Unification modulo Lists with Reverse as Solving Simple Sets of Word Equations

Siva Anantharaman, Peter Hibbs, Paliath Narendran, Michael Rusinowitch

Unification modulo Lists with Reverse as Solving Simple Sets of Word Equations

Siva Anantharaman[1], Peter Hibbs[2⋆⋆], Paliath Narendran[3], Michael Rusinowitch[4]

[1] LIFO - Université d'Orléans (France), e-mail: siva@univ-orleans.fr
[2] University at Albany–SUNY (USA), e-mail: peter.s.hibbs@gmail.com
[3] University at Albany–SUNY (USA), e-mail: pnarendran@albany.edu
[4] Loria-INRIA Université de Lorraine, Nancy (France), e-mail: rusi@loria.fr

Abstract. Decision procedures for various list theories have been investigated in the literature with applications to automated verification. Here we show that the unifiability problem for some list theories with a *reverse* operator is NP-complete. We also give a unifiability algorithm for the case where the theories are extended with a *length* operator on lists.

1 Introduction

Reasoning about data types such as lists and arrays is an important research area with many applications, such as formal program verification [18, 12]. Early work on this [9] focused on proving inductive properties. Important outcomes of this work include *satisfiability modulo theories* (SMT), starting with the pioneering work of Nelson and Oppen [20] and of Shostak [24]. (See [2] for a more recent syntactic, inference-rule based approach to developing SMT algorithms for lists and arrays.)

In this paper, we investigate the *unification* problem modulo two simple equational theories for lists. The constructors we shall use are the usual 'nil' and 'cons'. We only consider nil-terminated lists, or equivalently, only finite lists that are *proper* in the sense of LISP. (All our lists can actually be visualized as *flat-lists* in the sense of LISP.) We first examine lists with *right cons* (*rcons*) as the only operator (observer), and propose an algorithm for the unification problem modulo this theory (Section 2). We then consider the theory extended with a second operator *reverse* (named *rev*) and develop an algorithm to solve the unification problem over *rev* (Section 3). In both cases, the algorithm is based on a suitable reduction of the unification problem to solving equations on finite words over a finite alphabet, where every equation of the problem has at most one word variable on either side. Further reductions will then lead us to the case where the equations will be 'independent,' and each equation will involve a single word variable; they can be solved by the techniques presented in [7]. All of this can be done in NP with respect to the lengths of the equations of the initial problem. In Section 4 we show how the considerations of length of words can be built into the unification algorithms for the theories *rcons* and *reverse*. These could be of use in formal techniques based on word constraints (e.g., [1, 10, 16, 15]) or in constraint programming [6]. Several examples are given in Section 5, to illustrate how the method we have developed in this paper operates.

⋆⋆ Currently at Google Inc.

Related work. Motivated by constraint logic programming [6], some existential theories of list concatenation have been investigated in [25]. But these works do not consider any list reverse operator. With a view to derive NP decision procedures we reduce our unification problems, on lists with a reverse operator but without concatenation, to systems of word equations that we shall refer to as 'simple systems'. A different class of word equations systems, called quadratic, has been studied in [23], where it is shown that solving such systems of word equations is in NP *if* a simple exponential bound can be obtained on their shortest solution; however, to our knowledge this simple exponential bound has not yet been proved. In [10] it is shown that *if* word equations can be converted to a solved form, then satisfiability of word equations with length constraints is decidable. Satisfiability of quadratic regular-oriented word equations with length constraints is shown decidable in [10]. Again these results do not consider a reverse operator.

2 List theory with *rcons*

The reader is assumed to be familiar with the concepts and notation used in [3]. For terminology and a more in-depth treatment of unification the reader is referred to [5].

The signature underlying our study below, will be 2-sorted with two disjoint types: *element* and *list*. We assume there are finitely many constants (at least 2) of type *element*, while nil will be the unique constant of type *list*. The unification problems we consider are instances of *unification with constants* in the terminology of [5].

For better comprehension, we shall use in general the lower-case letters $x, y, z, u, v, \ldots$ for the variables to which are assigned terms of type *element*, and the upper-case letters $X, Y, Z, U, V, \ldots$, for the variables to which are assigned terms of type *list*; possibly with suffixes or indices, in both cases.

We introduce now the equational axioms of List theory with *rcons*:

$$\begin{aligned} rcons(\mathsf{nil}, x) &\approx \mathsf{cons}(x, \mathsf{nil}) \\ rcons(\mathsf{cons}(x, Y), z) &\approx \mathsf{cons}(x, rcons(Y, z)) \end{aligned}$$

where nil and cons are constructors; and cons, *rcons* are typed respectively as:

$$\begin{aligned} \mathsf{cons} &: \mathsf{element} \times \mathsf{list} \to \mathsf{list} \\ rcons &: \mathsf{list} \times \mathsf{element} \to \mathsf{list} \end{aligned}$$

We refer to this equational theory as RCONS. Orienting these from left to right produces a convergent system:

$$\begin{aligned} (1) \quad & rcons(\mathsf{nil}, x) \to \mathsf{cons}(x, \mathsf{nil}) \\ (2) \quad & rcons(\mathsf{cons}(x, y), z) \to \mathsf{cons}(x, rcons(y, z)) \end{aligned}$$

The following result helps simplifying equations in RCONS:

Lemma 1 *Let s_1, s_2, t_1, t_2 be terms such that $rcons(s_1, t_1) \approx_{\text{RCONS}} rcons(s_2, t_2)$. Then $s_1 \approx_{\text{RCONS}} s_2$ and $t_1 \approx_{\text{RCONS}} t_2$.*

2.1 Unifiability Complexity Analysis

Theorem 1. *Unifiability modulo* RCONS *is NP-hard.*

Proof. We will show this by reduction from 1-in-3-SAT. Given an instance of 1-in-3-SAT, we will construct a unification problem in our theory such that a unifier exists *if and only if* the instance of 1-in-3-SAT is satisfiable. The set of equations thus constructed will be referred to as S.

For each clause $C_i = (a_i \vee b_i \vee c_i)$ in the instance of 1-in-3-SAT, we add the following equation into S:

$$S_i: \ \mathsf{cons}(0, \mathsf{cons}(0, \mathsf{cons}(1, L_i))) \approx^? rcons(rcons(rcons(L_i, x_i), y_i), z_i)$$

where 0 and 1 are constants. Note that this equation has the following three solutions:

1 $L_i \mapsto \mathsf{nil}$, $x_i \mapsto 0$, $y_i \mapsto 0$, $z_i \mapsto 1$
2 $L_i \mapsto \mathsf{cons}(0, \mathsf{nil})$, $x_i \mapsto 0$, $y_i \mapsto 1$, $z_i \mapsto 0$
3 $L_i \mapsto \mathsf{cons}(0, \mathsf{cons}(0, \mathsf{nil}))$, $x_i \mapsto 1$, $y_i \mapsto 0$, $z_i \mapsto 0$

it also has the following solution:

$$L_i \mapsto \mathsf{cons}(0, \mathsf{cons}(0, \mathsf{cons}(1, rcons(rcons(rcons(M_i, x_i), y_i), z_i))))$$

but if we substitute this solution back into equation S_i and apply a series of decompositions, this gives us the following equation:

$$\mathsf{cons}(0, \mathsf{cons}(0, \mathsf{cons}(1, M_i))) \approx^? rcons(rcons(rcons(M_i, x_i), y_i), z_i)$$

Therefore, clearly $\{M_i, x_i, y_i, z_i\}$ has the same solution set as $\{L_i, x_i, y_i, z_i\}$ and must ultimately terminate in a solution of type 1, 2, or 3. If it does not terminate, then the unifier for L_i must be infinitely large and is thus not a valid unifying assignment. We associate the solutions of type 1, 2, and 3 with the truth assignments $\{a_i = false, b_i = false, c_i = true\}$, $\{a_i = false, b_i = true, c_i = false\}$, and $\{a_i = true, b_i = false, c_i = false\}$ respectively. Thus, if the constructed unification problem has a set of finite unifiers for these variables, then the original 1-in-3-SAT problem has a solution (which is given by the previous associations.) Similarly, if there is some satisfying assignment of the 1-in-3-SAT, then a set of finite unifiers for the unification problem can be constructed from that assignment by running the previous associations backward. □

To show that the problem is in NP, we first consider the set of variables of type *element* in the problem, and guess equivalence classes in this set. We then select one representative element from each equivalence class, and replace all instances of the other variables in that class with the chosen representative; whenever possible, choose a constant as representative. Clearly, no equivalence class may contain more than one constant. For every representative $\mathbf{x}$ of type *element* that is not a constant, introduce a fresh (symbolic) constant $c_{\mathbf{x}}$ to act as the representative of its class. This guessing step is clearly in NP. (If a unifier is found involving $c_{\mathbf{x}}$, then all instances of $c_{\mathbf{x}}$ may be replaced by $\mathbf{x}$ once again.)

Once this guessing step is done, all equations of the given unification problem will be of the following form (after RCONS-normalization if necessary):

$$\begin{aligned} &\mathsf{cons}(a_1, \ldots(\mathsf{cons}(a_k, rcons(rcons(\ldots rcons(X, b_l)\ldots, b_1))))) \\ \approx^? \ &\mathsf{cons}(c_1, \ldots(\mathsf{cons}(c_m, rcons(rcons(\ldots rcons(Y, d_n), \ldots, d_1))))) \end{aligned}$$

with X and Y not necessarily distinct. We will represent the sequences $\{a_i\}$, $\{b_i\}$, $\{c_i\}$, $\{d_i\}$ as finite words α, β, γ, δ respectively, over the constants. Such an equation can then be expressed as a word equation as follows:

$$\alpha X\beta \approx^? \gamma Y\delta$$

Clearly, this equation does not have a solution unless either α is a prefix of γ or vice-versus. Without loss of generality, let α be a prefix of γ and let $\alpha^{-1}\gamma$ denote the suffix of γ after α is removed. The equation may be simplified to the following: $X\beta \approx^? \alpha^{-1}\gamma Y\delta$. Similarly, either β or δ is a suffix of the other; there are two cases:

β is a suffix of δ. Let $\delta\beta^{-1}$ denote the remaining prefix of δ.
The equation is then simplified to $X \approx^? \alpha^{-1}\gamma Y\delta\beta^{-1}$
δ is a suffix of β. Let $\beta\delta^{-1}$ denote the remaining prefix of β.
The equation is thus simplified to $X\beta\delta^{-1} \approx^? \alpha^{-1}\gamma Y$

A word equation $\alpha X\beta \approx^? \gamma Y\delta$, is said to be *pruned*, if all common (non-empty) prefixes and suffixes from the two sides of the equation have been removed. If this cannot be done, the equation is unsolvable. Every pruned 1-variable equation is either of the form $\alpha X \approx^? X\beta$, or of the form $X \approx^? \gamma$, and every pruned 2-variable equation is either of the form $X \approx^? \alpha Y\beta$, or of the form $\alpha X \approx^? Y\beta$, for words α, β, γ. Equations of the form $X \approx^? \gamma$ or of the form $X \approx^? \alpha Y\beta$ are said to be in *solved form*. Those of the other types are said to be *unsolved*.

In the following subsection, we present a nondeterministic algorithm to solve any set of such equations, on finite words over a finite alphabet, each equation involving at most two variables, one on either side, appearing at most once. Such a set of equations will be said to be a *simple system*, or a *simple set*, of word equations. The following notions will be useful for presenting and analyzing our algorithm.

Definition 1. *Let U be a simple set of word equations.*

(i) The relation graph G_U *of U is the undirected graph $G = (\mathscr{V}, \mathscr{E})$ where the set of vertices $\mathscr{V}$ is the set of variables in U and the set of edges $\mathscr{E}$ contains (X, Y) iff there is an equation of the form $\alpha X\beta \approx^? \gamma Y\delta$ in U.*

(ii) For any two variables X, Y in U, the variable Y is said to be dependent *on X iff the graph G_U has an edge defined by an equation of the form $Y \approx^? \alpha X\beta$, with α or β (or both) non-empty; such a dependency is denoted as $Y \succ_U X$, or as $X \prec_U Y$.*

(iii) The graph G_U is said to present a dependency cycle *from a variable Y in U, iff for some variables $X_1, X_2, \ldots X_p$ in U, we have: $Y \succ_U X_1 \succ_U \cdots \succ_U X_p \succ_U Y$.*

Given a dependency relation $Y \approx^? \alpha X\beta$ on the variables X, Y in U, the variable Y is said to be the 'lhs' (left-hand-side) of this dependency; the edge on G_U between Y and X is called a *directed dependency edge* from Y to X. (By definition, at least one of α, β is supposed to be non-empty.) A *dependency path* from a node V to a node W is a sequence of dependency edges on G_U, from V to W.

2.2 NP-Solvability of Simple sets: Algorithm A

Algorithm **A** presented below is nondeterministic. We will show that, for any run of Algorithm **A** (successful or not) on any given simple set U of word equations, the total number of steps is polynomial wrt inputs. Moreover the equations generated in a run will be shown to have polynomial size wrt inputs (Section 2.3). Consequently, Algorithm **A** will produce, when successful on U, a system containing only a polynomial number of dependencies and 1-variable equations, each of them of polynomial size. By applying to this resulting system of 1-variable equations, (Lemma 3 followed by) a polynomial solvability check from [7], we will deduce that solvability of simple systems of word equations is in NP.

Under the runs of Algorithm **A**, dependencies chosen in Step 2 get marked; we assume that, initially, none of the dependencies in the given set U is marked.

Step 1. (*Pruning*) For each equation in U of the form $\alpha X\beta \approx^? \gamma Y\delta$, remove all common prefixes and suffixes from the two sides of that equation.
(i) If the two sides of some equation have non-common prefixes or suffixes, then EXIT with failure.
(ii) If for some variable X in U, there is a dependency cycle at X in the graph G_U, then EXIT with failure.

Step 2. Choose an *unmarked* dependency $X \approx^? \alpha Y\beta$ in U; replace all instances of X in all the other equations by $\alpha Y\beta$; *mark the chosen dependency*. GOTO Step 1.

Step 3.a. Select an arbitrary equation such that the variables on the right and left hand sides of the equation are distinct. If no such equation is available, EXIT.

Step 3.b. Let the selected equation be of the form $\alpha X \approx^? Y\beta$.
Guess a word u in $Prefixes(\alpha)$,
(i) If $\alpha = uv$ and $\beta = vw$, with $v \neq \lambda$, then *replace* the selected equation on X, Y by the two equations $\{X \approx^? w, Y \approx^? u\}$ and propagate this substitution through G_U; GOTO Step 1.
(ii) (*Splitting*) Otherwise, let Z be a fresh variable; and *replace* the selected equation on X, Y by the two solved forms: $X \approx^? Z\beta$ and $Y \approx^? \alpha Z$; GOTO Step 1.

Proposition 1 *Let U be a simple set of word equations. The number of steps needed for Algorithm* **A** *to halt is bounded by* $5n$ *where* n *is the initial number of variables in the given problem* U.

Proof. Let d be the number of unsolved variables (i.e., that are not the lhs of an equation). Initially $d \leq n$ where n is the initial number of variables in U. Let $d(k)$ be the value of d when we enter for the kth time in Step 3. Since Step 3 generates one fresh variable and two solved variables (that were not solved at previous steps: otherwise they would have been replaced at Step 2) we have $d(k+1) < d(k)$. Therefore Step 3 is applied at most n times. Hence the number of fresh variables generated (under Splitting) is at most n, and the maximum number of variables at any stage is at most $2n$. Therefore Step 2 can be applied at most $2n$ times, and the same holds also for Step 1. □

When a fresh variable 'Z' is introduced in Step 3.b.(ii) the graph G_U will be *dynamically extended* by the addition of a fresh node labelled with the variable Z; we also introduce two dependency edges from the nodes X and Y to the node Z, corresponding respectively to the two solved forms $X \approx^? Z\beta$, and $Y \approx^? \alpha Z$. Similarly, each time an equation derived under this step turns out (after Pruning) to be a solved form, a dependency edge will be added on the extended graph, between the corresponding nodes.

We note that when Algorithm **A** halts (without failure) on a given problem, we will be left with a set of equations each being either in solved form, or a simple 1-variable equation. Note also that the variables that are lhs of solved forms have a unique occurrence. Hence the resulting system is solvable iff each subsystem of 1-variable equations, on a given variable, is solvable.

We prove in Lemma 3 below, that every subsystem of the resulting 1-variable equations, on a given variable, can be replaced by a single equation (that may not be necessarily simple) on that variable, at polynomial cost; each such 1-variable equation can be checked for solvability, by a known polynomial algorithm from [7]. Prior to that we need to show that, when **A** halts without failure on any problem U, the length of any resulting simple 1-variable equation is polynomially bounded, wrt the size of U. In the following section we shall actually show more.

2.3 Lengths of Prefixes/Suffixes of equations are polynomially bounded

Note that in Steps 2 and 3.b of Algorithm **A**, when a dependency $X = \mu$ is selected, then every other equation e containing at least one occurrence of X is replaced by $e[X \leftarrow \mu]$ and immediately simplified by Pruning (Step 1). After these operations the resulting equation e' *replaces* e.

Suppose now, that a derived equation e' replaces an equation e under the propagation of a dependency (and after Pruning); let α, β denote respectively the prefix and suffix of the equation e, and α', β' those of e'. The replacing equation e' is said to be in '*excess-size*' wrt the equation it replaces, iff $|\alpha'| > |\alpha|$, or $|\beta'| > |\beta|$, or both.

It is easy to see that the propagation of solved forms of the type $Y \approx^? X$, or of the type $Y \approx^? \gamma$, cannot lead to replacing equations in excess-size. We can also check that, in any run of **A** a 1-*variable equation is never replaced by an equation in excess-size*; this follows from a simple case analysis (that we present in detail, in Appendix A). It can also be checked (see Appendix A), that the cases of Steps 2 and 3.b of **A** that can lead to replacing equations possibly in excess-size, are as follows:

- a 2-variable equation in *excess-size* can get derived, when a dependency is applied to the lhs (or the rhs) of a 2-variable equation.

- a 1-variable equation in *excess-size* can get derived, when a dependency is propagated onto a 2-variable equation on the same two variables.

- a solved form equation in *excess-size* can get derived when a dependency is propagated onto a solved form for the same variable, or on a 2-variable equation.

We already know that Algorithm **A** halts in polynomially many steps wrt the number of variables n of the given problem, and that the number of equations in U when **A** halts, is also polynomially bounded wrt n (each step generates at most one equation). We show now, that in the equations derived under **A**, even when they are in excess-size, the lengths of the prefixes/suffixes remain polynomially bounded wrt U.

Let us consider a 2-variable equation that gets derived under **A**: for instance, the 2-variable equation $\alpha_1 Y \approx^? W\beta_1$, on which is propagated the dependency $Y \approx^? \alpha X\beta$. The 2-variable equation will be replaced (after Pruning) by a 2-variable equation of the form $\alpha_1' X \approx^? W\beta_1'$, where: $|\alpha_1' = \alpha_1\,\alpha| \leq |\alpha_1| + |\alpha|$, and $|\beta_1'| \leq |\beta_1|$. To the variable X, brought in by the substitution in the equation derived $\alpha_1' X \approx^? W\beta_1'$, we attach the singleton sequence $[Y \succ X]$, and refer to it as the 'prefix-tag' (or 'ptag') of X in this equation. (Remember: by definition, either the prefix or the suffix of a dependency must be non-empty.) This ptag is to be seen as a tag, to notify that the unique dependency with Y as lhs, has served in the derivation of this fresh equation.

The replacing equation (in the example) will be in excess-size, iff α is non-empty. In the prefix $\alpha_1\,\alpha$ of X in the equation, α_1 is contributed by the 2-variable equation $\alpha_1 Y \approx^? W\beta_1$, and α is contributed by the dependency $Y \approx^? \alpha X\beta$ that is applied to that 2-variable equation. In other words, *if the equation derived is in excess-size*, the ptag *also carries the information* that the excess in the length of the prefix, is due to a portion contributed by the prefix of the dependency.

The ptag sequences grow incrementally, when a fresh equation derived gets replaced in turn, under a subsequent step of **A**, by a new fresh equation. For instance, suppose on the same example, that we have a second dependency of the form $X \approx^? \theta V\delta$. The fresh (replacing) 2-variable equation derived would then be of the form: $\alpha_1'\theta V \approx^? W\beta_1'$. The ptag of V (the variable brought in) in this equation (not necessarily in excess-size) would then be, by definition, the sequence $[Y \succ X, X \succ V]$.

Suffix-tags ('stags') are defined analogously: on the same example above, suppose for instance that we have a dependency $W \approx^? \tau Z\eta$. We would then derive an equation of the form $\alpha'' V \approx^? Z\beta''$; the ptag of V in this equation would still be $[Y \succ X, X \succ V]$, while the stag of Z (the variable brought in by the substitution) would be $[W \succ Z]$.

The ptags and stags can be defined, in formal terms, (recursively,) as follows:

Definition 2. *(i) For any variable in an equation of the given problem U, the ptag attached, wrt that equation, is set to be the empty sequence $[\,]$.*

(ii) Suppose that, under some step of the algorithm **A***:*

- *a dependency of the form $Y \approx^? \alpha X\beta$ is propagated onto an equation of the form $\gamma_1 Y \approx^? W\delta_1$ (resp. of the form $X \approx^? \gamma_1 Z\delta_1$);*
- *that the variable Y in $\gamma_1 Y \approx^? W\delta_1$ (resp. Z in $X \approx^? \gamma_1 Z\delta_1$) has an attached ptag of the form $[\mathsf{c}]$, where c is a (possibly empty) finite sequence of dependency relations on the graph G_U;*
- *and that from the propagation of the dependency (after Pruning), we derive an equation of the form $\gamma_1' X \approx^? W\delta_1'$ (resp. of the form $Y \approx^? \gamma_1' Z\delta_1'$).*

Then, the ptag attached to the variable X (resp. the variable Z), brought in by the substitution in the replacing equation, is set to be $[\mathsf{c}, Y \succ X]$

(iii) stags are defined analogously.

Note that the ptags/stags define a dependency chain of the form $Y \succ X \succ V \succ W \ldots$, on the variables of equations that get derived, under the Steps 2 and 3.b of **A**. .

Lemma 2 *Assume that algorithm* **A** *halts without failure on a given problem U. Then,* no variable can appear more the once in the dependency chain *defined by the ptag, or stag, of any equation derived under the runs of* **A** *on U.*

Proof. If we assume the contrary, then we get a dependency cycle on the (extended) relation graph of U; but then **A** would have exited with failure on U. $\square$

Corollary 1. *Assume that algorithm* **A** *halts without failure, on a given problem U. Then the length of the prefix of any resulting equation is polynomially bounded, wrt $N\,s$, where N is the total number of equations in U, and s is the maximum size of the prefixes or suffixes of the equations in U.*

Proof. By the above Lemma, the number of dependency relations in any ptag or stag is at most the number N_1 of dependencies, initial or derived under the runs of **A**; and we also know that N_1 is polynomial on N. On the other hand, the maximal (or 'worst') growth in the prefix size of any derived equation e, when **A** halts, would be when *each* dependency relation in its ptag sequence corresponds to a derived equation in excess-size. But, as observed above, this means, that the prefix α (or suffix β) of a dependency of the form $Y \approx^? \alpha X \beta$ whose propagation led to the derivation of the equation e, has contributed to the excess-size in the prefix (or suffix) of the variable X in e. On the other hand, we know that the length of the prefix (or suffix) of any any dependency in the problem, initial or derived under Steps 2 and 3.b on a first run of **A**, is polynomial on $N\,s$ (cf. Appendix A); an inductive argument, on the number of steps of **A** before it halts, proves that the same bound holds also for all derivations under the subsequent runs of **A**. That proves the corollary. $\square$

Note however, that when **A** halts without failure on a given problem, the resulting 1-variable equations may not be all independent. So, to be able to to apply [7] and conclude, it remains now to replace every subsystem formed of the resulting simple 1-variable equations *on the same variable*, by an equivalent single equation (which may not be simple) on that variable, but of polynomial size w.r.t. the size of U. This is the objective of our next lemma:

Lemma 3 *Any system S of* 1*-variable equations, of size m, on a given variable X, is equivalent to a* single 1*-variable equation of size $p(m)$ for some fixed polynomial p (where X can appear more than once on either side).*

Proof. We first recall the well-known 'trick' (see [14]) to build such a single equation from two equations:

$$\begin{cases} u &= v \\ u' &= v' \end{cases} \equiv \quad uau'ubu' = vav'vbv'$$

where a,b are two distinct constants. The resulting equation is of size $2|S|+4$. Since the initial system S is of size $|S| \geq 4$, we deduce that the resulting single equation has size $\leq 3|S|$. To iterate the process on a system W of n equations (indexed from 1 to n) we consider an integer k such that $k-1 \leq \log n < k$; by adding to the system $2^k - n$ trivial equations $X = X$, we get an extended system $V = (V_i)$ with equations indexed from 1 to 2^k. We shall show by induction, that V is equivalent to a single equation of size $\leq 3^k |V|$.

Assume (as inductive hypothesis) that we have derived, for the two systems $V' = (V_i)_1^{2^{k-1}}$ and $V'' = (V_i)_{2^{k-1}+1}^{2^k}$ two equivalent single equations e' and e'' respectively, of size $\leq 3^{k-1}|V'|$ and $\leq 3^{k-1}|V''|$ respectively. Now if we combine e' and e'' we obtain an equivalent single equation of size bounded by $\leq 3(3^{k-1}|V'| + 3^{k-1}|V''|) = 3^k(|V'| + |V''|) = 3^k(|V|)$. Getting back to system W, this means that W is equivalent to a single equation of size $\leq 3^k(|W| + 2^k - n)$. Since $k \leq \log n + 1$ we have $3^k(|W| + 2^k - n) \leq 3^{\log n+1}(|W| + 2^{\log n+1} - n) \leq 3n^{\log 3}(|W| + 2n^{\log 2} - n)$. Since n is bounded by $|W|$, we deduce the assertion of the lemma. $\square$

Theorem 2. *Solvability of a simple set U of word equations is in NP.*

Proof. Assume that Algorithm **A** halts without failure on the given problem U. We shall then be left with a final system of solved form equations, along with several (simple) 1-variable equations. Moreover (see Lemma 2, and Corollary 1) the size of these 1-variable equations is polynomially bounded wrt the size of U. Thanks to Lemma 3, every subsystem of these 1-variable equations involving a given variable X is equivalent to a single 1-variable equation in X (that may not be simple). Each of these resulting 1-variable equations can then be solved, *independently*, in polynomial time (see [7]). $\square$

We can now conclude:

Theorem 3. *Unifiability modulo* RCONS *is NP-complete.*

3 List theory with *rev*

The axioms of this theory are

$$\begin{aligned} rcons(\mathsf{nil}, x) &\approx \mathsf{cons}(x, \mathsf{nil}) \\ rcons(\mathsf{cons}(x, Y), z) &\approx \mathsf{cons}(x, rcons(Y, z)) \\ rev(\mathsf{nil}) &\approx \mathsf{nil} \\ rev(\mathsf{cons}(x, Y)) &\approx rcons(rev(Y), x) \end{aligned}$$

where nil and cons are constructors. Orienting each of the above equations to the right yields a convergent rewrite system (with 4 rules). But the term rewriting system we shall consider here, for the theory *rev*, is the following system of six rewrite rules:

$$\begin{array}{lrl} (1) & rcons(\mathsf{nil}, x) & \to \mathsf{cons}(x, \mathsf{nil}) \\ (2) & rcons(\mathsf{cons}(x, Y), z) & \to \mathsf{cons}(x, rcons(Y, z)) \\ (3) & rev(\mathsf{nil}) & \to \mathsf{nil} \\ (4) & rev(\mathsf{cons}(x, Y)) & \to rcons(rev(Y), x) \\ (5) & rev(rcons(X, y)) & \to \mathsf{cons}(y, rev(X)) \\ (6) & rev(rev(X)) & \to X \end{array}$$

which is again convergent. We shall refer to this equational theory as REV.

Actually, the two added rules (5) and (6) are derivable as inductive consequences of the first four rules. We shall prove this by induction on the length of the list-term X, where by 'length' of any ground list-term X, we shall mean the number of applications of cons in X at the outermost level.

We first prove the claim for the added rule (5): $rev(rcons(X, y)) \to \mathsf{cons}(y, rev(X))$. Suppose we have some ground term X. If $X = \mathsf{nil}$, then

$rev(rcons(X,y)) \approx rev(rcons(\mathsf{nil},y)) \to^+ \mathsf{cons}(y,\mathsf{nil})$ and
$\mathsf{cons}(y,rev(X)) \approx \mathsf{cons}(y,rev(\mathsf{nil})) \to^+ \mathsf{cons}(y,\mathsf{nil})$

If $X = \mathsf{cons}(a,Y)$ for some term a and some term Y of length n, then

$rev(rcons(X,y)) \approx rev(rcons(\mathsf{cons}(a,Y),y)) \to^+ rcons(rev(rcons(Y,y)),a)$ and
$\mathsf{cons}(y,rev(X)) \approx \mathsf{cons}(y,rev(\mathsf{cons}(a,Y))) \to^+ rcons(\mathsf{cons}(y,rev(Y)),a)$

By the inductive assumption $rev(rcons(Y,y)) = \mathsf{cons}(y,rev(Y))$ and we are done. We prove then the claim for the added rule (6): $rev(rev(X)) \to X$.

Clearly $rev(rev(\mathsf{nil})) \to^+ \mathsf{nil}$. Let $X = \mathsf{cons}(a,Y)$ for some term a and some term Y of length n. Then $rev(rev(X)) \approx rev(rev(\mathsf{cons}(a,Y))) \to rev(rcons(rev(Y),a))$ $\to \mathsf{cons}(a,rev(rev(Y))) \to \mathsf{cons}(a,Y) = X$.

From this point on, without loss of generality, we will consider all terms to be in normal form modulo this term rewrite system.

Lemma 4 *Let S_1, S_2, t_1, t_2 be terms such that:* $rcons(S_1,t_1) =_{\mathrm{REV}} rcons(S_2,t_2)$. *Then* $S_1 =_{\mathrm{REV}} S_2$ *and* $t_1 =_{\mathrm{REV}} t_2$.

Lemma 5 *Let S_1, S_2 be terms such that: $rev(S_1) =_{\mathrm{REV}} rev(S_2)$. Then* $S_1 =_{\mathrm{REV}} S_2$.

Lemma 6 *Unifiability modulo* REV *is NP-hard.*

Proof. The NP-hardness proof for unifiability modulo RCONS (as given in Section 2) remains valid for unifiability modulo REV as well. □

Theorem 4. *Unifiability modulo* REV *is in NP and is therefore NP-Complete.*

Proof. After normalization with the rules of REV, we can assume that, for every equation in the given unification problem, its lhs as well as its rhs are of one of the following two types:

$$cons(x_1, cons(x_2, \ldots (rcons(rcons(\ldots(rcons(X,y_1)\ldots))),$$
or
$$cons(x_1, cons(x_2, \ldots (rcons(rcons(\ldots(rcons(rev(Y),z_1)\ldots)))$$

If the lhs and the rhs of an equation are both of the first type, or both of the second type, then we can associate with it a word equation of the form $\alpha X\beta \approx^? \alpha' Y \beta'$, as in Section 2; we deal with all such equations first, exactly as we did in Section 2.

Once done with such equations, we consider equations (in the unification problem) whose lhs are of the first type, while their rhs are of the second type, or vice versa. To each such equation we can associate either a word equation of the form $\alpha X\beta \approx^? \alpha' Y^R \beta'$, or a word equation of the form $\alpha X\beta \approx^? \alpha' X^R \beta'$, where Y^R (resp. X^R) is a variable that stands for $rev(Y)$ (resp. for $rev(X)$); naturally, all these will be duly pruned.

The (pruned) word equations of a 'mixed' type, of the form $\alpha X \approx^? Y^R \beta$ involving two different variables X, Y will be handled by the addition of an *extra splitting inference step* to the algorithm **A**, say between its Steps 2 and 3. In concrete terms, such an equation will first get split by writing: $X \approx^? Z\beta$, and $Y^R \approx^? \alpha Z$, where Z is a fresh variable, then 'solving it locally' as $X \approx^? Z\beta'$, $Y \approx^? Z^R \alpha^R$; this substitution will then be propagated to all the other equations of the problem involving X or Y; the resulting equations derived thereby, will be treated similarly, and by the procedure that we present below for the equations involving a single variable.

We present now the part of the algorithm that deals with all the (pruned) word equations of the form $\alpha X\beta \approx^? \alpha' X^R \beta'$, on a given variable X of the problem. This part will be referred to as the *palindrome discovery* step of the algorithm. We will use the word *palindrome* to refer to a variable X that has to satisfy $X = X^R$. We maintain a list of variables that are known to be palindromes in our algorithm, which is initially empty. Clearly, if X is known to be a palindrome, then $\alpha X\beta \approx^? \alpha' X^R \beta'$ is the same as $\alpha X\beta \approx^? \alpha' X\beta'$ and need not be considered at this step in the algorithm.

In this part, we have two cases to consider:

Case 1: $X \approx^? \alpha'' X^R \beta''$. In this case, if $|\alpha''\beta''| = 0$, then we conclude that X is a palindrome. Else, if $|\alpha''\beta''| \neq 0$, then there is clearly no solution and we terminate with failure.

Case 2: $\alpha'' X \approx^? X^R \beta''$. In this case, we check for the existence of words u, v such that $\alpha'' = u^R v,\ \beta'' = vu$. If such a pair exists, we may conclude that $X = u$, and this solution can be propagated through the dependency graph, as in the flat-list case. Again, there cannot be more than $min(|\alpha|, |\beta|)$ of these solutions. If all such pairs are checked without finding a solution, then we resort to splitting and write $X = Z\beta'', X^R = \alpha'' Z$, where Z is fresh. This second equation gives us $X = Z^R \alpha''^R$ and therefore $Z\beta'' = Z^R \alpha''^R$. If $\beta'' \neq \alpha''^R$, then there is no solution and we may terminate with failure. Otherwise we may conclude that $Z = Z^R$ (and is therefore a palindrome) and replace all occurrences of X with $Z\beta''$.

Once we have finished this, we have to check with the equations of the form $\alpha X \approx^? X\beta$ involving the same variable X studied above. If X is not a palindrome, then we may use (possibly after grouping several equations with Lemma 3) the algorithm given in [7] to find a solution. If X is known to be a palindrome, then we still run the algorithm given in [7] to check for a solution, but we first check that the prefixes and suffixes of each equation (i.e., α, β) meet certain criteria.

In the case where $|\alpha|$ or $|\beta| \geq |X|$, the equation $\alpha X \approx^? X\beta$ implies that X has to be a prefix of α and a suffix of β. Therefore we may exhaustively check all palindrome prefixes and suffixes of α and β respectively for validity.

Remains to consider the case where $|\alpha|$ and $|\beta| < |X|$; then, according to the following Lemma 7, X is a solution if and only if there exist palindromes u, v, and a positive integer k such that $\alpha = uv$, $\beta = vu$ and $X = (uv)^k u$.

Lemma 7 *Let α, β and A be non-empty words such that A is a palindrome and $|\alpha| = |\beta| < |A|$. Then $\alpha A = A\beta$ if and only if there exist palindromes u, v, and a positive integer k such that $\alpha = uv$, $\beta = vu$ and $A = (uv)^k u$.*

Proof. If $\alpha = uv$, $\beta = vu$ and $A = (uv)^k u$ for palindromes u, v then

$$\alpha A = uv(uv)^k u = (uv)^{k+1} u = (uv)^k uvu = A\beta$$

Also, $A^R = ((uv)^k u)^R = u^R (v^R u^R)^k = u(vu)^k = (uv)^k u = A$. So, A is indeed a palindrome and satisfies $\alpha A = A\beta$.

It is well-known that for any equation $\alpha A = A\beta$ where $0 < |\alpha| = |\beta| < |A|$, α and β must be *conjugates*. That is, there must exist some pair of words u, v, such that $\alpha = uv$ and $\beta = vu$. Furthermore, A must be $(uv)^k u$ for some k. If A is also a palindrome, then α must be a prefix of A and β must be a suffix of A. Because A is a palindrome, α^R is therefore also a suffix of A. So, because $|\alpha| = |\beta|$, we may conclude that $\alpha^R = \beta$. The proof proceeds as follows:

$$\alpha^R = \beta \quad \text{implies} \quad (uv)^R = vu \quad \text{implies} \quad v^R u^R = vu \quad \text{implies} \quad (v^R = v \quad \text{and} \quad u^R = u)$$

Thus u and v are palindromes, $\alpha = uv$, $\beta = vu$, and $A = (uv)^k u$ for some $k > 0$. $\square$

4 List theories with *length*

In many cases of practical interest, list data types are 'enriched' with a length operator, under which, e.g., the list $\mathsf{cons}(a,\mathsf{nil})$ will have length 1, the list $\mathsf{cons}(a,\mathsf{cons}(b,\mathsf{nil}))$ will have length 2, etc. Solving equations on list terms in these cases will need to take into account length constraints. For instance $\mathsf{cons}(a,X) = \mathsf{cons}(a,Y)$ cannot be solved if $length(X) = s(length(Y)$ (where s stands for the successor function on natural integers). We shall be assuming in this section that a length operator is defined on the lists we consider, and that this operator is formally defined in terms of a (typed) convergent rewrite system, presented below in Sections 4.1 and 4.2. Our objective will be to solve equations on list terms subject to certain given length constraints. We will again reduce the problem to solving some word equations. It seems appropriate here to quote [16]: "The problem of solving a word equation with a length constraint (i.e., a constraint relating the lengths of words in the word equation) has remained a long-standing open problem". However, thanks to the special form of the word equations we deal with, we will be able to provide a decision algorithm in our case.

4.1 *length* with *rcons*

The Term Rewrite System: The (typed) rewrite rules for *rcons* with $length$ are given below, where the unary functions s and $length$ are typed as $s : \mathsf{nat} \to \mathsf{nat}$, $length : \mathsf{list} \to \mathsf{nat}$; the constant 0 is typed $0 \to \mathsf{nat}$;. This rewrite system is convergent:

$$\begin{aligned}
length(\mathsf{nil}) &\to 0\\
length(\mathsf{cons}(x,Y)) &\to \mathsf{s}(length(Y))\\
rcons(\mathsf{nil},x) &\to \mathsf{cons}(x,\mathsf{nil})\\
rcons(\mathsf{cons}(x,Y),z) &\to \mathsf{cons}(x,rcons(Y,z))\\
length(rcons(X,y)) &\to \mathsf{s}(length(X))
\end{aligned}$$

Variables of type nat will be denoted (in general) by the lower case letters $i, j, k, m, n, \ldots$ The last rule above, $length(rcons(x,y)) \to \mathsf{s}(length(x))$, is derived easily from the preceding rules, by induction.

The Unification Algorithm We shall assume that all instances of s and 0 in the equations of the given problem have been removed by successive applications of the inference rules below, where $\mathscr{E}\mathscr{Q}$ is a set of equations and $\uplus$ is the disjoint union:

$$\frac{\mathscr{E}\mathscr{Q} \uplus \{n \approx^? 0\}}{\mathscr{E}\mathscr{Q} \cup \{n \approx^? \mathit{length}(X'),\, X' \approx^? \mathsf{nil}\}}$$

$$\frac{\mathscr{E}\mathscr{Q} \uplus \{n \approx^? \mathsf{s}(m)\}}{\mathscr{E}\mathscr{Q} \cup \{n \approx^? \mathit{length}(X'),\, m \approx^? \mathit{length}(Y'),\, X' \approx^? \mathsf{cons}(x',Y')\}}$$

Assume further that our given unification problem U has been transformed into word equations of the form $\alpha X\beta \approx^? \gamma Y\delta$ and that those equations have been reduced to a set of mutually independent systems S_i of equations on one variable X_i. Let us call this set $\mathbb{S} = \bigcup S_i$ where each S_i is a set of equations on one variable. These independent systems of equations S_i may be solved, each producing a solution-set which forms a regular language [7]. We shall use L_i to refer to the solution-set to the system of equations S_i. By Theorem 3 in [7] either $L_i = F_i$ or $L_i = F_i \cup (u_i v_i)^+ u_i$ for some words u_i, v_i and some finite set of words F_i.

In the version of this problem without *length*, the problem of *unifiability* is now solved by simply checking each element of $\{L_i\}$ for (non-)emptiness. However, here the solutions may still be related by *length* equations. We must find a set of words $\{w_i \mid w_i \in L_i\}$ which satisfy length constraints of the form $|w_i| = |w_j| + c_{ij}$ for a constant, non-negative integer c_{ij}. Note that not all pairs i, j need have a constraint of this form.

For w_i we either try elements of F_i or a word of type: $(u_i v_i)^{n_i} u_i$. For the latter case constraints of the type $|w_i| = |w_j| + c_{ij}$ are equivalent to $(|u_i| + |v_i|)n_i + |u_i| = (|u_j| + |v_j|)n_j + |u_j| + c_{ij}$, where n_i, n_j are non-negative integer variables; so we can always reduce our problem to solving a finite number of linear diophantine equations.

4.2 *length* with *rcons* and *rev*

The Term Rewrite System: We now add the rewrite rules of *rev* as defined in Section 3. The resulting rewrite system (where $0, s, \mathit{length}$ are typed as in Section 4.1) is convergent:

$$\begin{aligned}
\mathit{length}(\mathsf{nil}) &\to 0\\
\mathit{length}(\mathsf{cons}(x,y)) &\to \mathsf{s}(\mathit{length}(y))\\
\mathit{rcons}(\mathsf{nil},x) &\to \mathsf{cons}(x,\mathsf{nil})\\
\mathit{rcons}(\mathsf{cons}(x,y),z) &\to \mathsf{cons}(x,\mathit{rcons}(y,z))\\
\mathit{rev}(\mathsf{nil}) &\to \mathsf{nil}\\
\mathit{rev}(\mathsf{cons}(x,y)) &\to \mathit{rcons}(\mathit{rev}(y),x)\\
\mathit{rev}(\mathit{rev}(x)) &\to x\\
\mathit{length}(\mathit{rcons}(x,y)) &\to \mathsf{s}(\mathit{length}(x))\\
\mathit{length}(\mathit{rev}(x)) &\to \mathit{length}(x)
\end{aligned}$$

The last rule, $\mathit{length}(\mathit{rev}(x)) \to \mathit{length}(x)$, is not originally in the theories of RCONS, REV or LENGTH but can be easily derived by induction from the other rules.

The Unification Algorithm We again assume that all instances of s and 0 have been removed from the equations of the given problem, by successive applications of the inference rules below:

$$\frac{\mathscr{E}\mathscr{Q} \uplus \{n \approx^? 0\}}{\mathscr{E}\mathscr{Q} \cup \{n \approx^? \mathit{length}(X'),\, X' \approx^? \mathsf{nil}\}}$$

$$\frac{\mathscr{E}\mathscr{Q} \uplus \{n \approx^? \mathsf{s}(m)\}}{\mathscr{E}\mathscr{Q} \cup \{n \approx^? \mathit{length}(X'),\, m \approx^? \mathit{length}(Y'),\, X' \approx^? \mathsf{cons}(x',Y')\}}$$

Once more, we assume that our unification problem has been transformed into word equations of the form $\alpha X \beta \approx^? \gamma Y \delta$ and that those equations have been reduced to a set of mutually independent systems of equations on one variable (without reverse) which may be required to be a palindrome. As before, let $\mathbb{S} = \{S_i\}$ where each S_i is a set of equations on one variable which are respectively solved by the languages L_i.

Now suppose that S_i has variable X that is not required to be a palindrome, then its solution-set is either F_i or $F_i \cup (u_i v_i)^+ u_i$ for some words u_i, v_i and some finite set of words F_i according to Theorem 3 of [7]. If X is required to be a palindrome then its solution-set contains either a finite set of palindromes F_i or $F_i \cup (u_i v_i)^+ u_i$, where u_i, v_i, are such that $\alpha = u_i v_i$, $\beta = v_i u_i$ according to Lemma 7 and Theorem 3 of [7].

The algorithm now continues as in the previous *length* with *rcons* case: if two solution-sets are related by $length$ equations, satisfiability may be checked by solving a finite set of linear diophantine equations.

5 Some illustrative examples

The following simple examples illustrate how our methods presented above will operate (either directly, or indirectly) in concrete situations.

Example 1. Consider the system formed of two 'list equations':

$$\mathit{cons}(x,X) \approx^? \mathit{rev}((\mathit{cons}(y,Y)),\ \mathit{cons}(a,X) \approx^? \mathit{rev}(\mathit{cons}(a,\mathit{rev}(X)))$$

As described in Section 3, this system will first get transformed to a system of two word equations: $xX \approx^? Y^R y,\ aX \approx^? Xa$.

We can apply the Splitting step of algorithm **A** to the first word equation, and derive: $X \approx^? Zy$, $Y^R \approx^? xZ$, where Z is fresh; the latter of these two will get transformed to the solved form: $Y \approx^? Z^R x$. We have thus derived two solved forms: $X \approx^? Zy$, $Y \approx^? Z^R x$. Propagating for X from the first of these solved forms in the second word equation would a priori give: $aZy \approx^? Zya$, so Pruning would imply: $y = a$. And the variable Z has to satisfy: $aZ \approx^? Za$; which is true for any of the assignments: $Z = \mathit{nil}$, $Z = a$, $Z = aa$, $Z = aaa$, etc. If we choose $Z = \mathit{nil}$, and $x = a$, we get the following solution for the given list equations: $X = \mathit{rcons}(\mathit{nil},a)$, $Y = \mathit{rcons}(\mathit{nil},a)$.

Example 2. Consider the single 1-variable word equation $abX \approx^? Xba$.

For solving this 1-variable equation, (instead of appealing to the general result of [7]) we could choose to see the equation as an instance of a 2-variable equation, and use Splitting, as an ad hoc technique: i.e., replace the X on the lhs by Zba, and the X on the rhs by abZ.

The equation would then become: $abZba \approx^? abZba$, on the single variable Z, which admits any value of Z as a solution. Now, each of the assignments $Z = a, Z = aba, Z = ababa$ satisfies the equalities $X = abZ = Zba$. So each of the assignments $X = aba, X = ababa, X = abababa$, is a solution for the given problem.

Example 3. We consider now the set of word equations: $abX \approx^? Yba$, $Y \approx^? X^R$.

This would be replaced first by the equation $abX \approx^? X^R ba$, and the equality $Y = X^R$. For solving the former we use Splitting, and write: $X \approx^? Zba$ and $X^R \approx^? abZ$.

The fresh variable Z has then to satisfy the condition that $Zba = Z^R ba$. That is to say: Z must be a palindrome. Any palindrome (on the given alphabet) is actually a solution. We thus deduce that the assignments $X = Zba, Y = abZ$, where Z is *any* palindrome, is a solution for the given set of equations.

Note: This also shows that unification modulo the theory rev is infinitary. (It is not difficult to see that unification modulo the theory rcons is not finitary either.)

Example 4. We consider now the set of word equations, subject to a length constraint:

$$abX \approx^? Xba,\ X \approx^? ababY,\ length(Y) = 1$$

This set would be transformed into a set of word equations:

$$\{abX \approx^? Xba,\ X \approx^? ababY,\ Y \approx^? yZ,\ Z \approx^? nil\}.$$

Propagation of the first dependency would give us the 1-variable equation $abababY \approx^? ababYba$, which, once pruned, would become: $abY \approx^? Yba$. Propagation of the other dependencies would give us the equation $aby \approx^? yba$; which admits as solution $y = a$. Thus the given problem admits as solution: $X = ababa, Y = a, Z = nil$.

Suppose now, that the length constraint given is either $length(Y) = 0$, or $length(Y) = 3$, instead of the one given above. Then, by what we have seen in the previous two examples (we know the forms of the possible solutions for Y, after the propagation of the first dependency; therefore) the problem thus modified would be unsatisfiable.

6 Conclusion and Future Work

We have shown that unifiability modulo the two theories RCONS, REV are both NP-complete. For that we have identified a new class of word equations, *simple sets,* which can be solved in NP. One possible direction for our future work would be to investigate other problems for these list theories; for instance we can show that that the uniform word problem for RCONS is undecidable (see Appendix B).

A second direction of future work would be to identify a class of *non-simple* sets of word equations, which can be solved by a suitable adaptation (and extension) of the algorithm **A**.

We also plan to investigate the interesting question of whether the results, such as membership in NP, hold with the addition of linear constant restrictions (as in [4, 25]), to the theory of REV. This could lead to a method to solve the positive fragment of REV. Disunification modulo REV is another interesting problem to investigate, and that may be reducible to the previous one.

References

1. P.A. Abdulla, M.F. Atig, Y-F. Chen, L. Holik, A. Rezine, P. Rümmer, J. Stenman. String Constraints for Verification. In: Computer Aided Verification. CAV 2014. *Lecture Notes in Computer Science* 8559. Springer, 2014.
2. A. Armando, M.P. Bonacina, S. Ranise, S. Schulz: New results on rewrite-based satisfiability procedures. ACM Trans. Comput. Log. 10(1): 4:1-4:51 (2009)
3. F. Baader, T. Nipkow. *Term Rewriting and All That*. Cambridge Univ Press, 1999.
4. F. Baader and K.U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *Journal of Symbolic Computation* 21(2):211–243, 1996.
5. F. Baader, W. Snyder. Unification Theory. In: John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.
6. A. Colmerauer. An Introduction to Prolog III. 69-90 *Communications of the ACM* Volume 33 Issue 7, July 1990.
7. R. Dabrowski, W. Plandowski. On Word Equations in One Variable. *Algorithmica* 60(4): 819–828 (2011).
8. V. Diekert, A. Jez, W. Plandowski. Finding All Solutions of Equations in Free Groups and Monoids with Involution. *Information and Computation* 251, 263–286 (2016).
9. J.V. Guttag, E. Horowitz, D.R. Musser. Abstract Data Types and Software Validation. *Commun. ACM* 21(12): 1048–1064 (1978).
10. V. Ganesh, M. Minnes, A. Solar-Lezama, M. Rinard. Word Equations with Length Constraints: What is Decidable? In: Hardware and Software: Verification and Testing. HVC 2012. *Lecture Notes in Computer Science* 7857. Springer, 2013.
11. P. Hibbs. *Unification modulo common list functions.* Doctoral Dissertation, University at Albany—SUNY, 2015.
12. D. Kapur. *Towards A Theory For Abstract Data Types.* Doctoral Dissertation, Massachusetts Institute of Technology, 1980.
13. D. Kapur, D.R. Musser. Proof by Consistency. *Artif. Intell.* 31(2): 125–157 (1987).
14. J. Karhumäki. Combinatorics on Words: A New Challenging Topic. Turku Centre for Computer Science, 12 (2004).
15. T. Liang, N. Tsiskaridze, A. Reynolds, C. Tinelli, C. Barrett. A Decision Procedure for Regular Membership and Length Constraints over Unbounded Strings. In: Lutz C., Ranise S. (eds), Frontiers of Combining Systems. FroCoS 2015. Lecture Notes in Computer Science 9322 (2015).
16. A.W. Lin, R. Majumdar. Quadratic Word Equations with Length Constraints, Counter Systems, and Presburger Arithmetic with Divisibility. arXiv preprint arXiv:1805.06701, 2018.
17. K. Morita. Universality of a reversible two-counter machine, Theoretical Computer Science, Volume 168, Issue 2, 1996, Pages 303-320.
18. D.R. Musser. Abstract Data Type Specification in the AFFIRM System. *IEEE Trans. Software Eng.* 6(1): 24–32 (1980).
19. D.R. Musser. On Proving Inductive Properties of Abstract Data Types. In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages* (POPL) 154–162 (1980).
20. G. Nelson, D.C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems* 1(2): 245–257 (1979).
21. D.C. Oppen. Reasoning About Recursively Defined Data Structures. *J. ACM* 27(3): 403–411 (1980).
22. W. Plandowski. An Efficient Algorithm For Solving Word Equations. In: *Proceedings of the ACM Symposium on the Theory of Computing '06*, 467–476 (2006).

23. J.M. Robson, V. Diekert. On Quadratic Word Equations. STACS 1999: 217-226 *Lecture Notes in Computer Science* 1563. Springer, 1999.
24. R.E. Shostak. Deciding Combinations of Theories. *J. ACM* 31(1): 1–12 (1984).
25. Klaus U. Schulz. On Existential Theories of List Concatenation. CSL 1994: 294-308 *Lecture Notes in Computer Science* 933. Springer, 1994.

Appendix A

Effect of Steps 2 and 3 of Algorithm A on word equations

We are mainly interested in how the sizes of the prefixes/suffixes of equations evolve, under the propagations of substitutions in these steps. For this, we also need to see how new dependencies could get derived.

I. Propagation of a dependency $Y \approx^? \alpha X\beta$, chosen in Step 2 of $\mathbf{A}$, onto the other equations in U, followed by Pruning (and assuming non-failure):
(i) A 1-variable equation of the form $\alpha_3 Y \approx^? Y\beta_3$ will be replaced by a 1-variable equation, of the form $\alpha_3' X = X\beta_3'$, where $|\alpha_3'| = |\alpha_3|$ and $|\beta_3'| = |\beta_3|$.

Proof. This equation will be a priori replaced by the 1-variable equation $\alpha_3\alpha X\beta \approx^? \alpha X\beta\beta_3$; after Pruning (if non-failure), we will get an equation of the form $\alpha_3' X = X\beta_3'$, with $\alpha_3\alpha = \alpha\alpha_3'$ and $\beta\beta_3 = \beta_3'\beta$. But this implies: $|\alpha_3'| = |\alpha_3|$ and $|\beta_3'| = |\beta_3|$. □
Note: The sizes of the prefix/suffix of the replacing equation remain unchanged.

(ii) A dependency in U with the same lhs, of the form $Y \approx^? \alpha_1 V\beta_1$ will be replaced:
- Either by a 2-variable equation, of the form $\alpha' X \approx^? V\beta'$, or of the form $\alpha_1' V \approx^? X\beta_1'$. (If $V = X$ we derive a 1-variable equation on X.)
- Or by a fresh dependency of the form $V \approx^? \alpha'' X\beta''$, or of the form $X \approx^? \alpha''' V\beta'''$.

Proof. The dependency $Y \approx^? \alpha_1 V\beta_1$ will be replaced a priori by the 2-variable equation $\alpha X\beta \approx^? \alpha_1 V\beta_1$; Pruning would lead to the following four possibilities
(a): we derive a 2-variable equation $\alpha' X \approx^? V\beta'$, with $\alpha = \alpha_1\alpha'$, $\beta_1 = \beta'\beta$;
(b): we derive a 2-variable equation $\alpha_1' V \approx^? X\beta_1'$, with $\alpha_1 = \alpha\alpha_1'$, $\beta = \beta_1'\beta$;
(c): we derive a solved form $V \approx^? \alpha'' X\beta''$, with $\alpha = \alpha_1\alpha''$, $\beta = \beta_1\beta''$;
(d): we derive a solved form $X \approx^? \alpha''' V\beta'''$, with $\alpha_1 = \alpha\alpha'''$, $\beta_1 = \beta\beta'''$.

Note: In the subcases (a), (b), (c), there could *be an increase in the size of the prefix (or of the suffix) of the replacing equation, wrt that in $Y \approx^? \alpha_1 V\beta_1$.*

(iii) A 2-variable equation, of the form $\alpha_2 Y \approx^? W\beta_2$ will be replaced by a 2-variable equation of the form $\alpha_2' X \approx^? W\beta_2'$, where: $|\alpha_2'| \le |\alpha_2| + |\alpha|$, and $|\beta_2'| \le |\beta_2|$

Proof. The equation $\alpha_2 Y \approx^? W\beta_2$ will be a priori replaced by the equation $\alpha_2\alpha X\beta \approx^? W\beta_2$; after Pruning (if non-failure), this will become the equation $\alpha_2' X \approx^? W\beta_2'$, with $\alpha_2' = \alpha_2\alpha$ and $\beta_2 = \beta_2'\beta$. (If $W = X$, we get a fresh 1-variable equation.) $\square$

Note: Replacing equation could *have a bigger prefix here, than the equation replaced.*

II. Steps 3 applied on a *chosen* 2-variable equation $\alpha X \approx^? Y\beta$
(followed by Pruning).

Effects of Step 3.b.(i) on the set of equations:

The number of sets of words $\{u, v, w\}$ satisfying the conditions in Step 3.b.(i) is at most $min(|\alpha|, |\beta|)$. On the other hand, there will be no increase in the size of the prefix or suffix of any equation derived under this step.

Effects of Step 3.b.(ii) on the set of equations:

The chosen 2-variable equation $\alpha X \approx^? Y\beta$ is replaced by two solved forms: $X \approx^? Z\beta$, and $Y \approx^? \alpha Z$, where Z is a fresh variable. We briefly show below the effects of their propagation, after a return via Step 2. The reasonings are entirely similar to those employed above, so not all details will be given.

(i) A 1-variable equation $\alpha_1 X \approx^? X\beta_1$ will be replaced (after Pruning, if non-failure) by a 1-variable equation of the form: $\alpha_1' Z \approx^? Z\beta_1'$, where: $\alpha_1' = \alpha$ and $|\beta_1'| = |\beta_1|$.
Note: The prefix/suffix lengths remain unchanged.

(ii) A 2-variable equation $\alpha_2 X \approx^? Y\beta_2$ will be replaced (after Pruning, if non-failure) by a 1-variable equation of the form: $\alpha_2' Z \approx^? Z\beta_2'$, where:

- Either ($\alpha_2 = \alpha\alpha_2'$ and $\beta_2 = \beta_2'\beta$) , or ($\alpha = \alpha_2\alpha_2'$ and $\beta = \beta_2'\beta_2$).

Note: The replacing 1*-variable equation could have a prefix or suffix of bigger size than the equation replaced $\alpha_2 X \approx^? Y\beta_2$, in the second case.*

(iii) A 2-variable equation $\alpha_3 X \approx^? V\beta_3$ will be replaced (after Pruning, if non-failure):

- Either by an equation of the form $\alpha_3 Z \approx^? V\beta_3'$, if $\beta_3 = \beta_3'\beta$;
- Or by a fresh solved form $V \approx^? \alpha_3 Z\beta_3'$, if $\beta = \beta_3'\beta_3$.

Note: The replacing equation here could have a suffix of bigger size than the equation replaced, in the second case.

(iv) A 2-variable equation of the form $\alpha_4 Y \approx^? V\beta_4$ is replaced by the 2-variable equation $\alpha_4\alpha Z \approx^? V\beta_4$.
Note: Replacing equation here would have a bigger prefix than the equation replaced.

Note also that the number of unsolved 2-variable equations in U is lowered by (at least) 1, under both Steps 3.b.

Suppose now, that a derived equation e' replaces an equation e under the propagation of a dependency (and after Pruning); let α, β denote respectively the prefix and suffix of the equation e, and α', β' those of e'. The replacing equation e' is said to be in '*excess-size*' wrt the equation it replaces, iff $|\alpha'| > |\alpha|$, or $|\beta'| > |\beta|$, or both.

It is easy to see that the propagation of solved forms of the type $Y \approx^? X$, or of the type $Y \approx^? \gamma$, cannot lead to replacing equations in excess-size. The case analysis I and II above show that, in any run of **A** a 1*-variable equation is never replaced by an* *equation in excess-size*; They also show that replacing equations in excess-size may get derived in the following cases under the runs of **A**:

- a 2-variable equation in *excess-size* can get derived, when a dependency is applied to the lhs (or the rhs) of a 2-variable equation (Case analysis I.(iii), II.(iv).)

- a 1-variable equation in *excess-size* can get derived, when a dependency is propagated onto a 2-variable equation on the same two variables (Case analysis II.(ii)).

- A solved form equation in *excess-size* can get derived when a dependency is propagated onto a solved form for the same variable, or on the lhs or rhs of a 2-variable equation (Case analysis I.(ii), II.(iii)).

The Case analysis I and II above show also that the length of the prefix (or suffix) of any dependency in the problem, initial or derived under the Steps 2 and 3.b in the first run of **A**, is polynomial on $N s$, where N is the total number of equations in the given problem, and s is the maximum length of the prefixes and suffixes in the given equations. $\square$

Appendix B

Automated program verification often relies on decision procedures to check whether an equality is a consequence of other equalities. These procedures are at the core of SMT solvers [2]. In general SMT solvers combine several theories of interest in programming such as linear arithmetic and the theory of lists. Several efficient decision procedures have been proposed for lists on signature $\{car, cdr, cons\}$ (e.g. [21]). We consider here the case of (flat-)lists presented by the *rcons*-axioms.

The *uniform word problem* (UWP) for the *rcons* theory is the following problem: Given a set of equations $\{l_i = r_i\}_{0\leq i\leq n}$ on the signature $\{nil, cons, rcons\}$ extended by a finite set of constants C, to decide whether the equation $\{l_0 = r_0\}$ is entailed by $\{l_i = r_i\}_{1\leq i\leq n}$, for *all instances of the variables* present in these equations, modulo the axioms of the *rcons* theory.

Theorem 5. *UWP for the rcons theory is undecidable.*

Proof. Our reduction is from the reachability problem for deterministic **reversible** two-counter machines, which is undecidable as a consequence of [17], where a universal reversible 2-counter machine has been constructed.

To ease notations, given constants a_i, b_i and list-term L, we will write terms in normal form w.r.t. the rules of *rcons*; e.g., we shall write :

$$\mathsf{cons}(a_1, \ldots (\mathsf{cons}(a_k, rcons(rcons(\ldots rcons(L, b_l) \ldots, b_1))))) \text{ as the string } a_1 \cdots a_k\, L\, b_l \cdots b_1.$$

Given a deterministic **reversible** two-counter machine M with counters C_1, C_2. We represent instruction labels l_i by constants q_i in C. We represent a counter unit by a constant $a \in C$ and an empty counter by a second constant $b \in C$. We formulate now the rewrite rules that simulate the (non stopping) instructions of M:

Machine Instructions	Rules
$l_n^{C_1}$: inc C_1	$q_n \to aq_{n+1}$
$l_n^{C_1}$: if C_1 is non-empty dec C_1	$aq_n \to q_{n+1}$
$l_n^{C_1}$: if C_1 is empty goto l_m	$bq_n \to bq_m$
$l_n^{C_2}$: inc C_2	$q_n \to q_{n+1}a$
$l_n^{C_2}$: if C_2 is non-empty dec C_2	$q_n a \to q_{n+1}$
$l_n^{C_2}$: if C_2 is empty goto l_m	$q_n b \to q_m b$

We denote by R_M the set of ground rules forming the second column above, and let E_M be the corresponding equational theory. Let $\to_M$ be the rewrite relation induced by R_M and $\to_{M,rcons}$ the relation

$$\approx^*_{\{(1),(2)\}} \circ \to_M \circ \approx^*_{\{(1),(2)\}}$$

Note that any term of the form $ba^i q_f a^j b$ is irreducible for $\to_{M,rcons}$. Let $\approx_{M,rcons}$ be the congruence relation generated by equations $\{(1),(2)\} \cup E_M$.

We represent a configuration (q,n,m) of M as the (string-)term $ba^n qa^m b$.

Assume first that M, starting from a configuration (l_0,n,m), reaches a configuration (l_f,k,j). Since each machine step can be mimicked by an equational replacement we have $ba^n q_0 a^m b \approx_{M,rcons} ba^k q_f a^j b$. Note that the *rcons* axioms $(1),(2)$ are essential for moving a label symbol q_i close to b and detect an empty counter.

Conversely, assume that $ba^n q_0 a^m b \approx_{M,rcons} ba^k q_f a^j b$. Let us consider a derivation with a minimal number of E_M steps from $ba^n q_0 a^m b$ to $ba^k q_f a^j b$: $s_0 = ba^n q_0 a^m b \approx \ldots \approx ba^k q_f a^j b = s_f$. By determinism of M we cannot have a subderivation of type: $s \leftarrow_{M,rcons} t \to_{M,rcons} s'$ with $s \neq s'$. By reversibility of M we cannot have a subderivation $s \to_{M,rcons} t \leftarrow_{M,rcons} s'$ with $s \neq s'$. In the two previous cases $s = s'$ is also ruled out by minimality of the derivation length. Hence the only possibilities for the derivation $s_0 = ba^n q_0 a^m b \approx \ldots \approx bq_f b = s_f$ are $s_0 (\to_{M,rcons})^* s_f$ and $s_0 (\leftarrow_{M,rcons})^* s_f$. By reversibility of M, s_f can be reached from s_0. $\square$