guarantee that the considered equations are always of polynomial length (observe that this modification in fact removes the necessity of using the bound on the Σ -exponent of periodicity). Unfortunately, as a side effect we get that the edges in our graph representation of all solutions are labelled with systems of linear Diophantine equations and each solution of such a system corresponds to one transformation of the solution of the word equation.

► **Theorem 11** ([24]). *Using the recompression based algorithm we can compute (in PSPACE) a finite graph representation of all solutions of a word equation. Each node and edge have a polynomial description, the whole graph has at most exponential number of nodes and edges.*

As in the case of the decision variant, the recompression-based algorithm has much lower space consumption, than previously known [46], the same applies to the size of the constructed representation.

The above characterization is combinatorial in nature. On the other hand, it is natural to characterize the class of languages that can be obtained as sets of solutions of a word equation. For instance, the question of whether it is an indexed language, in the sense of Aho, was posed a long time ago. Using extensions of the recompression technique and interpreting it in the algebraic setting it can be shown that the language of all solutions of a given word equation is an EDTOL language [3] (so, in particular, an index language). This is by no means an easy task, in particular, the block compression needs to be redesigned essentially from scratch.

► **Theorem 12** ([3]). *The set of solutions of a given word equation is an EDTOL language.*

Equations with one variable

As already mentioned, one of the investigated subclasses of word equations are the equations with one variable. It is easy to show that they can be decided in $\mathcal{O}(n^3)$ and improving to quadratic running time requires only a couple of observations [7]. The first nontrivial algorithm for this problem had an $\mathcal{O}(n \log n)$ running time [42], while Dąbrowski and Plandowski gave an algorithm with $\mathcal{O}(n + \#_X \log n)$ running time [13], where $\#_X$ denotes the number of occurrences of a variable X in the original equation.

It is easy to see that the recompression based algorithm becomes deterministic in case of one variable equations: it makes the following non-deterministic choices:

- What is the first (last) letter of $S(X)$?
- What is the length of the a -prefix a^ℓ (suffix a^r) of $S(X)$?

When the equation has only one variable, answers to both of those questions can be easily obtained from the equation.

► **Lemma 13.** *Without loss of generality word equation with one variable are of the form*

$$A_0 X A_1 \dots A_{k-1} X A_k = X B_1 \dots B_{\ell-1} X B_\ell, \quad (1)$$

where A_0 is a non-empty word and exactly one of the words A_k, B_ℓ is empty.

Let the first letter of A_0 be a . Each solution $S(X) \notin a^+$ has the same a -prefix as A_0 ; symmetric fact holds also for a -suffixes.

Lemma 13 yields a simple recompression based algorithm: in each phase we identify candidate solutions from a^+ , where a is the first letter of A_0 , and verify whether indeed such a candidate is a solution. Next we perform recompression: all remaining solutions have the same a -prefix (and suffix).

A natural implementation of this algorithm has the same running time as the algorithm by Dąbrowski and Plandowski, i.e. $\mathcal{O}(n + \#_X \log n)$. It is possible to improve the running time to linear, this requires several non-trivial improvements of the algorithm and usage of efficient data structures (suffix arrays with a structure for computing the *longest common prefix*, i.e. lcp). In particular:

- Instead of one equation the algorithm actually stores a system of equations and sometimes splits one equation into two smaller ones, in this way we save space.
- We keep track, which words are the same and for set of identical words we store one copy and represent all of them by pointers.
- We prove that for a certain class of solutions the algorithm reports such solutions within $\mathcal{O}(1)$ phases. In many places of the proofs we show that the corresponding solution is from this class.
- We improve the testing procedure: some of the candidate solutions are rejected based on their structural properties, moreover we use a much more precise cost analysis: we calculate for each word separately, whether it took part in a particular test or not. In this way we can establish that some tests took less time than linear (which is the time needed for reading the whole equation).

► **Theorem 14** ([23]). *Using a recompression based algorithm we can in linear time return all solutions of a word equation with one variable.*

Equations with regular constraints and inversion; equations in free groups

As already mentioned, it is natural and important to extend the word equations by regular constraints and inversion, in particular this leads to an algorithm for equations in free groups [8] (the reduction between those two problems is fully syntactical and does not depend on the particular algorithm for solving word equations). Note that it is unknown, whether the algorithm generating a representation of all solutions can be also extended by regular constraints and inversion. Thus the only previously known algorithm for representation of all solutions of an equation in a free group was due to Razborov [48], and it was based on Makanin's algorithm for word equations in free groups.

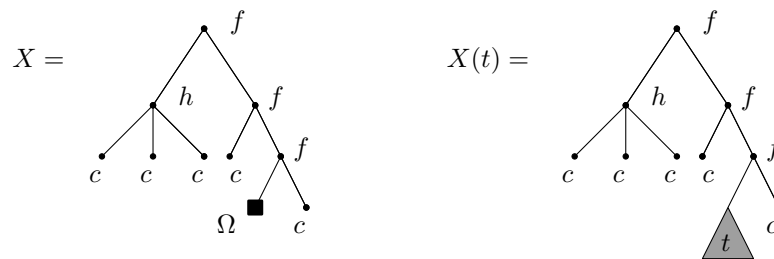
Adding the regular constraints to the recompression based algorithm WordEqSAT is fairly standard: We can encode all constraints using one non-deterministic finite automaton (the constraints for particular variables differ only in the set of accepting states). For each letter c we store its *transition function*, i.e. a function $f_c : Q \mapsto 2^Q$, which says that the automaton in state q after reading a letter c reaches a state in $f_c(q)$. This function is naturally extended to words: it still defines which states can be reached from q after reading w . Formally $f_{wa} = (f_w \circ f_a)(q) = \{p \mid \exists q' \in f_w(q) \text{ i } p \in f_a(q')\}$ for a letter a . If we introduce a new letter c (which replaces a word w) then we naturally define the transition function $f_c \leftarrow f_w$. We can express the regular constraints in terms of this function: saying that $S(X)$ is accepted by an automaton means that $f_{S(X)}(q_0)$ is one of the accepting states. So it is enough to guess the value of $f_{S(X)}$ which satisfies this condition; in this way we can talk about the value f_X for a variable X . Popping letters from a variable means that we need to adjust the transition function, i.e. when we replace X by aX then $f_X = f_a \circ f_{X'}$, we similarly define \bar{f}_X when we pop letters to the right.

More problems are caused by the *inversion*: intuitively it corresponds to taking the inverse element in the group and on the semigroup level we this is simulated by requiring that $\bar{\bar{a}} = a$ for each letter a and $\bar{a_1 a_2 \dots a_m} = \bar{a_m} \dots \bar{a_2} \bar{a_1}$. This has an impact on the compression: when we compress a pair ab to c , then we should also replace $\bar{ab} = \bar{b}\bar{a}$ by a letter \bar{c} . At the first sight this looks easy, but becomes problematic, when those two pairs are not disjoint, i.e. when $\bar{a} = a$ (or $\bar{b} = b$); in general we cannot exclude such a case and if it happens, in a sequence $ba\bar{b}$ during the pair compression for ba we want to simultaneously replace ba and $a\bar{b}$, which is not possible. Instead, we replace maximal fragments that can be fully covered with pairs ab or $\bar{b}\bar{a}$, in this case this: the whole triple $ba\bar{b}$. In the worst case (when $a = \bar{a}$ and $b = \bar{b}$) we need to replace whole sequences of the form $(ab)^n$, which is a common generalisation of both pairs and blocks compression.

As in the case of semigroups, this representation can be interpreted in the algebraic setting, which is even more natural, and can be used to show that the set of solutions is an EDTOL language.

► **Theorem 15** ([10], [3]). *A recompression based algorithm generates in polynomial space the description of all solutions of a word equation in free semigroups with inversion and regular constraints. This in particular provides a similar description in case of free groups with regular constraints and shows that the set of solutions is an EDTOL language.*

Trace equations. For our purposes it is better to view partially commuting words, i.e. traces, in terms of resources of letters: we equip letters of the alphabet Σ with resources, formally there is a function $\rho : \Sigma \rightarrow R$, where R is some finite set. Then two different letters commute if and only if they do not share a resource, i.e. $ab = ba$ for $a \neq b$ if and only if $\rho(a) \cap \rho(b) = \emptyset$. It is easy to see the equivalence of words with resources and traces.



■ **Figure 1** A context and the same context applied on an argument.

It is difficult to apply compression operations in trace equation to letters of different resources. On the other hand, when the set of resources of some letters are the same, they behave exactly like ordinary non-commuting words and the recompression approach can be applied to them. A natural approach to solving trace equation using recompression involves also another operation of “lifting” letters, i.e. increasing the set of resources of a letter. In this way the trace is partially “linearized”, as part of the commutation is removed.

It turns out that this approach can be implemented, and the algorithm alternates the lifting and compression operations, which is in contrast to previous approaches to trace equations, which linearized the trace once at the beginning. In particular, the results concerning the involution, regular constraints and equations in the corresponding groups, which are the well-known Right-angled Artin groups, also generalize to traces. The details are rather technical and are beyond the scope of a survey aimed at presenting recompression technique.

► **Theorem 16** ([9]). *The set of solutions of trace equation (with involution and regular constraints) is an EDTOL, its nonemptiness can be decided in PSPACE.*

The same results holds also for Right-angled Artin Groups.

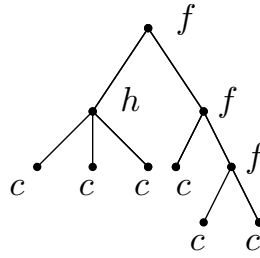
Context unification

Recall that the context unification is a generalisation of word equations to the case of terms. What type of equations we would like to consider? Clearly we consider terms over a fixed signature (which is usually part of the input), and allow occurrences of constants and variables. If we allow only that the variables represent full terms, then the satisfiability of such equations is decidable in polynomial time [49] and so probably does not generalise the word equations (which are NP-hard). This is also easy to observe when we look closer at a word equation: the words represented by the variables can be concatenated at both ends, i.e. they represent terms with a missing argument.

We arrive at a conclusion that our generalisation should use variables *with arguments*, i.e. the (second-order) variables take an argument that is a full term and can use it, perhaps several times. Such a definition leads to a *second-order unification*, which is known to be undecidable even in very restricted subcases [18, 14, 30, 32].

Thus we would like to have a subclass of second order unification that still generalises word equations. In order to do that we put additional restriction on the solutions: each argument can be used by the term *exactly once*. Observe that this still generalise the word equations: using the argument exactly once naturally corresponds to concatenation.

Formally, in the context unification problem [4, 5, 51], we consider an equation $u = v$ in which we use variables (representing closed terms), which we denote by letters x, y , as well as context variables (representing terms with one “hole” for the argument, they are usually



■ **Figure 2** Term $f(h(c, c, c), f(c, f(c, c)))$ represented as a tree, f is of arity 2, h arity 3, while c : 0.

called *contexts*), which we denote by letters X, Y . Syntactically, u and v are terms that use letters from signature Σ (which is part of the input), variables and context variables, the former are treated as symbols of arity 0, while the latter as symbols of arity 1. A *substitution* S assigns to each variable a closed term over Σ and to each context variable it assigns a *context*, i.e. a term over $\Sigma \cup \{\Omega\}$ in which the special symbol Ω has arity 0 and is used exactly once. (Intuitively it corresponds to a place in which we later substitute the argument). S is extended to u, v in a natural way, note that for a context variable X the term $S(X(t))$ is obtained by replacing in $S(X)$ the unique symbol Ω by $S(t)$. A *solution* is a substitution satisfying $S(u) = S(v)$.

► **Example 17.** Consider a signature $\{f, c, c'\}$, where f has arity 2 while c, c' have arity 0 and consider an equation $X(c) = Y(c')$, where X and Y are context variables. The equation has a solution $S(X) = f(\Omega, c'), S(Y) = f(c, \Omega)$ and then $S(X(c)) = f(c, c') = S(Y(c'))$.

We try to apply the main idea of the recompression also in the case of terms: we iterate local compression operations and we guarantee that the word (term) equation is polynomial size. Since several term problems were solved using compression-based methods [17, 31, 6, 15, 16], there is a reasonable hope that our approach may succeed.

Pair and block compression easily generalise to sequences of letters of arity 1 (we can think of them as words), unfortunately, there is no guarantee that a term has even one such letter. Intuitively, we rather expect that it has mostly leaves and symbols of larger arity. This leads us to another local compression operation: *leaf compression*. Consider a node labelled with f and its i -th child that is a leaf. We want to compress f with this child, leaving other children (and their subtrees) unchanged. Formally, given f of arity at least 1, position $1 \leq i \leq \text{ar}(f)$ and a letter c of arity 0 the $\text{LeafComp}(f, i, c, t)$ operation (*leaf compression*) replaces in term t nodes labelled with f and subterms $t_1, \dots, t_{i-1}, c, t_{i+1}, \dots, t_{\text{ar}(f)}$ (where c and position i are fixed, while other terms $t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_{\text{ar}(f)}$ – varying) by a term labelled with f' and subterms $t'_1, \dots, t'_{i-1}, t'_{i+1}, \dots, t'_{\text{ar}(f)}$ that are obtained by applying recursively LeafComp to terms $t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_{\text{ar}(f)}$; in other words, we first change the label from f to f' and then remove the i -th child, which has a label c and we apply such a compression to all occurrences of f and c in parallel.

The notion of crossing pair generalizes to this case in a natural way and the uncrossing replaces a term variable with a constant or replaces $X(t)$ with $X(f(x_1, \dots, x_i, t, x_{i+1}, \dots, x_\ell))$. Note that this introduces new variables.

Now the whole algorithm looks similar as in the case of word equations, we simply use additional compression operation. However, the analysis is much more involved, as the new uncrossing introduces fresh term variables. However, their number at any point can be linearly bounded and the polynomial upper-bound follows.

► **Theorem 18** ([22]). *Recompression based algorithm solves context unification in non-deterministic polynomial space.*

References

- 1 Julius Richard Büchi and Steven Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. *Mathematical Logic Quarterly*, 34(4):337–342, 1988. doi:10.1002/malq.19880340410.
- 2 Witold Charatonik and Leszek Pacholski. Word equations with two variables. In *IWWERT*, pages 43–56, 1991. doi:10.1007/3-540-56730-5_30.
- 3 Laura Ciobanu, Volker Diekert, and Murray Elder. Solution sets for equations over free groups are EDT0L languages. *IJAC*, 26(5):843–886, 2016. doi:10.1142/S0218196716500363.
- 4 Hubert Comon. Completion of rewrite systems with membership constraints. Part I: Deduction rules. *J. Symb. Comput.*, 25(4):397–419, 1998. doi:10.1006/jsc.1997.0185.
- 5 Hubert Comon. Completion of rewrite systems with membership constraints. Part II: Constraint solving. *J. Symb. Comput.*, 25(4):421–453, 1998. doi:10.1006/jsc.1997.0186.
- 6 Carles Creus, Adria Gascón, and Guillem Godoy. One-context unification with STG-compressed terms is in NP. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12)*, volume 15 of *LIPICs*, pages 149–164, Dagstuhl, Germany, 2012. Schloss Dagstuhl — Leibniz Zentrum fuer Informatik. doi:10.4230/LIPICs.RTA.2012.149.
- 7 Volker Diekert. Makanin’s Algorithm. In M. Lothaire, editor, *Algebraic Combinatorics on Words*, chapter 12, pages 342–390. Cambridge University Press, 2002.
- 8 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. doi:10.1016/j.ic.2005.04.002.
- 9 Volker Diekert, Artur Jež, and Manfred Kufleitner. Solutions of word equations over partially commutative structures. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *ICALP*, volume 55 of *LIPICs*, pages 127:1–127:14. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPICs.ICALP.2016.127.
- 10 Volker Diekert, Artur Jež, and Wojciech Plandowski. Finding all solutions of equations in free groups and monoids with involution. *Inf. Comput.*, 251:263–286, 2016. doi:10.1016/j.ic.2016.09.009.
- 11 Volker Diekert and Anca Muscholl. Solvability of equations in free partially commutative groups is decidable. *International Journal of Algebra and Computation*, 16:1047–1070, 2006. Conference version in Proc. ICALP 2001, 543–554, LNCS 2076. doi:10.1142/S0218196706003372.
- 12 Robert Dąbrowski and Wojciech Plandowski. Solving two-variable word equations. In *ICALP*, pages 408–419, 2004. doi:10.1007/978-3-540-27836-8_36.
- 13 Robert Dąbrowski and Wojciech Plandowski. On word equations in one variable. *Algorithmica*, 60(4):819–828, 2011. doi:10.1007/s00453-009-9375-3.
- 14 William M. Farmer. Simple second-order languages for which unification is undecidable. *Theor. Comput. Sci.*, 87(1):25–41, 1991. doi:10.1016/S0304-3975(06)80003-4.
- 15 Adria Gascón, Guillem Godoy, and Manfred Schmidt-Schauß. Context matching for compressed terms. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 93–102. IEEE Computer Society, 2008. doi:10.1109/LICS.2008.17.
- 16 Adria Gascón, Guillem Godoy, and Manfred Schmidt-Schauß. Unification and matching on compressed terms. *ACM Trans. Comput. Log.*, 12(4):26, 2011. doi:10.1145/1970398.1970402.
- 17 Adria Gascón, Guillem Godoy, Manfred Schmidt-Schauß, and Ashish Tiwari. Context unification with one context variable. *J. Symb. Comput.*, 45(2):173–193, 2010. doi:10.1016/j.jsc.2008.10.005.
- 18 Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theor. Comput. Sci.*, 13:225–230, 1981. doi:10.1016/0304-3975(81)90040-2.
- 19 Claudio Gutiérrez. Satisfiability of word equations with constants is in exponential space. In *FOCS*, pages 112–119, 1998. doi:10.1109/SFCS.1998.743434.
- 20 Lucian Ilie and Wojciech Plandowski. Two-variable word equations. *ITA*, 34(6):467–501, 2000. doi:10.1051/ita:2000126.

- 21 Joxan Jaffar. Minimal and complete word unification. *J. ACM*, 37(1):47–85, 1990.
- 22 Artur Jež. Context unification is in PSPACE. In Elias Koutsoupias, Javier Esparza, and Pierre Fraigniaud, editors, *ICALP*, volume 8573 of *LNCS*, pages 244–255. Springer, 2014. full version at <http://arxiv.org/abs/1310.4367>. doi:10.1007/978-3-662-43951-7_21.
- 23 Artur Jež. One-variable word equations in linear time. *Algorithmica*, 74:1–48, 2016. doi:10.1007/s00453-014-9931-3.
- 24 Artur Jež. Recompression: a simple and powerful technique for word equations. *J. ACM*, 63(1):4:1–4:51, March 2016. doi:10.1145/2743014.
- 25 Olga Kharlampovich and Alexei Myasnikov. Irreducible affine varieties over a free group. ii: Systems in triangular quasi-quadratic form and description of residually free groups. *Journal of Algebra*, 200:517–570, 1998.
- 26 Olga Kharlampovich and Alexei Myasnikov. Elementary theory of free non-abelian groups. *Journal of Algebra*, 302:451–552, 2006.
- 27 Antoni Kościelski and Leszek Pacholski. Complexity of Makanin’s algorithm. *J. ACM*, 43(4):670–684, 1996. doi:10.1145/234533.234543.
- 28 Antoni Kościelski and Leszek Pacholski. Makanin’s algorithm is not primitive recursive. *Theor. Comput. Sci.*, 191(1-2):145–156, 1998. doi:10.1016/S0304-3975(96)00321-0.
- 29 Markku Laine and Wojciech Plandowski. Word equations with one unknown. *Int. J. Found. Comput. Sci.*, 22(2):345–375, 2011. doi:10.1142/S0129054111008088.
- 30 Jordi Levy. Linear second-order unification. In Harald Ganzinger, editor, *RTA*, volume 1103 of *LNCS*, pages 332–346. Springer, 1996. doi:10.1007/3-540-61464-8_63.
- 31 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. doi:10.1093/jigpal/jzq010.
- 32 Jordi Levy and Margus Veanes. On the undecidability of second-order unification. *Inf. Comput.*, 159(1–2):125–150, 2000. doi:10.1006/inco.2000.2877.
- 33 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. doi:10.1007/10721975_11.
- 34 Jordi Levy and Mateu Villaret. Curryng second-order unification problems. In Sophie Tison, editor, *RTA*, volume 2378 of *LNCS*, pages 326–339. Springer, 2002. doi:10.1007/3-540-45610-4_23.
- 35 Markus Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.
- 36 Gennadii Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 2(103):147–236, 1977. (in Russian).
- 37 Gennadii Makanin. Equations in a free group. *Izv. Akad. Nauk SSR, Ser. Math.* 46:1199–1273, 1983. English transl. in *Math. USSR Izv.* 21 (1983).
- 38 Gennadii Makanin. Decidability of the universal and positive theories of a free group. *Izv. Akad. Nauk SSSR, Ser. Mat.* 48:735–749, 1984. In Russian; English translation in: *Math. USSR Izvestija*, 25, 75–88, 1985.
- 39 Yuri Matiyasevich. Some decision problems for traces. In Sergej Adian and Anil Nerode, editors, *LFCS*, volume 1234 of *LNCS*, pages 248–257. Springer, 1997. Invited lecture.
- 40 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. doi:10.1007/3-540-63104-6_4.
- 41 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. doi:10.3115/979617.979670.
- 42 S. Eyono Obono, Pavel Goralcik, and M. N. Maksimenko. Efficient solving of the word equations in one variable. In *MFCS*, pages 336–341, 1994. doi:10.1007/3-540-58338-6_80.

- 43 Wojciech Plandowski. Testing equivalence of morphisms on context-free languages. In Jan van Leeuwen, editor, *ESA*, volume 855 of *LNCS*, pages 460–470. Springer, 1994. doi:10.1007/BFb0049431.
- 44 Wojciech Plandowski. Satisfiability of word equations with constants is in NEXPTIME. In *STOC*, pages 721–725. ACM, 1999. doi:10.1145/301250.301443.
- 45 Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, 2004. doi:10.1145/990308.990312.
- 46 Wojciech Plandowski. An efficient algorithm for solving word equations. In Jon M. Kleinberg, editor, *STOC*, pages 467–476. ACM, 2006. doi:10.1145/1132516.1132584.
- 47 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. doi:10.1007/BFb0055097.
- 48 Alexander A. Razborov. *On Systems of Equations in Free Groups*. PhD thesis, Steklov Institute of Mathematics, 1987. In Russian.
- 49 John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
- 50 Aleksa Saarela. On the complexity of Hmelevskii’s theorem and satisfiability of three unknown equations. In Volker Diekert and Dirk Nowotka, editors, *Developments in Language Theory*, volume 5583 of *LNCS*, pages 443–453. Springer, 2009. doi:10.1007/978-3-642-02737-6_36.
- 51 Manfred Schmidt-Schauß. Unification of stratified second-order terms. Internal Report 12/94, Johann-Wolfgang-Goethe-Universität, 1994.
- 52 Manfred Schmidt-Schauß. A decision algorithm for stratified context unification. *J. Log. Comput.*, 12(6):929–953, 2002. doi:10.1093/logcom/12.6.929.
- 53 Manfred Schmidt-Schauß. Decidability of bounded second order unification. *Inf. Comput.*, 188(2):143–178, 2004. doi:10.1016/j.ic.2003.08.002.
- 54 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. doi:10.1007/BFb0052361.
- 55 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. doi:10.1006/j.sco.2001.0438.
- 56 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. doi:10.1007/3-540-55124-7_4.