

CADE 22

22nd International Conference on Automated Deduction
McGill University, Montreal, Canada
August 2009

UNIF 2009

23rd International Workshop on Unification

ADDCT 2009

Automated Deduction: Decidability, Complexity,
Tractability

Proceedings

Editors:

Christopher Lynch and Paliath Narendran (UNIF 2009)
Franz Baader, Silvio Ghilardi, Miki Hermann, Viorica Sofronie-Stokkermans,
Ashish Tiwari (ADDCT 2009)

Preface

This volume contains the joint proceedings of the 23rd International Unification Workshop (UNIF 2009), and of the Workshop “Automated Deduction: Decidability, Complexity, Tractability”, which were held on August 2, 2009, at the Trottier building at McGill University in Montreal, Canada.

UNIF is the main international meeting on unification. Previous UNIF workshops were held in Val D’Ajol, France (1987), Val D’Ajol, France (1988), Lambrecht, Germany (1989), Leeds, England (1990), Barbizon, France (1991), Dagstuhl, Germany (1992), Boston, USA (1993), Val D’Ajol, France (1994), Sitges, Spain (1995), Herrsching, Germany (1996), Orléans, France (1997), Rome, Italy (1998), Frankfurt, Germany (1999), Pittsburgh, USA (2000), Siena, Italy (2001), Copenhagen, Denmark (2002), Valencia, Spain (2003), Cork, Ireland (2004), Nara, Japan (2005), Seattle, USA (2006), Paris, France (2007), and Hagenberg, Austria (2008). UNIF 2009 was held as a workshop at the Conference on Automated Deduction. UNIF 2009 had one invited talk by Catherine Meadows (A Cryptographer’s Garden of Equational Theories: Interesting Unification Problems from Cryptography).

ADDCT 2009 is the second in a series of workshops dedicated to Automated Deduction: Decidability, Complexity, Tractability. The previous ADDCT workshop was held in Bremen, Germany on July 15th, 2007, together with the 21st Conference on Automated Deduction (CADE 21). 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); and *Application domains* for which complexity issues are essential (e.g. verification, security, databases, ontologies).

Many people helped to make UNIF 2009 and ADDCT 2009 a success. In particular, we wish to thank Brigitte Pientka, the conference chair of CADE 2009. We are also grateful to the members of the Program Committee of UNIF 2009 and ADDCT 2009.

July 2009

Chris Lynch
Paliath Narendran
UNIF 2009 program chairs

Franz Baader
Silvio Ghilardi
Miki Hermann
Viorica Sofronie-Stokkermans
Ashish Tiwari
ADDCT 2009 program chairs

UNIF 2009 Program Chairs

Christopher Lynch (Clarkson University, Potsdam, New York, USA)
Paliath Narendran (University at Albany–SUNY, Albany, New York, USA)

UNIF 2009 Program Committee

Franz Baader (TU Dresden, Dresden, Germany)
Santiago Escobar (Universidad Politecnica de Valencia, Valencia, Spain)
Christopher Lynch (Clarkson University, Potsdam, New York, USA)
Paliath Narendran (University at Albany–SUNY, Albany, New York, USA)
Christophe Ringeissen (LORIA - INRIA, Nancy, France)

ADDCT 2009 Program Chairs

Franz Baader (TU Dresden)
Silvio Ghilardi (U. degli Studi di Milano)
Miki Hermann (LIX, Ecole Polytechnique, Palaiseau)
Viorica Sofronie-Stokkermans (Max-Planck-Institut für Informatik, Saarbrücken)
Ashish Tiwari (SRI International)

ADDCT 2009 Program Committee

Carlos Areces (LORIA, Nancy)
Franz Baader (TU Dresden)
Peter Baumgartner (National ICT Australia)
Maria Paola Bonacina (U. Verona)
Christian Fermueller (TU Wien)
Silvio Ghilardi (U. degli Studi di Milano)
Miki Hermann (LIX, Ecole Polytechnique, Palaiseau)
Ullrich Hustadt (U. Liverpool)
Sava Krstic (Intel Corporation)
Christopher Lynch (Clarkson University, Potsdam, USA)
Silvio Ranise (U. Verona)
Viorica Sofronie-Stokkermans (Max-Planck-Institut für Informatik, Saarbrücken)
Lidia Tendera (Opole University)
Dmitry Tishkovsky (U. Manchester)
Ashish Tiwari (SRI International)
Luca Viganó (U. Verona)

External Reviewer

John Harrison

Table of Contents

Invited Talk

A Cryptographer’s Garden of Equational Theories: Interesting Unification Problems from Cryptography	1
<i>Catherine Meadows</i>	

Contributed Talks (UNIF 2009)

Implementing Rigid E-Unification (Extended Abstract)	2
<i>Michael Franssen</i>	
A Machine Checked Model of MGU Axioms: Applications of Finite Maps and Functional Induction	17
<i>Sunil Kothari, James Caldwell</i>	
Order-Sorted Unification with Regular Expression Sorts	32
<i>Temur Kutsia, Mircea Marin</i>	

Contributed Talks (ADDCT 2009)

A Tableau Method for Checking Rule Admissibility in S4	47
<i>Sergey Babenyshev, Vladimir Rybakov, Renate A. Schmidt, Dmitry Tishkovsky</i>	
Universality of Polynomial Positivity and a Variant of Hilbert’s 17th Problem (Work in Progress)	62
<i>Grant Olney Passmore, Leonardo de Moura</i>	
The Ackermann Approach for Modal Logic, Correspondence Theory and Second-Order Reduction: Extended Abstract (Presentation-only)	72
<i>Renate A. Schmidt</i>	
CTL-RP: A Computational Tree Logic Resolution Prover (Presentation-only)	75
<i>Lan Zhang, Ulrich Hustadt, Clare Dixon</i>	
Author Index	79

A Cryptographer's Garden of Equational Theories: Interesting Unification Problems from Cryptography

Catherine Meadows

Center for High Assurance Computer Systems
Naval Research Laboratory
Code 5543, Washington, DC 20375
e-mail meadows@itd.nrl.navy.mil

Abstract. Cryptographic protocol analysis is of a growing area of application of results from theoretical computer science. One prominent emerging area is the automated analysis of protocols in which the cryptographic algorithms obey different equational theories. In particular, this is a growing area of application for unification, and is a rich source of unification problems. In this talk we give an introduction to the application of unification to this problem, and a survey of the various theories of interest that arise. We also describe the state of the art and open problems.

Order-Sorted Unification with Regular Expression Sorts (Work in Progress)

Temur Kutsia^{1*} and Mircea Marin^{2**}

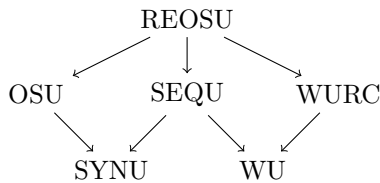
¹ Research Institute for Symbolic Computation
Johannes Kepler University Linz, Austria

² Graduate School of Systems and Information Engineering
University of Tsukuba, Japan

Abstract. We extend first-order order-sorted unification by permitting regular expression sorts for variables and in the domains of function symbols. The set of basic sorts is finite. The corresponding unification problem is infinitary. We conjecture that this unification problem is decidable and give a complete unification procedure.

1 Introduction

In first-order order-sorted unification [15], the set of basic sort S is assumed to be partially ordered, variables are of basic sorts $s \in S$ and function symbols have sorts of the form $w \rightarrow s$, where w is a finite word over S and $s \in S$. In this paper, we require S to be finite and extend the framework by introducing regular expression sorts R over S , allowing variables to be of sorts R and function symbols to have sorts $R \rightarrow s$. Another extension is that overloading function symbols is allowed. Under some reasonable conditions imposed over the signature according to [7], our terms have a least sort. We call the obtained problem regular expression order-sorted unification (REOSU) and show that it is infinitary, conjecture that it is decidable and give a complete unification procedure. REOSU extends some known problems as it is shown on the diagram below, illustrating its relations with syntactic unification (SYNU [11]), word unification (WU [13]), order-sorted unification (OSU [15]), sequence unification (SEQU [10]), and word unification with regular constraints (WURC [13]):



* Supported by JSPS Grant-in-Aid no. 20500025 for Scientific Research (C).

** Supported by the European Commission Framework 6 Programme for Integrated Infrastructures Initiatives under the project SCIENCE—Symbolic Computation Infrastructure for Europe (Contract No. 026133).

Following the arrows, from OSU one can obtain SYNU by restricting the sort hierarchy to be empty; SEQU problems without sequence variables (i.e., with individual variables only) constitute SYNU problems; on the other hand, WU is a special case of SEQU with constants, sequence variables, and only one flexible arity function symbol for concatenation; WU is also a special case of WURC where none of the variables are constrained; from REOSU we can get OSU (but with finitely many basic sort symbols only, because this is what REOSU considers) if instead of arbitrary regular sorts in function domains we allow only words over basic sorts, restrict variables to be of only basic sorts, and forbid function symbol overloading; SEQU can be obtained if we restrict REOSU with only one basic sort, say s , the variables that correspond to sequence variables in SEQU have sort s^* , individual variables are of sort s , and function symbols have the sort $s^* \rightarrow s$; finally, the WURC can be obtained from REOSU by the same restriction that gives WU from SEQU and, in addition, identifying the constants there to the corresponding sorts.

Order-sorted unification described in [12, 16] extends OSU from [15] in the way that is not compatible with REOSU.

In this paper we are dealing with REOSU in the empty theory (i.e., the syntactic case). As a future work, it would be interesting to see how equational OSU [9, 8] can be extended with regular expression sorts.

The paper is organized as follows: In Sect. 2 we give basic definitions and recall some known results. In Sect. 3 algorithms operating on sorts are given. Sect. 4 discusses decidability issues and describes a complete unification procedure. Sect. 5 concludes. Proofs can be found in the appendix.

For unification, we use the notation and terminology of [5]. For the notions related to sorted theories, we follow [7].

2 Preliminaries

Sorts. We consider a finite set \mathcal{B} of basic sorts, partially ordered with the relation \preceq . Its elements are denoted with lower case letters in **sans serif** font. $s \prec r$ means $s \preceq r$ and $s \neq r$. Regular expression sorts (shortly, sorts) are regular expressions over \mathcal{B} , built in the usual way: $R ::= s \mid 1 \mid R_1.R_2 \mid R_1+R_2 \mid R^*$.

The set of all regular expression sorts is denoted by \mathcal{R} . We use capital **SANS SERIF** font letters for them. They define the corresponding regular language in the standard way: $\llbracket s \rrbracket = \{s\}$, $\llbracket 1 \rrbracket = \{\epsilon\}$, $\llbracket R_1.R_2 \rrbracket = \{(\tilde{s}_1, \tilde{s}_2) \mid \tilde{s}_1 \in \llbracket R_1 \rrbracket, \tilde{s}_2 \in \llbracket R_2 \rrbracket\}$, $\llbracket R_1+R_2 \rrbracket = \llbracket R_1 \rrbracket \cup \llbracket R_2 \rrbracket$, $\llbracket R^* \rrbracket = \llbracket R \rrbracket^*$, where ϵ stands for the empty word.

We extend the ordering \preceq to words of basic sorts of equal lengths by $s_1 \cdots s_n \preceq r_1 \cdots r_n$ iff $s_i \preceq r_i$ for all $1 \leq i \leq n$. This ordering is extended to sets of words of basic sorts, defining $S_1 \preceq S_2$ iff for each $w_1 \in S_1$ there is $w_2 \in S_2$ such that $w_1 \preceq w_2$. Finally, we order the regular expression sorts with \preceq , defining $R_1 \preceq R_2$ iff $\llbracket R_1 \rrbracket \preceq \llbracket R_2 \rrbracket$. If $R_1 \preceq R_2$ and $R_2 \preceq R_1$ then we write $R_1 \simeq R_2$. If $R_1 \preceq R_2$ and not $R_2 \preceq R_1$, then $R_1 \prec R_2$.

The set of all \preceq -maximal elements in a set of sorts $S \subseteq \mathcal{R}$ is denoted $\max(S)$. R is a lower bound of S if $R \preceq Q$ for all $Q \in S$. A lower bound G of S is the

greatest lower bound, denoted $\text{glb}(S)$, if $R \preceq G$ for all lower bounds R of S . The sort $(\sum_{s \in \mathcal{B}} s)^*$ is the *top sort* and is denoted by \top . Obviously, $R \preceq \top$ for any R .

Terms. For each R we assume a countable set of variables \mathcal{V}_R such that $\mathcal{V}_{R_1} = \mathcal{V}_{R_2}$ iff $R_1 \simeq R_2$ and $\mathcal{V}_{R_1} \cap \mathcal{V}_{R_2} = \emptyset$ if $R_1 \not\simeq R_2$. Also, we assume as a signature a family of sets of function symbols $\{\mathcal{F}_{R,s} \mid R \in \mathcal{R}, s \in \mathcal{B}\}$ such that $\mathcal{F}_{R_1,s_1} = \mathcal{F}_{R_2,s_2}$ iff $R_1.s_1 \simeq R_2.s_2$. Moreover, the following conditions should be satisfied:

- Monotonicity: If $f \in \mathcal{F}_{R_1,s_1} \cap \mathcal{F}_{R_2,s_2}$ and $R_1 \preceq R_2$, then $s_1 \preceq s_2$.
- Preregularity: If $f \in \mathcal{F}_{R_1,s_1}$ and $R_2 \preceq R_1$, then there is a \preceq -least element in the set $\{s \mid f \in \mathcal{F}_{R,s} \text{ and } R_2 \preceq R\}$.
- Finite overloading: For each f , the set $\{\mathcal{F}_{R,s} \mid f \in \mathcal{F}_{R,s}\}$ is finite.

We say that R is a sort of x if $x \in \mathcal{V}_R$. Similarly, $R.s$ is a sort of f if $f \in \mathcal{F}_{R,s}$. Function symbols from $\mathcal{F}_{1,s}$ are called constants. We use the letters a, b, c to denote them. $\text{maxsort}(f)$ denotes the set $\max(\{R.s \mid f \in \mathcal{F}_{R,s}\})$. We will write $f : R \rightarrow s$ for $f \in \mathcal{F}_{R,s}$, $a : s$ for $a \in \mathcal{F}_{1,s}$, and $x : R$ for $x \in \mathcal{V}_R$. Setting $\mathcal{V} \in \cup_{R \in \mathcal{R}} \mathcal{V}_R$ and $\mathcal{F} = \cup_{R \in \mathcal{R}, s \in \mathcal{B}} \mathcal{F}_{R,s}$, we define the set of *sorted terms* (or, just *terms*) $\mathcal{T}(\mathcal{F}, \mathcal{V})$ over \mathcal{F} and \mathcal{V} as the least family $\{\mathcal{T}_R(\mathcal{F}, \mathcal{V}) \mid R \in \mathcal{R}\}$ of sets satisfying the following conditions:

- $\mathcal{V}_R \subseteq \mathcal{T}_R(\mathcal{F}, \mathcal{V})$.
- $\mathcal{T}_{R_1}(\mathcal{F}, \mathcal{V}) \subseteq \mathcal{T}_{R_2}(\mathcal{F}, \mathcal{V})$ if $R_1 \preceq R_2$.
- If $f : R \rightarrow s$ and $1 \preceq R$, then $f(\epsilon) \in \mathcal{T}_s(\mathcal{F}, \mathcal{V})$.
- If $f : R \rightarrow s$, $t_i \in \mathcal{T}_{R_i}(\mathcal{F}, \mathcal{V})$ for $1 \leq i \leq n$, $n \geq 1$, such that $R_1 \cdots R_n \preceq R$, then $f(t_1, \dots, t_n) \in \mathcal{T}_s(\mathcal{F}, \mathcal{V})$.

We abbreviate terms $a(\epsilon)$ with a .

Lemma 1. *For each term t there exists a \preceq -minimal sort R that is unique modulo \simeq such that $t \in \mathcal{T}_R(\mathcal{F}, \mathcal{V})$.*

This \preceq -minimal sort R is called *the sort* of t and is denoted by $\text{sort}(t)$. In the same way, the sort of a term sequence (t_1, \dots, t_n) , $n \geq 1$, is defined uniquely modulo \simeq as $\text{sort}(t_1) \cdots \text{sort}(t_n)$ and is denoted by $\text{sort}((t_1, \dots, t_n))$. When $n = 0$, i.e., for the empty sequence, $\text{sort}(\epsilon) = 1$.

The set of variables of a term t is denoted by $\text{var}(t)$. A term t is ground if $\text{var}(t) = \emptyset$. These notions extend to term sequences, sets of term sequences, etc. For a basic sort s , its semantics $\text{sem}(s)$ is the set $\mathcal{T}_s(\mathcal{F})$ of ground terms of sort s . Semantics of a regular sort is given as a set of ground term sequences of the corresponding sort: $\text{sem}(1) = \{\epsilon\}$, $\text{sem}(R_1.R_2) = \{(\tilde{s}_1, \tilde{s}_2) \mid \tilde{s}_1 \in \text{sem}(R_1), \tilde{s}_2 \in \text{sem}(R_2)\}$, $\text{sem}(R_1+R_2) = \text{sem}(R_1) \cup \text{sem}(R_2)$, $\text{sem}(R^*) = \text{sem}(R)^*$. This definition, together with the definition of \preceq and $\mathcal{T}_R(\mathcal{F}, \mathcal{V})$ implies that if $R \preceq Q$, then $\text{sem}(R) \subseteq \text{sem}(Q)$.

Substitutions. A substitution is a well-sorted mapping from variables to sequences of terms, which is identity almost everywhere. (A singleton sequence is identified with its sole member.) Substitutions are denoted with lower case

Greek letters, where ε stands for the identity substitution. Well-sortedness of σ means that $\text{sort}(\sigma(x)) \preceq \text{sort}(x)$ for all x . The notions of substitution application, term and term sequence instances, substitution composition, restriction, and subsumption are defined in the standard way. We use postfix notation for instances, juxtaposition for composition, and write $\sigma \leq_{\mathcal{X}} \vartheta$ for subsumption meaning that σ is more general than ϑ on the set of variables \mathcal{X} .

Lemma 2. *For a term t , a term sequence \tilde{t} , and a substitution σ we have $\text{sort}(t\sigma) \preceq \text{sort}(t)$ and $\text{sort}(\tilde{t}\sigma) \preceq \text{sort}(\tilde{t})$.*

Equation is a pair of term sequences, written as $\tilde{s} \doteq \tilde{t}$. A regular expression order sorted unification or, shortly, REOSU problem Γ is a finite set of equations between sorted term sequences $\{\tilde{s}_1 \doteq \tilde{t}_1, \dots, \tilde{s}_n \doteq \tilde{t}_n\}$. A substitution σ is a unifier of Γ if $\tilde{s}_i\sigma = \tilde{t}_i\sigma$ for all $1 \leq i \leq n$. A minimal complete set of unifiers of Γ is a set U of unifiers of Γ satisfying the following conditions:

- Completeness: For any unifier ϑ of Γ there is $\sigma \in U$ such that $\sigma \leq_{\text{var}(\Gamma)} \vartheta$.
- Minimality: If there are $\sigma_1, \sigma_2 \in U$ such that $\sigma_1 \leq_{\text{var}(\Gamma)} \sigma_2$, then $\sigma_1 = \sigma_2$.

We also consider a *closure* \bar{R} of a sort R , defined as follows: $\bar{\mathfrak{s}} = \sum_{r \preceq s} r$, $\bar{1} = 1$, $\overline{R_1 \cdot R_2} = \bar{R}_1 \cdot \bar{R}_2$, $\overline{R_1 + R_2} = \bar{R}_1 + \bar{R}_2$, $\overline{R^*} = \bar{R}^*$. Closure makes operations on sorts considered later easier.

Lemma 3. *Let $S, R \in \mathcal{R}$. Then $S \preceq R$ iff $\llbracket \bar{S} \rrbracket \subseteq \llbracket \bar{R} \rrbracket$.*

Corollary 1. *Let $S, R \in \mathcal{R}$. Then $S \simeq R$ iff $\llbracket \bar{S} \rrbracket = \llbracket \bar{R} \rrbracket$.*

Linear Form and Split of a Regular Expression. We recall the notion of linear form for regular expressions from [2], adapted the notation to our setting, using the set of basic sorts \mathcal{B} for alphabet: The pair (s, R) is called a monomial. A *linear form* of a regular expression R , denoted $lf(R)$, is a finite set of monomials defined recursively as follows:

$$\begin{aligned} lf(1) &= \emptyset & lf(R^*) &= lf(R) \odot R^* \\ lf(s) &= \{(s, 1)\} & lf(R \cdot Q) &= lf(R) \odot Q \quad \text{if } \epsilon \notin \llbracket R \rrbracket \\ lf(s+r) &= lf(s) \cup lf(r) & lf(R \cdot Q) &= lf(R) \odot Q \cup lf(Q) \quad \text{if } \epsilon \in \llbracket R \rrbracket \end{aligned}$$

These equations involve an extension of concatenation \odot that acts on a linear form and a regular expression and returns a linear form. It is defined as $l \odot 1 = l$ and $l \odot Q = \{(s, S \cdot Q) \mid (s, S) \in l, S \neq 1\} \cup \{(s, Q) \mid (s, 1) \in l\}$ if $Q \neq 1$.

As an example, $lf(R) = \{(s, R), (s, s \cdot (s \cdot s+r)^*), (r, (s \cdot s+r)^*)\}$ for $R = s^* \cdot (s \cdot s+r)^*$. The set $\overline{lf(R)}$ is defined as $\{s \cdot Q \mid (s, Q) \in lf(R)\}$.

Definition 1 (Split). *Let $S \in \mathcal{R}$. A split of S is a pair $(Q, R) \in \mathcal{R}^2$ such that (1) $Q \cdot R \preceq S$ and (2) if $(Q', R') \in \mathcal{R}^2$, $Q \preceq Q'$, $R \preceq R'$, and $Q' \cdot R' \preceq S$, then $Q \simeq Q'$ and $R \simeq R'$.*

We recall the definition of 2-factorization from [6]: A pair $(Q, R) \in \mathcal{R}^2$ is a *2-factorization* of $S \in \mathcal{R}$ if (1) $\llbracket Q \cdot R \rrbracket \subseteq \llbracket S \rrbracket$ and (2) if $(Q', R') \in \mathcal{R}^2$, $\llbracket Q \rrbracket \subseteq \llbracket Q' \rrbracket$, $\llbracket R \rrbracket \subseteq \llbracket R' \rrbracket$, and $\llbracket Q' \cdot R' \rrbracket \subseteq \llbracket S \rrbracket$, then $\llbracket Q \rrbracket = \llbracket Q' \rrbracket$ and $\llbracket R \rrbracket = \llbracket R' \rrbracket$.

Lemma 4. (Q, R) is a split of S iff $(\overline{Q}, \overline{S})$ is a 2-factorization of \overline{S} .

In [6] it has been shown that regular expressions admit finite factorization: The number of left factor-right factor pairs is finite. Moreover, they can be effectively computed. By the lemma above regular expressions admit finite splitting. For instance, we can split $s^*.r.r^*$ into $(s^*, s^*.r.r^*)$ and $(s^*.r.r^*, r^*)$.

3 Algorithms for Sorts

Deciding \preceq . If we did not have ordering on basic sorts, \preceq would be the standard inequality for regular word expressions which can be decided, for instance, by Antimirov’s algorithm [1] that employs partial derivatives. The problem is PSPACE-complete, but this rewriting approach has an advantage over the standard technique of translating regular expressions into automata: With it, in some cases solving derivations can have polynomial size, while any algorithm based on translation of regular expressions into DFA’s causes an exponential blow-up.

In our case, we can rely on the property $S \preceq R$ iff $\llbracket S \rrbracket \subseteq \llbracket R \rrbracket$, proved in Lemma 3. To decide the later inclusion, we do not need to take into account ordering on basic sorts. Hence, it can be decided by the original Antimirov’s algorithm on \overline{S} and \overline{R} .

Computing Greatest Lower Bounds. A greatest lower bound of regular expressions would be their intersection, if we did not have ordering on the basic sorts. Intersection can be computed either in the standard way, by translating them into automata, or by Antimirov & Mosses’s rewriting algorithm [3] for regular expressions extended with the intersection operator. Computation requires double exponential time.

Here we can employ the regular expression intersection algorithm to compute a greatest lower bound, with one modification: To compute intersection between two alphabet letters, instead of standard check whether they are the same, we compute the maximal elements in the set of their lower bounds. There can be several such maximal elements. This can be easily computed based on the ordering on basic sorts. Then we can take the sum of these elements and it will be their greatest lower bound. This construction allows to compute a greatest lower bound of two regular expressions, which is unique modulo \simeq .

An implementation of Antimirov-Mosses algorithm (see [14]) requires only minor modification to deal with the ordering on alphabet letters (basic sorts). Hence, for S and R we compute here $\text{glb}(S, R)$ and we know that if Q is a regular expression with $\llbracket Q \rrbracket = \llbracket S \rrbracket \cap \llbracket R \rrbracket$, then $\text{glb}(S, R) \simeq Q$.³

Computing Weakening Substitutions. Now we describe an algorithm that computes a substitution to weaken the sort of a term sequence towards a given sort. The necessity of such an algorithm can be demonstrated on the simple example:

³ We say that the computation of glb fails, if the (modification of) Antimirov-Mosses algorithm returns 0, and express it $\text{glb}(S, R) = \perp$.

Assume we want to unify x and $f(y)$ for $x : s$, $f : R_1 \rightarrow s_1$, $f : R_2 \rightarrow s_2$, $y : R_2$. Assume the sorts are ordered as $R_1 \preceq R_2$, $s_1 \preceq s \preceq s_2$. Then we can not unify x with $f(y)$ directly, because $\text{sort}(f(y)) = s_2 \not\preceq s = \text{sort}(x)$. However, if we weaken the sort of $f(y)$ to s_1 , then unification is possible. To weaken the sort of $f(y)$, we take its instance under the substitution $\{y \mapsto w\}$, where $\text{sort}(w) = R_1$, which gives $\text{sort}(f(w)) = s_1$. Hence, the substitution $\{y \mapsto w, x \mapsto f(w)\}$ is a unifier of x and $f(y)$, leading to the common instance $f(w)$.

A weakening pair is a pair of a term sequence \tilde{t} and a sort Q , written $\tilde{t} \rightsquigarrow Q$. A substitution ω is called a *weakening substitution* of a set W of weakening pairs iff $\text{sort}(\tilde{t}\omega) \preceq Q$ for each $\tilde{t} \rightsquigarrow Q \in W$.

The algorithm \mathfrak{W} that is supposed to compute weakening substitutions for a given set of weakening pairs, works by applying exhaustively the following rules, selecting a weakening pair and transforming it in all possible ways:

R-w: Remove a Weakening Pair

$$\{\tilde{t} \rightsquigarrow Q\} \cup W; \sigma \Longrightarrow W; \sigma \quad \text{if } \text{sort}(\tilde{t}) \preceq Q.$$

D1-w: Decomposition 1 in Weakening

$$\{(f(\tilde{t}), \tilde{s}) \rightsquigarrow Q\} \cup W; \sigma \Longrightarrow \{f(\tilde{t}) \rightsquigarrow s, \tilde{s} \rightsquigarrow S\} \cup W; \sigma$$

if $\text{sort}((f(\tilde{t}), \tilde{s})) \not\preceq Q$, $\text{var}((f(\tilde{t}), \tilde{s})) \neq \emptyset$, $\tilde{s} \neq \epsilon$ and $s.S \in \max(\overline{\text{f}}(Q))$.

D2-w: Decomposition 2 in Weakening

$$\{(x, \tilde{s}) \rightsquigarrow Q\} \cup W; \sigma \Longrightarrow \{x \rightsquigarrow Q_1, \tilde{s} \rightsquigarrow Q_2\} \cup W; \sigma$$

if $\text{sort}((x, \tilde{s})) \not\preceq Q$, $\tilde{s} \neq \epsilon$ and (Q_1, Q_2) is a split of Q .

AS-w: Argument Sequence Weakening

$$\{f(\tilde{t}) \rightsquigarrow Q\} \cup W; \sigma \Longrightarrow \{\tilde{t} \rightsquigarrow R\} \cup W; \sigma$$

where $\text{sort}(f(\tilde{t})) \not\preceq Q$, $\text{var}(f(\tilde{t})) \neq \emptyset$, $R.r \in \text{maxsort}(f)$, and $r \preceq Q$.

V-w: Variable Weakening

$$\{x \rightsquigarrow Q\} \cup W; \sigma \Longrightarrow W\sigma; \sigma\{x \mapsto w\}$$

where w is a fresh variable with $\text{sort}(w) = \text{glb}(\{\text{sort}(x), Q\})$.

F1-w: Failure 1 in Weakening

$$\{\tilde{t} \rightsquigarrow Q\} \cup W; \sigma \Longrightarrow \perp, \quad \text{if } \text{sort}(\tilde{t}) \not\preceq Q \text{ and } \text{var}(\tilde{t}) = \emptyset.$$

F2-w: Failure 2 in Weakening

$$\{f(\tilde{t}) \rightsquigarrow Q\} \cup W; \sigma \Longrightarrow \perp,$$

if $\text{sort}(f(\tilde{t})) \not\preceq Q$ and $r \not\preceq Q$ for each r with $f : R \rightarrow r$.

F3-w: Failure 3 in Weakening

$$\{x \rightsquigarrow Q\} \cup W; \sigma \Longrightarrow \perp, \quad \text{if } \text{glb}(\{\text{sort}(x), Q\}) = \perp.$$

By the exhaustive search described above, a complete search tree is generated whose branches form *derivations*. The branches that end with \perp are called failing branches. The branches that end with $\emptyset; \omega$ are called successful branches and ω

is a substitution computed by \mathfrak{W} at this branch. The set of all substitutions computed by \mathfrak{W} on the set of weakening pairs W is denoted by $weak(W)$. It is easy to see that the elements of $weak(W)$ are variable renaming substitutions.

It is essential that the signature has the finite overloading property, which guarantees that the rule AS-w does not introduce infinite branching. Since the linear form and split of a regular expression are both finite, the other rules do not cause infinite branching either. \mathfrak{W} is terminating, sound, and complete, as the following theorems show.

Theorem 1. \mathfrak{W} is terminating.

Theorem 2. \mathfrak{W} is sound: Each $\omega \in weak(W)$ is a weakening substitution of W .

Theorem 3. \mathfrak{W} is complete: For every weakening substitution ω' of W there exists $\omega \in weak(W)$ such that $\omega \leq_{vars(W)} \omega'$.

Example 1. Let $W = \{x \rightsquigarrow \mathbf{q}, f(x) \rightsquigarrow \mathbf{s}\}$ be a weakening problem with $x : r$, $f : \mathbf{s} \rightarrow \mathbf{s}$, $f : r \rightarrow r$ and the sorts $r_1 \prec r$, $r_2 \prec r$, $r_1 \prec \mathbf{q}$, $r_2 \prec \mathbf{q}$, $\mathbf{s} \prec r_1$, $\mathbf{s} \prec r_2$. Then the weakening algorithm computes $weak(W) = \{\{x \mapsto z, w \mapsto z\}\}$ where $w : r_1 + r_2$ and $z : \mathbf{s}$.

Example 2. Let $W = \{(x, y) \rightsquigarrow \mathbf{s}^*.r.r^*\}$ be a weakening problem with $x : \mathbf{q}_1^*.p_1^*$, $y : \mathbf{q}_2^*.p_2^*$, and the sorts $\mathbf{s} \prec \mathbf{q}_1$, $\mathbf{s} \prec \mathbf{q}_2$, $r \prec p_1$, $r \prec p_2$. Then the weakening algorithm computes $weak(W) = \{\{x \mapsto u_1, y \mapsto v_1\}, \{x \mapsto u_2, y \mapsto v_2\}\}$ where $u_1 : \mathbf{s}^*.r.r^*$, $v_1 : r^*$, $u_2 : \mathbf{s}^*$ and $v_2 : \mathbf{s}^*.r.r^*$.

Example 3. Let $W = \{x \rightsquigarrow \mathbf{q}^*\}$ be a weakening problem with $x : r^*$ and the sorts $\mathbf{s}_1 \prec r$, $\mathbf{s}_2 \prec r$, $\mathbf{s}_1 \prec \mathbf{q}$, $\mathbf{s}_2 \prec \mathbf{q}$, $p_1 \prec \mathbf{s}_1$, $p_2 \prec \mathbf{s}_2$. Then the weakening algorithm computes $weak(W) = \{\{x \mapsto w\}\}$ where $w : (\mathbf{s}_1 + \mathbf{s}_2)^*$.

4 Unification Type, Decidability and Unification Procedure

Unification Type. Let Γ_{re} be a REOSU problem and Γ_{seq} its version without sorts, i.e. a SEQU problem. Each unifier of Γ_{re} is either an unifier of Γ_{seq} or is obtained from an unifier of Γ_{seq} by composing it with a weakening substitution as follows: If $\sigma = \{x_1 \mapsto \tilde{t}_1, \dots, x_n \mapsto \tilde{t}_n\}$ is a unifier of Γ_{seq} , then the set of weakening substitutions for σ is $\Omega(\sigma) = weak(\{\tilde{t}_1 \rightsquigarrow sort(x_1), \dots, \tilde{t}_n \rightsquigarrow sort(x_n)\})$. For each $\omega_\sigma \in \Omega(\sigma)$, $\sigma\omega_\sigma$ is a unifier of Γ_{re} . Since SEQU is infinitary, the type of REOSU can be either infinitary or nullary, and we show now that it is not nullary.

Let S_{seq} be a minimal complete set of unifiers of Γ_{seq} and S_{re} be the set containing the unifiers of Γ_{re} that are either in S_{seq} or are obtained by weakening unifiers in S_{re} . Since $\{\sigma\omega_\sigma \mid \omega_\sigma \in \Omega(\sigma)\}$ is finite for each σ , we can assume that S_{re} contains only a minimal subset of it for each σ . The set S_{re} is complete. Assume by contradiction that it is not minimal. Then it contains σ' and ϑ' such that $\sigma' \leq_{var(\Gamma_{re})} \vartheta'$, i.e., there exists φ' such that $\sigma'\varphi' =_{var(\Gamma_{re})} \vartheta'$. If $\vartheta' \in S_{seq}$,

then we have $\sigma'\varphi' = \sigma\omega_\sigma\varphi' =_{\text{var}(\Gamma)} \vartheta'$ for an $\omega_\sigma \in \Omega(\sigma)$, which contradicts minimality of S_{seq} . If $\sigma' \in S_{\text{seq}}$, then $\sigma'\varphi' =_{\text{var}(\Gamma_{\text{re}})} \vartheta' = \vartheta\omega_\vartheta$ where $\omega_\vartheta \in \Omega(\vartheta)$. Since ω_ϑ is variable renaming, $\sigma'\varphi'\omega_\vartheta^{-1} =_{\text{var}(\Gamma_{\text{seq}})} \vartheta'$, which again contradicts minimality of S_{seq} . Both σ' and ϑ' can not be from S_{seq} because S_{seq} is minimal. If neither σ' nor ϑ' is in S_{seq} , then we have $\sigma\omega_\sigma\varphi' = \sigma'\varphi' =_{\text{var}(\Gamma_{\text{re}})} \vartheta' = \vartheta\omega_\vartheta$ and again a contradiction: $\sigma\omega_\sigma\varphi'\omega_\vartheta^{-1} =_{\text{var}(\Gamma_{\text{seq}})} \vartheta'$.

Hence, for any Γ_{re} there is a complete set of unifiers with no two elements comparable with respect to $\leq_{\text{var}(\Gamma_{\text{re}})}$, which implies that Γ_{re} has a minimal complete set of unifiers and REOSU is not nullary.

Is REOSU Decidable? In this subsection we reduce solvability of REOSU to solvability of a unification problem in the union of two disjoint theories: word unification with regular constraints and order-sorted syntactic unification with finitely many sorts. Sort symbols can be shared. Each of these problems is decidable [13, 16], but we do not know whether it implies decidability of REOSU. A version with regular constraints (or an order-sorted version) of Baader-Schulz combination method [4] would imply it, if the restrictions on the ingredient theories remain as reasonable as for the unsorted case (e.g. linear constant restrictions). However, we are not aware of any work on such a combination method.

Here we describe solvability-preserving transformations that transform a given REOSU problem Γ_1 into a problem of the union theory. First, we use variable abstraction on Γ_1 , replacing each subterm at depth 1 or deeper with a fresh variable of the same sort. For instance, with this transformation from $\Gamma = \{x \doteq f(g(y), a), g(x, f(z)) \doteq g(f(z), x)\}$ we obtain a new REOSU problem $\Gamma_2 = \{x \doteq f(u, v), u \doteq g(y), v \doteq a, g(x, w) \doteq g(w, x), w \doteq f(z)\}$.

In Γ_2 there can be equations of the form $x \doteq t$ such that $\text{sort}(t) \not\leq \text{sort}(x)$. If t is ground, we can immediately stop with failure. Otherwise, we weaken t towards $\text{sort}(x)$, apply weakening substitutions on Γ_2 obtaining finitely many instances, and repeat this step on each instance until we reach unification problems $\Gamma_3^1, \dots, \Gamma_3^k$ such that each equation of the form $x \doteq t$ in them satisfies $\text{sort}(t) \leq \text{sort}(x)$. The problems where it is not possible are discarded as unsolvable. Γ_1 is solvable iff Γ_3^i is solvable for some $1 \leq i \leq k$.

Now, Γ_3 's are transformed as follows: First, we introduce a new least sort n and a new function symbol $\text{conc} : \top \rightarrow n$. Next, we introduce new function symbols $f' : n \rightarrow s$ for each $f : R \rightarrow s$. The obtained signature is denoted by \mathcal{F}' . Then, every term in Γ_3 of the form $f(x_1, \dots, x_n)$ is replaced by $f'(\text{conc}(x_1, \dots, x_n))$. The new problem is denoted with Γ_4 . This transformation preserves solvability: If σ is a solution of Γ_3 , then Γ_4 has a solution σ' , obtained from σ with the same transformation as Γ_4 was obtained from Γ_3 . On the other hand, if σ' solves Γ_4 , we can obtain σ from it by getting rid of conc and replacing each f' with f . The obtained mapping is well-sorted and, therefore, is a substitution. It solves Γ_3 .

Γ_4 is a unification problem over the union of two disjoint signatures: \mathcal{F}' and $\{\text{conc}\}$. The signature \mathcal{F}' contains unary function symbols only, whose domain sort is n . We have only basic sorts (not regular expression sorts) in this theory, and they are finitely many. The theory with the signature conc can be seen as

elementary word unification problem with regular expression sorts. Hence, Γ_4 is a unification problem in the combination of two disjoint theories that share basic sorts: One is a first-order order-sorted unification with finitely many sorts and the other one is word unification with regular expression sorts. Decidability of the first one follows for decidability of elementary order-sorted unification [16]. Decidability of the second one follows from decidability of word unification with regular constraints, because one can encode the sort information as regular constraints over an extended alphabet. What is left to show is that the Baader-Schulz method remains correct in the presence of regular constraints (or in the order-sorted case). We conjecture this is true, but do not have a formal proof at the moment of writing this paper.

Conjecture 1. REOSU is decidable.

Unification Procedure. To compute unifiers for a REOSU problem, we could employ the SEQU unification procedure and then weaken each computed substitution. However, this is a generate and test approach that can be made better if we tailor weakening in the unification rules. This is what we do now.

The rules for REOSU procedure below transform a pair of a unification problem and a substitution into either again such a pair.

P: Projection

$$\Gamma; \sigma \Longrightarrow \Gamma\vartheta; \sigma\vartheta,$$

for $\vartheta = \{x_1 \mapsto \epsilon, \dots, x_n \mapsto \epsilon\}$ with $x_i \in \text{var}(\Gamma)$ and $1 \preceq \text{sort}(x_i)$ for $1 \leq i \leq n$.

T: Trivial

$$\{\tilde{t} \doteq \tilde{t}\} \cup \Gamma; \sigma \Longrightarrow \Gamma; \sigma.$$

TP: Trivial Prefix

$$\{(\tilde{r}, \tilde{t}) \doteq (\tilde{r}, \tilde{s})\} \cup \Gamma; \sigma \Longrightarrow \{\tilde{t} \doteq \tilde{s}\} \cup \Gamma; \sigma, \quad \text{if } \tilde{r} \neq \epsilon \text{ and } \tilde{t} \neq \tilde{s}.$$

D: Decomposition

$$\{(f(\tilde{t}), \tilde{t}') \doteq (f(\tilde{s}), \tilde{s}')\} \cup \Gamma; \sigma \Longrightarrow \{\tilde{t} \doteq \tilde{s}, \tilde{t}' \doteq \tilde{s}'\} \cup \Gamma; \sigma,$$

if $\text{glb}(\{\text{sort}(f(\tilde{t})), \text{sort}(f(\tilde{s}))\}) \neq \perp$ and $\tilde{t} \neq \tilde{s}$.

O: Orient

$$\{(t, \tilde{t}) \doteq (x, \tilde{s})\} \cup \Gamma; \sigma \Longrightarrow \{(x, \tilde{s}) \doteq (t, \tilde{t})\} \cup \Gamma; \sigma, \quad \text{where } t \notin \mathcal{V}.$$

WkE1: Weakening and Elimination 1

$$\{(x, \tilde{t}) \doteq (s, \tilde{s})\} \cup \Gamma; \sigma \Longrightarrow \{\tilde{t} \doteq \tilde{s}\}\vartheta \cup \Gamma\vartheta; \sigma\vartheta,$$

where $s \notin \mathcal{V}$, $x \notin \text{var}(s)$, $\omega \in \text{weak}(\{s \rightsquigarrow \text{sort}(x)\})$, and $\vartheta = \omega \cup \{x \mapsto s\omega\}$.

WkE2: Weakening and Elimination 2

$$\{(x, \tilde{t}) \doteq (y, \tilde{s})\} \cup \Gamma; \sigma \Longrightarrow \{\tilde{t} \doteq \tilde{s}\}\vartheta \cup \Gamma\vartheta; \sigma\vartheta,$$

where $\vartheta = \{x \mapsto w, y \mapsto w\}$ for a fresh variable w whose sort $\text{sort}(w) = \text{glb}(\{\text{sort}(x), \text{sort}(y)\})$ and $\text{sort}(w) \neq 1$.

WkWd1: Weakening and Widening 1

$$\{(x, \tilde{t}) \doteq (s, \tilde{s})\} \cup \Gamma; \sigma \Longrightarrow \{(z, \tilde{t}) \doteq \tilde{s}\} \vartheta \cup \Gamma \vartheta; \sigma \vartheta,$$

if $s \notin \mathcal{V}$, $x \notin \text{var}(s)$, there is $(r, R) \in \text{lf}(\text{sort}(x))$ with $R \neq 1$, $\omega \in \text{weak}(\{s \rightsquigarrow r\})$, z is a fresh variable with $\text{sort}(z) = R$ and $\vartheta = \omega \cup \{x \mapsto (s\omega, z)\}$.

WkWd2: Weakening and Widening 2

$$\{(x, \tilde{t}) \doteq (y, \tilde{s})\} \cup \Gamma; \sigma \Longrightarrow \{(z, \tilde{t}) \doteq \tilde{s}\} \vartheta \cup \Gamma \vartheta; \sigma \vartheta,$$

where (S, R) is a split of $\text{sort}(x)$ such that $R \neq 1$, w is a fresh variable with $\text{sort}(w) = \text{glb}(\{S, \text{sort}(y)\})$ and $\text{sort}(w) \neq 1$, z is a fresh variable with $\text{sort}(z) = R$, and $\vartheta = \{x \mapsto (w, z), y \mapsto w\}$.

WkWd3: Weakening and Widening 3

$$\{(x, \tilde{t}) \doteq (y, \tilde{s})\} \cup \Gamma; \sigma \Longrightarrow \{(z, \tilde{t}) \doteq \tilde{s}\} \vartheta \cup \Gamma \vartheta; \sigma \vartheta,$$

where (S, R) is a split of $\text{sort}(y)$ such that $R \neq 1$, w is a fresh variable with $\text{sort}(w) = \text{glb}(\{S, \text{sort}(x)\})$ and $\text{sort}(w) \neq 1$, z is a fresh variable with $\text{sort}(z) = R$, and $\vartheta = \{x \mapsto w, y \mapsto (w, z)\}$.

Note that $\text{sort}(w) \neq 1$ in WkWd2 and WkWd3 implies that in those rules $S \neq 1$.

We denote this set of transformation rules with \mathfrak{T} .

Theorem 4 (Soundness). *The rules in \mathfrak{T} are sound.*

To solve an unification problem Γ , we create the initial pair $\Gamma; \varepsilon$ and first apply the projection rule to it in all possible ways. From each obtained problem we select an equation and apply the other rules exhaustively to that selected equation, developing the search tree in a breadth-first way. If no rule applies, the problem is transformed \perp . The obtained procedure is denoted by $\mathfrak{P}(\Gamma)$. Branches in the search tree form *derivations*. The derivations that end with \perp are *failing derivations*. The derivations that end with $\emptyset; \varphi$ are *successful derivations*. The set of all φ 's at the end of successful derivations of $\mathfrak{P}(\Gamma)$ is called the *computed substitution set* of $\mathfrak{P}(\Gamma)$ and is denoted by $\text{comp}(\mathfrak{P}(\Gamma))$. From Theorem 4 by induction on the length of derivations one can prove that every $\varphi \in \text{comp}(\mathfrak{P}(\Gamma))$ is a unifier of Γ .

One can observe that under this control, variables are replaced with ϵ only at the projection phase. In particular, no variable introduced in intermediate stages gets eliminated with ϵ or replaced by a variable whose sort is 1.

Theorem 5 (Completeness). *Let Γ be a REOSU problem with a unifier ϑ . Then there exists $\sigma \in \text{comp}(\mathfrak{P}(\Gamma))$ such that $\sigma \leq_{\text{var}(\Gamma)} \vartheta$.*

Note that the set $\text{comp}(\mathfrak{P}(\Gamma))$, in general, is not minimal.⁴

⁴ However, if in the rules WkE1 and WkE2 the substitution ω is selected from a minimal subset of the corresponding weakening set, one can show that $\text{comp}(\mathfrak{P}(\Gamma))$ is almost minimal. (Almost minimality is defined in [10]). We do not go into the details here.

5 Conclusion

We studied unification in order-sorted theories with regular expression sorts. We showed how it generalizes some known unification problems, conjectured its decidability and gave a complete unification procedure. As the language can model quite naturally DTD and in some extent, XML Schema, one can see a possible application (perhaps of its fragments) in the area related to XML processing.

References

1. V. Antimirov. Rewriting regular inequalities (extended abstract). In H. Reichel, editor, *Fundamentals of Computation Theory, 10th International Symposium FCT'95*, volume 965 of *LNCS*, pages 116–125. Springer, 1995.
2. V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.
3. V. Antimirov and P. D. Mosses. Rewriting extended regular expressions. *Theor. Comput. Sci.*, 143(1):51–72, 1995.
4. F. Baader and K. U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *J. Symbolic Computation*, 21(2):211–244, 1996.
5. F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.
6. J.H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.
7. J. A. Goguen and J. Meseguer. Order-sorted algebra i: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theor. Comput. Sci.*, 105(2):217–273, 1992.
8. J. Hendrix and J. Meseguer. Order-sorted unification revisited. In G. Kniessel and J. Sousa Pinto, editors, *Pre-proceedings of the 9th International Workshop on Rule-Based Programming, RULE'09*, pages 16–29, 2008.
9. C. Kirchner. Order-sorted equational unification. Presented at the fifth International Conference on Logic Programming (Seattle, USA), 1988. Also as rapport de recherche INRIA 954, Dec. 88.
10. T. Kutsia. Solving equations with sequence variables and sequence functions. *J. Symbolic Computation*, 42(3):352–388, 2007.
11. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
12. M. Schmidt-Schauß. *Computational Aspects of an Order-sorted Logic with Term Declarations*. Number 395 in *Lecture Notes in Computer Science*. Springer, 1989.
13. K. U. Schulz. Makanin's algorithm for word equations – two improvements and a generalization. In K. Schulz, editor, *Word Equations and Related Topics*, number 572 in *LNCS*, pages 85–150. Springer, 1990.
14. M. Sulzmann. regexpr-symbolic: Regular expressions via symbolic manipulation. <http://hackage.haskell.org/cgi-bin/hackage-scripts/package/regexpr-symbolic>, 2009.
15. Ch. Walther. Many-sorted unification. *J. ACM*, 35(1):1–17, 1988.
16. Ch. Weidenbach. Unification in sort theories and its applications. *Annals of Mathematics and Artificial Intelligence*, 18(2):261–293, 1996.

A Proofs of Lemmas and Theorems

Lemma 1. *For each term t there exists a \preceq -minimal sort R that is unique modulo \simeq such that $t \in \mathcal{T}_R(\mathcal{F}, \mathcal{V})$.*

Proof. By structural induction. Let t be a variable $x \in \mathcal{V}_R$. Assume $x \in \mathcal{T}_Q(\mathcal{F}, \mathcal{V})$ for some Q . It implies that $x \in \mathcal{V}_{Q'}$ with $Q' \preceq Q$. Then $x \in \mathcal{V}_R \cap \mathcal{V}_{Q'}$, which implies $R \simeq Q'$. Hence, R is the unique \preceq -minimal sort modulo \simeq with $x \in \mathcal{T}_R(\mathcal{F}, \mathcal{V})$.

If t is a constant, the lemma follows from the monotonicity condition.

If $t = f(t_1, \dots, t_n)$, then, by the induction hypothesis, each t_i has the \preceq -minimal sort R_i , $1 \leq i \leq n$, that is unique modulo \simeq . Let $f : Q \rightarrow q$ such that $R_1 \cdots R_n \preceq Q$. By preregularity, there exists a \preceq -least s such that $f : S \rightarrow s$ and $R_1 \cdots R_n \preceq S$. Hence, s is the \preceq -minimal sort such that $t \in \mathcal{T}_s(\mathcal{F}, \mathcal{V})$. \square

Lemma 2. *For a term t , a term sequence \tilde{t} , and a substitution σ we have $sort(t\sigma) \preceq sort(t)$ and $sort(\tilde{t}\sigma) \preceq sort(\tilde{t})$.*

Proof. We prove $sort(t\sigma) \preceq sort(t)$ by induction on term structure. If t is a variable, then the lemma follows from the definition of substitution. If $t = f(\epsilon)$ then it is obvious. If $t = f(t_1, \dots, t_n)$, $n \geq 1$, then by the induction hypothesis $sort(t_i\sigma) \preceq sort(t_i)$ for each $1 \leq i \leq n$. Therefore, $sort((t_1, \dots, t_n)\sigma) \preceq sort(t_1, \dots, t_n)$ which, by the definition of sorted terms, implies that $sort(t\sigma) = sort(f(t_1, \dots, t_n)\sigma) \preceq sort(f(t_1, \dots, t_n)) = sort(t)$.

To prove $sort(\tilde{t}\sigma) \preceq sort(\tilde{t})$, we use induction on the length of \tilde{t} . If the length is 0, then the lemma is obvious. Otherwise, let \tilde{t} be (t, \tilde{t}') . By the case above we have $sort(t\sigma) \preceq sort(t)$. By the induction hypothesis we get $sort(\tilde{t}'\sigma) \preceq sort(\tilde{t}')$. Therefore, $sort(\tilde{t}\sigma) = sort((t, \tilde{t}')\sigma) \preceq sort((t, \tilde{t}')) = sort(\tilde{t})$. \square

Lemma 3. *Let $S, R \in \mathcal{R}$. Then $S \preceq R$ iff $\overline{[S]} \subseteq \overline{[R]}$.*

Proof. For any $Q \in \mathcal{R}$ and $v \in \llbracket Q \rrbracket$, let $v \downarrow Q := \{w \in \llbracket Q \rrbracket \mid w \preceq v\}$. By the definition of closure, we have that $\{\max(v \downarrow Q) \mid v \in \llbracket Q \rrbracket\} = \llbracket Q \rrbracket$. Now, we can reason as follows: $S \preceq R$ iff $\llbracket S \rrbracket \preceq \llbracket R \rrbracket$ iff $\{\max(v \downarrow \overline{[S]}) \mid v \in \overline{[S]}\} \preceq \{\max(v \downarrow \overline{[R]}) \mid v \in \overline{[R]}\}$ iff $\overline{[S]} \subseteq \overline{[R]}$. \square

Lemma 4. *(Q, R) is a split of S iff $(\overline{[Q]}, \overline{[R]})$ is a 2-factorization of $\overline{[S]}$.*

Proof. (Q, R) is a split of S iff (1) $Q.R \preceq S$ and (2) if $(Q', R') \in \mathcal{R}^2$, $Q \preceq Q'$, $R \preceq R'$, and $Q'.R' \preceq S$, then $Q \simeq Q'$ and $R \simeq R'$. By Lemma 3, these conditions are equivalent to (1') $\overline{[Q.R]} \subseteq \overline{[S]}$ and (2') if $(Q', R') \in \mathcal{R}^2$, $\overline{[Q]} \subseteq \overline{[Q']}$, $\overline{[R]} \subseteq \overline{[R']}$, and $\overline{[Q'.R']} \subseteq \overline{[S]}$, then $\overline{[Q]} = \overline{[Q']}$ and $\overline{[R]} = \overline{[R']}$. It is not hard to see that (1') and (2') are the same as saying that $(\overline{[Q]}, \overline{[R]})$ is a 2-factorization of $\overline{[S]}$. \square

Theorem 1. *The algorithm \mathfrak{W} is terminating.*

Proof. The complexity measure of a weakening pair $\tilde{t} \rightsquigarrow Q$ is 1 + the denotational length of \tilde{t} , and the complexity measure of a set W of weakening pairs is the multiset of the complexity measures of its constituent weakening pairs.

The multiset extension of the standard ordering on nonnegative integers is well-founded. The first five rules in \mathfrak{W} strictly decrease the measure for the sets they operate on, and the failure rules cause immediate termination. Hence, \mathfrak{W} is terminating. \square

Theorem 2. *The algorithm \mathfrak{W} is sound: Every $\omega \in \text{weak}(W)$ is a weakening substitution of W .*

Proof. It is enough to show that if a rule in \mathfrak{W} transforms $W_1; \sigma$ into $W_2; \sigma\vartheta$ and φ is a weakening substitution for W_2 , then $\vartheta\varphi$ is a weakening substitution for W_1 . For R-w, Lemma 2 implies it. For D1-w it follows from two facts: First, if $s.S \in \max(\overline{lf}(Q))$ then $s.S \preceq Q$, and second, \preceq -monotonicity of concatenation: If $R_1 \preceq Q_1$ and $R_2 \preceq Q_2$ then $R_1.R_2 \preceq Q_1.Q_2$. For D1-w it follows from \preceq -monotonicity of concatenation and from the definition of split. For AS-w, it is implied by the definition of *maxsort*, for V-w by the definition of glb and Lemma 2. \square

Theorem 3. *The algorithm \mathfrak{W} is complete: For every weakening substitution ω' of W there exists $\omega \in \text{weak}(W)$ such that $\omega \leq_{\text{vars}(W)} \omega'$.*

Proof. We construct the desired derivation recursively. The initial pair is $W; \epsilon$. Assume that $W_i; \sigma_i$ belongs to the derivation. Then there exists φ such that $\sigma_i\varphi =_{\text{var}(W)} \omega'$. Moreover, φ is a weakening substitution of $W\sigma_i$ and W_i . We want to extend the derivation with $W_{i+1}; \sigma_{i+1}$ such that $\sigma_{i+1} \leq_{\text{var}(W)} \omega'$. Let $\tilde{r} \rightsquigarrow Q$ be the selected weakening pair in W_i . If $\text{sort}(\tilde{r}) \preceq Q$, we proceed with R-w. If $\text{sort}(\tilde{r}) \not\preceq Q$, we have several cases depending on the shape of \tilde{r} :

- $\tilde{r} = (f(\tilde{t}), \tilde{s})$, $\tilde{s} \neq \epsilon$. \tilde{r} is not ground, otherwise φ would not be its weakening substitution. We select $s.S \in \max(\overline{lf}(Q))$ such that $\text{sort}(f(\tilde{t})\varphi) \preceq s$ and $\text{sort}(\tilde{s}\varphi) \preceq Q$ and proceed with D1-w. Then φ is a weakening substitution of W_{i+1} and $\sigma_{i+1} = \sigma_i \leq_{\text{var}(W)} \omega'$.
- $\tilde{r} = (x, \tilde{s})$, $\tilde{s} \neq \epsilon$. We select a split (Q_1, Q_2) of Q such that $\text{sort}(x\varphi) \preceq Q_1$ and $\text{sort}(\tilde{s}\varphi) \preceq Q_2$ and continue with D2-w. Again, φ is a weakening substitution of W_{i+1} and $\sigma_{i+1} = \sigma_i \leq_{\text{var}(W)} \omega'$.
- $\tilde{r} = f(\tilde{t})$. \tilde{r} is not ground, otherwise φ would not be its weakening substitution. We select R and s such that $R.r \in \text{maxsort}(f)$, $r \preceq Q$, and $\text{sort}(\tilde{t}\varphi) \preceq R$. Such R and r exist, because φ weakens W_i . Then we continue with the rule AS-w. φ is a weakening substitution of W_{i+1} and $\sigma_{i+1} = \sigma_i \leq_{\text{var}(W)} \omega'$.
- $\tilde{r} = x$. Then there exists w with $\text{sort}(w) = \text{glb}(\text{sort}(x), Q)$ such that $\varphi = \{x \mapsto w\}\varphi'$. We select such a w and continue derivation with V-w and the substitution $\vartheta = \{x \mapsto w\}$. Then φ' is a weakening substitution of W_{i+1} and $\sigma_{i+1} = \sigma_i\vartheta \leq_{\text{var}(W)} \omega'$.

Hence, we constructed the desired extension in all the cases. Since the algorithm is terminating, the derivation reaches a success end $\emptyset; \omega$ with the property $\omega \leq_{\text{vars}(W)} \omega'$. \square

Theorem 4 (Soundness). *The rules in \mathfrak{T} are sound.*

Proof. It is easy to check for each rule in \mathfrak{T} that if it performs a transformation $\Gamma, \sigma \Longrightarrow \Delta, \sigma\vartheta$ and φ is a unifier of Δ , then $\vartheta\varphi$ is a unifier of Γ . \square

Theorem 5 (Completeness). *Let Γ be a REOSU problem with a unifier ϑ . Then there exists $\sigma \in \text{comp}(\mathfrak{P}(\Gamma))$ such that $\sigma \leq_{\text{var}(\Gamma)} \vartheta$.*

Proof. We construct recursively the derivation that computes σ . The initial pair in the derivation is $\Gamma; \epsilon$ and $\epsilon \leq_{\text{var}(\Gamma)} \vartheta$. To choose a proper extension, we find all $x \in \text{var}(\Gamma)$ with $x\vartheta = \epsilon$ and make the projection step with the substitution σ_1 whose domain consists of these x 's only. Obviously, $\sigma_1 \leq_{\text{var}(\Gamma)} \vartheta$.

Now assume $\Gamma_n; \sigma_n$ belongs to the derivation. Then $\sigma_n \leq_{\text{var}(\Gamma)} \vartheta$, i.e., there exists φ such that $x\sigma_n\varphi = x\vartheta$ for all $x \in \text{var}(\Gamma)$. Moreover, it is easy to see that φ is a unifier of both $\Gamma\sigma_n$ and Γ_n and $x\varphi \neq \epsilon$ for any x . We want to extend the derivation with $\Gamma_{n+1}, \sigma_{n+1}$ such that $\sigma_{n+1} \leq_{\text{var}(\Gamma)} \vartheta$. Let $\tilde{t} \doteq \tilde{s}$ be the selected equation in Γ_n and represent Γ_n as $\{\tilde{t} \doteq \tilde{s}\} \cup \Gamma'_n$. Then we have the following cases:

1. \tilde{t} and \tilde{s} are either identical, or have identical nonempty prefixes, or their first elements are nonvariable terms with the same head. Then $\Gamma_{n+1}; \sigma_{n+1}$ is obtained by the rules \top , $\top\text{P}$, or D , respectively. Hence, $\sigma_{n+1} = \sigma_n \leq_{\text{var}(\Gamma)} \vartheta$.
2. The first element of \tilde{s} is a variable x , while \tilde{t} does not start with a variable. Then we apply the rule O and get $\sigma_{n+1} = \sigma_n \leq_{\text{var}(\Gamma)} \vartheta$.
3. The first element of \tilde{t} is a variable x , while \tilde{s} does not start with a variable. Since φ is a unifier of Γ_n and does not map x to ϵ , we have either $x\varphi = s\varphi$ or $x\varphi = (s\varphi, \tilde{s}')$ where s is the first element of \tilde{s} and $\tilde{s}' \neq \epsilon$. In the first case, $\text{sort}(s\varphi) \preceq \text{sort}(x)$, i.e., φ involves weakening of $\text{sort}(s)$ to $\text{sort}(x)$. We single out this weakening substitution ω from φ and extend the derivation with the rule WkE1 and substitution $\omega \cup \{x \mapsto s\omega\}$. In the second case we have $\text{sort}(s\varphi).\text{sort}(\tilde{s}') \preceq \text{sort}(x)$. Since $\text{sort}(s)$ is basic sort, there exists $(r, R) \in \text{lf}(\text{R})$ such that $\text{sort}(s\varphi) \preceq r$ and $\text{sort}(\tilde{s}') \preceq R$. Hence, φ involves weakening of $\text{sort}(s)$ to r . Therefore, extracting the weakening substitution ω from φ , we extend the derivation by WkWd1 and $\omega \cup \{x \mapsto (s\omega, z)\}$, where z is fresh with $\text{sort}(z) = R$. In either case $\sigma_{n+1} \leq_{\text{var}(\Gamma)} \vartheta$.
4. The first element of \tilde{t} is a variable x and the first element of \tilde{s} is another variable y . There are the following alternatives: $x\varphi = y\varphi$, $x\varphi = (y\varphi, \tilde{s}')$, or $y\varphi = (x\varphi, \tilde{t}')$, where $\tilde{s}' \neq \epsilon$ and $\tilde{t}' \neq \epsilon$. In the first case, φ also involves weakening for x and y and we proceed with WkE2 with the corresponding weakening substitution. In the second case $\text{sort}(y\varphi) \preceq S$ and $\text{sort}(\tilde{s}') \preceq R$ for a split (S, R) of $\text{sort}(x)$. We choose such a split and proceed with the rule WkWd2 together with a properly chosen substitution $\{x \mapsto (w, z), y \mapsto w\}$. The third case is analogous to the second one. In all the cases we have $\sigma_{n+1} \leq_{\text{var}(\Gamma)} \vartheta$.

The second step is to show that this sequence terminates. We define inductively the size of a term t , sequence of terms \tilde{t} , and a substitution σ with respect to a substitution ϑ as follows: $|x|_\vartheta = 0$, if $x\vartheta = \epsilon$, otherwise $|x|_\vartheta = 1$. $|f(\tilde{t})|_\vartheta = |\tilde{t}|_\vartheta + 2$, $|(t_1, \dots, t_n)|_\vartheta = |t_1|_\vartheta + \dots + |t_n|_\vartheta$, and $|\sigma|_\vartheta = \sum_{x \in \text{dom}(\sigma)} |x\sigma|_\vartheta$, where $\text{dom}(\sigma)$

is the domain of σ . Given Γ and ϑ , we define the size of $\Gamma_i; \sigma_i$ as the quadruple $|\Gamma_i; \sigma_i| = (k, l, m, n)$ where

- k is the number of distinct variables in Γ_i ;
- $l = |\vartheta|_\varepsilon - |\sigma_i|_{\text{var}(\Gamma)}|_\vartheta$, where $\sigma_i|_{\text{var}(\Gamma)}$ is the restriction of σ_i on the set of variables on Γ ;
- m is the multiset $\cup_{\tilde{t} \doteq \tilde{s} \in \Gamma_i} \{|\tilde{t}|_\vartheta, |\tilde{s}|_\vartheta\}$;
- n is the number of equations of the form $(t, \tilde{t}) \doteq (x, \tilde{s})$, $t \notin \mathcal{V}$ in Γ_i .

The sizes are compared lexicographically. The ordering is well-founded.

The projection rule is applied only once and does not increase the size of the unification problem/substitution pairs it operates on. The other rules strictly decrease the size: **T**, **TP**, **D** decrease m and do not increase k and l . **O** decreases n without increasing the others. **WkE1** and **WkE2** decrease k . **WkWd1**, **WkWd2**, and **WkWd3** decrease l and do not increase k . Hence, the derivation we have constructed above terminates. \square

A Machine Checked Model of MGU Axioms: Applications of Finite Maps and Functional Induction

Sunil Kothari and James Caldwell *

Department of Computer Science,
University of Wyoming, USA
{skothari,jlc}@cs.uwo.edu

Abstract. The most general unifier (MGU) of a pair of terms can be specified by four axioms. In this paper we generalize the standard presentation of the axioms to specify the MGU of a list of equational constraints and we formally verify that the unification algorithm satisfies the axioms. Our constraints are equalities between terms in a language of simple types. We model substitutions as finite maps from the Coq library *Coq.FSets.FMapInterface*. Since the unification algorithm is general recursive, we show termination using a lexicographic ordering on lists of constraints. Coq’s method of functional induction is the main proof technique used in proving the axioms.

1 Introduction

As a step toward a comprehensive library of theorems about unification and substitution, we verify the unification algorithm over a language of simple types. We take the axioms presented in [UN09] as our specification and show that the first-order unification algorithm is a model of the axioms. In the formalization we represent substitutions using Coq’s finite map library. This verification is a step toward a formal verification of an extended version of Wand’s constraint based type reconstruction algorithm [KC08]. The main idea behind our approach there is to have a multi-phase unification in the constraint solving phase. By formalizing the first-order unification, we will be able to extend the first-order unification to this multi-phase unification. We believe that the verification described here may be of interest in and of itself to researchers in the unification community.

In recent literature on machine certified proof of correctness of type inference algorithms (mostly on substitution-based type reconstruction algorithms), the most general unifier is axiomatized by a set of four axioms. In this paper, we follow Urban and Nipkow’s [UN09] axioms.

- (i) $mgu\ \sigma\ (\tau_1 \stackrel{e}{=} \tau_2) \Rightarrow \sigma(\tau_1) = \sigma(\tau_2)$
- (ii) $mgu\ \sigma\ (\tau_1 \stackrel{e}{=} \tau_2) \wedge \sigma'(\tau_1) = \sigma'(\tau_2) \Rightarrow \exists \delta. \sigma' \approx \sigma \circ \delta$
- (iii) $mgu\ \sigma\ (\tau_1 \stackrel{e}{=} \tau_2) \Rightarrow FTVS(\sigma) \subseteq FVC(\tau_1 \stackrel{e}{=} \tau_2)$
- (iv) $\sigma(\tau_1) = \sigma(\tau_2) \Rightarrow \exists \sigma'. mgu\ \sigma'(\tau_1 \stackrel{e}{=} \tau_2)$

* The work of the authors was partially supported by NSF 0613919.

The FTVS and FVC above refer to the free type variables and free variables of a constraint respectively. They are defined later in Section 2.

We give an axiomatic presentation of substitutions and provide a model using substitutions formalized with the Coq’s Finite map library. Using this presentation of substitutions, we prove the correctness of first order unification by showing that the unification algorithm satisfies the four axioms. Since the unification algorithm is not structurally recursive, we have to also prove the termination of the first-order unification algorithm by giving a measure and showing that it reduces on each recursive call. The entire verification is done in Coq [Cdt07], a theorem prover based on calculus of inductive constructions [CH88].

The rest of this paper is organized as follows: Section 2 introduces the concepts and terminologies needed for this paper and includes a description of substitutions as finite functions. Section 3 describes the formalization of a first-order unification algorithm in Coq. Section 4 describes the proof that the unification algorithm satisfies the four axioms and presents a number of supporting lemmas. Section 5 summarizes our current work and mentions further work.

2 Types and Substitutions

2.1 Types

Unification is implemented here over a language of types for (untyped) lambda terms. The language of types is given by the following grammar:

$$\tau ::= \text{TyVar } x \mid \tau_1 \rightarrow \tau_2$$

where $x \in \text{Var}$ is a variable and $\tau_1, \tau_2 \in \tau$ are type terms.

Thus, a type is either a type variable or a function type.

We have adopted the following conventions in this paper: atomic types (of the form $\text{TyVar } x$) are denoted by α, β, α' etc.; compound types by τ, τ', τ_1 etc.; substitutions by $\sigma, \sigma', \sigma_1$ etc. By convention, the type constructor \rightarrow associates to the right. List append is denoted by $++$. Small finite substitutions will be represented using the usual (enumerative) set notation. For example, a substitution that binds x to τ and y to τ' is denoted as $\{x \mapsto \tau, y \mapsto \tau'\}$. When necessary we follow Coq’s namespace conventions; every library function has a qualifier which denotes the library it belongs to. For example, $M.map$ is a function from the finite maps library, whereas $List.map$ is a function from the list library.

The work described here is being extended to the polymorphic case and so the language of types will be extended to include universally quantified type variables. Anticipating this, although all type variables occurring in types as defined here are free, we define the list of *free variables of a type* (FTV) as:

$$\begin{aligned} \text{FTV } (\text{TyVar } x) &\stackrel{\text{def}}{=} [x] \\ \text{FTV } (\tau \rightarrow \tau') &\stackrel{\text{def}}{=} \text{FTV } (\tau) ++ \text{FTV } (\tau') \end{aligned}$$

We also have a notion of equational constraints of the form $\tau \stackrel{e}{=} \tau'$. The list of *free variables of a constraint list*, denoted by FVC , is given as:

$$\begin{aligned} \text{FVC } [] &\stackrel{\text{def}}{=} [] \\ \text{FVC } ((\tau_1 \stackrel{e}{=} \tau_2) :: \mathbb{C}) &\stackrel{\text{def}}{=} \text{FTV}(\tau_1) ++ \text{FTV}(\tau_2) ++ \text{FVC}(\mathbb{C}) \end{aligned}$$

2.2 Substitutions

Substitutions are finite functions mapping type variables to types. Application of a substitution to a type is defined as:

$$\begin{aligned} \sigma(\text{TyVar}(x)) &\stackrel{\text{def}}{=} \text{if } \langle x, \tau \rangle \in \sigma \text{ then } \tau \text{ else } \text{TyVar}(x) \\ \sigma(\tau_1 \rightarrow \tau_2) &\stackrel{\text{def}}{=} \sigma(\tau_1) \rightarrow \sigma(\tau_2) \end{aligned}$$

Thus, if a variable x is not in the domain of the substitution, it lifts that variable to $\text{TyVar}(x)$. Application of a substitution to a constraint is defined similarly:

$$\sigma(\tau_1 \stackrel{e}{=} \tau_2) \stackrel{\text{def}}{=} \sigma(\tau_1) \stackrel{e}{=} \sigma(\tau_2)$$

Since substitutions are functions their equality is extensional; they are equal if they behave the same on all type variables.

$$\sigma \approx \sigma' \stackrel{\text{def}}{=} \forall \alpha. \sigma(\alpha) = \sigma'(\alpha)$$

Two type terms τ_1 and τ_2 are *unifiable* if there exists a substitution σ such that $\sigma(\tau_1) = \sigma(\tau_2)$. In such a case, σ is called a *unifier*. More formally, we denote solvability of a constraint by \models (read “solves”). We write $\sigma \models (\tau_1 \stackrel{e}{=} \tau_2)$, if $\sigma(\tau_1) = \sigma(\tau_2)$. We extend the solvability notion to a list of constraints and we write $\sigma \models \mathbb{C}$ if and only if for every $c \in \mathbb{C}$, $\sigma \models c$. A unifier σ is the *most general unifier* if there is a substitution σ' such that for any other unifier σ'' , $\sigma \circ \sigma' \approx \sigma''$. The substitution composition operator, \circ , is defined in the next section.

2.3 Implementing Substitutions as Finite Maps

The representation of substitutions plays an important role in the formalization exercise. In the verification literature, substitutions have been represented as functions, a list of pairs, and a set of pairs. We represent substitutions as finite functions (a.k.a finite maps in Coq). The literature on representing substitutions as finite maps is sparse. We use the Coq finite map library *Coq.FSets.FMapInterface*, which provides an axiomatic presentation of finite maps and a number of supporting implementations. However, it does not provide an induction principle for finite maps, and forward reasoning is often needed to use the library. The fact that we were able to reason about substitution composition without using an induction principle explains the power and expressiveness of the existing library. We found we did not need induction to reason on finite maps, though there are

natural induction principles we might have proved [CS95, MW85]. The most recent release of the library (ver. 8.2) supports one.

To consider the *domain* and *range* of a finite function (and this is the key feature of the function being finite), we use the finite map library function `M.elements`. `M.elements(σ)` returns the list of pairs (key-value pairs) corresponding to the finite map σ . The domain and the range of a substitution are defined as:

$$\begin{aligned} \text{dom}(\sigma) &\stackrel{\text{def}}{=} \text{List.map } (\lambda t.\text{fst } (t)) (\text{M.elements } (\sigma)) \\ \text{range}(\sigma) &\stackrel{\text{def}}{=} \text{List.flat_map } (\lambda t.\text{FTV } (\text{snd } (t))) (\text{M.elements } (\sigma)) \end{aligned}$$

The function `List.flat_map`, also known as `mapcan` in LISP and `concatMap` in Haskell, is defined in the Coq library `Coq.List.List` as:

$$\begin{aligned} \text{flat_map } f [] &\stackrel{\text{def}}{=} [] \\ \text{flat_map } f h :: t &\stackrel{\text{def}}{=} (f h) ++ \text{flat_map } f t \end{aligned}$$

The free type variables of a substitution, denoted by `FTVS`, is defined in terms of domain and range of a substitution as:

$$\text{FTVS}(\sigma) \stackrel{\text{def}}{=} \text{dom}(\sigma) ++ \text{range}(\sigma)$$

Applying a substitution σ' to a substitution σ means applying σ' to the range elements of σ .

$$\sigma'(\sigma) \stackrel{\text{def}}{=} \text{M.map } (\lambda t.\sigma'(t)) \sigma$$

The function `subst_diff` is needed to define substitution composition, and is defined as:

$$\text{subst_diff } \sigma \sigma' \stackrel{\text{def}}{=} \text{M.map2 } \text{choose_subst } \sigma \sigma'$$

In the above definition, the function `M.map2` is defined in Coq library as the function that takes two maps σ and σ' , and creates a map whose binding belongs to either σ or σ' based on the function `choose_subst`, which determines the presence and value for a key (absence of a value is denoted by `None`). The values in the first map are preferred over the values in the second map for a particular key.

$$\begin{aligned} \text{choose_subst } T1 T2 &\stackrel{\text{def}}{=} \text{match } (T1, T2) \text{ with} \\ &\quad | \text{Some } T3, \text{Some } T4 \Rightarrow \text{Some } T3 \\ &\quad | \text{Some } T3, \text{None} \Rightarrow \text{Some } T3 \\ &\quad | \text{None}, \text{Some } T4 \Rightarrow \text{Some } T4 \\ &\quad | \text{None}, \text{None} \Rightarrow \text{None} \end{aligned}$$

Finally, substitution composition is defined as:

$$\sigma \circ \sigma' \stackrel{\text{def}}{=} \text{subst_diff } \sigma'(\sigma) \sigma'$$

Substitution composition application to a type has the following property:

Theorem 1. [Composition Apply]

$$\forall \tau. (\sigma \circ \sigma')(\tau) = \sigma'(\sigma(\tau))$$

Proof. By induction on the type τ followed by case analysis on the binding's occurrence in the composed substitution and in the individual substitutions.

Interestingly, the base case (when τ is a type variable) is harder than the inductive case (when τ is a compound type). Incidentally, the same theorem has been formalized in Coq [DM99], where substitutions are represented as a list of pairs, but required 600 proof steps. We proved in about 100 proof steps.

3 Unification

3.1 The Algorithm

We use the following standard presentation of the first-order unification algorithm.

$$\begin{array}{lll} \text{unify } (\alpha \stackrel{e}{=} \alpha) :: \mathbb{C} & \stackrel{\text{def}}{=} & \text{unify } \mathbb{C} \\ \text{unify } (\alpha \stackrel{e}{=} \beta) :: \mathbb{C} & \stackrel{\text{def}}{=} & \{\alpha \mapsto \beta\} \circ \text{unify } (\{\alpha \mapsto \beta\} \mathbb{C}) \\ \text{unify } (\alpha \stackrel{e}{=} \tau) :: \mathbb{C} & \stackrel{\text{def}}{=} & \text{if } \alpha \text{ occurs in } \tau \text{ then Fail else } \{\alpha \mapsto \tau\} \circ \text{unify } (\{\alpha \mapsto \tau\} \mathbb{C}) \\ \text{unify } (\tau \stackrel{e}{=} \alpha) :: \mathbb{C} & \stackrel{\text{def}}{=} & \text{if } \alpha \text{ occurs in } \tau \text{ then Fail else } \{\alpha \mapsto \tau\} \circ \text{unify } (\{\alpha \mapsto \tau\} \mathbb{C}) \\ \text{unify } (\tau_1 \rightarrow \tau_2 & \stackrel{\text{def}}{=} & \text{unify } (\tau_1 \stackrel{e}{=} \tau_3 :: \tau_2 \stackrel{e}{=} \tau_4 :: \mathbb{C}) \\ & \stackrel{e}{=} \tau_3 \rightarrow \tau_4) :: \mathbb{C} & \\ \text{unify } [] & \stackrel{\text{def}}{=} & Id \end{array}$$

The algorithm presented above is still not quite ready for formalization since we have not represented failure. Coq provides an *option* type (also available in OCaml as a standard data type) to allow for failure.

$$\text{Inductive option } (A : \text{Set}) : \text{Set} := \text{Some } (_ : A) \mid \text{None}.$$

We use the option `None` to indicate failure and in the result `Some(σ)`, σ is the resulting substitution. The unification algorithm is fully formalized as shown in Appendix 7.1.

The above presentation of the unification algorithm is general recursive, *i.e.* the recursive call is not necessarily on a structurally smaller argument. Various papers have described the non-structural recursion aspect of first-order unification [Bov01, McB03]. To allow Coq to accept our definition of unification, we have to either give a measure that shows that recursive argument is smaller or give a well-founded ordering relation. We chose the latter. The annotation `{wf meaPairMLt}` in the specification is precisely that. The advantage of specifying the unification algorithm as shown above is that we get an induction principle for free. This induction principle will be used later in a Coq tactic named as `functional induction` for the axiom proofs. We will have more to say about the induction principle and the tactic later in Section 4.1.

3.2 Termination

Since the unification algorithm is general recursive, we need to give an ordering that is well-founded. We use the lexicographic ordering (\prec_3) on the triple (see below). The lexicographic ordering on the two triples $\langle n_1, n_2, n_3 \rangle$ and $\langle m_1, m_2, m_3 \rangle$ is defined as

$$\langle n_1, n_2, n_3 \rangle \prec_3 \langle m_1, m_2, m_3 \rangle \stackrel{\text{def}}{=} (n_1 < m_1) \vee (n_1 = m_1 \wedge n_2 < m_2) \vee (n_1 = m_1 \wedge n_2 = m_2 \wedge n_3 < m_3),$$

where $<$ and $=$ are the ordinary less-than inequality and equality on natural numbers. Our triple is similar to the triple proposed by others [Bov01, BS01, Apt03], but a little simpler. The triple is $\langle |C_{FVC}|, |C_{\rightarrow}|, |C| \rangle$, where

- $|C_{FVC}|$ is the number of *unique* free variables in a constraint list;
- $|C_{\rightarrow}|$ is the total number of arrows in the constraint list;
- $|C|$ is the length of the constraint list.

Table 1 shows how these components vary depending on the constraint at the

Original call	Recursive call	Conditions, if any	$ C_{FVC} $	$ C_{\rightarrow} $	$ C $
$(\alpha \stackrel{e}{=} \alpha) :: \mathbb{C}$	\mathbb{C}	$\alpha \in (\text{FVC } \mathbb{C})$	-	-	\downarrow
$(\alpha \stackrel{e}{=} \alpha) :: \mathbb{C}$	\mathbb{C}	$\alpha \notin (\text{FVC } \mathbb{C})$	\downarrow	-	\downarrow
$(\alpha \stackrel{e}{=} \beta) :: \mathbb{C}$	$\{\alpha \mapsto \beta\}\mathbb{C}$	$\alpha \neq \beta$	\downarrow	-	\downarrow
$(\alpha \stackrel{e}{=} \tau) :: \mathbb{C}$	$\{\alpha \mapsto \tau\}\mathbb{C}$	$\alpha \notin (\text{FTV } \tau) \wedge \alpha \notin (\text{FVC } \mathbb{C})$	\downarrow	\downarrow	\downarrow
$(\alpha \stackrel{e}{=} \tau) :: \mathbb{C}$	$\{\alpha \mapsto \tau\}\mathbb{C}$	$\alpha \notin (\text{FTV } \tau) \wedge \alpha \in (\text{FVC } \mathbb{C})$	\downarrow	\uparrow	\downarrow
$(\tau \stackrel{e}{=} \alpha) :: \mathbb{C}$	$\{\alpha \mapsto \tau\}\mathbb{C}$	$\alpha \notin (\text{FTV } \tau) \wedge \alpha \notin (\text{FVC } \mathbb{C})$	\downarrow	\downarrow	\downarrow
$(\tau \stackrel{e}{=} \alpha) :: \mathbb{C}$	$\{\alpha \mapsto \tau\}\mathbb{C}$	$\alpha \notin (\text{FTV } \tau) \wedge \alpha \in (\text{FVC } \mathbb{C})$	\downarrow	\uparrow	\downarrow
$(\tau_1 \rightarrow \tau_2 \stackrel{e}{=} \tau_3 \rightarrow \tau_4) :: \mathbb{C}$	$(\tau_1 \stackrel{e}{=} \tau_3) :: \mathbb{C}$	None	-	\downarrow	\uparrow

Table 1. Variation of termination measure components on the recursive call

head of the constraint list. The table closely follows the reasoning we used to satisfy the proof obligations (shown in the Appendix 7.3) generated by the above specification. We use $-$, \uparrow , \downarrow to denote whether the component is unchanged, increased or decreased, respectively. We could have used the finite sets here (for counting the unique free variables of a constraint list), but we went ahead with the unique lists (referred to as `NoDup` in Coq). We found the existing Coq list library offering plenty of support for lists in general, and unique lists in particular. Coq also provide a library to reason about lists modulo permutation. Together we were able to reason with the lists as finite sets. We also had to use the following lemma mentioned in the formalization of Sudoku puzzle by Laurent Théry [The06].

Lemma 1. [List subset membership and unique list length]

$$\forall l, l' : \text{list } D, \text{NoDup } l \Rightarrow \text{NoDup } l' \Rightarrow \text{List.incl } l \ l' \Rightarrow \neg \text{List.incl } l' \ l \Rightarrow (\text{List.length } l) < (\text{List.length } l')$$

The above lemma was essential for our termination proofs, since the Coq's List library does not provide any axioms on unique list inequalities.

4 MGU axioms

Note that each of the axioms, introduced earlier in Section 1, characterizes the MGU behavior on a pair of terms (a single constraint). In our verification, we will lift these axioms to a constraint list. This is necessary since constraint-based type reconstruction algorithms solve all the constraints in one go. The new axioms are:

- (i) $\text{unify } \mathbb{C} = \text{Some } \sigma \Rightarrow \sigma \models \mathbb{C}$
- (ii) $(\text{unify } \mathbb{C} = \text{Some } \sigma \wedge \sigma' \models \mathbb{C}) \Rightarrow \exists \sigma''. \sigma' \approx \sigma \circ \sigma''$
- (iii) $\text{unify } \mathbb{C} = \text{Some } \sigma \Rightarrow \text{FTVS}(\sigma) \subseteq \text{FVC}(\mathbb{C})$
- (iv) $\sigma \models \mathbb{C} \Rightarrow \exists \sigma'. \text{unify } \mathbb{C} = \text{Some } \sigma'$

We can now go into the proof of the above axioms. The underlying theme in all of the proofs below is the use of functional induction tactic in Coq. The tactic ensures that we have the right induction hypothesis, when we want to prove a property for the inductive case. We mention this general technique next.

4.1 Functional Induction in Coq

In Coq, the functional induction technique generates/uses the induction principle which is generated for the definitions defined using the `Function` keyword. The induction principle is shown in the Appendix 7.2. The induction principle is rather long because the actual specification is verbose and also because of the cases involved; there are 3 cases with 3 outcomes each.

In the next few sections, we mention only the important lemmas involved in the proofs of each of the axioms. For many of these lemmas, we give the main technique involved in the proofs.

4.2 Axiom i

Lemma 2. [Satisfy and compose subst]

$$\forall x. \forall \mathbb{C}. \forall \sigma. \forall \tau. \sigma \models \{x \mapsto \tau\}(\mathbb{C}) \Rightarrow (\{x \mapsto \tau\} \circ \sigma) \models \mathbb{C}$$

Proof. By induction on \mathbb{C} .

Lemma 3. [Membership in a constraint list invariant under substitution]

$$\forall x. \forall \mathbb{C}. \forall \tau, \tau_1, \tau_2. (\tau_1 \stackrel{e}{=} \tau_2) \in \mathbb{C} \Rightarrow \{x \mapsto \tau\}(\tau_1 \stackrel{e}{=} \tau_2) \in \{x \mapsto \tau\}(\mathbb{C})$$

Proof. By induction on τ .

Lemma 4. [Constraint satisfaction and membership in a list]

$$\forall \mathbb{C}. \forall \sigma. \forall \tau, \tau'. (\tau \stackrel{e}{=} \tau') \in \mathbb{C} \wedge \text{unify } \mathbb{C} = \text{Some } \sigma \Rightarrow \sigma \models (\tau \stackrel{e}{=} \tau')$$

Proof. By functional induction on $\text{unify } \mathbb{C}$ and theorem 1.

Theorem 2. $\forall \sigma. \forall \mathbb{C}. \text{unify } \mathbb{C} = \text{Some } \sigma \Rightarrow \sigma \models \mathbb{C}$

Proof. By functional induction on $\text{unify } \mathbb{C}$ and by theorem 1 and lemma 3.

4.3 Axiom ii

Lemma 5. [Equal substitution instance for singleton subst]

$$\forall \sigma. \forall \alpha. \forall \tau, \tau'. \alpha \notin (\text{FTV } \tau) \wedge \sigma(\alpha) = \sigma(\tau) \Rightarrow \sigma(\tau') = \sigma(\{\alpha \mapsto \tau\}(\tau'))$$

Proof. By induction on τ' .

Lemma 6. [Constraint satisfaction extended to a substitution instance of a constraint]

$$\forall \mathbb{C}. \forall \sigma. \forall \alpha. \forall \tau. \sigma \models \mathbb{C} \wedge \alpha \notin (\text{FTV } \tau) \wedge \sigma(\alpha) = \sigma(\tau) \Rightarrow \sigma \models \{\alpha \mapsto \tau\}(\mathbb{C})$$

Proof. By induction on \mathbb{C} and by lemma 5.

The following lemma lifts the extensional equality on type variables to any type.

Lemma 7. [Extensionality extended to any type]

$$\forall \sigma, \sigma'. \forall \alpha. \sigma(\alpha) = \sigma'(\alpha) \Leftrightarrow \forall \tau. \sigma(\tau) = \sigma'(\tau)$$

Proof. (\Rightarrow) By induction on τ .

(\Leftarrow) Trivial.

Theorem 3. $\forall \sigma. \forall \mathbb{C}. (\text{unify } \mathbb{C} = \text{Some } \sigma \wedge \sigma' \models \mathbb{C}) \Rightarrow \exists \sigma''. \sigma' \approx \sigma \circ \sigma''$

Proof. By functional induction on $\text{unify } \mathbb{C}$, and by theorem 1 and lemma 6, 7.

4.4 Axiom iii

Lemma 8. [Compose and domain membership]

$$\begin{aligned} &\forall \alpha, \alpha'. \forall \tau. \forall \sigma. \alpha' \in \text{dom_subst } (\{\alpha \mapsto \tau\} \circ \sigma) \\ &\Rightarrow \alpha' \in \text{dom_subst } \{\alpha \mapsto \tau\} \vee \alpha' \in \text{dom_subst } \sigma \end{aligned}$$

Lemma 9. [Compose and range membership]

$$\begin{aligned} &\forall \alpha, \alpha'. \forall \tau. \forall \sigma. (\alpha \notin (\text{FTV } \tau) \wedge \alpha' \in \text{range_subst } (\{\alpha \mapsto \tau\} \circ \sigma)) \\ &\Rightarrow \alpha' \in \text{range_subst } \{\alpha \mapsto \tau\} \vee \alpha' \in \text{range_subst } \sigma \end{aligned}$$

Without going into details, the following lemma helps us in proving Lemma 9. Note that the definition of `range_subst` contains references to higher order functions `M.map2` and this lemma helps in not having to reason about `M.map2`.

Lemma 10. [Subst range abstraction]

$$\forall \alpha. \forall \sigma. \alpha \in \text{range_subst } (\sigma) \Leftrightarrow \exists \alpha'. \alpha' \in \text{dom_subst } (\sigma) \wedge \alpha \in \sigma(\alpha')$$

Theorem 4. $\forall \sigma, \sigma'. \forall \mathbb{C}. \text{unify } \mathbb{C} = \text{Some } \sigma \Rightarrow \text{FTVS}(\sigma) \subseteq \text{FVC}(\mathbb{C})$

Proof. By functional induction on $\text{unify } \mathbb{C}$ and lemmas 8, 9.

4.5 Axiom iv

This axiom requires the notion of subterms, which we define below:

$$\begin{aligned} \text{subterms } \alpha &= [] \\ \text{subterms } (\tau_1 \rightarrow \tau_2) &= \tau_1 :: \tau_2 :: (\text{subterms } \tau_1) ++ (\text{subterms } \tau_2) \end{aligned}$$

Then we can define what it means for a term to be contained in another term.

Lemma 11. [Containment]

$$\forall \tau, \tau'. \tau \in (\text{subterms } \tau') \Rightarrow \forall \tau''. \tau'' \in (\text{subterms } \tau) \Rightarrow \tau'' \in (\text{subterms } \tau')$$

Proof. By induction on τ' .

A somewhat related lemma is used to show well foundedness of types.

Lemma 12. [Well founded types]

$$\forall \tau. \neg \tau \in (\text{subterms } \tau)$$

Proof. By induction on τ and by lemma 11.

Lemma 13. [Member subterms unequal]

$$\forall \tau, \tau'. \tau \in (\text{subterms } \tau') \Rightarrow \tau \neq \tau'$$

Proof. By case analysis on $\tau = \tau'$ and by lemma 12.

The following obvious but powerful lemma helps in proving the axiom:

Lemma 14. [Member subterms and apply subst]

$$\forall \sigma. \forall \alpha. \forall \tau. \alpha \in (\text{subterms } \tau) \Rightarrow \sigma(\alpha) \neq \sigma(\tau)$$

Proof. By induction on τ and by lemma 13.

Lemma 15. [Member arrow and subterms]

$$\begin{aligned} \forall \sigma. \forall \alpha. \forall \tau_1, \tau_2. \text{member } \alpha \text{ (FTV } \tau_1) = \text{true} \vee \text{member } \alpha \text{ (FTV } \tau_2) = \text{true} \\ \Rightarrow \alpha \in \text{subterms}(\tau_1 \rightarrow \tau_2) \end{aligned}$$

Proof. By induction on τ_1 , followed by induction on τ_2 .

A corollary from the above two gives us the required lemma.

Corollary 1. [Member apply subst unequal]

$$\begin{aligned} \forall \sigma. \forall \alpha. \forall \tau_1, \tau_2. \text{member } \alpha \text{ (FTV } \tau_1) = \text{true} \vee \text{member } \alpha \text{ (FTV } \tau_2) = \text{true} \\ \Rightarrow \sigma(\alpha) \neq \sigma(\tau_1 \rightarrow \tau_2) \end{aligned}$$

Proof. By lemma 14 and 15.

Theorem 5. $\forall \sigma. \forall \mathbb{C}. \sigma \models \mathbb{C} \Rightarrow \exists \sigma'. \text{unify } \mathbb{C} = \text{Some } \sigma'$

Proof. By functional induction on $\text{unify } \mathbb{C}$ and lemma 6 and corollary 1.

5 Related Work and Conclusions

5.1 Related Work

There are formalizations of the unification algorithm in a number of different theorem provers [Bla08, Pau85, Rou94]. Unification is fundamentally used in type inference. Many of the existing verifications of type inference algorithms [DM99, NN99, NN96, UN09] axiomatize the behavior of the MGU rather than provide an implementation as we do here.

We comment on the implementation in the CoLoR library [BDCG⁺06]. CoLoR is an extensive and very successful library supporting reasoning about termination and rewriting. Their Coq implementation of the unification algorithm was recently released [Bla08]. Our implementation differs from theirs in a number of ways. Perhaps the most significant difference is that we represent substitutions as finite maps, whereas in CoLoR the substitutions are represented by functions from type variables to a generalized term structure. The axioms verified here are not explicitly verified in CoLoR, however their library could serve as a basis for doing so. We believe that the lemmas supporting our verification could be translated into their more general framework but that the proofs would be significantly different because we use functional induction which follows the structure of our algorithm. The unification algorithm in CoLoR is specified in a significantly different style (as an iterated step function).

5.2 Future Work

The current work serves as a first step in verification of various constraint-based type reconstruction algorithms. The entire formalization is done in Coq 8.1.pl3 version in about 4400 lines of specifications and tactics, and is available online at <http://www.cs.uwo.edu/~skothari>. The choice of representing substitutions as finite functions was crucial. An induction principle for finite maps would have been useful for some of the proofs, and indeed there is a new version of the library in Coq 8.2 which provides this. We believe that this entire work should lead to a better understanding and appreciation of the finite maps library in Coq. These proofs are part of a larger effort to verify our extended version of Wand's algorithm, which handles the polymorphic let construct [Kot07, KC08].

6 Acknowledgments

We would like to thank Santiago Zanella (INRIA - Sophia Antipolis) for showing us how to encode lexicographic ordering for 3-tuples in Coq. Thanks also to Frederic Blanqui for answering our queries regarding the new release of CoLoR library. We are also thankful to Laurent Théry for making his Coq formulation of Sudoku available on the web, and to Stéphane Lescuyer and other Coq-club members for answering our queries on the Coq-club mailing list. We also want to thank anonymous referees for their detailed comments and suggestions (on an earlier draft of this paper), which greatly improved the presentation of this paper.

References

- [Apt03] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
- [BDCG⁺06] F. Blanqui, W. Delobel, S. Coupet-Grimal, S. Hinderer, and A. Korprowski. CoLoR, a Coq library on rewriting and termination. In *8th International Workshop on Termination (WST '06)*, pages 69–73, 2006.
- [Bla08] Frederic Blanqui. CoLor, a Coq library on rewriting and termination., January 2008. <http://color.inria.fr/doc/CoLoR.Term.WithArity.AUnif.html>.
- [Bov01] Ana Bove. Simple General Recursion in Type Theory. *Nordic J. of Computing*, 8(1):22–42, 2001.
- [BS01] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.
- [Cdt07] The Coq development team. *The Coq proof assistant reference manual*. INRIA, LogiCal Project, 2007. Version 8.1.3.
- [CH88] Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.
- [CS95] Graham Collins and Don Syme. A Theory of Finite Maps. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 122–137. Springer-Verlag, 1995.
- [DM99] C. Dubois and V. M. Morain. Certification of a Type Inference Tool for ML: Damas–Milner within Coq. *J. Autom. Reason.*, 23(3):319–346, 1999.
- [KC08] Sunil Kothari and James Caldwell. On Extending Wand’s Type Reconstruction Algorithm to Handle Polymorphic Let. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, pages 254–263. University of Athens, 2008.
- [Kot07] Sunil Kothari. Wand’s Algorithm Extended For The Polymorphic ML-Let. Technical report, University of Wyoming, 2007.
- [McB03] Conor McBride. First-order unification by structural recursion. *J. Funct. Program.*, 13(6):1061–1075, 2003.
- [MW85] Zohar Manna and Richard Waldinger. *The logical basis for computer programming. Volume 1: deductive reasoning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.
- [NN96] Dieter Nazareth and Tobias Nipkow. *Theorem Proving in Higher Order Logics*, volume 1125, chapter Formal Verification of Alg. W: The Monomorphic Case, pages 331–345. Springer Berlin / Heidelberg, 1996.
- [NN99] Wolfgang Naraschewski and Tobias Nipkow. Type Inference Verified: Algorithm W in Isabelle/HOL. *J. Autom. Reason.*, 23(3):299–318, 1999.
- [Pau85] L. C. Paulson. Verifying the Unification Algorithm in LCF. *Sci. of Computer Programming*, 5:143–169, 1985.
- [Rou94] J. Rouyer. *Developpement d’Algorithmes dans le Calcul des Constructions*. PhD thesis, Institut National Polytechnique de Lorraine, Nancy, France, 1994.
- [The06] Laurent Thery. Sudoku in Coq. 2006.
- [UN09] Christian Urban and Tobias Nipkow. *From Semantics to Computer Science*, chapter Nominal verification of algorithm W. Cambridge University Press, Not yet published 2009.

7 Appendix

7.1 First-order unification algorithm's specification in Coq

```

Function unify (c:list constr){wf meaPairMLt} :(option (M.t type)) :=
match c with
  nil => Some (M.empty type)
| h::t => (match h with
  EqCons (TyVar x) (TyVar y) =>
    if eq_dec_stamp x y
    then unify t
    else (match unify (apply_subst_to_constr_list
      (M.add x (TyVar y)
        (M.empty type)) t) with
      Some p => Some (compose_subst
        (M.add x (TyVar y)
          (M.empty type)) p)
      | None => None
    end)
  | EqCons (TyVar x) (Arrow ty3 ty4) =>
    if (member x (FTV ty3)) || (member x (FTV ty4))
    then None
    else (match (unify (apply_subst_to_constr_list
      (M.add x (Arrow ty3 ty4)
        (M.empty type)) t) with
      Some p => Some (compose_subst
        (M.add x (Arrow ty3 ty4)
          (M.empty type)) p)
      | None => None
    end)
  | EqCons (Arrow ty3 ty4)(TyVar x) =>
    if (member x (FTV ty3)) || (member x (FTV ty4))
    then None
    else (match (unify (apply_subst_to_constr_list
      (M.add x (Arrow ty3 ty4)
        (M.empty type))t)) with
      Some p => Some (compose_subst
        (M.add x (Arrow ty3 ty4)
          (M.empty type)) p)
      | None => None
    end )
  | EqCons (Arrow ty3 ty4)(Arrow ty5 ty6)=>
    unify ((EqCons ty3 ty5)::
      ((EqCons ty4 ty6)::t))
end)
end.

```

7.2 Induction principle used in the functional induction

forall P:list constr -> option (M.t type) -> Prop,

```

(forall c:list constr, c = nil -> P nil (Some (M.empty type))) ->
(forall (c:list constr) (h:constr) (t:list constr), c = h::t ->
forall x y:nat, h = EqCons (TyVar x) (TyVar y) ->
forall _x:x = y, eq_dec_stamp x y = left (x <> y) _x ->
  P t (unify t) -> P (EqCons (TyVar x) (TyVar y) :: t) (unify t)) ->
(forall (c:list constr) (h:constr) (t0:list constr), c = h :: t0 ->
forall x y:nat, h = EqCons (TyVar x) (TyVar y) ->
forall _x: x <> y, eq_dec_stamp x y = right (x = y) _x ->
  P (apply_subst_to_constr_list (M.add x (TyVar y) (M.empty type)) t0)
    (unify
      (apply_subst_to_constr_list
        (M.add x (TyVar y) (M.empty type)) t0)) ->
forall p:M.t type,
  unify (apply_subst_to_constr_list
    (M.add x (TyVar y) (M.empty type)) t0) = Some p ->
  P (EqCons (TyVar x) (TyVar y)::t0)
    (Some (compose_subst (M.add x (TyVar y) (M.empty type)) p))) ->
(forall (c:list constr) (h:constr) (t0:list constr), c = h::t0 ->
forall x y:nat, h = EqCons (TyVar x) (TyVar y) ->
forall _x:x <> y, eq_dec_stamp x y = right (x = y) _x ->
  P (apply_subst_to_constr_list (M.add x (TyVar y) (M.empty type)) t0)
    (unify
      (apply_subst_to_constr_list
        (M.add x (TyVar y) (M.empty type)) t0)) ->
unify (apply_subst_to_constr_list
  (M.add x (TyVar y) (M.empty type)) t0) = None ->
P (EqCons (TyVar x) (TyVar y)::t0) None ->
(forall (c:list constr) (h:constr) (t0:list constr), c = h::t0 ->
forall (x:nat) (ty3 ty4:type), h = EqCons (TyVar x) (Arrow ty3 ty4) ->
  member x (FTV ty3) || member x (FTV ty4) = true ->
  P (EqCons (TyVar x) (Arrow ty3 ty4)::t0) None ->
(forall (c:list constr) (h:constr) (t0:list constr), c = h::t0 ->
forall (x:nat) (ty3 ty4:type),
  h = EqCons (TyVar x) (Arrow ty3 ty4)->
  member x (FTV ty3) || member x (FTV ty4) = false ->
  P (apply_subst_to_constr_list
    (M.add x (Arrow ty3 ty4) (M.empty type)) t0)
    (unify
      (apply_subst_to_constr_list
        (M.add x (Arrow ty3 ty4) (M.empty type)) t0)) ->
forall p : M.t type,
  unify (apply_subst_to_constr_list
    (M.add x (Arrow ty3 ty4) (M.empty type)) t0) = Some p->
  P (EqCons (TyVar x) (Arrow ty3 ty4) :: t0)
    (Some (compose_subst (M.add x (Arrow ty3 ty4) (M.empty type)) p)))->
(forall (c:list constr) (h:constr) (t0:list constr), c = h :: t0 ->
forall (x:nat) (ty3 ty4:type), h = EqCons (TyVar x) (Arrow ty3 ty4)->
  member x (FTV ty3) || member x (FTV ty4) = false ->
  P (apply_subst_to_constr_list
    (M.add x (Arrow ty3 ty4) (M.empty type)) t0)

```

```

(unify
  (apply_subst_to_constr_list
    (M.add x (Arrow ty3 ty4) (M.empty type)) t0)) ->
unify
  (apply_subst_to_constr_list
    (M.add x (Arrow ty3 ty4) (M.empty type)) t0) = None ->
P (EqCons (TyVar x) (Arrow ty3 ty4)::t0) None ->
(forall (c:list constr) (h:constr) (t0:list constr), c = h::t0 ->
 forall (ty3 ty4:type) (x:nat),
  h = EqCons (Arrow ty3 ty4) (TyVar x) ->
 member x (FTV ty3) || member x (FTV ty4) = true ->
 P (EqCons (Arrow ty3 ty4) (TyVar x)::t0) None ->
(forall (c:list constr) (h:constr) (t0:list constr), c = h::t0 ->
 forall (ty3 ty4:type) (x:nat),
  h = EqCons (Arrow ty3 ty4) (TyVar x) ->
 member x (FTV ty3) || member x (FTV ty4) = false ->
 P
  (apply_subst_to_constr_list
    (M.add x (Arrow ty3 ty4) (M.empty type)) t0)
  (unify
    (apply_subst_to_constr_list
      (M.add x (Arrow ty3 ty4) (M.empty type)) t0)) ->
forall p : M.t type,
  unify
    (apply_subst_to_constr_list
      (M.add x (Arrow ty3 ty4) (M.empty type)) t0) = Some p ->
P (EqCons (Arrow ty3 ty4) (TyVar x)::t0)
  (Some (compose_subst
    (M.add x (Arrow ty3 ty4) (M.empty type)) p))) ->
(forall (c:list constr) (h:constr) (t0:list constr), c = h::t0 ->
 forall (ty3 ty4 : type) (x : nat),
  h = EqCons (Arrow ty3 ty4) (TyVar x) ->
 member x (FTV ty3) || member x (FTV ty4) = false ->
 P
  (apply_subst_to_constr_list
    (M.add x (Arrow ty3 ty4) (M.empty type)) t0)
  (unify (apply_subst_to_constr_list
    (M.add x (Arrow ty3 ty4) (M.empty type)) t0)) ->
  unify
    (apply_subst_to_constr_list
      (M.add x (Arrow ty3 ty4) (M.empty type)) t0) = None ->
P (EqCons (Arrow ty3 ty4) (TyVar x)::t0) None ->
(forall (c:list constr) (h:constr) (t:list constr), c = h :: t ->
 forall ty3 ty4 ty5 ty6:type,
  h = EqCons (Arrow ty3 ty4) (Arrow ty5 ty6) ->
 P (EqCons ty3 ty5::EqCons ty4 ty6::t)
  (unify (EqCons ty3 ty5::EqCons ty4 ty6::t)) ->
 P (EqCons (Arrow ty3 ty4) (Arrow ty5 ty6)::t)
  (unify (EqCons ty3 ty5::EqCons ty4 ty6::t))) ->
forall c:list constr, P c (unify c)

```

7.3 Proof Obligations

There are 5 proof obligations related to the 5 recursive call sites in the specification. The sixth proof obligation is to show that the ordering relation is well founded.

```

forall (c:list constr) (h:constr) (t:list constr), c = h::t ->
forall t0 t1:type, h = EqCons t0 t1 ->
forall x:nat, t0 = TyVar x ->
forall y:nat, t1 = TyVar y ->
forall anonymous:x = y, eq_dec_stamp x y = left (x <> y) anonymous ->
meaPairMLt (EqCons (TyVar x) (TyVar y) :: t)
-----
(2/6)
forall (c:list constr) (h:constr) (t:list constr), c = h::t ->
forall t0 t1 : type, h = EqCons t0 t1 ->
forall x : nat, t0 = TyVar x ->
forall y : nat, t1 = TyVar y ->
forall anonymous: x <> y, eq_dec_stamp x y = right (x = y) anonymous ->
meaPairMLt (apply_subst_to_constr_list (M.add x (TyVar y)
(M.empty type)) t)
(EqCons (TyVar x) (TyVar y) :: t)
-----
(3/6)
forall (c:list constr) (h:constr) (t:list constr), c = h::t ->
forall t0 t1 : type, h = EqCons t0 t1 ->
forall x : nat, t0 = TyVar x ->
forall ty3 ty4 : type, t1 = Arrow ty3 ty4 ->
member x (FTV ty3) || member x (FTV ty4) = false ->
meaPairMLt
  (apply_subst_to_constr_list (M.add x (Arrow ty3 ty4)
(M.empty type)) t)
  (EqCons (TyVar x) (Arrow ty3 ty4) :: t)
-----
(4/6)
forall (c:list constr) (h:constr) (t:list constr), c = h::t ->
forall t0 t1 : type, h = EqCons t0 t1 ->
forall ty3 ty4 : type, t0 = Arrow ty3 ty4 ->
forall x : nat, t1 = TyVar x ->
member x (FTV ty3) || member x (FTV ty4) = false ->
meaPairMLt
  (apply_subst_to_constr_list (M.add x (Arrow ty3 ty4)
(M.empty type)) t)
  (EqCons (Arrow ty3 ty4) (TyVar x) :: t)
-----
(5/6)
forall (c:list constr) (h:constr) (t:list constr), c = h::t ->
forall t0 t1 : type, h = EqCons t0 t1 ->
forall ty3 ty4 : type, t0 = Arrow ty3 ty4 ->
forall ty5 ty6 : type, t1 = Arrow ty5 ty6 ->
meaPairMLt (EqCons ty3 ty5 :: EqCons ty4 ty6 :: t)
(EqCons (Arrow ty3 ty4) (Arrow ty5 ty6) :: t)
-----
(6/6)
well_founded meaPairMLt

```

Implementing Rigid E-unification (Extended Abstract)

Michael Franssen (m.franssen@tue.nl)

Eindhoven University of Technology,
Dept. of Mathematics and Computer Science,
Den Dolech 2, 5612 AZ Eindhoven, The Netherlands

Abstract. Rigid E-unification problems arise naturally in automated theorem provers that deal with equality. In this paper we discuss how to implement a rigid E-unifier, focussing on efficiency. We improve the algorithm to compute rigid E-unifiers as proposed by Degtyarev and Voronkov [4] for practical situations where many similar rigid E-unification problems have to be solved. We implement the methods of Comon [1] and Nieuwenhuis [8] to solve symbolic ordering constraints generated by the algorithm. We compute solved forms of constraints without storing intermediate results and without performing any substitutions. Instead of computing an exponential number of simple systems for each solved form, we build a graph from the solved form such that a bounded backtracking algorithm can search for a satisfiable simple system without the need to generate them all.

1 Introduction

In automated theorem provers that allow equalities, a problem that has to be solved is this: given the equalities $s_1 = t_1, \dots, s_n = t_n$, can we derive $s = t$? Often, the equations contain free variables, called rigid variables. The problem then becomes: is there a grounding substitution θ such that given equations $\theta(s_1 = t_1), \dots, \theta(s_n = t_n)$, we can derive $\theta(s = t)$. Such θ is called a rigid E-unifier. Finding a rigid E-unifier has been proved NP-complete in [7]. We follow the lines of Degtyarev [4], Comon [1] and Nieuwenhuis [8] and implement each of their algorithms in an efficient way. For this we improve the \mathcal{BSE} calculus of Degtyarev and Voronkov, compute normal forms according to Comon's rewrite system \mathcal{R} without performing substitutions and search for a satisfiable simple system according to Nieuwenhuis without constructing them all.

Degtyarev and Voronkov [4] proposed the \mathcal{BSE} calculus to compute a rigid E-unifier closing a single leaf of a tableau. These solutions are later combined to close the entire tableau. They also proved completeness of their approach. Closing all leaves at once requires simultaneous rigid E-unification, which proved to be undecidable [3].

In this paper we propose the $e\mathcal{BSE}$ calculus which is more efficient than the \mathcal{BSE} calculus, when computing many similar rigid E-unifiers, since it allows caching of intermediate results.

Both the \mathcal{BSE} and $e\mathcal{BSE}$ calculus need to compute satisfiability of symbolic ordering constraints, which is NP-complete.

Comon [1] solved these constraints by using a rewrite system \mathcal{R} based on the lexicographical path ordering. The normal form of \mathcal{R} is a set of solved forms, each of which gives rise to an (exponential) number of simple systems. Checking satisfiability of a simple system is done in linear time. The original constraint is satisfiable if at least one of the simple systems is. Nieuwenhuis [8] loosened the definition of simple systems to reduce the number of case distinctions and improve performance over Comon's method.

We will implement Comon's \mathcal{R} by a set of mutually recursive functions. Efficiency is obtained by using a unifier representation that avoids performing substitutions and a computation order that does not require storing intermediate results.

Finding a satisfiable simple system along the lines of Nieuwenhuis requires the inspection of an exponential number of simple systems. We introduce an intermediate graph structure that represents all possible simple systems and then use a bounded backtracking algorithm to extract a satisfiable simple system from this graph (if possible). The intermediate graph structure allows us to efficiently construct the simple systems and abort the construction of unsatisfiable simple systems as soon as possible, effectively discarding whole sets of unsatisfiable simple systems at once.

Section 2 of this paper describes the concepts and definitions we use. Section 3 describes the theory of solving rigid E-unification problems: our $e\mathcal{BSE}$ calculus, the rewrite system \mathcal{R} and the definition of simple systems. In Section 4 we provide implementation details of these three steps. The results are discussed in Section 5.

2 Preliminaries

Definition 1 (formulas). *The grammar for formulas F is given by*

$$\begin{aligned} F &::= \top \mid \perp \mid F \wedge F \mid F \vee F \mid T = T \\ T &::= \mathcal{F}(T^*) \mid \mathcal{V} \end{aligned}$$

Where \mathcal{V} is a set of variables and \mathcal{F} is a set of function symbols.

$FV(P)$ denotes the free variables of a (set of) formula(s) P as usual. A substitution $\theta : \mathcal{V} \rightarrow T$ is a mapping from variables to terms. We will use \simeq to denote syntactic equality throughout this paper.

Definition 2 (Lexicographical Path Ordering LPO). *Let $s \simeq f(s_1, \dots, s_n)$ and $t \simeq g(t_1, \dots, t_m)$ be two terms. Let $>_{\mathcal{F}}$ be an arbitrary ordering of the function symbols in \mathcal{F} (called the precedence ordering). Then $s > t$ iff one of the following holds:*

1. $(\exists i : 1 \leq i \leq m : s_i > t)$
2. $f >_{\mathcal{F}} g \wedge (\forall j : 1 \leq j \leq m : s > t_j)$

$$3. f = g \wedge (\exists j : 1 \leq j \leq n : (\forall i : 1 \leq i < j : s_i \simeq t_i) \\ \wedge s_j \succ t_j \\ \wedge (\forall k : j < k \leq n : s \succ t_k))$$

The LPO is total on ground terms.

Definition 3 (Rigid E-unification problem). Let $s_1 = t_1, \dots, s_n = t_n$ be a list of equations and let $s = t$ be a single goal equation. The rigid E-unification problem denoted by $s_1 = t_1, \dots, s_n = t_n \vdash s = t$ is stated as: is there a substitution θ such that $\theta(s_1 = t_1 \wedge \dots \wedge s_n = t_n \Rightarrow s = t)$ is a tautology?

Rigid E-unification is shown to be NP-complete in [7].

Like in [9] we represent a substitution θ as a list

$$\theta' = \langle [x_1 \mapsto t_1], \dots, [x_n \mapsto t_n] \rangle$$

of one-point mappings. The substitution θ corresponding to θ' is then computed as θ_1 , where

$$\theta_i = [x_i \mapsto \theta_{i+1}(t_i)] \circ \theta_{i+1} \text{ for } 1 \leq i < n \\ \theta_n = [x_n \mapsto t_n]$$

During computation of a unifier the substitution will be extended to include a new mapping from a variable x to a term $\theta(t)$, where t is a (sub)term of the original unification problem and θ is the unifying substitution found so far. Hence, we need a representation for $[x \mapsto \theta(t)] \circ \theta$. If θ is represented by the list $\theta' = \langle [x_1 \mapsto t_1], \dots, [x_n \mapsto t_n] \rangle$, the adapted substitution is represented by $\langle [x \mapsto t], \theta' \rangle$. Hence, instead of actually computing the substitution θ or the term $\theta(t)$, we simply prepend $[x \mapsto t]$ to the list representation θ' . The list representation never becomes longer than the number of rigid variables. Also, the unification algorithm only uses the head-symbol of $\theta(t)$, which can be computed directly from t and the list representation θ' . Therefore, using the list representation for substitutions, we will avoid computing and performing substitutions altogether.

3 Solving rigid equation problems

The approach to solve rigid equation problems used in this paper can be summarized as follows:

- Use a variant of the calculus \mathcal{BSE} (see [4]) to rewrite the goal into a trivial form (i.e. $s = s$). During rewriting, we generate constraints of which the satisfiability has to be checked.
- In order to check satisfiability of the constraints, first rewrite them into a number of solved forms.
- To check satisfiability of a solved form, generate the corresponding simple systems and check if a satisfiable simple system exists.

In the following subsections, we will briefly describe each of these steps.

$$\begin{array}{l}
 \text{lrbs} \quad \frac{E \cup \{\langle l = r, \mathcal{C}_1 \rangle, \langle s[p] = t, \mathcal{C}_2 \rangle\} \vdash \langle e, \mathcal{C} \rangle}{E \cup \{\langle l = r, \mathcal{C}_1 \rangle, \langle s[p] = t, \mathcal{C}_2 \rangle, \langle s[r] = t, \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge l \succ r \wedge s[p] \succ t \wedge l \simeq p \rangle\} \vdash \langle e, \mathcal{C} \rangle} \quad p \notin \mathcal{V} \\
 \hspace{10em} \hspace{10em} \hspace{10em} \hspace{10em} \hspace{10em} \hspace{10em} \hspace{10em} \hspace{10em} \hspace{10em} s[r] \not\approx t \\
 \\
 \text{rrbs} \quad \frac{E \cup \{\langle l = r, \mathcal{C}_1 \rangle\} \vdash \langle s[p] = t, \mathcal{C}_2 \rangle}{E \cup \{\langle l = r, \mathcal{C}_1 \rangle\} \vdash \langle s[r] = t, \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge l \succ r \wedge s[p] \succ t \wedge l \simeq p \rangle} \quad p \notin \mathcal{V} \\
 \\
 \text{er} \quad \frac{E \vdash \langle s = t, \mathcal{C} \rangle}{\vdash \langle s = s, \mathcal{C} \wedge s \simeq t \rangle} \quad s \not\approx t
 \end{array}$$

Fig. 1. The eBSE-calculus as used in our implementation. $s[p]$ denotes a term s in which term p occurs. A rule may only be applied if the constraints are satisfiable.

3.1 The eBSE calculus

Our eBSE calculus, based on the BSE calculus in [4], is given in Figure 1. The rules' names remain the same: left rigid basic superposition (lrbs), right rigid basic superposition (rrbs) and equality resolution (er). The constraints ensure that rewriting is only done if the term being rewritten becomes smaller and hence, they ensure the algorithm terminates.

The original BSE calculus maintains one constraint that ensures correctness of the entire derivation. For instance, the lrbs rule of Degtyarev and Voronkov is:

$$\frac{E \cup \{l = r, s[p] = t\} \vdash e \cdot \mathcal{C}}{E \cup \{l = r, s[r] = t\} \vdash e \cdot \mathcal{C} \wedge l \succ r \wedge s[p] \succ t \wedge l \simeq p} \quad p \notin \mathcal{V}, s[r] \not\approx t$$

The equation $s[r] = t$ is removed from the set of equations when the rule is used. This is not necessary, since the equation as such is still valid, even though the constraint ensures that it can only be used with the same instances of the rigid variables. The new equation $s[r] = t$ is only valid when the constraint is satisfiable. Hence, we cannot use it in other derivations and we have to recompute applications of the lrbs rule for every rigid E problem we solve. Note that as soon as an unsolvable rigid E problem is tried, all possible applications of lrbs are computed.

However, within a theorem prover the initial set of equations is mostly the same for different rigid E problems. Since the eBSE calculus maintains a constraint for every equation and does not remove any equation when a rule is used, it allows us to pre-compute a closure E_c of the set E of equations with respect to lrbs. E_c is subsequently used to solve all problems of the form $E \vdash s = t$. More details can be found in [6].

The closure E_c of a set of equations E with respect to the lrbs rule is computed incrementally: assume E_c was computed from E . When E is extended by a new equation $s = t$, yielding $E' = E \cup \{s = t\}$, it is sufficient to consider all applications of lrbs with one equation from E_c and $s = t$ to compute E'_c . Hence, $E'_c = E_c \cup \{c(E_c, s = t)\}$, where $c(E_c, s = t)$ denotes the all the derived equations from E_c involving $s = t$. Computing $c(E_c, s = t)$ is much more efficient than computing E'_c from E' directly.

Equality rules:

- $$(D_1) f(v_1, \dots, v_n) \simeq f(u_1, \dots, u_n) \rightarrow_{\mathcal{R}} v_1 \simeq u_1 \wedge \dots \wedge v_n \simeq u_n$$
- $$(C_1) f(v_1, \dots, v_n) \simeq g(u_1, \dots, u_m) \rightarrow_{\mathcal{R}} \perp$$
- if $f \neq g$
- $$(R) x \simeq t \wedge P \rightarrow_{\mathcal{R}} x \simeq t \wedge P[x := t]$$
- if $x \in FV(P) \setminus FV(t)$, P is a conjunction of (in)equations
and $t \in \mathcal{V} \Rightarrow t \in FV(P)$.
- $$(O_1) s \simeq t[s] \rightarrow_{\mathcal{R}} \perp$$
- if $s \in \mathcal{V}$ and $s \neq t[s]$

Fig. 2. The equality rules of \mathcal{R} to compute solved forms. Note that these constitute regular Robinson unification.

3.2 Solved forms

A constraint puts restrictions on the values that may be assigned to rigid variables. The rewrite system \mathcal{R} is used to make these restrictions explicit. The normal form according to \mathcal{R} is either \top , \perp or a disjunction of constraints of the form

$$x_1 \simeq t_1 \wedge \dots \wedge x_n \simeq t_n \wedge u_1 \succ v_1 \wedge \dots \wedge u_m \succ v_m,$$

where x_1, \dots, x_n are variables not occurring in $t_1, \dots, t_n, u_1, \dots, u_m, v_1, \dots, v_m$ and where for every $i, 1 \leq i \leq m$ either u_i or v_i is a variable and v_i is not a sub-term of u_i or vice versa. Each of these disjuncts is called a solved form. Every conjunct in a solved form constitutes a value (solved part) or an upper or lower bound (constrained part) for a rigid variable. If $m = 0$ we have a solution and do not need to continue.

3.3 Simple systems

In order to decide whether a solved form is satisfiable, we have to find a ground substitution such that the constraint holds according to LPO. The solved part provides part of this substitution. For the constrained part we need to consider all corresponding simple systems and find one that is satisfiable.

The set of simple systems for a constrained part c is defined by:

- Compute the set $sub(c)$ of all sub-terms occurring in c .
- Consider all possible orderings of $sub(c)$, where every sub-term s_2 of $s_1 \in sub(c)$ occurs after s_1 .
- For each possible ordering, put either \simeq or \succ between the terms in all possible ways that are compatible with the original constrained part.

A simple system is not satisfiable iff it is trivially bottom. It is trivially bottom if it contains one of the following (subscript $_s$ means that the (in)equality holds according to the simple system s):

Inequality rules:

$$\begin{aligned}
 (D_2) \quad & f(v_1, \dots, v_n) \succ g(u_1, \dots, u_m) \rightarrow \mathcal{R} \\
 & \quad f(v_1, \dots, v_n) \succ u_1 \wedge \dots \wedge f(v_1, \dots, v_n) \succ u_m \\
 & \quad \text{if } f \succ_{\mathcal{F}} g. \\
 (D_3) \quad & f(v_1, \dots, v_n) \succ g(u_1, \dots, u_m) \rightarrow \mathcal{R} \\
 & \quad v_1 \succeq g(u_1, \dots, u_m) \vee \dots \vee v_n \succeq g(u_1, \dots, u_m) \\
 & \quad \text{if } g \succ_{\mathcal{F}} f. \\
 (D_4) \quad & f(v_1, \dots, v_n) \succ f(u_1, \dots, u_n) \rightarrow \mathcal{R} \\
 & \quad v_1 \succeq f(u_1, \dots, u_n) \vee \dots \vee v_n \succeq f(u_1, \dots, u_n) \\
 & \quad \vee (v_1 \succ u_1 \wedge f(v_1, \dots, v_n) \succ u_2 \wedge \dots \wedge f(v_1, \dots, v_n) \succ u_n) \\
 & \quad \vee (v_1 \simeq u_1 \wedge v_2 \succ u_2 \wedge \dots \wedge f(v_1, \dots, v_n) \succ u_n) \\
 & \quad \vee \dots \\
 & \quad \vee (v_1 \simeq u_1 \wedge v_2 \simeq u_2 \wedge \dots \wedge v_n \succ u_n) \\
 & \quad \vee v_1 \succeq f(u_1, \dots, u_n) \vee \dots \vee v_n \succeq f(u_1, \dots, u_n) \\
 (O_2) \quad & t[s] \succ s \rightarrow \mathcal{R} \top \\
 & \quad \text{if } t[s] \not\succeq s. \\
 (O_3) \quad & s \succ t[s] \rightarrow \mathcal{R} \perp \\
 (T_1) \quad & s \succ t \wedge t \succ s \rightarrow \mathcal{R} \perp
 \end{aligned}$$

Fig. 3. The inequality rules of \mathcal{R} to compute solved forms

1. $f(s_1, \dots, s_n) \simeq_s g(t_1, \dots, t_m)$ with f different from g .
2. $f(s_1, \dots, s_p) \simeq_s f(s'_1, \dots, s'_p)$ and $(\exists i \in 1 \dots p. \neg(s_i \simeq_s s'_i))$.
3. $s \simeq_s t$ and t is a proper sub-term of s or visa versa.
4. $f(s_1, \dots, s_p) \succ_s g(t_1, \dots, t_m)$ with $g \succ_{\mathcal{F}} f$ and $\neg(\exists i \in 1 \dots p. s_i \succeq_s g(t_1, \dots, t_m))$.
5. $f(s_1, \dots, s_p) \succ_s f(s'_1, \dots, s'_p)$ and $\neg(\langle s_1, \dots, s_p \rangle \succ_s^{lex} \langle s'_1, \dots, s'_p \rangle)$.

4 Implementing Rigid-E unification

In this section we introduce a more efficient variant of the \mathcal{BSE} calculus called the $e\mathcal{BSE}$ calculus, show how to compute the normal form of \mathcal{R} without performing substitutions or storing intermediate results and introduce an intermediate graph structure to find a satisfiable simple system.

4.1 Implementing the $e\mathcal{BSE}$ calculus

The $e\mathcal{BSE}$ calculus can be implemented directly. A single rule is applied as follows: first check the side-conditions, then compute the new constraint by adding the required conditions and check satisfiability, and finally compute E' or $s' = t'$ to generate the rigid E-unification problem that makes up the conclusion of the rule. The full calculus is used by first computing E_c and then recursively applying $rrbs$ and er . We terminate the proof search if all derivations have been tried or when a solution has been found.

To save memory, E can be implemented by a linked list of equation-constraint pairs. Whenever a set $E'_c = E_c \cup \{c(E_c, s = t)\}$ has to be represented, the first elements in the list represent $c(E_c, s = t)$ and the tail of the list is E_c , which already exists. That is, we represent all sets of equations by an upside-down tree.

4.2 Computing solved forms

We efficiently implement \mathcal{R} (see Figures 2 and 3) by: (1) avoiding performing substitutions (that is, we do not compute $P[x := t]$ for rule R) and (2) recursively unfolding \mathcal{R} on a depth first basis to avoid the representation of intermediate results. We will explain both techniques by a part of the implementation. The full implementation is given in the appendix.

To demonstrate (1), we consider the occurs check problem: does variable $v \in \mathcal{V}$ occur in term t ? This check is performed on a term t obtained from the initial unification problem after substitution on a term s . The substitution θ is represented by θ' . I.e. $t = \theta(s)$.

Instead of actually computing $\theta(s)$, we check if $s \in \mathcal{V}$ and if there is a mapping $[s \mapsto s']$ in the list representation θ' . If so, $\theta(s)$ is equal to $\theta(s')$ and hence, we consider s' instead of s . This is repeated until (1) either $s' \notin \mathcal{V}$, or (2) $s' \in \mathcal{V}$, but there is no corresponding mapping in the list anymore. In the first case we get $\theta(s) = \theta(f(s_1, \dots, s_n)) = f(\theta(s_1), \dots, \theta(s_n))$. We then perform the occurs check on $\theta(s_1)$ till $\theta(s_n)$ recursively, using only θ' and s_1 till s_n . In the second case we only need to check if v is s' . Hence, we merely use terms that already existed in the original problem without changing them.

We now first choose representations for solved forms and then show how to implement \mathcal{R} efficiently.

A single solved form will be represented by a type called "Solved", consisting of two parts: (1) the solved part θ' , representing a substitution as before and (2) the constrained part cp' , which is a set of inequalities where either the left hand side or the right hand side is a variable not occurring in the domain of θ' . cp' represents the inequations $\theta(cp')$, where θ is the substitution represented by θ' . cp' is implemented by a list of pairs of variables and references to already existing sub-terms. No substitutions are performed and no sub-terms are copied.

A set of solved forms is represented by the type "{Solved}". We represent \perp by the empty set of solved forms and we represent \top by a solved form with an empty list of one-point mappings with an empty set of inequalities.

We want to extend an existing constraint \mathcal{C} by a new partial constraint $s \simeq t$. Assume \mathcal{C} is already a solved form (i.e. $\mathcal{C} : Solved$). Instead of adding $s \simeq t$ to the representation and apply regular rewriting, we design a function `makeEqual` that takes C , s and t as arguments and returns a (possibly empty) set of solved forms. In this function, we unfold $\mathcal{C}.\theta'$ on s and t like before (conforming to rule R) and check which rule of \mathcal{R} applies. Instead of adding $\mathcal{C}.\theta'(s) \simeq \mathcal{C}.\theta'(t)$ we add the right hand side of the rewrite rule to \mathcal{C} by recursive calls. Unless $\mathcal{C}.\theta'(s) \simeq \mathcal{C}.\theta'(t)$ is part of the solved form, it is not added to the representation. Adding $\mathcal{C}.\theta'(s) \simeq \mathcal{C}.\theta'(t)$ to the solved form is done by adding $s \mapsto t$ to $\mathcal{C}.\theta'$ and

re-evaluating $\mathcal{C}.cp'$. Note that if $\mathcal{C}.cp'$ is empty, `makeEqual` is an implementation of regular Robinson unification.

Equally, we construct `makeGreater`, which adds $s \succ t$ to \mathcal{C} . We use convenience functions to apply `makeEqual` and `makeGreater` to $\{Solved\}$. The full abstract code is given in Appendix A.

4.3 Computing a satisfiable simple system

Since the number of simple systems corresponding to a solved form is exponential, we will not compute every simple system and then check its satisfiability. Instead, we build a graph representing all simple systems that are consistent with the lpo and the constraints given by the solved form. We use a backtracking algorithm to find a satisfiable simple system. Satisfiability is computed on the fly and backtracking is cut off early whenever possible.

In the simple systems graph the vertices represent sub-terms and the edges represent the lpo-relationships between them. The graph is constructed as follows:

- Construct a list of all sub-terms of all terms in the constrained part.
- Sort this list according to the partial ordering defined by the lpo (The ordering is partial, since the sub-terms contain variables).
- Represent the graph G by: (1) the set $G.V$ of all vertices (2) the set $G.M \subseteq G.V$ of minimal vertices and (3) the sets $G.s(v)$ of all direct successors of v (i.e. $G.s : G.V \rightarrow \mathcal{P}(G.V)$).
- Construct the graph adding all sub-terms from small to large.
- Add the edges representing the relations imposed by the constrained part. If a cycle is introduced there will be no satisfiable simple systems.

The algorithm in Figure 4 will search through all simple systems s represented by the simple systems graph. The arguments of the function have the following meaning:

- G is the representation of the simple systems graph.
- S is a partial simple system constructed so far. Initially, S is empty.
- in is an array stating for every vertex v the number of incoming edges when all vertices already in S are removed from the graph. If $in[v] > 0$, v may not be used to extend S , since obviously other vertices have to be added first. If v is already in S then $in[v] = -1$.
- $level$ is the number of \succ symbols already in S . Initially, this is 0.
- $levels$ is an array stating for every vertex v with $in[v] = 0$ the minimum value that $level$ must have before it may be used to extend S . $levels$ is updated during recursion whenever a vertex v is selected to extend S . The idea is that when v is (and the corresponding edges are) removed from G , the vertices $G.s(v)$ may not be prepended to extend S , unless at least one \succ has been inserted after v .

```

0 function findSimpleSystem( $G$  : Graph;  $S$  : SimpleSystem;
                            $in$  :  $G.V \rightarrow int$ ;  $level$  :  $int$ ;
                            $levels$  :  $G.V \rightarrow int$ 
                           ) {SimpleSystem}[[
1   $r := \emptyset$ ;
2  if  $|S| = |G.V| \rightarrow r := \{S\}$ 
3  else
4    foreach  $v \in G.V$  do
5      if  $in[v] = 0 \rightarrow$ 
6         $in[v] := -1$ ;
7        foreach  $n \in G.s(v)$  do
8           $in[n] := in[n] - 1$ ;
9           $levels[n] := level + 1$ 
10       od;
11       if  $level > levels[v] \wedge \text{valid}(v \simeq S) \rightarrow$ 
12          $r := r \cup \text{findSimpleSystem}(G, v \simeq S, in, level, levels)$ 
13       fi;
14       if  $\text{valid}(v \succ S) \wedge |S| \neq 0 \rightarrow$ 
15          $r := r \cup \text{findSimpleSystem}(G, v \succ S, in, level + 1, levels)$ 
16       fi
17       foreach  $n \in G.s(v)$  do  $in[n] := in[n] + 1$  od;
18        $in[v] := 0$ 
19     fi
20   od
21 fi;
22 return  $r$ 
23 ]]
```

Fig. 4. The backtracking algorithm to extract a satisfiable simple system from the simple systems graph.

The function *valid* checks the satisfiability constraints of the simple system. It assumes that S is already satisfiable and only checks if the extension (either $v \simeq$ or $v \succ$) introduces inconsistencies. This is done by checking the constraints stated in Section 3.3. Constraint 3 holds by construction.

Any simple system which contains $t_1 \simeq t_2$ has the same solutions as the simple system in which t_1 and t_2 are swapped. Therefore, extending a simple system with $t \simeq$ will only be considered if t is greater than that of the topmost term of the simple system according to some total ordering on terms (e.g. the alphabetic ordering on textual representation of terms).

Computation is aborted as soon as a satisfiable simple system is found.

5 Conclusion

In this paper we have shown how to fully implement a rigid E-unification algorithm. Although the problem itself is NP-complete [7], we paid much attention to efficiency.

We improved on the \mathcal{BSE} calculus with the $e\mathcal{BSE}$ calculus, allowing us to pre-compute and re-use applications of the *lrbs* rule. This allows us to easily embed the rigid E-unifier in a tableau based theorem prover, where each node contains one set of equations. This set is represented as a linked list, the tail of which is always the set of equations of the parent node. Hence, we avoid both multiple computation and multiple storage of equations.

We implemented \mathcal{R} using recursive functions without performing any substitutions or storing intermediate results. All solved forms are computed first and simple systems are only computed if there are no solved forms without constrained parts. This often allows us to skip the exponential part of computing rigid E-unifiers.

Finally, we computed a satisfiable simple system using an intermediate graph structure. This way we avoid constructing the exponential number of simple systems explicitly and also we cut off computations of simple systems as soon it is trivially bottom.

In our experiments so far it turns out that most rigid E-unification problems are solved without the exhaustive search for a satisfiable simple system. This yields an efficient implementation for our purposes. Unfortunately, since other rigid E-unification algorithms are embedded within theorem provers, we cannot directly compare our results to other implementations in an easy way.

Our next step will be the embedding of this rigid E-unifier in the tableaux based theorem prover of Cocktail [5]. Also, we want to translate the trees generated by the $e\mathcal{BSE}$ calculus into λ -terms, since this allows full embedding of the entire automated theorem prover in systems like Coq [2].

References

1. H. Comon, *Solving symbolic ordering constraints*, International Journal of Foundations of Computer Science **1** (1990), no. 4, 387–412.

2. Coq, *The Coq proof assistant*, URL: <http://coq.inria.fr/>, 2008.
3. A. Degtyarev and A. Voronkov, *The undecidability of simultaneous rigid E-unification*, Theoretical Computer Science **166** (1996), no. 1–2, 291–300.
4. A. Degtyarev and A. Voronkov, *What you always wanted to know about rigid E-unification*, Journal of Automated Reasoning **20** (1998), 47–80.
5. M. Franssen, *Cocktail: A tool for deriving correct programs*, Ph.D. thesis, Eindhoven University of Technology, 2000.
6. M. Franssen, *Implementing rigid e-unification*, Computing Science Report 08–24, Eindhoven Universiteit of Technology, 2008.
7. J.H. Gallier, P. Narendran, D. Plaisted, and W. Snyder, *Rigid E-unification is NP-complete*, In Proc. IEEE Conference on Logic in Computer Science (LICS), IEEE, 1988, pp. 338–346.
8. R. Nieuwenhuis, *Simple LPO constraint solving methods*, Information Processing Letters **47** (1993), no. 2, 65–69.
9. L.C. Paulson, *Designing a theorem prover*, Background: Computational Structures (S Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, eds.), Handbook of Logic in Computer Science, vol. 2, Oxford Science Publications, 1992, pp. 415–475.

APPENDIX

For your convenience this appendix contains some additional material that might be of interest.

A Abstract code implementation for \mathcal{R}

The abstract algorithm to compute solved forms for a given (in)equality is given in Figures 5 and 6. We provide line numbers to simplify the discussion that follows:

- **makeEqual** (lines 0 till 18) implements the rules D1, C1, R and O1. Note that if the constrained part is empty, makeEqual implements a regular Robinson unification algorithm. The idea is that the solved form is extended such that it also unifies $\theta(s)$ and $\theta(t)$ if possible. It returns a set of solved forms that satisfy this requirement. Note that this function will never abort, but instead return an empty set of solutions.
- **makeEqualSet** (lines 19 till 21) is a convenience function to apply the previous function to a set of solved forms instead of a single solved form and return the union of the results.
- **addSubst** (lines 22 till 26) implements the rule R and is called only by makeEqual. Instead of only adjusting θ' , which in our representation is trivial, addSubst also has to re-evaluate the constrained part of the solved form sol , since these inequations also might contain references to the variable v . The re-evaluation takes place by using makeGreater to add all the inequations of $sol.cp'$ to a solved form initially containing only $sol.\theta'$. This results in a complicated mutually recursive pattern between makeEqual and makeGreater. Termination of this recursion is discussed later.
- **occursCheck** (lines 27 till 34) checks if variable v occurs in $\theta(s)$.
- **makeGreater** (lines 35 till 64) implements all inequality rules of \mathcal{R} , except T2. The idea of this function is that the solved form sol is extended to a solved form that also satisfies $\theta(s) \succ \theta(t)$. Instead of adding the constraint to a set and rewriting it to a solved form, the rewriting takes place directly and only if $\theta(s) \succ \theta(t)$ is part of the solved form, $s \succ t$ is added to $sol.cp'$. Lines 28 and 29 unfold the substitution far enough to decide on the first function symbol of $\theta(s)$ and $\theta(t)$. The rules T1, O2 and O3 are applied in the cases where $\theta(s)$ or $\theta(t)$ are a variable (lines 30 till 35). Rule D2 (lines 38 till 41) is computed by recursively adding all the constraints of D2's right hand side to the solved form. Since every addition may return a set of solved forms, sol is put in a set and makeGreaterSet is used instead of makeGreater. Rule D3 (lines 42 and 43) simply unites the solved forms obtained from extending sol with all disjuncts of the right hand side of D3. Finally, D4 (lines 44 till 53) is computed in two steps. In line 45 the disjuncts $v_1 \succeq f(u_1, \dots, u_n)$ till $v_n \succeq f(u_1, \dots, u_n)$ are combined with sol to create the initial set S of solved forms that make up the result. Then, in line 46 till 52 the remaining disjuncts of D4 are computed one by one in variable H and joined with S . eq invariantly contains all solved forms obtained by extending sol

```

0 function makeEqual(sol :Solved; s, t : T) : {Solved} [|
1   while ( $\exists e. s \mapsto e \in \text{sol}.\theta'$ )  $\rightarrow s := \text{sol}.\theta'(s)$ ;
2   while ( $\exists e. t \mapsto e \in \text{sol}.\theta'$ )  $\rightarrow t := \text{sol}.\theta'(t)$ ;
3   if  $s \in \mathcal{V} \rightarrow$ 
4     if  $s = t \rightarrow$  return {sol} //special case of rule D1
5     elseif ( $\text{occursCheck}(\text{sol}.\theta', s, t) \rightarrow$  return  $\emptyset$  //rule O1
6     else return addSubst(sol, s, t) //rule R
7     fi
8   elseif  $t \in \mathcal{V} \rightarrow$  return makeEqual(sol, t, s)
9   else
10    let  $f(\text{sol}.\theta'(s_1), \dots, \text{sol}.\theta'(s_n)) :: s, g(\text{sol}.\theta'(t_1), \dots, \text{sol}.\theta'(t_m)) :: t$ ;
11    if  $f \simeq g \rightarrow$  //rule D1
12      SOL := {sol};
13      foreach  $i : 1 \leq i \leq n$  do  $H := \text{makeEqualSet}(SOL, s_i, t_i)$  od;
14      return SOL
15    else return  $\emptyset$  //rule C1
16    fi
17  fi
18 |]

19 function makeEqualSet(SOL: {Solved}; s, t : T) : {Solved} [|
20 return ( $\bigcup \text{sol} : \text{sol} \in SOL : \text{makeEqual}(\text{sol}, s, t)$ )
21 |]

22 function addSubst(sol :Solved; v :  $\mathcal{V}$ ; t : T) : {Solved} [|
23    $H := \{\{(v \mapsto t), \text{sol}.\theta'\}, \emptyset\}$ ;
24   for ( $s \succ t \in \text{sol}.\text{cp}'$ ) do  $H := \text{makeGreaterSet}(H, s, t)$  od;
25   return H
26 |]

27 function occursCheck( $\theta'$  :list of  $\mathcal{V} \mapsto T$ ; v :  $\mathcal{V}$ ; s : T) : bool [|
28   while ( $\exists e. [s \mapsto e] \in \theta'$ )  $\rightarrow s := \theta'(s)$ ;
29   if  $s \in \mathcal{V} \rightarrow$  return  $s = v$ 
30   else
31     let  $f(\theta'(s_1), \dots, \theta'(s_n)) :: s$ ;
32     return ( $\bigvee i : 1 \leq i \leq n : \text{occursCheck}(\theta', v, s_i)$ )
33   fi
34 |]

```

Fig. 5. implementing \mathcal{R} for equalities

```

35 function makeGreater(sol :Solved; s, t : T) : {Solved} [|
36 while ( $\exists e.s \mapsto e \in \text{sol}.\theta$ )  $\rightarrow$  s := sol. $\theta'$ (s);
37 while ( $\exists e.t \mapsto e \in \text{sol}.\theta$ )  $\rightarrow$  t := sol. $\theta'$ (t);
38 if (s  $\in \mathcal{V}$ )  $\vee$  (t  $\in \mathcal{V}$ )  $\rightarrow$ 
39   if s  $\in \mathcal{V} \wedge$  occursCheck(sol. $\theta'$ , s, t)  $\rightarrow$  return  $\emptyset$  //rule O3
40   elseif t  $\in \mathcal{V} \wedge$  occursCheck(sol. $\theta'$ , t, s)  $\rightarrow$  return {sol} //rule O2
41   elseif (t  $\succ$  s)  $\in \text{sol}.cp'$   $\rightarrow$  return  $\emptyset$  //rule T1
42   else return {sol. $\theta'$ , sol.cp'  $\cup$  {s  $\succ$  t}} //irreducible s  $\succ$  t
43 fi
44 else
45   let f( $\theta'(s_1), \dots, \theta'(s_n)$ ) :: s, g( $\theta'(t_1), \dots, \theta'(t_m)$ ) :: t;
46   if f  $\succ_{\mathcal{F}}$  g  $\rightarrow$  //D2
47     H := {sol};
48     foreach i : 1  $\leq i \leq m$  do H :=makeGreaterSet(H, s, ti) od;
49     return H
50   elseif g  $\succ_{\mathcal{F}}$  f  $\rightarrow$  //D3
51     return ( $\bigcup i : 0 \leq i \leq n : \text{makeEqual}(\text{sol}, s_i, t) \cup$ 
52               makeGreater(sol, si, t))
53   else //f  $\simeq$  g  $\rightarrow$  D4
54     S = ( $\bigcup i : 1 \leq i \leq n : \text{makeEqual}(\text{sol}, s_i, t) \cup$ 
55           makeGreater(sol, si, t));
56     eq := {sol};
57     for i := 1 to n do
58       H :=makeGreater(eq, si, ti);
59       foreach j : i + 1  $\leq j \leq n$  do H :=makeGreaterSet(H, s, tj) od;
60       S := S  $\cup$  H;
61       eq :=makeEqualSet(eq, si, ti)
62     od;
63     return S
64 fi
65 fi
66 [|
67 function makeGreaterSet(SOL: {Solved}; s, t : T) : {Solved} [|
68 return ( $\bigcup \text{sol} : \text{sol} \in \text{SOL} : \text{makeGreater}(\text{sol}, s, t)$ )
69 [|

```

Fig. 6. implementing \mathcal{R} for inequalities

with the equalities $s_1 \simeq t_1, \dots, s_{i-1} \simeq t_{i-1}$. Using this invariant H can be initialized efficiently.

- **makeGreaterSet** (lines 65 till 67) is, like `makeEqualSet`, a convenience function to apply the function `makeGreater` to a set of solved forms instead of a single solved form and return the union of the results.

Termination of the algorithm can be seen as follows: every recursive call either deals with sub-terms of the original arguments (calls to `makeEqual`, `makeEqualSet`, `makeGreater` or `makeGreaterSet`) or it eliminates one of the free variables (calls to `addSubst`).

A Tableau Method for Checking Rule Admissibility in **S4**

Sergey Babenyshev¹, Vladimir Rybakov¹, Renate A. Schmidt², and Dmitry Tishkovsky²

¹ Department of Computing and Mathematics,
Manchester Metropolitan University, UK

² School of Computer Science, The University of Manchester, UK

Abstract Rules that are admissible can be used in any derivations in any axiomatic system of a logic. In this paper we introduce a method for checking the admissibility of rules in the modal logic **S4**. Our method is based on a standard semantic ground tableau approach. In particular, we reduce rule admissibility in **S4** to satisfiability of a formula in a logic that extends **S4**. The extended logic is characterised by a class of models that satisfy a variant of the co-cover property. The class of models can be formalised by a well-defined first-order specification. Using a recently introduced framework for synthesising tableau decision procedures this can be turned into a sound, complete and terminating tableau calculus for the extended logic, and gives a tableau-based method for determining the admissibility of rules.

1 Introduction

Logical admissible rules were first considered by Lorenzen [8]. Initial investigations were limited to observations on the existence of interesting examples of admissible rules that are not derivable (see Harrop [6], Mints [9]). The area gained new momentum when Fridman [2] posed the question whether algorithms exist for recognising whether rules in intuitionistic propositional logic **IPC** are admissible. This problem and the corresponding problem for modal logic **S4** are solved affirmatively in Rybakov [11,12]. The same approach can be used for a broad range of propositional modal logics, for example **K4**, **S4**, **GL** [13]. Roziere [10] presents a solution to Fridman's problem for **IPC** that uses methods of proof theory.

The theory of admissible rules in **S4** does not have the finite approximation property [7] in the strict sense. The algorithm in [11] is based on the existence of a model (of bounded size) that witnesses the non-admissibility of a rule, but is not necessarily a model for all the other admissible rules. In [1] it is observed that a witnessing model can be obtained by filtration.

Admissibility of rules has direct connections to the problem of unification. A refined technique [13] is used for admissibility of rules with meta-variables, for unification, for unification with parameters, and for the solvability of logical equations in transitive modal logics.

Algorithms deciding admissibility for some transitive modal logics and IPC, based on projective formulae and unification, are described in Ghilardi [3,4,5]. They combine resolution and tableau approaches for finding projective approximations of a formula and rely on the existence of an algorithm for theorem proving. A practically feasible realisation for **S4** was reported in [19]. These algorithms were specifically designed for finding general solutions for matching and unification problems. In contrast, the original algorithm of [11] can be used to find only *some* solution of such problems in **S4**.

In this paper we focus on **S4** and introduce a new method for recognising the admissibility of rules. Our method is based on the reduction of the problem of rule admissibility in **S4** to the satisfiability of a certain formula in an extension of **S4**. We refer to the extended logic as $\mathbf{S4}^{u,+}$. $\mathbf{S4}^{u,+}$ can be characterised by a class of models that satisfy a variant of the co-cover property that is definable by modal formulae. This property is expressible in first-order logic and means that the semantics of the logic can be formalised in first-order logic. We exploit this property in order to devise a tableau calculus for $\mathbf{S4}^{u,+}$ in a recently introduced framework for automatically synthesising tableau calculi and decision procedures [16,17].

The tableau synthesis method of [17] works as follows.

- (i) The user defines the formal semantics of the given logic in a many-sorted first-order language so that certain well-definedness conditions hold.
- (ii) The method automatically reduces the semantic specification of the logic to Skolemised implicational forms which are then rewritten as tableau inference rules. These are combined with some default closure and equality rules.

The set of rules obtained in this way provides a sound and constructively complete calculus. Furthermore, this set of rules automatically has a subformula property with respect to a finite subformula operator. If the logic can be shown to admit finite filtration with respect to a well-defined first-order semantics then adding a general blocking mechanism produces a terminating tableau calculus [16].

We show how, using this method, a sound, complete and terminating tableau calculus can be synthesised for the extended logic $\mathbf{S4}^{u,+}$. This tableau calculus is then incorporated into a method for solving the rule admissibility problem in **S4**.

The paper is structured as follows. In Section 2 we define the syntax and semantics of the modal logic **S4**, and its extension $\mathbf{S4}^u$ with the universal modality, in such a way that they can be accommodated in the tableau synthesis framework of [17]. The section also defines standard modal logic constructions and notions required for the main results of the paper. In Section 3, we give definitions of derivable and admissible rules for **S4** and state Rybakov's criterion for testing admissibility of rules in **S4**. The reduction of rule admissibility in **S4** to satisfiability in $\mathbf{S4}^{u,+}$ is described in Section 4. In Section 5, we show how the formulaic variant of the co-cover property can be expressed by a set of first-order formulae that provide a suitable background theory for the specification of the

Definitions of connectives:

$$\begin{aligned} \forall x(\nu(\neg p, x) &\leftrightarrow \neg\nu(p, x)) \\ \forall x(\nu(p \vee q, x) &\leftrightarrow \nu(p, x) \vee \nu(q, x)) \\ \forall x(\nu(\diamond p, x) &\leftrightarrow \exists y(R(x, y) \wedge \nu(p, y))) \end{aligned}$$

Background theory:

$$\begin{aligned} \forall x, y, z(R(x, y) \wedge R(y, z) &\rightarrow R(x, z)) \\ \forall x R(x, x) \end{aligned}$$

Figure 1. Specification of the semantics of S4 in S_{S4}

semantics of $S4^{u,+}$ in the tableau synthesis framework. Within this framework we can then synthesise sound and complete tableau calculi for the logics $S4^{u,+}$ and $S4^u$. Under certain conditions it is possible to refine the rules of the calculus that are generated by default [17]. These conditions are true for the logics we consider and is discussed in Section 6. Section 7 describes how terminating tableau calculi can be obtained by adding the unrestricted blocking mechanism of [15,16]. In Section 8, we finally give an algorithm for testing rule admissibility, but also rule derivability, in S4. In Section 9 we conclude with a discussion of the applicability of the algorithm and method to other logics and various problems closely related to admissibility.

For lack of space some of the details of the tableau synthesis framework are omitted; for these the interested reader is referred to [17] and also [15,16].

2 Syntax and Semantics

We denote the language of the modal logic S4 by \mathcal{L}_{S4} . \mathcal{L}_{S4} is given by the standard modal language over a countable set of propositional variables $\{p, q, p_0, q_0, \dots\}$, the Boolean logical connectives \neg and \vee , and the modal connective \diamond . Other Boolean connectives such as \top , \perp , \wedge , \rightarrow , and \leftrightarrow and the modal connective \square are defined via \neg , \vee , and \diamond as usual.

Let \mathcal{L}_{S4}^u denote the extension of the language \mathcal{L}_{S4} with the ‘somewhere’ modality $\langle u \rangle$. The dual modality $\neg \langle u \rangle \neg$ is the universal modality, which is denoted by $[u]$.

For a formula ϕ , we denote by $\text{sub}(\phi)$ the set of all subformulae of ϕ . Let $\text{sub}(\Sigma) \stackrel{\text{def}}{=} \bigcup \{\text{sub}(\phi) \mid \phi \in \Sigma\}$ for any set of formulae Σ . We usually write $\text{sub}(\phi_1, \dots, \phi_n)$ rather than $\text{sub}(\{\phi_1, \dots, \phi_n\})$. A set of formulae Σ is called a *signature* iff $\text{sub}(\Sigma) = \Sigma$.

Following the tableau synthesis framework in [17], we accommodate \mathcal{L}_{S4}^u in a multi-sorted first-order specification language. This specification language includes a countable set $\{x, y, z, x_0, y_0, z_0, \dots\}$ of first-order variables that represent elements in models, the binary predicate symbol R of the background theory that represents the accessibility relation, and a binary (intersort) predicate symbol ν which represents the forcing relation \models . (ν can be thought of as a holds

predicate.) Figure 1 gives the first-order semantic specification of the standard semantics of **S4** in the framework. We denote the specification by $S_{\mathbf{S4}}$. In addition to the formulae of Figure 1, $S_{\mathbf{S4}}$ includes the usual congruence axioms for the equality predicate \approx , see [17] for details.

Thus, an **S4**-(*Kripke*) *model* (of a signature Σ) is a first-order model $M = \langle W^M, R^M, \nu^M \rangle$ that validates all the formulae of Figure 1 under arbitrary substitutions of $\mathcal{L}_{\mathbf{S4}}$ -formulae (from the signature Σ) for propositional variables p and q .

Let $\mathbf{S4}^u$ be the extension of **S4** with the somewhere (or universal) modality. The semantic specification of $\mathbf{S4}^u$, denoted by $S_{\mathbf{S4}^u}$, is the extension of the specification $S_{\mathbf{S4}}$ of **S4** with

$$(1) \quad \forall x (\nu(\langle u \rangle p, x) \leftrightarrow \exists y \nu(p, y)).$$

This defines the somewhere modality $\langle u \rangle$. An $\mathbf{S4}^u$ -model (of a signature Σ) is by definition a first-order model $M = \langle W^M, R^M, \nu^M \rangle$ that validates this formula in addition to all the formulae of Figure 1 under arbitrary substitutions of $\mathcal{L}_{\mathbf{S4}^u}$ -formulae (from the signature Σ) for propositional variables p and q .

Note that if M is a model of the signature Σ then M is also a model of any signature $\Sigma' \subseteq \Sigma$. In the other direction, it is clear that every model M of a smaller signature Σ' can be extended to a model of a bigger signature Σ by (re)defining the interpretation of ν on formulae from $\Sigma \setminus \Sigma'$ (by induction on their length). We often use these facts when defining models.

A (*Kripke*) *frame* of **S4** (resp. $\mathbf{S4}^u$) is a first-order structure $F = \langle W^F, R^F \rangle$ that validates all the formulae of the background theory of $S_{\mathbf{S4}}$ (resp. $S_{\mathbf{S4}^u}$). A model M is *based on a frame* F (or the *underlying frame* of M is F) iff $W^M = W^F$ and $R^M = R^F$.

A *cluster* of a model M is a set $U \subseteq W^M$ such that for all $w \in W^M$ and $u \in U$ it holds that $(w, u), (u, w) \in R^M$ iff $w \in U$. Any set of R^M -incomparable clusters of M is called an *anti-chain* in M .

Let Σ be a fixed signature and M a model of the signature Σ . We say that a formula $\phi \in \Sigma$ is *true (satisfied)* in a world $w \in W^M$ (in symbols $M, w \models \phi$) iff $(\phi, w) \in \nu^M$. A formula ϕ (from Σ) is *valid* in M (in symbols $M \models \phi$) iff $M, w \models \phi$ for every $w \in W^M$. And, as usual, ϕ is *satisfiable* in M iff there is $w \in W^M$ such that $M, w \models \phi$. A formula $\phi \in \Sigma$ is *valid* in a frame F (in symbols $F \models \phi$) iff it is true in every model M of the signature Σ based on F .

For a signature Σ and every element w of W^M we define a Σ -*type* $\tau^\Sigma(w)$ of w as the set of all formulae from Σ which are true in w , namely:

$$\tau^\Sigma(w) \stackrel{\text{def}}{=} \{\psi \in \Sigma \mid M, w \models \psi\}.$$

We omit the superscript Σ and write $\tau(w)$ if Σ is known from the context. A model M of a signature Σ is called Σ -*differentiated* iff $\tau^\Sigma(w) = \tau^\Sigma(v)$ implies $w = v$ for all $w, v \in W^M$. A model M of a signature Σ is Σ -*formulaic* iff for every element $w \in W^M$ there is a formula $\phi \in \Sigma$ such that $M, w \models \phi$ and $M, v \not\models \phi$ for all $v \in W^M \setminus \{w\}$. It is clear that if Σ is finite then every Σ -differentiated model is also Σ -formulaic.

We call a Kripke model M of a signature Σ' a *definable variant* of a model N of a signature Σ if M and N are based on the same frame and for every propositional variable $p \in \Sigma'$ there is a formula $\phi \in \Sigma$ such that $M, w \models p \iff N, w \models \phi$ for every $w \in W^M = W^N$.

Let Σ be the set of all formulae in n variables p_1, \dots, p_n . An S4-model M of signature Σ is called an *n-characterising* model for S4 iff $M \models \phi \iff \phi \in \text{S4}$ for every $\phi \in \Sigma$.

3 Rules Admissible for S4

Since the language of S4 contains conjunction, without loss of generality we consider only one-premise rules. A *rule* is a pair $\langle \alpha, \beta \rangle$ of \mathcal{L}_{S4} -formulae, usually written α/β . A rule $r = \alpha/\beta$ is *valid* in a model M (written $M \models r$) iff $M \models \alpha$ implies $M \models \beta$. A rule r is *valid* on a frame F (written $F \models r$) iff r is valid in any model M based on F . Two rules r_1, r_2 are *semantically equivalent*, or simply *equivalent*, if $F \models r_1 \iff F \models r_2$ for any frame F .

A rule $r = \alpha/\beta$ is *derivable* in S4 iff β is derivable from α and the theorems of S4 with the rule of modus ponens and the rule of necessitation. It is clear that r is derivable in S4 iff r is valid in every S4-model. The following variant of the deduction theorem in S4 can be proved by standard methods.

Theorem 1. *A rule α/β is derivable in S4 iff $[u]\alpha \wedge \neg\beta$ is unsatisfiable in S4^u .*

A rule $r = \alpha/\beta$ is *admissible* for the modal logic S4, written $r \in \text{Adm}(\text{S4})$, if for every substitution σ from $\sigma(\alpha) \in \text{S4}$ it follows that $\sigma(\beta) \in \text{S4}$.

A series $\text{Ch}_n(\text{S4})$, $n > 0$, of formulaic and n-characterising S4 models is described in [13]. These models are used in the description of the following admissibility criterion.

Theorem 2 (Corollary of Theorem 3.3.3 [13]). *A rule is admissible in S4 iff it is valid in a definable variant of $\text{Ch}_n(\text{S4})$ for each $n > 0$.*

The most important property for the result of this paper is that the models $\text{Ch}_n(\text{S4})$, $n > 0$, and their definable variants possess the co-cover property. By definition, a model M has the *co-cover property* (CCP) if for every finite anti-chain $D \subseteq W^M$ (D may be empty), there exists a one-element cluster $\{w\} \subseteq W^M$ such that

$$\{u \in W^M \mid (w, u) \in R^M\} = \{w\} \cup \bigcup_{v \in D} \{u \in W^M \mid (v, u) \in R^M\}.$$

The one-element cluster is called a *co-cover for D* . Note that a co-cover for the empty anti-chain is a maximal one-element cluster of M .

A rule r is said to be in *reduced normal form* if it has the form

$$\text{(rnf)} \quad r = \left(\bigvee_{1 \leq j \leq s} \phi_j \right) / p_0,$$

and each disjunct ϕ_j has the form

$$\phi_j = \bigwedge_{0 \leq i \leq n} p_i^{t(i,j,0)} \wedge \bigwedge_{0 \leq i \leq n} (\Diamond p_i)^{t(i,j,1)},$$

where (i) all ϕ_j are different, (ii) p_0, \dots, p_n denote propositional variables, (iii) t is a Boolean function $t : \{0, \dots, n\} \times \{1, \dots, s\} \times \{0, 1\} \rightarrow \{0, 1\}$ (i.e., $t(i, j, z) \in \{0, 1\}$), and (iv) $\alpha^0 \stackrel{\text{def}}{=} \neg\alpha$ and $\alpha^1 \stackrel{\text{def}}{=} \alpha$ for any formula α .

Using the renaming technique any modal rule can be transformed into an equivalent rule in reduced normal form [11].

Lemma 1. *Any rule $r = \alpha/\beta$ can be transformed in exponential time to an equivalent rule in reduced normal form.*

Proof. We describe the algorithm of [13] (Lemma 3.1.3 and Theorem 3.1.11). Let $r = \alpha/\beta$ be a rule. We need a set of new variables $\{q_\gamma \mid \gamma \in \text{sub}(\alpha, \beta)\}$. The first step is to replace $r = \alpha/\beta$ with $r_1 = \alpha \wedge (q_\beta \leftrightarrow \beta)/q_\beta$. It is easy to see that r is refuted on a frame F iff r_1 can be refuted on the same frame F . Therefore r and r_1 are equivalent.

Inductive step: Suppose the rule $r_i = \gamma_i/q_\beta$ was obtained in the i th step. We call a formula $\delta \in \text{sub}(\gamma_i) \cap \text{sub}(\alpha, \beta)$ *final*, if it is not a variable and not a proper subformula of any other formula in $\text{sub}(\gamma_i) \cap \text{sub}(\alpha, \beta)$. Let T_i be the set of all final formulae obtained at the i th step. We replace the rule r_i with a new one, namely $r_{i+1} = \gamma_{i+1}/q_\beta$, where

$$\gamma_{i+1} = t_i(\gamma_i) \wedge \bigwedge_{q_\gamma * q_\delta \in T_i} ((q_\gamma \leftrightarrow \gamma) \wedge (q_\delta \leftrightarrow \delta)) \wedge \bigwedge_{*q_\delta \in T_i} (q_\delta \leftrightarrow \delta),$$

and $t_i(\gamma_i)$ is the formula obtained from γ_i by replacing all final subformulae δ with q_δ . It is straightforward to check that r_i and r_{i+1} are equivalent.

Note that every inductive step reduces the maximal height of the non-Boolean subformulae of the rule. Therefore after a finite number of steps we get a premise γ_k , which is a Boolean combination of *literals* of the form p or $\Diamond p$, where p is a propositional variable. We denote this intermediate form by $\text{do}(r)$ (depth-one form).

Finally, we transform the premise of the obtained rule $r_N = \gamma_k/q_\beta$ into disjunctive normal form over literals. This requires no more than exponential time on the number of variables, i.e., on the number of subformulae of the original rule, which is the same as for the reduction of any Boolean formula to disjunctive normal form. \square

The reduced normal form, obtained in Lemma 1, is uniquely defined and is denoted by $\text{rnf}(r)$. Note that Lemma 1 proves more than the equivalence of r and $\text{rnf}(r)$. In particular, from the proof it follows that if r is refutable in a model N then $\text{rnf}(r)$ is refutable in a definable variant M of N with $M, w \models q_\gamma \iff N, w \models \gamma$ for all $\gamma \in \text{sub}(\alpha, \beta)$.

Let r be any rule in reduced normal form (rnf). Let $\Theta(r) \stackrel{\text{def}}{=} \{\phi_1, \dots, \phi_s\}$ be the set of all disjuncts of r . For every $\phi_j \in \Theta(r)$, let

$$\theta(\phi_j) \stackrel{\text{def}}{=} \{p_i \mid t(i, j, 0) = 1\} \quad \text{and} \quad \theta_\diamond(\phi_j) \stackrel{\text{def}}{=} \{p_i \mid t(i, j, 1) = 1\}.$$

That is, $\theta(\phi_j)$ denotes the set of variables of r with positive occurrences in ϕ_j , and $\theta_\diamond(\phi_j)$ is the set of variables p_i of r with the positive occurrence of $\diamond p_i$ in ϕ_j .

Historically the first algorithm for recognising admissible rules of S4 was based on the next theorem. Its formulation requires the following definition, which is taken from [11]. For every subset of disjuncts $W \subseteq \Theta(\text{rnf}(r))$, let $\mathcal{M}(\text{rnf}(r), W)$ denote the Kripke model in which $W^M \stackrel{\text{def}}{=} W$,

$$R^M \stackrel{\text{def}}{=} \{(\phi_1, \phi_2) \mid \theta_\diamond(\phi_2) \subseteq \theta_\diamond(\phi_1)\} \quad \text{and} \quad (p_i, \phi_j) \in \nu^M \stackrel{\text{def}}{\iff} p_i \in \theta(\phi_j).$$

Theorem 3 (Theorem 3.9.6 [13]). *A rule $\text{rnf}(r)$ is admissible for S4 iff for any set $W \subseteq \Theta(\text{rnf}(r))$, the model $\mathcal{M}(\text{rnf}(r), W)$ fails to have at least one of the following properties.*

- (a1) *There is $\phi_j \in W$ such that $\mathcal{M}(\text{rnf}(r), W), \phi_j \not\models p_0$.*
- (a2) *$\mathcal{M}(\text{rnf}(r), W), \phi_j \models \phi_j$ for all $\phi_j \in W$.*
- (a3) *For any subset \mathcal{D} of \mathcal{M} there exists $\phi_j \in W$ such that*

$$\theta_\diamond(\phi_j) = \theta(\phi_j) \cup \bigcup_{\phi \in \mathcal{D}} \theta_\diamond(\phi).$$

Note that in (a3), \mathcal{D} can be empty.

Theorem 3 implies that we can employ this algorithm for recognising the admissibility of a rule r in S4 [11].

- Step 1. Reduce rule r to depth-one-form $\text{do}(r)$.
- Step 2. Construct from the $\text{do}(r)$ the reduced form $\text{rnf}(r)$.
- Step 3. For every subset W of the set of disjuncts of $\text{rnf}(r)$, check whether the conditions of Theorem 3 hold.
- Step 4. If the conditions (a1)–(a3) of Theorem 3 are satisfied for some W , then r is not admissible for S4, otherwise r is admissible for S4.

Step 1 can be done in polynomial time, Step 2 can be done in exponential time, and Step 3 can be done in exponential time. This means the time complexity of the algorithm is bounded by a doubly-exponential function in the length of r .

4 Semantic Characterisation of Admissibility

We now give a semantic characterisation of admissibility in S4. We say that an S4^u-model satisfies the *formula definable co-cover property* if the following (infinite) set of axioms hold (for $n > 0$):

$$\begin{aligned} (\text{FCCP}) \quad & \exists x \forall p (\nu(\diamond p, x) \rightarrow \nu(p, x)) \\ & \forall x_1 \cdots \forall x_n \exists x \forall p (R(x, x_1) \wedge \cdots \wedge R(x, x_n) \wedge \\ & \nu(\diamond p, x) \rightarrow (\nu(p, x) \vee \nu(\diamond p, x_1) \vee \cdots \vee \nu(\diamond p, x_n))). \end{aligned}$$

Let $S_{\mathcal{FCCP}}$ be the semantic specification consisting of the (FCCP) formulae and the formulae in S_{S4^u} . Let $\mathcal{FCCP}(\Sigma)$ be the class of all $S4^u$ -models of the signature Σ that satisfy all instances of the formulae of $S_{\mathcal{FCCP}}$ under substitutions of $\mathcal{L}_{S4^u}^u$ -formulae in Σ for propositional variables.

Let $S4^{u,+}$ be the modal logic with the language $\mathcal{L}_{S4^u}^u$ that has $S_{\mathcal{FCCP}}$ as semantic specification. The following theorem is a direct consequence of the definitions above and the fact that all n -characterising models $\text{Ch}_n(S4)$ for $S4$ satisfy the (FCCP) formulae.

Theorem 4. $S4^{u,+}$ is a conservative extension of $S4$.

Now we prove that $S4^{u,+}$ has the effective finite model property. Let Σ be a fixed signature and M an $S4^{u,+}$ -model of the signature Σ . We define the *filtrated* (through Σ) model \overline{M} as follows. The equivalence \sim on the set W^M is defined by $w \sim v \iff \tau(w) = \tau(v)$. Further, $[w] \stackrel{\text{def}}{=} \{v \in W^M \mid w \sim v\}$ and $W^{\overline{M}} \stackrel{\text{def}}{=} \{[w] \mid w \in W^M\}$. Next, $R^{\overline{M}} \stackrel{\text{def}}{=} \{([w], [v]) \mid M, v \models \psi \text{ implies } M, w \models \Diamond\psi \text{ for every } \Diamond\psi \in \Sigma\}$ and, finally, $\nu^{\overline{M}} \stackrel{\text{def}}{=} \{(\psi, [w]) \mid (\psi, w) \in \nu^M\}$ for every $\psi \in \Sigma$.

The following lemma can be proved by induction on the structure of formulae in Σ by verifying that all semantic conditions in $S_{\mathcal{FCCP}}$ hold.

Lemma 2 (Filtration Lemma). \overline{M} is an $S4^{u,+}$ -model of the signature Σ .

Note that by definition \overline{M} is Σ -differentiated and it is finite whenever Σ is finite.

Theorem 5 (The Effective Finite Model Property). Let ϕ be a formula and n the length of ϕ (i.e., the number of symbols in ϕ). If ϕ is satisfiable in an $S4^{u,+}$ -model (of the signature $\text{sub}(\phi)$) then it is satisfiable in a finite $S4^{u,+}$ -model (of the signature $\text{sub}(\phi)$) and its size does not exceed 2^n .

Theorem 6. $\alpha/\beta \in \text{Adm}(S4)$ iff $[u]\alpha \wedge \neg\beta$ is unsatisfiable in $S4^{u,+}$.

Proof. Suppose α/β is not admissible and p_1, \dots, p_n are all the propositional variables occurring in α and β . Then there is a model M —a definable variant of $\text{Ch}_n(S4)$ such that $M \not\models \alpha/\beta$. This model has the co-cover property and it is routine to transform M into an $S4^{u,+}$ -model satisfying $[u]\alpha \wedge \neg\beta$.

For the converse, let $\Sigma \stackrel{\text{def}}{=} \text{sub}([u]\alpha \wedge \neg\beta)$ and suppose $[u]\alpha \wedge \neg\beta$ is satisfiable in an $S4^{u,+}$ -model. Then it is satisfiable in a finite Σ -differentiated $S4^{u,+}$ -model M (of the signature Σ), by the Filtration Lemma. Let $\text{sub}_{\Diamond}(\alpha, \beta) \stackrel{\text{def}}{=} \{\Diamond\gamma \mid \gamma \in \text{sub}(\alpha, \beta)\} \cup \text{sub}(\alpha, \beta)$ for any \mathcal{L}_{S4} -formulae α and β . For every $w \in W^M$ let $\tau_{\Diamond}(w) \stackrel{\text{def}}{=} \{\gamma \in \text{sub}_{\Diamond}(\alpha, \beta) \mid M, w \models \gamma\}$ and

$$\phi(w) \stackrel{\text{def}}{=} \bigwedge_{\gamma \in \text{sub}(\alpha, \beta)} q_{\gamma}^{\chi(\gamma)} \wedge \bigwedge_{\gamma \in \text{sub}(\alpha, \beta)} \Diamond q_{\gamma}^{\chi(\Diamond\gamma)},$$

where χ is the characteristic function of the set $\tau_{\Diamond}(w)$. Let us consider the model $M^* \stackrel{\text{def}}{=} \langle W^{M^*}, R^{M^*}, \nu^{M^*} \rangle$, where

$$- W^{M^*} \stackrel{\text{def}}{=} \{\phi(w) \mid w \in W^M\},$$

- $(\phi(u), \phi(v)) \in R^{M^*} \stackrel{\text{def}}{\iff} (u, v) \in R^M$,
- $(q_\gamma, \phi(w)) \in \nu^{M^*} \stackrel{\text{def}}{\iff} M, w \models \gamma$.

Each $\phi(w)$ is a disjunct in reduced normal form $\text{rnf}(r)$. Therefore we have that

- $W^{M^*} \subseteq \Theta(\text{rnf}(r))$,
- $\langle W^{M^*}, R^{M^*} \rangle$ is isomorphic to the underlying frame of M ,
- M^* satisfies conditions (a1)–(a3) of Theorem 3.

By Theorem 3, $\text{rnf}(r)$ is not admissible, and hence neither is r .

5 Synthesising a Tableau Calculus

We now apply the method of [17] to generate a sound and constructively complete tableau calculus for $S4^{u,+}$. In order to apply the method we must ensure that the semantic specification $S_{\mathcal{FCCP}}$ of $S4^{u,+}$ is well-defined in the sense of [17]. First, $S_{\mathcal{FCCP}}$ must be normalised in the following sense: (i) all the formulae specifying semantics are \mathcal{L}_{S4}^u -open, i.e., they do not contain quantifiers of propositional variables, and (ii) all the formulae in $S_{\mathcal{FCCP}}$ are divided into three groups: positive and negative definitions of the semantics of the connectives of the logic, and a background theory that imposes frame conditions. It is also required that all formulae in the background theory do not contain any non-atomic modal terms.

Every definition of the \mathcal{L}_{S4}^u connectives in $S_{\mathcal{FCCP}}$ (in Figure 1 and (1)) can be split into two implications. The resulting set of formulae can be divided into the required two groups of positive and negative connective definitions. The third group, the background theory of $S4^{u,+}$, consists of formulae specifying the reflexivity and transitivity for the relation R and the (FCCP) formulae.

The main difficulties for the normalisation of the specification $S_{\mathcal{FCCP}}$ are the occurrences of the non-atomic modal term $\Diamond p$ and the quantification of the variable p in (FCCP). To solve this problem we first replace every formula $\nu(\Diamond p, y)$ by its semantic equivalent $\exists z(R(y, z) \wedge \nu(p, z))$ and transform the resulting formulae into the prenex normal form. This gives us:

$$\begin{aligned} & \exists x \forall p \forall y ((R(x, y) \wedge \nu(p, y)) \rightarrow \nu(p, x)) \\ & \forall x_1 \cdots \forall x_n \exists x \forall p \forall y \exists z (R(x, x_1) \wedge \cdots \wedge R(x, x_n) \wedge \\ & (R(x, y) \wedge \nu(p, y)) \rightarrow (\nu(p, x) \vee (\nu(p, z) \wedge (R(x_1, z) \vee \cdots \vee R(x_n, z))))). \end{aligned}$$

These formulae are still not \mathcal{L}_{S4}^u -open formulae as required in [17] because of the quantifiers on p . However, using Skolemisation it is possible to eliminate all existential quantifiers preceding the p -quantifiers in the formulae and then we can omit the quantifiers of p . In addition, we split the long formulae in two parts. We get:

$$\begin{aligned} \text{(FCCP')} \quad & \forall y ((R(g_0, y) \wedge \nu(p, y)) \rightarrow \nu(p, g_0)) \\ & \forall x_1 \cdots \forall x_n (R(g_n(x_1, \dots, x_n), x_1) \wedge \cdots \wedge R(g_n(x_1, \dots, x_n), x_n)) \\ & \forall x_1 \cdots \forall x_n \forall y \exists z ((R(g_n(x_1, \dots, x_n), y) \wedge \nu(p, y)) \rightarrow \\ & (\nu(p, g_n(x_1, \dots, x_n)) \vee (\nu(p, z) \wedge (R(x_1, z) \vee \cdots \vee R(x_n, z))))). \end{aligned}$$

Decomposition tableau rules:

$$\begin{array}{c}
 \frac{\nu(\neg p, x)}{\neg\nu(p, x)} \\
 \frac{\nu(p \vee q, x)}{\nu(p, x) \mid \nu(q, x)} \\
 \hline
 \nu(\diamond p, x) \\
 \hline
 R(x, f_\diamond(p, x)), \nu(p, f_\diamond(p, x)) \\
 \frac{\nu(\langle u \rangle p, x)}{\nu(p, f_u(p))}
 \end{array}
 \qquad
 \begin{array}{c}
 \frac{\neg\nu(\neg p, x)}{\nu(p, x)} \\
 \frac{\neg\nu(p \vee q, x)}{\neg\nu(p, x), \neg\nu(q, x)} \\
 \hline
 \nu(\neg\diamond p, x) \\
 \hline
 \neg R(x, y) \mid \neg\nu(p, y) \\
 \frac{\nu(\neg\langle u \rangle p, x), y \approx y}{\neg\nu(p, y)}
 \end{array}$$

Theory tableau rules:

$$\frac{x \approx x}{R(x, x)} \qquad \frac{x \approx x, y \approx y, z \approx z}{\neg R(x, y) \mid \neg R(y, z) \mid R(x, z)}$$

Infinite set of (FCCP) tableau rules ($n > 0$):

$$\begin{array}{c}
 (\text{cc}^0): \frac{p \approx p, y \approx y}{\neg R(g_0, y) \mid \neg\nu(p, y) \mid \nu(p, g_0)} \\
 (\text{cc}_0^n): \frac{x_1 \approx x_1, \dots, x_n \approx x_n, y \approx y}{R(g_n(\bar{x}), x_1), \dots, R(g_n(\bar{x}), x_n)} \\
 (\text{cc}_1^n): \frac{p \approx p, x_1 \approx x_1, \dots, x_n \approx x_n, y \approx y}{\neg R(g_n(\bar{x}), y) \mid \neg\nu(p, y) \mid \nu(p, g_n(\bar{x})) \mid R(x_1, h_n(p, \bar{x}, y)), \nu(p, h_n(p, \bar{x}, y)) \mid \dots}
 \end{array}$$

Closure tableau rules:

$$\frac{\nu_1(p, x), \neg\nu_1(p, x)}{\perp} \qquad \frac{R(x, y), \neg R(x, y)}{\perp}$$

Figure 2. Generated tableau rules for $S4^{u,+}$

Here, g_n ($n \geq 0$) denote the introduced Skolem symbols.

We use the notation $S'_{\mathcal{FCCP}}$ for the semantic specification obtained from $S_{\mathcal{FCCP}}$ where all (FCCP) formulae have been replaced by the corresponding (FCCP') formulae. It is clear that for every first-order structure M , the universal closure of $S_{\mathcal{FCCP}}$ and the universal closure of $S'_{\mathcal{FCCP}}$ are equisatisfiable in M . Hence, the transformed semantics $S'_{\mathcal{FCCP}}$ is equivalent to $S_{\mathcal{FCCP}}$ and axiomatises the same class of models.

Now we are ready to synthesise tableau calculi from $S'_{\mathcal{FCCP}}$. The generated tableau rules are given in Figure 2. The symbols f_\diamond , f_u , g_n , h_n denote Skolem functions and g_0 denotes a Skolem constant.

Let T be the tableau calculus consisting of the rules of Figure 2 and the standard tableau rules for equality. The equality tableau rules are obtained from the equality congruence axioms, which are always included in the background theory of any semantic specification, see [17].

Given a formula ϕ , and assuming our aim is to determine the satisfiability of ϕ , the start of any tableau derivation is the formula $\nu(\phi, a)$, where a is an arbitrary constant a that does not occur in the rules of T . a can be viewed as

a Skolem constant introduced for $\exists x$ in the formula $\exists x \nu(\phi, x)$. The rules of the calculus are applied top-down in the familiar way.

We assume the following definitions from [16,17]. Let T denote a tableau calculus and ϕ is a formula. We denote by $T(\phi)$ a finished tableau derivation for testing the satisfiability of ϕ . That is, all branches in the tableau derivation are fully expanded and all applicable rules of T have been applied in $T(\phi)$. As usual we assume that all the rules of the calculus are applied non-deterministically. A branch of a tableau derivation is *closed* if a closure rule has been applied, otherwise the branch is called *open*. The tableau derivation $T(\phi)$ is *closed* if all its branches are closed and $T(\phi)$ is *open* otherwise. A calculus T is *sound* (for a logic L) iff for any formula ϕ , each $T(\phi)$ is open whenever ϕ is satisfiable in an L -model. T is *complete* iff for any unsatisfiable formula ϕ there is a $T(\phi)$ which is closed. T is *constructively complete* (for L) iff for any open branch in a finished tableau derivation in T it is possible to construct an L -model from terms of the branch such that the model reflects all the formulae occurring in the branch. (Constructive completeness is a stronger notion than completeness.) T is said to be *terminating* if every finished open tableau derivation in T has a finite open branch.

It is easy to check that the specification $S'_{\mathcal{FCCP}}$ for $S4^{u,+}$ is well-defined in the sense of [17]. A consequence of [17, Theorems 1 and 2] is this result.

Theorem 7. *T is a sound and constructively complete calculus for $S4^{u,+}$.*

6 A Refined Tableau Calculus

The generated calculus T can be refined as follows. First, the rules (cc_1^n) can be replaced by these equivalent rules:

$$\frac{p \approx p, \quad x_1 \approx x_1, \quad \dots, \quad x_n \approx x_n, \quad y \approx y}{\neg R(g_n(\bar{x}), y) \mid \neg \nu(p, y) \mid \nu(p, g_n(\bar{x})) \mid \nu(\diamond p, x_1) \mid \dots \mid \nu(\diamond p, x_n)}$$

It is clear that the rules are sound for $S4^{u,+}$. Furthermore, the rules (cc_1^n) are derivable from these rules and the diamond rule.

Second, it is possible to refine the calculus by moving negated conclusions in certain rules up to numerator positions. We move formulae that contain only propositional variables and do not contain any complex modal terms upwards. It is not difficult to check for each rule that the condition given in [17, Theorem 3] for this refinement to preserve soundness and constructive completeness is true.

Third, we can apply the second refinement described in [17, Section 5]. For this we extend the language \mathcal{L}_{S4}^u by introducing a countable set $\{i, j, k, \dots\}$ of nominal variables and logical connectives (acting on nominals) which correspond to Skolem functions and constants. The @ operator can be introduced and specified in such a way that $@_i \phi \stackrel{\text{def}}{=} [u](i \rightarrow \phi)$ for every nominal term i and formula ϕ (of the extended language). It is not difficult to see that the semantics of all the connectives of the extended language can be represented in the language itself (see [17, Section 5]).

Decomposition tableau rules:

$$\begin{array}{l}
(\neg): \frac{\@_i \neg \neg p}{\@_i p} \\
(\vee): \frac{\@_i (p \vee q)}{\@_i p \mid \@_i q} \\
(\diamond): \frac{\@_i \diamond p}{\@_i \diamond f_\diamond(p, i), \ \@_{f_\diamond(p, i)} p} \\
((u)): \frac{\@_i (u) p}{\@_{f_u(p)} p}
\end{array}
\qquad
\begin{array}{l}
(\neg\vee): \frac{\@_i \neg (p \vee q)}{\@_i \neg p, \ \@_i \neg q} \\
(\neg\diamond): \frac{\@_i \neg \diamond p, \ \@_i \diamond j}{\@_j \neg p} \\
(\neg(u)): \frac{\@_i \neg (u) p, \ \@_j j}{\@_j \neg p}
\end{array}$$

Theory tableau rules:

$$(\text{refl}): \frac{\@_i i}{\@_i \diamond i} \qquad (\text{trans}): \frac{\@_i \diamond j, \ \@_j \diamond k}{\@_i \diamond k}$$

Infinite set of (FCCP') tableau rules ($n > 0$):

$$\begin{array}{l}
(\text{cc}'_0): \frac{\@_{g_0} \diamond i, \ \@_i p}{\@_{g_0} p} \\
(\text{cc}'_0): \frac{\@_{i_1} i_1, \ \dots, \ \@_{i_n} i_n}{\@_{g_n(\bar{i})} \diamond i_1, \ \dots, \ \@_{g_n(\bar{i})} \diamond i_n} \\
(\text{cc}'_1): \frac{\@_{g_n(\bar{i})} \diamond j, \ \@_j p}{\@_{g_n(\bar{i})} p \mid \@_{i_1} \diamond p \mid \dots \mid \@_{i_n} \diamond p}
\end{array}$$

Closure tableau rules:

$$(\perp): \frac{\@_i p, \ \@_i \neg p}{\perp}$$

Figure 3. Refined tableau rules for $\mathbf{S4}^{u,+}$

Summing up, the refined rules we obtain are given in Figure 3. In these rules, i, j, k, i_1, \dots, i_n denote nominal variables, and f_\diamond, f_u , and g_n ($n \geq 0$) denote ‘nominal functions’ which correspond to the Skolem functions with same names.

Let T^+ be the tableau calculus which consists of the rules of Figure 3 and the refined equality rules given in Figure 4. In T^+ , a tableau derivation for testing the satisfiability of ϕ starts with a formula $\@_{i_0} \phi$ where i_0 is a *fresh* nominal constant.

Theorem 8. T^+ is a sound and constructively complete tableau calculus for $\mathbf{S4}^{u,+}$.

7 A Terminating Tableau Calculus

Our proof of Theorem 5 that $\mathbf{S4}^{u,+}$ has the effective finite model property uses a filtration argument. That is, in the terminology of [16], $\mathbf{S4}^{u,+}$ admits finite filtration. Using the results of [16], this means that the tableau calculi generated in Section 5 and 6 can be turned into terminating tableau calculi. In particular, we are interested only in the refined calculus T^+ .

$$\begin{array}{l}
(\text{ref}\approx): \frac{\@_i p}{\@_i i} \quad (\text{sym}\approx): \frac{\@_i j}{\@_j i} \quad (\text{trans}\approx): \frac{\@_i j, \@_j k}{\@_i k} \\
(\text{con}\approx_0): \frac{\@_i p, \@_i j}{\@_j p} \quad (\text{con}\approx_1): \frac{\@_i \diamond j, \@_j k}{\@_i \diamond k}
\end{array}$$

Figure 4. Refined equality congruence rules

Adding the following unrestricted blocking rule to T^+ gives a terminating tableau calculus.

$$(\text{ub}): \frac{\@_i i, \@_j j}{\@_i j \mid \@_i \neg j}$$

The conditions that blocking must satisfy are:

- (b1) The rules (\diamond) and $(\langle u \rangle)$ are never applied to formulae of the form $\@_i \diamond j$ and, respectively, $\@_i \langle u \rangle j$ where j is a nominal term.
- (b2) If $\@_i j$ appears in a branch and $i < j$ (i.e., nominal i appeared strictly earlier than nominal j in the derivation) then all further applications of the tableau rules which produce new nominals (in our case the (\diamond) , $(\langle u \rangle)$, (cc^0) and (cc'_0) rules) to the formulae with occurrences of j are not performed within the branch.
- (b3) In every open branch there is some node from which point onwards before any application of any tableau rule that produces new nominals (i.e., the (\diamond) , $(\langle u \rangle)$, (cc^0) and (cc'_0) rules) all possible applications of the (ub) rule have been performed.

We denote the extended calculus by $T_{S4^{u,+}}$.

Since T^+ is sound and constructively complete for $S4^{u,+}$, and $S4^{u,+}$ admits finite filtration the results in [15] allow us to state:

Theorem 9. $T_{S4^{u,+}}$ is a sound, (constructively) complete and terminating tableau calculus for $S4^{u,+}$.

Let T_{S4^u} be the tableau calculus which consists of the same set of rules as $T_{S4^{u,+}}$ but excludes the (FCCP') rules: (cc^0) , (cc'_0) , and (cc'_1) . Applying tableau synthesis to $S4^u$ in a similar way gives the following result.

Theorem 10. T_{S4^u} is a sound, (constructively) complete and terminating tableau calculus for $S4^u$.

8 A Tableau Method for Testing Rule Admissibility

Putting all the results together (in particular Theorems 1, 6, 9 and 10) here is a method for determining whether a modal rule is admissible in S4, or not.

Step 1. Given an S4-rule α/β , rewrite it to $[u]\alpha \wedge \neg\beta$.

Step 2. Use the tableau calculus T_{S4^u} to test the satisfiability of $[u]\alpha \wedge \neg\beta$ in $S4^u$.

- Step 3. If $T_{S4^u}([u]\alpha \wedge \neg\beta)$ is closed, i.e., $[u]\alpha \wedge \neg\beta$ is unsatisfiable in $S4^u$, then stop and return **'derivable'**.
- Step 4. Otherwise (i.e., $T_{S4^u}([u]\alpha \wedge \neg\beta)$ is open), *continue* the tableau derivation with the rules in T_{S4^u} plus the rules (cc'^0) , (cc'_0^n) , and (cc'_1^n) . In particular, continue the derivation with a finite open branch of $T_{S4^u}([u]\alpha \wedge \neg\beta)$ using the rules of $T_{S4^{u,+}}$ until the derivation stops.
- Step 5. If $T_{S4^{u,+}}([u]\alpha \wedge \neg\beta)$ is closed then return **'not derivable and admissible'**. Otherwise, i.e., if $T_{S4^{u,+}}([u]\alpha \wedge \neg\beta)$ is open, return **'not admissible'**.

The answers returned by the method are either **'derivable'**, **'not derivable and admissible'**, or **'not admissible'**. If a rule is derivable it is also admissible (but not conversely).

9 Concluding Remarks

A major difficulty in dealing with $S4$ -admissibility, is that the theory of $S4$ -admissible rules does not have the finite approximation property [7], in this sense: for a rule $r \notin \text{Adm}(S4)$, there is no single finite Kripke model M *separating* r from $\text{Adm}(S4)$, (i.e., such that $M \models \text{Adm}(S4)$, but $M \not\models r$). Therefore the tableau algorithm that we have introduced in this paper builds open branches that represent a (possibly) infinite $\text{Adm}(S4)$ -model and is a counter-model to a non-admissible rule. The subsequent filtration provides a finite model (not necessarily an $\text{Adm}(S4)$ -model) witnessing the refutation.

A known algorithm that can handle $S4$ -admissibility appears in Zucchelli [19] and is based on the research of Ghilardi [3,4,5] on projective approximations. From the abstract [19], it follows that this algorithm combines resolution and tableaux approaches for finding projective approximations of a formula and relies on the existence of an algorithm for theorem proving. This algorithm is specifically constructed for describing general solutions (maximal general unifiers, maximal since there can be more than one) for matching and unification problems (all other solutions can be obtained as substitution variants of general solutions). Applicability of this algorithm to the admissibility problem is a side effect, depending on some specific properties of $S4$ (see [5]). In contrast, the original algorithm of [11] can be used to find only *some* solution of matching and unification problems in $S4$. In particular, this can be done through the relation:

$$\text{an equation } \alpha \equiv \beta \text{ is solvable} \quad \text{iff} \quad \text{a rule } \alpha \equiv \beta / \perp \text{ is not admissible.}$$

We expect that our method can be modified for finding the general solutions of logical equations as well.

Recently Wolter and Zakharyashev [18] showed that modal logics with the *universal modality* that are situated between K^u and $K4^u$ are undecidable with respect to admissibility and even with respect to just unification. They posed the question [18] whether the logic $S4^u$ is decidable with respect to admissibility. This question is solved positively in Rybakov [14]. We strongly believe that our tableau method can also be extended to rule admissibility in $S4^u$.

In fact, our method of replacing the first-order co-cover condition with its *formulaic* variant (which is also first-order but in the extended language) is not restricted to S4. We expect that it can be modified to deal with a number of other modal logics, especially those covered by the general theory of [13]. Similarly, it can be applied to their superintuitionistic counterparts (either through Gödel's translation or directly) and to transitive modal logics augmented with the universal modality [14].

References

1. S. Babenyshev. The decidability of admissibility problems for modal logics S4.2 and S4.2Grz and superintuitionistic logic KC. *Algebra and Logic*, 31(4):205–216, July 1992.
2. H. Fridman. One hundred and two problems in mathematical logic. *J. Symbolic Logic*, 40(3):113–130, 1975.
3. S. Ghilardi. Unification in intuitionistic logic. *J. Symbolic Logic*, 64(2):859–880, 1999.
4. S. Ghilardi. Best solving modal equations. *Ann. Pure Appl. Logic*, 102(3):183–198, 2000.
5. S. Ghilardi and L. Sacchetti. Filtering unification and most general unifiers in modal logic. *J. Symbolic Logic*, 69(3):879–906, 2004.
6. R. Harrop. Concerning formulas of the types $a \rightarrow b \vee c$, $a \rightarrow \exists x b(x)$ in intuitionistic formal system. *J. Symbolic Logic*, 25:27–32, 1960.
7. V. R. Kiyatkin, V. V. Rybakov, and T. Oner. On finite model property for admissible rules. *Math. Logic Quarterly*, 45:505–520, 1999.
8. P. Lorenzen. *Einführung in die operative Logik und Mathematik*. Springer, 1955.
9. G. Mints. Derivability of admissible rules. *J. Soviet Math.*, 6(4):417–421, 1976.
10. P. Roziere. Admissible and derivable rules. *Math. Structures Computer Sci.*, (3):129–136, 1993.
11. V. V. Rybakov. A criterion for admissibility of rules in modal system S4 and the intuitionistic logic. *Algebra and Logic*, 23(5):369–384, 1984.
12. V. V. Rybakov. Rules of inference with parameters for intuitionistic logic. *J. Symbolic Logic*, 57(3):912–923, 1992.
13. V. V. Rybakov. *Admissibility of logical inference rules*. Elsevier, 1997.
14. V. V. Rybakov. Logics with universal modality and admissible consecutions. *J. Appl. Non-Classical Logics*, 17(3):381–394, 2007.
15. R. A. Schmidt and D. Tishkovsky. Using tableau to decide expressive description logics with role negation. In *Proc. ISWC'07*, vol. 4825 of *Lect. Notes Comput. Sci.*, pp. 438–451. Springer, 2007.
16. R. A. Schmidt and D. Tishkovsky. A general tableau method for deciding description logics, modal logics and related first-order fragments. In *Proc. IJCAR'08*, vol. 5195 of *Lect. Notes Comput. Sci.*, pp. 194–209. Springer, 2008.
17. R. A. Schmidt and D. Tishkovsky. Automated synthesis of tableau calculi. In *Proc. TABLEAUX'09*, vol. 5607 of *Lect. Notes Artif. Intell.*, pp. 310–324. Springer, 2009.
18. F. Wolter and M. Zakharyashev. Undecidability of the unification and admissibility problems for modal and description logics. *ACM Trans. Comput. Logic*, 9(4):1–20, 2008.
19. D. Zucchelli. Studio e realizzazione di algoritmi per l'unificazione nelle logiche modali. Laurea specialistica in informatica (Masters Thesis), Università degli Studi di Milano, July 2004. In Italian. Supervisor: Silvio Ghilardi.

Universality of Polynomial Positivity and a Variant of Hilbert’s 17th Problem

Grant Olney Passmore and Leonardo de Moura
{g.passmore@ed.ac.uk, leonardo@microsoft.com}

LFCS, University of Edinburgh and Microsoft Research

Abstract. We observe that the decision problem for the \exists theory of real closed fields (RCF) is simply reducible to the decision problem for RCF over a connective-free \forall language in which the only relation symbol is a strict inequality. In particular, every \exists RCF sentence φ can be settled by deciding a proposition of the form “polynomial p (which is a sum of squares) takes on strictly positive values over the reals,” with p simply derived from φ . Motivated by this observation, we pose the goal of isolating a syntactic criterion characterising the positive definite (i.e., strictly positive) real polynomials. Such a criterion would be a strictly positive analogue to the fact that every positive semidefinite (i.e., non-negative) real polynomial is a sum of squares of rational functions, as established by Artin’s positive solution to Hilbert’s 17th Problem. We then prove that every positive definite real polynomial is a ratio of a Real Nullstellensatz witness and a positive definite real polynomial. Finally, we conjecture that every positive definite real polynomial is a product of ratios of Real Nullstellensatz witnesses and examine an interesting ramification of this conjecture¹.

1 Motivation

Real polynomial (non-strict) positivity is a classically studied topic in real algebraic geometry with many applications to mainstream mathematics, theoretical computer science, engineering, and the natural sciences. For example, the termination of rewriting systems [5], the construction of Lyapunov functions for proving the stability of dynamical systems [3][7], and the distinguishing between separable and entangled quantum states [1] are problems that can be in many cases solved by proving the non-negativity of certain associated polynomials in $\mathbb{R}[\mathbf{x}]$.

¹ We are extremely grateful to a number of helpful anonymous referees. In particular, the first referee pointed out that our main conjecture (Conjecture 1) is in fact known to be true, and follows as a simple corollary of the Krivine-Stengle Positivstellensatz. This referee then even took the time to construct an original proof of this conjecture, given that the original references were not easily accessible. We must express our true appreciation for this amazingly generous referee. We will attach this referee’s proof as an appendix to our paper.

The work in this paper is motivated primarily by the wish for improved decision methods for the quantifier-free fragment of the elementary theory of real closed fields. In particular, we observe that the satisfiability of any boolean combination of polynomial equations and inequalities over \mathbb{R}^n can be reduced to the strict positivity of an associated polynomial that is itself a sum of squares of real polynomials. Wishing to make use of this observation in practice, we then ask: What form must a strictly positive real polynomial have?

Artin's positive solution to Hilbert's 17th Problem established a syntactic criterion for the non-negativity of a real polynomial, namely that every non-negative real polynomial is a sum of squares of real rational functions. Is there an analogous, sharper criterion for the strictly positive real polynomials?

2 Preliminaries

In the sequel, let $\mathbb{R}[\mathbf{x}]$ denote the polynomial ring $\mathbb{R}[x_1, \dots, x_n]$, \mathbb{R}^+ the set of strictly positive reals, \mathcal{L} the elementary language of ordered rings, and $QF_{\mathcal{L}}$ the collection of quantifier-free \mathcal{L} -formulæ. We first recall some basic real algebraic preliminaries and then observe that the decision problem for the existential fragment of RCF is simply reducible to the problem of whether or not a sum of squares of real polynomials is positive definite.

Definition 1. A polynomial $p \in \mathbb{R}[\mathbf{x}]$ is positive definite (resp. semidefinite) iff $\forall \mathbf{r} \in \mathbb{R}^n (p(\mathbf{r}) > 0)$ (resp. $\forall \mathbf{r} \in \mathbb{R}^n (p(\mathbf{r}) \geq 0)$). We say p is PD (resp. PSD) if p is positive definite (resp. semidefinite).

Definition 2. A set $S \subseteq \mathbb{R}^n$ is semialgebraic iff $S = \{\mathbf{r} \in \mathbb{R}^n \mid \langle \mathbb{R}, +, -, *, <, 0, 1 \rangle \models \varphi(\mathbf{r})\}$ for some $\varphi \in QF_{\mathcal{L}}$.

Definition 3. A real algebraic variety is the locus of real zeros of a finite system of real polynomials. That is, given $p_1, \dots, p_m \in \mathbb{R}[\mathbf{x}]$,

$$\begin{aligned} \mathcal{V}_{\mathbb{R}}(p_1, \dots, p_m) &= \{\mathbf{r} \in \mathbb{R}^n \mid p_1(\mathbf{r}) = 0 \wedge \dots \wedge p_m(\mathbf{r}) = 0\} \\ &= \mathcal{V}(p_1, \dots, p_m) \cap \mathbb{R}^n \end{aligned}$$

where $\mathcal{V}(p_1, \dots, p_m)$ is the (complex) algebraic variety of classical algebraic geometry.

Observe that every real algebraic variety is semialgebraic.

Definition 4. The projection $\Pi_X : 2^{\mathbb{R}^n} \rightarrow 2^{\mathbb{R}^{n-k}}$ of a semialgebraic set $S \subseteq \mathbb{R}^n$ onto a lower-dimensional Euclidean space w.r.t. a collection of eliminated coordinates (wlog) $X = \{1, \dots, k\}$ ($1 \leq k \leq n$) is defined as follows:

$$\Pi_X(S) = \{\langle r_{k+1}, \dots, r_n \rangle \in \mathbb{R}^{n-k} \mid \langle r_1, \dots, r_n \rangle \in S\}.$$

As every semialgebraic set $S \subseteq \mathbb{R}^n$ is of the form $S = \{\mathbf{r} \in \mathbb{R}^n \mid \langle \mathbb{R} \rangle \models \varphi(\mathbf{r})\}$ for some $\varphi \in QF_{\mathcal{L}}$, we can describe the geometric operation of projection strictly in terms of operations upon formulæ:

$$\Pi_X(S) = \{\langle r_{k+1}, \dots, r_n \rangle \in \mathbb{R}^{n-k} \mid \langle \mathbb{R} \rangle \models \exists x_1 \dots \exists x_k \varphi(x_1, \dots, x_k, r_{k+1}, \dots, r_n)\},$$

where $\langle \mathbb{R} \rangle$ is an abbreviation for $\langle \mathbb{R}, +, -, *, <, 0, 1 \rangle$.

And so we see that the geometric operation of projection corresponds to the logical operation of existential quantification. Surprisingly, adding a projection operator to $QF_{\mathcal{L}}$ does not increase the expressive power of the language w.r.t. the collections of real vectors that can be defined. The following important result is what ultimately allows the theory of real closed fields to admit elimination of quantifiers.

Theorem 1 (Tarski-Seidenberg). *The collection of semialgebraic sets is closed under projection.*

It is a marvelous fact that quantifier-free equivalents of all \mathcal{L} -formulae can be found algorithmically [6], though infeasibly.

3 Universality of Polynomial Positivity

We now use the well-known Rabinowitsch encoding to observe the folklore² result that every \exists RCF sentence is equivalent to the \exists -closure of a single polynomial equation. We phrase the result geometrically in terms of projection. We then observe that with a simple application of the Intermediate Value Theorem, which holds over every RCF, we can reduce the truth of any \exists RCF formula to (the falsity of) a statement proclaiming the strict positivity of a derived polynomial which is a sum of squares of real polynomials.

Theorem 2. *Every semialgebraic set is the projection of a real algebraic variety.*

Proof. Let $S \subseteq \mathbb{R}^n$ be given as $S = \{\mathbf{r} \in \mathbb{R}^n \mid \langle \mathbb{R}, +, -, *, <, 0, 1 \rangle \models \varphi(\mathbf{r})\}$ for some $\varphi \in QF_{\mathcal{L}}$. By induction on φ (which is quantifier-free) we obtain an equivalent RCF formula Ψ whose quantifier-free matrix consists of a single polynomial equation. This is done by using the Rabinowitsch equivalences (where z is a fresh variable):

$$\begin{aligned} (p \neq 0) &\iff \exists z(pz - 1) = 0, \\ (p \leq q) &\iff \exists z(q - p - z^2 = 0), \\ (p < q) &\iff \exists z(q - p)z^2 - 1 = 0, \\ \bigwedge_{i=1}^v p_i = 0 &\iff \sum_{i=1}^v (p_i)^2 = 0, \\ \bigvee_{i=1}^v p_i = 0 &\iff \prod_{i=1}^v p_i = 0. \end{aligned}$$

² This observation appears already in 1954 in Seidenberg's seminal paper [4]. We are grateful to an anonymous reviewer for pointing this out.

Note that the obtained Ψ is quantifier-free iff φ is a negation-free boolean combination of polynomial equations. In that case, the projection used to obtain S from the real algebraic variety defined by Ψ is the trivial projection, and thus S is itself a real algebraic variety. Otherwise, $\Psi = \exists \mathbf{x} \exists \mathbf{z} (p(\mathbf{x}, \mathbf{z}) = 0)$ for some $p \in \mathbb{R}[\mathbf{x}, \mathbf{z}]$ s.t. $\mathbf{z} = \langle z_1, \dots, z_k \rangle$ and k is the number of inequality symbols appearing in φ .

The following immediate corollary places an upper-bound on the dimension of the ambient Euclidean space containing a real algebraic variety whose projection is S .

Corollary 1. *If $S \subseteq \mathbb{R}^n$ is semialgebraic, then $S = \Pi_Z(T)$ where $T \subseteq \mathbb{R}^{n+k}$ is a real algebraic variety s.t. $k = |Z|$ is the total number of inequality and negated-equality symbols appearing in a defining $QF_{\mathcal{L}}$ formula for S .*

In fact, by a difficult construction of Motzkin [2], the following stronger result is known:

Theorem 3. *If $S \subseteq \mathbb{R}^n$ is semialgebraic, then $S = \Pi_{\{z\}}(T)$ where $T \subseteq \mathbb{R}^{n+1}$ is a real algebraic variety. Note that in this construction, T is defined by an equation with only one more variable (z) than appear in a $QF_{\mathcal{L}}$ formula defining S .*

With these results in hand, we are ready to make the following simple observation which motivates the rest of the work in this article:

Theorem 4. *The decision problem for the \exists theory of RCF is simply reducible to the decision problem for RCF over a connective-free \forall language in which the only relation symbol is a strict inequality. In particular, every \exists RCF sentence φ can be settled by deciding a proposition of the form “polynomial \mathbb{P} takes on strictly positive values over the reals,” where \mathbb{P} is simply derived from φ .*

Proof. We show that any $\exists \mathcal{L}$ -sentence φ can be either trivially decided by ground evaluation or by deciding a statement of the form “polynomial p takes on strictly positive values over \mathbb{R} .” Let Ψ be the sentence obtained from φ via the inductive process in Theorem 2 s.t. $(\varphi \iff \Psi) \in RCF$. Note that $\Psi = \exists \mathbf{x} \exists \mathbf{z} (p(\mathbf{x}, \mathbf{z}) = 0)$ for some $p \in \mathbb{R}[\mathbf{x}, \mathbf{z}]$ s.t. $\mathbf{z} = \langle z_1, \dots, z_k \rangle$. Let $\theta(p)$ be the degree zero constant term of p and let $\mathbf{0}$ be the zero vector in \mathbb{R}^{n+k} . If $\theta(p) = 0$ then $p(\mathbf{0}) = 0$ and so Ψ (and hence φ) holds. Otherwise, we have either $\theta(p) > 0$ or $\theta(p) < 0$. Let Ψ' be s.t. $\Psi' = \Psi$ if $\theta(p) > 0$ and $\Psi' = \exists \mathbf{x} \exists \mathbf{z} ((-1) * p(\mathbf{x}, \mathbf{z}) = 0)$ otherwise. Denote the resulting LHS polynomial in Ψ' as \mathbb{P} . As the Intermediate Value Theorem holds over every real closed field, we now have

$$\neg \varphi \iff \forall \mathbf{x} \forall \mathbf{z} (\mathbb{P}(\mathbf{x}, \mathbf{z}) > 0).$$

And so the truth of φ can be settled by deciding the strict positivity of $\mathbb{P}(\mathbf{x}, \mathbf{z})$ and negating the result.

The following corollary makes it clear that a method for deciding whether or not a sum of squares of real polynomials is PD is sufficient for the \exists theory of RCF.

Corollary 2. *Strict polynomial positivity for sums of squares of real polynomials is complete for the \exists theory of the reals.*

Proof. This is immediate, as the formula $\Psi = \exists \mathbf{x} \exists \mathbf{z} (p(\mathbf{x}, \mathbf{z}) = 0)$ obtained in the inductive construction of Theorem 2 is either already of the form $\exists \mathbf{x} \exists \mathbf{z} (\sum (p_i(\mathbf{x}, \mathbf{z}))^2 = 0)$ (e.g., by top-level applications of the encoding of conjunction: $(p = 0) \wedge (q = 0) \iff p^2 + q^2 = 0$), or it is of the form $\exists \mathbf{x} \exists \mathbf{z} (p(\mathbf{x}, \mathbf{z}) = 0)$ where p is not a sum of squares of real polynomials. In the latter case, note that $\exists \mathbf{x} \exists \mathbf{z} (p(\mathbf{x}, \mathbf{z}) = 0)$ is equivalent to $\exists \mathbf{x} \exists \mathbf{z} ((p(\mathbf{x}, \mathbf{z}))^2 = 0)$ and the result follows.

4 A PD Variant of Hilbert’s 17th Problem

Recall that a real polynomial is PSD iff it assumes only non-negative values for all real values of its variables. Artin’s positive solution to Hilbert’s 17th Problem characterises the PSD real polynomials with the following syntactic criterion:

Theorem 5 (Artin’s Positive Solution to Hilbert’s 17th Problem). *Every PSD real polynomial is a sum of squares of rational functions.*

The allowance of *rational functions* above is necessary to obtain a complete characterisation, as there are real polynomials (such as the dehomogenized bivariate Motzkin form, $x^4y^2 + x^2y^4 - 3x^2y^2 + 1$) that are PSD but not sums of squares of real polynomials. By the results in the previous section, we know that the truth of any \exists RCF sentence φ can be settled by deciding if a certain real polynomial derived from φ is PD. Thus, we are interested in obtaining a syntactic criterion characterising the class of PD polynomials. Such a criterion for PD polynomials would be analogous to that of PSD polynomials that is given in Hilbert’s 17th Problem. Indeed, it should have the property that every real polynomial meeting the criterion is PD, just as every real polynomial meeting the criterion of being a sum of squares of rational functions is PSD.

In working to obtain a syntactic criterion for PD polynomials, a natural first attempt might take the following form: Every PD polynomial p is PSD, so by Artin’s Theorem p is a sum of squares of rational functions. But, as p never obtains 0, p must have a positive constant term. Thus, we have

$$p = \sum_{i=1}^k \left(\frac{q_i}{s_i} \right)^2 = \mathbf{p} + c$$

s.t. $q_i, s_i \in \mathbb{R}[\mathbf{x}]$ and (i) $\mathbf{p}(\mathbf{0}) = 0$, (ii) $c \in \mathbb{R}^+$, and (iii) $\forall \mathbf{r} \in \mathbb{R}^n (\mathbf{p}(\mathbf{r}) > -c)$. All of these requirements would certainly be met if p was of the form

$$p = \left(\sum_{i=1}^k \left(\frac{q_i}{s_i} \right)^2 \right) + r$$

with $r \in \mathbb{R}^+$. And so it is natural to ask the following question.

Question 1. Is every PD real polynomial a sum of a sum of squares of rational functions and a positive constant?

We will construct a simple example showing that this is not true in general.

Definition 5. Let $(\sum(\mathbb{R}[\mathbf{x}])^2 + \mathbb{R}^+) = \{(\sum_{i=1}^m (p_i)^2) + c \mid p_i \in \mathbb{R}[\mathbf{x}] \mid c \in \mathbb{R}^+ \mid m \in \mathbb{N}\}$, and $(\sum(\mathbb{R}(\mathbf{x}))^2 + \mathbb{R}^+) = \{(\sum_{i=1}^m (\frac{p_i}{q_i})^2) + c \mid p_i, q_i \in \mathbb{R}[\mathbf{x}] \text{ s.t. } q_i \neq 0 \mid c \in \mathbb{R}^+ \mid m \in \mathbb{N}\}$.

Lemma 1. $\sum(\mathbb{R}(\mathbf{x}))^2 + \mathbb{R}^+$ does not contain every PD real polynomial.

Proof. Let $\varphi \in QF_{\mathcal{L}}$ be the formula $(x = 0 \wedge x \neq 0)$. Using the transformation in Theorem 2 we have $\exists x(\varphi(x)) \iff \exists x \exists z((xz - 1)^2 + x^2 = 0)$. Let $p(x, z) = (xz - 1)^2 + x^2$. As φ is unsatisfiable over \mathbb{R} , we have $\forall x \forall z(p(x, z) \neq 0)$. But as the constant term of p is 1, we have by Theorem 4 that $\forall x \forall z(p(x, z) > 0)$. We now observe that $p(x, z) \notin (\sum(\mathbb{R}(\mathbf{x}))^2 + \mathbb{R}^+)$. Suppose otherwise. Then, we must have $\forall x \forall z(p(x, z) > k)$ for some $k \in \mathbb{R}^+$. Let $\frac{1}{\epsilon} \in \mathbb{R}$ be s.t. $0 < \frac{1}{\epsilon} < k$. But then $p(\frac{1}{\sqrt{2\epsilon}}, \sqrt{2\epsilon} + 1) = \frac{1}{\epsilon}$. Contradiction.

So we see that even in the bivariate case, there are PD real polynomials that are not simply sums of a sum of squares of rational functions and a positive constant. The situation is altogether more subtle. We now introduce Stengle's Real Nullstellensatz and exploit it to isolate one interesting syntactic criterion for PD real polynomials.

4.1 Stengle's Real Nullstellensatz and a PD Criterion

Stengle's Real Nullstellensatz guarantees the existence of an algebraic proof object certifying the unsatisfiability of a system of polynomial equations over \mathbb{R}^n . The result takes the following form.

Theorem 6 (Stengle's Real Nullstellensatz). Let $S = \{p_1 = 0, \dots, p_k = 0\}$ be a system of real polynomial equations. Then, S is unsatisfiable over \mathbb{R}^n iff

$$\exists \mathbb{P} \in \mathcal{I}(p_1, \dots, p_k) \text{ s.t. } \mathbb{P} = \left(\sum_{i=1}^m (q_i)^2 \right) + 1,$$

where $\mathcal{I}(p_1, \dots, p_k) = \{\sum_{i=1}^k p_i r \mid r \in \mathbb{R}[\mathbf{x}]\}$ is the $\mathbb{R}[\mathbf{x}]$ -ideal of $\{p_1, \dots, p_k\}$ and $q_1, \dots, q_m \in \mathbb{R}[\mathbf{x}]$. \mathbb{P} is called a **Real Nullstellensatz witness**.

Observe that Stengle's Real Nullstellensatz is presenting another way in which the unsatisfiability of an $QF_{\mathcal{L}}$ formula over \mathbb{R}^n can be reduced to the existence of an associated strictly positive polynomial.

We now use this result to isolate a syntactic criterion for PD real polynomials.

Theorem 7. Every PD real polynomial is a ratio of a Real Nullstellensatz witness and a PD real polynomial.

Proof. Let $p \in \mathbb{R}[\mathbf{x}]$ be PD. Then it follows that $\exists \mathbf{x}(p(\mathbf{x}) = 0)$ is unsatisfiable over \mathbb{R}^n , and thus by Stengle's Real Nullstellensatz it follows that $\exists q \in \mathcal{I}(p)$ s.t. $(pq = (\sum_{i=1}^m (q_i)^2) + 1)$ for some $q_1, \dots, q_m \in \mathbb{R}[\mathbf{x}]$. But as $pq = (\sum_{i=1}^m (q_i)^2) + 1$, it follows that pq is itself PD, and so q must be PD as well. Thus, it then follows that $p = \frac{(\sum_{i=1}^m (q_i)^2) + 1}{q}$.

Note that the process involved in the proof of Theorem 7 can be iterated. That is, given that p and q are both PD with $p = \frac{(\sum_{i=1}^m (q_i)^2) + 1}{q}$, it then follows again by Stengle's Real Nullstellensatz that $\exists r \in \mathcal{I}(q)$ s.t. $(qr = (\sum_{i=1}^{m'} (r_i)^2) + 1)$ for some $r_1, \dots, r_{m'} \in \mathbb{R}[\mathbf{x}]$. So as it then follows that r must be PD and therefore $q = \frac{(\sum_{i=1}^{m'} (r_i)^2) + 1}{r}$ and thus

$$p = \frac{((\sum_{i=1}^m (q_i)^2) + 1) r}{(\sum_{i=1}^{m'} (r_i)^2) + 1}$$

where r is PD. But then by the same argument, we have

$$p = \frac{((\sum_{i=1}^m (q_i)^2) + 1) \left((\sum_{i=1}^{m''} (s_i)^2) + 1 \right)}{\left((\sum_{i=1}^{m'} (r_i)^2) + 1 \right) s}$$

where s is PD, and so on.

Based upon this observation, we make the following conjecture.

Conjecture 1. Every PD polynomial is a finite product of ratios of Real Nullstellensatz witnesses.

Finally, we observe that if this conjecture holds, then it follows that every PD polynomial is in fact a ratio of only two polynomials in $(\sum(\mathbb{R}[\mathbf{x}])^2 + \mathbb{R}^+)$. This proof is assisted by the following lemmata.

Lemma 2. *If $p, q \in (\sum(\mathbb{R}[\mathbf{x}])^2 + \mathbb{R}^+)$, then $p + q \in (\sum(\mathbb{R}[\mathbf{x}])^2 + \mathbb{R}^+)$.*

Proof. Immediate.

Lemma 3. *If $p, q \in (\sum(\mathbb{R}[\mathbf{x}])^2 + \mathbb{R}^+)$, then $pq \in (\sum(\mathbb{R}[\mathbf{x}])^2 + \mathbb{R}^+)$.*

Proof. Let $p, q \in (\sum(\mathbb{R}[\mathbf{x}])^2 + \mathbb{R}^+)$ s.t. (wlog) $p = \sum_{i=1}^m (p_i)^2 + k_1$, $q = \sum_{i=1}^n (q_i)^2 + k_2$ for some $p_i, q_i \in \mathbb{R}[\mathbf{x}]$ and $k_1, k_2 \in \mathbb{R}^+$. Then,

$$\begin{aligned} pq &= \left(\sum_{i=1}^m (p_i)^2 + k_1 \right) \left(\sum_{i=1}^n (q_i)^2 + k_2 \right) \\ &= \left(\sum_{i=1}^m (p_i)^2 \right) \left(\sum_{i=1}^n (q_i)^2 \right) + k_1 \left(\sum_{i=1}^n (q_i)^2 \right) + k_2 \left(\sum_{i=1}^m (p_i)^2 \right) + k_1 k_2 \\ &= \left(\sum_{i=1}^m \sum_{j=1}^n (p_i)^2 (q_j)^2 \right) + k_1 \left(\sum_{i=1}^n (q_i)^2 \right) + k_2 \left(\sum_{i=1}^m (p_i)^2 \right) + k_1 k_2 \\ &= \left(\sum_{i=1}^m \sum_{j=1}^n (p_i q_j)^2 \right) + \left(\sum_{i=1}^n (\sqrt{k_1} q_i)^2 \right) + \left(\sum_{i=1}^m (\sqrt{k_2} p_i)^2 \right) + k_1 k_2. \end{aligned}$$

Clearly, $\sum_{i=1}^n (\sqrt{k_1} q_i)^2$, $\sum_{i=1}^m (\sqrt{k_2} p_i)^2$, and $k_1 k_2 \in (\sum(\mathbb{R}[\mathbf{x}])^2 + \mathbb{R}^+)$. By Lemma 2, it suffices to show $(\sum_{i=1}^m \sum_{j=1}^n (p_i q_j)^2) \in (\sum(\mathbb{R}[\mathbf{x}])^2 + \mathbb{R}^+)$. But $\sum_{j=1}^n (p_c q_j)^2 \in (\sum(\mathbb{R}[\mathbf{x}])^2 + \mathbb{R}^+)$ for each fixed $(1 \leq c \leq n)$, and so by Lemma 2 the result follows.

Theorem 8. *If a real polynomial p is a finite product of ratios of Real Nullstellensatz witnesses, then $p = \frac{(\sum_{i=1}^m (q_i)^2) + k_1}{(\sum_{i=1}^{m'} (r_i)^2) + k_2}$ for some $q_1, \dots, q_m, r_1, \dots, r_{m'} \in \mathbb{R}[\mathbf{x}]$, and $k_1, k_2 \in \mathbb{R}^+$.*

Proof. Immediate by Lemmas 2 and 3.

5 Conclusion

In conclusion, we have observed a simple reduction from the \exists theory of RCF to a restricted theory in which every sentence is of the form “polynomial p (which is a sum of squares) is strictly positive over the reals.” Motivated by this observation, we posed the question of isolating syntactic criterion for PD real polynomials analogous to that given by Artin’s positive solution to Hilbert’s 17th Problem for PSD polynomials. We then found one such criterion, namely that every PD real polynomial is a ratio Real Nullstellensatz witness and a PD real polynomial. Finally, we conjectured that every PD real polynomial is a finite product of ratios of Real Nullstellensatz witnesses and proved that if this holds, then every PD real polynomial is in fact a ratio of two polynomials in $(\sum(\mathbb{R}[\mathbf{x}])^2 + \mathbb{R}^+)$.

References

1. A. C. Doherty, Pablo A. Parrilo, and Federico M. Spedalieri. Distinguishing separable and entangled states. *Phys. Rev. Lett.*, 88(18):187904, Apr 2002.

2. T. S. Motzkin. The real solution set of a system of algebraic inequalities. 1970.
3. Pablo A. Parrilo. Semidefinite programming relaxations for semialgebraic problems, 2001.
4. A. Seidenberg. A new decision method for elementary algebra. *The Annals of Mathematics*, 60(2), 1954.
5. Joachim Steinbach. Proving polynomials positive. In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 191–202, London, UK, 1992. Springer-Verlag.
6. Alfred Tarski. A decision method for elementary algebra and geometry. Technical report, Rand Corporation, 1948.
7. A. Tiwari. Abstractions for hybrid systems. *Formal Methods in Systems Design*, 32:57–83, 2008.

APPENDIX

We now present a proof of Conjecture 1 constructed by the aforementioned generous and much appreciated anonymous referee.

Theorem 9. *Every PD polynomial is a finite product of ratios of Real Nullstellensatz witnesses. In fact, every PD polynomial is a ratio of two Real Nullstellensatz witnesses.*

Proof. Assuming $p > 0$, then $-p \geq 0$ is inconsistent, so by the usual PSatz (e.g. Theorem 4.6 in the Parrilo reference [3]), there are two sums of squares of polynomials S_1 and S_2 such that

$$-p * S_1 + S_2 = -1.$$

Now the trick is to use the algebraic identity

$$\begin{aligned} 4p &= (p+1)^2 - (p-1)^2 \\ &= (p+1)^2 + (-1)(p-1)^2 \\ &= (p+1)^2 + (S_2 - S_1p)(p-1)^2. \end{aligned}$$

Let $S'_1 = S_1(p-1)^2$, and $S'_2 = S_2(p-1)^2$, both obviously still sums of squares of real polynomials. Then,

$$\begin{aligned} 4p &= (p+1)^2 + S'_2 - S'_1p \\ (4 + S'_1)p &= (p+1)^2 + S'_2 \\ (4 + S'_1)p &= p^2 + 2p + 1 + S'_2 \\ (2 + S'_1)p &= 1 + p^2 + S'_2 \end{aligned}$$

and therefore as desired

$$p = \frac{1 + p^2 + S'_2}{2 + S'_1}.$$

Observe that $2 + S'_1$ is a Real Nullstellensatz witness as $2 + S'_1 = 1 + (1 + S'_1)$ with $1 + S'_1$ clearly a sum of squares of real polynomials.

That this result unconditionally subsumes Theorem 8 is readily seen.

The Ackermann Approach for Modal Logic, Correspondence Theory and Second-Order Reduction: Extended Abstract

Renate A. Schmidt

School of Computer Science, The University of Manchester, UK

Abstract. This research is concerned with second-order quantifier elimination in modal logic. In particular, I define a refined Ackermann-based approach for eliminating quantified propositional symbols from modal formulae. I prove correctness results and show the approach can solve two new classes of modal problems that have wider scope than existing classes known to be solvable by second-order quantifier elimination methods. Two uses of the approach are discussed: computing first-order frame correspondence properties for modal axioms and rules, and equivalently reducing second-order modal problems.

1 Second-Order Quantifier Elimination

An application of second-order quantifier elimination is correspondence theory in modal logic. Propositional modal logics, when defined axiomatically, have a second-order flavour, but can often be characterized by classes of semantic structures which satisfy first-order conditions. With the help of second-order quantifier elimination methods these first-order conditions, called frame correspondence properties, can often be derived automatically from the axioms. For example, using the standard relational translation method the modal axiom $\mathbf{D} = \forall p[\Box p \rightarrow \Diamond p]$ translates to this second-order formula:

$$(1) \quad \forall P \forall x [\forall y [R(x, y) \rightarrow P(y)] \rightarrow \exists z [R(x, z) \wedge P(z)]].$$

This formula is equivalent to a first-order formula, namely $\forall x \exists y [R(x, y)]$, and is the first-order correspondence property of axiom \mathbf{D} . It can be derived automatically with a second-order quantifier elimination method by eliminating the second-order quantifier $\forall P$ from (1).

Here, I am interested in methods using a substitution-rewrite approach to second-order quantifier elimination. Such methods include the Sahlqvist-van Benthem substitution method for modal logic, the DLS algorithm introduced by Szalas (1993), Doherty, Lukaszewicz and Szalas (1997), and the SQEMA algorithm introduced by Conradie, Goranko and Vakarelov (2006). The focus is in particular on the methods underlying the DLS and SQEMA algorithms. Both these algorithms exploit a general substitution property found in Ackermann (1935). This property, called *Ackermann's Lemma*, tells us when quantified

predicate symbols are eliminable from second-order formulae. For propositional and modal logic Ackermann's Lemma can be formulated as follows. In any model,

$$(2) \quad \exists p[(\alpha \rightarrow p) \wedge \beta(p)] \quad \text{is equivalent to} \quad \beta_\alpha^p,$$

provided these two conditions hold: (i) p is a propositional symbol that does not occur in α , and (ii) p occurs only negatively in β . The formula β_α^p denotes the formula obtained from β by uniformly substituting α for all occurrences of p in β . This property is also true, when the polarity of p is switched, that is, all occurrences of p in β are positive and the implication in the left conjunct is reversed. Applied from left-to-right the equivalence (2) eliminates the second-order quantifier $\exists p$ and all occurrences of p . This idea can be turned into an algorithm for eliminating existentially quantified propositional symbols. I refer to this algorithm as the *basic Ackermann algorithm*. It can be seen to form the skeleton for the DLS and SQEMA algorithms, and forms the basis of the refined approach, introduced in [2], for which this is an extended abstract.

2 A Refined Ackermann Approach

Like the SQEMA algorithm, rather than translating the modal axiom into second-order logic and then passing it to a second-order quantifier elimination method, the approach performs second-order quantifier elimination directly in modal logic. Only in a subsequent step the translation to first-order logic is performed. For example, given the formula $\mathbf{D} = \forall p[\Box p \rightarrow \Diamond p]$ the approach first eliminates $\forall p$ and returns the formula $\Diamond \top$. Subsequently this is translated to first-order logic to give the expected seriality property $\forall x \exists y[R(x, y)]$.

The approach is defined for propositional multi-modal tense logics, more precisely, the logic $\mathbf{K}_{(m)}^n(\sim, \pi+)$ with forward and backward looking modalities, nominals and second-order quantification over propositional symbols. This means that the approach can also be used for problems in basic modal logic.

A main motivation for this work has been to gain a better understanding of when quantifier elimination methods succeed, and to pinpoint precisely which techniques are crucial for successful termination. I define two new classes of formulae for which the approach succeeds: the class \mathcal{C} and an algorithmic version called $\mathcal{C}^>$. For lack of space the classes are not defined here but can be seen to define normal forms for when Ackermann-based second-order quantifier elimination methods succeed. \mathcal{C} and $\mathcal{C}^>$ subsume both the Sahlqvist class of formulae and the class of monadic-inductive formulae of Goranko and Vakarelov (2006). I present minimal requirements for successful termination for all these classes.

I consider two applications of the approach:

- (i) Computing correspondence properties for modal axioms and modal rules. For example, equivalently reducing axiom \mathbf{D} to the seriality property, or equivalently reducing the modal rule $\Box p / \Diamond p$ to $\forall x \exists y \exists z[R(x, y) \wedge R(z, y)]$.
- (ii) Equivalently reducing second-order modal problems. For example, the second-order modal formula $\forall p \forall q[\Box(p \vee q) \rightarrow (\Box p \vee \Box q)]$ equivalently reduces to $\forall p[\Diamond p \rightarrow \Box p]$, or the axiom \mathbf{D} equivalently reduces to $\Diamond \top$.

While the approach follows the idea of the basic Ackermann algorithm and is closely related to the DLS algorithm and the SQEMA algorithm, I introduce a variety of enhancements and principles novel to existing substitution-rewrite approaches.

First, which propositional symbols are to be eliminated can be flexibly specified, and the approach is not limited to eliminating all propositional symbols. Second, in order to be able to ensure effectiveness and avoid unintended looping, the approach is enhanced with ordering refinements. In the approach an ordering on the non-base symbols (these are the symbols that we want to eliminate) must be specified and determines the order in which these symbols are eliminated. At the same time the ordering is used to delimit the way that inference rules are applied. Third, for reasons of efficiency and improved success rate, it is beneficial to incorporate techniques for pruning the search space. A general notion of redundancy is thus included. It is designed so that it is possible to define practical simplification and optimization techniques in a flexible way. Fourth, the approach is defined in terms of calculi given by sets of inference rules.

It turns out that the approach allows for a more fine-grained analysis of the computational behaviour of Ackermann-based approaches and more general results can be formulated. The following properties are shown in [2].

1. Any derivation in the approach is guaranteed to terminate and the obtained formula is logically equivalent to the input formula.
2. Any problem in the class $\mathcal{C}^>$ is effectively and successfully reducible using some ordering. For the subclass \mathcal{C} of $\mathcal{C}^>$ redundancy elimination is not needed and the ordering is immaterial.
3. Whenever the approach successfully eliminates all propositional symbols for a modal formula α then (a) $\neg\alpha$ is d-persistent and hence canonical, and (b) the formula returned is equivalent to α .
4. All modal axioms equivalent to the conjunction of formulae reducible to clauses in \mathcal{C} and $\mathcal{C}^>$ are elementary and canonical.

Property 1 means that the refined modal Ackermann approach is correct and terminating. Property 2 says that it decides the second-order quantifier elimination problem of the classes \mathcal{C} and $\mathcal{C}^>$. Properties 2–4 strengthen Sahlqvist’s theorem and the corresponding result for monadic-inductive formulae. The significance of Property 4 is that axioms that are equivalent to first-order properties and are canonical can be used to provide sound and complete axiomatizations of modal logics.

For a full account of the approach I refer to [2] and Chapter 13 in [1].

References

1. D. M. Gabbay, R. A. Schmidt, and A. Szalas. *Second-Order Quantifier Elimination: Foundations, Computational Aspects and Applications*. College Publications, 2008.
2. R. A. Schmidt. Improved second-order quantifier elimination in modal logic. In *Proc. JELIA’08*, volume 5293 of *LNAI*, pages 375–388. Springer, 2008. The long version is under review for publication in a journal.

CTL-RP: A Computational Tree Logic Resolution Prover

Lan Zhang, Ullrich Hustadt and Clare Dixon*

Department of Computer Science, University of Liverpool
Liverpool, L69 3BX, UK

{Lan.Zhang, U.Hustadt, CLDixon}@liverpool.ac.uk

1 Introduction

Temporal logics are considered important tools in many different areas of artificial intelligence and computer science, including the specification and verification of concurrent and distributed systems. Here we present CTL-RP, the first resolution theorem prover for CTL [3], a branching-time temporal logic, which implements the sound and complete clausal resolution calculus $R_{CTL}^{\lambda, S}$ [7] based on an earlier calculus by [2]. The calculus $R_{CTL}^{\lambda, S}$ is designed in order to allow the use of classical first-order resolution techniques to emulate the rules of the calculus. We take advantage of this approach in the development of our prover CTL-RP which uses the first-order theorem prover SPASS [5].

This presentation is based on a paper accepted for publication in a Special Issue of AI Communications on Practical Aspects of Automated Reasoning [6].

2 Normal form for CTL SNF_{CTL}^g and clausal resolution calculus $R_{CTL}^{\lambda, S}$

CTL is an extension of propositional logic with temporal operators \square (always in the future), \circ (at the next moment in time), \diamond (eventually in the future), \mathcal{U} (until), and \mathcal{W} (unless) and path quantifiers \mathbf{A} (for all future paths) and \mathbf{E} (for some future path), interpreted over infinite tree structures.

The calculus $R_{CTL}^{\lambda, S}$ operates on formulae in a clausal normal form called Separated Normal Form with Global Clauses for CTL, denoted by SNF_{CTL}^g . The language of SNF_{CTL}^g clauses is defined over an extension of CTL in which we label certain formulae with an index *ind* taken from a countably infinite index set Ind and it consists of formulae of the following form.

$$\begin{array}{ll}
 \mathbf{A}\square(\text{start} \Rightarrow \bigvee_{j=1}^k m_j) & \text{(initial clause)} \\
 \mathbf{A}\square(\text{true} \Rightarrow \bigvee_{j=1}^k m_j) & \text{(global clause)} \\
 \mathbf{A}\square(\bigwedge_{i=1}^n l_i \Rightarrow \mathbf{A}\circ \bigvee_{j=1}^k m_j) & \text{(\mathbf{A}-step clause)} \\
 \mathbf{A}\square(\bigwedge_{i=1}^n l_i \Rightarrow \mathbf{E}_{\langle ind \rangle} \circ \bigvee_{j=1}^k m_j) & \text{(\mathbf{E}-step clause)} \\
 \mathbf{A}\square(\bigwedge_{i=1}^n l_i \Rightarrow \mathbf{A}\diamond l) & \text{(\mathbf{A}-sometime clause)} \\
 \mathbf{A}\square(\bigwedge_{i=1}^n l_i \Rightarrow \mathbf{E}_{\langle ind \rangle} \diamond l) & \text{(\mathbf{E}-sometime clause)}
 \end{array}$$

* This work was supported by EPSRC grant EP/D060451/1.

where **start** is a propositional constant (which is only true at the root of the infinite tree structure), l_i ($1 \leq i \leq n$), m_j ($1 \leq j \leq k$) and l are literals, that is atomic propositions or their negation, ind is an element of \mathbf{Ind} . The symbol ind is akin to a Herbrand constant representing a path in the infinite tree structure. As all clauses are of the form $\mathbf{A}\Box(P \Rightarrow D)$ we often simply write $P \Rightarrow D$ instead.

A set of transformation rules which allows us to transform an arbitrary CTL formula into an equi-satisfiable set of $\text{SNF}_{\text{CTL}}^g$ clauses is given in [7].

$\mathbf{R}_{\text{CTL}}^{\succ, S}$ consists of two types of resolution rules, *step* resolution rules (SRES1 to SRES8) and *eventuality* resolution rules (ERES1 and ERES2). Motivated by refinements of propositional and first-order resolution, we restrict the applicability of step resolution rules by means of an atom ordering \succ and a selection function S , which helps to prune the search space dramatically. Due to lack of space, we only present one of the step resolution rules, omitting the ordering and selection restriction, and one of the eventuality resolution rules. In the following l is a literal, P , P_j^i and Q are conjunctions of literals, and C , C_j^i and D are disjunctions of literals.

$$\text{SRES2} \frac{P \Rightarrow \mathbf{E}_{\langle ind \rangle} \Box (C \vee l) \quad Q \Rightarrow \mathbf{A} \Box (D \vee \neg l)}{P \wedge Q \Rightarrow \mathbf{E}_{\langle ind \rangle} \Box (C \vee D)}$$

$$\text{ERES1} \frac{P^\dagger \Rightarrow \mathbf{E} \Box \mathbf{E} \Box l \quad Q \Rightarrow \mathbf{A} \Diamond \neg l}{Q \Rightarrow \mathbf{A} (\neg P^\dagger \mathcal{W} \neg l)}$$

where $P^\dagger \Rightarrow \mathbf{E} \Box \mathbf{E} \Box l$ represents a set, $\Lambda_{\mathbf{E} \Box}$, of $\text{SNF}_{\text{CTL}}^g$ clauses $P_1^1 \Rightarrow *C_1^1 \dots, P_1^n \Rightarrow *C_1^n, \dots, P_{m_1}^1 \Rightarrow *C_{m_1}^1, \dots, P_{m_n}^n \Rightarrow *C_{m_n}^n$ with each $*$ either being empty or being an operator in $\{\mathbf{A} \Box\} \cup \{\mathbf{E} \Box ind \mid ind \in \mathbf{Ind}\}$ and for every i , $1 \leq i \leq n$, $(\bigwedge_{j=1}^{m_i} C_j^i) \Rightarrow l$ and $(\bigwedge_{j=1}^{m_i} C_j^i) \Rightarrow (\bigvee_{i=1}^n \bigwedge_{j=1}^{m_i} P_j^i)$ are provable.

The inference rules of $\mathbf{R}_{\text{CTL}}^{\succ, S}$ can be used to decide the satisfiability of a given set N of $\text{SNF}_{\text{CTL}}^g$ clauses by computing the saturation N' of N using at most an exponential number of inference steps with each step taking at most exponential time; N is unsatisfiable iff N' contains a clause **true** \Rightarrow **false** or **start** \Rightarrow **false**. This gives a complexity optimal EXPTIME decision procedure for CTL.

3 CTL-RP

In order to obtain an efficient CTL theorem prover and to reuse existing state-of-the-art first-order resolution theorem provers, we adopt an approach analogous to that used in [4] to implement a resolution calculus for PLTL to implement the calculus $\mathbf{R}_{\text{CTL}}^{\succ, S}$ and the associated decision procedure for CTL. In our implementation of $\mathbf{R}_{\text{CTL}}^{\succ, S}$, we first transform all $\text{SNF}_{\text{CTL}}^g$ clauses except **A**- and **E**-sometime clauses into first-order clauses. Then we are able to use first-order ordered resolution with selection to emulate step resolution. For this part of the implementation we are using the theorem prover SPASS. **A**- and **E**-sometime clauses cannot be translated to first-order logic. Therefore, we continue to use the eventuality resolution rules ERES1 and ERES2 for inferences with **A**- and **E**-sometime clauses, respectively, and use the loop search algorithm presented

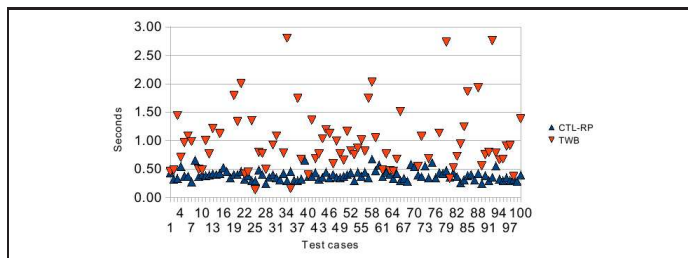


Fig. 1. Performance of CTL-RP and TWB on a set of benchmark formulae

in [7] to find suitable premises for these rules and we compute the results of applications of the eventuality resolution rules in the form of first-order clauses.

Besides CTL-RP, there is only one other CTL theorem prover we know of, namely a CTL module for the Tableau Workbench (TWB) [1]. We have created several sets of benchmark formulae that we have used to compare CTL-RP version 00.09 with TWB version 3.4. The comparison was performed on a Linux PC with an Intel Core 2 CPU@2.13 GHz and 3GB main memory, using the Fedora 9 operating system. In Figure 1, we show the experimental results on one of those sets of benchmark formulae. This set of benchmark formulae consists of one hundred unsatisfiable CTL formulae such that each formula specifies a randomly generated state transition system with at most 16 states, using 30 so-called transition specification formulae over four propositional variables, together with five so-called property specifications for that system. The graph in Figure 1 indicates the CPU time in seconds required by TWB and CTL-RP to establish the unsatisfiability of each benchmark formula. CTL-RP shows a much more stable performance on these benchmarks than TWB.

References

1. P. Abate and R. Goré. The Tableaux Workbench. In *Proc. TABLEAUX'03*, volume 2796 of *LNCS*, pages 230–236. Springer, 2003.
2. A. Bolotov. *Clausal Resolution for Branching-Time Temporal Logic*. PhD thesis, Manchester Metropolitan University, 2000.
3. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, volume 131 of *LNCS*, pages 52–71. Springer, 1982.
4. U. Hustadt and B. Konev. TRP++: A Temporal Resolution Prover. In *Collegium Logicum*, pages 65–79. Kurt Gödel Society, 2004.
5. C. Weidenbach, R. A. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic. System description: Spass version 3.0. In *Proc. CADE-21*, volume 4603 of *LNCS*, pages 514–520, 2007.
6. L. Zhang, U. Hustadt, and C. Dixon. CTL-RP: A computational tree logic resolution prover. To appear in *AI Communications*. <http://www.csc.liv.ac.uk/~ullrich/publications/paar-aicom.pdf>.
7. L. Zhang, U. Hustadt, and C. Dixon. First-order Resolution for CTL. Technical Report ULCS-08-010, Dep. of Computer Science, University of Liverpool, 2008.

Author Index

Babenyshev, Sergey	47
Caldwell, James	17
de Moura, Leonardo	62
Dixon, Clare	75
Franssen, Michael	2
Hustadt, Ullrich	75
Kothari, Sunil	17
Kutsia, Temur	32
Marin, Mircea	32
Meadows, Catherine	1
Passmore, Grant Olney	62
Rybakov, Vladimir	47
Schmidt, Renate A.	47, 72
Tishkovsky, Dmitry	47
Zhang, Lan	75