

ON UNIFICATION MODULO ONE-SIDED DISTRIBUTIVITY: ALGORITHMS, VARIANTS AND ASYMMETRY *

ANDREW M. MARSHALL ^a, CATHERINE MEADOWS ^b, AND PALIATH NARENDRAN ^c

^a University of Mary Washington
e-mail address: marshall@umw.edu

^b Naval Research Laboratory
e-mail address: catherine.meadows@nrl.navy.mil

^c University at Albany–SUNY
e-mail address: dran@cs.albany.edu

ABSTRACT. An algorithm for unification modulo *one-sided* distributivity is an early result by Tidén and Arnborg. More recently this theory has been of interest in cryptographic protocol analysis due to the fact that many cryptographic operators satisfy this property. Unfortunately the algorithm presented in the paper, although correct, has recently been shown not to be polynomial time bounded as claimed. In addition, for some instances, there exist most general unifiers that are exponentially large with respect to the input size. In this paper we first present a new polynomial time algorithm that solves the *decision problem* for a non-trivial subcase, based on a typed theory, of unification modulo one-sided distributivity. Next we present a new polynomial algorithm that solves the decision problem for unification modulo one-sided distributivity. A construction, employing string compression, is used to achieve the polynomial bound. Lastly, we examine the one-sided distributivity problem in the new asymmetric unification paradigm. We give the first asymmetric unification algorithm for one-sided distributivity.

1. INTRODUCTION

Equational unification has long been a core component of automated deduction and more recently has found application in symbolic cryptographic protocol analysis [6]. In particular, the algorithm for unification modulo a *one-sided* distributivity axiom

$$X \times (Y + Z) = X \times Y + X \times Z$$

is an early result by Tidén and Arnborg [22]. More recently this theory has been of interest in protocol analysis due to the fact that many cryptographic operators satisfy this property. Unfortunately the algorithm presented in the paper, although elegant and correct, has recently been shown not to be polynomial time bounded as claimed [17]. In addition, for some instances, there exist most general unifiers (mgus) that are exponentially large with

2012 ACM CCS: [Theory of computation]: Logic.

Key words and phrases: Unification, One-Sided Distributivity, Equational Logic, Asymmetric Unification.

* A preliminary version of a portion of this work appeared as [15].

respect to the input size. In this paper we examine the *decision* problem for one-sided distributivity. More formally we consider the decision problem for *elementary* unification modulo this theory, where the terms can only contain symbols in the signature of the theory and variables. This is the theory considered by Tidén and Arnborg [22]. We first present a new polynomial time algorithm which solves the *decision* problem for a non-trivial subcase, based on a typed theory, of unification modulo one-sided distributivity. This subcase happens to be sufficient to express the negative complexity result in [17]. Next we present a new polynomial algorithm which solves the *decision* problem for unification modulo one-sided distributivity. We employ string compression through the use of straight line programs, which allows us to achieve the polynomial bound. Compression by straight line programs proves to be sufficient for our results, however the use of compression in unification and matching is not novel to this paper. See for example [9] and [12] for some pioneering work on using compression in unification and other related problems.

Since our initial results [15], a new unification paradigm has been developed in [5] and is based on newly identified requirements arising from the symbolic analysis of cryptographic protocols. In order to satisfy these requirements and to apply state space reduction techniques, it is usually necessary for at least part of this state to be in normal form, and to remain in normal form even after unification is performed. This requirement can be expressed as an *asymmetric* unification problem $\{s_1 =^\downarrow t_1, \dots, s_n =^\downarrow t_n\}$ where the $=^\downarrow$ denotes a unification problem with the restriction that any unifier leaves the right-hand side of each equation irreducible. Given our motivation in protocol analysis, our final result is to consider the theory in the newly developed paradigm and give the first asymmetric unification algorithm for one-sided distributivity.

2. PAPER OUTLINE

Let us give a brief preview of the remaining portions of the paper.

- Section 3 presents the preliminary background material.
- Section 4 presents an overview of the complexity result concerning the original Tidén and Arnborg [22] algorithm.
- Section 5 presents the first contribution of this paper. We consider a restricted version of the one-sided distributivity problem, which is still sufficiently expressive to contain the family of problems presented in Section 4. For this new restricted version of the problem we develop a new polynomial time bounded decision algorithm (Algorithm 1). This section also provides an introduction to the methods used to solve the main problem. The solution to the main problem builds on Algorithm 1 primarily by the addition of string compression.
- Section 6 contains the main contribution of this paper. Here we present a new polynomial bounded algorithm (Algorithm 2) for the decision unification problem over a theory of one-sided distributivity. The result is achieved by building on Algorithm 1 and using polynomial methods for solving several problems on compressed strings.
- Section 7 considers the one-sided distributivity problem in the new asymmetric unification paradigm. This new paradigm has only recently been identified (see [5]) and is important to the area of cryptographic protocol analysis. Here we present the first asymmetric unification algorithm for the theory of one-sided distributivity. Although the algorithm is not polynomial bounded, it should (much like the original Tidén and Arnborg algorithm) perform well on most problems. In addition, the algorithm is relatively simple, consisting

of a small set of inference rules (see Figure 10). One could ask why Algorithm 2 is not extended to the asymmetric problem? Unfortunately, the string compression methods required for the polynomial result in Section 6 do not easily extend to encapsulate the additional information needed in an asymmetric unification problem. This remains an open problem.

3. PRELIMINARIES AND GENERAL RESULTS

We use the standard notation of equational unification [2] and term rewriting systems [1]. The set of Σ -terms, denoted by $T(\Sigma, \mathcal{X})$, is built over the signature Σ and the (countably infinite) set of variables \mathcal{X} . The terms $t|_p$ and $t[u]_p$ denote respectively the subterm of t at the position p , and the term t having u as subterm at position p . The symbol of t occurring at the position p (resp. the top symbol of t) is written $t(p)$ (resp. $t(\epsilon)$). The set of positions of a term t is denoted by $Pos(t)$, the set of non variable positions for a term t over a signature Σ is denoted by $Pos(t)_\Sigma$. A Σ -rooted term is a term whose top symbol is in Σ . The set of variables of a term t is denoted by $Var(t)$.

A Σ -substitution θ is an endomorphism of $T(\Sigma, \mathcal{X})$ denoted by $\{X_1 \mapsto t_1, \dots, X_n \mapsto t_n\}$ if there are only finitely many variables X_1, \dots, X_n not mapped to themselves. We call domain of θ the set of variables $\{X_1, \dots, X_n\}$ and range of θ the set of terms $\{t_1, \dots, t_n\}$. Application of a substitution θ to a term t (resp. a substitution ϕ) may be written $t\theta$ (resp. $\phi\theta$) or in functional notation as $\theta(t)$.

Given a first-order signature Σ , and a set E of Σ -axioms (i.e., pairs of Σ -terms, denoted by $l = r$), the *equational theory* $=_E$ is the congruence closure of E under the law of substitutivity. By a slight abuse of terminology, E will be often called an equational theory.

Given an equational theory E , an E -unification problem is a set of equations

$$\mathcal{S} = \{s_1 =^? t_1, \dots, s_m =^? t_m\}$$

A solution to \mathcal{S} , called an *E-unifier*, is a substitution δ such that $\delta(s_i) =_E \delta(t_i)$ for all $1 \leq i \leq m$. A substitution δ is *more general modulo E* than θ on a set of variables V , denoted as $\delta \leq_E^V \theta$, if and only if there is a substitution τ such that $\delta\tau(X) =_E \theta(X)$ for all $X \in V$. Two substitutions θ_1 and θ_2 are *equivalent modulo E* on a set of variables V , denoted as $\theta_1 \equiv_E^V \theta_2$, if and only if $\theta_1(X) =_E \theta_2(X)$ for all $X \in V$. For a substitution θ and a set of variables V , $\theta|_V$ denotes the restriction of the substitution to the variables in V , i.e.,

$$\theta|_V = \{X \mapsto \theta(X) \mid X \in V\}$$

We call a set Γ of substitutions a *complete set of E-unifiers* of \mathcal{S} if and only if (i) for every $\theta \in \Gamma$, θ is an E -unifier and (ii) for every E -unifier θ , there is a substitution $\delta \in \Gamma$ where $\delta \leq_E^{Var(\mathcal{S})} \theta$ holds. A complete set of E -unifiers Γ of a unification problem \mathcal{S} is *minimal* if and only if for any two E -unifiers δ and θ in Γ , $\delta \leq_E^{Var(\mathcal{S})} \theta$ implies that $\delta = \theta$.

Equational unification problems are classified based on the function symbols that appear in them, i.e., their signature (Sig). An E -unification problem S is *elementary* if and only if $Sig(S) = Sig(E)$. S is called an E -unification problem *with constants* if $Sig(S) \setminus Sig(E)$ contains only free constants. Finally, if there are uninterpreted function symbols in $Sig(S) \setminus Sig(E)$, S is called a general E -unification problem.

of equations, where each equation is in one of the following forms:

$$X =^? Y, X =^? Y + Z, \text{ and } X =^? Y \times Z$$

A simple decomposition algorithm can transform a set of equations into the above form and maintain unifiability (see [22]).

4.1. The Tidén-Arnborg Algorithm. In [22] Tidén and Arnborg developed an elegant algorithm which is based on the following results.

Theorem 4.1. (Tidén and Arnborg [22])

In the theory of one-sided distributivity:

- (1) *The set of equations $\{U =^? X_1 \circ X_2, U =^? T_1 \circ T_2\}$ has precisely the same unifiers as the set of equations $\{U =^? X_1 \circ X_2, X_1 =^? T_1, X_2 =^? T_2\}$, where \circ is \times or $+$.*
- (2) *Every unifier for the set of equations $\{U =^? V \times W, W =^? W_1 + W_2, X =^? V \times W_1, Y =^? V \times W_2\}$ is a unifier for the set of equations $\{U =^? V \times W, U =^? X + Y\}$, where W_1 and W_2 are fresh variables.*

The key steps in the algorithm can be described by the deduction rules of Figure 1.

(a)	$\frac{\{U =^? V\} \uplus \mathcal{EQ}}{\{U =^? V\} \cup \{U \mapsto V\}(\mathcal{EQ})} \quad \text{if } U \text{ occurs in } \mathcal{EQ}$
(b)	$\frac{\mathcal{EQ} \uplus \{U =^? V \times W, U =^? X \times Y\}}{\mathcal{EQ} \cup \{U =^? V \times W, V =^? X, W =^? Y\}}$
(c)	$\frac{\mathcal{EQ} \uplus \{U =^? V + W, U =^? X + Y\}}{\mathcal{EQ} \cup \{U =^? V + W, V =^? X, W =^? Y\}}$
(d)	$\frac{\mathcal{EQ} \uplus \{U =^? V \times W, U =^? X + Y\}}{\mathcal{EQ} \cup \{U =^? V \times W, W =^? W_1 + W_2, X =^? V \times W_1, Y =^? V \times W_2\}}$

Figure 1: Tidén and Arnborg Inference Rules

The W_1, W_2 in rule (d) are fresh variables and \uplus is *disjoint union*. Furthermore, rule (d) (the “splitting rule”) is applied only when the other rules cannot be applied. A set of equations is said to be *abc-reduced*¹ if and only if none of the rules (a), (b) and (c) can be applied to it. A *sum transformation* is defined as a binary relation between two abc-reduced systems, S_1 and S_2 , where S_2 is obtained from S_1 by applying rule (d), followed by exhaustive applications of rules (a), (b) and (c). Clearly, a sum transformation is applicable if and only if some variable occurs as the left-hand side in more than one equation.

The algorithm also makes use of two graphs. The graphs are used to detect two types on non-unifiability errors. We include the definitions here for completeness.

¹Such a system of equations is called *simple* in [22]. However, simple has come to denote a theory that is subterm collapse free (see [3]).

Definition 4.2. The *dependency graph* ($D(S)$) of an abc-reduced system, S , is an edge labeled, directed multi-graph. It has as vertices the variables of S . For an equation $X = Y + Z$ in S it has an l_+ -labeled edge (X, Y) and an r_+ -labeled edge (X, Z) . An equation $X = Y \times Z$ similarly generates two edges with labels l_\times and r_\times .

Definition 4.3. The *sum propagation graph* ($P(S)$) of an abc-reduced system S is a directed simple graph. It has as vertices the equivalence classes of the symmetric, reflexive, and transitive closure of the relation defined by the r_\times -edges in the dependency graph of S . It has an edge (V, W) iff there is an edge in the dependency graph from a vertex in V to a vertex in W with label l_+ or r_+ .

It can be seen that by using cycle checking on $D(S)$ we can detect all the occur-check like errors that may develop as the algorithm works with the system of equations. We know these are indeed errors due to the following property.

Theorem 4.4. *The one-sided distributive axiom is subterm-collapse free.*

Proof. If we consider the convergent system

$$X \times (Y + Z) \rightarrow X \times Y + X \times Z$$

we can see that the rule is non-size-reducing. Therefore, we cannot reduce a term t to a subterm of itself. \square

This implies that the system is simple ([3]) and therefore occur-checks must be detected as they imply non-unifiability.

The propagation graph is needed to detect non-unifiable systems that cause infinitely many applications of the splitting rule (d). An example of this type of system is the following two equations:

$$Z =^? V_2 + V_3, Z =^? V_1 \times V_3.$$

These types of systems are shown not to have a finite unifier [22]. However, they will never produce a cycle in the dependency graph, thus the propagation graph is needed.

We can conclude this overview of the original algorithm with some of the results proven for it in [22]:

Theorem 4.5. *From Tidén and Arnborg [22]:*

- (1) *The algorithm formed by applying the sum transformation with the rules of Figure 1 is sound, complete and terminating.*
- (2) *If the system is not unifiable either the dependency graph (Definition 4.2) or the propagation graph (Definition 4.3) will contain a cycle after a finite number of steps.*
- (3) *If either the dependency or the propagation graph contain a cycle, the initial system is not unifiable.*
- (4) *The algorithm produces a final solved form, which provides a unique most general unifier for the initial system.*

4.2. Complexity Result. In [17] a family of *unifiable, abc-reduced* systems is presented, on which the Tidén-Arnborg algorithm runs in exponential time.

Definition 4.6. [17] Let EQ be a subset of the set of abc-reduced systems defined as follows: all multiplications are of the form $X_i =^? T \times Y_j$ (or $Y_j =^? T \times X_i$) where T is a unique variable and all additions are of the form $X_i =^? X_{i1} + X_{i2}$ or $Y_i =^? Y_{i1} + Y_{i2}$.

That is variables are represented using X and Y along with subscripts. The actual family of instances that causes the exponential growth is a subset of EQ defined as:

Definition 4.7. [17] For $n \geq 0$, let $\sigma(n)$ be the set of equations

$$\begin{aligned} X_{1^i} &=^? X_{1^{i+1}} + X_{1^{i2}}, \\ Y_{2^i} &=^? Y_{2^{i1}} + Y_{2^{i+1}}, \\ Y_{2^{i1}} &=^? T \times X_{1^{i2}}, \\ X &=^? T \times Y, \\ X_{1^{i+1}} &=^? X_{1^{i+2}} + X_{1^{i+12}} \end{aligned}$$

for all $0 \leq i \leq n$. Where X_{l^i} denotes i concatenations of $l \in \{1, 2\}$, i.e., $X_{1^3 2} = X_{11112}$.

It is shown in [17] that a system of equations, as defined in Definition 4.7, will result in exponentially many applications of the sum transformation.

The result can be viewed graphically in the following manner. Let variables represent nodes in a graph and create downward edges for variables related by an addition operation and lateral edges for variables related by a multiplication operation (this is essentially the $D(S)$ definition). The edges to the unique variable T will not affect the complexity and so can be ignored. We can see such graphs in the following two examples. Figure 2 represents an initial set of equations and Figure 3 is the same system after application of the Tidén-Arnberg algorithm. Essentially, we see that the new variables and paths that are created at each level of the graph are the cause of the complexity growth and will need to be avoided.

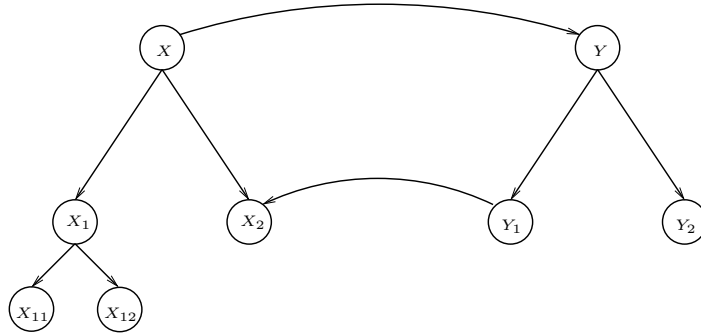


Figure 2: Graph for $\sigma(0)$

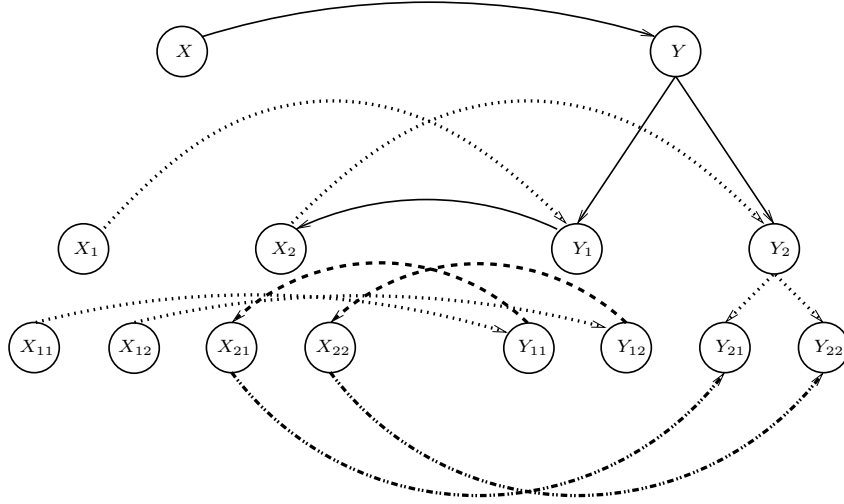


Figure 3: After 4 applications of the sum transformation

In the Tidén-Arnborg algorithm this exponential behavior is due to an exponential number of application of rule (d) from Figure 1. This is the rule that creates the new variables and paths seen in Figure 3. We develop a new algorithm in Section 5 which ensures a polynomial number of application of a rule equivalent to rule (d) from Figure 1 and this algorithm is sufficient to ensure polynomial time and solve the unification problem for the Single Homomorphism (introduced in the next section) restricted form of the problem. However, when applied to the full problem it proves insufficient, see example 6.1. The solution is to introduce the use of string compression, which is done in Section 6.

5. TYPED SYSTEM AND SINGLE HOMOMORPHISM

We present a typed system interpretation of one-sided distributive unification. We begin with the simplest non-trivial subcase, the case of a *single homomorphism*. This is non-trivial because the exponential complexity result in [17] holds in this case as well. Consider a ‘type’ system based on two types τ_1 and τ_2 . We let all left multiplication variables be of type τ_1 and all right variables of type τ_2 . Thus

$$\begin{aligned} \times &: \tau_1 * \tau_2 \rightarrow \tau_2, \\ + &: \tau_2 * \tau_2 \rightarrow \tau_2, \end{aligned}$$

If there is only a single variable of type τ_1 in the input equations then we can consider the multiplication operation as a homomorphism h over $+$. Thus, we can view an equation of the form $X = T \times Y$, where T is the single variable of type τ_1 , as the homomorphism equation $X = h(Y)$. This is the single homomorphism case, it restricts the number of valid terms from the general case but it is sufficient for encoding the exponential example in [17] and it yields a much simplified decision algorithm.

5.0.1. *Single homomorphism and the General Algorithm.* There are two primary reasons for considering this sub-case:

- (1) The Algorithm for the single homomorphism case (Algorithm 1) is more efficient than the algorithm that solves the general case (Algorithm 2).

This is due to the *SLPs*. In Algorithm 2, every step dealing with compression must use *SLPs* and thus must employ various subroutines for dealing with *SLPs*, of which the best complexity measures are all of quadratic or greater polynomial complexity. However, in this restricted case binary encoding provides suitable compression. Operations dealing with compressed objects are reduced to addition and subtraction, i.e., linear complexity.

- (2) Algorithm 2 is built from Algorithm 1.

Algorithm 1 uses the same underlying method used in Algorithm 2. Both algorithms approach the problem by ordering the equivalence classes (defined below) and “processing” each class, one at a time. However, the processing is less complex in this restricted theory since it does not need to deal with *SLPs*. This leads to a similar algorithm which is easier to understand.

5.1. Data Structures.

Definition 5.1. We define the following relations (X, Y and Z are variables):

- $X \succ_h Y$ if $X = h(Y)$.
- $X \succ_{l_+} Y$ if $X = Y + Z$.
- $X \succ_{r_+} Z$ if $X = Y + Z$.
- $X \succ_a Z$ if $X = Y + Z$ or $X = Z + Y$.

We use the following two graphs, that are similar to the dependency and propagation graphs used in [22], see Definitions 4.2 and 4.3. For a unification problem S in standard form we construct the following two graphs.

Definition 5.2. A *path labeled dependency graph* (\mathcal{LD}) is a directed graph such that the nodes in the graph correspond to variables of type τ_2 . We form two kinds of edges:

- (i) *Lateral edges*, where for each equation of the form $X =^? h(Y)$, we have an edge from node X to node Y labeled with a *label variable*, h^1 . Thus, for single edges corresponding to a single homomorphism the label is h^1 . For paths corresponding to multiple homomorphisms (compound paths which will be constructed during the running of the algorithm) the label is h^n , $n \in \mathbb{N}^+$, where n is the number of homomorphisms/single edges composing the path. We will use π (possible with subscripts) to denote a path in the graph. For example, for a path between nodes X and Y we write, $X \xrightarrow{\pi} Y$. Where π is understood to represent some h^j , $j \in \mathbb{N}$. The path length, denote $|\pi|$, is j .

- (ii) *Downward edges*, where for each equation of the form $X =^? X_1 + X_2$, we have directed edges from node X to node X_1 and from node X to node X_2 .

Definition 5.3. The *path labeled propagation graph* (\mathcal{LP}) is a directed simple graph. Its vertices are the equivalence classes of the symmetric, reflexive, and transitive closure of the relation defined by \succ_h on the \mathcal{LD} graph for the same system. Edges exist between equivalence classes $[X]$ and $[Y]$ if there exist variables $U \in [X]$ and $V \in [Y]$ such that $U \succ_a V$.

Note, that we can order the equivalence classes/nodes of \mathcal{LP} . Let \sim_h stand for the reflexive, symmetric and transitive closure of \succ_h . Thus \sim_h defines a set of equivalence classes over a set of variables. Denote these classes as $[Y]_h$. The \mathcal{LP} graph has exactly these classes as its nodes. We can define a strict partial ordering \succ_h on the \sim_h -equivalence classes based on \succ_a . That is, $[X]_h \succ_h [Y]_h$ if and only if there exist $K_1 \in [X]_h$ and $K_2 \in [Y]_h$ such that $K_1 \succ_a K_2$, i.e., an edge from the node $[X]_h$ to the node $[Y]_h$. This ordering will be important as it provides an ordering strategy for applying the unification algorithm.

These graphs, mainly the \mathcal{LD} , will be the primary data structure and will be modified via the set of graph *saturation* rules. The rules are very similar to the original Tidén-Arnberg rules however they primarily act not on the set of equations but on the \mathcal{LD} graph. This is due to the need for compression, where acting on a fully uncompressed set of equations results in the original algorithm. Note, that we still need the \mathcal{LP} graph for detecting the set of non-unifiable systems. An example of this is the following set of equations.

$$\{X =^? V + Y, X =^? h(Y)\}$$

The \mathcal{LP} graph and the sum propagation graph of [22] (Definition 4.3) are the same for the single homomorphism systems. This is easy to see as both graphs will contain the same equivalence classes and thus nodes and both graphs have the same edges. Therefore, each time the algorithm updates the \mathcal{LD} graph (i.e., the inference rules modify the \mathcal{LD} graph) it also updates the \mathcal{LP} graph and checks for cycles. Likewise, if cycles are found the algorithm terminates with failure.

5.2. Algorithm Presentation. Before presenting the rules, we need to discuss several problems the algorithm needs to solve when dealing with compressed paths. During saturation we derive *path constraints* of the form $\pi_1 =^? \pi_2$ or $\pi_1 \prec^? \pi_2$. For the single homomorphism case, because there is just one homomorphism, $\pi_1 =^? \pi_2$ is simply a check if the lengths are equal, i.e., if $|\pi_1| = |\pi_2|$. For the prefix check $\pi_1 \prec^? \pi_2$, in the single homomorphism case we only need to check if the length of π_1 is less than π_2 , i.e., $|\pi_1| < |\pi_2|$. It is important to note that path lengths are kept in binary representation. This compression is significant as it allows us to avoid exponential growth in the path lengths. In addition to path constraints we will need to perform several *path computations*, specifically we need to *concatenate paths* and *compute path suffixes*. These operations can be accomplished, in the single homomorphism case, by simple addition and subtraction.

We now introduce a set of inference rules. *Rule (0) acts on the system S and rules (i) through (vii) act on the \mathcal{LD} graph of S .* Rule (0) is simple variable replacement. Rules (i) - (iii) are cancellation rules that follow directly from the rules of Figure 1. Rule (vi) is a failure rule that corresponds to occur-check type errors. Rules (iv), (v), (vii) are path completion rules. Rule (vii) is the same path propagation rule from the Tidén-Arnberg algorithm, justified by the axioms of the system; see Figure 1 rule(d). *However, in rule (vii) we do not create the new variables W_1 and W_2 unless W has no child variables related along a \succ_a edge.*

Before giving the algorithm details let us give a high-level overview of the process.

- (1) The algorithm begins with a unification problem, S , in standard form.
- (2) From the set of equations S it generates the \mathcal{LD} graph and from the \mathcal{LD} graph it generates the \mathcal{LP} graph.
- (3) Next the algorithm applies the set of “cancellation” inference rules. These are rules which do not create new edges or nodes in the graph and clearly terminate.

(0)	$\frac{S \uplus \{U =? V\}}{\{U \mapsto V\}(S) \cup \{U =? V\}}$	if U occurs in S
(i)	$\frac{U =? U_1 + U_2, U =? U_3 + U_4,}{U =? U_1 + U_2, U_3 = U_1, U_4 = U_2}$	
(ii)	$\frac{X \xrightarrow{\eta} Y, \quad Z \xrightarrow{\pi} Y \quad \eta = \pi }{X =? Z, \quad Z \xrightarrow{\pi} Y}$	
(iii)	$\frac{X \xrightarrow{\eta} Y, \quad X \xrightarrow{\pi} Z, \quad \eta = \pi }{X \xrightarrow{\pi} Z, \quad Y =? Z}$	
(iv)	$\frac{X \xrightarrow{h^j} Y, \quad X \xrightarrow{h^i} Z, \quad j < i}{Y \xrightarrow{h^{i-j}} Z, \quad X \xrightarrow{h^i} Z}$	
(v)	$\frac{X \xrightarrow{h^i} Y, Y \xrightarrow{h^j} Z}{X \xrightarrow{h^{i+j}} Z, \quad Y \xrightarrow{h^j} Z}$	
(vi)	$\frac{S}{FAIL}$	if \mathcal{LP} or \mathcal{LD} are cyclic
(vii)	$\frac{U \xrightarrow{\eta} W, U =? U_1 + U_2,}{U \xrightarrow{\eta} W, W =? W_1 + W_2, U_1 \xrightarrow{\eta} W_1, U_2 \xrightarrow{\eta} W_2}$	

Figure 4: Inference Rules for the Single Homomorphism Problem.

- (4) The algorithm works in a top down ordering on the equivalence classes, using the relation \succ_h to order the classes. Each class is “processed” using the inference rules. This is done by applying the rules to the nodes in \mathcal{LD} graph which are contained in the current “selected” class.
- (5) After each new class is processed the algorithm applies the cancellation rules and re-checks for any errors.
- (6) During this process two things can happen:
 - (a) Cycles can be found in either graph implying non-unifiability.
 - (b) The inference rules are exhaustively applied and no cycles occur, implying unifiability.

The algorithm for the single homomorphism subcase is presented in Algorithm 1.

We next discuss the correctness and complexity of Algorithm 1; most of these results will follow directly from [22].

5.3. Correctness. Correctness of the inference rules can be assured due to the correctness proof of the algorithm presented in [22] and the following lemmas.

Lemma 5.4. *Soundness of rules (i) through (vii) are direct consequences of the “sum transformation”² method of [22] and variable replacement.*

Proof. The soundness of the rules follow from Theorem 4.1. Therefore, we know that the set of equations $\{X =? Y \circ Z, X =? V \circ W\}$, where \circ is $+$ or \times , has the same solutions as the set

²See Section 4 for the definition of sum transformation.

Algorithm 1 Unification modulo a Single Homomorphism

(Input: A system of equations in standard form)

(1: Generate data structures) Generate the graphs, \mathcal{LD} and \mathcal{LP} .

(2: Clean up the system) Exhaustively apply the rules (0), (i), (ii), (iii) and (iv).

(3: Error checking) Apply graph cycle checking to the two graphs (i.e., rule (vi)). If a cycle is found *stop* with failure.

(4: Process equivalence class) Select an equivalence class based on the strict partial ordering \succ_h . That is, we select the largest element of \succ_h that has not yet been processed. Thus, if we select the class $[X]_h$ then there does not exist a class $[Y]_h$ such that $[Y]_h$ has not been processed and $[Y]_h \succ_h^+ [X]_h$. Clearly, if \succ_h is not a strict partial ordering then there is a cycle in the \mathcal{LP} graph.

First we apply rule (v) — this is done by starting with the sink node of the path and working back to the start node of the path. Once rule (v) has been exhaustively applied we apply rule (vii) if applicable.

(5: Check if Complete) If no inference rules can be applied and no cycles exist, then exit with success, else return to Step 2.

$\{X =^? Y \circ Z, Y =^? V, Z =^? W\}$, that the set of equation $\{X =^? Y \times Z, X =^? V + W\}$ has the same solutions, over the shared variables, as the set $\{X =^? Y \times Z, Z =^? V_1 + V_1, W =^? Y \times V_1, V =^? Y \times V_2\}$. \square

The \mathcal{LD} , and \mathcal{LP} , graphs are simply graphical representations of a system of equations, which Algorithm 1 transforms by application of one or more of the inference rules. Lemma 5.4 ensures that each transformation is sound. It remains to be shown that if the algorithm terminates without failure then the system is indeed unifiable.

Lemma 5.5. *Given a system of equations S in standard form if no failure errors occur Algorithm 1 transforms S , through its \mathcal{LD} graph representation, into dag-solved form.*

Proof. Let D be the final \mathcal{LD} graph and consider the definition of dag-solved form.

- The first condition is satisfied as each variable is represented by a node in the graph. If the left hand sides X_i were not distinct, then a cancellation or path propagation rule, (vii), could be applied.
- The second condition is satisfied as the paths correspond to a distinct ordering and there are no cycles in the graph. \square

Therefore, if the system is unifiable the algorithm will report that fact. We need to show that if the system is not unifiable the algorithm correctly reports that as a failure. Directly from [22] we get the following two results.

Lemma 5.6. *Cycles in the \mathcal{LD} graph for a system S in standard form imply that S is not unifiable.*

Proof. This is due to Theorem 4.4, which shows that the one-sided distributive axiom is subterm-collapse free. The constraint to a typed system does not remove the property that the system is simple. Therefore, a cycle in the \mathcal{LD} graph will imply a cycle in the system of equations and a non-unifiability error for a simple system. \square

Lemma 5.7. *Cycles in the \mathcal{LP} graph for a system S , in standard form, imply that S is not unifiable.*

Proof. The \mathcal{LP} graph contains the same information, for the single homomorphism systems, as is contained in the propagation graph of [22] (Definition 4.3). Both graphs will contain the same equivalence classes and thus nodes and both graphs have the same edges. The result then follows from Theorem 4.5. \square

Theorem 5.8. *Algorithm 1 is correct.*

Proof. Follows from Lemma 5.4 to Lemma 5.6. \square

5.4. Complexity. First we get the following result from the cancellative nature of the rules (i) through (iii).

Lemma 5.9. *Given a \mathcal{LD} graph rules (0)-(iii) can only be applied a polynomial number of times with respect to the initial set of nodes in the graph.*

In addition, we get the following clear result.

Lemma 5.10. *Given a \mathcal{LD} graph rule (iv) can only be applied a polynomial number of times with respect to the initial set of nodes in the graph.*

Lemma 5.11. *Each equivalence class formed by closure along \succ_h -related nodes has a unique sink.*

Proof. If a class has no sink then there is a cycle and the system is not unifiable. Now assume we have at least one sink. Rules (iii), (ii), (iv) and (v) ensure that each node can have at most one lateral outgoing edge. \square

Lemma 5.12. *Processing an equivalence class (Step 4) takes polynomial time with respect to the number of variables in the class.*

Proof. By rules (i) through (iv) each variable in the class will have at most one outgoing edge and all paths will lead to the sink. Applying (v) exhaustively starting from the sink is therefore bounded linearly by the number of variables in the class. In addition, (vii) is also bounded by the number of variables in the class as it can be applied at most once for each variable in the class. Moreover, it can create 2 new variables at most once for each class. \square

Lemma 5.13. *The number of \sim_h -equivalence classes for a system S can never increase.*

Proof. New variables created by rule (vii) don't create new equivalence classes as they are added to pre-existing classes. \square

Due to the fact that each equivalence class contains a single sink, by Lemma 5.11, we get the following.

Lemma 5.14. *A maximum of 2 new nodes can be added to an equivalence class from any one higher equivalence class.*

In addition by rule (vii) we get the following.

Lemma 5.15. *During processing the number of paths added to a \sim_h -equivalence class from a higher, by \succ_h , \sim_h -equivalence class cannot exceed the number of nodes in the lower equivalence class.*

Combining the above results we get the following.

Theorem 5.16. *The running time of Algorithm 1 is polynomial with respect to the initial set of equations.*

Proof. Processing an equivalence class is polynomial bounded by Lemma 5.12.

By Lemma 5.14 and Lemma 5.15 the classes can only grow by a constant amount as each class is processed and by Lemma 5.13 the number of classes cannot increase. \square

This section covers a decision algorithm for the single homomorphism subcase. The obvious extension to this problem results in the *multiple homomorphism* problem. In the multiple homomorphism case we may have a finite set of variables of type τ_1 but we can still consider them as homomorphisms h_1, \dots, h_n . Although we do not go into any more details here, the multiple homomorphism case is also interesting. Unlike the single homomorphism case compression is needed for the multiple homomorphism case. This is due to the fact that, unlike the single homomorphism case, the label variables are not the same and therefore just keeping the path lengths is not sufficient. But, the multiple homomorphism case does not require all the methods presented in the next section for the general case, due to the type system, i.e., labeled variables cannot also be nodes in the \mathcal{LD} graph.

6. GENERAL ALGORITHM

We now consider the general problem, with no type system. Let us give a brief overview of the section.

Section Summary.

- We begin with a discussion on why the string compression methods are required to achieve a polynomial bound, Example 6.1.
- We next introduce the new graph data structures in Section 6.1.
- Section 6.2 provides a high-level overview of the new algorithm.
- Section 6.3 presents the new algorithm.
- Section 6.4 discusses issues with label variables which are a key difference between the general algorithm developed in this section and the Single Homomorphism algorithm developed in the previous section.
- Details on the SLP operations used in the algorithm and their complexity is covered in Section 6.5.
- Correctness is proven in Section 6.6 and 6.7.
- Finally, the complexity proof is covered in Section 6.8.

Example 6.1. One consequence of this new graph interpretation is that the label variable paths, if not compressed, could grow exponentially in length with respect to the initial set of label variables. This can be seen using the same example used to prove the exponential result in Section 4, $\sigma(n)$. If the algorithm presented below (without compression) is applied to the system $\sigma(n)$, we do not get an exponential number of applications of the sum transformation; rather we get label paths of exponential length. The growth is due to the path string being copied and then doubled at each consecutive level. Although this doubling of the string leads to the exponential growth, it also requires the re-use of the string and this suggests the use of string compression. Therefore, we keep each of these paths compressed in the form of straight line programs.

Consider again the $\sigma(0)$ system and assume there is a single label variable, a , for the initial system. If Algorithm 2 is applied but does not use string compression the final length of the string labeling the longest path at level n will be $2^n - 1$. In $\sigma(0)$ (Figure 5) this is $2^2 - 1 = 3$. For larger n the result is undesirably long paths, as seen in Figure 6. However, these strings can easily be compressed via a *SLP*.

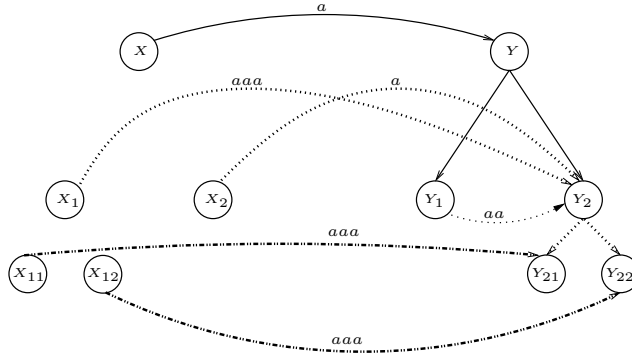


Figure 5: Exponential path length, initial graph

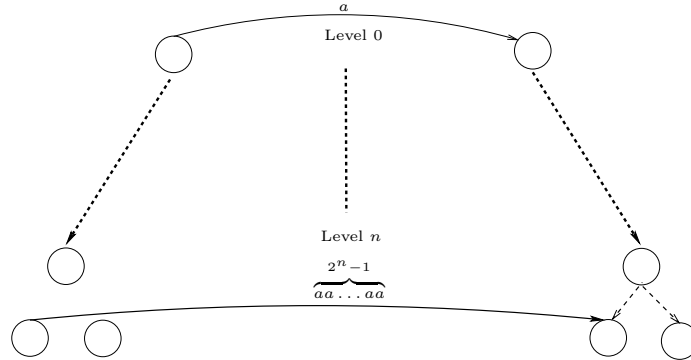


Figure 6: Exponential path length, final graph

6.1. Data Structures. As in the single homomorphism case we interpret the equations of a unification problem as graphs.

Definition 6.2. we define the following relations:

- $X \succ_{r_*} Y$ if $X = Z \times Y$.
- $X \succ_{l_*} Z$ if $X = Z \times Y$.
- $X \succ_m Y$ if $X \succ_{l_*} Y$ or $X \succ_{r_*} Y$.
- $X \succ_{l_+} Y$ if $X = Y + Z$.
- $X \succ_{r_+} Z$ if $X = Y + Z$.
- $X \succ_a Z$ if $X = Y + Z$ or $X = Z + Y$.

We denote the transitive closure of a relation R as R^+ .

Definition 6.3. A *path labeled dependency graph* (\mathcal{LD}) is a directed graph such that the nodes in the graph correspond to variables of the unification problem S . We form three kinds of edges:

(1) *Lateral Edges*, where for each equation of the form $X =^? Z \times Y$, we have an edge from node X to node Y labeled with the top nonterminal of a *SLP* generating the *label variable*, Z . Label variables are kept as *straight line programs*, where the terminals corresponds to the label variable. Each label variable, Z , is given a unique single production *SLP*. Therefore, lateral edge and path labels correspond to the top nonterminal of the *SLP* generating the label variables corresponding to those edges.

We denote a path and its label, π , of one or more lateral edges between nodes X and Y by $X \xrightarrow{\pi} Y$. For the general case paths are the composition of any number of the label variables. A path π is notation for a path X_1, \dots, X_n for some $n \in \mathbb{N}$ and is kept altogether compressed as a *SLP*. Therefore, $X \xrightarrow{\pi} Y$ corresponds to the equations $X =^? \pi \times Y$ and is a compact representation of the equation

$$X =^? X_1 \times X_2 \times \dots \times X_n \times Y$$

where the string generated by π is of the form $X_1 \cdot X_2 \cdot \dots \cdot X_n$

(2) *Downward Edges*, where for each equation of the form $X =^? X_1 + X_2$, we have directed edges from node X to node X_1 and from node X to node X_2 .

(3) *Relation Edges*, where for each node X in the graph such that there exists a path $X \xrightarrow{\pi} Y$ and for each terminal/label variable K_i in the *SLP* π , we have a single edge from X to the *node* K_i in the graph.

These edges will only be used for cycle checking and could even be generated just before the graph is checked for cycles in the algorithm.

We explain several points below that should help clarify the need for such a graph.

- The initial \mathcal{LD} graph will be built from an initial unification problem, S , in standard form. That initial graph will not have any composite paths labeled by a *SLP* with more than one production. The composite paths will be added later by the algorithm. In addition, when constructing the \mathcal{LD} graph each variable X from the set of label variables is given a unique *SLP*. For example, a label variable X would be given a *SLP* $\pi_X \rightarrow X$ and all lateral edges formed by an equation with X as the label variable would be labeled by π_X . This implies that different lateral edges can have the same edge label. For example, in the \mathcal{LD} graph of Figure 7 the edges $X \rightarrow Y$ and $L_2 \rightarrow L_3$ have the same *SLP* label because they used the same label variable in the equations $X =^? Z_1 \times Y$ and $L_2 =^? Z_1 \times L_3$.
- The algorithm presented later will build up the composite paths and unlike in the initial graph it will not in general be the case that for a $X \xrightarrow{\pi} Y$ all the terminals (label variables) in π are \succ_{l_*} related to X . This is the reason for the additional “Relation” edges.
- Lateral Paths are kept as *SLPs* because if not kept compressed the paths could grow exponential in size, during running of the algorithm. By keeping the initial label variables as *SLPs* when we build longer composite paths we can create the new *SLP* labels by “concatenating” the *SLPs*.
- The *Relation* edges are only needed during graph cycle checking operations required by the algorithm. Thus, as the information about these edges is maintained by the set of terminals for each *SLP*, we will just generate these edges before cycle checking.

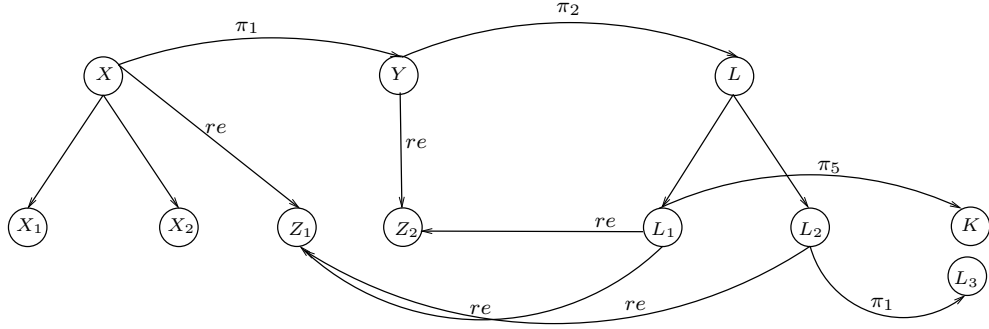


Figure 7: LD graph example

Example 6.4. Let us consider an example \mathcal{LD} graph for the following system of equations (re denotes relation edges).

$X =^? X_1 + X_2$, $X =^? \pi_1 \times Y$, $Y =^? \pi_2 \times L$, $L =^? L_1 + L_2$, $L_1 =^? \pi_5 \times K$, $L_2 =^? \pi_1 \times L_3$ where the $SLPs$ are: $\pi_1 \rightarrow Z_1$, $\pi_2 \rightarrow Z_2$ and $\pi_5 \rightarrow \pi_3\pi_4$, $\pi_3 \rightarrow \pi_2\pi_2$, $\pi_4 \rightarrow \pi_1\pi_1$. The corresponding \mathcal{LD} graph is given in Figure 7.

Definition 6.5. The *path labeled propagation graph* \mathcal{LP} is a directed simple graph. Its vertices are the equivalence classes of the symmetric, reflexive, and transitive closure of the relation defined by \succ_{r^*} on the \mathcal{LD} graph for the same system. Edges exist between equivalence classes $[X]$ and $[Y]$ if there exist variables $U \in [X]$ and $V \in [Y]$ such that $U \succ_a V$.

Similar to the typed case we can order the equivalence classes/nodes of \mathcal{LP} . Let \sim_r stand for the reflexive, symmetric and transitive closure of \succ_{r^*} . Thus \sim_r defines a set of equivalence classes over a set of variables. Denote these classes as $[Y]_r$. The \mathcal{LP} graph has exactly these classes as its nodes. We can define a strict partial ordering \succ_r on the \sim_r -equivalence classes based on \succ_a . That is, $[X]_r \succ_r [Y]_r$ if and only if there exist $K_1 \in [X]_r$ and $K_2 \in [Y]_r$ such that $K_1 \succ_a K_2$. This ordering will again be important as it provides an ordering strategy for applying the rules of the unification algorithm.

Again, we also need the \mathcal{LP} graph due to a specific type of non-unifiable system. These are systems that require infinite unifiers but will always cause a cycle in the \mathcal{LP} graph. An example of this is the following set of equations.

$$\{X =^? X_1 + X_2, X =^? V \times X_2\}$$

Lemma 6.6. *Let \mathcal{S} be a system of equations with variables U, W in \mathcal{S} such that $U \succ_a W$ and $U \succ_{r^*} W$. Then \mathcal{S} is not unifiable.*

Proof. This system can be seen to cause a cycle not only in \mathcal{LP} but also in $P(\mathcal{S})$ (see Definition 4.3). This is due to forming the equivalence classes by closure along \succ_r related nodes. It is shown in [22] Lemma 11, if $P(\mathcal{S})$ contains a cycle there is no unifier for the system \mathcal{S} (See Theorem 4.5). \square

Each time the algorithm updates the \mathcal{LD} graph it also updates the \mathcal{LP} graph and checks for cycles. Likewise, if cycles are found the algorithm terminates with failure.

It can be seen that the propagation graph of [22] and the \mathcal{LP} graph are the same. This is due to fact that both graphs contain the same equivalence classes and equivalent edges

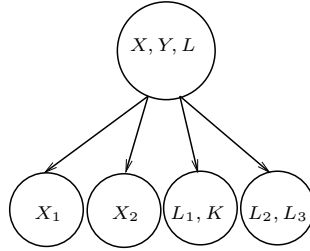


Figure 8: LP graph example

between the classes. Figure 8 is an example \mathcal{LP} graph for the same set of equations used to form the graph of Figure 7.

The algorithm will work by “saturating” the graphs. A set of transformation rules is used to either convert the graph into a solved form or detect a cycle in the graph. The first case implies unifiability and the second non-unifiability. During saturation we derive *path constraints* of the form $\pi_1 =^? \pi_2$ or $\pi_1 <^? \pi_2$. The constraint $\pi_1 <^? \pi_2$, is a prefix check (i.e., whether the string produced by the *SLP* π_1 is a prefix of the string produced by the *SLP* π_2) and $\pi_1 =^? \pi_2$, similarly, is an equality check. In addition to path constraints we will need to perform several *path computations*: specifically we need to *concatenate paths*, *compute path suffixes* and find a single pair of *mismatched terminals* in two equated *SLP* produced strings, all without decompressing the *SLPs*.

6.2. High Level Overview. Before giving the inference rules and algorithm details let us give a high-level overview of the process. The algorithm works based on the idea of collecting equations into sets. Each set correspond to the equations forming one of the equivalence classes. We can then order the equivalence classes in such a way that if we proceed top down in this order, converting each set into a solved form (processing), we do not have to revisit any class. This combined with the fact that we don’t create new classes requiring processing provides us with a well defined structure for the execution of the algorithm. Briefly, the algorithm proceeds as follows:

- The algorithm begins with a unification problem, S , in standard form. From the set of equations S it generates a graph interpretation, the \mathcal{LD} graph and makes note of the set of label variables, \mathcal{V} . Note that this process does not discard S . From the \mathcal{LD} graph the algorithm generate the \mathcal{LP} graph. The nodes of the \mathcal{LP} graph are the equivalence classes, each of which corresponds to a set of nodes from the \mathcal{LP} graph.
- Next the algorithm reduces the graph to a normal-form by applying the set of cancellation inference rules. These rules are applied to the entire system which results in a system where no additional cancellation rules can be applied. All of these rules work directly on the \mathcal{LD} graph, except rule (0) which works on the set S .
- Next the algorithm selects an equivalence class to process based on the class ordering. It then applies a set of rules on the nodes of the \mathcal{LP} graph which correspond to that equivalence class. The effect of these rules is to transform the equivalence class into a compressed solved form.
- This process is repeated along with error checking, each time reducing the number of classes remaining to be processed.

- Finally, it could happen that during the process two label variables are equated. If this occurs the algorithm updates S with the new equality (rule (0)), rebuilds the two graphs and the process of processing equivalence classes is restarted. Since there is only a finite number of label variables, which cannot be increased, each time two label variables are equated the number of label variables is reduced. Therefore the total number of times the process can be restarted is equal to the initial number of label variables in S .

6.3. Algorithm Presentation. We first present the set of inference rules (Fig 9) for a unification problem S in standard form. The rules are applied to the graph \mathcal{LD} , except rule (0) which is applied to S , and as that graph is updated the \mathcal{LP} graph is updated.

(0)	$\frac{S \uplus \{U =? V\}}{\{U \mapsto V\}(S) \cup \{U =? V\}}$	if U occurs in S
(i)	$\frac{U =? U_1 + U_2, U =? U_3 + U_4}{U =? U_1 + U_2, U_3 = U_1, U_4 = U_2}$	
(ii)	$\frac{X \xrightarrow{\pi} Y, \quad Z \xrightarrow{\eta} Y \quad \eta = \pi}{X =? Z, \quad X \xrightarrow{\pi} Y}$	
(iii)	$\frac{X \xrightarrow{\eta} Y, \quad X \xrightarrow{\pi} Z, \quad \eta = \pi}{Y =? Z, \quad X \xrightarrow{\pi} Z}$	
(iv)	$\frac{X \xrightarrow{\eta} Y, \quad X \xrightarrow{\pi} Z, \quad \eta \prec \pi}{Y \xrightarrow{\eta^{-1}\pi} Z, \quad X \xrightarrow{\pi} Z}$	
(v)	$\frac{X \xrightarrow{\eta} Y, \quad X \xrightarrow{\pi} Y, \quad \eta \neq \pi}{\eta =? \pi}$	
(vi)	$\frac{X \xrightarrow{\eta} Y, \quad X \xrightarrow{\pi} Z, \quad \eta \not\prec \pi}{\eta \prec? \pi}$	if $ \eta < \pi $
(vii)	$\frac{U \xrightarrow{\eta} W, U =? U_1 + U_2, W =? W_1 + W_2}{W =? W_1 + W_2, U_1 \xrightarrow{\eta} W_1, U_2 \xrightarrow{\eta} W_2, U \xrightarrow{\eta} W}$	
(viii)	$\frac{U \xrightarrow{\eta} W, U =? U_1 + U_2,}{W =? W_1 + W_2, U_1 \xrightarrow{\eta} W_1, U_2 \xrightarrow{\eta} W_2, U \xrightarrow{\eta} W}$	
(ix)	$\frac{S}{FAIL}$	if \mathcal{LP} or \mathcal{LD} are cyclic
(x)	$\frac{X \xrightarrow{\pi} Y, Y \xrightarrow{\eta} Z}{X \xrightarrow{\tau=(\pi\eta)} Z, \quad Y \xrightarrow{\eta} Z}$	

Figure 9: Inference Rules for the One-sided Distributivity Decision Procedure.

Rule Zero. Rule (0) is simply a variable replacement rule but it has a special action on label variables: *if a label variable is equated to a non-label variable, then the non-label variable is replaced by the label variable.* This rule acts directly on the system S by doing variable replacement whenever there is an equation between two variables. That is, for an equation of the form $U = V$ between two variables, the rule replaces all occurrences of V in S with U . If one of the variables is a label variable, say V , and one is not, say U , then the non-label variable is replaced in S by the label variable. So in this example, all occurrences of U are replaced by V . Therefore, after any variables are equated, we apply this rule *eagerly*. Note, *whenever a rule creates an equation of the form $X =^? Y$, those two nodes in the graph are equated and rule (0) applies that equation to the set S .* Therefore, rule (0) is the only rule that acts on and changes the set of equations S . All other rules modify the two graph data structures.

Rule (i) is due to the cancellative nature of the $+$ operator and directly corresponds to the canceling operation in [22] (see Figure 1). Rules (ii), (iii) and (iv) are due to the cancellative nature of \times ([22]). Rules (v) and (vi) check the path constraints and attempt to find label variables that have to be equated in order to satisfy the path constraint. These rules and rules (iv) and (x) are explained in more detail in Section 6.5. Rules (vii) and (viii) directly correspond to the splitting rule (Rule (d) of Figure 1) of [22] and are direct consequences of the distributive axiom. These two versions are just modifications to work in the modified graph setting. The difference between the two rules is that rule (viii) creates new variables (W_1 and W_2) and rule (vii) does not. Rule (ix) is a failure rule, which corresponds to detecting a cycle in the graphs in the Tidén-Arnborg algorithm. Finally, rule (x) is a path completion rule, justified by the soundness of variable replacement. This rule is also responsible for building the *SLPs* with more than one production. The rule creates a new *SLP*, τ , corresponding to the “concatenation” of the two *SLPs* π and η . More details on rule (x) are given in Section 6.5.

We also keep and update the length of the string each *SLP* generates. Note, that this information can be efficiently computed in a bottom up manner for any *SLP* since productions ending in a terminal symbol have string length 1 and productions with two non-terminals have length equal to the sum of the lengths of the strings generated by the two non-terminals. However, since we build our *SLP* bottom up we can keep track of this information using simple addition and subtraction when constructing new *SLPs* through concatenating and taking the suffix.

Algorithm 2 uses the following notation. Let r_1 and r_2 denote inference rules. Then, $r_1^!$ indicates *exhaustive* application of the rule r_1 . The composite rule $r_1^!r_2$ means, apply r_1 until it cannot be applied any more and then try to apply r_2 . Note that even if r_1 cannot be applied the rule $r_1^!r_2$ can still be used if r_2 can be applied. Thus $r_1^!$ does not indicate that r_1 *must* be applied but rather that if r_1 can be applied we do so exhaustively. $r_1 + r_2$ indicates choice: apply rule r_1 *or* rule r_2 . Therefore, the last composite rule in Algorithm 2 implies that rule (vii) has the lowest priority and that rule (viii) is only applied *once* in the processing of a single equivalence class.

Algorithm 2 is presented above. Let us now give some additional explanation of what each step accomplishes before proceeding to the proof details.

- (1) The first generates the two data structures needed to check error conditions.

³Again, if \succ_r is not strict partial ordering there must be a cycle in the \mathcal{LP} graph

Algorithm 2 One-sided Distributive Unification

(Input: A system of equations in standard form)

(1: Generate data structures) Generate the 2 graphs, \mathcal{LD} and \mathcal{LP} . Make a note of the initial label variables in S ; denote this set as \mathcal{V} .

(2: Clean up the system) Exhaustively apply the following composite rule:

$$(0 + i + ii + iii)$$

(3: Error checking) Apply graph cycle checking to the graphs (i.e., rule (ix)). If a cycle is found *stop* with failure. If the graphs have no cycles and are in dag-solved form, *exit* with success.

(4: Process equivalence class) Select an equivalence class based on the strict partial ordering \succ_r . That is, we select the largest element of \succ_r that has not yet been processed. Thus, if we select the class $[X]_r$ then there does not exist a class $[Y]_r$ such that $[Y]_r$ has not been processed and $[Y]_r \succ_r^+ [X]_r$ ³.

Process the selected class using the following composite rule:

$$(v + vi)(iv)^!(x)^!(viii)(vii)^!$$

Rule (x) is applied by starting with the sink node of the path and working back to the start node of the path. Rule (iv) is applied based the \succ_{r^} partial ordering, starting from the source nodes and working down to the sinks. In addition, if rule (v) or rule (vi) is applied label variables will be equated.*

(5: Checking) If any of the variables in \mathcal{V} are equated go back to Step 1 else go back to Step 2.

- (2) Step two consists of an exhaustive application of the “cancellation” inference rules, i.e., rules (0) , (i) , (ii) , (iii) . These rules are the simplest rules as they either reduce the number of variables, rule (0) , or reduce the number of edges in the graph, rules (i) , (ii) , (iii) . In addition, the rules don’t create any new SLPs or edges. By applying these inference rules first we reduce the problem to a “normal-form”, where change can now only occur via rules $(iv) - (x)$. Some of the rules must check equality between SLP, which can be done in polynomial time, see section 6.5.2. Lemma 6.13 shows that the inference rules are sound and Lemma 6.25 shows that this step of the algorithm is polynomially bounded.

(a) Note that redundant edges are removed by rule (iii) . That is, if there are two edges

$$X \xrightarrow{\pi} Y, \quad X \xrightarrow{\eta} Y \quad \text{and} \quad \eta = \pi$$

Then, rule (iii) will remove one of the redundant edges.

- (3) Step three is an error checking step which corresponds to the cycles checking of the two graphs. The correctness of the step is due to the fact that the system is simple and therefore cycle, or occur-checks, are errors. See section 6.7 for full proofs.
- (4) Step four is the step responsible for processing equivalence classes and corresponds to the application of the inferences rules (v) , (vi) , (iv) , (x) , $(viii)$ and (vii) in a specific order. The order that rules are applied is important as it ensures that the number of steps required to process any equivalence class is bounded by a polynomial, see Lemma 6.26.

- (5) Step five checks to see if any label variables were equated and if so, it goes back to step one, where we rebuild the graphs. This is done to ensure we don't miss any variables which are part of the compressed labels. Since the number of label variables is reduced at each application, Lemma 6.8 shows the number of times this can happen is bounded by the initial number of label variables.

6.4. Label Variables. We need several results about the label variables and their interaction with the new variables. Let \mathcal{V}_0 denote the set of initial label variables for a system S and \mathcal{V} the set of label variables at any point during the application of Algorithm 2. Let \mathcal{Z} denote the set of fresh variables created by rule (viii) during application of Algorithm 2.

Lemma 6.7. *During and after the application of Algorithm 2, $\mathcal{V} \cap \mathcal{Z} = \emptyset$.*

Proof. By the definition, it is not possible to apply rule (0) such that a newly created variable is made a label variable. The only way to make a new variable a label variable is to create a new lateral edge and make its label a new variable. The rules creating new lateral edges are (iv), (vii) and (viii). But, the labels of these edges are all composed of pre-existing label variables. \square

Lemma 6.8. *During and after the application of Algorithm 2 on a system S in standard form, $|\mathcal{V}| \leq |\mathcal{V}_0|$*

Proof. Follows directly from rule (0) and Lemma 6.7. \square

Actually we can get a similar result for the original Tidén-Arnborg algorithm if we also assume that variable replacement in that algorithm replaces newly created variables by original variables. Stated more precisely we get the following lemma.

Lemma 6.9. *Let \mathcal{V}_l denote the set of left multiplication variables, i.e., $Z \in \mathcal{V}_l$ iff there exists an equation of the form $X = Z \times Y$ for some variables X and Y . Let Sum denote the sum transformation operation as defined in [22] (See Section 4 to recall the definition). Also, assume that if a pre-existing variable is equated to a variable created by Sum, the new variable is replaced by the pre-existing one. Then it is never the case that a new variable created by Sum is in \mathcal{V}_l .*

Proof. The Sum operation does not create new left multiplication variables. Therefore, the only way to get a new variable Z into \mathcal{V}_l is by equating it with a variable already in \mathcal{V}_l . \square

Thus, we can assume that there will never be an equation of the form $X = Z \times Y$, where Z is a fresh variable created by Algorithm 2.

6.5. Graph and SLP Operations. We first examine the problem of graph cycle checking and then we cover the details of the SLP operations.

6.5.1. *Graph Cycle Checking and Updating.* The \mathcal{LD} graph is updated by the algorithm as the inference rules operate on it. The \mathcal{LP} graph is built from the \mathcal{LD} graph and thus can be updated after updating the \mathcal{LD} graph. We note that the \mathcal{LD} and \mathcal{LP} graphs can use standard cycle checking algorithms. With the additional observation that we can add the relation edges in polynomial time with respect to the number of nodes in the \mathcal{LD} graph, we get the following Lemma.

Lemma 6.10. *The \mathcal{LP} and \mathcal{LD} graphs for a system S in standard form can be checked for cycles in polynomial time with respect to the size of S .*

6.5.2. *SLP Operations.* Algorithm 2 requires the use of some type of string compression due to the need to keep path labels from growing exponentially. But, we still need to know how the label variables are related along paths, e.g., for error checking. Therefore, we cannot just keep a *set* of the variables forming the path, the terminals of the *SLP*, because this removes essential information. We first examine how the *SLP* are formed and then we will discuss how the operations are used by each rule in Algorithm 2 along with a presentation of their complexity. For convenience, Table 1 gives a listing of the *SLP* algorithms required by Algorithm 2 and a listing of where polynomial time algorithms have been developed and studied for that problem. See [14, 19] for surveys on algorithms on compressed strings, including *SLPs*. More details are given in the following discussion

Forming SLPs. We first encode the label variables as *SLPs*. Each unique label variable is encoded as a unique *SLP*. For example, when creating the \mathcal{LD} graph for two equations $X =? Y * Z$ and $K =? Y * L$ only one *SLP* is created, $\pi_Y \rightarrow Y$, and two edges are labeled by that *SLP*, i.e., by the top nonterminal π_Y . Then, larger or additional *SLPs* are formed, bottom up, by the inference rules (x) and (iv). In addition, *we only keep a single copy of each unique SLP*. This implies we only keep the set of all productions. When creating a new larger *SLP* we need only create a new top production. For example, if we have two pre-existing *SLPs* π_i and π_j that we wish to concatenate we don't need to duplicate all the productions; simply adding a top production $\pi_{(ij)} \rightarrow \pi_i \pi_j$ to the set of productions is sufficient. Likewise, when constructing a suffix, we may need to create new productions that are added to the set of productions but we do not delete the productions contained in the prefix since they generate other *SLPs*. Note that rules (vii) and (viii) don't create new *SLPs* but just use pre-existing ones.

<i>SLP operations required by Algorithm 2</i>	<i>Reference to Polynomial Algorithm</i>
The concatenation of two <i>SLP</i>	[4, 11, 12]
<i>SLP</i> equality	[13, 16, 20, 18]
<i>SLP</i> prefix and suffix	[13, 8, 9, 12]
Find one pair of non-equal terminals in a pair of non-equal <i>SLPs</i>	[8, 13, 16]

Table 1: Algorithms for *SLP* Operation

Using SLPs. We now examine the *SLP* operations used by each rule and their complexity. Rule (x): Rule (x) forms a new *SLP* by concatenating two existing *SLPs*. *Rule (x) is applied by starting with the sink node of the path and working back to the start node.*

This ensures a minimal number of applications of rule (x). To concatenate two *SLPs*, $I = (\Sigma, N_I, P_I)$ and $J = (\Sigma, N_J, P_J)$,⁴ we create a new *SLP*, $K = (\Sigma, N_K, P_K)$. Let π_I and π_J be the top nonterminals of I and J respectively. Then, $N_K = N_I \cup N_J \cup \{\pi_K\}$ and $P_K = P_I \cup P_J \cup \{\pi_K \rightarrow \pi_I \pi_J\}$. This is a simplified version, with just two *SLPs*, of the method presented in [12], for concatenating n strings. There it is shown, by a constructive proof,⁵ that the *SLP*, G , generating the new string satisfies $|G'| \leq |G| + n - 1$ and $\text{depth}(G') \leq \text{depth}(G) + \lceil \log(n) \rceil$. Rule (x) is just concatenating the *SLPs* but we could also balance the resulting *SLP*. It is shown in [20] that for a *SLP*, G , generating a text of length m with n rules we can construct a *SLP*, G' , in $\mathcal{O}(n \log(m))$ time, such that G' has a depth of $\mathcal{O}(\log(m))$ and $\mathcal{O}(n \log(m))$ rules. This could improve results which depend on the depth of a *SLP*. However, for our purposes we will use the simple concatenation method as it is sufficient for our results and allows for a simpler complexity analysis. The following result easily follows.

Lemma 6.11. *Let $I = (\Sigma, N_I, P_I)$ and $J = (\Sigma, N_J, P_J)$ be two *SLPs*. Then we can construct in linear time, without decompression, a *SLP* $K = (\Sigma, N_K, P_K)$ that generates the concatenation of the two strings generated by I and J such that $|K| = |P_I \cup P_J| + 1$ and $\text{depth}(K) \leq \max\{\text{depth}(I), \text{depth}(J)\} + 1$.*

Additional algorithms and notes on concatenation can be found in [4, 11, 20].

Rules (ii) and (iii): These two rules require that we can decide if two compressed strings are equal, $\pi_1 \stackrel{?}{=} \pi_2$. The area of fully compressed pattern matching is an active area and there are many algorithms that will solve this problem in polynomial time ($\mathcal{O}(n^3)$ time for a *SLP* of size n [13]). We cite the following, non-exhaustive, list of papers for excellent algorithms; [13, 16, 20, 18].

Rule (iv): We can partially order the nodes in each equivalence class based on the lateral edges, i.e., based on the \succ_{r_*} relation. *Rule (iv) is applied based on this partial ordering, starting from the source nodes and working down to the sinks.*

We do not apply rule (iv) to a node X if rule (iv) can be applied to a node Y such that there is a lateral path from Y to X . Rule (iv) requires that we can decide if one *SLP* π_1 is a prefix of an *SLP* π_2 , $\pi_1 \prec^? \pi_2$, in polynomial time with respect to π_2 . This problem has been solved in [13], $\mathcal{O}(n^3)$ time for a *SLP*, π_2 , of size n . We also need to extract the suffix in compressed form, $\pi_3 = \pi_1^{-1} \pi_2$. Because we build the *SLPs* bottom up and keep the length information. A simple polynomial-time recursive algorithm can accomplish this. See also [8, 12, 13, 21] for additional efficient methods for computing the suffix (and prefix). For example, it has been shown in [12] that if G is a *SLP* generating the word v , then for any suffix v' there exists a *SLP* G' that generates v' and satisfies $|G'| \leq |G| + \text{depth}(G)$ and $\text{depth}(G') \leq \text{depth}(G)$. For completeness a simple, but polynomial, suffix algorithm is presented in Appendix A and from this algorithm we have the following result.

Lemma 6.12. *Let $I = (\Sigma, N_I, P_I)$ and $J = (\Sigma, N_J, P_J)$ be two *SLPs* such that the string generated by J is a prefix of the string generated by I . Then, in $\mathcal{O}(|I|^4)$ time a *SLP**

⁴ For every *SLP* the set of terminals will be a subset of Σ , the initial set of label variables.

⁵ We have replaced “singleton context free grammar” with *SLP* in the statement, just to stay consistent with the naming in this paper.

$K = (\Sigma, N_K, P_K)$ can be constructed that generates the suffix of I after removing the prefix J such that $|K| \leq |I| + \text{depth}(I)$ and $\text{depth}(K) \leq \text{depth}(I)$.

Rules (v) and (vi): These rules handle the situation where two label paths should be equal, or one a prefix of the other, but are found not to be. We then need to check if they can be made equal. We accomplish this by finding at least one pair, (X, Y) , of terminals (label variables) in the corresponding *SLPs* such that these terminals form a mismatch, $X \neq Y$. One pair will do for each application of the rule because by the cancellative nature of \times , all mismatched pairs of terminals *must* be equated. Therefore, we do not have to try all different combinations of setting pairs equal or unequal. It is sufficient to select the first mismatch, equate the variables and construct the resulting new problem. Note that we are finding a single pair of terminals that form a mismatch in the string, not finding *all* the positions where the strings generated by the *SLPs* differ, a NP-hard problem ([13]). In [8] the authors have developed a nice polynomial, $\mathcal{O}(n^3)$, algorithm for finding the first mismatch. A mismatch can also be found using the algorithms in [13, 16] or by a simple recursive algorithm, using the *SLP* equality algorithm of [13] as a subroutine. The result is a simple $\mathcal{O}(n^4)$ algorithm, n being the size of the largest *SLP*.

The way rules (v) and (vi) work in Algorithm 2 is if in the \mathcal{LD} graph one of the rules is satisfied, then a pair of label variables will be found (by the *SLP* algorithm) and equated (through the use of rule (0)). This will cause the set of label variables, \mathcal{V} , to be reduced and thus the number of label variables in the system S to be reduced. The algorithm then returns to step 1 and rebuilds a new \mathcal{LD} graph from the newly modified system.

6.6. Correctness. We now examine the correctness of the above algorithm. Rather than reconstructing all the proofs from “scratch” we can reuse some result proven by Tidén and Arnborg since we are working on a compressed version of the original algorithm.

Lemma 6.13. *The non-failure rules of Algorithm 2 maintain the set of unifiers.*

Proof. The new rules are essentially equivalent to the original set of rules, only modified to work on a compressed version of the problem. This can be seen by considering a path

$$X \xrightarrow{\pi} Z$$

and remove the compression. The path is a graphical representation of the equation.

$$X \stackrel{?}{=} \pi \times Z$$

where the string generated by π is of the form $X_1 \cdot X_2 \cdot \dots \cdot X_n$, for some label variables X_1, \dots, X_n . This is a compressed form of the following equation.

$$X \stackrel{?}{=} X_1 \times X_2 \times \dots \times X_n \times Z$$

This was constructed from n equations in standard form using variable replacement. These equations are of the form

$$X \stackrel{?}{=} X_1 \times K_1, K_1 \stackrel{?}{=} X_2 \times K_2, \dots, K_{n-1} \stackrel{?}{=} X_n \times Z$$

Therefore, the result follows from Theorem 4.1 which is proven in [22]. Let us now examine the rules.

- Rule (x) and rule (0) follow from variable replacement.
- Rule (i) follows from Theorem 4.1 part 1.

- Now consider rules (ii) and (iii), since $\pi = \eta$, these rules follow from $|\pi|$ applications of Theorem 4.1 part 1. The same holds for rule (iv) except that η is a prefix.
- Rules (v) and (vi) also follow from Theorem 4.1 part 1 because by part 1 all the label variables in π would have to be equated to the corresponding variables in η , so we are safe in selecting one pair. More specifically, consider two paths and (v).

$$X \xrightarrow{\pi} Z, X \xrightarrow{\eta} Z$$

correspond to two equations, uncompressed

$$X =^? X_{i_1} \times X_{i_2} \times \dots \times X_{i_n} \times Z, X =^? X_{j_1} \times X_{j_2} \times \dots \times X_{j_n} \times Z$$

Both could be expanded out into standard form and by applying Theorem 4.1 part 1 we equate each pair $X_{i_i} = X_{j_i}$.

- Finally, rules (viii) and (vii) follow from Theorem 4.1 part 2. \square

Note that rule (ix) is a failure condition that is handled by cycle checking the graph.

Lemma 6.14. *If Algorithm 2 exits with success on a system S in standard form, then S is unifiable.*

Proof. The result follows from Lemma 6.13 and the fact that the set of inference rules transforms S into a dag-solved form, which implies unifiability [7]. This can be seen by examining the set of rules and the definition of dag solved-form.

Part (a) is satisfied because if there existed some X_i such that $X_i =^? t_1$ and $X_i =^? t_2$ (for terms t_1 and t_2) one of the inference rules would be applicable.

Part (b) is satisfied because there are no cycles in the graph and thus the equations can be arranged in the proper order. \square

Lemma 6.15. *If Algorithm 2 terminates with failure on a system S in standard form, then S is not unifiable.*

Proof. Follows from Lemma 6.17. \square

From these results we get the following.

Theorem 6.16. *The decision Algorithm 2 is correct.*

Proof. Follows from Lemma 6.14 and Lemma 6.15. \square

6.7. Failure Conditions. Graph cycle checking is employed to detect failure conditions. We argue in this section that if a cycle is found this corresponds to a non-unifiable system.

Lemma 6.17. *A system S in standard form is not unifiable if there exists a cycle in any of the corresponding \mathcal{LP} or \mathcal{LD} graphs for that system.*

Proof. We consider the following cases:

Case 1: Assume that the \mathcal{LD} graph for a system S contains a cycle. Then the cycle was created by zero or more applications of the inference rules and implies that a variable is a proper subterm of itself. By Theorem 4.4 these cycles correspond to non-unifiable systems.

Case 2: Assume the the \mathcal{LP} graph for a system S contains a cycle. This implies there is a cycle between the \sim_r equivalence classes. It is shown in [22] that cycles between \sim_r equivalence classes of this form correspond to non-unifiable systems due to the need for an infinite unifier (see Theorem 4.5). \square

Therefore, if cycles are found we can safely conclude that the system is not unifiable and return an error.

Finally, one could ask if some infinite systems (of Lemma 6.6) that are found in the algorithm of [22] could be missed by the current algorithm due to not creating the same number of new variables. This is shown not to be the case in the following lemma.

Lemma 6.18. *Cycles in the sum propagation graph of [22] for a system S in standard form imply cycles in the \mathcal{LP} graph for S .*

Proof. Clearly if the cycle exists in the initial system or is created by one or more applications of the cancellation rules (a)–(c) then the same cycle will be created in the \mathcal{LP} graph. Thus, assume the cycle is created by creating new equations by rule (d). That is, by rule (d) the following equations are created

$$X =^? X_1 + X_2, \quad X =^? X_3 \times X_4$$

where X_1, \dots, X_4 are newly created variables. But, then we need to equate X_4 and X_2 . Equating variables can only happen through rules (b) and (c) and would require two pre-existing equations of the form $X =^? L_1 + L_2$, $X =^? L_3 \times L_2$ but this is already a cycle in the \mathcal{LP} graph. \square

6.8. Complexity. We establish the polynomial time bound in this section.

Lemma 6.19. *The number of \sim_r equivalence classes never increases.*

Proof. Rule (viii) is the only rule that creates new variables but these variables are contained in pre-existing equivalence classes. \square

Lemma 6.20. *The number of sinks in any equivalence class after processing is at most one. Besides, every non-sink node in the class has exactly one outgoing edge.*

Proof. If there is no sink in the class, then this implies a cycle and thus a non-unifiable system. Therefore, let us assume there is no cycle and thus at least one sink. In addition, there must be at least one source node. It can be seen that rules (ii), (iii), (iv), (v) and (vi) ensure that all the nodes in the class have at most a single outgoing edge. \square

We now prove several small results about new variables and new lateral edges that will be useful in the complexity result.

Lemma 6.21. *The maximum number of new variables added to the system S is equal to twice the number of equivalence classes.*

Proof. Rule (viii) is the only rule that can add variables and this rule can only add two variables for each sink. By Lemma 6.20 there is a single sink for each class. By Lemma 6.19 the number of equivalence classes never increases. Rule (viii) adds two new variables to a lower class for each sink in the upper class. A class is only processed one time, thus the number of variables that can be added is double the number of equivalence classes. \square

Lemma 6.22. *Let $[X]_r$ be a \sim_r -equivalence class. Assume there exist K \sim_r -equivalence classes one level above $[X]_r$ by the \succ_r ordering. Denote the K classes as C_1, C_2, \dots, C_K and assume that each class C_i contains n_i variables, such that $N_K = \sum_{i=1}^K n_i$. Then the total number of lateral edges added to $[X]_r$ by the K higher classes is $\leq 2N_K$.*

Proof. Processing each C_i will produce $n_i - 1$ edges in C_i each connecting one node to the sink of that class. If each of these edges is propagated down by rule (vii) or rule (viii) each class could add a total of $2(n_i - 1)$ edges to $[X]_r$. Doing the sum we get that the K higher classes could add no more than $2N_K - 2K$ edges to $[X]_r$. \square

Lemma 6.23. *The maximum number of lateral edges added to any \sim_r -equivalence class of a system S in standard form is $\mathcal{O}(V_0 + M)$, where V_0 is the initial number of variables in S and M is the number of equivalence classes.*

Proof. Follows from Lemma 6.22. \square

These last two lemmas give a bound on the number of edges added to a class from an outside class, using rules (vii) and (viii). We now need to consider the edges added to a class during the processing of the class itself.

Lemma 6.24. *Let $[X]_r$ be a \sim_r -equivalence class. The number of lateral edges in $[X]_r$ does not increase during application of step (2) or step (4).*

Proof. This follows from the set of inference rules. Rules (vii) and (viii) only create edges at a lower equivalence class. The only rule creating a new edge inside the class is rule (iv). But rule (iv) also deletes an edge, thus leaving the number of lateral edges unchanged. \square

Lemma 6.25. *The number of inference rule applications used during a single application of step (2) of Algorithm 2 is bounded $\mathcal{O}(N + E)$, where N is the number of variables/nodes and E the number of edges in the \mathcal{LD} -graph at the start of step (2).*

Proof. Clearly rules (i) - (iii) are linearly bounded by the number of edges and (0) by the number of variables. \square

Lemma 6.26. *The number of inference rule applications used to process a single equivalence class, C_i , (step (4) of Algorithm 2) is bounded by $\mathcal{O}(N_i * L_i)$ where L_i is the number of lateral edges and N_i the number nodes in the class C_i being processed.*

Proof. Rules (v) and (vi) will equate label variables therefore by Lemma 6.8 the number of times they can be applied is equal to the number of label variables. Rule (viii) can be applied at most once for each class. Rule (vii) can be applied once for each variable in the class. Rule (x) is applied by starting with the sink node of the path and working back to the start node of the path, thus if there are l edges in the equivalence class at the start of the application of rule (x) it will be applied at most l times. In addition because at the start of the application of rule (x) every node, but the sink, has at most one outgoing lateral edge the number of application of rule (x) is also bound by $n - 1$, where n is the number of nodes in the class.

Let us now consider rule (iv). Let l_i be the number of edges of the equivalence class C_i to be processed, including edges added from higher classes, and let n_i be the number of nodes in C_i . Let us also denote a node for which rule (iv) is applicable as a (iv)-peak. That is a (iv)-peak is a node, X , with two edges leaving X such that the inference rule (iv) is satisfied. Note that a single node can form more than one (iv)-peak. Now, if we apply

rule (iv) to each node X forming a (iv)-peak, based on the lateral edge partial ordering (by \succ_{r^*}), until X is no longer a peak we have removed X from the set of nodes forming (iv)-peaks. The number of times we can apply rule (iv) to each node is bounded by the number of edges leaving that node. It can be seen that each application of rule (iv) removes a (iv)-peak but could add a new (iv)-peak. But, the new (iv)-peak will be lower in the \succ_{r^*} path from the initial (iv)-peak node to the sink. As each path must end in a sink the number of these new peaks is naturally bounded by the length of the path. In addition, because rule (iv) both removes and adds an edge it can not increase the number of peaks. Therefore, we can make the following worst case assumption. Assume that rule (iv) can be applied l_i times to each node. Then, as rule (iv) will remove one node from the set of peaks after l_i applications the total number of applications of rule (iv) in C_i is $\leq n_i * l_i$. \square

We have bounds on the number of classes, the number of new edges and nodes, and the number of applications of the inference rules. Finally, we need to bound the size of the *SLPs*. Recall Definition 3.1 for the size of a *SLP*.

Lemma 6.27. *The largest, in size, SLP constructed by Algorithm 2 on any unification problem S is $\mathcal{O}(|S|^4)$ where $|S|$ is the initial number of equations.*

Proof. Assume that we have M topologically sorted, by \succ_r, \sim_r -equivalence classes, C_1, C_2, \dots, C_M , each containing l_1, l_2, \dots, l_M lateral edges, for a total of L , and n_1, n_2, \dots, n_M , for a total of N , variables. In addition, let l'_i denote the number of lateral edges in class C_i at the start of processing and n'_i the number of variables at the start of processing. n'_i and l'_i may differ from n_i and l_i because nodes and edges can be added when processing classes above C_i .

We need to consider both rule (x) and rule (iv) as these are the rules that can add new grammar productions and create larger *SLPs*. Recall two facts about these two rules, given two *SLPs* $I = (\Sigma, N_I, P_I)$ and $J = (\Sigma, N_J, P_J)$.

- (1) For rule (x), creating the new *SLP* K , by Lemma 6.11 $|K| = |P_I \cup P_J| + 1$ and $depth(K) \leq \max\{depth(I), depth(J)\} + 1$.
- (2) For rule (iv), creating the new *SLP* K , by Lemma 6.12 $|K| \leq |I| + depth(I)$ and $depth(K) \leq depth(I)$.

For the analysis we assume that for rule (x) $depth(K) = \max\{depth(I), depth(J)\} + 1$ and for rule (iv) $|K| = |I| + depth(I)$. Therefore, rule (x) adds just one new grammar production and rule (iv) adds $depth(I)$ new grammar productions. We want to give a bound on the grammar productions created at each level in the sort of classes and thus the largest *SLP* produced will be bounded by the total number of *unique* productions.

Compute the Maximum Depth: First note that the *depth* of any *SLP* is only increased by rule (x) and only by 1. Let us first examine the *depth* of a *SLP* in a class C_i . Let D_i be the *depth* of the largest, in *depth*, *SLP* in C_i at the start of processing. Then, since the application of rule (x) is bound by $n - 1$, where n is the number of nodes in the class, the largest, in *depth*, *SLP* produced in C_i by rule (x) is

$$(n'_i - 1) + D_i \tag{6.1}$$

where if C_i is a source node in the \succ_r ordering, $D_i = 1$. Now assume there are k classes, denoted as C_{j_1}, \dots, C_{j_k} , above C_i in the \succ_r ordering. Thus, from the $i - 1$ classes above C_i in the sort, at the start of processing C_i , k of them are related to C_i by \succ_r . As the classes not related to C_i by \succ_r will not contribute any nodes or edges to C_i we need only consider

the k classes.

Claim 1:

$$D_i \leq \sum_{x=1}^k (n'_{j_x} - 1) + 1$$

Proof of Claim 1:

First, there must exist at least one “source” class in the k classes. By the bound given in (6.1), the more source classes we have the less *depth* we add as for each source class, C_s , $D_s = 1$. Thus, for the worst case analysis let us assume there is only one source class, say C_{j_1} , from the k classes. Second, for the worst case analysis when processing any one of the k class we want to ensure we are always adding *depth* to the previous largest, in *depth*, *SLP*. Thus, assume that the k classes form a chain, each class adding the maximum number of productions to the largest, in depth, *SLP* passed down from the class above and then passing that new *SLP* to the next class. The process starts with class C_{j_1} and ends at C_i , i.e., like a total ordering. If we compute the depth of the final deepest *SLP* in this chain, using (6.1) as a bound of the depth at each level, we obtain the following bound on maximum depth.

$$\sum_{x=1}^k (n'_{j_x} - 1) + 1 \tag{6.2}$$

□

Compute the Maximum Size: Now let us consider rule (iv) on the same class C_i . Lemma 6.26 bounds the number of applications of rule (iv) for any class based on l'_i and n'_i . From Lemma 6.22, we can make the worst case assumption, $l'_i = l_i + 2 \sum_{x=1}^k (n_{j_x} - 1)$. This results in a larger then worst-case bound for the number of applications of rule (iv) on the equivalence class C_i :

$$n'_i (l_i + 2 \sum_{x=1}^k (n_{j_x} - 1)) \tag{6.3}$$

By Lemma 6.12 rule (iv) can add up to $depth(\pi)$ new grammar productions when applied to a *SLP* π . We can make a worst-case assumption that each time rule (iv) is applied the *SLP* it is applied to has the maximum depth. Therefore, combining the bound (6.3) with (6.2) we get the following bound on the number of new grammar productions rule (iv) can add during processing of a class C_i :

$$n'_i (l_i + 2 \sum_{x=1}^k (n_{j_x} - 1)) (\sum_{x=1}^k (n'_{j_x} - 1) + 1) \tag{6.4}$$

We also have from (6.1) that the number of new grammar productions produced by rule (x) during processing of class C_i is bounded by

$$n'_i - 1 \tag{6.5}$$

Let us make an additional worst-case assumption, that each edge in each initial class contains a unique single production *SLP*. This of course cannot happen as the initial number of unique *SLPs* before processing for all classes combined is the number of label variables. Combining this assumption about the unique starting edges with the number of applications of (x) and the number of new grammar productions created by (iv) we get the following

bound on the number of possible new grammar productions added by the processing of a class C_i :

$$n'_i - 1 + n'_i(l_i + 2 \sum_{x=1}^k (n_{j_x} - 1)) (\sum_{x=1}^k (n'_{j_x} - 1) + 1) \quad (6.6)$$

Therefore, to get the total number we add up this value from each class from 1 to M . Lemma 6.21 implies that the total number of new variables added to the system is $\leq 2M$, thus we can assume that $\sum_{x=1}^k (n_{j_x} - 1) \leq (N + 2M)$ and $\sum_{x=1}^k (n'_{j_x} - 1) + 1 \leq (N + 2M)$. Recall that N is the total number of initial variables. With these assumptions for any class C_i , $1 \leq i \leq M$:

$$\begin{aligned} & n'_i - 1 + n'_i(l_i + 2 \sum_{x=1}^k (n_{j_x} - 1)) (\sum_{x=1}^k (n'_{j_x} - 1) + 1) \\ & \leq (N + 2M) + (N + 2M)(L + 2(N + 2M))(N + 2M) \end{aligned}$$

Therefore, adding M of these we get:

$$M [(N + 2M) + (N + 2M)(L + 2(N + 2M))(N + 2M)]$$

Since the equations are in standard form there are at most 3 variables per equation. This implies that N , L and M are $\leq 3|S|$, where $|S|$ is the total number of equations. Therefore, we get the upper bound $\mathcal{O}(|S|^4)$. \square

Definition 6.28. Let P_{slp} denote the largest polynomial which bounds the run-time for any of the required *SLP* operations. This polynomial is in terms of the largest *SLP*, which by Theorem 6.27 is $\mathcal{O}(|S|^4)$.

From [13] we could assume that $P_{slp} = \mathcal{O}(n^3)$, where n is the size of the largest *SLP*.

Theorem 6.29. *The worst-case running time of Algorithm 2 is $\mathcal{O}(|S|^4 * P_{slp}(|S|^4))$, where $|S|$ is the initial number of equations in standard.*

Proof. First let V denote the initial number of label variables and let M denote the initial number of equivalence classes. Lemma 6.8 shows that V does not increase and Lemma 6.19 shows that M does not increase.

First we consider a general overview of the run time of the algorithm.

- (1) Since V does not increase and each time the algorithm returns to step (1) it equates two label variables, thus decreasing V , step (1) is applied at most V times.
- (2) Since the algorithm process an equivalence class once and M does not increase the algorithm applies steps (2) though (5) a maximum of M times.

Now let us assume that each step (1) through (5), has an associated polynomial, P_i , $i \in \{1, 2, 3, 4, 5\}$, which bounds the maximum run-time of that step in terms of $|S|$ the initial number of equations in standard form. Based on the general observations above we get the following polynomial, \mathcal{P} , which bounds the running time of Algorithm 2

$$\mathcal{P} = V(P_1 + M(P_2 + P_3 + P_4)) \quad (6.7)$$

It remains to be shown that each P_i is indeed a polynomial in $|S|$ which bounds the run-time of step i . Before examining each step in more detail we present a few useful facts.

- First, rule (0) is the only rule that affects the initial set of equations and this rule only equates two variables. Therefore, we can bound the run-time by $|S|$ without concern that $|S|$ will increase.

- Let V_0 denote the set of all variables in the initial standard form set of equations. Then, by the structure of the equations we can see that there are at most 3 variables per equation and $V_0 \leq 3 * |S|$. This also implies that $V \leq 3 * |S|$ and $M \leq 3 * |S|$.
- It also easily follows that L the number of lateral edges is bounded by $|S|$, in fact $L \leq |S|$.

We now consider each step in Algorithm 2.

- (1) Step (1), by standard graph construction methods, is bounded by $\mathcal{O}(V_0 * |S|)$, which results in $P_1 \leq C_1 * |S|^2$, for some constant C_1 .
- (2) By Lemma 6.25 the number of inference rules applied at step (2) is $\mathcal{O}(N + E)$, where N is the number of nodes and E the number of edges in the \mathcal{LD} graph. Let V_0 denote the initial number of variables in S , then $N \leq V_0 + 2 * M$, since by Lemma 6.21 the maximum number of variables that can be added is twice the number of equivalence classes. Next, by Lemma 6.23 The maximum number of lateral edges added to any equivalence class is $\mathcal{O}(V_0 + M)$. We can thus conservatively say that the maximal number of lateral edges is $\mathcal{O}(M(V_0 + M))$. Since the number of downward edges does not change we get the bound of $C_2(V_0 + 2 * M) + M(V_0 + M)$. Rewriting in terms of $|S|$, $P_2 \leq C_2 * (|S|^2 + |S|^2 + |S|^2) * P_{slp}(|S|^4)$, for some constant C_2 .
- (3) Using standard graph cycle checking we get that $P_3 \leq C_3 * |S|$, for some constant C_3 .
- (4) By Lemma 6.26, the number of rules applied for class i is $\mathcal{O}(N_i * L_i)$ where L_i is the number of lateral edges and N_i the number nodes in the class i being processed. Thus for each class we get a run-time bound of $\mathcal{O}((N_i * L_i) * P_{slp}(|S|^4))$. Rewriting in terms of $|S|$ we get $P_4 \leq C_4 * |S|^2 P_{slp}(|S|^4)$, for some constant C_4 .

Now plugging all these into Equation (6.7), letting $\mathcal{C} = \text{Max}(C_1, C_2, C_3, C_4)$ and replacing V and M in terms of S we get

$$\mathcal{P} \leq \mathcal{C} * (|S|^3 + |S|^4 * P_{slp}(|S|^4) + |S|^3 + |S|^4 * P_{slp}(|S|^4)) \quad (6.8)$$

or

$$\mathcal{O}(|S|^4 * P_{slp}(|S|^4)) \quad (6.9)$$

□

7. ON ASYMMETRIC UNIFICATION AND ONE-SIDED DISTRIBUTIVITY

Our work on a polynomial bounded procedure was partially motivated by its potential application to cryptographic protocol analysis. Since our initial results [15], a new unification paradigm has been developed in [5] and is based on newly identified requirements arising from the symbolic analysis of cryptographic protocols. The analysis involves the unification-based exploration of a space in which the states obey equational theories that can be expressed as a decomposition $R \uplus E$, where R is a set of rewrite rules that is confluent, terminating and coherent modulo E . In order to apply state space reduction techniques, it is usually necessary for at least part of this state to be in normal form, and to remain in normal form even after unification is performed. This requirement can be expressed as an *asymmetric* unification problem $\{s_1 =^\downarrow t_1, \dots, s_n =^\downarrow t_n\}$ where the $=^\downarrow$ denotes a unification problem with the restriction that any unifier leaves the right-hand side of each equation irreducible.

Let us review a few definitions needed for asymmetric unification problems. A *rewrite rule* is an ordered pair $l \rightarrow r$ such that $l, r \in T(\Sigma, \mathcal{X})$ and $l \notin \mathcal{X}$. The rewrite relation on $T(\Sigma, \mathcal{X})$, written $t \rightarrow_R s$, holds between t and s iff there exists a non-variable $p \in \text{Pos}_\Sigma(t)$,

$l \rightarrow r \in R$ and a substitution δ , such that $t|_p = l\delta$ and $s = t[r\delta]_p$. The relation $\rightarrow_{R/E}$ on $T(\Sigma, \mathcal{X})$ is $=_E \circ \rightarrow_R \circ =_E$. The relation $\rightarrow_{R,E}$ on $T(\Sigma, \mathcal{X})$ is defined as: $t \rightarrow_{R,E} t'$ if there exists a position $p \in Pos_\Sigma(t)$, a rule $l \rightarrow r \in R$ and a substitution δ such that $t|_p =_E l\delta$ and $t' = t[r\delta]_p$. The transitive (resp. transitive and reflexive) closure of $\rightarrow_{R,E}$ is denoted by $\rightarrow_{R,E}^+$ (resp. $\rightarrow_{R,E}^*$). A term t is $\rightarrow_{R,E}$ irreducible if there is no term t' such that $t \rightarrow_{R,E} t'$. t is then said to be a R, E -normal form (or just normal form). If $\rightarrow_{R,E}$ is confluent and terminating we denote the irreducible version of a term t by $t \downarrow_{R,E}$.

Definition 7.1. We call (Σ, E, R) a *decomposition* of an equational theory Δ over a signature Σ if $\Delta = R \uplus E$ and R and E satisfy the following conditions:

- (1) E is variable preserving, i.e., for each $s = t$ in E we have $Var(s) = Var(t)$.
- (2) E has a finitary and complete unification algorithm.
- (3) For each $l \rightarrow r \in R$ we have $Var(r) \subseteq Var(l)$.
- (4) R is confluent and terminating modulo E , i.e., the relation $\rightarrow_{R/E}$ is confluent and terminating.
- (5) R is coherent modulo E , i.e., $\forall t_1, t_2, t_3$ if $t_1 \rightarrow_{R,E} t_2$ and $t_1 =_E t_3$ then $\exists t_4, t_5$ such that $t_2 \rightarrow_{R,E}^* t_4$, $t_3 \rightarrow_{R,E}^+ t_5$, and $t_4 =_E t_5$.

Definition 7.2. (Asymmetric Unification). Given a decomposition (Σ, E, R) of an equational theory, a substitution δ is an *asymmetric R, E -unifier* of a set \mathcal{S} of asymmetric equations $\{s_1 =^\downarrow t_1, \dots, s_n =^\downarrow t_n\}$ iff for each asymmetric equations $s_i =^\downarrow t_i$, δ is an $(E \cup R)$ -unifier of the equation $s_i =^? t_i$ and $(t_i \downarrow_{R,E})\delta$ is in R, E -normal form. A set of substitutions Ω is a *complete set of asymmetric R, E -unifiers* of \mathcal{S} (denoted $CSAU(\mathcal{S})$) iff: (i) every member of Ω is an asymmetric R, E -unifier of \mathcal{S} , and (ii) for every asymmetric R, E -unifier θ of \mathcal{S} there exists a $\delta \in \Omega$ such that $\delta \leq_E^{Var(\mathcal{S})} \theta$.

Example 7.3. Let $R = \{X \oplus 0 \rightarrow X, X \oplus X \rightarrow 0, X \oplus X \oplus Y \rightarrow Y\}$ and E be the associativity and commutativity (AC) axioms for \oplus . Consider the equation $Y \oplus X =^\downarrow X \oplus a$. The substitution $\delta_1 = \{Y \mapsto a\}$ is an asymmetric solution since the right hand side will remain irreducible after applying δ_1 . But, $\delta_2 = \{Y \mapsto a, X \mapsto 0\}$ is not an asymmetric unifier, although it is a unifier, since $0 \oplus a \rightarrow_{R,E} a$.

We consider the one-sided distributivity theory in this new asymmetric setting. First, we need to present the axioms as a theory decomposition. In this case the theory decomposition is simple. Let $\Delta = R \cup E$, where $R = \{X \times (Y + Z) \rightarrow X \times Y + X \times Z\}$ and $E = \emptyset$.

One way of approaching the asymmetric unification problem is to start with the symmetric unifiers and then try modifying them, if need be, into asymmetric unifiers. Thus we could have first obtained the symmetric unifier using the original Tidén-Arnberg algorithm. This method looks feasible as far as decidability is concerned, but instead we develop an algorithm where failures can be detected much earlier.

In what follows we are going to assume that variables are always mapped to R, \emptyset -normal forms. We can do this by assuming, without loss of generality, that all substitutions are R, \emptyset -normalized.

Based on Δ , the following inference rules represent an asymmetric algorithm and are a simple modification of the original Tidén-Arnberg algorithm to the new asymmetric domain. The soundness of the rules follow directly from the rules presented in Section 3. In addition, since the asymmetric restriction does not affect the system being subterm collapse free, the error conditions of the original algorithm, and the graphs used to detect them, remain

unchanged. The only additional error conditions, rules (e) and (f), follow due to the rewrite rule, $R = \{X \times (Y + Z) \rightarrow X \times Y + X \times Z\}$, which requires that we apply a reduction to the \times rooted term. Likewise, rules (e') and (f') would imply failure because a reduction could be applied to a term with an irreducible restriction. We denote the algorithm, consisting of the following inference rules along with the error checking, as *Algorithm 3*.

(a)	$\frac{\mathcal{EQ} \uplus \{U =\downarrow V\}}{\{U =\downarrow V\} \cup \{U \mapsto V\}(\mathcal{EQ})}$	if U occurs in \mathcal{EQ}
(b)	$\frac{\mathcal{EQ} \uplus \{U =\downarrow V \circ W, U =\downarrow X \circ Y\}}{\mathcal{EQ} \cup \{U =\downarrow V \circ W, X =\downarrow V, Y =\downarrow W\}}$	where \circ is either $+$ or \times
(c)	$\frac{\mathcal{EQ} \uplus \{U =\downarrow V \circ W, X \circ Y =\downarrow U\}}{\mathcal{EQ} \cup \{U =\downarrow V \circ W, X =\downarrow V, Y =\downarrow W\}}$	
(d)	$\frac{\mathcal{EQ} \uplus \{V \circ W =\downarrow U, X \circ Y =\downarrow U\}}{\mathcal{EQ} \cup \{V \circ W =\downarrow U, X =\downarrow V, Y =\downarrow W\}}$	
(e)	$\frac{\mathcal{EQ} \uplus \{U =\downarrow V \times W, U =\downarrow X + Y\}}{FAIL}$	
(e')	$\frac{\mathcal{EQ} \uplus \{U =\downarrow V \times W, W =\downarrow X + Y\}}{FAIL}$	
(f)	$\frac{\mathcal{EQ} \uplus \{U =\downarrow V \times W, X + Y =\downarrow U\}}{FAIL}$	
(f')	$\frac{\mathcal{EQ} \uplus \{U =\downarrow V \times W, X + Y =\downarrow W\}}{FAIL}$	
(g)	$\frac{\mathcal{EQ} \uplus \{V \times W =\downarrow U, U =\downarrow X + Y\}}{\mathcal{EQ} \cup \{V \times W =\downarrow U, W_1 + W_2 =\downarrow W, V \times W_1 =\downarrow X, V \times W_2 =\downarrow Y\}}$	
(h)	$\frac{\mathcal{EQ} \uplus \{V \times W =\downarrow U, X + Y =\downarrow U\}}{\mathcal{EQ} \cup \{V \times W =\downarrow U, W_1 + W_2 =\downarrow W, V \times W_1 =\downarrow X, V \times W_2 =\downarrow Y\}}$	

Figure 10: Asymmetric Inference Rules for *Algorithm 3*.

In addition to error checking remaining the same, the soundness of the above procedure can be shown by showing each rule is sound and this follows since each rule is just an asymmetric instantiation of the sound symmetric rules presented in Section 4. In addition, we can assume termination since the original algorithm is terminating.

In the following let \circ denote either $+$ or \times .

Lemma 7.4. *The set of equations*
 $\{U =\downarrow V \circ W, U =\downarrow X \circ Y\}$
and the set of equations
 $\{U =\downarrow V \circ W, X \circ Y =\downarrow U\}$
have the same asymmetric solutions as the set
 $\{U =\downarrow V \circ W, X =\downarrow V, Y =\downarrow W\}$.

Proof. The fact that the equations have the same unifiers is a result of Theorem 4.1. Next note that the asymmetric restrictions are maintained since the instances of $V \circ W$, V and W must be irreducible. \square

Lemma 7.5. *The set of equations*
 $\{V \circ W =^\downarrow U, X \circ Y =^\downarrow U\}$
has the same asymmetric solutions as the set
 $\{V \circ W =^\downarrow U, X =^\downarrow V, Y =^\downarrow W\}$.

Proof. The fact that the equations have the same unifiers is a result of Theorem 4.1. We can, without loss of generality, assume that all substitutions are R, \emptyset normalized. This implies that variables are always mapped to R, \emptyset -normal forms and we can apply an irreducibility restriction to them without restricting the solution space. This implies the correctness of the last two equations $X =^\downarrow V, Y =^\downarrow W$. \square

Lemma 7.6. *The following sets of equations have no asymmetric R, \emptyset solutions:*
 $\{U =^\downarrow V \times W, U =^\downarrow X + Y\},$
 $\{U =^\downarrow V \times W, X + Y =^\downarrow U\}.$

Proof. This is due to the orientation of R which requires a reduction in the \times -rooted equation in order to move a $+$ to the top. \square

Lemma 7.7. *The set of equations*
 $\{V \times W =^\downarrow U, U =^\downarrow X + Y\}$
and the set of equations
 $\{V \times W =^\downarrow U, X + Y =^\downarrow U\}$
have the same asymmetric solutions as the set
 $\{V \times W =^\downarrow U, W_1 + W_2 =^\downarrow W, V \times W_1 =^\downarrow X, V \times W_2 =^\downarrow Y\}$, *where W_1 and W_2 are fresh variables.*

Proof. The fact that the equations have the same unifiers is a result of Theorem 4.1. In addition, since we can assume that variables are mapped to R, \emptyset -normal forms and all substitutions are normalized we are free to place an irreducibility restriction on a variable without reducing the set of solutions. \square

Recall that in addition to the two failure rules we maintain the two graphs used to detect failure in the original symmetric algorithm.

Lemma 7.8. *The error conditions of Algorithm 3 are correct.*

Proof. This follows from Lemma 7.6 and the fact that adding the asymmetric restriction does not change the fact that the theory is still simple, i.e., cycles imply failure. Thus, the use of the graph based method to detect cycles is still valid. In addition, it has been shown in [22] that systems that cause non-termination are not unifiable and are detectable via the sum propagation graph method. Since adding irreducibility constraints does not increase the types of systems which cause non-termination and the systems are still detectable via the graph method, the use of a propagation graph to detect all non-terminating systems is still correct. \square

Theorem 7.9. *Asymmetric R, \emptyset unification is decidable.*

Proof. Lemma 7.8 shows that if a system is not asymmetrically unifiable it will be detected by one or more of the failure rules or graphs. In addition, Lemma 7.8 shows that non-terminating systems will also be detected implying Algorithm 3 terminates.

Lemma 7.4, Lemma 7.5 and Lemma 7.7 show that Algorithm 3 transforms a system into a solved-form maintaining the set of solutions. This implies that the substitution constructed from the final solved form is an asymmetric solution to the initial problem. \square

Theorem 7.10. *Algorithm 3 produces a complete set of asymmetric unifiers.*

Proof. Consider an asymmetric problem, S , and its solved form, S' produced by Algorithm 3. Let $\delta_{S'}$ denote the substitution obtained from S' in the standard way. Recall that a substitution obtained from a dag solved form is idempotent, i.e., $\delta_{S'} = \delta_{S'}\delta_{S'}$. Let θ be an asymmetric solution to S and let $X \in \text{Var}(S)$.

- (1) If $X \notin \text{Var}(S')$, $X\delta_{S'} = X$ and $X\delta_{S'}\theta = X\theta$.
- (2) If $X \in \text{Var}(S')$, then there are two cases.
 - (a) $X\delta_{S'} \mapsto X$, in which case $X\delta_{S'}\theta = X\theta$.
 - (b) $X\delta_{S'} \mapsto t_i$, for some term t_i . This implies there is an equation in S' of the form $X =_{\downarrow} t_i$. Recall Lemma 7.4, Lemma 7.5 and Lemma 7.7 show that Algorithm 3 transforms a system into a solved-form maintaining the set of solutions. Thus, $X\theta =_{\Delta} t_i\theta$. This implies that $X\delta_{S'}\theta =_{\Delta} t_i\theta =_{\Delta} X\theta$. \square

Therefore, we can obtain an asymmetric unification algorithm by modifying the original symmetric algorithm. This new algorithm has the following beneficial characteristics:

- Much as the original algorithm of [22], this new algorithm is conceptually easy to grasp, and easy to implement.
- Again, like the Tidén and Arnborg algorithm, the new asymmetric algorithm should perform well computationally on most problem instances, since it is unlikely a problem will have the structure needed to force the exponential behavior.

Complexity.

Definition 7.11. For $n \geq 0$, let $\sigma'(n)$ be the set of equations

$$\begin{aligned} X_{1^{i+1}} + X_{1^i 2} &=_{\downarrow} X_{1^i}, \\ Y_{2^{i+1}} + Y_{2^i 1} &=_{\downarrow} Y_{2^i}, \\ T \times X_{1^i 2} &=_{\downarrow} Y_{2^i 1}, \\ T \times Y &=_{\downarrow} X, \\ X_{1^{i+2}} + X_{1^{i+1} 2} &=_{\downarrow} X_{1^{i+1}} \end{aligned}$$

for all $0 \leq i \leq n$, where X_{l^i} denotes i concatenations of $l \in \{1, 2\}$, i.e., $X_{1^3 2} = X_{1112}$.

A simple modification to the $\sigma(n)$ definition (see Definition 7.11) again results in a family of equations, this time asymmetric, which cause exponentially many applications of the inference rules. The new definition, $\sigma'(n)$, simply places the irreducibility restriction on the variables which are already irreducible with respect to the rewrite rule. Since we can assume, without loss of generality, that substitutions are fully reduced via R, \emptyset -rewriting, the irreducibility restrictions will not be violated. Therefore, the action of the new algorithm does not change, in terms of complexity, from the action of the original algorithm on $\sigma(n)$.

An Open Problem: Polynomial-Time Decision Algorithms for Asymmetric Unification modulo One-Sided Distributivity. The polynomial algorithm developed in Section 6 relies on the use of *SLPs* to ensure the polynomial bound. The *SLP* compression method can be used because the critical information, path labels, are maintained by the compression method. In addition, there are polynomial bounded algorithms for answering the required questions regarding *SLP* compressed strings. However, when asymmetric unification is considered we are forced to also keep track of the irreducibility restriction. This information would unfortunately be lost in the current compression method. The current compression scheme used in Algorithm 2 would need to be modified, to maintain the irreducibility constraints, before the algorithm could be applied to the asymmetric case.

Therefore, a polynomial time asymmetric algorithm based on compression is still an open problem. There are a couple of possible approaches:

- (1) Develop a method of encoding the irreducibility restriction into the same *SLPs*. This seems like it may be possible, but it also requires ensuring the *SLP* algorithms used in Section 6.5.2 can be applied, in polynomial time, to these new encodings.
- (2) Use a different compression method. This may also be possible, for example perhaps using the methods developed in [9]. Again, we would need to ensure all the operations used in Section 6.5.2 could be done on the new compression method in polynomial time.

8. CONCLUSIONS

Three problems are solved in this paper:

- (1) We have developed a new polynomial time algorithm which solves the *decision* problem for a non-trivial subcase, based on a typed theory, of unification modulo one-sided distributivity. This subcase happens to be sufficient to express the negative complexity result in [17]. The new algorithm is conceptually easy to understand and more efficient than the new algorithm solving the general problem.
- (2) We developed the first polynomial time algorithm which solves the *decision* problem for unification modulo one-sided distributivity.
- (3) We developed the first algorithm that solves the asymmetric unification problem for unification modulo one-sided distributivity. That is, the algorithm produces the most general asymmetric unifier. Although this new algorithm is not polynomial, it is conceptually easy to grasp and easily implemented. In addition, it should perform well computationally on most problem instances.

Although the focus of this paper is on decision procedures and complexity we can note that all the algorithms presented in the paper compute unifiers. In the asymmetric case a complete set of unifiers can be obtained from the computed solved forms. In the compressed case, a resulting solved form is actually a compressed representation of a unifier.

9. ACKNOWLEDGMENTS

We thank the reviewers for their helpful comments, especially their suggestions for simplifying Algorithm 2.

REFERENCES

- [1] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.
- [2] Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.
- [3] Hans-Jürgen Bürkert, Alexander Herold, and Manfred Schmidt-Schauß. On equational theories, unification, and (un)decidability. *J. Symb. Comput.*, 8(1-2):3–49, July 1989.
- [4] Francisco Claude and Gonzalo Navarro. Self-indexed text compression using straight-line programs. In Rastislav Kráľovič and Damian Niwiński, editors, *Mathematical Foundations of Computer Science 2009*, volume 5734 of *Lecture Notes in Computer Science*, pages 235–246. Springer Berlin Heidelberg, 2009.
- [5] Serdar Erbatur, Santiago Escobar, Deepak Kapur, Zhiqiang Liu, Christopher A. Lynch, Catherine Meadows, José Meseguer, Paliath Narendran, Sonia Santiago, and Ralf Sasse. Asymmetric unification: A new unification paradigm for cryptographic protocol analysis. In MariaPaola Bonacina, editor, *Automated Deduction, CADE-24*, volume 7898 of *Lecture Notes in Computer Science*, pages 231–248. Springer Berlin Heidelberg, 2013.
- [6] Santiago Escobar, Catherine Meadows, and José Meseguer. A rewriting-based inference system for the NRL Protocol Analyser and its meta-logical properties. *Theoretical Computer Science*, 367:162–202, 2006.
- [7] Jean H. Gallier and Wayne Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67:203–260, 1989.
- [8] Adria Gascón, Guillem Godoy, and Manfred Schmidt-Schauß. Unification with singleton tree grammars. In Ralf Treinen, editor, *20th International Conference on Rewriting Techniques and Applications (RTA)*, volume 5595 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2009.
- [9] Adrià Gascón, Guillem Godoy, and Manfred Schmidt-Schauß. Unification and matching on compressed terms. *ACM Transactions on Computational Logic*, 12(4), 2011.
- [10] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 257–321, 1991.
- [11] Jordi Levy, Manfred Schmidt-Schauß, and Mateu Villaret. Monadic second-order unification is NP-complete. In Vincent van Oostrom, editor, *15th International Conference on Rewriting Techniques and Applications (RTA)*, volume 3091 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2004.
- [12] Jordi Levy, Manfred Schmidt-Schauß, and Mateu Villaret. The complexity of monadic second-order unification. *SIAM J. Computation*, 38(3):1113–1140, 2008.
- [13] Yury Lifshits. Processing compressed texts: A tractability border. In Bin Ma and Kaizhong Zhang, editors, *CPM*, volume 4580 of *Lecture Notes in Computer Science*, pages 228–240. Springer, 2007.
- [14] Markus Lohrey. Algorithms on SLP-Compressed Strings: A Survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.
- [15] Andrew M. Marshall and Paliath Narendran. New algorithms for unification modulo one-sided distributivity and its variants. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pages 408–422. Springer Berlin Heidelberg, 2012.
- [16] Masamichi Miyazaki, Ayumi Shinohara, and Masayuki Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In Alberto Apostolico and Jotun Hein, editors, *CPM*, volume 1264 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 1997.
- [17] Paliath Narendran, Andrew M. Marshall, and Bibhu Mahapatra. On the complexity of the Tiden-Arnborg algorithm for unification modulo one-sided distributivity. In *UNIF 24, EPTCS*, volume 42, pages 54–63, 2010. <http://dx.doi.org/10.4204/EPTCS.42.5>.
- [18] Wojciech Plandowski. Testing equivalence of morphisms on context-free languages. In *Second Annual European Symposium on Algorithms (ESA 94)*, pages 460–470. LNCS 855, Springer-Verlag, 1994.
- [19] Wojciech Rytter. Algorithms on compressed strings and arrays. In Jan Pavelka, Gerard Tel, and Miroslav Bartošek, editors, *SOFSEM99: Theory and Practice of Informatics*, volume 1725 of *Lecture Notes in Computer Science*, pages 48–65. Springer Berlin Heidelberg, 1999.
- [20] Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302:211 – 222, 2003.
- [21] Manfred Schmidt-Schauß. Matching of Compressed Patterns with Character-Variables. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12)*, volume 15

of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 272–287, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

- [22] Erik Tidén and Stefan Arnborg. Unification problems with one-sided distributivity. *Journal of Symbolic Computation*, 3(1/2):183–202, 1987.

APPENDIX A: COMPUTING A SUFFIX

For completeness we present in a very simple recursive algorithm for computing the compressed suffix in polynomial time. One could also use the methods developed in [8, 12, 13, 21] to efficiently compute the suffix and prefix. The algorithm only requires the size of the string produced by the prefix and the actual *SLP* containing the prefix and suffix. We assume also that the size of the string produced for each *SLP* is contained in the data structure. Let π_1 denote the large *SLP* containing the suffix and let π_2 denote the prefix. Let $diff = \|\pi_1\| - \|\pi_2\|$, i.e., $diff$ is the size of the string produced by the suffix. The algorithm returns the suffix *SLP*, denoted as π_3 .

Algorithm 3 BuildSuffix

```

(Input:  $\pi_1, diff$ )
Create a SLP pointer:  $temp = \pi_1$ 
while  $\|RightChild(temp)\| \geq diff$  do
   $temp = RightChild(temp)$ 
end while
if  $\|temp\| == d_f$  then
  return  $temp$ 
else
  Create new non-terminal  $\pi_3$ .
   $RightChild(\pi_3) = RightChild(temp)$ 
   $LeftChild(\pi_3) = BuildSuffix(LeftChild(temp), diff - \|RightChild(\pi_3)\|)$ 
  return  $\pi_3$ 
end if

```

Theorem 9.1. *Algorithm 3 runs in $\mathcal{O}(depth(\pi_1))$, $|\pi_3| \leq |\pi_1| + depth(\pi_1)$ and $depth(\pi_3) \leq depth(\pi_1)$.*

Proof. Consider the recursive call in Algorithm 3. The algorithm only uses a single linear recursive call and the recursion is always called on a non-terminal one level lower in π_1 . Therefore, the algorithm is bounded by $depth(\pi_1)$. In addition, a new rule/non-terminal, is created for each recursive call for a maximum of $depth(\pi_1)$ new rules, thus $|\pi_3| \leq |\pi_1| + depth(\pi_1)$. \square