CADE-24

24th International Conference on Automated Deduction
Lake Placid, New York, 9-14 June, 2013

Workshop Program

Automated Deduction: Decidability, Complexity, Tractability (ADDCT)

Silvio Ghilardi, Ulrike Sattler,
Viorica Sofronie-Stokkermans, Ashish Tiwari

10 June 2013

Preface

This volume contains the papers presented at the workshop ADDCT 2013: Automated Deduction: Decidability, Complexity, Tractability, held in Lake Placid on June 10, 2013, affiliated with the 24th Conference on Automated Deduction (CADE 24). ADDCT 2013 is the third of the ADDCT workshops: the previous workshops were ADDCT 2007 (held in Bremen, together with CADE 21) and ADDCT 2009 (held in Montreal together with CADE 22).

The goal of ADDCT is to bring together researchers interested in

- *Decidability*, in particular decision procedures based on logical calculi such as: resolution, rewriting, tableaux, sequent calculi, or natural deduction, but also decidability in combinations of logical theories;
- *Complexity*, especially complexity analysis for fragments of first- (or higher) order logic and complexity analysis for combinations of logical theories (including parameterized complexity results);
- *Tractability* (in logic, automated reasoning, algebra, ...);
- *Application domains* for which complexity issues are essential (verification, security, databases, ontologies, ...).

With the development of computer science these problems are becoming extremely important. Although general logical formalisms (such as predicate logic or number theory) are undecidable, decidable theories or decidable fragments thereof (sometimes even with low complexity) often occur in mathematics, in program verification, in the verification of reactive, real time or hybrid systems, as well as in databases and ontologies. It is therefore important to identify such decidable fragments and design efficient decision procedures for them. It is equally important to have uniform methods (such as resolution, rewriting, tableaux, sequent calculi, ...) which can be tuned to provide algorithms with optimal complexity.

The programme of ADDCT 2013 includes one invited talk, by Vijay Ganesh, on decidability and undecidability results for word equations with length and regular expression constraints and 5 contributed papers. We allowed the possibility of submitting to ADDCT not only original papers, but also presentation-only papers, describing work presented in papers which are already published. We thank the programme committee and the additional reviewers for their careful referee reports.

June 2013

Silvio Ghilardi
Università degli Studi di Milano

Viorica Sofronie-Stokkermans
University Koblenz-Landau and
Max-Planck-Institut für Informatik, Saarbrücken

Ulrike Sattler
University of Manchester

Ashish Tiwari
Stanford Research International

The workshop organizers greatly benefited from using the EasyChair system.

Conference Organization

Programme Chairs

Silvio Ghilardi
Ulrike Sattler
Viorica Sofronie-Stokkermans
Ashish Tiwari

Programme Committee

Carlos Areces
Franz Baader
Peter Baumgartner
Maria Paola Bonacina
Christian Fermüller
Silvio Ghilardi
Rajeev Gore
Ullrich Hustadt
Carsten Lutz
Christopher Lynch
Felix Klaedtke
Silvio Ranise
Ulrike Sattler
Renate Schmidt
Viorica Sofronie-Stokkermans
Ashish Tiwari
Luca Vigano

External Reviewers

Jinbo Huang
Barbara Morawska

Table of Contents

(Un)Decidability Results for Word Equations with Length and Regular Expression Constraints

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 and validity problems for languages that can express solutions to word equations with length constraints. The atomic formulas over this language are equality over string terms (word equations), linear inequality over the length function (length constraints), and membership in regular sets. These questions are important in logic, program analysis, and formal verification. Variants of these questions have been studied for many decades by mathematicians. More recently, practical satisfiability procedures (aka SMT solvers) for these formulas have become increasingly important in the context of security analysis for string-manipulating programs such as web applications.

We prove three main theorems. First, we give a new proof of undecidability for the validity problem for the set of sentences written as a $\forall\exists$ quantifier alternation applied to positive word equations. A corollary of this undecidability result is that this set is undecidable even with sentences with at most two occurrences of a string variable. Second, we consider Boolean combinations 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 combinations of word equations and length constraints is decidable. Third, we show that the satisfiability problem for quantifier-free formulas over word equations in *regular solved form*, length constraints, and the membership predicate over regular expressions is also decidable.

1 Introduction

The complexity of the satisfiability problem for formulas over finite-length strings (theories of strings) has long been studied, including by Quine [23], Post, Markov and Matiyasevich [17], Makanin [15], and Plandowski [12,20,21]. While much progress has been made, many questions remain open especially when the language is enriched with new predicates.

Formulas over strings have become important in the context of automated bugfinding [8, 25], and analysis of database/web applications [7, 14, 27]. These program analysis and bugfinding tools read string-manipulation programs and generate formulas expressing their results. These formulas contain equations over string constants and variables, membership queries over regular expressions, and inequalities between string lengths. In practice, formulas of this form have been solved by off-the-shelf satisfiability procedures such as HAMPI [8, 13] or Kaluza [25]. In this context, a deeper understanding of the theoretical aspects of the satisfiability problem for this class of formulas may be useful in practice.

Problem Statement: We address three problems. First, what is a boundary for decidability for fragments of the theory of word equations? Namely, is the $\forall\exists$-fragment of the theory of word equations decidable? Second, is the satisfiability problem for quantifier-free formulas over word equations and the length function decidable under some minimal practical conditions? Third, is the satisfiability problem for quantifier-free formulas over word equations, the length function, and regular expressions decidable under some minimal practical conditions?

The question of whether the satisfiability problem for the quantifier-free theory of word equations and length constraints is decidable has remained open for several decades. Our decidability results are a partial and conditional solution. Matiyasevich [18] observed 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 (which resolved the famous problem) [17, 18].

Summary of Contributions:

1. We show that the validity problem (decision 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 3)
2. We show that if word equations can be converted to a *solved form* then the satisfiability problem for Boolean combinations of word equations and length constraints is decidable. (Section 4)
3. The above-mentioned decidability result has immediate practical impact for applications such as bug-finding in JavaScript and PHP programs. We empirically studied the word equations in the formulas generated by the Kudzu JavaScript bugfinding tool [25] and verified that most word equations in such formulas are either already in solved form or can be automatically and easily converted into one. (Section 4). We further show that the satisfiability problem for quantifier-free formulas constructed out of Boolean combinations of word equations in *regular solved form* with length constraints and the membership predicate for regular sets is also decidable. This is the first such decidability result for this set of formulas. (Section 5)

We now outline the layout of the rest of the paper. In Section 2 we define a theory of word equations, length constraints, and regular expressions. In Section 3 we prove the undecidability of the theory of $\forall\exists$ sentences over positive word equations. In Section 4 (resp. Section 5) we give a conditional decidability result for the satisfiability problem for the quantifier-free theory of word equations and length constraints (resp. word equations, length constraints, and regular expressions). Finally, in Section 6 we provide a comprehensive overview of the decidability/undecidability results for theories of strings over the last several decades.

2 Preliminaries

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 nonnegative integers. The empty string is represented by ϵ.
Terms: Terms may be 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* or interchangeably by $\cdot$). A length term (t_{len} in Figure 1) is an element of var_{int}, an element of Con_{int}, the length function applied to a string term, a constant integer multiple of a length term, or a sum of length terms.
Atomic Formulas: There are three types of atomic formulas: (1) word equations ($A_{wordeqn}$), (2) length constraints (A_{length}), or (3) membership in a set defined by a regular expression (A_{regexp}). Regular expressions are defined inductively, where constants and the empty string form the base case, and the operations of concatenation, alternation, and Kleene star are used to build up more complicated expressions (see details in [10]). Regular expressions may not contain variables.
Formulas: Formulas are defined inductively over atomic formulas (see Figure 1). We include quantifiers of two kinds: over string variables and over integer variables.
Formula Nomenclature: We now establish notation for the classes of formulas we will analyze. Define $\mathcal{L}^1_{e,l,r}$ to be the first-order two-sorted language over which the formulas described above (Figure 1) are

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

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

constructed. This language contains word equations, length constraints, and membership in given regular sets. 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). Let $\mathcal{L}^1_{e,l}$ be the analogous set of first-order formulas restricted to word equations and length constraints as the only atomic formulas, and let $\mathcal{L}^1_{e}$ be the collection of formulas whose only atomic formulas are word equations. 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 $\mathcal{L}^1_{e}$, respectively.

Recall that a formula is in *prenex normal form* if all quantifiers appear at the front of the expression: that is, the formula has a string of quantifiers and then a Boolean combination of atomic formulas. It is a standard result (see, for example [6]) that any first-order formula can be translated into prenex normal form. We therefore assume that all formulas are given in this form. Intuitively, a variable is *free* in a formula if it is not quantified. For example, in the formula $\forall y \phi(y, x)$, the variable y is *bound* while x is *free*. For a full inductive definition, see [6]. A formula with no free variables is called a *sentence*.

2.2 Semantics and 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).

We fix a string alphabet, Σ. Given an $\mathcal{L}^1_{e,l,r}$ formula θ, an *assignment* for θ (with respect to Σ) is a map from the set of free variables in θ to $\Sigma^* \cup \mathbb{N}$ (where string variables are mapped to strings and integer variables are mapped to numbers). Given such an assignment, θ can be interpreted as an assertion about Σ^* and $\mathbb{N}$. If this assertion is true, then we say that θ itself is *true* under the assignment. If there is some assignment which makes θ true, then θ is called *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. Note that this is a broad definition: equisatisfiable formulas may have different numbers of assignments and, in fact, 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. Satisfiability procedures must have three properties: soundness, completeness, and termination. Soundness and completeness guarantee that the procedure returns "satisfiable" if and only if the input formula is indeed satisfiable. Termination means that the procedure halts on all inputs. In a practical implementation, some of these requirements may be relaxed for the sake of improved typical performance.

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 a given sentence in Q is true under all assignments.

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* [21]. Namely, fix a set of new variables V disjoint from all of 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. (See example 1 below.)

Another tool for compactly encoding many solutions to a formula is the use of *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. (See example 2 below.)

Combining these two representations, we also 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

We consider some sample formulas and their solution sets. The string alphabet is $\Sigma = \{a, b\}$. (Many of the examples in this paper are from existing literature by Plandowski et al. [21].)

Example 1 *Consider the $\mathcal{L}^0_e$ formula which is a word equation $X = aYbZa$ with three variables (X, Y, Z) and two string constants (a, b). The set of all solutions to this equation is described by the assignment $X \mapsto aybza, Y \mapsto y, Z \mapsto z$, where $V = \{y, z\}$ is the set of unfixed parts. Any choice of $y, z \in \Sigma^*$ yields a solution to the equation.*

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. The map $X \mapsto (ab)^i a$ with $i \geq 0$ is also an assignment which gives a solution. In fact, this assignment (with integer parameters) exactly describes all possible solutions of the word equation.*

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

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

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

3 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 combinations of word equations) whose prenex normal form has $\forall\exists$ as its quantifier prefix is undecidable.

3.1 Proof Idea

We do a reduction from the halting problem for two-counter machines, which is known to be undecidable [10], to the problem in question. To do so, we encode computation histories as strings. The choice of two-counter machine makes this proof cleaner than other undecidability proofs for this set of formulas (see Section 6 for a comparison with earlier work). The basic proof strategy is as follows: given a two-counter machine M and a finite string w, we construct an $\mathcal{L}^1_e$ sentence $\forall S \exists S_1, \ldots, S_4 \theta(S, S_1, \ldots, S_4)$ such that M does not halt on w iff this $\mathcal{L}^1_e$ sentence is valid. By the construction of θ, this will happen exactly when all assignments to the string variable S are not codes for halting computation histories of M over w. The variables $S_1, \ldots, S_4$ are used to refer to substrings of S and the quantifier-free formula θ expresses the property of S not coding a halting computation history.

3.2 Recalling Two-counter Machines

A *two-counter machine* is a deterministic machine which has a finite-state control, two semi-infinite storage tapes, and a separate read-only semi-infinite input tape. All tapes have a left endpoint and no right endpoint. All tapes are composed of cells, each of which may store a symbol from the appropriate alphabet (the alphabet of the storage tapes is {Z, blank}; the alphabet of the input alphabet is some fixed finite set). 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. Each tape has a corresponding tape-head that may move left, move right, or stay put. The input tape-head cannot move past the right end of the input string. The initial position of all the tape-heads is the leftmost cell of their respective tapes. At each point in the computation, the cell being scanned by each tape-head is called that tape's *current cell.*

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. A non-negative 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 to the right or to the left. We can test whether the number stored in one of the storage tapes is zero by checking if the contents of the current cell of that tape is Z. But, the equality of two numbers stored on the storage tapes 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 [10].

More formally, a two-counter machine M is a tuple $\langle Q, \Delta, \{Z, b, c\}, \delta, q_0, F\rangle$ 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.
– Δ is the finite alphabet of the input tape, $\{Z, b\}$ and $\{Z, c\}$ are the storage tape alphabets for the first and second tapes, respectively. (The distinct blank symbols for the two tapes are a notational convenience.)
– δ is the transition function for the control of M. This function maps the domain, $Q \times \Delta \times \{Z, b\} \times \{Z, c\}$ into $Q \times \{in, stor1, stor2\} \times \{L, R\}$. In words, given a control state and the contents of 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) to move, and whether this tape-head moves left (L) or right (R).

3.3 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. The set of IDs will be determined both by the machine and the given input to the machine.
Definition of ID: An instantaneous description (ID) of a computation step of a two-counter machine M running on input w is the concatenation of the following components.

– Current control state of M: represented by a character over the finite alphabet Q.
– The input w and an encoding of the current input tape cell. The encoding uses string constants to represent the integers between 0 and $|w| - 1$; let N_i denote the string constant encoding the number i.
– 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.

Each component of an ID is separated from the others by an appropriate special character. In what follows, we will suppress discussion of this separator and we will assume that it is appropriately located inside each ID. A lengthy but technically trivial modification of our reduction formula could be used to allow for the case where this separator is missing.
Definition of Initial ID: For any two-counter machine M and each input w, there is exactly one initial ID, denoted $Init_{M,w}$. This ID is the result of concatenating the string representations of the following data:

Initial state q_0 of M, w, 0, ϵ, ϵ. The "0" says that the current cell of the input tape contains the 0th letter of w. The two "ϵ"s represent the contents of the two storage tape: both are empty at this point.
Definition of Final ID: We use the standard convention that a two-counter machine halts only after the storage tapes contain the unary representation of the number 0 and the input tape-head has moved to the leftmost position of its tape. The ID of the machine at the end of a computation is therefore the concatenation of representations of $q_f, w, 0, \epsilon, \epsilon$, where q_f is one of the finitely many final control states $q_f \in F$ of M. Observe that there are only finitely many Final IDs.

3.4 Computation History of a Two-counter Machine as a String

A *well-formed computation history* of a two-counter machine M as it processes a given input w is the concatenation of a sequence of IDs separated by the special symbol #. The first ID in the sequence is the initial ID of M on w, and for each i, ID_{i+1} is the result of transforming ID_i according to the transition function of M. A well-formed computation 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 #. If a finite computation history is not accepting, it is either not well-formed or rejecting.

3.5 Alphabet for String Formulas and The Universe of Strings

Given a two-counter machine M and an input string w, we define the associated finite alphabet

$$\Sigma_0 = \{\#q_i N_j w : q_i \in Q, 0 \le j < |w|\}.$$

This alphabet includes all possible *initial segments* of IDs, not including the data about the contents of the storage tapes. We also define $\Sigma_1 = b$ and $\Sigma_2 = c$. We define the alphabet of strings as $\Sigma \equiv \{\Sigma_0 \cup \Sigma_1 \cup \Sigma_2\}$, and the universe of strings as Σ^*. Thus, each valid ID will be in the regular set $\Sigma_0 \Sigma_1^* \Sigma_2^*$.

3.6 The Undecidability Theorem

Theorem 4 *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 an $\mathcal{L}_e^1$-formula $\theta_{M,w}(S, S_1, \ldots, S_4, U, V)$ which describes the conditions for $S_1, \ldots, S_4$ to be substrings of S and S to fail to code an accepting computation history of M over w. Thus,

$$\forall S\, \exists S_1, S_2, S_3, S_4, U, V \;\; (\theta_{M,w}(S, S_1, \cdots, S_4, U, V))$$

is valid if and only if it is not the case that M halts and accepts on w. For brevity, we write θ for $\theta_{M,w}$.

Structure of θ: We will define θ as the disjunction of ways in which S could fail to encode an accepting computation history: either S does not start with the Initial ID, or S does not end with any of the Final IDs, or S is not a well-formed sequence of IDs, or it does not follow the transition function of M over w.

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

Note that the variables S_i $(i = 1, \ldots, 4)$ represent substrings of S.

- **NotInit and NotFinal:** The set NotInit is a finite set of string constants for strings with length at most that of the Initial ID $Init_{M,w}$ which are not equal to $Init_{M,w}$. Similarly, NotFinal is a set of string constants for strings that that are not equal to any of the Final IDs, but have the same or smaller length.
- **NotWellFormedSequence**: This subformula asserts that S is not a sequence of IDs. Recall that, by definition, the set of well-formed IDs is described by the regular expression $\Sigma_0\Sigma_1^*\Sigma_2^* = \Sigma_0 b^* c^*$, where strings in Σ_0 (as defined above) include the ID separator # as well as codes for the control state, w, and letter of w being scanned. A well-formed sequence of IDs is a string of the form $(\Sigma_0 b^* c^*)^* - \epsilon$. Thus, the set described by **NotWellFormedSequence** should be $\Sigma^* - (\Sigma_0 b^* c^*)^*$. In fact, we can characterize this regular set entirely in terms of word equations: a string over $\Sigma = \Sigma_0 \cup \{b, c\}$ is not a well-formed sequence of IDs if and only if it starts with b or c, or contains cb. 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 guaranteed by the following formula NotWellFormedSequence():
$$(S = \epsilon) \vee (S = S_1 \cdot c \cdot b \cdot S_4).$$
- **Next**:
$Next()$ asserts that the pair of variables S_2, S_3 form a legal transition. It is a disjunction over all (finitely many) possible pairs of IDs defined by the transition function:
$$\bigvee_{(q_2,d,g_1,g_2,q_3,t,m)\in\delta;0\leq n_2,n_3<|w|} S_2 = \#q_2 N_{n_2} wUV \wedge S_3 = \#q_3 N_{n_3} w f(U) g(V)$$
where $d = w(n_2)$; $g_1 = Z$ if $U = \epsilon$ and $g_1 = b$ otherwise; $g_2 = Z$ if $V = \epsilon$ and $g_2 = c$ otherwise; and $f(U), g(V), N_{n_3}$ are the results of modifying the stack contents represented by U, V and input tape-head position according to whether the value of t is in, $stor1$, or $stor2$ and whether m is L or R. Note that the disjunction is finite and is determined by the transition function and w. Also note that each of $\#q_2 N_{n_2} w$ and $\#q_3 N_{n_3} w$ is a single letter in Σ_0.

Simplifying the formula: The formula θ contains negated equalities in the subformula $\neg Next$. However, each of these may be replaced by a disjunction of equalities because $Q, |w|, \delta$ are each finite. Hence, we can translate θ to a formula containing only conjunctions and disjunctions of positive word equations. We also observe 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]. Thus, we get the final theorem.

Theorem 5 *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 $S_1, \cdots, S_4, U, V$ are bounded since they are substrings of S. The length function, $len(S_i) \leq len(S)$, can be used to bound these quantifiers.

Corollary 6 *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.*

4 Decidability Theorem

In this section we demonstrate the existence of an algorithm deciding whether any $\mathcal{L}_{e,l}^0$ formula has a satisfying assignment, under a minimal and practical condition.

[1] We thank Professor Rupak Majumdar for observing this and other improvements.

4.1 Word Equations and Length Constraints

Word equations by themselves are decidable [21]. Also, systems of inequalities over integer variables are decidable because these are expressible as quantifier-free formulas in the language of Presburger arithmetic and Presburger arithmetic is known to be decidable [22]. 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 describe our observations of word equations in formulas generated by the Kudzu JavaScript bugfinding tool [25]. In particular, we saw that these equations either already appeared in solved form or could be algorithmically converted into one.

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

The crux of the difficulty in establishing an unconditional decidability result is that it is not known whether the length constraints implied by a set of word equations have a finite representation [21]. In the case when the implied constraints do have a finite representation, we look for a satisfying assignment to both the implied and explicit constraints. Such a solution can be translated into a satisfying assignment of the word equations when the implied constraints of the system of equations is equisatisfiable with the system itself.

4.3 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.[2]

– Every formula in $\mathcal{S}$ is of the form $X = t$, where X is a variable occurring in w and t is the result of finitely many concatenations of constants in w (with possible integer parameters) and possible unfixed parts. (Recall the definitions for integer parameters and unfixed parts from Section 2.) All integer parameters i in $\mathcal{S}$ are linear, of the form ci 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 the conjunction of all the formulas in $\mathcal{S}$, denoted $\wedge\mathcal{S}$. If there is an algorithm which converts any given word equation to solved form (if one exists, and halts in finite time otherwise), and if $\wedge\mathcal{S}$ is the output of this algorithm when given w, we say that the *effective solved form* of w is $\wedge\mathcal{S}$. Solved form equations can have integer parameters, whereas $\mathcal{L}^0_{e,l}$ formulas cannot. The solved form is used to extract all necessary and sufficient length information *implied by* w.

Example 7 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:

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

Example 8 Unsatisfiable Solved Form Example: *Consider the formula*

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

[2] The idea of solved form is well known in equational reasoning, theorem proving, and satisfiability procedures for rich logics (aka SMT solvers).

with variables X, Y. The set of solutions to the equation $abX = Xba$ is described by the map $X \mapsto (ab)^i a$ with $i \geq 0$ (recall Example 2). Hence the solved form for the system of two equations is:

$$X = (ab)^i a \wedge Y = (ab)^{i-1} a \qquad (i > 0)$$

The length constraints implied by this system are

$$len(X) = 2c + 1 \wedge len(Y) = 2c - 1 \wedge len(X) < 2 \qquad\qquad (c > 0).$$

This is unsatisfiable. Hence, the original formula is also unsatisfiable.

Example 9 Word Equations Without a Solved Form: *Not all word equations can be written in solved form. Consider the equation*

$$XabY = YbaX.$$

The map $X \mapsto a, Y \mapsto aa$ is a solution, as is $X \mapsto bb, Y \mapsto b$. However, it is known that the solutions to this equation cannot be expressed using linear integer parameters [21]. Thus, not all satisfiable systems of equations can be expressed in solved form.

4.4 Why Solved Form?

For word equations with an equivalent solved form, 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 from the (implied and given) length constraints, because it guarantees 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 [22]. This insight forms the basis of our decidability results. It is noteworthy that most word equations that we have encountered in practice [25] are either in solved form or can be automatically converted into one.

4.5 Proof Idea for Decidability

Without loss of generality, we consider formulas that are the 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.

Example 10 *Consider the equation $X = abY$. We have the following set $\mathcal{R}$ of implied length constraints:*

$$\{len(X) = 2 + len(Y), len(Y) \geq 0\}.$$

The set $\mathcal{R}$ is finite but exhaustive. That is, any other length constraint implied by the equation $X = abY$ is either in $\mathcal{R}$ or is implied by $\mathcal{R}$. Consider the $\mathcal{L}^0_{e,l}$ formula

$$X = abY \wedge len(Y) > 1,$$

Note that $X = abY$ is satisfiable, say by the assignment with unfixed parts $X \mapsto aby, Y \mapsto y$. It remains to check whether there is a solution (represented by some choice of the unfixed part) which satisfies the length constraints $\mathcal{R} \cup \{len(Y) > 1\}$. A solution to the set of integer inequalities is $len(X) = 4, len(Y) = 2$. Translating this to a solution of the original formulas amount to "back-solving" for the exponent of unfixed parts in the solution to the word equation. That is, since $X \mapsto aby, Y \mapsto y$ is a satisfying assignment, we can pick any string of length 2 for y: say, $X \mapsto abab, Y \mapsto ab$.

Taking this example further, consider the $\mathcal{L}^0_{e,l}$ formula

$$X = abY \wedge len(Y) > 1 \wedge len(X) \leq 2.$$

The set of length constraints is now: $\{len(X) = 2 + len(Y), len(Y) \geq 0, len(Y) > 1, len(X) \leq 2\}$. *This is not satisfiable, so neither is the original formula.*

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 use it to prove 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 satisfy all constraints in $\mathcal{R}$. And, $\mathcal{R}$ 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.

4.6 Decidability Theorem

We prove a set of lemmas culminating in the 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. Since a word equation w is logically equivalent to its solved form $\mathcal{S}$, 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 $\mathcal{S}$. In $\mathcal{R}$, we will have integer variables associated with each string variable in w, integer variables associated with each unfixed part appearing in the RHS of an equation in $\mathcal{S}$, and integer variables associated with each integer parameter appearing in the RHS of an equation in $\mathcal{S}$. For each X appearing in w, consider the equation in $\mathcal{S}$ whose LHS is X: $X = t_1 \cdots t_n$, where each t_i is either (1) a constant from w, (2) a constant from w raised to some integer parameter, or (3) an unfixed part. This equation implies a length equation of the form: $len(X) = C + i_1c_1 + \cdots + i_kc_k + len(y_1) + \cdots len(y_j)$, where C is the sum of the lengths of constants in w that appear on the RHS without an integer parameter; the c_i terms are the lengths of constants with integer parameters; and there are terms for each unfixed part appearing in the equation. The only other length constraints associated with this equation say that the unfixed parts and the integer parameters may be arbitrarily chosen: $i_r \geq 0$, $len(y_s) \geq 0$ for each $1 \leq r \leq k$ and $1 \leq 1 \leq s \leq j$. Note that the minimum length of X is the expression above where we choose each $i_r = 0$ and each $len(y_s) = 0$. Let $\mathcal{R}$ be the union over X in w of the (finitely many) length constraints associated with X discussed above. Since $\mathcal{S}$ is finite, so is $\mathcal{R}$.

It remains to prove that $\mathcal{R}$ is complete. By definition of solved form, all length constraints implied by $\mathcal{S}$ are of the form included in $\mathcal{R}$. Thus, $\mathcal{R}$ is complete for $\mathcal{S}$. Since $\mathcal{S}$ is logically equivalent with w, they imply the same length constraints. Hence, $\mathcal{R}$ is complete for w as well.

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. Since $\mathcal{R}$ is finite, the conjunction of all its elements is a formula of $\mathcal{L}^0_{e,l}$

($\Rightarrow$) If w is satisfiable, then so is $\mathcal{R}$: Suppose w is satisfiable and consider some satisfying assignment w. Then since $\mathcal{R}$ is implied by w, the lengths of the strings in this assignment satisfy all the constraints in $\mathcal{R}$. Thus, this set of lengths witnesses the satisfiability of $\mathcal{R}$.

($\Leftarrow$) If $\mathcal{R}$ is satisfiable, then so is w: Suppose $\mathcal{R}$ is satisfiable. Any solution of $\mathcal{R}$ gives a collection of lengths for the variables in w. An assignment that satisfies w is given by choosing arbitrary strings of the prescribed length for the unfixed parts and choosing values of the integer parameters prescribed by the solution of $\mathcal{R}$.

Theorem 11 *The satisfiability problem for $\mathcal{L}^0_{e,l}$ formulas is decidable, provided that there is an algorithm to obtain the solved forms of word equations for which they exist.*

Proof. We assume without loss of generality that the given $\mathcal{L}^0_{e,l}$ formula is the conjunction of a single word equation with some number of length constraints. (Generalizing to arbitrary $\mathcal{L}^0_{e,l}$ formulas is straightforward.) 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).

Plandowski's algorithm [21] decides satisfiability of word equations; known algorithms for formulas of Presburger arithmetic can decide the satisfiability of systems of linear length constraints. Thus, begin by running these algorithms (in parallel) to decide if (separately) ϕ and θ are satisfiable. If either of these return UNSAT, we return UNSAT.

Using the assumption that the word equation ϕ has an effective solved form, compute this form $\mathcal{S}$ and the associated (complete and finite) implied set $\mathcal{R}$ of linear length constraints (as in Lemma 1). By Lemma 2, it is now sufficient to check the satisfiability of $(\wedge\mathcal{R}) \wedge \theta$. This can be done by a second application of an algorithm for formulas in Presburger arithmetic, because the length constraints implied by ϕ are all linear. If this system of linear inequalities is satisfiable, return SAT, otherwise, we return UNSAT.

This procedure is a sound, complete and terminating procedure for $\mathcal{L}^0_{e,l}$-formulas whose word equations have effective solved forms.

4.7 Practical Value of Solved Form and the Decidability Result

JavaScript programs often process strings. These strings are entered into input forms on web-pages or are substrings used by JavaScript programs to dynamically generate web-pages or SQL queries. During the processing of these strings, JavaScript programs often concatenate these strings to form larger strings, use strings in assignments, compare string lengths, construct equalities between strings as part of if-conditionals or use regular expressions as basic "sanity-checks" of the strings being processed. Hence, any program analysis of such JavaScript programs results in formulas that contain string constants and variables, the concatenation operation, regular expressions, word equations, and uses of the length function.

In their paper on an automatic JavaScript testing program (Kudzu) and a practical satisfiability procedure for strings [25], 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 [2, 9]) 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 and generate program inputs for program testing.

We obtained more than 50,000 string constraints (word equations + length constraints) from the Kaluza team (http://webblaze.cs.berkeley.edu/2010/kaluza/). Kaluza is a solver for string constraints, where these constraints are obtained from bug-finding and string analysis of web applications. The constraints are divided into satisfiable and unsatisfiable constraints. We wrote a simple Perl script to count the number of equations per file and the number of equations already in solved form (identifier = expression). We then computed the ratio to see how many examples from this actual data set are already in solved form.

Experimental Results The results are divided into groups based on whether the constraints were satisfiable or not. For satisfiable small equations (approximately 30-50 constraints per file), about 80% were already in solved form. For satisfiable large equations (around 200 constraints per file), this number rose to approximately 87%. Among the unsatisfiable and small equations (less than 20 constraints per file), again about 80% were already in solved form. Large (greater than 4000 constraints) unsatisfiable equations were in solved form a slightly smaller percentage of the time: 75%.

5 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. Thus, it may seem 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.

Example 12 *Consider the $\mathcal{L}^0_{e,l,r}$ formula*

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

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

$$\begin{cases} len(X) = 2i + 1, i \geq 0 & \textit{implied by the word equation and regular expression} \\ len(X) \leq 3. \end{cases}$$

This system of length constraints is easily seen to be simultaneously satisfiable: let $i = 0$ or 1 and hence $len(X) = 1$ or 3. However, the formula is **not** *satisfiable since solutions of the word equation are $X \in (ab)^*a$ 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.

Lemma 3. *If a word equation has a solved form without unfixed parts that is also a regular expression, then there is a finite set of linear length constraints that can be effectively computed from this solved form and which are equisatisfiable with the equation.*

Proof. It is sufficient to recall the fact, from [1], that 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.

Theorem 13 *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.*

The proof is a straightforward extension of the conditional decidability proof given in Section 4.

6 Related Work

In his original 1946 paper, Quine [23] 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 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 [21]. Stronger results have been found where equations are restricted to those where each variable occurs at most twice [24] or in which there are at most two variables [3, 4, 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. Durnev [5] 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 $\mathcal{L}^1_{e,l}$ 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 [26] 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.

Möller [19] studies word equations and related theories as motivated by questions from hardware verification. More specifically, Möller proves the undecidability of the existential fragment of a theory of fixed-length bit-vectors, with a special finite but possibly arbitrary concatenation operation, the extraction of substrings and the equality predicate. Although this theory is related to the word equations that we study, it is more powerful because of the finite but possibly arbitrary concatenation.

References

1. Achim Blumensath. Automatic structures. Diploma thesis, RWTH-Aachen, 1999.
2. C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler. EXE: automatically generating inputs of death. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 322–335. ACM, 2006.
3. W. Charatonik and L. Pacholski. Word equations with two variables. In H. Abdulrab and J.-P. Pécuchet, editors, *IWWERT*, volume 677 of *Lecture Notes in Computer Science*, pages 43–56. Springer, 1991.
4. R. Dabrowski and W. Plandowski. On word equations in one variable. *Algorithmica*, 60(4):819–828, 2011.
5. V. Durnev. Undecidability of the positive $\forall\exists^3$-theory of a free semigroup. *Siberian Mathematical Journal*, 36(5):1067–1080, 1995.
6. H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer-Verlag, 1994.
7. M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In D.S. Rosenblum and S.G. Elbaum, editors, *ISSTA*, pages 151–162. ACM, 2007.
8. V. Ganesh, A. Kiezun, S. Artzi, P.J. Guo, P. Hooimeijer, and M.D. Ernst. HAMPI: A string solver for testing, analysis and vulnerability detection. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2011.
9. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In V. Sarkar and M.W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.
10. J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Pearson/Addison Wesley, 2007.
11. Lucian Ilie and Wojciech Plandowski. Two-variable word equations. *ITA*, 34(6):467–501, 2000.
12. J. Karhumäki, F. Mignosi, and W. Plandowski. The expressibility of languages and relations by word equations. *J. ACM*, 47(3):483–505, 2000.

13. A. Kiezun, V. Ganesh, P.J. Guo, P. Hooimeijer, and M.D. Ernst. HAMPI: a solver for string constraints. In G. Rothermel and L.K. Dillon, editors, *ISSTA*, pages 105–116. ACM, 2009.
14. Rupak Majumdar. Private correspondence. SWS, MPI, Kaiserslautern, Germany, 2010.
15. G.S. Makanin. The problem of solvability of equations in a free semigroup. *Math. Sbornik*, 103:147–236, 1977. English transl. in Math USSR Sbornik 32 (1977).
16. S. S. Marchenkov. Unsolvability of positive $\forall\exists$-theory of free semi-group. *Sibirsky mathmatichesky jurnal*, 23(1):196–198, 1982.
17. Yu. Matiyasevich. Word equations, Fibonacci numbers, and Hilbert's tenth problem. Unpublished. Available at http://logic.pdmi.ras.ru/?yumat/Journal/jcontord.htm, 2006.
18. Yu. Matiyasevich. Computation paradigms in light of Hilbert's Tenth Problem. In S.B. Cooper, B. Löwe, and A. Sorbi, editors, *New Computational Paradigms*, pages 59–85. Springer New York, 2008.
19. Oliver Möller. $\exists BV_{[n]} solvability$. Unpublished Manuscript. SRI International, Menlo Park, CA, USA, October 1996.
20. W. Plandowski. Satisfiability of word equations with constants is in PSPACE. In *FOCS*, pages 495–500. IEEE Computer Society, 1999.
21. W. Plandowski. An efficient algorithm for solving word equations. In J.M. Kleinberg, editor, *STOC*, pages 467–476. ACM, 2006.
22. 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.
23. W. V. Quine. Concatenation as a basis for arithmetic. *The Journal of Symbolic Logic*, 11(4):105–114, 1946.
24. J.M. Robson and V. Diekert. On quadratic word equations. In C. Meinel and S. Tison, editors, *STACS*, volume 1563 of *Lecture Notes in Computer Science*, pages 217–226. Springer, 1999.
25. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *IEEE Symposium on Security and Privacy*, pages 513–528. IEEE Computer Society, 2010.
26. K. Schulz. Makanin's algorithm for word equations-two improvements and a generalization. In K. Schulz, editor, *Word Equations and Related Topics*, volume 572 of *Lecture Notes in Computer Science*, pages 85–150. Springer Berlin / Heidelberg, 1992.
27. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In J. Ferrante and K.S. McKinley, editors, *PLDI*, pages 32–41. ACM, 2007.

Congruence closure with ACI function symbols

Tanji Hu and Robert Givan

Purdue University ECE
{hut,givan}@purdue.edu

Abstract. Congruence closure is the following well known reasoning problem: given a premise set of equations between ground terms over uninterpreted function symbols, does a given query equation follow using the axioms of equality? Several methods have been provided for polynomial-time answers to this question. Here we consider this same setting, but where some of the function symbols are known to be associative, commutative, and idempotent (ACI). Given these additional axioms, does the query equation follow from the premise equations? We provide a sound and complete cubic-time procedure correctly answering such questions. The problem requires exponential space when adding only AC function symbols [18], but requiring idempotence restores tractability . Our procedure is defined by providing a sound and complete "local" rule set for the problem [11]. A "local formula" is a formula mentioning only terms appearing in the premises or query. A local rule set is one for which any derivable local formula has a derivation using only local intermediate formulas. Closures under local rule sets can immediately be constructed in polynomial time by refusing to infer non-local formulas. Finally, we present results on the integration of ACI function symbols and equality inference rules into more general local rule sets.

1 Introduction

Congruence closure is a well studied algorithmic problem: given a set of ground equations over a first-order term language of uninterpreted function symbols, which other ground equations between terms are entailed? Previous study has provided a small variety of approaches to efficiently solving this problem [23, 21, 16] as well as a great deal of work on related algorithms [2, 4, 3, 15]. The importance of this problem is apparent in the foundational nature of equality in almost every deductive setting, in systems representing expressive knowledge for almost any purpose. Keynote examples have been deductive support for advanced compilers and other program analysis tools (e.g., [9]), and reasoning systems for representing mathematics (e.g., [8, 19, 5]).

A range of prior work has been conducted on congruence closure in the presence of additional axioms about some or all of the function symbols that are present, resulting in the "uniform word problem" in a variety of algebraic structures [22, 13, 6, 7]. Much of this work does not allow arbitrary uninterpreted function symbols in addition to those affected by the axioms considered (typically commutativity and/or associativity). Notable among the prior related work is [18] showing that the uniform word problem for commutative semigroups requires exponential space. This result implies that other methods discussing associative-commutative congruence closure that can solve this word problem, e.g. [2], will require at least exponential time to terminate in the worst case.

Our work here studies a case that we believe has been missed in the above panoply of results, but has a distinguished tradeoff between the desired expressiveness and the desired tractability. Here, we allow arbitrary uninterpreted function symbols, some subset of which is labeled as associative, commutative, and idempotent (ACI). ACI functions that arise naturally most obviously include intersection/union, and/or, and integer maximization/minimization. Program analysis frequently involves integer maximization or Boolean abstractions.

Here, we provide a cubic-time complete inference algorithm for congruence closure in this setting. Our procedure is defined using the technology of *local rule sets* [11, 20]; local rule sets are those for which the computed inference relation is unchanged by restricting inference to the terms mentioned in the premises and query. Any local rule set describes a polynomial-time decidable inference relation. In addition to providing a complete inference relation for the ACI congruence closure problem using a local rule set, we consider the problem of adding the ACI designation to function symbols in an arbitrary rule set. With no restrictions on the rule set, adding this designation may be expensive; however, we present a natural restriction on inference rules for which any local rule set can be augmented by ACI designations while retaining locality and thus polynomial-time decidability. Under this restriction, ACI function-symbol arguments can only be accessed by the inference rules in manners independent of the order or multiplicity of their presentation.

We proceed as follows: first, we present brief technical background on locality and on congruence closure, including a local rule set for congruence closure. Second, we present a simple rule set for reasoning about the ACI properties of terms resulting from ACI function symbols. Third, we prove this rule set together with the congruence closure rule set provides semantically sound and complete inference for the congruence closure problem with multiple ACI function symbols. Fourth, we prove that the resulting rule set is local, immediately providing a cubic-time procedure for the inference relation. Finally, we discuss integration with arbitrary rule sets restricted to access ACI terms via tuples of their arguments.

2 Technical Background

2.1 Locally Restricted Inference

We adopt the following definitions for reference almost directly from [11]. In the these definitions, Σ is any set of ground atomic formulas in first-order logic, and φ any single ground atomic formula.

Definition 1. *A* Horn clause *is a first order formula of the form* $(\psi_1 \wedge \ldots \wedge \psi_n) \rightarrow \psi$ *where* ψ *and the* ψ_i *are atomic formulas. For any* rule set *of Horn clauses* R*, we write* $\Sigma \vdash_R \varphi$ *whenever* $\Sigma \cup U(R) \vdash \varphi$ *in first-order logic, where* $U(R)$ *is the set of universal closures of Horn clauses in* R*.*

We can characterize $\vdash$ syntactically by defining derivations under the rules R.

Definition 2. *A* derivation *from Σ using rule set R is a sequence of ground atomic formulas $\psi_1, \ldots, \psi_n$ such that ψ_n is φ and for each ψ_i there exists a Horn clause $\theta_1 \wedge \ldots \wedge \theta_k \rightarrow \theta$ in R and a ground substitution σ such that $\sigma[\theta]$ is ψ_i and each formula of the form $\sigma[\theta_j]$ is either a member of Σ a formula ψ_j for $j < i$. The* length *of the derivation is the number n of ground formulas in the sequence.*

We then have $\Sigma \vdash_R \varphi$ if and only if there is a derivation of φ from Σ using R. Next, by restricting inference to terms mentioned in Σ or φ, we get a polynomial-time decidable inference relation that may or may not the same as $\vdash_R$.

Definition 3. *Let Υ be a set of terms that is closed under the subterm relation. We say that a ground atomic formula ψ is* local *to Υ if every term appearing in ψ is in Υ. For Γ a set of ground atomic formulas, let $\Upsilon(\Gamma)$ be the subterm-closed set of terms appearing in Γ. We write $\Sigma \Vdash_{R,\Upsilon} \varphi$ if there exists a local derivation of φ from Σ, i.e. one such that every atomic formula in the derivation is local to Υ. We omit Υ to get $\Sigma \Vdash_R \varphi$ when Υ is $\Upsilon(\Sigma \cup \{\varphi\})$.*

McAllester [20] provides a simple proof that the inference relation $\Vdash_R$ is polynomial-time decidable for any finite R; a straight-forward inference procedure can grow a set of derivable ground atoms from Σ by repeatedly considering each inference rule at polynomial time cost, staying always within the polynomially many ground atoms local to $\Upsilon(\Sigma \cup \{\varphi\})$.

Definition 4 (McAllester, 1993). *The rule set R is* local *if the restricted inference relation $\Vdash_R$ is the same as the unrestricted inference relation $\vdash_R$.*

Every inference relation that can be defined by a local rule set is then polynomial-time decidable. It has also been shown [11] that every polynomial-time predicate can be defined by a local rule set.

2.2 Congruence Closure

The following inference rules E define a complete, local inference relation for ground atomic equational premises and ground equational queries.

(eq-refl) $\rightarrow x = x$ (eq-symm) $x = y \rightarrow y = x$

(eq-trans) $x = y \wedge y = z \rightarrow x = z$ (eq-congr1) $x = y \rightarrow f(x) = f(y)$

(eq-congr2) $x_1 = y_1 \wedge x_2 = y_2 \rightarrow g(x_1, x_2) = g(y_1, y_2)$

For notational simplicity, we assume wlog that all function symbols have arity at most 2. We note that the eq-congr rules are actually abbreviations for finitely many rules, one for each function symbol of the given arity. When the equality rules are included with other rule sets, we will also assume eq-congr rules for each predicate symbol of the larger rule set, representing the same principle of substitution of equals: e.g., $x = y \wedge P(x) \rightarrow P(y)$.

The rule set E has been proven complete even when locally restricted ($\Vdash_E$ is complete), proving E is local and providing a polynomial-time decision procedure for this problem, called *congruence closure* [21]. We refer to any rule set containing the rules of E, among others, as an *equational* rule set.

3 A Congruence/ACI Rule Set

Here, we provide a set ACI of inference rules for binary function symbols known to be ACI (associative, commutative, and idempotent ($f(x,x) = x$)). We assume now that some subset of the function symbols have been designated as ACI. For each such function symbol f, we introduce a new binary predicate $\preccurlyeq_f$, and the rules in ACI are schemas providing one rule for each such function symbol. The rules can be best understood by thinking of the new atoms $x \preccurlyeq_f y$ as standing for "there exists w such that $f(x,w) = y$."

(ACI-trans)	$x \preccurlyeq_f y \ \wedge\ y \preccurlyeq_f z \ \rightarrow\ x \preccurlyeq_f z$	(ACI-refl)	$\rightarrow\ x \preccurlyeq_f x$
(ACI-sup)	$x \preccurlyeq_f z \ \wedge\ y \preccurlyeq_f z \ \rightarrow\ f(x,y) \preccurlyeq_f z$	(ACI-sub1)	$\rightarrow\ x \preccurlyeq_f f(x,y)$
(ACI-antisym)	$x \preccurlyeq_f y \ \wedge\ y \preccurlyeq_f x \ \rightarrow\ x = y$	(ACI-sub2)	$\rightarrow\ y \preccurlyeq_f f(x,y)$

Here, the new predicates $\preccurlyeq_f$ implement inference from the ACI axioms, but these predicates are not intended to be part of premise sets or query formulas. We designate the new predicates as *hidden*, prohibiting them in premise sets and queries. This designation affects the claims we later make of rule-set locality and completeness. In each case, these claims refer to inference problems without the hidden predicates in the premises or query.

In the coming sections, we will prove this rule set, in combination with the equality rules E, is both correct ($\vdash_{E \,\cup\, \mathrm{ACI}}$ is sound and complete) and local ($\Vdash_{E \,\cup\, \mathrm{ACI}}$ is the same as $\vdash_{E \,\cup\, \mathrm{ACI}}$), thus providing a polynomial time inference procedure for the congruence closure problem in the presence of known ACI function symbols.

4 Correctness of the Congruence/ACI Rule Set

In this section we prove that the inference rules $E \cup \mathrm{ACI}$ are sound and complete for ground inference on equations. Since our rules and our premise sets are first-order formulas, completeness here refers to standard first-order logic interpretations. Note again that the introduced predicates $\preccurlyeq_f$ are considered hidden and do not appear in premise sets or queries.

Definition 5. *The* ACI-congruence theory $\mathcal{A}$ *is the set of first-order axioms of equality (reflexivity, symmetry, transitivity, and substitution of equals for equals) together with the axioms of associativity, commutativity, and idempotence for each ACI function symbol. A ground premise set Σ of equations entails a ground query equation φ in the ACI-congruence theory, written $\Sigma \cup \mathcal{A} \models \varphi$, when every first-order interpretation satisfying $\Sigma \cup \mathcal{A}$ also satisfies φ.*

Theorem 1 (Soundness and Completeness). *For any ground premise set of equations Σ and ground query equation φ, we have $\Sigma \cup \mathcal{A} \models \varphi$ if and only if $\Sigma \vdash_{E \,\cup\, \mathrm{ACI}} \varphi$.*

Proof. We consider each direction of the theorem separately.

(Soundness) A simple induction on derivation length shows that every ground atom in a derivation from Σ using $E \cup \mathrm{ACI}$ satisfies the following invariants:

$=$: Atoms $x = y$, for any terms x and y, satisfy $\Sigma \cup \mathcal{A} \models x = y$.

$\preccurlyeq_f$: Atoms $x \preccurlyeq_f y$, for any terms x and y and ACI function symbol f, satisfy $\Sigma \cup \mathcal{A} \models \exists z f(x, z) = y$.

Each rule preserves these invariants. The rule ACI-antisym is the hardest to check. We must check that $\mathcal{A} \cup \{\exists z f(x, z) = y, \exists w f(y, w) = x\} \models x = y$. Introducing Skolem constants we have $f(x, c_1) = y$ and $f(y, c_2) = x$. From these we can show $f(x, f(y, f(c1, c2)))$ is equal to x and also to y using the given premises. For instance $f(x, f(y, f(c1, c2))) = f(f(f(x, c1), y), c2) = f(f(y, y), c2)$ $= f(y, c2) = x$. Thus the conclusion of the rule, $x = y$, is entailed as stated in the invariant for $=$. Soundness of all derived equation atoms is then immediate.

(Completeness) We prove completeness with a standard model-construction approach. We exhibit a first-order interpretation $\mathcal{I}$ that satisfies $\Sigma \cup \mathcal{A}$ and satisfies exactly equations φ that are derivable from Σ using rules $E \cup$ ACI. We start by defining an equivalence relation $\triangleq$ on terms based on the derivable equations: $t_1 \triangleq t_2$ if and only if $\Sigma \vdash_{E \,\cup\, \mathrm{ACI}} t_1 = t_2$. The rules in E imply that $\triangleq$ is an equivalence relation. We take the domain $D_{\mathcal{I}}$ of the interpretation to be the set $\{[t]_{\triangleq} \mid t \text{ a term}\}$ of all equivalence classes of terms under $\triangleq$.

Next, we define the interpretation under $\mathcal{I}[\![f]\!]$ of each function symbol f. We show the two argument case, which includes all ACI function symbols, but this definition is easily generalized to function symbols of any number of arguments. We define $I(f)$ on domain objects $[t_1]_{\triangleq}$ and $[t_2]_{\triangleq}$ to be $[f(t_1, t_2)]_{\triangleq}$. The inference rule (eq-congr) for the symbol f ensures that the defined output of f on two domain elements does not depend on which terms t_1 and t_2 are selected from the equivalence classes.

We next define the interpretation $\mathcal{I}$ for the predicate symbols. Like any first-order interpretation, $\mathcal{I}$ interprets equality as the identity function. We define $\mathcal{I}[\![\preccurlyeq_f]\!]$, for each ACI function-symbol f, to be true on domain objects $[t_1]_{\triangleq}$ and $[t_2]_{\triangleq}$ if and only if $\Sigma \vdash_{E \,\cup\, \mathrm{ACI}} t_1 \preccurlyeq_f t_2$. Again, the (eq-congr) rule for the predicate $\preccurlyeq_f$ ensures the truth value does not depend on the choice of t_1 and t_2.

A straightforward induction on the compositional structure of an arbitrary term t shows that the interpretation $\mathcal{I}[\![t]\!]$ of t under $\mathcal{I}$ is $[t]_{\triangleq}$. It then follows that $\mathcal{I}$ satisfies an equation $t_1 = t_2$ if and only if $t_1 = t_2$ is derivable from Σ using $E \cup$ ACI: i.e., we have $\Sigma \vdash_{E \,\cup\, \mathrm{ACI}} t_1 = t_2$ if and only if $[t_1]_{\triangleq} = [t_2]_{\triangleq}$, by definition, if and only if $\mathcal{I}[\![t_1]\!] = \mathcal{I}[\![t_2]\!]$, by our inductive lemma, and thus if and only if $\mathcal{I} \models t_1 = t_2$, as desired. It is then immediate that $\mathcal{I}$ satisfies Σ and all equations derived from Σ, and no others.

It remains to argue that $I \models \mathcal{A}$. We have that $\mathcal{I}$ satisfies the first-order axioms of equality because $\mathcal{I}[\![=]\!]$ is the identify function. Consider the associativity axiom for f, $\forall x \forall y \forall z f(f(x, y), z) = f(x, f(y, z))$. For any terms t_1, t_2, t_3, the ACI rule set justifies the following derivation: $t_1 \preccurlyeq_f f(t_1, t_2)$, $t_2 \preccurlyeq_f f(t_1, t_2)$, $f(t_1, t_2) \preccurlyeq_f f(f(t_1, t_2), t_3)$, $t_1 \preccurlyeq_f f(f(t_1, t_2), t_3)$, $t_2 \preccurlyeq_f f(f(t_1, t_2), t_3)$, $t_3 \preccurlyeq_f f(f(t_1, t_2), t_3)$, $f(t_2, t_3) \preccurlyeq_f f(f(t_1, t_2), t_3)$, $f(t_1, f(t_2, t_3)) \preccurlyeq_f f(f(t_1, t_2), t_3)$, $\ldots$, $f(f(t_1, t_2), t_3) \preccurlyeq_f f(t_1, f(t_2, t_3))$, $f(f(t_1, t_2), t_3) = f(t_1, f(t_2, t_3))$. Thus, $[f(f(t_1, t_2), t_3)]_{\triangleq}$ is the same as $[f(t_1, f(t_2, t_3))]_{\triangleq}$. Now considering arbitrary domain members $[t_1]_{\triangleq}$,$[t_2]_{\triangleq}$, and $[t_3]_{\triangleq}$, the definition of $\mathcal{I}$ implies that

$$\mathcal{I}[\![f]\!](\mathcal{I}[\![f]\!]([t_1]_{\triangleq}, [t_2]_{\triangleq}), [t_3]_{\triangleq}) = [f(f(t_1, t_2), t_3)]_{\triangleq},$$

and likewise $\mathcal{I}[\![f]\!]([t_1]_{\triangleq}, \mathcal{I}[\![f]\!]([t_2]_{\triangleq}, [t_3]_{\triangleq}))$ is $[f(t_1, f(t_2, t_3))]_{\triangleq}$, and thus every instance of associativity is satisfied by $\mathcal{I}$, and so is the associativity axiom. Similar arguments justify commutativity $(\forall x \forall y f(x, y) = f(y, x))$ and idempotence $(\forall x f(x, x) = x)$.

We have thus exhibited $\mathcal{I}$ satisfying $\Sigma \cup \mathcal{A}$ but not satisfying φ for any φ such that $\Sigma \not\Vdash_{E \cup \text{ACI}} \varphi$, as desired. Q.E.D.

5 Effective Decidability of the Congruence/ACI Rule Set

We next show that the rule set $E \cup \text{ACI}$ is local $(\vdash_{E \cup \text{ACI}} = \Vdash_{E \cup \text{ACI}})$, implying immediately that the inference relation $\vdash_{E \cup \text{ACI}}$ on ground equations is polynomial-time decidable. It then follows by Theorem 1 that entailment from ground premise sets in theory $\mathcal{A}$ is polynomial-time decidable.

The rule-set property of locality has been shown undecidable in general [11], but sub-classes of the local rule sets (inductively local rule sets [11] and bounded local rule sets [20]) have been shown to be automatically recognizable with procedures that fail to terminate on inputs representing local rule sets outside the sub-class. We have not been able to get these procedures to terminate on our rule set $E \cup \text{ACI}$, leaving unresolved(prior to this work) the question of locality for this rule set.

A semantic proof of locality can be constructed by showing that the partial model constructed by locally restricted inference can always be extended (embedded) to a full model. This technique was first presented in [10] and exploited in analyzing theory combinations in [24, 14]. The latter work can be used to handle the uninterpreted function symbols present in our setting, and Dedekind-MacNeille completion [17] can be used to show the semantic embeddability needed to construct the full model in the case where we have only one ACI function symbol [1]. However, we do not know of work extending Dedekind-MacNeille completion to multiple function symbols (i.e. multiple partial orders). A single completion step on one partial order destroys the ordering information in the others. Due to the difficulty we encountered constructing a semantic proof along these lines, we instead present a direct syntactic proof of locality. This proof sheds syntactic insight on why local restriction does not change the inference relation, and is of value even if the semantic proof can be repaired. For the remainder of this section, we use the shorthand K to abbreviate $E \cup \text{ACI}$ for readability.

One approach to proving locality of K is to prove semantic completeness of the restricted inference $\Vdash_K$. However, unlike in the pure congruence closure case $(\Vdash_E)$, the model-theoretic completeness proof of Theorem 1 does not adapt directly to the $\Upsilon(\Sigma, \varphi)$-restricted inference of $\Vdash_K$. The proof encounters difficulties in defining ACI function symbols. In the pure congruence-closure case, function symbols can be arbitrarily defined in cases that produce terms in equivalence classes outside of $\Upsilon(\Sigma, \varphi)$, producing a simple finite model, but the ACI theory restricts these definitions in complex ways. Due to the difficulties we encountered with a semantic approach, we instead prove locality with a syntactic analysis below.

For this proof, we first note that our definitions immediately imply that $\Sigma \vdash_K \varphi$ implies $\Sigma \Vdash_{K,\Upsilon} \varphi$ for some sufficiently large finite term set Υ containing $\Upsilon(\Sigma \cup \{\varphi\})$ and all terms appearing in the derivation underlying $\Sigma \vdash_K \varphi$. It follows that we can prove locality by proving that, for any ground premise set Σ and ground equation φ, and every finite subterm-closed term set Υ containing at least $\Upsilon(\Sigma \cup \{\varphi\})$, $\Sigma \Vdash_K \varphi$ if and only if $\Sigma \Vdash_{K,\Upsilon} \varphi$. $(*)$

We will prove $(*)$ by induction on the construction of Υ from $\Upsilon(\Sigma \cup \{\varphi\})$.

(Base case) $\Upsilon = \Upsilon(\Sigma \cup \{\varphi\})$. In this case, the claim $(*)$ holds trivially.

(Inductive case) We suppose the claim $(*)$ holds for a subterm-closed term set Υ containing $\Upsilon(\Sigma \cup \{\varphi\})$. Let α be an arbitrary term not in Υ such that every proper subterm of α is in Υ. We show that the claim $(*)$ holds with the set $\Upsilon' = \Upsilon \cup \{\alpha\}$ in place of Υ. Here, we will assume that α is $g(t_1, t_2)$ for some (ACI-labeled or not) g and terms t_1 and t_2; the other arity cases are similar but do not include the ACI possibility.

To show this, we consider all derivations from Σ using rules in K within Υ', characterizing every atomic formula that can appear in such a derivation. We show by a second, nested induction on the length of the derivation that the derived formula must fall into one of several classes. To define these classes of formula, we need some new notation. In order to use inference within Υ to characterize the terms t such that $t \preccurlyeq_g \alpha$ can be proven within Υ', if g is ACI we define the g-closure of the set $\{t_1, t_2\}$, written $\widehat{g}(\{t_1, t_2\})$, to be the closure of the set $\{t_1, t_2\}$ under the following two operations:

- For any term y in the set, add every z such that $\Sigma \Vdash_{K,\Upsilon} z \preccurlyeq_g y$.
- For any terms x and y in the set, add $g(x, y)$ if that term is in Υ.

In order to use inference within Υ to characterize those equations that are provable in Υ' between α and other terms e in Υ, we define $\langle\langle g(t_1, t_2) = e \rangle\rangle$ to hold if one of the following holds about inference with Υ:

- $\Sigma \Vdash_{K,\Upsilon} t_1 \preccurlyeq_g e \;\;\wedge\;\; \Sigma \Vdash_{K,\Upsilon} t_2 \preccurlyeq_g e \;\;\wedge\;\; e \in \widehat{g}(\{t_1, t_2\})$, and g is ACI, or
- $\Sigma \Vdash_{K,\Upsilon} x_1 = t_1 \;\;\wedge\;\; \Sigma \Vdash_{K,\Upsilon} x_2 = t_2 \;\;\wedge\;\; \Sigma \Vdash_{K,\Upsilon} g(x_1, x_2) = e$ for some terms x_1 and x_2.

These properties ensure that $\alpha = e$ will be inferred in Υ'. The first bullet characterizes equations by ACI inference and the second characterizes standard congruence inference. We will need the following lemmas about these definitions.

Lemma 1. *If $\langle\langle g(t_1, t_2) = e_1 \rangle\rangle$ then $\langle\langle g(t_1, t_2) = e_2 \rangle\rangle$ iff $\Sigma \Vdash_{K,\Upsilon} e_1 = e_2$.*

We are now ready to characterize the classes of formulas that can appear in derivations from Σ using rules K within Υ'. Every such formula must fall in one of the following classes[1]:

[1] In some cases class C2 will be contained in class C3; otherwise, all the classes are disjoint.

[C1] A formula derivable from Σ using rules K within Υ,

[C2] A reflexive $=$ or $\preccurlyeq_f$ formula about α, for any ACI f,

[C3] An atomic formula β, excluding $\preccurlyeq_g$, local to Υ' and mentioning α where the same atom is derivable within Υ with α replaced by some term $e \in \Upsilon$ (so that $\Sigma \Vdash_{K,\Upsilon} [e/\alpha]\beta$), where $\langle\!\langle g(t_1,t_2) = e \rangle\!\rangle$,

[C4] An atom $e \preccurlyeq_g \alpha$ for $e \in \widehat{g}(\{t_1,t_2\})$, or

[C5] An atom $\alpha \preccurlyeq_g e$ for which both $\Sigma \Vdash_{K,\Upsilon} t_1 \preccurlyeq_g e$ and $\Sigma \Vdash_{K,\Upsilon} t_2 \preccurlyeq_g e$.

Classes C4 and C5 are considered empty if g is not labeled ACI. These classes cover all formulas that can appear in derivations under $\Vdash_{K,\Upsilon'}$. This can be verified by induction on the length of the derivation, checking that each inference rule preserves the claim. We discuss the critical (ACI-antisym) inference rule here as an example. The rule antecedents are $x \preccurlyeq_f y$ and $y \preccurlyeq_f x$. We must show that the consequent $x = y$ falls in one of the five given classes. There are several cases to consider (up to symmetries):

1. $\alpha \notin \{x, y\}$. Both antecedents and then the conclusion must be in class C1.
2. $x = y = \alpha$. The conclusion is in class C2.
3. $x \neq y = \alpha$, $f \neq g$. Then $x \in \Upsilon$. Both antecedents must be in class C3. So there must be $e_1 \in \Upsilon$ such that $\langle\!\langle g(t_1,t_2) = e_1 \rangle\!\rangle$ and $\Sigma \Vdash_{K,\Upsilon} x \preccurlyeq_f e_1$. Likewise there must be $e_2 \in \Upsilon$ such that $\langle\!\langle g(t_1,t_2) = e_2 \rangle\!\rangle$ and $\Sigma \Vdash_{K,\Upsilon} e_2 \preccurlyeq_f x$. It follows from Lemma 1 that $\Sigma \Vdash_{K,\Upsilon} e_1 = e_2$. It then follows that $\Sigma \Vdash_{K,\Upsilon} x \preccurlyeq_f e_2$ using that (eq-congr) is in K, and that $\Sigma \Vdash_{K,\Upsilon} e2 = x$ using that (aci-antisym) is in K. Lemma 1 then implies $\langle\!\langle g(t_1,t_2) = x \rangle\!\rangle$. But then since $\Sigma \Vdash_{K,\Upsilon} x = x$, the rule conclusion $x = \alpha$ satisfies the C3 class invariant.
4. $x \neq y = \alpha$, $f = g$. Then $x \in \Upsilon$. Antecedent $x \preccurlyeq_g \alpha$ is in class C4, and antecedent $\alpha \preccurlyeq_g x$ is in class C5. We need to show the conclusion $x = \alpha$ is in class C3. To do so, we exhibit $e \in \Upsilon$ such that $\Sigma \Vdash_{K,\Upsilon} x = e$ and $\langle\!\langle g(t_1,t_2) = e \rangle\!\rangle$. From the antecedents, using the induction hypothesis about classes C4 and C5, we have $\Sigma \Vdash_{K,\Upsilon} t_1 \preccurlyeq_g x$ and $\Sigma \Vdash_{K,\Upsilon} t_2 \preccurlyeq_g x$, as well as $x \in \widehat{g}(\{t_1,t_2\})$. Then by definition, $\langle\!\langle g(t_1,t_2) = x \rangle\!\rangle$, so x is the desired e.

These cases together demonstrate that any instance of (ACI-antisym) in a derivation satisfying the claim up to that point will extend that derivation with a formula that also satisfies the claim. Together with similar analyses of all the other rules [12], we conclude that every formula in a derivation from Σ using K within Υ' falls in one of these five classes.

Since every formula in these classes either mentions α (classes C2 to C5) or is derivable within Υ (class C1), we have shown that for any φ local to Υ, $\Sigma \Vdash_{K,\Upsilon} \varphi$ if and only if $\Sigma \Vdash_{K,\Upsilon'} \varphi$, which together with our (outer) induction hypothesis implies that $(*)$ holds for Υ', as desired. We have thus shown the following theorem.

Theorem 2. *The rule set* $E \cup \mathrm{ACI}$ *is local. The inference relation* $\vdash_{E \cup \mathrm{ACI}}$ *is polynomial-time decidable.*

The techniques discussed in [20] provide a straightforward $\Theta(n^3)$ procedure via the locally restricted inference process.

6 Integrating ACI Functions into Other Rule Sets

We now turn our attention to general equational rule sets for which we may wish to designate some of the function symbols as ACI. We suppose an arbitrary equational rule set R is known to be local and consider the question of whether $R \cup \mathrm{ACI}$, with some function symbol f labeled to be ACI, remains local.

In fact, it is easy to see that the addition of ACI in $R \cup \mathrm{ACI}$ will not in general preserve locality. Rules in R may contain specific nested applications of the ACI-labeled function symbol f, and if the local expressions contain ACI-equivalent terms that don't match that specific nesting, the rules involved will not be part of derivations. For example, the single-rule local rule set $\{P(f(x, f(y, z))) \;\rightarrow\; Q(z)\}$ can draw no conclusions with local derivations on premise set $P(f(f(a, b), c))$, but upon expanding the rule set with the ACI rules, can conclude $Q(c)$. Rules like this one are using the function symbol f in a manner somehow inconsistent with the assumption that f is ACI.

As an example, consider the following local rule set for binary intersections and unions [11]:

$$\begin{array}{lll}
\rightarrow\; x \subseteq x & x \subseteq y \;\wedge\; y \subseteq z \;\rightarrow\; x \subseteq z & \\
\rightarrow\; y \subseteq x \cup y & x \subseteq z \;\wedge\; y \subseteq z \;\rightarrow\; x \cup y \subseteq z & \rightarrow\; x \subseteq x \cup y \\
\rightarrow\; x \cap y \subseteq y & z \subseteq x \;\wedge\; z \subseteq y \;\rightarrow\; z \subseteq x \cap y & \rightarrow\; x \cap y \subseteq x
\end{array}$$

The techniques in this section are designed to enable the addition of complete inference from the ACI properties for intersection and union to rule sets like this without losing the locality property that ensures efficient inference, by reformulating the ACI function symbols (in this case $\cup$ and $\cap$) as being applied to single arguments that are tuples of terms, where ACI properties are managed by equating ACI-equivalent tuples. What we show here is that a rule set that is local before adding these ACI properties on tuple argument, will remain so after this addition is made. The remainder of this section formalizes this idea. As an introduction, consider the tuple formulation of the rule set just presented:

$$\begin{array}{lll}
\rightarrow\; x \subseteq x & x \subseteq y \;\wedge\; y \subseteq z \;\rightarrow\; x \subseteq z & \forall x \in \lambda, x \subseteq z \;\rightarrow\; \bigcup\lambda \subseteq z \\
x \in \lambda \;\rightarrow\; x \subseteq \bigcup\lambda & x \in \lambda \;\rightarrow\; \bigcap\lambda \subseteq x & \forall x \in \lambda, z \subseteq x \;\rightarrow\; z \subseteq \bigcap\lambda
\end{array}$$

Here, the λ rule variable represents a tuple of terms, with implicitly ACI function symbols $\bigcup$ and $\bigcap$ being applied to such tuples. Rule antecedents involving $\in$ with tuples are abbreviations as discussed below. The rule set is restricted from accessing the tuple structure in any other way than with the types of $\in$ antecedents shown here and formalized below. What we prove here, with some substantial difficulty, is that if the rule set is local before this transformation, without the ACI properties on its tuples, it will remain local when the ACI properties are enforced.

6.1 Every/Some ACI Rule Sets

Here, we propose to limit the ways that the rule set R can mention the ACI function symbols to ensure that ACI inferences do not interact badly with the rules. Our proposal below supports two interactions between ACI terms and rules: testing that every (nested) argument to an ACI function symbol satisfies a predicate, and forcing a variable to represent an arbitrary (nested) argument to the ACI symbol. In addition, we note that any number of ACI function symbols can be represented if the theory supports a single ACI "tupling" function symbol. So we limit consideration to implementing a single ACI function symbol for tupling.

Kinds. In order to enforce separation between the ACI inference and the arbitrary rule set R, we introduce the concept of "kind" into the representation. For simplicity here, without loss of generality, we assume there are just two kinds: basic-term ($\mathcal{B}$) and tuple ($\mathcal{T}$). Every function symbol and predicate symbol has a signature over the kinds, so that every well-formed term has a syntactic kind. Applications of functions and predicates to expressions of the wrong kind are considered ill-formed. Every variable in a rule also has a kind and rule instances can only be formed by substituting variables with terms of the matching kind. The equational rules E, contained in any equational R, are assumed duplicated for each kind, with (eq-congr) present for each signature of function symbol or predicate symbol. Throughout this section, t, x, y, and z are basic-term variables, and λ is a tuple variable.

Tuples. The only expressions of tuple kind are formed from two function symbols: $\langle t\rangle$ coerces basic-term t into a (singleton) tuple, and $(\lambda_1 \cdot \lambda_2)$ combines two tuples. The function symbol $(\cdot)$ is the only function symbol labeled ACI. Every predicate symbol that operates on tuple arguments is designated hidden, i.e. cannot appear in premise sets, and besides equality on tuples the only such predicate symbols are introduced by the rules for ACI inference ($\preccurlyeq_{(\cdot)}$, $Z_\exists$, and $Z_{\forall P}$, introduced below). All other predicate symbols are not hidden. The only function symbols that have tuple arguments have exactly one argument—it is these function symbols that are *implicitly ACI* by accepting their arguments in tuple form. We write $x \in \lambda$ by abuse of notation to mean that x is a maximal basic-term subexpression of the tuple λ.

Rule set restrictions. We restrict the rule set R to contain only basic-term variables and expressions, except for the equational rules in E for tuples, mentioned above, and for the specific enrichment to use tuples that we propose next. We allow tuple variables in rules in the following forms:

- as sub-expressions of a basic-term (i.e. as an argument tuple), or
- in "every" or "some" antecedents, as detailed below.

Moreover, we require each tuple variable in a rule to occur at least once as a sub-expression of a basic-term.

Some. We allow rule antecedents of the form $x \in \lambda$. The basic-term variable x may be used elsewhere in the rule; this antecedent can be thought of as binding x for use elsewhere in the rule. Semantically, the "some" antecedent abbreviates $\exists z\, s = (\langle x\rangle \cdot z)$.

Every. We allow rule antecedents of the form $\forall x{\in}\lambda\, P(x)$ where P is any one-argument predicate symbol on basic-terms. For notational simplicity here, we assume P has no other arguments, but the extension to allow arbitrary other basic-term arguments not mentioning x is straightforward. The basic-term variable x must not appear in the rule outside of this antecedent. We think of the $\forall$ as binding x locally to this antecedent. Semantically, the "every" antecedent abbreviates $\forall x\, \exists z\, \lambda = (\langle x \rangle \cdot z) \; \rightarrow \; P(x)$.

Syntactically, we eliminate "every" and "some" antecedents as follows. Introduce a new hidden predicate $Z_\exists$ of signature (basic-term $\times$ tuple) for the implementation of "some". For each predicate P appearing in any "every" antecedent, introduce a new hidden predicate $Z_{\forall P}$ on tuples, i.e., of signature (tuple). Replace each antecedent $x \in s$ with the atom $Z_\exists(x, \lambda)$, and each antecedent $\forall x {\in} \lambda\, P(x)$ with the atom $Z_{\forall P}(\lambda)$. Then, the following rules are added to R; it is straightforward to verify that these rules are sound under the semantics just given for "every" and "some" antecedents:

$$\begin{array}{rlrl}
(Z_\exists\text{-sub1}) & Z_\exists(x, \lambda_1) \;\rightarrow\; Z_\exists(x, (\lambda_1 \cdot \lambda_2)) & (Z_\exists\text{-init}) & \rightarrow\; Z_\exists(x, \langle x \rangle) \\
(Z_\exists\text{-sub2}) & Z_\exists(x, \lambda_2) \;\rightarrow\; Z_\exists(x, (\lambda_1 \cdot \lambda_2)) & (Z_{\forall P}\text{-init}) & P(x) \;\rightarrow\; Z_{\forall P}(\langle x \rangle) \\
(Z_\forall\_\text{combine}) & Z_{\forall P}(\lambda_1) \;\wedge\; Z_{\forall P}(\lambda_2) \;\rightarrow\; Z_{\forall P}((\lambda_1 \cdot \lambda_2)) & &
\end{array}$$

where the $Z_{\forall P}$ rules are present once for each predicate P appearing in an "every" antecedent.

Definition 6. *An* every/some rule set *is an equational rule set R for the kinds basic-term and tuple in which every tuple sub-expression is a tuple variable that occurs within a basic-term expression, and may also occur within "every" antecedents and/or "some" antecedents. The only predicate and function symbols involving the tuple kind are the tuple-constructors $\langle\rangle$ and $(\cdot)$, implicitly ACI function symbols (signature tuple $\rightarrow$ basic-term), and those abbreviated by every/some. Each such rule set abbreviates a rule set with no every/some antecedents via the transformation just described.*

For our purposes here, it is important to note that an every/some rule set does not yet have any ACI properties enforced. The rule set will not be able to infer the equivalence of tuples that are ACI variants of each other, and so function applications of implicitly ACI function symbols (applied to tuples) will also not benefit from the ACI properties. All inference on such terms will depend on the argument order and multiplicity. What we wish to show is that we can *add* the ACI properties while preserving the locality of the rule set.

6.2 Adding ACI to Every/Some Rule Sets Preserves Locality

Here we prove that we can add the theory ACI for tuples to any local every/some rule set R, preserving locality and thus polynomial-time decidability. The proof is rather technical and carefully constructed. We provide the key major structure here and refer to our website supplement [12] for many technical verifications underlying the proof.

Definition 7. *For any set of rules R, subterm-closed set of terms Υ, premise set Σ local to Υ, and positive integer k, and we write $C_k(\Sigma, R, \Upsilon)$ for the set of all formulas derivable from Σ using R with a derivation local to Υ of length at* *most k. We write $B_k(\Sigma, R, \Upsilon)$ for the non-hidden subset of $C_k(\Sigma, R, \Upsilon)$. Finally, we write $C(\Sigma, R, \Upsilon)$ for $\bigcup_{k=1}^{\infty} C_k(\Sigma, R, \Upsilon)$, and likewise $B(\Sigma, R, \Upsilon)$.*

We will also need some notation and invariants characterizing exactly when tuples become related by inference.

Definition 8. *Given two sets of basic-term expressions S_1 and S_2 and a set of formulas Γ, we write $(S_1 \subseteq_{eq} S_2) \in \Gamma$ if there are equations in Γ to make S_1 a subset of S_2, i.e. if for every $x \in S_1$ there is $y \in S_2$ such that $x = y \in \Gamma$. Extending this notation to tuple expressions λ_1 and λ_2, we write $(\lambda_1 \subseteq_{eq} \lambda_2) \in \Gamma$ to abbreviate $(\{x \mid x \in \lambda_1\} \subseteq_{eq} \{y \mid y \in \lambda_2\}) \in \Gamma$. Finally, we write $(\lambda_1 =_{eq} \lambda_2) \in \Gamma$ if both $(\lambda_1 \subseteq_{eq} \lambda_2) \in \Gamma$ and $(\lambda_2 \subseteq_{eq} \lambda_1) \in \Gamma$.*

Using this new notation, the locality of a rule set R implies for instance that $B(\Sigma, R, \Upsilon) =_{\Upsilon} B(\Sigma, R, \Upsilon')$ for every subterm-closed Υ, premise set Σ local to Υ, and Υ' containing Υ. Here, we introduce the notation $=_{\Upsilon}$ to represent equality between two sets of formulas after restricting each set to the formulas local to Υ. We likewise define $\subseteq_{\Upsilon}$.

We can now state the key invariants regarding ACI inference on tuples. These invariants are stated for any every/some rule set R, subterm-closed Υ, premise set Σ local to Υ and positive integer k. We temporarily abbreviate the set of consequences $C_k(\Sigma, R \cup \text{ACI}, \Upsilon)$ as Γ_k and $C(\Sigma, R \cup \text{ACI}, \Upsilon)$ as Γ_∞:

1. (ACI-1) $\lambda_1 = \lambda_2 \in \Gamma_k$ implies $(\lambda_1 =_{\text{eq}} \lambda_2) \in \Gamma_k$, and conversely,
   $(\lambda_1 =_{\text{eq}} \lambda_2) \in \Gamma_k$ implies $\lambda_1 = \lambda_2 \in \Gamma_\infty$.
2. (ACI-2) $\lambda_1 \preccurlyeq_{(\cdot)} \lambda_2 \in \Gamma_k$ implies $(\lambda_1 \subseteq_{\text{eq}} \lambda_2) \in \Gamma_k$, and conversely,
   $(\lambda_1 \subseteq_{\text{eq}} \lambda_2) \in \Gamma_k$ implies $\lambda_1 \preccurlyeq_{(\cdot)} \lambda_2 \in \Gamma_\infty$.
3. (ACI-3) $Z_{\forall P}(\lambda) \in \Gamma_k$ implies $P(t) \in \Gamma_k$ for every $t \in \lambda$, and conversely,
   $P(t) \in \Gamma_k$ for every $t \in \lambda$ implies $Z_{\forall P}(\lambda) \in \Gamma_\infty$.
4. (ACI-4) $Z_{\exists}(t', \lambda) \in \Gamma_k$ implies $t = t' \in \Gamma_k$ for some $t \in \lambda$, and conversely,
   $t = t' \in \Gamma_k$ for $t \in \lambda$ implies $Z_{\exists}(t', \lambda) \in \Gamma_\infty$.

These invariants are easily demonstrated by induction on the length of derivations for the forward directions, showing that each rule preserves these invariants, and induction on the tuple structures for the converses. The same invariants, dropping ACI-2, can be shown replacing $R \cup \text{ACI}$ by R, but in stating ACI-1 using a narrower definition of $(\lambda_1 \subseteq_{\text{eq}} \lambda_2)$ that requires directly matching tuple structure from λ_1 and λ_2 (for lack of ACI rules).

We are now ready to develop the main theorem of this section. We restrict consideration to a particular local every/some rule set R, arbitrary subterm-closed term set Υ, and premise set Σ. We first consider the effect on inference from Σ using $R \cup \text{ACI}$ when we add a single basic-term to Υ. It is in this case that the locality of the base rule set R comes into play.

Lemma 2. *(Basic-term Extension) For any basic-term expression α, where all proper subexpressions of α are in Υ, $B(\Sigma, R \cup \text{ACI}, \Upsilon \cup \{\alpha\}) =_{\Upsilon} B(\Sigma, R \cup \text{ACI}, \Upsilon)$.*

Proof. The backward containment is immediate because $B(,,\Upsilon)$ is clearly monotone in Υ, as every derivation local to Υ is local to any superset of Υ.

To show the forward containment, the central proof idea is to observe that if $R \cup \mathrm{ACI}$ has a locality violation, then we can construct such a locality violation for R on a premise set that contains Σ along with some additional premises (those non-hidden formulas that could have been derived by ACI if ACI were in use). Since the property of rule-set locality is a property that applies to all premise sets, the larger premise set cannot generate a locality violation under R (i.e. there can be no R-derivable fact from the larger premise set that is not derivable by locally restricted inference from that premise set).

To formalize this idea, we introduce an enriched premise set $\Sigma' = B(\Sigma, R \cup \mathrm{ACI}, \Upsilon)$ and consider the consequences under inference from Σ' using rule sets $R \cup \mathrm{ACI}$, within the enlarged term set $\Upsilon \cup \{\alpha\}$. The key observation, discussed next, is that any derivation of a new consequence within Υ using $R \cup \mathrm{ACI}$ will have to start by using R alone to get the first new consequence within Υ. But R alone cannot get new consequences within Υ as R is local.

We now show the desired forward containment, but for Σ': $B(\Sigma', R \cup \mathrm{ACI}, \Upsilon \cup \{\alpha\}) \subseteq_\Upsilon B(\Sigma', R \cup \mathrm{ACI}, \Upsilon)$. To show this, suppose for contradiction that the desired containment is false, so that there must be some formula φ local to Υ in $B(\Sigma', R \cup \mathrm{ACI}, \Upsilon \cup \{\alpha\})$ but not in $B(\Sigma', R \cup \mathrm{ACI}, \Upsilon)$. In this context, we refer to formulas local to Υ in $B(\Sigma', R \cup \mathrm{ACI}, \Upsilon \cup \{\alpha\})$ but not in $B(\Sigma', R \cup \mathrm{ACI}, \Upsilon)$ as *newly derivable formulas*; φ is a newly derivable formula. Also here we refer to formulas local to Υ as *local*; φ is a newly derivable local formula.

Consider any derivation of φ from Σ' using $R \cup \mathrm{ACI}$ within $\Upsilon \cup \{\alpha\}$. We refer to formulas in the derivation as *earlier* or *later* in the derivation according to their index in the sequence (with lower index corresponding to earlier). Consider the earliest newly derivable local formula β in the derivation, which occurs at latest at φ. Then every non-hidden local formula in the (prefix) derivation of β is in $B(\Sigma', R \cup \mathrm{ACI}, \Upsilon)$ and thus in Σ', as Σ' is already closed under $R \cup \mathrm{ACI}$ within Υ. We show that there is a derivation of β from Σ' using only rules in R, which contradicts the locality of R (which can't derive new formulas local to Υ by using α), to conclude the proof, as argued below.

Observe no instance of a rule in ACI within $\Upsilon \cup \{\alpha\}$ can mention α, since every expression in the ACI rules is a tuple expression, and α is not a subexpression of any tuple expression in $\Upsilon \cup \{\alpha\}$. Thus every formula in any ACI rule instance used in the derivation of β is local to Υ.

We now show that β is in $B(\Sigma', R, \Upsilon \cup \{\alpha\})$. We show that there is no formula in the derivation of β being considered that can be justified only by an ACI rule, from the derivation to that point and the premises Σ'. To show this, we show that there can be no earliest such formula η. Supposing, for contradiction, there is such η, then η is local to Υ, as just argued for ACI rule instances in the derivation of β, and cannot be a member of Σ' (or ACI rules would not be the only justification for η). Formula η cannot be a $\preccurlyeq_{(\cdot)}$ formula or it could only be justified by another ACI rule and thus would not be the earliest choice. Formula η cannot be a $Z_\exists$ or $Z_{\forall P}$ formula or it would not be a possible consequent of an ACI rule. So, η must not be hidden. Thus, η is a non-hidden local formula in the derivation of β—so, by our choice of β, we have $\eta \in \Sigma'$, the premise set, contradicting the choice of η as requiring justification by an ACI rule. We conclude from this contradiction that every formula in the derivation of β is either in Σ' or can be justified by a rule in R, so β is in $B(\Sigma', R, \Upsilon \cup \{\alpha\})$. Since β is newly derivable, $\beta \notin B(\Sigma', R \cup \mathrm{ACI}, \Upsilon)$ and thus $\beta \notin B(\Sigma', R, \Upsilon)$. This violates the locality of R, completing our proof that $B(\Sigma', R \cup \mathrm{ACI}, \Upsilon \cup \{\alpha\}) \subseteq_\Upsilon B(\Sigma', R \cup \mathrm{ACI}, \Upsilon)$.

But Σ' can be replaced by Σ in this claim, as the inference under $R \cup \mathrm{ACI}$ within Υ that constructs Σ' from Σ is already included in both $B(,,)$ closures being considered. This gives us the desired containment to conclude our proof of the lemma. Formally,

$$\begin{aligned} B(\Sigma', R \cup \mathrm{ACI}, \Upsilon \cup \{\alpha\}) &= B(B(\Sigma, R \cup \mathrm{ACI}, \Upsilon), R \cup \mathrm{ACI}, \Upsilon \cup \{\alpha\}) \\ &= B(\Sigma, R \cup \mathrm{ACI}, \Upsilon \cup \{\alpha\}) \end{aligned}$$

and likewise

$$\begin{aligned} B(\Sigma', R \cup \mathrm{ACI}, \Upsilon) &= B(B(\Sigma, R \cup \mathrm{ACI}, \Upsilon), R \cup \mathrm{ACI}, \Upsilon) \\ &= B(\Sigma, R \cup \mathrm{ACI}, \Upsilon). \end{aligned}$$

Q.E.D. (Basic-term Extension Lemma)

A separate induction on derivation length is needed to handle extensions of Υ by new tuple expressions, leveraging the invariants stated above on tuple inference ((ACI-1) to (ACI-4)).

Lemma 3. *(Tuple Extension) For any tuple expression α, where all proper subexpressions of α are in Υ, $B(\Sigma, R \cup \mathrm{ACI}, \Upsilon \cup \{\alpha\}) =_\Upsilon B(\Sigma, R \cup \mathrm{ACI}, \Upsilon)$.*

Proof. The backward containment is again immediate. We prove the forward containment by induction on k to show, for all k, $B_k(\Sigma, R \cup \mathrm{ACI}, \Upsilon \cup \{\alpha\}) \subseteq_\Upsilon B(\Sigma, R \cup \mathrm{ACI}, \Upsilon)$. Please see our website supplement [12] for technical verification that each inference rule preserves this property of derivations, leveraging the tuple invariants (ACI-1) to (ACI-4). Here we discuss in detail the argument for one example inference rule chosen to illustrates all the key ideas.

Suppose for induction that $B_{k-1}(\Sigma, R \cup \mathrm{ACI}, \Upsilon \cup \{\alpha\}) \subseteq_\Upsilon B(\Sigma, R \cup \mathrm{ACI}, \Upsilon)$. Consider a k-step derivation ending in a non-hidden formula φ justified by an instance of an inference rule from the basic rule set R, but not an equational rule from E. We show that the conclusion of this derivation is a member of $B(\Sigma, R \cup \mathrm{ACI}, \Upsilon)$. We will show (a) every antecedent formula of the inference instance used is in $C(\Sigma, R \cup \mathrm{ACI}, \Upsilon)$, and (b) the consequent φ is local to Υ and not hidden. From these two statements we can conclude that $\varphi \in B(\Sigma, R \cup \mathrm{ACI}, \Upsilon)$ as desired. To see (b), observe that φ is not hidden, so it cannot mention the new tuple α and so, being local to $\Upsilon \cup \{\alpha\}$ must also be local to Υ.

Then, to argue for (a), consider an arbitrary antecedent formula β of the rule instance. We have $\beta \in C_{k-1}(\Sigma, R \cup \mathrm{ACI}, \Upsilon \cup \{\alpha\})$ since β appears in a k-step derivation as an antecedent. Please refer to our website supplement [12] for details of the induction proof. Q.E.D. (Tuple Extension Lemma)

By a simple induction on the construction of subterm-closed set Υ' containing Υ, these two lemmas directly imply that $R \cup \mathrm{ACI}$ is local.

Theorem 3. *For any local every/some rule set R, the rule set $R \cup \mathrm{ACI}$ is local.*

It follows that $R \cup \mathrm{ACI}$ defines a polynomial-time decidable inference relation.

7 Conclusion

We have shown a local rule set provably providing sound and complete congruence closure with ACI function symbols. We also provide a detailed example integrating ACI inference into other local rule sets preserving locality.

References

1. Anonymous Reviewer: Personal email communication (March 2013)
2. Bachmair, L., Ramakrishnan, I., Tiwari, A., Vigneron, L.: Congruence closure modulo associativity and commutativity. FroCoS pp. 245–259 (2000)
3. Bachmair, L., Tiwari, A.: Abstract congruence closure and specializations. In: Proceedings of the 17th International Conference on Automated Deduction. pp. 64–78. CADE-17, Springer-Verlag, London, UK, UK (2000)
4. Bachmair, L., Tiwari, A., Vigneron, L.: Abstract congruence closure. J. Autom. Reason. 31(2), 129–168 (Dec 2003)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions. SpringerVerlag (2004)
6. Bloniarz, P., Hunt III, H., Rosenkrantz, D.: Algebraic structures with hard equivalence and minimization problems. J. ACM 31(4), 879–904 (1984)
7. Burris, S.: Polynomial time uniform word problems. Mathematical Logic Quarterly 41(2), 173–182 (1995)
8. Conchon, S., Contejean, E., Kanig, J., Lescuyer, S.: CC (X): Semantic combination of congruence closure with solvable theories. Elec. Notes in Theor. Comp. Science 198(2), 51–69 (2008)
9. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. Tools and Algorithms for the Construction and Analysis of Systems pp. 337–340 (2008)
10. Ganzinger, H.: Relating semantic and proof-theoretic concepts for polynomial time decidability of uniform word problems. In: Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on. pp. 81–90. IEEE (2001)
11. Givan, R., McAllester, D.: Polynomial-time computation via local inference relations. ACM Transactions on Computational Logic 3, 521–541 (Oct 2002)
12. Hu, T., Givan, R.: Additional proof details for ADDCT 2013 paper submisison. https://engineering.purdue.edu/∼relation/addct13.html (2013)
13. Hunt III, H., Rosenkrantz, D., Bloniarz, P.: On the computational complexity of algebra on lattices. SIAM Journal on Computing 16(1), 129–148 (1987)
14. Ihlemann, C., Sofronie-Stokkermans, V.: On hierarchical reasoning in combinations of theories. In: Automated Reasoning, pp. 30–45. Springer (2010)
15. Kapur, D.: Shostak's congruence closure as completion. In: Proceedings of the 8th International Conference on Rewriting Techniques and Applications. pp. 23–37. RTA '97, Springer-Verlag, London, UK, UK (1997)
16. Kozen, D.: Complexity of finitely presented algebras. In: Proceedings of the ninth annual ACM symposium on Theory of computing. pp. 164–177. STOC '77, ACM, New York, NY, USA (1977)
17. MacNeille, H.M.: Partially ordered sets. Trans. Amer. Math. Soc 42(3), 416–460 (1937)
18. Mayr, E., Meyer, A.: The complexity of the word problems for commutative semigroups and polynomial ideals. Advances in mathematics 46(3), 305–329 (1982)
19. McAllester, D.: Ontic: A Knowledge Representation System for Mathematics. MIT Press, Cambridge, MA (1989)
20. McAllester, D.: Automatic recognition of tractability in inference relations. Journal of the ACM 40(2), 284–303 (April 1993)
21. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. Journal of the ACM 27(2), 356–364 (April 1980)
22. Rehof, J., Mogensen, T.: Tractable constraints in finite semilattices. Science of Computer Programming 35(2), 191–221 (1999)
23. Shostak, R.E.: An algorithm for reasoning about equality. Communications of the ACM 21(7), 583–585 (Jul 1978), `http://doi.acm.org/10.1145/359545.359570`
24. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Automated Deduction–CADE-20, pp. 219–234. Springer (2005)

Automatic Decidability for Theories with Counting Operators

(Presentation-only paper)

Elena Tushkanova[1,2], Christophe Ringeissen[1], Alain Giorgetti[1,2], and Olga Kouchnarenko[1,2]

[1] Inria, Villers-les-Nancy, F-54600, France
[2] FEMTO-ST Institute (UMR 6174), University of Franche-Comté, Besançon, F-25030, France

The full paper will appear in the proceedings of the 24th International Conference on Rewriting Techniques and Applications (RTA 2013), published as LIPIcs proceedings, `http://www.dagstuhl.de/en/publications/lipics`

1 Introduction

Decision procedures for satisfiability modulo background theories of classical datatypes are at the core of many state-of-the-art verification tools. Designing and implementing these satisfiability procedures remains a very hard task. To help the researcher with this time-consuming task, an important approach based on rewriting has been investigated in the last decade [2, 1]. The rewriting-based approach allows building satisfiability procedures in a flexible way by using a general calculus for automated deduction, namely the paramodulation calculus [9] (also called superposition calculus). The paramodulation calculus is a refutation-complete inference system at the core of all equational theorem provers. In general this calculus provides a semi-decision procedure that halts on unsatisfiable inputs by generating an empty clause, but may not terminate on satisfiable ones. However, it also terminates on satisfiable inputs for some theories axiomatising standard datatypes such as arrays, lists, etc, and thus provides a decision procedure for these theories. A classical termination proof consists in considering the finitely many cases of inputs made of the (finitely many) axioms and any set of ground flat literals. This proof can be done by hand, by analysing the finitely many forms of clauses generated by saturation, but the process is tedious and error-prone. To simplify this process, a schematic paramodulation calculus has been developed [5] to build the schematic form of the saturations. It can be seen as an abstraction of the paramodulation calculus: If it halts on one given abstract input, then the paramodulation calculus halts for all the corresponding concrete inputs. More generally, schematic paramodulation is a fundamental tool to check important properties related to decidability and combinability [6].

To ensure efficiency, it is very useful to have built-in axioms in the calculus, and so to design paramodulation calculi modulo theories. This is particularly important for arithmetic fragments due to the ubiquity of arithmetics in applications of formal methods. For instance, paramodulation calculi have been developed for Abelian Groups [4, 7] and Integer Offsets [8]. In [8], the termination of paramodulation modulo Integer Offsets is proved manually. Therefore, there is an obvious need for a method to automatically prove that an input theory admits a decision procedure based on paramodulation modulo Integer Offsets.

In this paper, we introduce theoretical underpinnings that allow us to automatically prove the termination of paramodulation modulo Integer Offsets. To this aim, we design a new schematic paramodulation calculus to describe saturations modulo Integer Offsets. Our approach requires a new form of schematization to cope with arithmetic expressions. The interest of schematic paramodulation relies on a correspondence between a derivation using (concrete) paramodulation and a derivation using schematic paramodulation: Roughly speaking, the set of derivations obtained by schematic paramodulation over-approximates the set of derivations obtained by (concrete) paramodulation.

Our approach has been developed and validated thanks to a proof system [10] implemented in the rewriting logic-based environment Maude.

2 Paramodulation Calculus

As in [10] we consider only unitary clauses, i.e. clauses composed of at most one literal. The *Unitary Paramodulation Calculus*, denoted by $\mathcal{UPC}$ [10] corresponds to the standard paramodulation calculus restricted to the case of unit clauses.

The paramodulation-based calculus $\mathcal{UPC}_I$ defined in [8] adapts the paramodulation calculus $\mathcal{UPC}$ to the theory of Integer Offsets, so that it can serve as a basis for the design of decision procedures for Integer Offsets extensions. Technically, the axioms of the theory of Integer Offsets are directly integrated in the simplification rules of $\mathcal{UPC}_I$. The theory of Integer Offsets is axiomatized by the set of axioms $\{\forall X.\ \mathsf{s}(X) \neq 0,\ \forall X, Y.\ \mathsf{s}(X) = \mathsf{s}(Y) \Rightarrow X = Y,\ \forall X.\ X \neq \mathsf{s}^n(X) \text{ for all } n \geq 1\}$ over the signature $\Sigma_I := \{0 : \textsc{int}, \mathsf{s} : \textsc{int} \to \textsc{int}\}$. Compared to [3], our theory of Integer Offsets does not consider the predecessor function. Following [8, Section 5], a possible Integer Offsets extension is the theory LLI of lists with length whose signature is $\Sigma_{LLI} = \{\mathsf{car} : \textsc{lists} \to \textsc{elem}, \mathsf{cdr} : \textsc{lists} \to \textsc{lists}, \mathsf{cons} : \textsc{elem} \times \textsc{lists} \to \textsc{lists}, \mathsf{len} : \textsc{lists} \to \textsc{int}, \mathsf{nil} :\to \textsc{lists}, 0 :\to \textsc{int}, \mathsf{s} : \textsc{int} \to \textsc{int}\}$ and whose set of axioms $Ax(LLI)$ is $\{\mathsf{car}(\mathsf{cons}(X, Y)) = X$, $\mathsf{cdr}(\mathsf{cons}(X, Y)) = Y$, $\mathsf{len}(\mathsf{cons}(X, Y)) = \mathsf{s}(\mathsf{len}(Y))$, $\mathsf{cons}(X, Y) \neq \mathsf{nil}$, $\mathsf{len}(\mathsf{nil}) = 0\}$.

3 Schematic Paramodulation

The *Schematic Unitary Paramodulation Calculus* $\mathcal{SUPC}$ is an abstraction of $\mathcal{UPC}$. Indeed, any concrete saturation computed by $\mathcal{UPC}$ can be viewed as an instance of an abstract saturation computed by $\mathcal{SUPC}$ [6, Theorem 2]. Hence, if $\mathcal{SUPC}$ halts on one given abstract input, then $\mathcal{UPC}$ halts for all the corresponding concrete inputs. More generally, $\mathcal{SUPC}$ is an automated tool to check properties of $\mathcal{UPC}$ such as termination, stable infiniteness and deduction completeness [6].

$\mathcal{SUPC}$ is almost identical to $\mathcal{UPC}$, except that literals are constrained by conjunctions of atomic constraints of the form $const(x)$ which restricts the instantiation of the variable x by only constants. An implementation of *Paramodulation* and *Schematic Paramodulation* calculi $\mathcal{UPC}$ and $\mathcal{SUPC}$ is presented in [10].

In the following, we extend the schematic calculus $\mathcal{SUPC}$ for $\mathcal{UPC}$ to get a schematic calculus for $\mathcal{UPC}_I$, named $\mathcal{SUPC}_I$.

4 Schematic Paramodulation Calculus for Integer Offsets

This section introduces a new schematic calculus named $\mathcal{SUPC}_I$. It is a schematization of $\mathcal{UPC}_I$ taking into account the axioms of the theory of Integer Offsets within a framework based on schematic paramodulation [6, 10].

The theory of Integer Offsets allows us to build arithmetic expressions of the form $\mathsf{s}^n(t)$ for $n \geq 1$. The idea investigated here is to represent all terms of this form in a unique way. To this end, we consider a new operator $\mathsf{s}^+ : \text{INT} \to \text{INT}$ such that $\mathsf{s}^+(t)$ denotes the infinite set of terms $\{s^n(t) \mid n \geq 1\}$. Let us introduce the notions of schematic clause and instance of schematic clause handled by $\mathcal{SUPC}_I$. These notions extend the ones used in [6] for the schematization of $\mathcal{PC}$.

Definition 1 (Schematic Clause) *A* schematic clause *is a constrained clause built over the signature extended with* s^+*. An* instance of a schematic clause *is a constraint instance where each occurrence of* s^+ *is replaced by some* s^n *with* $n \geq 1$*.*

The calculus $\mathcal{SUPC}_I$ takes as input a set of schematic literals, G_0, that represents all possible sets of ground literals given as inputs to $\mathcal{UPC}_I$:

$$\begin{aligned} G_0 = \{\bot, x = y \parallel const(x, y), x \neq y \parallel const(x, y), u = \mathsf{s}^+(v) \| \varphi\} \\ \cup \textstyle\bigcup_{f \in \Sigma_T} \{f(x_1, \ldots, x_n) = x_0 \parallel const(x_0, x_1, \ldots, x_n)\} \end{aligned}$$

where u, v are flat terms of sort INT whose variables are all constrained (in φ), and x, y are constrained variables of the same sort.

The calculus $\mathcal{SUPC}_I$ is depicted in Fig. 1. It re-uses most of the rules of $\mathcal{SUPC}$ – Figs. 1(a) and 1(b) – and complete them with one new contraction rule named *Schematic Deletion* and two reduction rules – presented in Fig. 1(c) – which are simplification rules for Integer Offsets.

Whenever a literal is generated by superposition or simplification, the rewrite system $R\mathsf{s}^+ = \{\ \mathsf{s}^+(\mathsf{s}(x)) \to \mathsf{s}^+(x),\ \mathsf{s}(\mathsf{s}^+(x)) \to \mathsf{s}^+(x),\ \mathsf{s}^+(\mathsf{s}^+(x)) \to \mathsf{s}^+(x)\ \}$ is applied eagerly to simplify terms containing s^+. The rewrite system $R\mathsf{s}^+$ is also applied in the *Schematic Deletion* rule to implement a form of subsumption check via a morphism π replacing all the occurences of s by s^+ ($\pi(\mathsf{s}(t)) = \mathsf{s}^+(\pi(t))$ for any t, $\pi(x) = x$ if x is a variable).

It is important to note that $\mathcal{SUPC}_I$ may diverge without the new *Schematic Deletion* rule. To illustrate this point, let us take a look at the theory of lists with length. In fact, the calculus generates a schematic clause $\mathsf{len}(a) = \mathsf{s}(\mathsf{len}(b)) \| const(a, b)$ which will superpose with a renamed copy of itself, i.e. with

Superposition $$\frac{l[u'] \bowtie r \| \varphi \qquad u = t \| \psi}{\sigma(l[t] \bowtie r \| \varphi \wedge \psi)}$$

if i) $\sigma(u) \not\leq \sigma(t)$, ii) $\sigma(l[u']) \not\leq \sigma(r)$, and iii) u' is not an unconstrained variable.

Reflection $$\frac{u' \neq u \| \psi}{\bot}$$ **if** $\sigma(\psi)$ is satisfiable.

Above, u and u' are unifiable and σ is the most general unifier of u and u'.

(a) Schematic expansion inference rules

Subsumption $$\frac{S \cup \{L \| \psi, L' \| \psi'\}}{S \cup \{L \| \psi\}}$$

if a) $L \in Ax(T)$, $\psi = \emptyset$ and L' is an instance of L; or b) $L' = \sigma(L)$, $\psi' = \sigma(\psi)$, where σ is a renaming or a mapping from constrained variables to constrained variables.

Simplification $$\frac{S \cup \{C[l'] \| \varphi, l = r\}}{S \cup \{C[\sigma(r)] \| \varphi, l = r\}}$$

if i) $l = r \in Ax(T)$, ii) $l' = \sigma(l)$, iii) $\sigma(l) > \sigma(r)$, and iv) $C[l'] > (\sigma(l) = \sigma(r))$.

Tautology $$\frac{S \cup \{u = u \| \varphi\}}{S}$$

Deletion $$\frac{S \cup \{L \| \varphi\}}{S}$$ **if** φ is unsatisfiable.

Schematic Del. $$\frac{S \cup \{C' \| \varphi, C[\mathsf{s}^+(t)] \| \psi\}}{S \cup \{C[\mathsf{s}^+(t)] \| \psi\}}$$

if $\sigma(\pi(C') \downarrow_{R\mathsf{s}^+}) = C[\mathsf{s}^+(t)]$, $\sigma(\varphi) = \psi$, for a renaming σ.

(b) Schematic contraction inference rules

R1 $$\frac{S \cup \{\mathsf{s}(u) = \mathsf{s}(v) \| \varphi\}}{S \cup \{u = v \| \varphi\}}$$

R2 $$\frac{S \cup \{\mathsf{s}(u) = t \| \varphi, \mathsf{s}(v) = t \| \psi\}}{S \cup \{\mathsf{s}(v) = t \| \psi, u = v \| \psi \wedge \varphi\}}$$ **if** $\mathsf{s}(u) > t, \mathsf{s}(v) > t, u > v$

Above, all the variables in u, v, t are constrained.

(c) Schematic ground reduction inference rules

Fig. 1: Inference rules of $\mathcal{SUPC}_I$

$\mathsf{len}(a') = \mathsf{s}(\mathsf{len}(b'))\ \| const(a', b')$ to generate a schematic clause of a new form $\mathsf{len}(a) = \mathsf{s}(\mathsf{s}(\mathsf{len}(b')))\ \| const(a, b')$. Without the *Schematic Deletion* rule this process continues to generate deeper and deeper schematic clauses so that $\mathcal{SUPC}_I$ will diverge. The *Schematic Deletion* rule applies to the theory of lists with length since G_0 already contains the non-flat schematic literal $\mathsf{len}(a) = \mathsf{s}^+(\mathsf{len}(b)) \| const(a, b)$.

As in [5, 6], we are interested in satisfying the following properties:

– Any clause in a saturation generated by the paramodulation calculus with any possible input is an instance of a schematic clause in a saturation generated by the schematic paramodulation calculus with the input G_0.
– The termination of the schematic paramodulation calculus with the input G_0 implies the termination of the paramodulation calculus with any possible input.

The new form of schematization introduced for arithmetic expressions requires adapting the proofs done for the standard case [11]. Our schematic paramodulation calculus for Integer Offsets provides us with an automatic proof method for the theories considered in [8], where the termination proofs are done manually.

References

1. A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. New results on rewrite-based satisfiability procedures. *ACM Trans. Comput. Logic*, 10(1):1 – 51, 2009.
2. A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *J. Inf. Comput*, 183(2):140 – 164, 2003.
3. Maria Paola Bonacina and Mnacho Echenim. On Variable-inactivity and Polynomial T-Satisfiability Procedures. *J. Log. Comput.*, 18(1):77–96, 2008.
4. G. Godoy and R. Nieuwenhuis. Superposition with completely built-in abelian groups. *Journal of Symbolic Computation*, 37(1):1–33, 2004.
5. C. Lynch and B. Morawska. Automatic decidability. In *LICS*, pages 7–16, Copenhagen, Denmark, July 2002. IEEE Computer Society.
6. C. Lynch, S. Ranise, C. Ringeissen, and D.-K. Tran. Automatic decidability and combinability. *J. Inf. Comput*, 209(7):1026–1047, 2011.
7. E. Nicolini, C. Ringeissen, and M. Rusinowitch. Combinable extensions of Abelian groups. In R. Schmidt, editor, *CADE*, volume 5663 of *LNCS*, pages 51–66. Springer, 2009.
8. E. Nicolini, C. Ringeissen, and M. Rusinowitch. Satisfiability procedures for combination of theories sharing integer offsets. In S. Kowalewski and A. Philippou, editors, *TACAS*, volume 5505 of *LNCS*, pages 428–442. Springer, 2009.
9. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.
10. E. Tushkanova, A. Giorgetti, C. Ringeissen, and O. Kouchnarenko. A rule-based framework for building superposition-based decision procedures. In F. Durán, editor, *WRLA*, volume 7571 of *LNCS*, pages 221–239. Springer, 2012.
11. E. Tushkanova, C. Ringeissen, A. Giorgetti, and O. Kouchnarenko. Automatic Decidability for Theories with Counting Operators. In F. van Raamsdonk, editor, *RTA'13*, 2013.

Utilizing Higher-order Unifiability Algorithms in the Resolution Calculus

Tomer Libal

Microsoft Research - Inria Joint Center / Ecole Polytechnique
1 rue Honoré d'Estienne d'Orves, Palaiseau, 91120 France
`tomer.libal@inria.fr`

Abstract. Unifiability algorithms for higher-order logic are algorithms which decide the unification problem for sub-classes of higher-order logic by providing a witness. They contrast with unification procedures by deciding unification problems of infinitary nature, which might have infinitely many most general unifiers. Unification procedures for these sub-classes return a complete set of these unifiers and do not terminate. The common practice in automated deduction for higher-order logic is to utilize unification procedures and to force their termination by restricting the size of the generated unifiers. The unifiability algorithms, which are complete for certain sub-classes, allow us to have a more semantical approach. In this paper we claim that the standard resolution calculi for higher-order automated deduction do not take full advantage of the strengths of these algorithms and suggest a new calculus. We prove that this calculus can have an exponential speed-up over the traditional calculi.

1 Introduction

When one needs to refute higher-order formulas, the constrained resolution calculus [8] is normally preferred. Since there are possibly infinitely-many most-general unifiers, one can either eagerly compute a finite number or postpone the application of the unification rules. Another problem is the undecidability of the unifiability problem of higher-order terms [6]. The two common solutions to this problem are to either restrict the depth and size of the searched-for unifiers or again, to postpone the application of the unification rules and decrease the chances of non-termination. There also exist algorithms which decide the unifiability question for sub-classes of higher-order logic, such as for free groups [15] and semi-groups [14], monadic second-order [4] and bounded higher-order [20]. The main reasons these algorithms are not used in practice in automated deduction is the NP-hardness of the problems they solve and the absence of specialized calculi which can take advantage of their special feature, which is to decide the unifiability problem by giving a finite set of witnesses. Although the problems they solve are NP-hard, it was shown that at least with regard to most of the above problems, they are in fact in NP [12] which might imply their usability in automated deduction in practice. Naive uses of the constrained resolution calculus can run into one of the following two problems: If the algorithms are used just for trimming non-unifiable branches by deciding if there exists a unifier, then we face serious efficiency problems as the sets of constraints keep growing. On the other hand, an eager computation of the finite set of witnesses suffers from the "locality" property - the order of choosing clauses to resolve upon, even if all are required for the refutation, greatly affects the efficiency of the search. In this paper we introduce a specialized form of the constrained resolution calculus for utilizing efficiently these unifiability algorithms and for eliminating the "locality" property. Although we were not been able to show that the "locality" property actually harms the completeness of the search for a refutation, we have been able to show an exponential speed-up of the presented resolution calculus.

This paper is organized as follows. In the first section we introduce the abstract notions of unification and unifiability algorithms and the traditional calculus which utilizes higher-order pre-unification, the constrained resolution calculus. The second section is used for the presentation of the hybrid resolution calculus. We prove in this section its relative completeness with regard to the constrained resolution calculus. In the last section we present an infinite sequence of sets of clauses, on which the hybrid resolution calculus finds a refutation exponentially faster than the constrained resolution calculus.

2 Preliminaries

2.1 Higher-order Unification

In this section we will define the general form of unification and unifiability algorithms. We assume our language to be the simply typed lambda calculus [3] whose type set contains at least the type o over a signature containing at least the logical symbols $\mathtt{T}$, $\mathtt{F}$, $\neg$, $\vee$ and Π_α for each type α with types $o, o, o \to o, o \to o \to o$ and $(\alpha \to o) \to o$ respectively and the equality symbol $\doteq_\alpha$ of type $\alpha \to \alpha \to o$ for each type α. The equality symbol $=$ denotes syntactic equality between terms. *variables* are denoted by the symbols x, y, z while *constant symbols* are denoted by the rest of the lowercase Latin letters. Both might occur with sub or superscripts. The notions of free and bound variables are defined as usual. The *head* of a term is its topmost symbol which is not a λ-binder. A term whose head is a variable is called a *flex* term while a term whose head is a constant or a bound variable is called a *rigid* term. A *formula* is any term in the language which is of type o. A *literal* is a formula labeled by an intended truth value and is denoted by $[f]^v$ where f is a formula and $v \in \{\mathtt{T}, \mathtt{F}\}$. A *clause* is a disjunction $L_1 \vee \ldots \vee L_n$ of literals with $\vee$ having the usual properties (associativity, etc.), $[\mathtt{T}]^{\mathtt{F}}$ and $[\mathtt{F}]^{\mathtt{T}}$ are the identity elements and $[\mathtt{T}]^{\mathtt{T}}$ and $[\mathtt{F}]^{\mathtt{F}}$ are the absorbing elements. Except for the above-mentioned terms, the rest of the terms will be denoted in polish notation. A *substitution* is a mapping σ of variables to terms of the same type such that for some finite set of variables S, $\sigma(y) = y$ for all $y \notin S$. We extend the notion of substitutions to apply also to arbitrary terms as usual.

The *composition* of substitutions is defined as usual and is denoted by $\circ$. The substitutions mentioned in this paper are all normalized [22]. We assume all terms to be β-normalized and in η-expanded form unless otherwise stated (see for example [21]).

Definition 1 (Unification constraints). *A unification constraint is a literal of the form $[t \doteq_\alpha s]^F$ where t and s are terms of type α.*

Since the type α of the symbol $\doteq_\alpha$ can be derived from the types of its arguments, we will omit the subscript α from this symbol in the rest of the paper.

Definition 2 (Unification problems). *A unification problem is a disjunction of unification constraints. Given any clause, the unification problem associated with it is the disjunction of unification constraints in this clause.*

Example 1. The unification problem associated with the clause $[x(b)]^{\mathtt{T}} \vee [xa \doteq fgb]^{\mathtt{F}} \vee [a \doteq y]^{\mathtt{F}}$ is $[xa \doteq fgb]^{\mathtt{F}} \vee [a \doteq y]^{\mathtt{F}}$.

Definition 3 (Solved forms). *A unification constraint in η-normal form $[x \doteq t]^F$ is in solved form in a unification problem S if x does not occur elsewhere in S or in t. A unification problem P is in solved form if it contains only solved unification constraints. For a unification problem P in solved form, we denote by σ_P the substitution $[t/x \mid [x \doteq t]^F \in P]$. A unification problem is in pre-solved form if it contains only solved constraints or constraints of the form $t \doteq s$ where both t and s are flex terms. The substitution σ_P in this case is $\sigma_{P'} \circ \eta_P$ where P' is the sub-problem containing all solved constraints and η_P is a fixed substitution mapping all variables in the problem to fixed terms according to the types of the variables [9].*

Example 2. The problem $[x \doteq fa]^{\mathtt{F}} \vee [y(ga) \doteq zb]^{\mathtt{F}}$ is in pre-solved form while $[x \doteq fa]^{\mathtt{F}}$ is in solved form.

Definition 4 (Unifiers). *Given a unification constraint $[t \doteq s]^F$, a substitution σ is called a unifier for it if $\sigma(t) = \sigma(s)$. Let the relation $=_v$ extends $=$ such that $t =_v s$ if either $t = s$ or both t and s are flex terms, then a substitution σ is called a pre-unifier of the unification constraint if $\sigma(t) =_v \sigma(s)$. A substitution is called a (pre-)unifier of a unification problem if it (pre-)unifies all the constraints in it.*

Definition 5 (Most general unifiers). *A substitution σ is more general than a substitution θ, denoted $\sigma \leq \theta$ if there is a substitution δ, such that $\sigma \circ \delta = \theta$. A unifier for a unification problem is called most general if there is no other unifier of the problem, up to renaming of free variables, which is more general.*

Example 3. The substitution $[fy/x]$ is a most general unifier of the problem $[gxa \doteq g(fy)a]^{\mathtt{F}}$. Another, less general unifier, is $[a/y, fa/x]$.

Definition 6 (Complete sets of unifiers). *Given a unification problem P, we denote by $\texttt{unifiers}(P)$ the set of all its unifiers. The set Q is called a complete set of unifiers for P if $Q \subseteq \texttt{unifiers}(P)$ and for every substitution $\sigma \in \texttt{unifiers}(P)$, there exists a substitution $\theta \in Q$ such that $\theta \leq \sigma$.*

Definition 7 (Unification transformations). *A unification transformation is a rule of the form*

$$\frac{C \vee D}{\sigma(C \vee D')}$$

where D and D' are unification problems and C is a clause without unification constraints, σ is a substitution such that $\texttt{unifiers}(\sigma(C \vee D')) \subseteq \texttt{unifiers}(C \vee D)$.

Definition 8 (Unification procedures). *A unification procedure for a class of problems S is any set of unification transformations T such that for every unification problem $P \in S$ and unifier $\sigma \in \texttt{unifiers}(P)$, there is a sequence of transformations from T on P resulting in a solved problem P' such that $\sigma_{P'} \leq \sigma$. A pre-unification procedure is defined similarly where P' is a problem in pre-solved form.*

The most famous higher-order pre-unification procedure is Huet's [9]. In general Higher-order unification procedures do not terminate. Nevertheless, there are procedures for restricted classes, such as for problems with unifiers of restricted depth, which terminate.

Definition 9 (Unifiability algorithms). *A unifiability algorithm for a class of problems S is any set of unification transformations T together with a function Π from problems in S to well-founded measures such that for every unification problems $P, P' \in S$ such that P' is obtained from P using a rule in T, $\Pi(P') < \Pi(P)$ and such that if P is unifiable, we can obtain a problem P' in solved form. We will refer to this function, when the unifiability algorithm is given, just as Π.*

Definition 10 (Measure's bound). *Given a function Π as above, we define its bound for a given problem P as the maximal number of steps which can be taken before the measure $\Pi(P)$ reaches its minimal element. We will denote this value by $\texttt{bound}(\Pi(P))$.*

Note that a unifiability algorithm effectively decides the unifiability of a unification problem in its class.

The most well-known unifiability algorithm is for string unification [14]. Other algorithms are for monadic second-order unification [4] and several algorithms for context [5, 19], distributive [18], linear [11] and bounded higher-order unification [20, 13].

2.2 Huet's Constrained Resolution Calculus

In this section we will introduce the constrained resolution calculus [8].

An important aspect of clause normalization is Skolemization. We will use the Skolem terms defined in [17]

Definition 11 (Skolemization). *Given a clause C, let $x_1^{\alpha_1}, .., x_n^{\alpha_n}$ be the set of all free variables occurring in C where α_i is the type of variable x_i for $0 < i \leq n$, then a Skolem term of type α for C, which will be denoted by s_α is the term $f(x_1, .., x_n)$ for f a new function symbol of type $\alpha_1 \rightarrow .. \rightarrow \alpha_n \rightarrow \alpha$.*

The constrained resolution calculus is based on literals and clauses. Therefore, it is necessary to have rules for the normalization of terms into clauses.

Definition 12 (Simplification rules). *The set of simplification rules, which are used for normalizing terms into clauses, is given in Fig. 1.*

$$\frac{C \vee [\neg D]^{\mathtt{T}}}{C \vee [D]^{\mathtt{F}}}\ (\neg^T) \qquad \frac{C \vee [\neg D]^{\mathtt{F}}}{C \vee [D]^{\mathtt{T}}}\ (\neg^F)$$

$$\frac{C \vee [D_1 \vee D_2]^{\mathtt{T}}}{C \vee [D_1]^{\mathtt{T}} \vee [D_2]^{\mathtt{T}}}\ (\vee^T) \quad \frac{C \vee [D_1 \vee D_2]^{\mathtt{F}}}{C \vee [D_1]^{\mathtt{F}}}\ (\vee_l^F) \quad \frac{C \vee [D_1 \vee D_2]^{\mathtt{F}}}{C \vee [D_2]^{\mathtt{F}}}\ (\vee_r^F)$$

$$\frac{C \vee [\Pi_\alpha A]^{\mathtt{T}}}{C \vee [A x^\alpha]^{\mathtt{T}}}\ (\Pi^T)^1 \qquad \frac{C \vee [\Pi_\alpha A]^{\mathtt{F}}}{C \vee [A s_\alpha]^{\mathtt{T}}}\ (\Pi^F)^2$$

1. x is a new variable not occurring in A or C
2. s_α is a new Skolem term of type α

Fig. 1. Simplification Rules

The resolution and factorization rules, given next, correspond to cuts and contractions over terms which are not syntactically equal and their correctness is based on the unifiability of the added unification constraint.

Definition 13 (Resolution and factorization rules). *The resolution and factorization rules are given in Fig. 2.*

$$\frac{[A]^p \vee C \qquad [B]^{\neg p} \vee D}{C \vee D \vee [A \doteq B]^{\mathtt{F}}}\ \texttt{(Resolve)} \qquad \frac{[A]^p \vee [B]^p \vee C}{[A]^p \vee C \vee [A \doteq B]^{\mathtt{F}}}\ \texttt{(Factor)}$$

Fig. 2. Resolution and factorization rules

Since the simplification rules eliminate logical constants, such symbols cannot occur inside unification constraints. Therefore, a search for unifiers containing logical symbols will always fail. Huet's solution to the problem was to add splittings rules which try to instantiate set variables with different terms that contain logical symbols.

Definition 14 (Splitting rules). *The set of splitting rules is given in Fig. 3 where Y, Z and z are new variables and s_α a new Skolem term.*

$$\frac{C \vee [X(\overline{t_n})]^{\mathtt{T}}}{C \vee [Y]^{\mathtt{T}} \vee [Z]^{\mathtt{T}} \vee [X(\overline{t_n}) \doteq (Y \vee Z)]^{\mathtt{F}}}\ (S_\vee^T) \qquad \frac{C \vee [X(\overline{t_n})^p}{C \vee [Y]^{\neg p} \vee [X(\overline{t_n}) \doteq \neg Y)]^{\mathtt{F}}}\ (S_\neg^{TF})$$

$$\frac{C \vee [X(\overline{t_n})]^{\mathtt{F}}}{C \vee [Y]^{\mathtt{F}} \vee [X(\overline{t_n}) \doteq (Y \vee Z)]^{\mathtt{F}}}\ (S_\vee^{F_l}) \qquad \frac{C \vee [X(\overline{t_n})]^{\mathtt{F}}}{C \vee [Z]^{\mathtt{F}} \vee [X(\overline{t_n}) \doteq (Y \vee Z)]^{\mathtt{F}}}\ (S_\vee^{F_r})$$

$$\frac{C \vee [X(\overline{t_n})]^{\mathtt{T}}}{C \vee [Y z^\alpha]^{\mathtt{T}} \vee [X(\overline{t_n}) \doteq \Pi_\alpha Y]^{\mathtt{F}}}\ (S_\Pi^T) \qquad \frac{C \vee [X(\overline{t_n})]^{\mathtt{F}}}{C \vee [Y s_\alpha]^{\mathtt{F}} \vee [X(\overline{t_n}) \doteq \Pi_\alpha Y]^{\mathtt{F}}}\ (S_\Pi^F)$$

Fig. 3. Splitting Rules

Definition 15 (Variants). *Let C be a clause, V the set of all free variables in C and σ a substitution mapping each variable in V to a new variable, then $C\sigma$ is a variant of C.*

Definition 16 (Constrained resolution calculus). *The constrained resolution calculus contains the rules given in figures 1, 2 and 3 as well as the rules of a unification or a unifiability algorithm.*

Definition 17 (Search strategies). *Given a set of clauses to choose from and a set of rules to apply, a search strategy chooses one rule and one or more clauses such that the rule can be applied to the chosen clauses.*

Definition 18 (Derivations). *Given a set of initial clauses and a search strategy, a derivation in the constrained resolution calculus is a sequence of clauses such that each clause is obtained, using a rule from the calculus, from variants of clauses occurring earlier in the sequence or from variants of initial clauses while respecting the search strategy at each step.*

Example 4. Fig. 4 shows a derivation of the empty clause from the following clause set using a standard first-order unification algorithm.

$$\{[Pa]^{\mathtt{T}}, [Px]^{\mathtt{F}} \vee [Pfx]^{\mathtt{T}}, [Pffa]^{\mathtt{F}}\} \qquad (1)$$

The following lemma will be used later in the paper.

Lemma 1. *If a clause set S is refutable using the constrained resolution calculus, then we can obtain a refutation of S containing a clause C, such that above it we have only rule applications from figures 1, 2 and 3 and below it we have only unification rules.*

$$\begin{tikzcd}
 & & & {[Px]^{\mathrm{F}} \vee [Pfx]^{\mathrm{T}}} \arrow[dll] \arrow[dddl] & \\
{[Pa]^{\mathrm{T}}} \arrow[dr] & {[Px_1]^{\mathrm{F}} \vee [Pfx_1]^{\mathrm{T}}} \arrow[d] & & & \\
 & {[Pfx_1]^{\mathrm{T}} \vee [Pa \doteq Px_1]^{\mathrm{F}}} \arrow[d] & & & \\
 & {[Pfa]^{\mathrm{T}}} \arrow[dr] & {[Px_2]^{\mathrm{F}} \vee [Pfx_2]^{\mathrm{T}}} \arrow[d] & & \\
 & & {[Pfx_2]^{\mathrm{T}} \vee [Pfa \doteq Px_2]^{\mathrm{F}}} \arrow[d] & & \\
 & & {[Pffa]^{\mathrm{T}}} \arrow[dr] & & {[Pffa]^{\mathrm{F}}} \arrow[dl] \\
 & & & {[Pffa \doteq Pffa]^{\mathrm{F}}} \arrow[d] & \\
 & & & \Box &
\end{tikzcd}$$

Fig. 4. A derivation of the empty clause for clause set 1

Proof. We need to show that no rule among the ones from figures 1, 2 and 3 depends on a substitution generated by the unification rules. For the splitting rules it is clear as if they are applicable after a substitution is applied they are also applicable before. For the simplification rules, we might generate a literal by applying a substitution and then apply simplification rules. But, in this case we can apply splitting on the same variable and allow unification (later) to decompose the term. With regard to `(Resolve)` and `(Factor)`, if they can be applied before they can always be applied (using splittings if necessary) also afterwards.

Remark 1. The constrained resolution calculus can be applied in a fully lazy mode by postponing the application of the unification rules. The above lemma shows that when applied in this mode, the simplification rules are not required for completeness.

Theorem 1 ([8]). *A finite set of formulas is unsatisfiable with regard to Henkin's semantics [7] if and only if it is refutable by the constrained resolution calculus using a pre-unification procedure for the simply typed lambda calculus.*

Remark 2. For the above theorem to be correct, it was shown [2] that infinitely-many extensionality initial clauses must be added.

3 The Hybrid Resolution Calculus

Our main goal in this paper is to describe a resolution calculs which utilizes unifiability algorithms instead of unification procedures. The motivation for that is clear: unifiability algorithms terminate on much larger and more interesting classes of problems than unification procedures. A trivial example is the following string unification problem [10].

$$x_1 b x_2 b \dots b x_n = x_2 x_2 x_2 b x_3 x_3 x_3 b \dots b x_n x_n x_n b a a a \tag{2}$$

which has a unique unifier σ such that $\sigma(x_1) = a^{3^n}$. This problem is clearly not included in any unification class with a terminating algorithm - the depth of terms we need to search for cannot be smaller than 3^n, but the size of the problem is only $6n - 2$. But, on the other hand, a unifiability algorithm for this problem exists [14].

The hybrid resolution calculus described in this section will use two different unification approaches in parallel. The first will be to apply the unifiability algorithms eagerly in order to find a witness for the unifiability of the *current* set of constraints. The second will be to keep track of the *global* search for a refutation.

This will allow us to backtrack, once the witness we have found does not suffice, to the same unification problem again but this time compute a witness based, not on the local Π of the problem, but on the Π of the problem that was not unifiable by the original witness. In this way we are assured that our unifiability algorithms always compute the "correct" witnesses required for a refutation.

In order to keep track of the search, we will use search graphs.

Definition 19 (Search graphs). *Given a clause set S, a search graph for S is a directed graph, with a one-to-one labeling function $\mathtt{lbl}$ from nodes to clauses such that:*

- *for all clauses in S, there are nodes bearing them as labels.*
- *if there is an edge from node v_1 to node v_2 then the clause $\mathtt{lbl}(v_2)$ can be obtained from clause $\mathtt{lbl}(v_1)$ using one rule from the sets defined in definitions 12, 13 and 14.*

A full search graph is a search graph that in addition satisfies:

- *if there are nodes v and v_1 and $\mathtt{lbl}(v)$ is obtained from $\mathtt{lbl}(v_1)$ and some clause c by a binary rule, then there is a node v_2 such that $\mathtt{lbl}(v_2) = c$ and there is an edge from v_2 to v. We will consider only full search graphs from now on.*

Note that since we might have many ways to derive each clause using the allowed rules, we might also have many edges coming into each node in the search graph.

Example 5. Fig. 5 shows a possible search graph for the search from Ex. 4. As can be seen, the main role of the graphs is to factor out the unification rules.

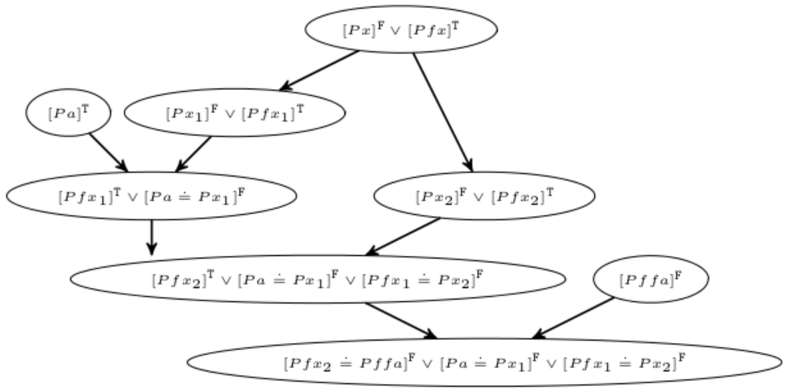
$[Px]^{\mathtt{F}} \vee [Pfx]^{\mathtt{T}}$
$[Pa]^{\mathtt{T}}$
$[Px_1]^{\mathtt{F}} \vee [Pfx_1]^{\mathtt{T}}$
$[Pfx_1]^{\mathtt{T}} \vee [Pa \doteq Px_1]^{\mathtt{F}}$
$[Px_2]^{\mathtt{F}} \vee [Pfx_2]^{\mathtt{T}}$
$[Pfx_2]^{\mathtt{T}} \vee [Pa \doteq Px_1]^{\mathtt{F}} \vee [Pfx_1 \doteq Px_2]^{\mathtt{F}}$
$[Pffa]^{\mathtt{F}}$
$[Pfx_2 \doteq Pffa]^{\mathtt{F}} \vee [Pa \doteq Px_1]^{\mathtt{F}} \vee [Pfx_1 \doteq Px_2]^{\mathtt{F}}$

Fig. 5. Some search graph for the refutation in Ex. 4.

We will also define the following function on nodes in search graphs.

Definition 20 (Maximal descendant). *Given a node v, its maximal descendant is the node whose label has the maximal Π-value for the associated unification problem of all nodes which are descendants of v.*

We can now define the hybrid calculus.

Definition 21 (Dynamic Π). *Given a function Π, a clause C and an ongoing search for a refutation denoted by the search graph G and let n be the node corresponding to C, then the dynamic Π, denoted by Π_G, computes for C the value $\Pi(P')$ where P' is the unification problem in the clause labeling the maximal descendant of n in G.*

Definition 22 (Hybrid resolution calculus). *The hybrid resolution calculus is identical to the constrained resolution calculus but utlizes the dynamic Π_G when applying the unification rules where G is the current search graph.*

The correctness of the calculus will be proved with regard to the constrained resolution calculus. In order to prove the above theorem, we need some more technical terms.

Definition 23 (Skeletons). *A skeleton of a derivation D in the constrained resolution calculus together with a unifiability algorithm is a sequence of clauses $\mathtt{skeleton}(D)$ created recursively on D as follows:*

- *if D is an initial clause then $\mathtt{skeleton}(D) = D$.*
- *if D is a clause obtained using a rule ρ applied to previous clauses $D_1, .., D_n$:*
  - *if ρ is from figures 12, 13 and 14, then $\mathtt{skeleton}(D)$ is obtained from $\mathtt{skeleton}(D_1), .., \mathtt{skeleton}(D_n)$ using ρ.*
  - *else ρ must be a unifiability rule and therefore $n = 1$. In this case we take $\mathtt{skeleton}(D) = \mathtt{skeleton}(D_1)$.*

Example 6. The skeleton of the refutation in Ex. 4 is identical (when denoted as an acyclic graph) to the graph in Fig. 5. Note that in general the search graphs might be very complex and we chose a simple search graph for the matter of demonstration only.

Lemma 2. *For any derivation D of a clause C which is obtained using the constrained resolution calculus without any unifiability rules, if the unification problem associated with C belongs to the class S of unification problems and is unifiable, then there is a unifier σ of C such that we can obtain a derivation of the clause $C\sigma$ using the hybrid resolution calculus and a unifiability algorithm for class S.*

Proof. Let G be the current search graph, we prove by induction on the structure of $\texttt{skeleton}(D)$. Let Sk be the last clause in $\texttt{skeleton}(D)$.

– if Sk is an initial clause, we can derive it also using the hybrid calculus.
– if Sk is obtained by a rule application which does not introduce new unification constraints then we can apply the same rule using the hybrid calculus.
– otherwise, Sk is obtained by a rule ρ which introduces unification constraints. Since we do not apply substitutions in D, these unification constraints occur also in C. Let P be the unification problem associated with C. Since P is contained in the class S of unification problems, it is also unifiable by some unifier σ which can be obtained by applying the unifiability algorithm on P using the measure $\Pi(P)$. Let $C_1, \ldots, C_n$ be the clauses generating Sk. Clearly, the unification problems associated with them are subsets of P and since C is either the maximal descendant of $C_1, \ldots, C_n$ or has the same Π-value as the maximal descendant, there are substitutions $\sigma_1, \ldots, \sigma_n$, which unify them respectively (using Π_G) such that there is a substitution θ such that $\sigma = \sigma_1 \circ \ldots \circ \sigma_n \circ \theta$. Therefore, according to the induction hypthesis (and note that G does not change), there are derivations of $C_1\sigma_1, \ldots, C_n\sigma_n$ using the hybrid calculus and we can apply ρ and unification rules of the unifiability algorithm in order to obtain $C\sigma$.

Theorem 2 (Relative-completeness). *Given a finite set of formulas S, if S is refutable using the constraint resolution calculus with a unifiability algorithm A and function Π, then it is refutable using the hybrid resolution calculus with A and Π.*

Proof. Since S is refutable using the constraint resolution calculus, we can apply Lemma 1 and obtain a derivation of a clause C containing no unification rule and which is unifiable by σ. Since σ can be computed by A using Π, we can use Lemma 2 in order to obtain a derivation of $C\sigma$ in the hybrid resoluton calculus. But, since C contains only unification constraints, $C\sigma$ is the empty clause and we have obtained a refutation.

4 A Speed-up Result

In order to demonstrate how the hybrid calculus takes advantage of the special attributes of the unifiability algorithms, we will define in this section a scheme of clause sets on which the hybrid resolution calculus performs better.

We first need to define a search strategy to be used by both calculi in order to choose the next clauses and literals to process as well as an evaluation function which can be applied to each calculus and be used in order to compare their performances.

Definition 24 (The search strategy). *Given a set of clauses to choose from and a set of rules to apply, the search strategy chooses clauses and the next rule to apply as follows:*

- *choose shortest clauses according to the number of characters.*
- *choose shallowest literals, where the depth of a literal is the maximal depth of a term in it.*
- *compute first unifiers mapping variables to terms of minimal depth.*
- *choose a unification transformation first if possible, otherwise, choose a transformation respecting the previous rules.*

The first two rules in the above strategy are normally used in the search for refutations as they increase the probability for a shorter refutation and simpler unification. The third rule is a consequence of most unification and unifiability procedures which apply unification rules on one symbol at a time and therefore compute minimal unifiers first.

Here is the place to discuss why we force both calculi to apply unification transformations before other transformations. The first reason is, of course, that if we postpone the application of the unification rules to the end, the hybrid calculus will perform in an identical way to the constrained resolution calculus. In fact, this is exactly the meaning of the word hybrid in the calculus name, namely, to combine the lazy and eager approaches. The reason why we would like to apply unification eagerly is clear: with the lazy approach we might traverse paths in the search which cannot be unifiable. Therefore, the search space grows much faster. In general, the size of the search space is a major bottle neck for the efficiency of the search [1].

Definition 25 (Evaluation function for derivations). *Given a resolution calculus R, a unifiability algorithm A , a search strategy S and a clause set C, the evaluation function Ψ for R, A, S and C is computed as follows: let D be the refutation obtained using R, A and S on C and let $\sigma_1, \ldots, \sigma_n$ be all the substitutions computed in D, then $\Psi(R, A, S, C) = \Sigma_{i=1}^{n} \Sigma_{x \in V_i} \mathtt{d}(\sigma_i(x))$ where V_i contains all the higher-order variables in the domain of substitution σ_i for $0 < i \leq n$ and $\mathtt{d}$ computes the depth of a term. If there is no refutation obtainable then $\Psi(R, A, S, C)$ is undefined.*

The motivation for this measure is that we will need only unification rules and rules from Def. 13 in order to refute the clause set below and the number of applications of the rules from Def. 13 will be much smaller than the number of application of unification rules. We ignore the size of the terms mapped to first-order variables in the measure as their computation requires normally one step and does not depend on the depth of the terms. With regard to the application rules themselves, although we have abstracted over the concrete rules in this paper, in all the unifiability algorithms mentioned, the size of the terms mapped to higher-order variables indeed determines the overall complexity of the algorithms.

Our choice of the clause set will be based on the search strategy defined above. When a search for a refutation is applied to the chosen clause set, the unifiers computed using the non-dynamic Π will not suffice for the resolution of later clauses and we will have to choose different clauses. The dynamic Π will allows us to proceed with the search without regard to the clauses chosen and therefore to have a significant speedup.

Definition 26 (The clause set). *Let $n > 0, m > 0$ and for a given unifiability algorithm and a function Π let:*

- $\Gamma(n) = [P_1(zc)]^T \vee .. \vee [P_n(zc)]^T \vee [Q(zc)]^T$
- $\Delta(i,m) = [P_i(\overbrace{a..a}^{m}\, y_i)]^F$ *for all* $0 < i \leq n$
- $\Lambda(m) = [Q(\overbrace{a..a}^{v+1}\, x)]^F$ *where* $v = \boldsymbol{bound}(\Pi(P_1(zc) \doteq P_1(\overbrace{a..a}^{m}\, y_1)))$

where the types of the predicates P_i and Q is $\iota \to o$ for $0 < i \leq n$, z and a are of type $\iota \to \iota$ and the rest of the terms are of type ι. The clause set $\Xi(n,m)$ is defined to be:

$$\Xi(n,m) = \{\Gamma(n), \Delta(1,m), \ldots, \Delta(n,m), \Lambda(m)\} \tag{3}$$

We abbreviate the constrained resolution calculus as `CRC`, the hybrid resolution calculus as `HRC` and the search strategy as `STG`.

Lemma 3. *Given a unifiability algorithm A with a function Π, the constrained resolution calculus is evaluated, when running on $\Xi(n,m)$, to* $\Psi(\mathit{CRC}, A, \mathit{STG}, \Xi(n,m)) = 2^n(\Sigma_{i=m}^{v}\Sigma_{j=1}^{i} i) + v + 1$.

Proof. The only possible resolution step can take place when starting with the clause $\Gamma(n)$. We resolve it with one of the Δs in order, after the elimination of all unification constraints, to obtain a unifier mapping z to $\lambda u.\overbrace{a..a}^{m}\, u$. We keep resolving the rest of the Δs until we are left with the Λ clause only. Clearly the substitution found so far does not suffice to allow us to add Λ to the derivation and we backtrack to the first step in order to compute a unifier mapping z to $\lambda u.\overbrace{a..a}^{m+1}\, u$. We continue in this way until reaching a unifier mapping z to $\lambda u.\overbrace{a..a}^{v}\, u$ which is the largest unifier which can be computed by A (see Def. 10). Since the size of the term in the Λ clause is defined to be larger than the depth of v (see Def. 10, 26), we cannot find a substitution that will allow us to resolve Γ and Λ and must choose another derivation. We note that each derivation which will start by resolving Γ with one of the Δs will not suffice and will add to the measure $\Sigma_{i=m}^{v}\Sigma_{j=1}^{i}i$. We note as well that we have 2^n possibilities to choose a subset of the Δs until the empty subset will be chosen according to our search strategy and as the last run which completes the refutation chooses the clause Λ first and the substitution computed adds $v+1$ to the total evaluation, $\Psi(\mathtt{CRC}, A, \mathtt{STG}, \Xi(n,m)) = 2^n(\Sigma_{i=m}^{v}\Sigma_{j=1}^{i}i) + v + 1$.

Lemma 4. *Given a unifiability algorithm and a function Π, the hybrid resolution calculus is evaluated, when running on the clause set,*
to $\Psi(\mathit{HRC}, A, \mathit{STG}, \Xi(n.m)) = \Sigma_{i=m}^{v+1}\Sigma_{j=1}^{i}i$.

Proof. We have a similar execution but since we are using a dynamic Π, upon reaching the clause Λ and backtracking, we can compute the right substitution. Therefore, no attempt to choose different clauses is made. The evaluation is computed by taking into account the backtracking only and is
$\Psi(\mathtt{CRC}, A, \mathtt{STG}, \Xi(n,m)) = \Sigma_{i=m}^{v+1}\Sigma_{j=1}^{i}i$.

Corollary 1. *There exists a search strategy and an infinite sequence of clause sets such that refuting them is exponentially faster when using the hybrid resolution calculus over the constrained resolution calculus.*

Proof. Since v does not depend on n and since $\Sigma_{i=m}^{v}\Sigma_{j=1}^{i}i < (v-m)(\frac{v(v+1)}{2}) < v^3$, we get that by using our clause set, starting with $n \geq v$, there is an exponential speed-up by using the hybrid calculus.

Remark 3. When comparing the running time of the two calculi, denoted by our measure Ψ, we can also encounter examples where the hybrid calculus will perform worse. Such cases will happen when we expand a path in the search which is not unifiable at some point as the dynamic Π will allow us to try larger substitutions than the non-dynamic one and therefore to have a decrease in performance.

5 Conclusion

There are only a few examples where interesting arithmetical problems could be proved using a fully-automated theorem prover [16]. The main reason for that is that arithmetical problems are normally better denoted in second-order logic. Higher-order automated deduction is not so practical due to the added complexity of higher-order unification and the undecidability of the unification problem. Syntactical restrictions on unifiers, such as restricting the depth of terms, are commonly used in order to get around this problem. These restrictions perform well on some examples but poorly on others. The power of more semantical unification procedures, which are complete with regard to specific unification problems, did not reach, as far as the author is aware, the automated deduction field. A first step towards their integration is to design a calculus which can take advantage of their benefits. Such a calculus was introduced in this paper and we have shown that this calculus can perform exponentially better when dealing with unifiability algorithms. We intend in the future to investigate the completeness of the two calculi with regard to eager applications of unifiability algorithms as we believe the ”locality” property might harm the completeness of the constrained resolution calculus. This open problem can be phrased in the following way: is there a unifiability algorithm for an interesting class S and a refutable (with respect to S) set of clauses such that no refutation of this set exists when using the constrained resolution calculus with the search strategy defined in the previous section? From the relative completeness result in Thm. 2, we know this is not the case with the hybrid resolution calculus.

Another extension of the results in this paper is to test this calculus in practice. The relationship between some unifiability algorithms and arithmetics hints that such a calculus might indeed be of use in practice. An example for this relationship is the bounded higher-order case, where the bound corresponds to set operations.

References

1. Eli Ben-Sasson. Size space tradeoffs for resolution. In *Symp. Theor. Comput.*, pages 457–464, 2002.
2. Benzmüller C. Comparing approaches to resolution based higher-order theorem proving. *Synthese*, 133(1-2):203–335, 2002.
3. Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.
4. William M. Farmer. A unification algorithm for second-order monadic terms. *Ann. Pure Appl. Logic*, 39(2):131–174, 1988.
5. Adria Gascón, Guillem Godoy, Manfred Schmidt-Schauß, and Ashish Tiwari. Context unification with one context variable. *J. Symb. Comput.*, 45(2):173–193, 2010.
6. Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theor. Comput. Sci.*, 13:225–230, 1981.
7. Leon Henkin. Completeness in the theory of types. *J. Symb. Log.*, 15(2):pp. 81–91, 1950.
8. Gérard P. Huet. *Constrained Resolution: A Complete Method for Higher Order Logic*. PhD thesis, Case Western Reserve University, 1972.
9. Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.
10. A. Koscielski and L. Pacholski. Complexity of unification in free groups and free semi-groups. In *Symp. Found. Comput. Sci.*, pages 824–829 vol.2, Washington, DC, USA, 1990. IEEE Computer Society.
11. Jordi Levy. Linear second-order unification. volume 1103 of *Lect. Not. Comput. Sci.*, 1996.
12. Jordi Levy, Manfred Schmidt-Schauß, and Mateu Villaret. Bounded second-order unification is np-complete. In *RTA*, pages 400–414, 2006.
13. Tomer Libal. Bounded higher-order unification using regular terms. In *EPiC*, 2013. To appear.

14. G S Makanin. The problem of solvability of equations in a free semigroup. *Math. USSR-Sbornik*, 32(2):129, 1977.
15. G. S. Makanin. On the decidability of the theory of free groups (in russian). In *Fund. Comput. Theor.*, pages 279–284, 1985.
16. William Mccune. Solution of the robbins problem. *J. Auto. Reaso.*, 19:263–276, 1997.
17. Dale A Miller. A compact representation of proofs. *Studia Logica*, 46(4):347–370, 1987.
18. Manfred Schmidt-Schauß. A decision algorithm for distributive unification. *Theor. Comput. Sci.*, 208:111–148, 1998.
19. Manfred Schmidt-Schauß. A decision algorithm for stratified context unification. *J. Log. Comput.*, 12(6):929–953, 2002.
20. Manfred Schmidt-Schauß and Klaus U. Schulz. Decidability of bounded higher-order unification. *J. Symb. Comput.*, 40(2):905–954, August 2005.
21. Wayne Snyder and Jean H. Gallier. Higher-order unification revisited: Complete sets of transformations. *J. Symb. Comput.*, 8(1/2):101–140, 1989.
22. Wayne S. Snyder. *Complete sets of transformations for general unification.* PhD thesis, Philadelphia, PA, USA, 1988. AAI8824793.

Exact Global Optimization on Demand (Presentation Only)

Leonardo de Moura
Microsoft Research, Redmond

Grant Olney Passmore
LFCS, Edinburgh and Clare Hall, Cambridge

April 20, 2013

Abstract

We present a method for exact global nonlinear optimization based on a real algebraic adaptation of the conflict-driven clause learning (CDCL) approach of modern SAT solving. This method allows polynomial objective functions to be constrained by real algebraic constraint systems with arbitrary boolean structure. Moreover, it can correctly determine when an objective function is unbounded, and can compute exact infima and suprema when they exist. The method requires computations over over real closed fields containing infinitesimals (cf. [1]). Finally, we briefly sketch how this method can be adapted to linear integer arithmetic, and, more generally, to various theories of arithmetic possessing computable nonstandard models.

References

[1] Leonardo de Moura and Grant Olney Passmore, *Computation over Real Closed Infinitesimal and Transcendental Extensions of the Rationals*, 24th International Conference on Automated Deduction (CADE-24), 2013.

On the conjunctive fragments of theory of linear arithmetic

Pavlos Eirinakis[1] *, Salvatore Ruggieri[2], K. Subramani[3] *, and Piotr Wojciechowski[3] *

[1]DMST, Athens University of Economics and Business
peir@aueb.gr
[2]Dipartimento di Informatica, Università di Pisa
ruggieri@di.unipi.it
[3]LDCSEE, West Virginia University
ksmani@csee.wvu.edu,pwojciec@mix.wvu.edu

Abstract. A discussion of recent developments on the conjunctive fragments of theory of linear arithmetic.

1 Introduction

In this paper, we present recent developments on the conjunctive fragments of theory of linear arithmetic. *Quantified linear programming* [1, 2] is the problem of checking whether a linear system is satisfiable with respect to a given quantifier string. Hence, it represents a generalization of linear programming. Hence, *Quantified Linear Program* (QLP) is a set of linear inequalities, where all variables are either existentially or universally quantified. By extending the quantification of variables to *implications* of two linear systems, we explore *Quantified Linear Implications* (QLIs) [3–5]. QLIs correspond to inclusion queries of polyhedral solution sets of two linear systems with respect to a given quantifier string.

QLPs represent a rich language that is ideal for expressing schedulability specifications in real-time scheduling [6–9]. In real-time scheduling, however, it may be the case that the dispatcher has already obtained a schedule (solution) but then some constraints are slightly altered. QLIs can be then utilized to decide whether the dispatcher needs to recompute a solution or can still use the current one. Moreover, QLIs can be used to model reactive systems [10, 11], where the universally quantified variables represent the environmental input, while the existentially quantified variables represent the system's response.

2 Quantified Linear Programming

A QLP is a linear system whose variables are either existentially or universally (with bounds) quantified according to a given quantifier string:

$$\exists \mathbf{x}_1\, \forall \mathbf{y}_1 \in [\mathbf{l}_1, \mathbf{u}_1]\ \ldots \exists \mathbf{x}_n\, \forall \mathbf{y}_n \in [\mathbf{l}_n, \mathbf{u}_n]\ \ \mathbf{A} \cdot \mathbf{x} + \mathbf{N} \cdot \mathbf{y} \leq \mathbf{b} \tag{1}$$

* This research was supported in part by the National Science Foundation through Award CCF-0827397 and Award CNS-0849735.

$$\begin{tikzcd}
 & \begin{array}{c}\langle 1,\exists,\mathbf{BB}\rangle\\ \textbf{(NP-complete)}\end{array} & & \\
\begin{array}{c}\langle 1,\exists,\mathbf{BL}\rangle\\ \textbf{(P)}\end{array} \arrow[ur] & \begin{array}{c}\langle 1,\exists,\mathbf{LB}\rangle\\ \textbf{(NP-complete)}\end{array} \arrow[u] & \begin{array}{c}\langle 1,\exists,\mathbf{BR}\rangle\\ \textbf{(P)}\end{array} \arrow[ul] & \begin{array}{c}\langle 1,\exists,\mathbf{RB}\rangle\\ \textbf{(P)}\end{array} \arrow[ull] \\
\begin{array}{c}\langle 1,\exists,\mathbf{LL}\rangle\\ \textbf{(P)}\end{array} \arrow[u] \arrow[ur] & \begin{array}{c}\langle 1,\exists,\mathbf{LR}\rangle\\ \textbf{(P)}\end{array} \arrow[u] \arrow[ur] & \begin{array}{c}\langle 1,\exists,\mathbf{RL}\rangle\\ \textbf{(P)}\end{array} \arrow[ull] \arrow[ur] & \begin{array}{c}\langle 1,\exists,\mathbf{RR}\rangle\\ \textbf{(P)}\end{array} \arrow[ul] \arrow[u]
\end{tikzcd}$$

Fig. 1. Complexity of $\exists\forall$ classes of QLIs. Arrows denote inclusions.

where $\mathbf{x}_1 \ldots \mathbf{x}_n$ is a partition of $\mathbf{x}$ with, possibly, $\mathbf{x}_1$ empty; $\mathbf{y}_1 \ldots \mathbf{y}_n$ is a partition of $\mathbf{y}$ with, possibly, $\mathbf{y}_n$ empty; and $\mathbf{l}_i$, $\mathbf{u}_i$ are lower and upper bounds in $\Re$ for $\mathbf{y}_i$, $i = 1, \ldots, n$.

The Fourier-Motzkin existential quantifier elimination method and a universal quantifier elimination method have been employed to provide a method for deciding QLPs [2].

Theorem 1. *Deciding a QLP of the form (1) is in* **PSPACE**.

The special case of **E**-QLP problems, which are of the form $\exists \mathbf{y}\ \forall \mathbf{x} \in [\mathbf{l}, \mathbf{u}]\ \mathbf{A}\mathbf{x} + \mathbf{N} \cdot \mathbf{y} \leq \mathbf{b}$, are solvable in polynomial time [2, Theorem 8.1]. Another special case is **F**-QLP problems, which are of the form $\forall \mathbf{y} \in [\mathbf{l}, \mathbf{u}]\ \exists \mathbf{x}\ \ \mathbf{A} \cdot \mathbf{x} + \mathbf{N} \cdot \mathbf{y} \leq \mathbf{b}$. Deciding an **F**-QLP is **coNP-complete** [2, Theorem 8.2].

3 Quantified Linear Implication

QLIs extend the notion of inclusion of linear systems to arbitrary quantifiers:

$$\exists \mathbf{x}_1\ \forall \mathbf{y}_1\ \ldots \exists \mathbf{x}_n\ \forall \mathbf{y}_n\ \ [\mathbf{A} \cdot \mathbf{x} + \mathbf{N} \cdot \mathbf{y} \leq \mathbf{b} \rightarrow \mathbf{C} \cdot \mathbf{x} + \mathbf{M} \cdot \mathbf{y} \leq \mathbf{d}] \tag{2}$$

where $\mathbf{x}_1 \ldots \mathbf{x}_n$ and $\mathbf{y}_1 \ldots \mathbf{y}_n$ are partitions of $\mathbf{x}$ and $\mathbf{y}$ respectively, and where $\mathbf{x}_1$ and/or $\mathbf{y}_n$ may be empty. The following result can be obtained through a reduction from the generic Q3SAT problem.

Theorem 2. *Deciding a QLI of the form (2) is* **PSPACE-hard**.

Let $\mathbf{Q}(\mathbf{x}, \mathbf{y})$ denote the quantifier string, namely $\exists \mathbf{x}_1\ \forall \mathbf{y}_1\ \ldots \exists \mathbf{x}_n\ \forall \mathbf{y}_n$ in the QLI (2). A nomenclature is introduced in [3] to represent the classes of QLIs. Consider a triple $\langle A, Q, R\rangle$. Let A denote the number of quantifier alternations in $\mathbf{Q}(\mathbf{x}, \mathbf{y})$ and Q the first quantifier of $\mathbf{Q}(\mathbf{x}, \mathbf{y})$. Also, let R be an $(A+1)$-character string, specifying for each quantified set of variables in $\mathbf{Q}(\mathbf{x}, \mathbf{y})$ whether they appear on the **L**eft, on the **R**ight, or on **B**oth sides of the implication.

In [12], problem $\langle 0, \forall, \mathbf{B} \rangle$ is shown in $\mathbf{P}$ by reducing the problem to a finite number of linear programs, which are in $\mathbf{P}$ by [13]. Various classes of 1- and 2-quantifier alternation QLIs are examined in [3–5]. Indicatively, we present the class of 1-quantifier alternation QLIs starting with $\exists$ in Figure 1. The case of 1-quantifier alternation QLIs starting with $\forall$ is shown to be symmetric (see [5, Figure 4]).

Finally, let us examine the case of k alternations of quantifiers. In [5], problem $\langle k, \exists, \mathbf{B}^{k+1} \rangle$ with k odd is shown to be $\mathbf{\Sigma_k^P}$**-hard**, while problem $\langle k, \forall, \mathbf{B}^{k+1} \rangle$ with k even is shown to be $\mathbf{\Pi_k^P}$**-hard**, where $\mathbf{B}^{k+1}$ denotes the string $\underbrace{\mathbf{B} \ldots \mathbf{B}}_{k+1}$.

References

1. K. Subramani, "An analysis of quantified linear programs," in *Proceedings of the 4th International Conference on Discrete Mathematics and Theoretical Computer Science (DMTCS)*, ser. Lecture Notes in Computer Science, C.S. Calude, et. al., Ed., vol. 2731. Springer-Verlag, July 2003, pp. 265–277.
2. ——, "On a decision procedure for quantified linear programs," *Annals of Mathematics and Artificial Intelligence*, vol. 51, no. 1, pp. 55–77, 2007.
3. P. Eirinakis, S. Ruggieri, K. Subramani, and P. Wojciechowski, "Computational complexities of inclusion queries over polyhedral sets," in *International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, FL, 2012.
4. ——, "A complexity perspective on entailment of paramererized linear constraints," *Constraints*, vol. 17, no. 4, pp. 461–487, 2012.
5. ——, "On quantified linear implications," *Annals of Mathematics and Artificial Intelligence - DOI: 10.1007/s10472-013-9332-3*, 2013. [Online]. Available: http://link.springer.com/article/10.1007%2Fs10472-013-9332-3
6. R. Gerber, W. Pugh, and M. Saksena, "Parametric dispatching of hard real-time tasks," *IEEE Transactions on Computing*, vol. 44, no. 3, pp. 471–479, 1995.
7. S. Choi and A. Agrawala, "Dynamic dispatching of cyclic real-time tasks with relative timing constraints," *Real-Time Systems*, vol. 19, no. 1, pp. 5–40, 2000.
8. K. Subramani, "An analysis of partially clairvoyant scheduling," *Journal of Mathematical Modelling and Algorithms*, vol. 2, no. 2, pp. 97–119, 2003.
9. ——, "An analysis of totally clairvoyant scheduling," *Journal of Scheduling*, vol. 8, no. 2, pp. 113–133, 2005.
10. T. Koo, B. Sinopoli, A. Sangiovanni-Vincentelli, and S. Sastry, "A formal approach to reactive system design: unmanned aerial vehicle flight management system design example," in *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*, 1999, pp. 522 – 527.
11. R. Hall, "Specification, validation, and synthesis of email agent controllers: A case study in function rich reactive system design," *Automated Software Engineering*, vol. 9, no. 3, pp. 233–261, 2002.
12. K. Subramani, "On the complexity of selected satisfiability and equivalence queries over boolean formulas and inclusion queries over hulls," *Journal of Applied Mathematics and Decision Sciences (JAMDS)*, vol. 2009, pp. 1–18, 2009.
13. L. G. Khachiyan, "A polynomial algorithm in linear programming," *Doklady Akademiia Nauk SSSR*, vol. 224, pp. 1093–1096, 1979, english Translation: *Soviet Mathematics Doklady*, Volume 20, pp. 1093–1096.

Author Index