Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2011-006 February 1, 2011

What is Decidable about Strings?

Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

CSAIL

What is Decidable about Strings?

Vijay Ganesh[†], Mia Minnes[*], Armando Solar-Lezama[†] and Martin Rinard[†]

[†]Massachusetts Institute of Technology
{vganesh, asolar, rinard} @csail.mit.edu
[*]University of California, San Diego
minnes@math.ucsd.edu

Abstract. We prove several decidability and undecidability results for the satisfiability/validity problem of formulas over a language of finite-length strings and integers (interpreted as lengths of strings). The atomic formulas over this language are equality over string terms (word equations), linear inequality over length function (length constraints), and membership predicate over regular expressions (r.e.). These decidability questions are important in logic, program analysis and formal verification. Logicians have been attempting to resolve some of these questions for many decades, while practical satisfiability procedures for these formulas are increasingly important in the analysis of string-manipulating programs such as web applications and scripts.
We prove three main theorems. First, we consider Boolean combination of quantifier-free formulas constructed out of word equations and length constraints. We show that if word equations can be converted to a *solved form*, a form relevant in practice, then the satisfiability problem for Boolean combination of word equations and length constraints is decidable. Second, we show that the satisfiability problem for word equations in solved form that are regular, length constraints and r.e. membership predicate is also decidable. Third, we show that the validity problem for the set of sentences written as a $\forall\exists$ quantifier alternation applied to positive word equations is undecidable. A corollary of this undecidability result is that this set is undecidable even with sentences with at most two occurrences of a string variable.

1 Introduction

Algorithms for deciding the satisfiability problem for formulas over finite-length strings (theories of strings) have been studied for a long time by mathematicians and logicians such as Quine [22], Post, Markov and Matiyasevich [17], Makanin [15], and Plandowski [12, 19, 20]; however, many interesting questions about the existence and complexity of such algorithms are still open.

More recently, formulas over strings have become important in the context of automated bugfinding [24], and analysis of database/web applications [9, 27]. These program analysis and bugfinding tools analyze string-manipulating programs and generate string formulas containing equations over string constants and variables, membership queries over regular expressions, and linear inequalities over the length function. Increasingly, formulas thus generated are solved by off-the-shelf satisfiability procedures such as HAMPI [13] or Kaluza [24]. The solutions to these formulas form the output of the program analysis application. In this context, a deeper understanding of the theoretical aspects of the satisfiability problem for theories of strings has become even more relevant, especially theorems that can be useful in practice.

Problem Statement: Is there an algorithm that decides whether a given formula in the language finite-length strings and integers is *satisfiable* or not (Similar question, for validity of sentences). Atomic formulas in this language are equations over string terms (word equations), linear inequalities over length terms, and membership predicate over regular expressions. A string term is constructed out of string constants, string variables and the concatenation function. Length terms are constructed out of integer constants and variables, addition, multiplication by constant, and the length function whose domain is strings and range is non-negative integers (see Figure 1). While much progress has been made on the decidability of the satisfiability problem for word equations [15, 19, 20], many related problems have remained open for decades. For example, the question of whether the satisfiability problem for quantifier-free theory of word equations and length constraints is decidable has remained open. Matiyasevich [18] noted the relevance of this question to a novel resolution of Hilbert's Tenth Problem. In particular, he showed that if the satisfiability problem for the quantifier-free theory of word equations and length constraints is undecidable, then it gives us a new way to prove Matiyasevich's theorem [17, 18]. Similarly, the decidability question for the satisfiability problem for the quantifier-free theory of word equations, regular expressions, and length constraints is also open.

In this paper, we provide the first decidability results that address the above-mentioned open questions under certain minimal and practical conditions. These decidability results are important in the context of program analysis and verification of web applications and scripts [14, 24]. We also prove an undecidability result. More specifically, we prove the following theorems.

Summary of Contributions:

1. We show that if word equations can be converted to a *solved form* [2] then the satisfiability problem for Boolean combination of word equations and length constraints is decidable. This is the first such decidability result that we know of for these formulas. (Section 3)
2. We provide evidence of how this decidability result can be useful in practice. We empirically studied the word equations in the formulas generated by the Kudzu JavaScript bugfinding tool [24], and verified that all word equations in it are either already in solved form or can be algorithmically converted into one. (Section 3)
3. We further show that the satisfiability problem for quantifier-free formulas constructed out of Boolean combination of word equations in *solved form* that are regular (i.e., the solved form is a regular expression), length constraints, and membership predicate over regular expressions is also decidable. This is the first such decidability result for this set of formulas. (Section 4)
4. We show that the validity problem for the set of sentences written as a $\forall\exists$ quantifier alternation applied to positive word equations (i.e., AND-OR combination of word equations without any negation) is undecidable. (Section 5)

2 Preliminaries

Figure 1 gives the syntax of formulas that we consider here.

2.1 Syntax

Variables: We fix a disjoint two-sorted set of variables $var = var_{str} \cup var_{int}$; var_{str} consists of string variables, denoted $X, Y, S, \ldots$ and var_{int} consists of integer variables, denoted $m, n, \ldots$.
Constants: We also fix a two-sorted set of constants $Con = Con_{str} \cup Con_{int}$. Moreover, $Con_{str} \subset \Sigma^*$ for some finite alphabet, Σ, whose elements are denoted $f, g, \ldots$. Elements of Con_{str} will be referred to as *string constants* or *strings*. Elements of Con_{int} are integers. The empty string is represented by ϵ.
Terms: Formally, terms are either string terms or length terms. A string term (t_{str} in Figure 1) is either an element of var_{str}, an element of Con_{str}, or a concatenation of string terms (denoted by the function *concat*

$$
\begin{array}{lll}
F & ::= Atomic \;\mid\; F \wedge F \;\mid\; F \vee F \;\mid\; \neg F & \\
 & \mid \exists x.F(x) \;\mid\; \forall x.F(x) & \\
Atomic & ::= A_{wordeqn} \;\mid\; A_{length} \;\mid\; A_{regexp} & \\
A_{wordeqn} & ::= t_{str} = t_{str} & \\
A_{length} & ::= t_{len} \leq c & \text{where } c \in Con_{int} \\
A_{regexp} & ::= t_{str} \in RE & \text{where RE is a regular expression} \\
t_{str} & ::= a \;\mid\; X \;\mid\; concat(t_{str}, ..., t_{str}) & \text{where } a \in Con_{str} \;\&\; X \in var_{str} \\
t_{len} & ::= m \;\mid\; v \;\mid\; len(t_{str}) \;\mid\; \Sigma_{i=1}^{n} c_i * t_{len}^{i} & \text{where } m, n, c_i \in Con_{int} \;\&\; v \in var_{int}
\end{array}
$$

Fig. 1. The syntax of $\mathcal{L}^1_{e,l,r}$-formulas.

or interchangeably by $\cdot$). A length term (t_{len} in Figure 1) is either an element of var_{int}, or an element of Con_{int}, or the length function applied to a string term, or a constant integer multiple of length term, or a summation of length terms.
Atomic Formulas: Atomic formulas are word equations ($A_{wordeqn}$), length constraints (A_{length}), or membership predicate over regular expressions (A_{regexp}) as given in Figure 1. Regular expressions are defined in the usual way [1] (regular expression may not contain variables).
Formulas: Formulas are defined in the usual way as a Boolean combination of formulas, possibly with quantifiers (see Figure 1). We allow quantifiers over both string and integer variables.
Formula Nomenclature: We now establish notation for the languages over which formulas are built. Define $\mathcal{L}^1_{e,l,r}$ to be the first-order two-sorted language over which the formulas described above (Figure 1) are constructed. This language contains word equations, length constraints, and membership predicate over regular expressions. The superscript 1 in $\mathcal{L}^1_{e,l,r}$ denotes that this language allows quantifiers, and the subscripts l, e, r stand for length, equation and regular expressions respectively. Similarly, let $\mathcal{L}^1_{e,l}$ be the analogous collection of formulas obtained when excluding membership predicate over regular expressions, and let $\mathcal{L}^1_{e}$ be the collection of formulas when the only atomic formulas are word equations (thus excluding both membership predicate over regular expressions and length constraints). Define $\mathcal{L}^0_{e,l,r}$ to be the set of quantifier-free $\mathcal{L}^1_{e,l,r}$ formulas. Similarly, $\mathcal{L}^0_{e,l}$ and $\mathcal{L}^0_{e}$ are the quantifier-free versions of $\mathcal{L}^1_{e,l}$ and $\mathcal{L}^1_{e}$ classes of formulas respectively. We assume that formulas are always written in *prenex normal form*. Intuitively, a variable is *free* in a formula if it is not quantified. For example, variable y appearing in a formula $\forall y \phi(y, x)$ is *bound*, while x is *free*. For a full inductive definition, see [8]. A formula with no free variables is called a *sentence*.

2.2 Definitions

For a word w, $len(w)$ denotes the length of w. For a word equation of the form $t_1 = t_2$, we refer to t_1 as the left hand side (LHS), and t_2 as the right hand side (RHS). For an $\mathcal{L}^1_{e,l,r}$ formula θ, an *assignment* to the variables of θ is a mapping from the string variables (resp. integer variables) of θ to string elements (resp. integer elements) of the domain. If such an assignment exists, we say θ is *true* under the assignment. If θ is true under such an assignment, then the assignment is called a *solution* or a *satisfying assignment* to θ (and θ is said to be *satisfiable*). An $\mathcal{L}^1_{e,l,r}$-formula with no satisfying assignment is called an *unsatisfiable* formula. We say two formulas θ, ϕ are *equisatisfiable* if θ is satisfiable iff ϕ is satisfiable (they may have different number of assignments, and need not even be from the same language).

The *satisfiability problem* for a set S of formulas is the problem of deciding whether any given formula in S is satisfiable or not. We say that the satisfiability problem for a set S of formulas is decidable if there exists an algorithm (or *satisfiability procedure*) that solves its satisfiability problem (i.e., the procedure is sound, complete and terminating). We say a satisfiability procedure P is sound, complete and terminating if P terminates on all inputs and returns satisfiable iff the input formula is satisfiable (In a practical setting, some of these requirements may be relaxed).

Analogous to the definition of the satisfiability problem for formulas, we can define the notion of the *validity problem* (aka decision problem) for a set Q of sentences in a language L. The validity problem for a set Q of sentences is the problem of determining whether the elements of Q are valid or not, i.e., true under all assignments (We refer the reader to [8] for a full definition of validity of sentences). We say a set of L-sentences Q is decidable, if there exists an algorithm (or decision procedure) that can determine whether any element of S is valid or not. Otherwise we say that Q is undecidable. All definitions in this section also apply to other sets of formulas such as $\mathcal{L}^0_{e,l,r}$.

2.3 Representation of Solutions to String Formulas

It will be useful to have compact representations of sets of solutions to string formulas. For this, we use Plandowski's terminology of *unfixed parts* [20]. Namely, fix a set of new variables V disjoint from Σ, Con, and var. For θ an $\mathcal{L}^1_{e,l,r}$ formula, an *assignment with unfixed parts* is a mapping from the free variables of θ to string elements of the domain or V. Such an assignment represents the family of solutions to θ where each element of V is consistently replaced by a string element in the domain.

Another means by which we encapsulate many solutions to a formula θ into a compact form is via *integer parameters*. If i is a non-negative integer, we write u^i to denote the i-fold concatenation of the string u with itself. An *assignment with integer parameters* to the formula θ is a map from the free variables of θ to string elements of the domain, perhaps with integer parameters occurring in the exponents (Notice that integer parameters are merely a shorthand). Combining the above two definitions, we may consider assignments with unfixed parts and integer parameters. These assignments will provide the general framework for representing solution sets to $\mathcal{L}^1_{e,l,r}$ formulas compactly.

2.4 Examples

Below we give some example formulas and their solution sets. (Some of these examples are from existing literature by Plandowski et al. [20].)

Example 1 *Consider the following $\mathcal{L}^0_e$ formula or word equation $X = aYbZa$. The set of all solutions to this equation can be described by the assignment $X \mapsto aybza, Y \mapsto y, Z \mapsto z$, where $V = \{y, z\}$ are the unfixed parts.*

Example 2 *Consider the equation $abX = Xba$ with one variable X. This is a formula in $\mathcal{L}^0_e$. The map $X \mapsto aba$ is a solution. Moreover, the map $X \mapsto (ab)^i a$ with $i \geq 0$ is a set of assignments with integer parameters which exactly describes all possible solutions.*

Example 3 *Consider the $\mathcal{L}^0_{e,l,r}$ formula*

$$abX = Xba \wedge X \in (ab \mid ba)(ab)^* a \wedge len(X) <= 5.$$

The two solutions to this formula are $X = aba$ and $X = ababa$.

3 Decidability Theorem

In this section we demonstrate the existence of an algorithm deciding whether any $\mathcal{L}^0_{e,l}$ formula has a satisfying assignment or not, under a minimal and practical condition.

3.1 Word Equations and Length Constraints

It is interesting to note that word equations by themselves are decidable [20], and so are systems of inequalities over integer variables (since length constraints can be replaced by integer variables to get a quantifier-free Presburger arithmetic [21] formula, and Presburger arithmetic is known to be decidable [21]). In this section we show that if word equations can be converted into *solved form*, the satisfiability problem for quantifier-free formulas over word equations and length constraints (i.e., $\mathcal{L}^0_{e,l}$ formulas) is decidable. Furthermore, we empirically studied the word equations in the formulas generated by the Kudzu JavaScript bugfinding tool [24], and verified that all word equations in it are either already in solved form or can be algorithmically converted into one.

3.2 What is Hard about Deciding Word Equations and Length Constraints?

The crux of the difficulty in establishing an unconditional decidability result is that, in order to decide the satisfiability of a conjunction of word equations and length constraints, it is necessary to check the satisfiability of the conjunction of the given length constraints and the length constraints implied by the word equations. In order to accomplish this we have to show two things: First, the satisfiability problem for the conjunction of implied length constraints and given length constraints together is decidable. We know how to decide linear constraints over the integers [21], but non-linear constraints in general are not decidable [17]. Unfortunately, there is no theorem that limits the length constraints implied by word equations to be linear. Second (and perhaps more importantly), even if the implied length constraints are linear, it is unclear whether the implied length constraints are in some sense equisatisfiable with the word equations. In the absence of such a property, it is not clear how to finitize the search over the solutions to the length constraints that may be consistent with the solutions to the word equations or vice-versa.

3.3 Definition of Solved Form

A word equation w has a *solved form* if there is a finite set $\mathcal{S}$ of formulas (possibly with integer parameters) that is logically equivalent to w and satisfies the following conditions. The idea of solved form is well known in equational reasoning and satisfiability procedures for rich logics (aka SMT solvers) [2].

- Every formula in $\mathcal{S}$ is of the form $X = t$, where X is a variable occurring in w, and t is the results of finitely many concatenations of constants in w (with possible integer parameters) and possible unfixed parts and integer parameters. (Recall the definitions for integer parameters and unfixed parts from Section 2.) All integer parameters i in $\mathcal{S}$ are linear (i.e., of the form $c * i$ where c is an integer constant).
- Every variable in w occurs exactly once on the LHS of an equation in $\mathcal{S}$ and never on the RHS of an equation in $\mathcal{S}$.

The solved form corresponding to w is denoted as the conjunction of all the formulas in $\mathcal{S}$: $\wedge\mathcal{S}$. Note that formulas in $\mathcal{S}$ are not necessarily in the language $\mathcal{L}^0_{e,l}$ of word equations and length constraints. Solved form equations can have integer parameters, whereas $\mathcal{L}^0_{e,l}$ formulas cannot. The solved form is not used as such in the satisfiability procedure discussed below. Instead, the solved form is used to extract all necessary and sufficient length information *implied by w*. Moreover, if there is an algorithm which converts any given word equation to a solved form (if one exists), and if $\wedge\mathcal{S}$ is the output of this algorithm when given w, we say that the *effective solved form* of w is $\wedge\mathcal{S}$.

Example 4 Word Equations Without a Solved Form: *Not all word equations can be written in solved form. Consider the equation*

$$XabY = YbaX$$

It is known that the solutions to this equation cannot be expressed using linear integer parameters [20].

Example 5 Satisfiable Solved Form Example: *Consider the system of word equations*

$$Xa = aY \wedge Ya = Xa.$$

This formula can be converted into solved form as follows:

$$X = a^i \wedge Y = a^i \qquad (i > 0).$$

Example 6 Unsatisfiable Solved Form Example: *Consider the formula*

$$abX = Xba \wedge X = abY \wedge len(X) < 2$$

with variables X, Y. The solution to the equation $abX = Xba$ are described by the map $X \mapsto (ab)^i a$ with $i \geq 0$. Hence the solved form for this equation is:

$$X = (ab)^i a$$

However, the entire formula is unsatisfiable. This example illustrated the need for checking satisfiability of given length constraints, and length constraints implied by the word equations.

3.4 Why Solved Form?

The solved form crucially addresses the issues we discussed regarding the difficulty of proving decidability. If word equations can be converted into a solved form, we are guaranteed that all length information implied by the word equations can be represented in a finite and *complete* (defined below) manner. The completeness property enables a satisfiability procedure to decouple the word equations part from the (implied and given) length constraints, because it implies that the word equation is equisatisfiable with the implied length constraints. Furthermore, solved form guarantees that the implied length constraints are linear inequalities, and hence their satisfiability problem is decidable [21]. This insight forms the basis of our decidability results. It also helps to note that all word equations that we have encountered so far in practice [24], are either in solved form or can be converted into one.

3.5 Proof Idea for Decidability

Without loss of generality, we will only consider conjunction of word equations and length constraints (the result can be easily extended to arbitrary boolean combination of such formulas). Let $\phi \wedge \theta$ be an $\mathcal{L}^0_{e,l}$-formula, where ϕ is a conjunction of word equations and θ is a conjunction of length constraints. Observe that ϕ implies a certain set of length constraints. For example, if ϕ were the equation $X = abY$ then we can conclude the following set $\mathcal{R}$ of length constraints: $len(X) = 2 + len(Y), len(Y) \geq 0$.

In the above example, the set of implied length constraints is finite but exhaustive. To be more precise about what we mean by exhaustive, observe that any other length constraint implied by the equation $X = abY$ is either in $\mathcal{R}$ or is implied by $\mathcal{R}$ (a form of completeness or sufficiency). This fact allows one to easily check the satisfiability of $X = abY$ with any given length constraints. It turns out that the set of implied length constraints for word equations that have a solved form is also finite and exhaustive. We prove this fact below, and leverage it in proving that a sound, complete and terminating satisfiability procedure exists for $\mathcal{L}^0_{e,l}$ formulas with word equations in solved form.

Definitions: We say that a set $\mathcal{R}$ of length constraints is *implied by a word equation* ϕ if the lengths of the strings in any solution of ϕ satisfies all constraints in $\mathcal{R}$. A set $\mathcal{R}$ of length constraints implied by a word equation ϕ is *complete* for ϕ if any length constraint implied by ϕ is either in $\mathcal{R}$ or is implied by a subset of $\mathcal{R}$. These definitions can be suitably extended to a Boolean combination of word equations.

Discussion on Finiteness and Completeness: Finiteness ensures that these length constraints in $\mathcal{R}$ can be processed by a satisfiability procedure in finite time. Completeness guarantees that the satisfiability procedure will have all the length information it needs to correctly decide the satisfiability of the original word equation(s). In other words, completeness implies that the implied length constraints are equisatisfiable with the original word equation(s). This allows us to decide $\phi \wedge \theta$ (ϕ is the word equation, and θ is the given conjunction of length constraints) by independently deciding ϕ and $\mathcal{R} \wedge \theta$, and combining the results appropriately. Finally, we give an algorithm for obtaining finite and complete sets of length constraints for any word equation in solved form. We use this to prove that the resulting satisfiability procedure for the satisfiability of $\mathcal{L}^0_{e,l}$ formulas is sound, complete and terminating.

3.6 Decidability Theorem

We prove a set of lemmas, followed by the actual decidability theorem.

Lemma 1. *If a word equation w has a solved form $\mathcal{S}$, then there exists a set $\mathcal{R}$ of linear length constraints, implied by w, that is finite and complete. Moreover, there is an algorithm which, given w, computes this set $\mathcal{R}$ of constraints.*

Proof. By definition of solved form, a word equation w is logically equivalent to its solved form $\mathcal{S}$ (i.e., every solution to w is a solution to $\mathcal{S}$ and vice-versa). Hence, the set of length constraints implied by w is equivalent to the set of length constraints implied by $\mathcal{S}$. The solved form for w is a set S of equations of the form $X = t_1 t_2 \cdots t_n$, where X is a variable that occurs in w. Moreover, $\mathcal{S}$ contains exactly one such equation for each variable X that occurs in w. Recall that each term t_i in S is either the concatenation of constants or unfixed parts (possibly with integer parameters). The integer parameters are linear; that is, they appear as $c * i$ with c a positive integer in a term such as $(a_1 a_2 ... a_k)^{c*i}$.

For each X appearing in w, the corresponding equation in $\mathcal{S}$ implies that

$$len(X) = len(t_1) + len(t_2) + \cdots + len(t_n),$$

where each $len(t_i)$ is either a constant or a constant multiple of an integer parameter or an unfixed part. If t_i is an unfixed part, we also conclude the constraint $len(t_i) \geq 0$. For each X a single suitable inequality of the form $len(X) \geq c$, for some c that denotes the minimum length of X, is also added to $\mathcal{R}$. Let $\mathcal{R}$ be the union of all these constraints for each variable X in w. Since $\mathcal{S}$ is finite, there are finitely many variables X, and finitely many length constraints per X that are added to $\mathcal{R}$. Hence, by construction $\mathcal{R}$ is finite.

Next, we show that the set $\mathcal{R}$ is complete. By construction of $\mathcal{S}$, every equation in $\mathcal{S}$ can at most imply two length constraints for each variable X (an equality and an inequality as described above), and one inequality per unfixed part (any other implied constraint is redundant). Every equation in solved form has only one variable, and finitely many unfixed parts. All these length constraints are included in $\mathcal{R}$ by construction. Since w is logically equivalent to $\mathcal{S}$, they imply the same set of length constraints. Hence, any length constraint implied by w is in $\mathcal{R}$ or is implied by a subset of $\mathcal{R}$. Hence $\mathcal{R}$ is complete for w. The existence of the algorithm to compute $\mathcal{R}$ given w in solved form follows directly from this proof.

Lemma 2. *If a word equation w has a solved form $\mathcal{S}$, then w is equi-satisfiable with the length constraints $\mathcal{R}$ derived from $\mathcal{S}$.*

Proof. First of all the finiteness of $\mathcal{R}$ allows us to treat it as a formula (conjunction of its elements).

($\Rightarrow$)If w is satisfiable, then so is $\mathcal{R}$: Since w is satisfiable, and w and $\mathcal{S}$ are logically equivalent by definition, it follows that $\mathcal{S}$ is satisfiable. Since, $\mathcal{S}$ is satisfiable and $\mathcal{R}$ is implied by $\mathcal{S}$ (i.e., contains only length constraint implied by $\mathcal{S}$) it follows that $\mathcal{R}$ is satisfiable.

($\Leftarrow$) If $\mathcal{R}$ is satisfiable, then so is w: Any length constraint implied by $\mathcal{S}$ is in $\mathcal{R}$, i.e., $\mathcal{R}$ is complete for $\mathcal{S}$. In other words, by construction of $\mathcal{S}$ and $\mathcal{R}$, for every variable in w there is a finite number of necessary and sufficient length constraints, and they are all in $\mathcal{R}$. It follows that any assignment to $\mathcal{R}$ has a corresponding assignment to $\mathcal{S}$, and hence to w. (The solved form actually gives us something stronger than equisatisfiability. It gives a bijection over the satisfying assignments of w and $\mathcal{R}$.)

Theorem 7 *The satisfiability problem for $\mathcal{L}_{e,l}^0$ formulas is decidable, provided that there is an algorithm to obtain the solved forms of given word equations.*

Proof. It is sufficient to consider $\mathcal{L}_{e,l}^0$ formulas with a single word equation and conjunction of length constraints (The proof is easily extended to Boolean combination of word equations and length constraints). We prove the theorem by constructing a satisfiability procedure, and showing that it is sound, complete, and terminating. Let the input to the algorithm be a formula $\phi \wedge \theta$, where ϕ is the word equation and θ is a conjunction of length constraints. The output of the algorithm is *satisfiable* (SAT) or *unsatisfiable* (UNSAT).

Recall that Plandowski's algorithm [20] takes word equations as input and returns whether they are satisfiable or not. The first step of our satisfiability procedure runs Plandowski's algorithm with input ϕ in parallel with the satisfiability algorithm for θ as a formula in Presburger arithmetic (known to be decidable [21]). If either of these return UNSAT, we return UNSAT.

Next, we use the assumption that the word equation ϕ has an effective solved form and generate both the solved form $\mathcal{S}$ and the associated complete, and finite implied set $\mathcal{R}$ of linear length constraints (as in Lemma 1). Consider the conjunction of θ and $\mathcal{R}$ (finiteness of $\mathcal{R}$ allows us treat it as a formula). This is a system of linear integer equalities and inequalities, and hence can be decided using a satisfiability procedure for Presburger arithmetic. Given the way $\mathcal{R}$ is constructed, we do not need to worry about considering an unbounded number of implied constraints. Hence, the satisfiability procedure only has to do finite amount of work in this case.

What remains to be established is the soundness and completeness. Since $\mathcal{R}$ is complete for $\mathcal{S}$ (and hence for ϕ) it follows that $\phi \wedge \theta$ is equisatisfiable with $\mathcal{R} \wedge \theta$ (by Lemma 2). Thus, the above procedure is a sound, complete and terminating procedure for $\mathcal{L}^0_{e,l}$-formulas whose word equations have effective solved forms.

3.7 Practical Value of Solved Form and the Decidability Result

In their paper [24] on an automatic JavaScript testing program (Kudzu) and a practical satisfiability procedure for strings Saxena et al. mention generating more than 50,000 $\mathcal{L}^0_{e,l,r}$ formulas where the length of the string variables is bounded (i.e., the string variables range over a finite universe of strings). Kudzu takes as input a JavaScript program and (implicit) specification, and does some automatic analysis (a form of concrete and symbolic execution [4,10]) on the input program. The result of the analysis is a string formula that captures the behavior of the program-under-test in terms of the symbolic input to this program. A solution of such a formula is a test input to the program-under-test. Kudzu uses the Kaluza string solver to solve these formulas to generate program inputs.

JavaScript programs often process strings. These strings occur as part of input forms on web-pages or as parts of strings used by JavaScript programs to dynamically generate web-pages or as part of an SQL query being constructed by these programs. During the processing of these strings, JavaScript programs often concatenate these strings to form larger strings, use strings in assignments, use string length for copying strings and comparisons, construct equalities between strings as part of if-conditionals or use regular expressions in order to check the sanity of the strings being processed. Hence, any program analysis of such JavaScript programs results in formulas that contains string constants and variables, concatenation operation, regular expressions, word equations and length function.

We analyzed the string formulas generated by the Kudzu automatic tester [24], and found that all of the word equations in these formulas are either in solved form or can be converted into one. These formulas ranged in size from a few to dozens of equalities, and were derived from 18 real-world JavaScript applications. Our analysis of string formulas generated by Kudzu bolsters our claim that most, if not all, word equations that occur in practice are in solved form. All the test inputs generated by Kudzu are available at the following URL: `http://webblaze.cs.berkeley.edu/2010/kaluza/`.

4 Word Equations, Length, and Regular Expressions

We now consider whether the previous result can be extended to show that the satisfiability problem for $\mathcal{L}^0_{e,l,r}$ formulas is decidable, provided that there is an algorithm to obtain the solved forms of given word equations. A generalization of the proof strategy from above looks promising. That is, given a membership test in a regular set $X \in RE$, we can extract from the structure of the regular expression a constraint on the length of X that is expressible as a linear inequality. To do so, we recall the following fact [3].

Lemma 3. *Given a regular set R, the set of lengths of strings in R is a finite union of arithmetic progressions. Moreover, there is an algorithm to extract the parameters of these arithmetic progressions from the regular expression defining R.*

Using the above Lemma, the set of length constraints implied by an arbitrary regular expression can be expressed as a finite system of linear inequalities. Thus, it seems that the same machinery as in the $\mathcal{L}^0_{e,l}$ theorem may be applied to the broader context of $\mathcal{L}^0_{e,l,r}$. However, there remain some subtleties to resolve. These are best elucidated by example.

Example Consider the $\mathcal{L}^0_{e,l,r}$ formula

$$abX = Xba \;\wedge\; X \in (ab)^*b \;\wedge\; \text{len}(X) \leq 3.$$

A naïve translation of each component into length constraints gives us the following:

$$\begin{cases} \mathrm{len}(X) = 2i + 1, i \geq 0 & \textit{implied by the word eqn \& r.e.} \\ \mathrm{len}(X) \leq 3. \end{cases}$$

This system of length constraints is easily seen to be simultaneously satisfiable: let $i = 0$ or 1 and hence $\mathrm{len}(X) = 1$ or 3. However, the formula is **not** satisfiable since the word equation corresponds to $X = (ab)^*a$ and the regular expression requires any solution to end in a b. Thus, in order to address $\mathcal{L}^0_{e,l,r}$ formulas, we must take into account more information than is encapsulated by the length constraints imposed by regular expressions. In particular, if we impose the additional restriction that the word equations must have solved form (without unfixed parts) that are also regular expressions, then we can get a decidability result for $\mathcal{L}^0_{e,l,r}$ formulas.

Theorem 8 *The satisifiability problem for $\mathcal{L}^0_{e,l,r}$ formulas is decidable, provided that there is an algorithm to obtain the solved forms of the given word equations, and the solved form equations do not contain unfixed parts and are regular expressions.*

Proof. Let

$$\theta(X) \wedge \phi \wedge (X \in RE)$$

be a $\mathcal{L}^0_{e,l,r}$ formula, where $\theta(X)$ is a word equation, ϕ is a conjunction of length constraints and $X \in RE$ is a regular expression membership constraint. The proof can be easily extended to a Boolean combination of atomic formulas. Using the above Lemma 3, it is easy to establish that this satisfiability procedure is sound, complete and terminating. The satisfiability procedure proceeds as follows:

– If any of $\theta(X)$, ϕ or $X \in RE$ is UNSAT, return UNSAT.
– Convert $\theta(X)$ into solved form where it is a regular expression, i.e., $X \in RE_1$ (The theorem assumes that such a form exists for $\theta(X)$). Compute the intersection of the two regular expressions, $X \in RE \cap RE_1$. If $RE \cap RE_1$ is empty, return UNSAT.
– Extract equisatisfiable length constraints ψ from $X \in RE \cap RE_1$ as suggested above using the Lemma 3. if $\psi \wedge \phi$ is UNSAT, return UNSAT. Else return SAT.

5 The Undecidability Theorem

In this section we prove that the validity problem for the set of $\mathcal{L}^1_e$ sentences over positive word equations (AND-OR combinaton of word equations without negation) whose prefix normal form contains $\forall\exists$ quantifier alternation is undecidable.

5.1 Proof Idea

We do a reduction from the halting problem for two-counter machines, which is known to be undecidable [1], to the problem in question. We do this by essentially encoding computation histories as strings. The use of two-counter machine is crucial for the proof to go through, as will become clear later.

The basic proof strategy is as follows: given a two-counter machine M and a finite string w, we construct a $\mathcal{L}^1_e$ sentence $\forall S \exists S_1, ...S_4 \theta(S, S_1, ..., S_4)$ such that M does not halts on w iff $\forall S \exists S_1, ..., S_4 \theta(S, S_1, ..., S_4)$ is valid, i.e., all assignments to the string variable S are not halting computational histories of M over w. The variables $S_1, ..., S_4$ denote substrings of S. A similar strategy is used by Michael Sipser [26] to show that the problem of deciding whether a context-free grammar can generate all strings is undecidable.

5.2 Recalling Two-counter Machines

A *two-counter machine* is a deterministic machine which has a finite-state control, two semi-infinite storage tapes whose tape alphabet contains only two symbols, Z and blank, and a separate read-only semi-infinite input tape [1]. All tapes have a left endpoint, but the storage tapes have no right endpoint. All tapes are composed of cells, each of which may store a symbol from the appropriate alphabet. Each tape has a corresponding tape-head that may move left, right or stay put. The input tape-head cannot move past the right end of the input string. The tape-heads scan the tape cell by cell, may read the symbol stored on the cell being scanned by it or write in that cell (input tape is read-only). The cell being scanned by the input tape-head is called the *current cell*. The initial position of all the tape-heads is the leftmost cell of their respective tapes. The input to the machine is a finite string written on the input tape, starting at the leftmost cell. A special character follows the input string on the tape to mark the end of the input.

The symbol Z serves as a *bottom of stack* marker on the storage tapes. Hence, it appears initially on the cell scanned by the tape head and may never appear on any other cells. The finite control may only write and erase blanks from the storage tapes. An integer i can be represented on the storage tape by moving the tape head i cells to the right of Z. A number stored on the storage tape can be incremented or decremented by moving the tape head right or left. We can test whether a number is zero by checking whether Z is scanned by the head, but equality between two numbers cannot be directly tested. It is well known that the two-counter machine can simulate an arbitrary Turing machine. Consequently, the halting problem for two-counter machines is undecidable [1].

More formally, a two-counter machine M is a tuple $\langle Q, \varDelta, \varGamma, \delta, q_0, F\rangle$ where,

- Q is the finite set of control states of M. $q_0 \in Q$ is the initial control state, and $F \subseteq Q$ is the set of final control states.
- $\varDelta$ is the finite alphabet of the input tape, and $\varGamma = \{Z, B\}$ is the storage tape alphabet, where Z is used as the end marker for the two semi-infinite storage tapes and B is the blank symbol (In the language of strings we use b and c, one for each storage tape).
- δ is the **transition function** for the control of M:

$$\begin{aligned}\delta :&Q \times \varDelta \times \varGamma \times \varGamma \rightarrow \\ &Q \times \{\text{input}, stor1, stor2\} \times \{B \cup \text{erase} \cup \text{NOP}\} \times \{L, R\}\end{aligned}$$

  In words, given a state of the machine and the letter on the current cell of each tape, the transition function specifies the next state of the machine, a tape-head (input or one of the storage tapes stor1 or stor2) to move, specifies the corresponding action on the storage tapes (place a blank or erase a blank or perform a No OPeration), and specifies whether the corresponding tape head moves left (L) or right (R)

5.3 Instantaneous Description of Two-counter Machines as Strings

We define *instantaneous descriptions* (ID) of two-counter machines in terms of strings. Informally, the ID of a machine represents its *entire configuration* at any instant in terms of machine parameters such as the current control state, current input-tape letter being read by the machine, and current storage-tape contents. **Definition of ID:** The instantaneous description (ID) of a two-counter machine M on an input w can be described by a string given by concatenating the following components listed below (Each component of an ID is separated from the others by an appropriate special character. In the discussion that follows we will not refer to the separator explicitly anywhere, and we will assume that it is present in the appropriate positions for every ID. Note that this internal separator is different from the one that separates IDs from each other):

- Current control state of M: represented by a character over the finite alphabet Q.
- The finite-length input string w and the current input tape cell (or numbers between $\{0, \cdots, |w| - 1\}$ encoded as string constants) being scanned by the input tape-head.
- The finite distances of the two storage heads from the symbol Z, represented as a string of blanks (i.e., in unary representation). For convenience, we will use the symbol b to denote the blanks on storage tape 1, and c on storage tape 2.

Definition of Initial ID: The initial instantaneous description (ID) of a two-counter machine M with a given input string w is a concatenation of the following (Note that for any two-counter machine and each input w, there is exactly one initial ID; call it $Init_{M,w}$): Initial state q_0 of M, followed by $w = w_0u$, encoding the contents of the input tape by w_0u, where w_0 is the first letter of the input string w. Followed by number 0 encoded as a string constant (denoted by N_0), indicating that the current cell is the 0^{th} letter of w. Followed by two string constants that are concatenations of suitable number of blanks to denote 0 on each tape (i.e., elements of b^* for storage tape 1 and c^* for storage tape 2).
Definition of Final ID: The final IDs of a two-counter machine are strings that are very similar to the Initial ID. The only difference is that the control states are chosen to be one of the finitely many final control states $q_f \in F$ of M. We use the convention that after reaching a final state, there is a series of dummy moves wherein the storage tape-heads move to the leftmost cell, and the storage tapes contain the unary representation of the number 0. Similarly, we use the convention that the input tape-head has moved to the leftmost position (i.e., 0^{th} letter of the input w) on the input tape. Observe that there are only finitely many Final IDs.

5.4 Computational History of a Two-counter Machine as a String

A *well-formed computational history* of a two-counter machine M for a given input w is the concatenation of a sequence of IDs separated by the special symbol #, where the sequence must start with the initial ID of M on w. Moreover, for each i less than the length of the sequence of IDs, ID_{i+1} is the result of transforming ID_i according to the transition function of M. A well-formed computational history of the machine M on the string w is called *accepting* if it is a finite string whose last ID is a Final ID of M on w. The last ID of a string is defined to be the rightmost substring following a separator #. Otherwise, we call the finite string non well-formed or rejecting.

5.5 Alphabet for String Formulas and The Universe of Strings

Given a two-counter machine M and an input string w, we first define a finite alphabet by doing a cross product over the ID separator #, and the elements of the finite set Q (states of M), and the set of input tape-head positions $0, \cdots, |w| - 1 \in N$, and w to get the alphabet Σ_0:

$$\Sigma_0 \equiv \{\#q_1 0 w, \cdots, \#q_k \cdot |w| - 1 \cdot w\}$$

This alphabet allows to treat the *intial segment* of the IDs of M (the part without the storage tapes) as letters. We also define $\Sigma_1 = b$ and $\Sigma_2 = c$ (These will represent the storage tape contents for tape 1 and 2 resp.). We define the alphabet of strings as $\Sigma \equiv \{\Sigma_0 \cup \Sigma_1 \cup \Sigma_2\}$, and the universe of strings as Σ^*.

5.6 The Undecidability Theorem

Theorem 9 *The validity problem for the set of $\mathcal{L}_e^1$ sentences over positive word equations with $\forall\exists$ quantifier alternation is undecidable.*

Proof. **By Reduction:** We reduce the halting problem for two-counter machines to the decision problem in question. Given a pair $\langle M, w\rangle$ of a two-counter machine M and an arbitrary input w to M, we construct a $\mathcal{L}_e^1$-formula $\theta_{M,w}(S, S_1, ..., S_4, U, V)$ such that

$$\begin{array}{l} M \text{ does not accept and halt on } w \iff \\ \text{all assignments to } S \text{ in } \exists S_1, S_2, S_3, S_4 \theta_{M,w}(S, S_1, \cdots, S_4) \\ \text{are not a computational history of } M \text{ over } w\text{, where } S_i \text{ are substrings of S} \end{array}$$

What Does θ Say?: For brevity, we write θ for $\theta_{M,w}$. θ says S does not start with the Initial ID, or does not end with any of the Final IDs, or is not a well-formed sequence of IDs or does not follow the transition function of M over w.

Structure of θ: First, let NotInit denote the finite set of string constants that has the same (or smaller) length as the Initial ID $Init_{M,w}$, but is not equal to $Init_{M,w}$. Note that there can only be finitely many such strings over the alphabet Σ since the alphabet is finite and $Init_{M,w}$ is a finite string. Similarly define the finite set NotFinal of string constants that are not equal to the Final IDs, but have the same or smaller length. Again note that there can only finitely many such strings. The formula $\forall S \exists S_1, \cdots, S_4 \theta(S, \cdots, S_4)$ is constructed as follows:

$$\begin{aligned} \forall S \exists S_1, S_2, S_3, S_4, U, V((& \bigvee_{E \in \text{NotInit}} S = E \cdot S_1) \vee \\ & (\bigvee_{E \in \text{NotFinal}} S = S_1 \cdot E) \vee \\ & \text{StartsWithInit\_FollowedBy\_BadStorages(S)} \vee \\ & \text{NotWellFormedSequence}(S, S_1, \cdots, S_4) \vee \\ & ((S = S_1 \cdot S_2 \cdot S_3 \cdot S_4) \wedge (Ub = bU) \wedge (Vc = cV) \wedge \neg Next(S, S_1, \cdots, S_4, U, V))) \end{aligned}$$

- **NotInit and NotFinal:** The first two disjuncts state that S either does not start with the initial ID or does not end with any Final IDs. Note that E above is not a variable. It is simply a shorthand for the elements of the finite sets NotInit and NotFinal.
- **StartsWithInit_FollowedBy_BadStorages():** Another possibility of a not well-formed computational history is when S starts with $Init_{M,w}$ but has non-zero b or c on the storage tapes (b and c are string constants). First observe that a arbitrary sequence of b's can be represented by word equation $Ub = bU$ (Similarly, for $Vc = cV$):

$$\begin{aligned} &(S = Init_{M,w} \cdot b \cdot U \cdot V \cdot S_1 \wedge (bU = Ub) \wedge (cV = Vc)) \vee \\ &(S = Init_{M,w} \cdot c \cdot V \cdot S_1 \wedge (cV = Vc)) \end{aligned}$$

- **NotWellFormedSequence()**: Asserts that S contains at least one sequence of letters from the alphabet Σ such that this sequence is not well-formed, i.e., is not a sequence of IDs. First, observe that any well-formed computational history is a sequence of IDs. Hence, any string that is NOT well-formed should be a satisfying assignment to S in $\theta(S)$. This fact is captured by the formula NotwellFormedSequence$(S, S_1, \cdots, S_4)$ below. Observe that a well-formed ID is a regular expression of the form $\Sigma_0 b^* c^*$, and a well-formed sequence of IDs is a string of the form $(\Sigma_0 b^* c^*)^* - \epsilon$ (We do not allow the empty string ϵ to be a well-formed sequence). Hence, the set of all non well-formed strings is given by the regular expression

$$\text{NotWellFormed} \equiv \Sigma^* - (\Sigma_0 b^* c^*)^*$$

  We constructed the deterministic finite automata for this regular expression, and observed that any string defined by NotWellFormed either starts with b or a c, or has the sequence cb in it. Fortunately, these facts can be easily captured using word equations, and we will not need to use regular expressions. The fact that a non well-formed sequence may start with b or c is already captured by the NotInit formula above. The fact that a non well-formed sequence contains cb or is an ϵ is encoded by the following formula NotWellFormedSequence():

$$(S = \epsilon) \vee (S = S_1 \cdot c \cdot b \cdot S_4)$$

- **Encoding Legal Transitions as Next$(\mathbf{S}, \cdots, \mathbf{S_4}, \mathbf{U}, \mathbf{V})$:** We encode a legal transition using the formula $Next()$ defined below, where S_2, S_3 are well-formed IDs. The conjuncts to $\neg Next()$ in $\theta()$ above, assert that S can be partitioned into four substrings, and there exists some sequence of b's captured by the word equation $Ub = bU$ representing storage-1 contents, $Vc = cV$ capture storage-2). Given this, $Next$ is a big disjunction over all possible finitely-many pairs of IDs defined by the transition function:

$$\bigvee_{\text{Transitions}} S_2 = q_2 n_2 w U V \Rightarrow S_3 = q_3 n_3 w U^1 V^1$$

  $Next()$ asserts that the pair of variables S_2, S_3 form a legal transition. Note that $q_2, q_3 \in Q$, $n_2, n_3 \in N$ are all constants determined by the transition function, and w is the input string to the two-counter machine M. To be more precise, the transition function states that given the current control state q_2, the current cell contents at location n_2 in w, current storage tape contents U and V, the next state should be q_3, the new position of the input tape-head should be n_3 and the value of the storage tapes changed from U and V to U^1 and V^1 respectively. Note that U, V are strings ranging over b^*, and U^1 is either U (No Op) or Ub (increment operation) or $U^1 b = U$ (decrement operation) depending on the type of instruction in the current cell in w (Similarly for V^1). Note that U^1 and V^1 are not string variables, but just shorthands defined in terms of U and V respectively. The above disjunction over transitions is finite since the transition function of M is finite, and hence $Next$ is a formula.

Removing Dis-equations: Note that dis-equalities in θ can always be replaced by a disjunction of equalities. This is because the negated equalities in θ are of the form $Var_{str} \neq ID$, and since Q, N, w are all finite, and since storage tape contents are already quantified away, we can represent a disequality as a disjunction of equalities. Hence, we only have positive word equations in θ. Finally, an important observation about our undecidability result is that the formula we constructed in the proof can be easily converted to a formula which has at most two occurrences of any variable [1]. Hence, we get the following final theorem.

[1] We would like to thank Professor Rupak Majumdar with this and other improvements

Theorem 10 *The validity problem for the set of $\mathcal{L}_e^1$ sentences with $\forall\exists$ quantifier alternation over positive word equations, and with at most two occurrences of any variable is undecidable.*

Bounding the Inner Existential Quantifiers: Observe that in θ all the inner quantifiers $S_1, \cdots, S_4$ are bounded since they are substrings of S. One way to express the bounding is to use the length function, e.g., $len(S_1) <= len(S)$. Hence, we observe that the set of $\mathcal{L}_{e,l}^1$ sentences with a single universal quantifier followed by a block of inner bounded existential quantifiers is undecidable.
Choice of Two-counter Machines: The reason we chose two-counter machines as opposed to vanilla Turing machines is that the contents of the storage tapes (defined by the regular expression b^*) are easily represented as word equations: $Xb = bX$. On the other hand, capturing the tape contents of Turing machines using word equations seems impossible.

6 Related Work

In his original 1946 paper, Quine [22] showed that the first-order theory of string equations (i.e., quantified sentences over Boolean combination of word equations) is undecidable. Due to the expressibility of many key reliability and verification questions within this theory, this work has been extended in many ways.

One line of research studies fragments and modifications of this base theory which are decidable. Notably, in 1977, Makanin famously proved that the satisfiability problem for the quantifier-free theory of word equations is decidable [15]. In a sequence of papers, Plandowski and co-authors showed that the complexity of this problem is in PSPACE [19, 20]. Stronger results have been found where equations are restricted to those where each variable occurs at most twice [23] or in which there are at most two variables [5, 6, 11]. In the first case, satisfiability is shown to be NP-hard; in the second, polynomial (which was improved further in the case of single variable word equations).

Concurrently, many researchers have looked for the exact boundary between decidability and undecidability. During our research on related work, we found that Durnev [7] and Marchenkov [16] both showed that the $\forall\exists$ sentences over word equations is undecidable. Note that Durnev's result is closest to our undecidability result. The main difference is that our proof is considerably simpler because of the use of two-counter machines, as opposed to certain non-standard machines used by Durnev. We also note corollaries regarding number of occurences of a variable, and $\mathcal{L}_{e,l}^1$ sentences with a single universal followed by bounded existentials. On the other hand, Durnev uses only 4 string variables to prove his result, while we use 7. We believe that we can reduce the number of variables, at the expense of a more complicated proof.

Word equations augmented with additional predicates yield richer structures which are relevant to many applications. In the 1970s, Matiyasevich formulated a connection between string equations augmented with integer coefficients whose integers are taken from the Fibonacci sequence and Diophantine equations [17]. In particular, he showed that proving undecidability for the satisfiability problem of this theory would suffice to solve Hilbert's 10th Problem in a novel way. Schulz [25] extended Makanin's satisfiability algorithm to the class of formulas where each variable in the equations is specified to lie in a given regular set. This is a strict generalization of the solution sets of word equations. [12] shows that the class of sets expressible through word equations is incomparable to that of regular sets.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Prentice Hall (India) Ltd., 1979.

2. C. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories*. PhD thesis, Stanford University, 2003.
3. A. Blumensath. Automatic structures. Diploma thesis, RWTH-Aachen, 1999.
4. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *13th ACM Conference on Computer and Communications Security (CCS '06)*, 2006.
5. W. Charatonik and L. Pacholski. Word equations in two variables. In *Proceedings of the Second International Workshop on Word Equations and Related Topics*, volume 677 of *LNCS*, pages 43–57, 1991.
6. R. Dąbrowski and W. Plandowski. On word equations in one variable. In *MFCS*, volume 2420 of *LNCS*, pages 212–220, 2002.
7. V. Durnev. Undecidability of the positive $\forall\exists^3$-theory of a free semigroup. *Siberian Mathematical Journal*, 36(5):1067–1080, 1995.
8. H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer-Verlag, 1994.
9. M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2007.
10. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *In ACM Conference on Programming Language Design and Implementation (PLDI)*, 2005.
11. L. Ilie and W. Plandowski. Two-variable word equations (extended abstract). In *STACS*, volume 1770 of *LNCS*, pages 122–132, 2000.
12. J. Karhumäki, W. Plandowski, and F. Mignosi. The expressibility of languages and relations by word equations. In *Automata, Languages and Programming, 24th International Colloquium, (ICALP)*, pages 98–109, 1997.
13. A. Kieżun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *International Symposium on Software Testing and Analysis (ISSTA '09)*, 2009.
14. R. Majumdar. Private correspondence, 2010. SWS, MPI, Kaiserslautern, Germany.
15. G. Makanin. The problem of solvability of equations in a free semigroup. *Sbornik: Mathematics*, 32(2), 1977.
16. S. S. Marchenkov. Unsolvability of positive $\forall\exists$-theory of free semi-group. *Sibirsky mathmatichesky jurnal*, 23(1):196–198, 1982.
17. Y. Matiyasevich. Word equations, Fibonacci numbers, and Hilbert's tenth problem (extended abstract). In *Workshop on Fibonacci Words*, Turku, Finland, Sept. 2006.
18. Y. Matiyasevich. Computation paradigms in light of Hilbert's tenth problem. In *New Computational Paradigms*, pages 59–85. Springer New York, 2008.
19. W. Plandowski. Satisfiability of word equations with constants is in PSPACE. In *40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 495–500, Oct. 1999.
20. W. Plandowski. An efficient algorithm for solving word equations. In *ACM Symposium on Theory of Computing (STOC)*, 2006.
21. M. Presburger. Über de vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchen, die addition als einzige operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématicienes des Pays Slaves*, pages 92–101, 395, Warsaw, 1927.
22. W. V. Quine. Concatenation as a basis for arithmetic. *The Journal of Symbolic Logic*, 11(4):105–114, 1946.
23. J. M. Robson and V. Diekert. On quadratic word equations. In *STACS '99*, volume 1563 of *LNCS*, pages 217–226, 1999.
24. P. Saxena, D. Akhawe, S. Hanna, S. McCamant, F. Mao, and D. Song. A symbolic execution framework for JavaScript. In *Accepted in the international proceedings of IEEE Security & Privacy*, May 2010.
25. K. U. Schulz. Makanin's algorithm for word equations - two improvements and a generalization. In *IWWERT '90: Proceedings of the First International Workshop on Word Equations and Related Topics*, pages 85–150, London, UK, 1992. Springer-Verlag.
26. M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.
27. G. Wassermann and Z. Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *ACM Conference on Programming Language Design and Implementation (PLDI '07)*, 2007.