32nd International Workshop on Unification (UNIF 2018)

— Informal Proceedings —

Oxford, 7th July, 2018

Preface

This volume contains the papers presented at UNIF 2018: The 32nd International Workshop on Unification held on July 7, 2018 in Oxford.

UNIF 2018 was affiliated with the Third International Conference on **Formal Structures for Computation and Deduction FSCD 2018**, part of the Federated Logic Conference FLoC 2018.

There were 10 submissions. Each submission was reviewed by at least 3, and on the average 3.2, Program Committee members and external reviewers. The committee decided to accept 8 papers for presentation and 2 for short presentation. The program also includes 2 invited talks:

Silvio Ghilardi on **"Handling Substitutions via Duality"**, and Adrià Gascón on **"Compressed Term Unification: Results, Uses, Open Problems, and Hopes"**.

We would like to thank all members of the Program Committee and the subreviewers for their high quality reviews and discussion held on the EasyChair platform that assured the scientific standard of UNIF in its 32nd edition.

Finally, we would like to thank the UNIF Steering Committee and the FSCD and FLoC scientific and organising team for the support before and during the workshop.

May 27, 2018
Brasília
Toulouse

Mauricio Ayala-Rincón
Philippe Balbiani

Program Committee

Maria Alpuente	Universitat Politècnica de València
Mauricio Ayala-Rincón	Universidade de Brasília
Franz Baader	TU Dresden
Philippe Balbiani	Institut de Recherche en Informatique de Toulouse
Eduardo Bonelli	UN de Quilmes and Stevens Institute of Technology
Iliano Cervesato	Carnegie Mellon University
Wojciech Dzik	University of Silesia
Santiago Escobar	Universitat Politècnica de València
Maribel Fernández	King's College London
Çiğdem Gencer	Istanbul Aydın University
Silvio Ghilardi	Dipartimento di Matematica, Università degli Studi di Milano
Rosalie Iemhoff	Utrecht University
Emil Jeřábek	Czech Academy of Sciences
Temur Kutsia	Johannes Kepler University Linz
Jordi Levy	IIIA-CSIC
Christopher Lynch	Clarkson University
Catherine Meadows	University of Hawaii at Manoa
Paliath Narendran	University at Albany
Christophe Ringeissen	INRIA
Manfred Schmidt-Schauss	Inst. für Informatik, Goethe-University Frankfurt am Main

Additional Reviewers

Cornell, Kimberly
Dougherty, Daniel
Hono, Daniel
Kutz, Yunus
Morawska, Barbara

Table of Contents

32nd International Workshop on Unification (UNIF 2018), Informal Proceedings
Oxford, 7th July, 2018

Handling Substitutions via Duality

Silvio Ghilardi

Università degli Studi di Milano Milano,
silvio.ghilardi@unimi.it

Abstract

Many interesting problems concerning intuitionistic and intermediate propositional logics (as well as other nonclassical and modal propositional logics) are related to properties of substitutions: among them, besides unification, we have rule admissibility, characterization of projective formulae, definability of maximum and minimum fixpoints, finite periodicity theorems, etc. Since most of these questions can be stated in category-theoretic terms, they are sensible to an approach via duality techniques. The available duality for finitely presented Heyting algebras involves both sheaves (giving the appropriate geometric framework) and bounded bisimulations (handling the combinatorics of definability aspects): we show how to use such duality to attack and solve the above problems in a uniform way. [Recent and new results come from joint work [18] with Luigi Santocanale]

Duality techniques have a long tradition in algebraic logic: given the well-known correspondence between logical calculi and suitable algebraic varieties, it is possible to transfer logical problems into the algebraic context and, inside such a context, such problems can be further reformulated in geometric/topological terms by relying on duality theorems. Among such dualities, we recall Stone spaces duality for Boolean algebras, Priestley spaces duality for distributive lattices, Esakia spaces duality for Heyting algebras, modal spaces duality for modal algebras, etc. (see textbooks like [8, 21, 6, 28] for relevant information).

Duality approaches are particularly suitable for questions involving substitutions, because substitutions can bee seen as algebraic homomorphisms. Unification theory is a typical example in this sense: unification type is a categorical invariant [1, 12] and as such it can be transposed to dual contexts, as fruitfully exemplified for instance in [12, 15, 7] for some first cases of locally finite varieties. Unification theory may become a powerful tool in order to analyze rule admissibility [2, 13, 14, 9]. There are many other questions involving substitutions that are sensible to duality techniques: we quote for instance the problem of the finite convergence to a fixpoint for monotone formulae, see [23] for a survey. Other classical topics, like interpolation and uniform interpolation [24], apparently do not seem to directly refer to substitutions, but they still involve a categorical structure whose meaning clearly appears after dualization [19, 27, 26, 20].

It should be noticed however that the above problems require specific dualities for the restricted subcategory of finitely presented (or of finitely generated free) algebras, rather than a duality for the category of all algebras: in logical terms, whereas arbitrary algebras correspond

- via the Lindenbaum-Tarski construction - to *theories* inside a given logic, finitely presented algebras correspond to *finitely axiomatized theories* in such a logic. Duality theorems for such restricted subcategories can be obtained basically via two techniques, namely finite step-frames constructions [10, 11, 3, 5, 4] and bounded bisimulations [19, 29]. We shall refer to the duality theorem from [19], whose main feature is that of embedding dual categories of finitely presented algebras inside a *sheaf topos*: in this setting, the geometric environment shows how to find relevant mathematical structures (products, equalizers, images,...) using their standard definitions in sheaves and presheaves; on the other hand, the combinatorial aspects show that such constructions are definable, thus meaningful from the logical side. In this sense our ingredients of combinatorial nature (Ehrenfeucht-Fraïssé games and bounded bisimulations) replace the topological ingredients which are common in the algebraic logic literature (working with arbitrary algebras instead of finitely presented ones).

The plan of the talk is the following:

- we show how to exploit finite dualities in order to determine the unification type in locally finite varieties [15];
- we recall some dualities working in non locally finite cases [19];
- we revisit unification results from [13, 14, 17] in these duality contexts;
- as a new recent application, we give a semantic proof [18] of Ruitenburg's Theorem [25].

We recall here the statement of the latter. Let us call an infinite sequence

$$a_1, a_2, \dots, a_i, \dots$$

ultimately periodic iff there are N and k such that for all $s_1, s_2 \geq N$, we have that $s_1 \equiv s_2 \mod k$ implies $a_{s_1} = a_{s_2}$. If (N, k) is the smallest (in the lexicographic sense) pair for which this happens, we say that N is an *index* and k a *period* for the ultimately periodic sequence $\{ a_i \}_i$. Thus, for instance, an ultimately periodic sequence with index N and period 2 looks as follows

$$a_1, \dots, a_N, a_{N+1}, a_N, a_{N+1}, \dots$$

A typical example of an ultimately periodic sequence is the sequence of the iterations $\{ f^i \}_i$ of an endo-function f of a finite set. Whenever infinitary data are involved, ultimate periodicity comes often as a surprise.

Ruitenburg's Theorem is in fact a surprising result stating the following: take a formula $A(x, \underline{y})$ of intuitionistic propositional calculus (IPC) (by the notation $A(x, \underline{y})$ we mean that the only propositional letters occurring in A are among $x, \underline{y}$ - with $\underline{y}$ being, say, the tuple $y_1, \dots, y_n$) and consider the sequence $\{ A^i(x, \underline{y}) \}_{i \geq 1}$ so defined:

$$A^1 :\equiv A, \quad \dots, \quad A^{i+1} :\equiv A(A^i/x, \underline{y}) \tag{1}$$

where the slash means substitution; then, *taking equivalence classes under provable bi-implication in* (IPC), *the sequence* $\{ [A^i(x, \underline{y})] \}_{i \geq 1}$ *is ultimately periodic with period 2.* The latter means that there is N such that

$$\vdash_{IPC} A^{N+2} \leftrightarrow A^N \quad . \tag{2}$$

An interesting consequence of this result is that *least (and greatest) fixpoints of monotonic formulae are definable in* (IPC) [22, 23, 16]: this is because the sequence (1) becomes increasing when evaluated on $\perp/x$ (if A is monotonic in x), so that the period is decreased to 1. Thus the index of the sequence becomes a finite upper bound for the fixpoints approximation convergence.

Ruitenburg's Theorem was shown in [25] via a, rather involved, purely syntactic proof. The proof has been recently formalized inside the proof assistant COQ by T. Litak (see `https://git8.cs.fau.de/redmine/projects/ruitenburg1984`). Here we supply a purely semantic proof, using duality and bounded bisimulation machinery (details are available in the preprint [18]).

References

[1] M. H. Albert. Category equivalence preserves unification type. *Algebra Universalis*, 36:457–466, 1996.

[2] F. Baader and S. Ghilardi. Unification in modal and description logics. *Logic Journal of the IGPL*, 19(6):705–730, 2011.

[3] N. Bezhanishvili and M. Gehrke. Finitely generated free Heyting algebras via Birkhoff duality and coalgebra. *Logical Methods in Computer Science*, 7(2), 2011.

[4] N. Bezhanishvili and S. Ghilardi. The bounded proof property via step algebras and step frames. *Ann. Pure Appl. Logic*, 165(12):1832–1863, 2014.

[5] N. Bezhanishvili, S. Ghilardi, and M. Jibladze. Free modal algebras revisited: the step-by-step method. In *Leo Esakia on Duality in Modal and Intuitionistic Logics*, Trends in Logic. Springer, 2014.

[6] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press, 2001.

[7] Leonardo Cabrer. Unification on subvarieties of pseudocomplemented distributive lattices. *Notre Dame J. Form. Log.*, 57(4):477–502, 2016.

[8] A. Chagrov and M. Zakharyaschev. *Modal Logic*. The Clarendon Press, 1997.

[9] W. Dzik and P. Wojtylak. Modal consequence relations extending S4.3: an application of projective unification. *Notre Dame J. Form. Log.*, 57(4):523–549, 2016.

[10] S. Ghilardi. Free Heyting algebras as bi-Heyting algebras. *Math. Rep. Acad. Sci. Canada XVI.*, 6:240–244, 1992.

[11] S. Ghilardi. An algebraic theory of normal forms. *Annals of Pure and Applied Logic*, 71:189–245, 1995.

[12] S. Ghilardi. Unification through projectivity. *J. Log. Comput.*, 7(6):733–752, 1997.

[13] S. Ghilardi. Unification in intuitionistic logic. *J. Symb. Log.*, 64(2):859–880, 1999.

[14] S. Ghilardi. Best solving modal equations. *Ann. Pure Appl. Logic*, 102(3):183–198, 2000.

[15] S. Ghilardi. Unification, finite duality and projectivity in varieties of Heyting algebras. *Ann. Pure Appl. Logic*, 127(1-3):99–115, 2004.

[16] S. Ghilardi, M. João Gouveia, and L. Santocanale. Fixed-point elimination in the intuitionistic propositional calculus. In *Foundations of Software Science and Computation Structures, FOSSACS 2016, Proceedings*, pages 126–141, 2016.

[17] S. Ghilardi and L. Sacchetti. Filtering unification and most general unifiers in modal logic. *J. Symb. Log.*, 69(3):879–906, 2004.

[18] S. Ghilardi and L. Santocanale. Ruitenburg's theorem via duality and bounded bisimulations. *CoRR*, abs/1804.06130, 2018.

[19] S. Ghilardi and M. Zawadowski. *Sheaves, Games and Model Completions*. Kluwer, 2002.

[20] T. Kowalski and G. Metcalfe. Uniform interpolation and coherence. *CoRR*, abs/1803.09116, 2018.

[21] M. Kracht. *Tools and techniques in modal logic*, volume 142 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., 1999.

[22] S. I. Mardaev. Least fixed points in Grzegorczyk's logic and in the intuitionistic propositional logic. *Algebra and Logic*, 32(5):279–288, 1993.

[23] S. I. Mardaev. Definable fixed points in modal and temporal logics : A survey. *Journal of Applied Non-Classical Logics*, 17(3):317–346, 2007.

[24] A. M. Pitts. On an interpretation of second order quantification in first order intuitionistic propositional logic. *J. Symb. Log.*, 57(1):33–52, 1992.

[25] W. Ruitenburg. On the period of sequences $(a^n(p))$ in intuitionistic propositional calculus. *The Journal of Symbolic Logic*, 49(3):892–899, September 1984.

[26] S. J. van Gool, G. Metcalfe, and C. Tsinakis. Uniform interpolation and compact congruences. *Ann. Pure Appl. Logic*, 168(10):1927–1948, 2017.

[27] S. J. van Gool and L. Reggio. An open mapping theorem for finitely copresented Esakia spaces. *Topology and its Applications*, 240:69–77, 2018.

[28] Y. Venema. Algebras and coalgebras. In P. Blackburn, J. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*, pages 331–426. Elsevier, 2007.

[29] A. Visser. Uniform interpolation and layered bisimulation. In *Gödel '96 (Brno, 1996)*, volume 6 of *Lecture Notes Logic*, pages 139–164. Springer, Berlin, 1996.

32nd International Workshop on Unification (UNIF 2018), Informal Proceedings
Oxford, 7th July, 2018

Compressed Term Unification: Results, applications, open problems, and hopes.

Adrià Gascón[1]

Warwick University and The Alan Turing Institute, UK
agascon@turing.ac.uk

Abstract

Already in the classic first-order unification problem, the choice of a suitable formalism for term representation has a significant impact in computational efficiency. One can find other instances of this situation in some variants of second-order unification, where representing partial solutions efficiently leads to better algorithms. In this talk I will present some compression schemes for terms and discuss computational complexity results for variants of first and second-order unification on compressed terms. I'll show how these results build up on each other and discuss open problems in compressed term unification, as well as potential approaches to solutions.

1 Introduction

The problem of checking satisfiability of a set of equations plays a central role in any mathematical science. From the perspective of computer science, a lot of effort is devoted to finding efficient decision procedures for different families of equations. The problem of *satisfiability of word equations*, also known as *Word Unification (WU)*, figures prominently as one of the most intriguing problems of that form. The first algorithm for that problem was given by Makanin [26], and the best known upper bound (PSPACE) is due to Plandowski [31]. On the other hand, its PSPACE-hardness is an open question. Several particular cases of that problem, such as the ones that result from fixing the number of variables in the equations to a constant, have also been studied. For instance, efficient algorithms for satisfiability of word equations with one [6, 29, 18] and two [5] variables have been discovered.

Another fundamental operation in symbolic computation systems is the well-known *first-order unification problem*. This problem consists of solving equations of the form $s \doteq t$, where s and t are terms with first-order variables. The goal is to find a mapping from variables to (first-order) terms that would make the terms s and t syntactically equal. This problem was firstly introduced as such in the work by J.A. Robinson, which established the foundations of automated theorem proving and logic programming. More concretely, Robinson presented in [33] a procedure to determine the validity of a first-order sentence that has term unification as its main ingredient. Later, term unification was also used by Knuth and Bendix as a key component of their critical pairs method to determine local confluence of term rewrite systems (see [1] for a general survey on unification theory). The syntactic unification and matching problems were deeply investigated in the last century. Among other results, linear time algorithms were discovered [27, 30]. Moreover, more expressive variants of term unification such as *unification modulo theories* have also drawn a lot of attention. In this notion of term unification, equality between terms is interpreted under equational theories such as associativity, commutativity, and distributivity, among others [1].

An interesting connection between word and term unification is the *Context Unification (CU)* problem. In CU, the terms s, t in the equation $s \doteq t$ may contain context variables. For example, consider the equation $F(f(x, b)) \doteq f(a, F(y))$. where x, y are first-order variables ranging over terms and F is a context variable that can be replaced by any context. One of the possible solutions of this instance is the substitution $\{F \mapsto f(a, \bullet), x \mapsto a, y \mapsto b\}$. Note that when we instantiate F by $f(a, \bullet)$ in the equation, replacing the occurrence of $\bullet$ by the argument of F in each of its occurrences, we get $f(a, f(x, b)) \doteq f(a, f(a, y))$, and thus both sides of the equations become equal after applying $\{x \mapsto a, y \mapsto b\}$. Note that, simply using a unary signature, WU reduces to CU. On the other hand, CU is a particular case of second-order unification, which is undecidable [15]. The decidability of CU remained open for a long time, until recently a PSPACE algorithm was presented by Jeż [19].

The example above is in fact an instance of the so-called *one context unification* problem, denoted 1-CU. In 1-CU, only one context variable, possibly with many occurrences, may appear in the input terms. One of the motivations for the study of this problem is its close relationship to interprocedural program analysis [17], whose goal is to compute all simple invariants of imperative procedural programs. Although it is known that 1-CU can be solved in non deterministic polynomial time [10], whether the problem is NP-hard or a polynomial time algorithm exists is an open problem.

Another particular case of context unification is *context matching*. The input of context matching is an equation $s \doteq t$ such that s may contain context variables and first-order variables, and t does not contain variables of any kind. Although it is known to be NP-complete, there are several subcases of context matching that can be solved efficiently [9, 36].

Interesting applications of one context unification and matching arise in the search/extraction of information from tree data structures. For example, a simple matching equation of the form $F(s) \doteq t$, where F is the context variable, t is ground, and s may contain first-order variables but it does not contain occurrences of F, corresponds to searching instances of s within t. Context matching also captures, for example, a conjunctive search of the form $F_1(s_1) \doteq t \wedge \cdots \wedge F_n(s_n) \doteq t$, where the F_is are pairwise different and do not occur elsewhere. These equations correspond to searching for a subterm u_i of t that can be matched by s_i, for every $i \in \{1, \ldots, n\}$; with the additional constraint that variables within the s_is must have a common instance in t, see [16] for the analysis of conjunctive query mechanisms over trees. More generally, multiple occurrences of the same context variable in the term s of a context equation $s \doteq t$, correspond to searching for instances of subterms of t that differ at, at most, one position. This has applications in computational linguistics [28]. It is also easy to encode questions that ask for subtrees that are equal up to several positions.

Term representation

To give a complete description of the problems stated above, one has to precisely state how is the input represented. Besides an *explicit* tree representation for terms, in this talk we will consider two more succinct representations: *Directed Acyclic Graphs* (DAGs) and *Singleton Tree Grammars* (STGs), also known as *tree straight-line program*. Similarly as DAGs allow compression by exploiting the reuse of repeated instances of a subterm in a term, STGs are

PROBLEM	TERM REPRESENTATION FORMALISM		
	Explicit	*DAG*	*STG*
Context Unification	PSPACE [19]	?	?
Context Matching	NP-Complete	NP-Complete	NP-Complete
One-context Unification	NP [10]	NP [4]	NP [4]
Left-linear One-context Unification (and similar variants)	Ptime	Ptime [14]	?
2-*restricted context Unification*	Ptime	Ptime [13]	?
k-context Matching	Ptime	Ptime	Ptime [7]
First-order Unification	Ptime	Ptime	Ptime [9, 8]
First-order Matching	Ptime	Ptime	Ptime [7]
Equality of unordered ranked trees	Ptime	Ptime	Ptime [24]
Equality of unordered unranked trees	Ptime	Ptime	Ptime [11]

Figure 1: Some recent and classical results in first-order and context unification, for different variants and formalisms for term representation.

a grammar-based compression mechanism based on the reuse of repeated (multi)contexts. An interesting property of the STG formalism is that many operations on terms can be efficiently performed directly in their compressed representation (see [21] for a survey). Some examples are linear subpattern matching [34], i.e. finding instances of a linear terms s within a ground term t, first-order and unification and matching and 1-CU [9, 12, 4], equivalence checking [3], congruence closure [35], and membership in several classes of languages represented by tree automata [22, 25]. Moreover, STG compressors have already been developed and proven useful in practice for XML representation and processing [3, 23], and termination analysis of Term Rewrite Systems [2]. More generally, STGs are a very useful concept for the analysis of unification problems since, roughly speaking, allow to represent solutions in a succinct but still efficiently verifiable form. This observation is related to Jeż's *recompression technique*, which led to many results including the positive answer to decidability of Context Unification mentioned above. In fact, the relationship between compression and solvability of work equations goes back to the work of Plandowski and Rytter [32] and Plandowski's work on word unification [31].

2 Selected Results & Open Problems

Figure 1 collects some classical results in unification. We consider three variants for each problem, depending of whether the input equations are represented explicitly as trees, DAGs, or STGs. The main major open problem in this space has to do with the exact complexity of the context unification problem. While the problem is known to be in PSPACE, as mentioned above, it is only known to be NP-hard. As in fact this is not even known for the simpler case of word unification, that problem is the natural starting point.

Open Problem 1. *Is there a non-deterministic polynomial time decision procedure for WU?*

For the particular case of 1-CU, the main open question is whether there is a polynomial time decision procedure for this problem, as it is only known to be in NP. The problem, for the case where the input is given a DAG, was conjectured to be in P in [14], where many particular cases were shown to be solvable in polynomial time. In fact, given the results from [14], it suffices to find a polynomial time algorithm to the following reduced form.

Definition 2.1. *An 1-CU instance $\mathcal{I}$ is called* reduced *if it is of the form*

$$\begin{aligned}\{F(u_i) \doteq x_i \mid i = 1, 2, \dots\} \cup \{F(v_j) \doteq s \mid j = 1, 2, \dots\}\\ \cup\ \{F(w_k) \doteq t \mid k = 1, 2, \dots\}\end{aligned} \tag{1}$$

where s, t do not contain F; that is, the right hand-side of the equations have at most two non-variable terms.

Open Problem 2. *Is there a polynomial time decision procedure for reduced 1-CU?*

A particular case of 1-CU that is instrumental in solving the particular cases of [14] is the so-called 2-restricted 1-CU. In this particular case, the context variable occurs at most twice in the input. While we have a polynomial time algorithm for this problem [13] even in the case where the input is given as a DAG, it is not known whether the results in [14] and [13] extend to the case where the input is a STG. In fact, before tackling that more general problem, one should consider a simpler case that was left open in [34]: whether STG-compressed submatching (of nonlinear terms s in t) can be solved in polynomial time or not. This in fact corresponds to solving the 1-CU equation $F(s) = t$, for terms s and t containing arbitrary first-order variables, but not the context variable F.

Open Problem 3. *Is there a polynomial time algorithm STG-compressed term submatching?*

Note that this problem is related with finding a redex of a term rewriting rule in a term. Another open problem with a similar flavour is the one of deciding whether, for STG-compressed terms s and t, and an order $<$ on terms, $s < t$ holds. This problem can be parameterized by the class of the ordering $<$, e.g. lexicographic path orderings or knuth-Bendix orderings. This brings us to our next open problem.

Open Problem 4. *Let s and t be STG-compressed terms, and let $<$ be a KBO ordering on terms, and thus parameterized by a weight function and a precedence order on the signature of s and t. Is there a polynomial time decision procedure for $s < t$?*

All the results regarding STGs from Figure 1, as well as the general unification results that rely on STGs for a succinct representation of sets of solutions of potentially exponential size, critically rely on a polynomial time algorithm for equivalence checking of STG-compressed terms. This classical result by Plandowski, has been recently extended to unordered and unranked trees [11]. A potential extension of such works could consider grammar-based tree compression mechanisms more general than STGs. In particular, STGs do not allow repeated occurrences of parameters and hence, roughly speaking, do not allow *copying*. If we drop that restriction, we obtain nonlinear STGs, a formalism equivalent to Lamping's sharing graphs [20] that allows for a doubly exponential compression ratio [3]. However, the existence of an efficient equivalence checking procedure for nonlinear STGs is open, since the best known complexity upper bound for this problem in PSPACE [3]. I believe that this bound can be improved to co-NP, but the existence of a polynomial time algorithm does not seem plausible since, in the worst-case, $O(2^n)$ bits are needed to simply store the size of a term represented with a non-linear STG of size n. In any case, no hardness result is known.

Open Problem 5. *Is there a non-deterministic polynomial time decision procedure for equivalence checking of nonlinear STGs?*

3 Conclusion

In this talk I will review some of the results from Figure 1 and related techniques, and discuss potential approaches to solve the above open questions.

References

[1] Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.

[2] Alexander Bau, Markus Lohrey, Eric Nöth, and Johannes Waldmann. Compression of rewriting systems for termination analysis. In Femke van Raamsdonk, editor, *RTA*, volume 21 of *LIPIcs*, pages 97–112. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[3] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Efficient memory representation of XML document trees. *Inf. Syst.*, 33(4-5):456–474, 2008.

[4] Carles Creus, Adrià Gascón, and Guillem Godoy. One-context Unification with STG-Compressed Terms is in NP. In *RTA*, pages 149–164, 2012.

[5] Robert Dabrowski and Wojciech Plandowski. Solving two-variable word equations (extended abstract). In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, pages 408–419, 2004.

[6] Robert Dabrowski and Wojciech Plandowski. On word equations in one variable. *Algorithmica*, 60(4):819–828, 2011.

[7] Adrià Gascón, Guillem Godoy, and Manfred Schmidt-Schauß. Context matching for compressed terms. In *LICS*, pages 93–102, 2008.

[8] Adrià Gascón, Guillem Godoy, and Manfred Schmidt-Schauß. Unification with singleton tree grammars. In *RTA*, pages 365–379, 2009.

[9] Adrià Gascón, Guillem Godoy, and Manfred Schmidt-Schauß. Unification and Matching on Compressed Terms. *ACM Trans. Comput. Log.*, 12(4):26, 2011.

[10] Adrià Gascón, Guillem Godoy, Manfred Schmidt-Schauß, and Ashish Tiwari. Context Unification with One Context Variable. *J. Symb. Comput.*, 45(2):173–193, 2010.

[11] Adrià Gascón, Markus Lohrey, Sebastian Maneth, Carl Philipp Reh, and Kurt Sieber. Grammar-based compression of unranked trees. In *CSR*, volume 10846 of *Lecture Notes in Computer Science*, pages 118–131. Springer, 2018.

[12] Adrià Gascón, Sebastian Maneth, and Lander Ramos. First-order unification on compressed terms. In *RTA*, pages 51–60, 2011.

[13] Adrià Gascón, Manfred Schmidt-Schauß, and Ashish Tiwari. Two-restricted one context unification is in polynomial time. In *CSL*, volume 41 of *LIPIcs*, pages 405–422. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[14] Adrià Gascón, Ashish Tiwari, and Manfred Schmidt-Schauß. One context unification problems solvable in polynomial time. In *LICS*, pages 499–510. IEEE Computer Society, 2015.

[15] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theor. Comput. Sci.*, 13:225–230, 1981.

[16] Georg Gottlob, Christoph Koch, and Klaus U. Schulz. Conjunctive queries over trees. *J. ACM*, 53(2):238–272, 2006.

[17] Sumit Gulwani and Ashish Tiwari. Computing procedure summaries for interprocedural analysis. In *16th European Symposium on Programming Languages and Systems, ESOP 2007, part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007*, pages 253–267, 2007.

[18] Artur Jeż. One-variable word equations in linear time. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, pages 324–335, 2013.

[19] Artur Jeż. Context unification is in PSPACE. In *Proc. ICALP 2014, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 244–255. Springer, 2014.

[20] John Lamping. An algorithm for optimal lambda calculus reduction. In *POPL*, pages 16–30, 1990.

[21] Markus Lohrey. Grammar-based tree compression. In *DLT*, volume 9168 of *Lecture Notes in Computer Science*, pages 46–57. Springer, 2015.

[22] Markus Lohrey and Sebastian Maneth. The complexity of tree automata and XPath on grammar-compressed trees. *Theor. Comput. Sci.*, 363(2):196–210, 2006.

[23] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. XML tree structure compression using repair. *Inf. Syst.*, 38(8):1150–1167, 2013.

[24] Markus Lohrey, Sebastian Maneth, and Fabian Peternek. Compressed tree canonization. In *ICALP (2)*, volume 9135 of *Lecture Notes in Computer Science*, pages 337–349. Springer, 2015.

[25] Markus Lohrey, Sebastian Maneth, and Manfred Schmidt-Schauß. Parameter reduction and automata evaluation for grammar-compressed trees. *J. Comput. Syst. Sci.*, 78(5):1651–1669, 2012.

[26] G. S. Makanin. On the decidability of the theory of free groups (in russian). In *Fundamentals of Computation Theory, FCT '85, Cottbus, GDR, September 9-13, 1985*, pages 279–284, 1985.

[27] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[28] Joachim Niehren, Manfred Pinkal, and Peter Ruhrberg. A uniform approach to underspecification and parallelism. In *ACL*, pages 410–417, 1997.

[29] S. Eyono Obono, Pavel Goralcik, and M. N. Maksimenko. Efficient solving of the word equations in one variable. In *Mathematical Foundations of Computer Science 1994, 19th International Symposium, MFCS'94, Kosice, Slovakia, August 22 - 26, 1994, Proceedings*, pages 336–341, 1994.

[30] Mike Paterson and Mark N. Wegman. Linear unification. *J. Comput. Syst. Sci.*, 16(2):158–167, 1978.

[31] Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, 2004.

[32] Wojciech Plandowski and Wojciech Rytter. Application of lempel-ziv encodings to the solution of words equations. In *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 731–742. Springer, 1998.

[33] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[34] Manfred Schmidt-Schauß. Linear Compressed Pattern Matching for Polynomial rewriting (extended abstract). In *TERMGRAPH*, pages 29–40, 2013.

[35] Manfred Schmidt-Schauß, David Sabel, and Altug Anis. Congruence closure of compressed terms in polynomial time. In *FroCoS*, pages 227–242, 2011.

[36] Manfred Schmidt-Schauß and Jürgen Stuber. The complexity of linear and stratified context matching problems. *Theory Comput. Syst.*, 37(6):717–740, 2004.

Bounded ACh Unification

Ajay Kumar Eeralla[1]*and Christopher Lynch[2]

[1] Department of Computer Science, University of Missouri
Columbia, USA, `ae266@mail.missouri.edu`

[2] Department of Computer Science, Clarkson University
Potsdam, USA, `clynch@clarkson.edu`

Abstract

We consider the problem of unification modulo an equational theory ACh, which consists of a function h which is homomorphic over an associative-commutative operator $+$. Unification modulo ACh is undecidable, so we define a bounded ACh unification problem. In this bounded version of ACh unification we essentially bound the number of times h can be recursively applied to a term, and only allow solutions that satisfy this bound. There is no bound on the number of occurrences of h in a term, and the $+$ symbol can be applied an unlimited number of times. We give inference rules for solving bounded ACh unification, and we prove that the rules are sound, complete and terminating. We have implemented the algorithm in Maude and give experimental results. We argue that this algorithm is useful in cryptographic protocol analysis.

1 Introduction

Unification is a method to find a solution for a set of equations. For instance, consider an equation $x + y \stackrel{?}{=} a + b$, where x and y are variables, and a, and b are constants. If $+$ is an uninterpreted function symbol then the equation has one solution $\{x \mapsto a,\, y \mapsto b\}$, and this unification is called syntactic unification. If the function symbol $+$ has the property of commutativity then the equation has two solutions: $\{x \mapsto a,\, y \mapsto b\}$ and $\{x \mapsto b,\, y \mapsto a\}$; and this is called unification modulo the commutativity theory.

Unification modulo equational theories plays a significant role in symbolic cryptographic protocol analysis [7]. An overview and references for some of the algorithms may be seen in [8, 6]. One such equational theory is the distributive axioms: $x \times (y + z) = (x \times y) + (x \times z); (y + z) \times x = (y \times x) + (z \times x)$. A decision algorithm is presented for unification modulo two-sided distributivity in [12]. A sub-problem of this, unification modulo one-sided distributivity, is in greater interest since many cryptographic protocol algorithms satisfy the one-sided distributivity. In their paper [13], Tiden and Arnborg presented an algorithm for unification modulo one-sided distributivity: $x \times (y+z) = (x \times y) + (x \times z)$, and also it has been shown that it is undecidable if we add the properties of associativity $x+(y+z) = (x+y)+z$ and a one-sided unit element $x \times 1 = x$. However, some counterexamples [11] have been presented showing that the complexity of the algorithm is exponential, although they thought it was polynomial-time bounded.

For practical purposes, one-sided distributivity can be viewed as the homomorphism theory, $h(x + y) = h(x) + h(y)$, where the unary operator h distributes over the binary operator $+$. Homomorphisms are highly used in cryptographic protocol analysis. In fact, homomorphism is a common property that many election voting protocols satisfy [9].

Our goal is to present a novel construction of an algorithm to solve unification modulo the homomorphism theory over a binary symbol $+$ that also has the properties of associativity

*Ajay K. Eeralla was partially supported by NSF CNS-1314338

and commutativity (ACh), which is an undecidable unification problem [10]. Given that ACh unification is undecidable but necessary to analyze cryptographic protocols, we developed an approximation of ACh unification, which we show to be decidable.

In this paper, we present an algorithm to solve a modified general unification problem modulo the ACh theory, which we call *bounded ACh unification.* We define the *h-height* of a term to be basically the number of h symbols recursively applied to each other. We then only search for ACh unifiers of a bounded h-height. The number of occurrences of the $+$ symbol is not bounded. In order to accomplish this we define the *h-depth* of a variable, which is the number of h symbols on top of a variable. We develop a set of inference rules for ACh unification that keep track of the h-depth of variables. If the h-depth of any variable exceeds the bound κ then the algorithm terminates with no solution. Otherwise, it gives all the unifiers or solutions to the problem.

2 Preliminaries

We assume the reader is familiar with basic notation of unification theory and term rewriting systems (see for example [3, 4]).

Definition 1 (More General Substitution)**.** A substitution σ is more general than substitution θ if there exists a substitution η such that $\theta = \sigma\eta$, denoted as $\sigma \lesssim \theta$. Note that the relation $\lesssim$ is a quasi-ordering, i.e., reflexive and transitive.

Definition 2 (Unifier, Most General Unifier)**.** A substitution σ is a unifier or solution of two terms s and t if $s\sigma = t\sigma$; it is a most general unifier if for every unifier θ of s and t, $\sigma \lesssim \theta$. Moreover, a substitution σ is a solution of set of equations if it is a solution of each of the equations. If a substitution σ is a solution of a set of equations Γ, then it is denoted by $\sigma \models \Gamma$.

A set of identities E is a subset of $\mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathcal{T}(\mathcal{F}, \mathcal{V})$ and are represented in the form $s \approx t$. An equational theory $=_E$ is induced by a set of fixed identities E and it is the least congruence relation that is closed under substitution and contains E.

Definition 3 (E-Unification Problem, E-Unifier, E-Unifiable)**.** Let $\mathcal{F}$ be a signature and E be an equational theory. An E-unification problem over $\mathcal{F}$ is a finite set of equations $\Gamma = \{s_1 \overset{?}{=}_E t_1, \ldots, s_n \overset{?}{=}_E t_n\}$ between terms. An E-unifier or E-solution of two terms s and t is a substitution σ such that $s\sigma =_E t\sigma$. An E-unifier of Γ is a substitution σ such that $s_i\sigma =_E t_i\sigma$ for $i = 1, \ldots, n$. The set of all E-unifiers is denoted by $\mathcal{U}_E(\Gamma)$ and Γ is called E-unifiable if $\mathcal{U}_E(\Gamma) \neq \emptyset$. If $E = \emptyset$ then Γ is a syntactic unification problem.

Let $\Gamma = \{s_1 \overset{?}{=}_E t_1, \ldots, s_n \overset{?}{=}_E t_n\}$ be a set of equations, and let θ be a substitution. We say that θ satisfies Γ modulo equational theory E if θ is an E-solution of each equation in Γ, that is, $s_i\theta =_E t_i\theta$ for $i = 1, \ldots, n$. We write it as $\theta \models_E \Gamma$. Let $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ and θ be substitutions, and let E be an equational theory. We say that θ satisfies σ in the equational theory E if $x_i\theta =_E t_i\theta$ for $i = 1, \ldots, n$. We write it as $\theta \models_E \sigma$.

Definition 4 (Complete Set of E-Unifiers)**.** A complete set of E-unifiers of an E-unification problem Γ is a set S of idempotent E-unifiers of Γ such that for each $\theta \in \mathcal{U}_E(\Gamma)$ in Γ there exists $\sigma \in S$ with $\sigma \lesssim_E \theta|Var(\Gamma)$, where $Var(\Gamma)$ is the set of variables in Γ.

A complete set S of E-unifiers is minimal if for two distinct unifiers σ and θ in S, one is not more general than the other; i.e., if $\sigma \lesssim_E \theta|Var(\Gamma)$ and $\sigma, \theta \in S$ then $\sigma = \theta$. A minimal complete set of unifiers for a syntactic unification problem Γ has only one element if it is not empty. It is denoted by $mgu(\Gamma)$ and can be called most general unifier of unification problem Γ.

2.1 ACh Theory

The equational theory we consider is the theory of a homomorphism over a binary function symbol $+$. The symbol $+$ has the properties associativity and commutativity. We abbreviate this theory as ACh. The signature $\mathcal{F}$ includes a unary symbol h, and a binary symbol $+$, and other uninterpreted function symbols with fixed-arity. The function symbols h and $+$ in the signature $\mathcal{F}$ satisfy the identities: $x + (y + z) \approx (x + y) + z$ (Associativity, A for short); $x + y \approx y + x$ (Commutativity, C for short); $h(x + y) \approx h(x) + h(y)$ (Homomorphism, h for short).

2.2 h-Depth Set

For convenience, we assume that our unification problem is in *flattened* form, i.e., that every equation in the problem is in one of the following forms: $x \stackrel{?}{=} y$, $x \stackrel{?}{=} h(y)$, $x \stackrel{?}{=} y_1 + \cdots + y_n$, and $x \stackrel{?}{=} f(x_1, \ldots, x_n)$, where x, y, y_i, and x_i are variables, and f is a free symbol with $n \geq 0$. The first kind of equations are called *VarVar equations*. The second kind are called *h-equations*. The third kind are called *+-equations*. The fourth kind are called *free equations*.

Definition 5 (Graph $\mathbb{G}(\Gamma)$)**.** Let Γ be a unification problem. We define a graph $\mathbb{G}(\Gamma)$ as a graph where each node represents a variable in Γ and each edge represents a function symbol in Γ. To be exact, if an equation $w \stackrel{?}{=} f(x_1, \ldots, x_n)$, where f is a symbol with $n \geq 1$, is in Γ then the graph $\mathbb{G}(\Gamma)$ contains n edges $w \xrightarrow{f} x_1, \ldots, w \xrightarrow{f} x_n$. For a constant symbol c, if an equation $w \stackrel{?}{=} c$ is in Γ then the graph $\mathbb{G}(\Gamma)$ contains a vertex w. Finally, the graph $\mathbb{G}(\Gamma)$ contains two vertices if an equation $w \stackrel{?}{=} y$ is in Γ.

Definition 6 (h-Depth)**.** Let Γ be a unification problem and let x be a variable that occurs in Γ. Let h be a unary symbol and let f be a symbol (distinct from h) with arity greater than or equal to 1 and occur in Γ. We define h-depth of a variable x as the maximum number of h-symbols along a path to x in $\mathbb{G}(\Gamma)$, and it is denoted by $h_d(x, \Gamma)$. That is, $h_d(x, \Gamma) := \max\{h_{dh}(x, \Gamma), h_{df}(x, \Gamma), 0\}$, where $h_{dh}(x, \Gamma) := \max\{1 + h_d(y, \Gamma) \mid y \xrightarrow{h} x \text{ is an edge in } \mathbb{G}(\Gamma)\}$ and $h_{df}(x, \Gamma) := \max\{h_d(y, \Gamma) \mid \text{there exists } f \neq h \text{ such that } y \xrightarrow{f} x \text{ is in } \mathbb{G}(\Gamma)\}$.

Definition 7 (h-Height)**.** We define h-height of a term t as the following:

$$h_h(t) := \begin{cases} h_h(t') + 1 & \text{if} \quad t = h(t') \\ \max\{h_h(t_1), \ldots, h_h(t_n)\} & \text{if} \quad t = f(t_1, \ldots, t_n), f \neq h \\ 0 & \text{if} \quad t = x \,\text{or}\, c \end{cases}$$

where f is a function symbol with arity greater than or equal to 1.

Without loss of generality, we assume that h-depth and h-height is not defined for a variable that occurs on both sides of the equation. This is because the occur check rule—concludes the problem with no solution—presented in the next section has higher priority over the h-depth updating rules.

Definition 8 (h-Depth Set)**.** Let Γ be a set of equations. The set h-depth of Γ, denoted $h_{ds}(\Gamma)$, is defined as $h_{ds}(\Gamma) := \{(x, h_d(x, \Gamma)) \mid x \text{ is a variable appearing in } \Gamma\}$. In other words, the elements in the h-depth set are of the form $(x,\, c)$, where x is a variable that occur in Γ and c is a natural number representing the h-depth of x. Maximum value of h-depth set $\triangle$ is the maximum of all c values and it is denoted by $MaxVal(\triangle)$, i.e., $MaxVal(\triangle) := \max\{c \,|\, (x,\, c) \in \triangle\}$.

Definition 9 (Bounded E-Unification Problem, Bounded E-Unifier)**.** A κ bounded E-unification problem over $\mathcal{F}$ is a finite set of equations $\Gamma = \{s_1 \overset{?}{=}_E t_1, \dots, s_n \overset{?}{=}_E t_n\},\ s_i, t_i \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, where E is an equational theory, and κ is a positive integer. A κ bounded E-unifier or κ bounded E-solution of Γ is a substitution σ such that $s_i\sigma =_E t_i\sigma$, $h_h(s_i\sigma) \leq \kappa$, and $h_h(t_i\sigma) \leq \kappa$ for all i.

3 Inference System $\mathfrak{I}_h$

3.1 Problem Format

An inference system is a set of inference rules that transforms an equational unification problem into other. In our inference procedure, we use a set triple $\Gamma||\triangle||\sigma$, where Γ is a unification problem modulo the ACh theory, $\triangle$ is an h-depth set, and σ is a substitution. Let $\kappa \in \mathbb{N}$ be a bound on the h-depth of the variables. A substitution θ satisfies the set triple $\Gamma||\triangle||\sigma$ if θ satisfies every equation in Γ and σ, $MaxVal(\triangle) \leq \kappa$, and we write that relation as $\theta \models \Gamma||\triangle||\sigma$. We also use a special set triple $\perp$ for no solution in the inference procedure. Generally, the inference procedure is based on priority of rules and also uses don't care determinism when there is no priority. i.e., any one rule applied from a set of rules without priority. Initially, Γ is the non-empty set of equations to solve and the substitution σ is the identity substitution. The inference rules are applied until either the set of equations is empty with most general unifier σ or $\perp$ for no solution. Of course, the substitution σ is a κ bounded E-unifier of Γ.

An inference rule is written in the following form $\frac{\Gamma||\triangle||\sigma}{\Gamma'||\triangle'||\sigma'}$. This means that if something matches the top of this rule, then it is to be replaced with the bottom of the rule. In the proofs we will write inference rules as follows: $\Gamma||\triangle||\sigma \Rightarrow_{\mathfrak{I}_h} \{\Gamma_1||\triangle_1||\sigma_1, \cdots, \Gamma_n||\triangle_n|\sigma_n\}$ meaning to branch and replace the left hand side with one of the right hand sides in each branch. The only inference rule that has more than one branch is *AC Unification*. So we often just write inference rules as follows: $\Gamma||\triangle||\sigma \Rightarrow_{\mathfrak{I}_h} \Gamma'||\triangle'||\sigma'$. Let $\mathcal{OV}$ be the set of variables occurring in the unification problem Γ and let $\mathcal{NV}$ be a new set of variables such that $\mathcal{NV} = \mathcal{V} \setminus \mathcal{OV}$. Unless otherwise stated we assume that $x,\, x_1, \dots, x_n$, and $y,\, y_1, \dots, y_n,\, z$ are variables in $\mathcal{V}$, $v,\, v_1, \dots, v_n$ are in $\mathcal{NV}$, and terms $t,\, t_1, \dots, t_n,\, s,\, s_1, \dots, s_n$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$, and f and g are uninterpreted function symbols. Recall that h is a unary, and the associativity and the commutativity operator $+$. A fresh variable is a variable that is generated by the current inference rule and has never been used before.

For convenience, we assume that every equation in the problem is in one of the following flattened forms: $x \overset{?}{=} y$, $x \overset{?}{=} h(y)$, and $z \overset{?}{=} y + x$, where x, y, and z are variables. If not, we apply flattening rules to put the equations into that form. These rules are performed before any other inference rule. They put the problem into flattened form and all the other inference rules leave the problem in flattened form, so there is no need to perform these rules again later. It is necessary to update the h-depth set $\triangle$ with the h-depth values for each variable during the inference procedure.

3.2 Inference Rules

Flattening. We present a set of inference rules for flattening. The variable v represents a fresh variable in the following rules.

Flatten Both Sides

$$\frac{\{t_1 \overset{?}{=} t_2\} \cup \Gamma || \triangle || \sigma}{\{v \overset{?}{=} t_1,\, v \overset{?}{=} t_2\} \cup \Gamma || \{(v,\, 0)\} \cup \triangle || \sigma} \text{ if } t_1 \text{ and } t_2 \notin \mathcal{V}$$

Flatten Left +

$$\frac{\{t \overset{?}{=} t_1 + t_2\} \cup \Gamma || \triangle || \sigma}{\{t \overset{?}{=} v + t_2,\, v \overset{?}{=} t_1\} \cup \Gamma || \{(v,\, 0)\} \cup \triangle || \sigma} \qquad \text{if } t_1 \notin \mathcal{V}$$

Flatten Right +

$$\frac{\{t \overset{?}{=} t_1 + t_2\} \cup \Gamma || \triangle || \sigma}{\{t \overset{?}{=} t_1 + v,\, v \overset{?}{=} t_2\} \cup \Gamma || \{(v,\, 0)\} \cup \triangle || \sigma} \qquad \text{if } t_2 \notin \mathcal{V}$$

Flatten Under h

$$\frac{\{t_1 \overset{?}{=} h(t)\} \cup \Gamma || \triangle || \sigma}{\{t_1 \overset{?}{=} h(v),\, v \overset{?}{=} t\} \cup \Gamma || \{(v,\, 0)\} \cup \triangle || \sigma} \qquad \text{if } t \notin \mathcal{V}$$

Update h-Depth Set. We also present a set of inference rules to update the h-depth set. We apply these rules immediately after applying any other rule in the inference system.
Update h

$$\frac{\{x \overset{?}{=} h(y)\} \cup \Gamma || \{(x,\, c_1),\, (y,\, c_2)\} \cup \triangle || \sigma}{\{x \overset{?}{=} h(y)\} \cup \Gamma || \{(x,\, c_1),\, (y,\, c_1 + 1)\} \cup \triangle || \sigma} \qquad \text{If } c_2 < (c_1 + 1)$$

Update +

1. $$\frac{\{x_1 \overset{?}{=} y_1 + y_2\} \cup \Gamma || \{(x_1,\, c_1),\, (y_1,\, c_2),\, (y_2,\, c_3)\} \cup \triangle || \sigma}{\{x_1 \overset{?}{=} y_1 + y_2\} \cup \Gamma || \{(x_1,\, c_1),\, (y_1,\, c_1),\, (y_2,\, c_3)\} \cup \triangle || \sigma} \text{ If } c_2 < c_1$$

2. $$\frac{\{x_1 \overset{?}{=} y_1 + y_2\} \cup \Gamma || \{(x_1,\, c_1),\, (y_1,\, c_2),\, (y_2,\, c_3)\} \cup \triangle || \sigma}{\{x_1 \overset{?}{=} y_1 + y_2\} \cup \Gamma || \{(x_1,\, c_1),\, (y_1,\, c_2),\, (y_2,\, c_1)\} \cup \triangle || \sigma} \text{ If } c_3 < c_1$$

Splitting Rule. This rule takes the homomorphism theory into account. In this theory, we can not solve equation $h(y) \overset{?}{=} x_1 + x_2$ unless y can be written as the sum of two new variables $y = v_1 + v_2$, where v_1 and v_2 are in $\mathcal{NV}$. Without loss of generality we generalize it to n variables $x_1, \ldots, x_n$.

$$\frac{\{w \overset{?}{=} h(y), w \overset{?}{=} x_1 + \cdots + x_n\} \cup \Gamma || \triangle || \sigma}{\{w \overset{?}{=} h(y), y \overset{?}{=} v_1 + \cdots + v_n, x_1 \overset{?}{=} h(v_1), \ldots, x_n \overset{?}{=} h(v_n)\} \cup \Gamma || \triangle' || \sigma}$$

where $n > 1$, $y \neq w$, $\triangle' = \{(v_1,\, 0), \ldots, (v_n,\, 0)\} \cup \triangle\}$, and $v_1, \ldots, v_n$ are fresh variables in $\mathcal{NV}$.
Trivial. The Trivial inference rule is to remove trivial equations in the given problem Γ.

$$\frac{\{t \overset{?}{=} t\} \cup \Gamma || \triangle || \sigma}{\Gamma || \triangle || \sigma}$$

Variable Elimination (VE). The Variable Elimination rule is to convert the equations into assignments. In other words, it is used to find the most general unifier. The rule VE-2 is performed last after all other inference rules have been performed. The rule VE-1 is performed eagerly.

1.
$$\frac{\{x \overset{?}{=} y\} \cup \Gamma||\triangle||\sigma}{\Gamma\{x \mapsto y\}||\triangle||\sigma\{x \mapsto y\} \cup \{x \mapsto y\}}$$

2.
$$\frac{\{x \overset{?}{=} t\} \cup \Gamma||\triangle||\sigma}{\Gamma||\triangle||\sigma\{x \mapsto t\} \cup \{x \mapsto t\}} \qquad \text{if } t \notin \mathcal{V} \text{ and } x \text{ does not occur in } t$$

Decomposition. The Decomposition rule decomposes an equation into several sub-equations if both sides top symbol matches.

$$\frac{\{x \overset{?}{=} f(s_1, \dots, s_n), x \overset{?}{=} f(t_1, \dots, t_n)\} \cup \Gamma||\triangle||\sigma}{\{x \overset{?}{=} f(t_1, \dots, t_n),\ s_1 \overset{?}{=} t_1, \dots, s_n \overset{?}{=} t_n\} \cup \Gamma||\triangle||\sigma} \qquad \text{if } f \neq +$$

AC Unification. The AC Unification rule calls an AC unification algorithm to unify the AC part of the problem. Notice that we apply AC unification only once when no other rule except VE-2 can apply. In this inference rule Ψ represents the set of all equations with the + symbol on the right hand side. Γ represents the set of equations not containing a + symbol. *Unify* is a function that returns one of the complete set of unifiers returned by the AC unification algorithm. *GetEqs* is a function that takes a substitution and returns the equational form of that substitution. In other words, $GetEqs([x_1 \mapsto t_1, \dots, x_n \mapsto t_n]) = \{x_1 \overset{?}{=} t_1, \dots, x_n \overset{?}{=} t_n\}$.

$$\frac{\Psi \cup \Gamma||\triangle||\sigma}{GetEqs(Unify\ \Psi) \cup \Gamma||\triangle||\sigma}$$

Note that we have written the rule for one member of the complete set of AC unifiers of Ψ. This will branch on every member of the complete set of AC unifiers of Ψ.

Occur Check. It is to determine if a variable on the left hand side of an equation occurs on the other side of the equation. If it does, then there is no solution to the unification problem. This rule has the highest priority.

$$\frac{\{x \overset{?}{=} f(t_1, \dots, t_n)\} \cup \Gamma||\triangle||\sigma}{\perp} \qquad \text{If } x \in \mathcal{V}ar(f(t_1, \dots, t_n)\sigma)$$

where $\mathcal{V}ar(f(t_1, \dots, t_n)\sigma)$ represents set of all variables that occur in $f(t_1, \dots, t_n)\sigma$.

Clash. This rule checks if the *top symbol* on both sides of an equation is the same. If not, then there is no solution to the problem, unless one of them is h and the other $+$.

$$\frac{\{x \overset{?}{=} f(s_1, \dots, s_m),\ x \overset{?}{=} g(t_1, \dots, t_n)\} \cup \Gamma||\triangle||\sigma}{\perp} \qquad \text{If } f \notin \{h, +\} \text{ or } g \notin \{h, +\}$$

Bound Check. The Bound Check is to determine if a solution exists within the bound κ, a given maximum h-depth of any variable in Γ. If one of the h-depths in the h-depth set $\triangle$ exceeds the bound κ, then the problem has no solution.

$$\frac{\Gamma||\triangle||\sigma}{\perp} \qquad \text{If } MaxVal(\triangle) > \kappa$$

Unification Problem	Real Time	Solution	# Sol.	Bound
$\{h(y) \overset{?}{=} y+x\}$	674ms	$\perp$	0	10
$\{h(y) \overset{?}{=} y+x\}$	15880ms	$\perp$	0	20
$\{h(y) \overset{?}{=} x_1+x_2\}$	5ms	Yes	1	10
$\{h(h(x)) \overset{?}{=} h(h(y))\}$	2ms	Yes	1	10
$\{x+y_1 \overset{?}{=} x+y_2\}$	3ms	Yes	1	10
$\{v \overset{?}{=} x+y, v \overset{?}{=} w+z, s \overset{?}{=} h(t)\}$	46ms	Yes	10	10
$\{v \overset{?}{=} x_1+x_2, v \overset{?}{=} x_3+x_4, x_1 \overset{?}{=} h(y), x_2 \overset{?}{=} h(y)\}$	100ms	Yes	6	10
$\{h(h(x)) \overset{?}{=} v+w+y+z\}$	224ms	Yes	1	10
$\{v \overset{?}{=} (h(x)+y), v \overset{?}{=} w+z\}$	55ms	Yes	7	10
$\{f(x,y) \overset{?}{=} h(x_1)\}$	0ms	$\perp$	0	10
$\{f(x_1,y_1) \overset{?}{=} f(x_2,y_2)\}$	1ms	Yes	1	10
$\{v \overset{?}{=} x_1+x_2, v \overset{?}{=} x_3+x_4\}$	17ms	Yes	7	10
$\{f(x_1,y_1) \overset{?}{=} g(x_2,y_2)\}$	0ms	$\perp$	0	10
$\{h(y) \overset{?}{=} x, y \overset{?}{=} h(x)\}$	0ms	$\perp$	0	10

Table 1: Tested results with bounded ACh-unification algorithm

4 Proof of Correctness

The proposed inference process eventually halts.

Lemma 1. There is no infinite sequence of inference rules.

Our inference system is truth-preserving.

Lemma 2 (**Soundness**)**.** Let $\Gamma||\triangle||\sigma \Rightarrow_{\mathfrak{I}_h} \{\Gamma_1||\triangle_1||\sigma_1, \cdots, \Gamma_n||\triangle_n|\sigma_n\}$ be an inference rule. Let θ be a substitution such that $\theta \models \Gamma_i||\triangle_i||\sigma_i$. Then $\theta \models \Gamma||\triangle||\sigma$.

Of course, our inference system never loses any solution.

Lemma 3 (**Completeness**)**.** Let $\Gamma||\triangle||\sigma$ be a set triple. Let $\Gamma||\triangle||\sigma \Rightarrow_{\mathfrak{I}_h} \{\Gamma_1||\triangle_1||\sigma_1, \cdots,$ $\Gamma_n||\triangle_n|\sigma_n\}$ be an inference rule. If $\theta \models \Gamma||\triangle||\sigma$, then there exists an i and a θ', whose domain is the variables in $Var(\Gamma_i) \setminus Var(\Gamma)$, such that $\theta\theta' \models \Gamma_i||\triangle_i||\sigma_i$.

5 Implementation

We have implemented the algorithm in Maude [5]. We chose the Maude language because the inference rules are very similar to the rules of Maude and an implementation will be integrated into the Maude-NPA tool at some time. The Maude-NPA tool is written in Maude. The system specifications are Ubuntu 14.04 LTS, Intel Core i5 3.20 GHz, and 8 GiB RAM with Maude 2.6.

We give a table to show some of our results. In the given table, we use five columns: Unification problem, Real Time, time to terminate the program in ms (milli seconds), Solution either $\perp$ for no solution or Yes for solutions, # Sol. for number of solutions, and Bound κ. It makes sense that the real time keeps increasing as the given h-depth κ increases for the first problem where the other problems give solutions, but in either case the program terminates.

6 Conclusion

We introduced a set of inference rules to solve the unification problem modulo the homomorphism theory h over an AC symbol $+$, by enforcing bound k on the h-depth of any variable.

Homomorphism is a property which is very common in cryptographic algorithms. So, it is important to analyze cryptographic protocols in the homomorphism theory. Some of the algorithms and details in this direction can be seen in [2, 6, 1]. However, none of those results perform ACh unification because that is undecidable. One way around this is to assume that an identity and an inverse exist, but because of the way the Maude-NPA works it would still be necessary to unify modulo ACh. So an unification algorithm there becomes crucial. We believe that our approximation is a good way to deal with it. We also tested some problems and the results are shown in Table 1.

References

[1] S. Anantharaman, H. Lin, C. Lynch, P. Narendran, and M. Rusinowitch. Cap unification: application to protocol security modulo homomorphic encryption. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 192–203. ACM, 2010.

[2] S. Anantharaman, H. Lin, C. Lynch, P. Narendran, and M. Rusinowitch. Unification Modulo Homomorphic Encryption. In *booktitle of Automated Reasoning*, pages 135–158. Springer, 2012.

[3] F. Baader and T. Nipkow. *Term Rewriting and All that.* Cambridge University Press, 1998.

[4] F. Baader and W. Snyder. Unification Theory. In *Handbook of Automated Reasoning*, pages 447–533. Elsevier, 2001.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and Carolyn L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic.* Springer, 2007.

[6] S. Escobar, D. Kapur, C. Lynch, C. Meadows, J. Meseguer, P. Narendran, and R. Sasse. Protocol Analysis in Maude-NPA Using Unification Modulo Homomorphic Encryption. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, pages 65–76. ACM, 2011.

[7] S. Escobar, C. Meadows, and J. Meseguer. Maude-Npa: cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures*, pages 1–50. Springer, 2007.

[8] D. Kapur, P. Narendran, and L. Wang. An E-unification Algorithm for Analyzing Protocols That Use Modular Exponentiation. In *Rewriting Techniques and Applications*, pages 165–179. Springer, 2003.

[9] S. Kremer, M. Ryan, and B. Smyth. Election Verifiability in Electronic Voting Protocols. In *Computer Security – ESORICS*, pages 389–404. Springer, 2010.

[10] P. Narendran. Solving Linear Equations over Polynomial Semirings. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 466–472. IEEE Computer Society, 1996.

[11] P. Narendran, Andrew M. Marshall, and B. Mahapatra. On the Complexity of the Tiden-Arnborg Algorithm for Unification modulo One-Sided Distributivity. In *Proceedings 24th International Workshop on Unification*, pages 54–63. Open Publishing Association, 2010.

[12] M. Schmidt-Schauß. A Decision Algorithm for Distributive Unification. In *Theoretical Computer Science*, pages 111–148. Elsevier, 1998.

[13] E. Tidén and Stefan Arnborg. Unification Problems with One-Sided Distributivity. In *booktitle of Symbolic Computation*, pages 183–202. Springer, 1987.

32nd International Workshop on Unification (UNIF 2018), Informal Proceedings
Oxford, 7th July, 2018

Towards Generalization Methods for Purely Idempotent Equational Theories [*]

David M. Cerna and Temur Kutsia

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz
{David.Cerna , Temur.Kutsia}@risc.jku.at

Abstract

In *Generalisation de termes en theorie equationnelle. Cas associatif-commutatif*, a pair of terms was presented over the language $\{f(,),g(,),a,b\}$, where f and g are interpreted over an idempotent equational theory, i.e. $g(x,x)=x$ and $f(x,x)=x$, resulting in an infinite set of generalizations. While this result provides an answer to the complexity of the idempotent generalization problem for arbitrarily idempotent equational theories (theories with two or more idempotent functions) the complexity of generalization for equational theories with a single idempotent function symbols was left unsolved. We show that the two idempotent function symbols example can be encoded using a single idempotent function and two uninterpreted constants thus proving that idempotent generalization, even with a single idempotent function symbol, can result in an infinite set of generalizations. Based on this result we discuss approaches to handling generalization within idempotent equational theories.

1 Introduction

Anti-unification or term generalization algorithms aim at computing generalizations for given terms. A generalization of t and s is a term r such that t and s are substitution instances of r, i.e. there exists σ and μ such that $r\sigma = t$ and $r\mu = s$. Interesting generalizations are those that are least general (lggs). However, it is not always possible to have a unique lgg. In these cases the task is either to compute a minimal complete set of generalizations, or to impose restrictions so that uniqueness is guaranteed.

In particular, we consider anti-unification problems which allow equational interpretations of the function symbols and constants present in the term signature. This is known as equational anti-unification or E-generalization. When the equational theory does not interpret any of the function symbols or constants in the term signature the resulting generalizations are referred to as syntactic. For most of the commonly considered equational theories the minimal complete set of generalizations is finite, for example, theories including commutativity and associative function symbols discussed in [1]. However, as pointed out in [8], this need not be the case. A pair of terms constructed from the signature $\{f(\cdot,\cdot),g(\cdot,\cdot),a,b\}$ where f and g are interpreted as idempotent functions resulted in an infinite set of generalizations, though it was not shown to be the minimal complete set. While the case of two idempotent function symbols was addressed in [8], the case of generalization for terms constructed from a signature with a single

[*]This research is supported by the FWF project P28789-N32.

idempotent function symbol, i.e. $\{f(\cdot,\cdot),a,b\}$ was not discussed. This gap implies an interesting question concerning modular generalization algorithms like the ones discussed in [1].

The result reported in this paper has been motivated by its influence on developing a combination method for signature-disjoint generalization theories. Namely, as shown by Pottier in [8], anti-unification problems with two idempotent function symbols may have infinitely many incomparable generalizations. If anti-unification problems with one idempotent symbol had only finitely many incomparable solutions, it would be a serious problem for the prospect of developing a combination method: finitary generalization algorithms would have been impossible to combine. However, our result shows that it is not the case.

Combination methods for unification algorithms, constraint solvers, and decision procedures have been studied in detail [2, 3, 5, 4, 6, 9, 10]. Though surprisingly, it has been shown that term generalization when the signature contains a function which is associative, commutative and idempotent is finitary. This follows from Theorem 2 of [7]. Such varying results provide motivations for investigating term generalization as discussed in this paper and removes an obstacle to study such methods for generalization algorithms as well.

2 Preliminaries

We now outline the basic concepts needed to understand term generalizations and the results outlined in later sections. Our term language $\mathscr{L}$ is built from a finite signature of function and constant symbols Σ and a countable set variable symbols $\mathscr{V}$. Function symbols have a fixed arity, i.e. the number of arguments, and constant symbols are essentially function symbols with arity zero. If necessary, we denote the set of variables of a term t by $\text{Vars}(t)$.

Each symbol $f \in \Sigma$ in the signature has an associated equational theory $Ax(f)$. When $Ax(f)$ is empty the function or constant symbol is left uninterpreted. We will only consider in this work function symbols f interpreted as *idempotent*, $Ax(f) = \{\mathbf{I}\}$, that is binary functions such that $f(x,x) = x$.

When two terms s and t are equivalent over an equational theory $\mathscr{E}$ we write $s =_{\mathscr{E}} t$. In this work we will only consider the equational theory $\mathbf{I}_F$ where F is a set of function symbols interpreted as idempotent.

A *Substitution* is a finite set of pairs $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ where X_i is a variable, t_i is a term, and the X's are pairwise distinct variables. The notions of substitution *domain* and *range* are also standard and are denoted, respectively, by Dom and Ran.

We use postfix notation for substitution applications, writing $t\sigma$ instead of $\sigma(t)$. As usual, the application $t\sigma$ affects only the occurrences of variables from $\text{Dom}(\sigma)$ in t. The *composition* of σ and ϑ is written as juxtaposition $\sigma\vartheta$ and is defined as $x(\sigma\vartheta) = (x\sigma)\vartheta$ for all x.

A substitution σ_1 is *more general* than σ_2, written $\sigma_1 \preceq \sigma_2$, if there exists ϑ such that $X\sigma_1\vartheta = X\sigma_2$ for all $X \in \text{Dom}(\sigma_1) \cup \text{Dom}(\sigma_2)$. The strict part of this relation is denoted by $\prec$. The relation $\preceq$ is a partial order and generates the equivalence relation which we denote by $\simeq$. We overload $\preceq$ by defining $s \preceq t$ if there exists a substitution σ such that $s\sigma = t$. The focus of this work is generalization in the presence of equational axioms thus we need a more general concept of ordering of substitutions/terms by their generality. We say for two terms s,t are $s =_{\mathscr{E}} t$ if they are equivalent modulo $\mathscr{E}$. Under this notion of equality we can say that a substitution σ_1 is *more general modulo an equational theory $\mathscr{E}$* than σ_2 written $\sigma_1 \preceq_{\mathscr{E}} \sigma_2$ if there exists ϑ such that $X\sigma_1\vartheta =_{\mathscr{E}} X\sigma_2$ for all $X \in \text{Dom}(\sigma_1) \cup \text{Dom}(\sigma_2)$ Note that $\prec$ and $\simeq$ and the term extension are generalized accordingly. From this point on we will use the ordering relation modulo an equational theory when discussing generalization.

A term t is called a *generalization* or an *anti-instance* modulo an equational theory $\mathscr{E}$ of two terms t_1 and t_2 if $t \preceq_{\mathscr{E}} t_1$ and $t \preceq_{\mathscr{E}} t_2$. It is the *least general generalization* (lgg in short), aka a *most specific anti-instance,* of t_1 and t_2, if there is no generalization s of t_1 and t_2 which satisfies $t \prec_{\mathscr{E}} s$.

An *anti-unification problem* (Briefly AUP) is a triple $X : t \triangleq s$ where t and s are terms constructed from the signature Σ, and X does not occur in t and s. The variable X is called a *generalization variable.*

Generalization variables are written with capital letters $X, Y, Z, \ldots$. Note that generalization variables are not used explicitly in this work but they sere syntactic purpose in most algorithms defined in literature, thus we keep them to conform with common syntactic expressions. The size of a set of AUPs is defined as $|\{X_1 : t_1 \triangleq s_1, \ldots, X_n : t_n \triangleq s_n\}| = \sum_{i=1}^{n} |t_i| + |s_i|$. A *generalization* of an AUP $X : t \triangleq s$ is a term r such that there exists substitutions σ_1 and σ_2 such that $\mathrm{Dom}(\sigma_1) = \mathrm{Dom}(\sigma_2) = \mathcal{V}(r)$ and $r\sigma_1 = t$ and $r\sigma_2 = s$

A generalization r of $X : t \triangleq s$ is *least general* (or *most specific*) modulo an equational theory $\mathscr{E}$ if there is no generalization r' of $X : t \triangleq s$ such that $r \prec_{\mathscr{E}} r'$.

3 Idempotent Generalization with two symbols

We will now consider an alphabet $\Sigma = \{f(\cdot,\cdot), g(\cdot,\cdot), a, b\}$. Both f and g are idempotent function symbols (our equational theory is $\mathscr{E} = \mathbf{I}_{f,g}$), that is $f(x,x) =_{\mathbf{I}_{f,g}} x$ and $g(x,x) =_{\mathbf{I}_{f,g}} x$. Now we consider the following generalization problem:

$$X : f(a,b) \triangleq g(a,b) \tag{1}$$

The seemingly simple generalization problem of Equation 1 results in an infinite set of least general generalizations. This follows from the production of the first two least general generalizers $g(f(a,x), f(y,b))$ and $f(g(a,x), g(y,b))$ which we refer to as G_1 and G_2, respectively. It is quite simple to check that these two terms are indeed generalizers and are least general. In [8], an infinite set of generalizations was produced by the following recursive construction:

$$S_0 = G_1 \qquad\qquad S_{n+1} = f(G_1, g(S_n, G_2)) \tag{2}$$

Notice that the generalizer produced at each step is least general and is not comparable with the generalizers produced at the previous step and thus, the construction produces an infinite sequence of incomparable least general generalizers. However this is not the minimal complete set being that the construction is limited to repeated use of $\{G_1, G_2\}$. Any previously constructed generalizer can be used. Essentially, let $h \in \{f, g\}$ and $\mathscr{S}$ the set of least general generalizations of Equation 1, then $h(S_1, S_2)$ is a least general generalizations of Equation 1 when S_1 is distinct from S_2. We elucidate this construction further after presenting our solution to the generalization problem for one idempotent function symbol.

4 Idempotent Generalization with a single function symbol

We will now consider an alphabet $\Sigma = \{h(\cdot,\cdot), a, b\}$ where h is an idempotent function symbol (our equational theory is $\mathscr{E} = \mathbf{I}_h$). Our goal is to show that a term signature with a single binary function symbol interpreted as idempotent also allows the construction of AUPs with infinitely many lggs. We solve this problem by encoding the two symbol case into the one symbol case. Essentially we write $f(\cdot,\cdot)$ as $h(a, h(\cdot,\cdot))$ and $g(\cdot,\cdot)$ as $h(b, h(\cdot,\cdot))$. Thus, the generalization problem from the previous section is now:

$$h(a, h(a,b)) \triangleq h(b, h(a,b)) \tag{3}$$

The reader might notice right away that this has a solution $h(x, h(a,b))$, however, this solution isn't of much interest to us because we cannot produce an infinite construction using it alone, but it can be considered as one of the least generalizers within the construction. Also, it happens to be the case that there are two additional least general generalizers which are incomparable to it. These generalizers, which are incomparable to $h(x, h(a,b))$, are as follows:

$$G'_1 = h(h(x, h(x,b)), h(a, h(x,b))) \qquad G'_2 = \qquad h(f(x, h(a,x)), h(h(x,b), h(a,b)))$$

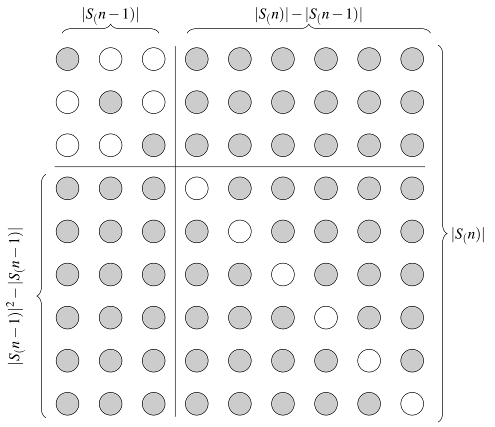

Figure 1: Geometric proof of Theorem 1 for $|A| = 3$.

Notice that these generalizers are even simpler than those produced in the previous example given that the domain of the substitutions contain a single variable x. Furthermore, this variable is substituted by one of the two constants. Using the recursive construction outlined in Equation 2, replacing G_1 and G_2 by G_1' and G_2' we produce a similar infinite set as in the Pottier example.

Concerning the construction of all least general generalizations constructable from the set $\{h(x,h(a,b)),G_1',G_2'\}$ consider the following theorem.

Theorem 1. *Let A be a finite set, $P(S,S') = \{(a,b) | a \in S\ ,\ b \in S'\ ,\ a \neq b\}$, and S_n the following recursive set construction:$S_0 = \{\emptyset\}$, $S_1 = A$, and $S_{(n+1)} = S_n \cup P(S_n,S_n)$. Then for $n \geq 1$, $|S_{(n+1)}| = |S_n|^2 - |S_{(n-1)}|^2 + |S_{(n-1)}|$.*

Proof. Let us consider the case of $S_2 = S_1 \cup P(S_1,S_1)$ we know that $|S_1| = |A|$ and that $|P(S_1,S_1)| = |A|^2 - |A|$ because $(a,a) \notin P(S_1,S_1)$ for $a \in A$. Thus, $|S_2| = |A|^2$ which is precisely given by the formula in the theorem $|S_2| = |S_1|^2 - |S_0|^2 + |S_0| = |A|^2 - 1 + 1 = |A|^2$. For the induction hypothesis, let us assume the theorem holds for S_n and show that it holds for $S_{(n+1)}$. We know that S_n contains $S_{(n-1)}$ by definition and thus we can consider the subsets of S_n, $S_{(n-1)}$ and $S_n \setminus S_{(n-1)}$, Note that the elements of $P(S_{(n-1)},S_{(n-1)})$ are already members of $S_n \setminus S_{(n-1)}$ and thus do not need to be considered. But we do need to consider the following cases $P(S_{(n-1)},S_n \setminus S_{(n-1)})$, $P(S_n \setminus S_{(n-1)},S_{(n-1)})$, $P(S_n \setminus S_{(n-1)},S_n \setminus S_{(n-1)})$ which have size $|S_{(n-1)}|(|S_n| - |S_{(n-1)}|$, $|S_{(n-1)}|(|S_n| - |S_{(n-1)}|)$, $(|S_n| - |S_{(n-1)}|)^2 - (|S_n| - |S_{(n-1)}|)$, respectively. Thus, we get that the size $|S_{(n+1)}|$ is the following:

$$2 \cdot |S_{(n-1)}|(|S_n| - |S_{(n-1)}| + (|S_n| - |S_{(n-1)}|)^2 - (|S_n| - |S_{(n-1)}|) + |S_n| =$$

$$2|S_{(n-1)}||S_n| - 2|S_{(n-1)}|^2 + (|S_n|^2 - 2|S_n||S_{(n-1)}| + |S_{(n-1)}|^2 + |S_{(n-1)}| = |S_n|^2 - |S_{(n-1)}|^2 + |S_{(n-1)}|$$

Proving the induction step. See Figure 1 for a geometric proof of the theorem. □

Concerning the $O(2^{2^n})$ growth rate in terms of the initial set size $|S_1|$, consider the ratio between the smaller square's area $|S(n-1)|$ and the larger square's $(|S(n-1)|^2 - |S(n-1)|)^2$ which is precisely $1 : O(|S_{n-1}|^2)$. Iterating this provides $O(2^{2^n})$ growth rate.

5 Conclusion

We have shown that even a simple term signature with a single binary function interpreted as idempotent results in an infinite set of generalizations. Theorem 1 provides information concerning the growth of the set of least general generalizations in terms of the number of nestings of idempotent function symbols. Further analysis provides a growth rate of $O(2^{2^n})$ in terms of the number of nested function symbols. This implies that the minimal complete set of generalizations is at least as large as this construction and thus infinite. However, we have not provided a precise construction of the minimal complete set of generalizations, only a lower bound. In future work we plan to investigate the construction of the minimal complete set of generalizations and hopefully find a precise expression of its construction as well as an understanding of modular algorithms for idempotent generalization.

6 References

References

[1] M. Alpuente, S. Escobar, J. Espert, and J. Meseguer. A modular order-sorted equational generalization algorithm. *Inf. Comput.*, 235:98–136, 2014.

[2] F. Baader and K. U. Schulz. Combination techniques and decision problems for disunification. *Theor. Comput. Sci.*, 142(2):229–255, 1995.

[3] F. Baader and K. U. Schulz. Unification in the union of disjoint equational theories:combining decision procedures. *J. Symb. Comput.*, 21(2):211–243, 1996.

[4] F. Baader and K. U. Schulz. Combining constraint solving. In H. Comon, C. Marché, and R. Treinen, editors, *Constraints in Computational Logics: Theory and Applications, International Summer School, CCL'99 Gif-sur-Yvette, France, September 5-8, 1999, Revised Lectures*, volume 2002 of *Lecture Notes in Computer Science*, pages 104–158. Springer, 1999.

[5] P. Chocron, P. Fontaine, and C. Ringeissen. A gentle non-disjoint combination of satisfiability procedures. In *IJCAR 2014*, pages 122–136, 2014.

[6] S. Erbatur, D. Kapur, A. M. Marshall, P. Narendran, and C. Ringeissen. Unification and matching in hierarchical combinations of syntactic theories. In C. Lutz and S. Ranise, editors, *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings*, volume 9322 of *Lecture Notes in Computer Science*, pages 291–306. Springer, 2015.

[7] B. Konev and T. Kutsia. Anti-unification of concepts in description logic EL. In C. Baral, J. P. Delgrande, and F. Wolter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016.*, pages 227–236. AAAI Press, 2016.

[8] L. Pottier. Generalisation de termes en theorie equationnelle. cas associatif-commutatif. Research Report 1056, INRIA, 1989.

[9] M. Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. *J. Symb. Comput.*, 8(1/2):51–99, 1989.

[10] C. Tinelli and C. Ringeissen. Unions of non-disjoint theories and combinations of satisfiability procedures. *Theor. Comput. Sci.*, 290(1):291–353, 2003.

32nd International Workshop on Unification (UNIF 2018), Informal Proceedings
Oxford, 7th July, 2018

Proximity-Based Generalization*

Temur Kutsia and Cleo Pau

RISC, Johannes Kepler University Linz, Austria
{kutsia,ipau}@risc.jku.at

Abstract

We study anti-unification under the condition that some mismatches in the names of function symbols is tolerated. The mismatches are expressed by proximity relations, which are reflexive, symmetric, but not transitive fuzzy binary relations. Their crisp version corresponds to undirected graphs. Computing all maximal clique partitions in them is needed to compute least general generalizations with respect to proximity relations. We report about our progress in developing both all-clique-partitions and anti-unification algorithms.

1 Introduction

In this paper we study anti-unification under the condition that some mismatches between function symbol names is tolerated. Names which are 'close to each other' should not be distinguished. Such a treatment of symbols is adequate in situations where one has to manage imprecise information, for example, for detecting clones in copied and slightly modified software code.

The relation of 'being close' is not transitive: For instance, if one considers two cities being close to each other if the distance between them is not more than 200 km, then Salzburg is close to Linz (133 km) and Linz is close to Vienna (185 km), but Salzburg is not close to Vienna (300 km). Nontransitivity has to be dealt in a special way. Proximity relations (reflexive symmetric fuzzy binary relations) characterize the notion of 'being close' numerically. They become crisp once we fix the threshold from which on the distance between the objects can be called 'close'.

We consider a first-order language where function symbols are in a proximity relation with each other. To compute generalizations in this language, we assume that the threshold (so called λ-cut) is fixed and we know which symbols are close to each other and which are not. The obtained crisp relation can be represented as an undirected graph, and symbols that belong to a complete subgraph (a clique) of it should be considered 'close' to each other. But the same symbol might belong to two or more cliques, and we need to choose one of them to which the symbol is assigned. Hence, we essentially need to partition the graph into maximal cliques, treat the symbols in the same clique as being the same, and apply the first-order anti-unification algorithm to compute generalizations.

This approach leads to two problems. First, while one can easily compute one maximal vertex-clique partition of a graph [7, 4], computing all maximal ones is more involved. But we need them to compute least general generalizations. We were not able to find an appropriate algorithm in the literature and, therefore, decided to design and implement one ourselves. It is optimal in the sense that each maximal

*Supported by Austrian Science Fund (FWF) under the project P 28789-N32

clique partition is computed only once, and generating and discarding false answers is avoided. We briefly describe the algorithm in this paper.

The second problem is related to the anti-unification algorithm. Computing all maximal clique partitions of the given proximity relation at the beginning and then switching to the standard anti-unification would be an overkill, because many non-minimal answers will be computed, or the same answers might be returned many times. Therefore, it is more reasonable to incorporate the clique partitioning procedure into the anti-unification algorithm and perform partitions only on demand. The algorithm described in this paper works in this way.

In related work, Aït-Kaci and Pasi [1] studied anti-unification with respect to similarity relations, which differ from proximity relations by being transitive and, in this way, are more specific. On the other hand, they allow arity mismatch between symbols, which we do not consider in this paper. Unification with proximity relations has been studied in [6], from which we take the basic definitions.

This work is still in progress. We are currently in the process of a detailed investigation of the properties of presented algorithms, and implementing anti-unification. The latter, after implementation, will be included into our online open-source library of unification and anti-unification algorithms [3].

2 Terms, Substitutions, Proximity Relations

We consider first-order terms defined as usual: $t := x \mid f(t_1, \dots, t_n)$, where x is a variable and f is an n-ary function symbol with $n \geq 0$. We use the letters f, g, h, a, b, c, d and e for function symbols, x, y, z, u, v and w for variables, and s, t, l, and r for terms.

For a term t, its *set of positions* $pos(t)$ is a set of sequences of positive integers defined as follows: If t is a variable, then $pos(t) = \{\epsilon\}$, where ϵ is the empty sequence; If $t = f(s_1, \dots, s_n)$, then $pos(t) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{i.p \mid p \in pos(s_i)\}$. By $t[p]$ we denote the symbol in t at position p. The notation $t|_p$ denotes the subterm of t at position p.

A *substitution* is a mapping from variables to terms, which is the identity almost everywhere. We will use the traditional finite set representation of substitutions. The lower case Greek letters are used to denote substitutions, with the exception of the identity substitution for which we write Id. The related notions such as substitution application, term instance, substitution composition, etc. are defined in the usual way, see, e.g. [2].

A binary *fuzzy relation* on a set S is a fuzzy subset on $S \times S$, that is, a mapping from $S \times S$ to the real interval $[0, 1]$. If $\mathcal{R}$ is a fuzzy relation on S and λ is a number $0 \leq \lambda \leq 1$, then the λ-*cut* of $\mathcal{R}$ on S, denoted $\mathcal{R}_\lambda$, is an ordinary (crisp) relation on S defined as $\mathcal{R}_\lambda := \{(x, y) \mid \mathcal{R}(x, y) \geq \lambda\}$.

A fuzzy relation $\mathcal{R}$ on a set S is called a *proximity relation* on S iff it is reflexive, i.e., $\mathcal{R}(x, x) = 1$ for all $x \in S$, and symmetric, i.e., $\mathcal{R}(x, y) = \mathcal{R}(y, x)$ for all $x, y \in S$. A special class of proximity relations are *similarity relations,* which are *transitive* proximity relations: $\mathcal{R}(x, z) \geq \min(\mathcal{R}(x, y), \mathcal{R}(y, z))$ for all $x, y, z \in S$.

A cut value of a proximity relation $\mathcal{R}$ on S is a number λ such that $\mathcal{R}(x, y) = \lambda$ for some $x, y \in S$. The set of cut values of $\mathcal{R}$ are called approximation levels of $\mathcal{R}$.

Given a proximity relation $\mathcal{R}$ on a set S and $\lambda \in [0, 1]$, a *proximity block of level* λ (or, shortly, a λ-*block*) is a subset B of S such that the restriction of $\mathcal{R}_\lambda$ to B is a total relation, and B is maximal with this property.

Below we consider proximity relations defined on the set of variables and function symbols and assume that no variable is close to any function symbol and to any other variable. Hence, variables always belong to singleton blocks of level 1. Also, function symbols of different arities are not close to each other, i.e., each block consists of symbols of the same arity.

The notion of proximity is defined for terms. The intuition behind it, according to [6], is based on the following idea: two terms t_1 and t_2 are λ-approximate when they have the same set of positions;

their symbols, in their corresponding positions, belong to the same λ-block; and a certain symbol is always assigned to the same λ-block (throughout a computation). The following definition formalizes this intuition:

Definition 1. Given a proximity relation $\mathcal{R}$ on $\mathcal{F}$ and $\lambda \in [0,1]$, two terms t and s are λ-*approximate* (or λ-*close*) with respect to $\mathcal{R}$, written $t \approx_{\mathcal{R},\lambda} s$, if the following conditions hold:

1. $pos(t) = pos(s)$, i.e, the terms have exactly the same positions, hence, the same structure.
2. For all positions $p \in pos(t)$, $t[p]$ and $s[p]$ belong to the same λ-block of the relation $\mathcal{R}$.
3. For all positions $p, p' \in pos(t)$ with $p \neq p'$,
    (a) If $t[p] = t[p']$, then $s[p]$ and $s[p']$ belong to the same λ-block of $\mathcal{R}$.
    (b) If $s[p] = s[p']$, then $t[p]$ and $t[p']$ belong to the same λ-block of $\mathcal{R}$.

When $\mathcal{R}$ is clear from the context, we will just write $t \approx_\lambda s$. The relation $\preceq_{\mathcal{R},\lambda}$ modifies its classical counterpart, the instantiation quasi-ordering $\preceq$ (see, e.g., [2]) by replacing equality with $\approx_{\mathcal{R},\lambda}$: A term t is more general than s at level λ with respect to $\mathcal{R}$, written $t \preceq_{\mathcal{R},\lambda} s$, iff there exists a substitution φ such that $t\varphi \approx_{\mathcal{R},\lambda} s$.

The strict part of the $\preceq_{\mathcal{R},\lambda}$ relation is denoted by $\prec_{\mathcal{R},\lambda}$. The fact that $\approx_{\mathcal{R},\lambda}$ is not a transitive relation implies that $\preceq_{\mathcal{R},\lambda}$ is not a quasi-ordering.

Given a proximity relation $\mathcal{R}$ and a cut λ, a term r is a common $(\lambda,\mathcal{R})$-generalization of t and s iff it is more general than both t and s at level λ with respect to $\mathcal{R}$. It is a least general common $(\lambda,\mathcal{R})$-generalization ($(\lambda,\mathcal{R})$-lgg) of t and s iff it is a common $(\lambda,\mathcal{R})$-generalization of t and s and there is no term l such that $l \prec_{\mathcal{R},\lambda} r$ holds.

Below we assume that λ is fixed and, hence, consider crisp version of proximity relations. Such a relation can be represented as undirected graph, whose maximal cliques (maximal complete subgraphs) are counterparts to blocks. The goal is to design an algorithm which computes $\mathcal{R}$-lggs for a given pair of terms and the proximity relation $\mathcal{R}$. We do not distinguish between $\mathcal{R}$ and the graph that represents it.

Example 1. If (a,b) and (a,c) both belong to $\mathcal{R}$ but (b,c) does not, then $f(a,a)$ and $f(b,b)$ are close to each other, but $f(a,a)$ and $f(b,c)$ are not. The latter pair has two $\mathcal{R}$-lggs: $f(a,x)$ and $f(x,a)$.

A *clique* in an undirected graph $G = (V,E)$ is a set of vertices $W \subseteq V$ such that for each pair of vertices in W there is an edge in E. A clique is *maximal* if it is not a proper subset of another clique. A *clique partition* of a graph G is a set of its cliques $\{C_1,\dots,C_n\}$ such that $\bigcup_{i=1}^n C_i = V$ and $C_i \cap C_j = \emptyset$ for all $1 \leq i,j \leq n$, $i \neq j$.

Let $S_1 = \{C_1,\dots,C_n\}$ and $S_2 = \{D_1,\dots,D_m\}$ be two sets of cliques of the same graph. We say that S_1 is *subsumed* by S_2, written $S_1 \sqsubseteq S_2$, iff for all $1 \leq i \leq n$ there exists $1 \leq j \leq m$ such that $C_i \subseteq D_j$. If S_1 and S_2 are, in particular, partitions, then we also say that S_1 is a *subpartition* of S_2 if $S_1 \sqsubseteq S_2$. A clique partition of a graph is *maximal* if it is not properly subsumed by any other partition of the graph. A graph may have several maximal clique partitions. We will use them in the anti-unification algorithm in Sect. 3. In Sect. 4 we discuss an algorithm that computes all maximal clique partitions in a graph.

3 The Proximity-Based Anti-Unification Algorithm

Our anti-unification algorithm works on tuples $A; C; S; \mathcal{R}; G$, called configurations. Here A, C, and S are sets of anti-unification triples (AUTs, constructions of the form $x : s \triangleq t$, meaning that x is a variable that generalizes s and t), $\mathcal{R}$ is a crisp version of a proximity relation, and G is a term. The rules transform configurations into configurations. Intuitively, the problem set A contains AUTs that have not been solved yet, the set C contains AUTs of the form $x : a \triangleq b$, where a and b are constants such that $(a,b) \in \mathcal{R}$ and the AUTs are not solved yet. The store S contains the already solved AUTs, $\mathcal{R}$ is the proximity relation which gets more and more refined during computation by identifying symbols that belong to the same clique in some partition of $\mathcal{R}$, and G is the generalization which becomes more and more specific as the algorithm progresses by applying the rules.

Dec: **Decomposition**

$$\{x_1 : f_1(s^1_1,\dots,s^1_{k_1}) \triangleq g_1(t^1_1,\dots,t^1_{k_1}),\dots,x_n : f_n(s^n_1,\dots,s^n_{k_n}) \triangleq g_n(t^n_1,\dots,t^n_{k_n})\};\ C;\ S;\ \mathcal{R};\ G \Longrightarrow$$
$$\{y^j_1 : s^j_1 \triangleq t^j_1,\dots,y^j_{k_j} : s^j_{k_j} \triangleq t^j_{k_j} \mid 1 \le j \le m\};\ C;$$
$$\{x_j : f_j(s^j_1,\dots,s^j_{k_j}) \triangleq g_j(t^j_1,\dots,t^j_{k_j}) \mid m+1 \le j \le n\} \cup S;\ \mathcal{R}';\ G\vartheta,$$

where (a) $k_i > 0$ and $(f_i, g_i) \in \mathcal{R}$ for all $1 \le i \le n$; (b) there exist a maximal vertex-clique partition P of the subrelation $\mathcal{Q} = \{(f_1,g_1),\dots,(f_n,g_n)\} \subseteq \mathcal{R}$ and the index $1 \le m \le n$ such that for each (f_j,g_j), $1 \le j \le m$, there is a clique $Cl \in P$ with $f_j, g_j \in Cl$, and for no (f_j,g_j), $m+1 \le j \le n$ there is such a clique; (c) $\mathcal{R}'$ is obtained from $\mathcal{R}$ by replacing the subrelation $\mathcal{Q}$ by its partition P; (d) $\vartheta = \{x_j \mapsto f_j(y^j_1,\dots,y^j_{k_j}) \mid 1 \le j \le m\}$.

Sol: **Solve**

$$\{x : f(\tilde{s}) \triangleq g(\tilde{t})\} \uplus A;\ C;\ S;\ \mathcal{R};\ G \Longrightarrow A;\ C; \{x : f(\tilde{s}) \triangleq g(\tilde{t})\} \cup S;\ \mathcal{R};\ G, \qquad \text{if } (f,g) \notin \mathcal{R}.$$

Post: **Postpone**

$$\{x : a \triangleq b\} \uplus A;\ C;\ S;\ \mathcal{R};\ G \Longrightarrow A;\ \{x : a \triangleq b\} \cup C; S;\ \mathcal{R};\ G, \qquad \text{if } (a,b) \in \mathcal{R}.$$

Gen-Con: **Generalize Constants**

$$\varnothing; \{x_1 : a_1 \triangleq b_1,\dots,x_n : a_n \triangleq b_n\};\ S;\ \mathcal{R};\ G \Longrightarrow \varnothing;\ \varnothing;\ \{x_j : a_j \triangleq b_j \mid m+1 \le j \le n\} \cup S;\ \mathcal{R}';\ G\vartheta,$$

where (a) $(a_i, b_i) \in \mathcal{R}$ for all $1 \le i \le n$; (b) there exist a maximal vertex-clique partition P of the subrelation $\mathcal{Q} = \{(a_1,b_1),\dots,(a_n,b_n)\} \subseteq \mathcal{R}$ and the index $1 \le m \le n$ such that for each (a_j,b_j), $1 \le j \le m$ there is a clique $Cl \in P$ with $a_j, b_j \in Cl$, and for no (a_j,b_j), $m+1 \le j \le n$ there is such a clique; (c) $\mathcal{R}'$ is obtained from $\mathcal{R}$ by replacing $\mathcal{Q}$ by its partition P; (d) $\vartheta = \{x_i \mapsto a_i \mid 1 \le i \le m\}$.

To anti-unify two terms s and t with respect to the proximity relation $\mathcal{R}$, we create the initial tuple $\{x : s \triangleq t\}; \varnothing; \varnothing; \mathcal{R}; x$ and apply the rules in all ways as long as possible. In the search space, branching is caused by all possible maximal clique partitions in Dec and Gen-Con. Generalizations in successful branches form the computed result. We call this algorithm $\mathsf{PR}\text{-}\mathsf{AU}_{\mathsf{lin}}$. The subscript lin indicates that it computes linear generalizations (i.e., those in which each generalization variable appears at most once).

Theorem 1. $\mathsf{PR}\text{-}\mathsf{AU}_{\mathsf{lin}}$ terminates and computes a minimal complete set of linear generalizations.

Proof sketch. Termination follows form the fact that Gen-Con can be applied only once and the other rules strictly reduce the number of symbols in A. In computed answers, no generalization variable appears more than once, because there is no rule that would merge them. Hence, computed generalizations are linear. They are also lggs among linear generalizations, because (a) the algorithm decomposes the terms as much as possible, and (b) it maximizes the number of nonvariable subterms appearing in generalizations, which is done with the help of clique partitions of subrelations (not of the entire relation!) at each decomposition and constant generalization steps. All linear lggs are computed, because branching at Dec and Gen-Con rules explores all maximal clique partitions. □

Example 2. For terms $f(g_1(g_2(a)), g_2(a), a)$ and $f(g_2(g_3(b)), g_3(c), b)$ and the relation $\mathcal{R}$ given in the form of maximal clique set (not a partition) $\{\{f\}, \{g_1, g_2\}, \{g_2, g_3\}, \{a, b\}, \{b, c\}\}$, the algorithm $\mathsf{PR\text{-}AU_{lin}}$ returns two $\mathcal{R}$-lggs: $f(g_1(z_1), y_2, a)$ and $f(y_1, g(y_2), a)$ and misses the nonlinear one $f(g_1(y_2), y_2, y_3)$. We illustrate now how the algorithm works:

$$\begin{aligned}
&\{x : f(g_1(g_2(a)), g_2(a), a) \triangleq f(g_2(g_3(b)), g_3(c), b)\};\ \varnothing;\ \varnothing;\ \mathcal{R};\ x \Longrightarrow_{\mathsf{Dec}}\\
&\{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)),\ y_2 : g_2(a) \triangleq g_3(c),\ y_3 : a \triangleq b\};\ \varnothing;\ \varnothing;\ \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow_{\mathsf{Post}}\\
&\{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)),\ y_2 : g_2(a) \triangleq g_3(c)\};\ \{y_3 : a \triangleq b\};\ \varnothing;\ \mathcal{R};\ f(y_1, y_2, y_3).
\end{aligned}$$

At this stage, the subrelation $\{(g_1, g_2), (g_2, g_3)\}$ of $\mathcal{R}$ can be partitioned in two ways, which gives two new relations $\mathcal{R}_1 = \{\{f\}, \{g_1, g_2\}, \{g_3\}, \{a, b\}, \{b, c\}\}$ and $\mathcal{R}_2 = \{\{f\}, \{g_1\}, \{g_2, g_3\}, \{a, b\}, \{b, c\}\}$. Therefore, we can use the Dec rule and proceed in two different ways:

Alternative 1. Proceeding by $\mathcal{R}_1$.

$$\begin{aligned}
&\{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)),\ y_2 : g_2(a) \triangleq g_3(c)\};\ \{y_3 : a \triangleq b\};\ \varnothing;\ \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow_{\mathsf{Dec}}\\
&\{z_1 : g_2(a) \triangleq g_3(b)\};\ \{y_3 : a \triangleq b\};\ \{y_2 : g_2(a) \triangleq g_3(c)\};\ \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Sol}}\\
&\varnothing;\ \{y_3 : a \triangleq b\};\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b)\};\ \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\varnothing;\ \varnothing;\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b)\};\ \mathcal{R}_{11};\ f(g_1(z_1), y_2, a).
\end{aligned}$$

where $\mathcal{R}_{11} = \{\{f\}, \{g_1, g_2\}, \{g_3\}, \{a, b\}, \{c\}\}$. Note that if we required in the condition of the $\mathsf{Gen\text{-}Con}$ rule to partition the relation itself (instead of its subrelation), we would get also $\mathcal{R}_{12} = \{\{f\}, \{g_1, g_2\}, \{g_3\}, \{a\}, \{b, c\}\}$, which would lead to another successful branch

$$\begin{aligned}
&\varnothing;\ \{y_3 : a \triangleq b\};\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b)\};\ \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\varnothing;\ \varnothing;\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b), y_3 : a \triangleq b\};\ \mathcal{R}_{12};\ f(g_1(z_1), y_2, y_3).
\end{aligned}$$

However, the computed generalization is not an lgg, since it is more general than the previous one.

Alternative 2. Proceeding by $\mathcal{R}_2$.

$$\begin{aligned}
&\{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)),\ y_2 : g_2(a) \triangleq g_3(c)\};\ \{y_3 : a \triangleq b\};\ \varnothing;\ \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow_{\mathsf{Dec}}\\
&\{z_2 : a \triangleq c\};\ \{y_3 : a \triangleq b\};\ \{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b))\};\ \mathcal{R}_2;\ f(y_1, g(z_2), y_3) \Longrightarrow_{\mathsf{Sol}}\\
&\varnothing;\ \{y_3 : a \triangleq b\};\ \{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)), z_2 : a \triangleq c\};\ \mathcal{R}_2;\ f(y_1, g(y_2), y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\varnothing;\ \varnothing;\ \{y_2 : g_2(a) \triangleq g_3(c)\};\ \mathcal{R}_{21};\ f(y_1, g(y_2), a),
\end{aligned}$$

where $\mathcal{R}_{21} = \{\{f\}, \{g_1\}, \{g_2, g_3\}, \{a, b\}, \{c\}\}$. Again, if we were allowed to partition the whole $\mathcal{R}_2$ in $\mathsf{Gen\text{-}Con}$, we would get another partition $\mathcal{R}_{22} = \{\{f\}, \{g_1\}, \{g_2, g_3\}, \{a\}, \{b, c\}\}$, which would give the following successful branch:

$$\begin{aligned}
&\varnothing;\ \{y_3 : a \triangleq b\};\ \{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)), z_2 : a \triangleq c\};\ \mathcal{R}_2;\ f(y_1, g(y_2), y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\varnothing;\ \varnothing;\ \{y_2 : g_2(a) \triangleq g_3(c), y_3 : a \triangleq b\};\ \mathcal{R}_{22};\ f(y_1, g(y_2), y_3).
\end{aligned}$$

However, the solution obtained in this branch is more general than the previous one.

As the next step, we extend the algorithm to add a rule for merging variables. It uses a partial function *refine*$(\{s_1 \approx t_1, \ldots, s_n \approx t_n\}, \mathcal{R})$, which is supposed to refine the given relation $\mathcal{R}$ into a new relation $\mathcal{R}'$ so that $s_i \approx_{\mathcal{R}'} t_i$, $1 \le i \le n$, if such a refinement of $\mathcal{R}$ exists.

Mer: **Merge**

$A;\ C;\ \{x : s_1 \triangleq t_1,\, y : s_2 \triangleq t_2\} \uplus S;\ \mathcal{R};\ G \Longrightarrow A;\ C;\ \{x : s_1 \triangleq t_1\} \cup S;\ \mathcal{R}';\ G\{y \mapsto x\}$,
where $\mathcal{R}' = \mathit{refine}(\{s_1 \approx s_2, t_1 \approx t_2\}, \mathcal{R})$.

The obtained algorithm is denoted by PR-AU. The function *refine* is defined as follows:

$$\mathit{refine}(\varnothing, \mathcal{R}) = \mathcal{R}. \qquad \mathit{refine}(\{t \approx t\} \uplus E, \mathcal{R}) = \mathit{refine}(E, \mathcal{R}).$$
$$\mathit{refine}(\{f(s_1, \dots, s_n) \approx g(t_1, \dots, t_n)\} \uplus E, \mathcal{R}) = \mathit{refine}(\{s_1 \approx t_1, \dots, s_n \approx t_n\} \cup E, \mathcal{R}'),$$

if $(f, g) \in \mathcal{R}$ and $\mathcal{R}' = \mathcal{R} \setminus (S_1 \cup S_2)$, where $S_1 = \{(f, h) \mid (g, h) \notin \mathcal{R}\} \cup \{(h, f) \mid (g, h) \notin \mathcal{R}\}$ and $S_2 = \{(g, h) \mid (f, h) \notin \mathcal{R}\} \cup \{(h, g) \mid (f, h) \notin \mathcal{R}\}$. Otherwise, *refine* is not defined.

From the specification of *refine* it follows that Mer is correct. It is also an alternative to the rules above, meaning that it would introduce additional branching, because of which PR-AU might recompute the same solution on different branches. For instance, in Example 2, when we proceed by $\mathcal{R}_1$, we get two branches that compute the same result (recall that $\mathcal{R}_{11} = \{\{f\}, \{g_1, g_2\}, \{g_3\}, \{a, b\}, \{c\}\}$):

$$\begin{aligned}
&\{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)),\ y_2 : g_2(a) \triangleq g_3(c)\};\ \{y_3 : a \triangleq b\};\ \varnothing;\ \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow_{\mathsf{Dec}}\\
&\{z_1 : g_2(a) \triangleq g_3(b)\};\ \{y_3 : a \triangleq b\};\ \{y_2 : g_2(a) \triangleq g_3(c)\};\ \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Sol}}\\
&\varnothing;\ \{y_3 : a \triangleq b\};\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b)\};\ \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\varnothing;\ \varnothing;\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b)\};\ \mathcal{R}_{11};\ f(g_1(z_1), y_2, a) \Longrightarrow_{\mathsf{Mer}}\\
&\varnothing;\ \varnothing;\ \{y_2 : g_2(a) \triangleq g_3(c)\};\ \mathcal{R}_{11};\ f(g_1(y_2), y_2, a)
\end{aligned}$$

and

$$\begin{aligned}
&\{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)),\ y_2 : g_2(a) \triangleq g_3(c)\};\ \{y_3 : a \triangleq b\};\ \varnothing;\ \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow_{\mathsf{Dec}}\\
&\{z_1 : g_2(a) \triangleq g_3(b)\};\ \{y_3 : a \triangleq b\};\ \{y_2 : g_2(a) \triangleq g_3(c)\};\ \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Sol}}\\
&\varnothing;\ \{y_3 : a \triangleq b\};\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b)\};\ \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Mer}}\\
&\varnothing;\ \{y_3 : a \triangleq b\};\ \{y_2 : g_2(a) \triangleq g_3(c)\};\ \mathcal{R}_1;\ f(g_1(y_2), y_2, y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\varnothing;\ \varnothing;\ \{y_2 : g_2(a) \triangleq g_3(c)\};\ \mathcal{R}_{11};\ f(g_1(y_2), y_2, a).
\end{aligned}$$

However, we can not postpone Mer till the end, after A and C get empty (as it is usually done in anti-unification algorithms), because in this case we will miss solutions, as the example below shows.

Example 3. Let $\mathcal{R} = \{\{f\}, \{h_1\}, \{h_2\}, \{a\}, \{g_0, g_1\}, \{g_0, g_2\}\}$, $s = f(g_0(a), h_1(g_0(a)), h_1(g_0(a)))$, and $t = f(g_1(a), h_2(g_2(a)), h_2(g_0(a)))$. Then PR-AU computes two solutions: $f(g_0(a), y_2, y_3)$ and $f(y_1, y_2, y_2)$. The first one is obtained by applying Sol before Mer, and the second one in the other way around. However, if Mer is applied only at the very end, then the first solution is not computed.

Merging variables can significantly increase the size of the computed set of generalizations:

Example 4. Let the arity of f be $n + 2$, $\mathcal{R} = \{\{f\}, \{h_1\}, \{h_2\}, \{a\}, \{g_0, g_1\}, \dots, \{g_0, g_n\}\}$ and

$$\begin{aligned}
s &= f(g_0(a),\ h_1(g_0(a)), \dots,\ h_1(g_0(a)),\ h_1(g_0(a))),\\
t &= f(g_1(a),\ h_2(g_2(a)), \dots,\ h_2(g_n(a)),\ h_2(g_0(a))).
\end{aligned}$$

PR-AU$_{\mathsf{lin}}$ computes only one generalization: $f(g_0(a), y_2, \dots, y_n, y_{n+1}$. With PR-AU, we have, in addition, $n - 1$ other generalizations: $f(y_1, y_2, \dots, y_n, y_2), \dots, f(y_1, y_2, \dots, y_n, y_n)$.

Theorem 2. PR-AU computes a minimal complete set of generalizations.

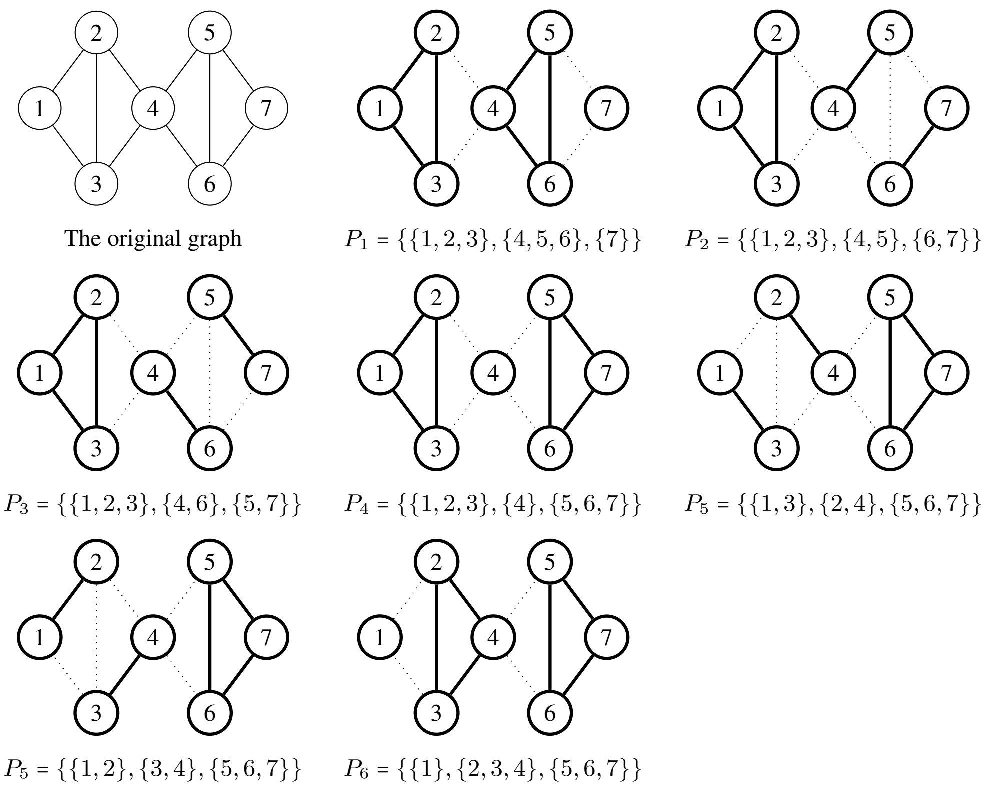
The original graph
$P_1 = \{\{1,2,3\},\{4,5,6\},\{7\}\}$
$P_2 = \{\{1,2,3\},\{4,5\},\{6,7\}\}$
$P_3 = \{\{1,2,3\},\{4,6\},\{5,7\}\}$
$P_4 = \{\{1,2,3\},\{4\},\{5,6,7\}\}$
$P_5 = \{\{1,3\},\{2,4\},\{5,6,7\}\}$
$P_5 = \{\{1,2\},\{3,4\},\{5,6,7\}\}$
$P_6 = \{\{1\},\{2,3,4\},\{5,6,7\}\}$

Figure 1: All maximal clique partitions of a graph.

4 Computing All Maximal Clique Partitions in a Graph

The anti-unification algorithm in the previous section relies on the computation of all maximal clique partitions in an undirected graph. We describe the corresponding algorithm here.

First, we compute all maximal cliques of the given graph and give each of them a name. All maximal cliques can be computed, e.g., by Bron-Kerbosch algorithm [5]. For the graph in Fig. 1, there are four of them: $C_1 = \{1,2,3\}, C_2 = \{2,3,4\}, C_3 = \{4,5,6\}, C_4 = \{5,6,7\}$. These cliques will get revised during computation by removing elements from them. At the end, we report those which are not empty.

After computing the initial cliques, we collect all shared vertices and indicate among which cliques they are shared. In the graph in Fig. 1, the shared vertices are 2, 3, 4, 5, and 6. We have $2 \in C_1 \cap C_2$, $3 \in C_1 \cap C_2$, $4 \in C_2 \cap C_3$, $5 \in C_3 \cap C_4$, and $6 \in C_3 \cap C_4$.

Our goal is to compute each solution exactly once. At the end, it can happen that some cliques consist of shared vertices only. However, such cliques can have any of the names of the original cliques they originate from. For instance, the node 4 alone can form a clique either as C_2 or C_3, giving two identical partitions which differ only by the clique names:

$$\begin{aligned} &C_1 = \{1,2,3\}, \quad C_2 = \{4\}, \quad C_3 = \varnothing, \qquad C_4 = \{5,6,7\}, \\ &C_1 = \{1,2,3\}, \quad C_2 = \varnothing, \qquad C_3 = \{4\}, \quad C_4 = \{5,6,7\}. \end{aligned}$$

We want to avoid such duplicates. Therefore, for such alternatives we choose one single clique to which they belong in this configuration, and forbid the others. For the example graph in Fig. 1, we can allow the vertices 2 and/or 3 to form a clique as C_2, the vertex 4 to form a clique as C_3, and the vertices 5 and/or 6 to form a clique as C_4. (Note that allowing does not necessarily mean that we will get result cliques of that form.) Thus, the forbidden configurations are $C_1 \neq \{2\}, C_1 \neq \{3\}, C_1 \neq \{2,3\}, C_2 \neq \{4\}, C_3 \neq \{5\}, C_3 \neq \{6\}, C_3 \neq \{5,6\}$.

Starting from the initial set of cliques, our algorithm All-Maximal-Clique-Partitions performs the following steps:

1. Compute the set of shared vertices and the forbidden configurations.

2. If the set of shared vertices is empty, return the current set of cliques and stop.

3. Select a shared vertex and nondeterministically assign it to one of the cliques it belongs to. Remove the vertex from the other cliques and from the set of shared vertices.

4. For each pair of cliques C_i, C_j, where $C_i \subseteq C_j$, make C_i empty and adjust the set of shared elements. In addition, if C_i was the chosen clique for the shared elements, remove those elements from the forbidden list of C_j.

5. If the union of two nonempty cliques is a subset of an original clique, or if a forbidden configuration arises, stop the development of this branch with failure. Otherwise go to step 2.

Checking for the subset relations is needed to avoid computing cliques which are not maximal. For instance, the partition $C_1 = \{1,2\}$, $C_2 = \{3\}$, $C_3 = \{4\}$, $C_4 = \{5,6,7\}$ should be rejected because $\{1,2\} \cup \{3\}$ is a subset of the original C_1 clique. Step 5 helps to detect such situations early.

The partitions shown in Fig. 1 correspond to the following final values of cliques, computed by the All-Maximal-Clique-Partitions algorithm:

$$\begin{array}{lllll}
P_1: & C_1 = \{1,2,3\}, & C_2 = \varnothing, & C_3 = \{4,5,6\}, & C_4 = \{7\} \\
P_2: & C_1 = \{1,2,3\}, & C_2 = \varnothing, & C_3 = \{4,5\}, & C_4 = \{6,7\} \\
P_3: & C_1 = \{1,2,3\}, & C_2 = \varnothing, & C_3 = \{4,6\}, & C_4 = \{5,7\} \\
P_4: & C_1 = \{1,2,3\}, & C_2 = \varnothing, & C_3 = \{4\}, & C_4 = \{5,6,7\} \\
P_5: & C_1 = \{1,2\}, & C_2 = \{2,4\}, & C_3 = \varnothing, & C_4 = \{5,6,7\} \\
P_6: & C_1 = \{1,2\}, & C_2 = \{3,4\}, & C_3 = \varnothing, & C_4 = \{5,6,7\} \\
P_7: & C_1 = \{1\}, & C_2 = \{2,3,4\}, & C_3 = \varnothing, & C_4 = \{5,6,7\}.
\end{array}$$

5 Conclusion

We designed and implemented an algorithm to compute all maximal vertex-clique partitions in an undirected graph and used it in the computation of proximity-based least general generalizations. The next steps are to study the properties of both algorithms in detail and to implement the one for anti-unification. A more remote goal is to study applicability of proximity-based anti-unification in program analysis and clone detection.

References

[1] H. Aït-Kaci and G. Pasi. Lattice operations on terms with fuzzy signatures. *CoRR*, abs/1709.00964, 2017.

[2] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[3] A. Baumgartner and T. Kutsia. A library of anti-unification algorithms. In E. Fermé and J. Leite, editors, *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014*, volume 8761 of *Lecture Notes in Computer Science*, pages 543–557. Springer, 2014.

[4] J. Bhasker and T. Samad. The clique-partitioning problem. *Computers and Mathematics with Applications*, 22(6):1–11, 1991.

[5] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[6] P. J. Iranzo and C. Rubio-Manzano. Proximity-based unification theory. *Fuzzy Sets and Systems*, 262:21–43, 2015.

[7] C.-J. Tseng. *Automated synthesis of data paths in digital systems*. PhD thesis, Dept. of Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburg, PA, 1984.

About the unification type of topological logics over Euclidean spaces

Çiğdem Gencer

Faculty of Arts and Sciences, Istanbul Aydın University

1 Introduction

Topological logics (TLs) are formalisms for reasoning about topological relations (contact, connectedness, etc) between regions [5, 13, 14, 15]. Their languages are obtained from the language of Boolean algebras by the addition of predicates representing these relations. Interpreted over mereotopological spaces, the formulas of these languages describe configurations of concrete objects. Recently, the validity problem determined by different classes of mereotopological spaces has been intensively investigated [8, 9, 10].
In this paper, we introduce a new inference problem for TLs, the unifiability problem, which extends the validity problem by allowing one to replace variables by terms before testing for validity. For example, within the context of the mereotopology of all regular closed polygons of the real plane, the formula $C(p, q) \to x \not\equiv 0 \wedge x \leq p \cup q$, read "if p is in contact with q then x is nonempty and x is contained in $p \cup q$", is not valid but can be made valid after replacing x either by $p \cup (q \cap x)$, or by $q \cup (p \cap x)$.
There is a wide variety of situations where unifiability problems arise. Suppose the formula $\varphi(p_1, \ldots, p_m)$ describes a given geographic configuration of constant regions $p_1, \ldots, p_m$ and the formula $\psi(x_1, \ldots, x_n)$ represents a desirable geographic property of variable regions $x_1, \ldots, x_n$. It may happen that $\varphi(p_1, \ldots, p_m) \to \psi(x_1, \ldots, x_n)$ is not valid in the considered geographic environment. Hence, one may ask whether there are n-tuples $(a_1, \ldots, a_n)$ of terms such that $\varphi(p_1, \ldots, p_m) \to \psi(a_1, \ldots, a_n)$ is valid in this environment. Moreover, one may be interested to obtain, if possible, the most general n-tuples $(a_1, \ldots, a_n)$ of terms such that $\varphi(p_1, \ldots, p_m) \to \psi(a_1, \ldots, a_n)$ is valid.
In this paper, we adapt to the problem of unifiability with constants in TLs (interpreted over the mereotopology of all regular closed polygons of the real plane) the line of reasoning developed by Balbiani and Gencer [4] within the simpler context of the problem of unifiability without constants in Boolean Region Connection Calculus (interpreted over Kripke models). This adaptation is far from obvious. Our main result is that, within the context of the mereotopology of all regular closed polygons of the real plane, unifiable formulas always have finite complete sets of unifiers.

2 Syntax

Terms Let CON be a countable set of *constants* (p, q, etc) and VAR be a countable set of *variables* (x, y, etc). Let $(p_1, p_2, \ldots)$ be an enumeration of CON without repetitions and $(x_1, x_2, \ldots)$ be an enumeration of VAR without repetitions. An *atom* is either a constant, or a variable. The *Boolean terms* (a, b, etc) are defined by the rule

- $a, b ::= p \mid x \mid 0 \mid a^\star \mid (a \cup b)$.

The other Boolean constructs for terms (for instance, 1 and $\cap$) are defined as usual. Reading terms as regions, the constructs 0, * and $\cup$ should be regarded as the empty region, the complement operation and the union operation. As a result, the constructs 1 and $\cap$ should be regarded as the full region and the intersection operation. For all $m, n \geq 0$, let $TER_{m,n}$ be the set of all terms whose constants form a subset of $\{p_1, \ldots, p_m\}$ and whose variables form a subset of $\{x_1, \ldots, x_n\}$. Let TER be the set of all terms.

Formulas The *formulas* (φ, ψ, etc) are defined by the rule

- $\varphi, \psi ::= C(a, b) \mid a \equiv b \mid \bot \mid \neg\varphi \mid (\varphi \vee \psi)$.

Here, a and b are terms whereas C is the predicate of contact and $\equiv$ is the predicate of equality. We use the notation $a \leq b$ for $a \cup b \equiv b$. For $C(a, b)$ and $a \equiv b$, we propose the readings "a is in contact with b" and "a is equal to b". As a result, for $a \leq b$, we propose the reading "a is contained in b". The other connectives for formulas (for instance, $\top$ and $\wedge$) are defined as usual. A formula is *equational* iff $\equiv$ is the only predicate possibly occurring in it. Let FOR be the set of all formulas and FOR^{eq} be the set of all equational formulas. Note that FOR and FOR^{eq} are denoted $\mathcal{C}$ and $\mathcal{B}$ in [8, 9, 10].

3 Semantics

Topological spaces A *topological space* is a structure of the form (X, τ) where X is a nonempty set and τ is a set of subsets of X such that the following conditions hold:

- $\emptyset$ is in τ,
- X is in τ,
- if $\{A_i : i \in I\}$ is a finite subset of τ then $\bigcap\{A_i : i \in I\}$ is in τ,
- if $\{A_i : i \in I\}$ is a subset of τ then $\bigcup\{A_i : i \in I\}$ is in τ.

The subsets of X in τ are called *open sets* whereas their complements are called *closed sets.* In this paper, we will interest with the topological space $(\mathbb{R}^2, \tau_{\mathbb{R}^2})$, i.e. the real plane $\mathbb{R}^2$ together with its ordinary topology $\tau_{\mathbb{R}^2}$.

Regular closed subsets Let (X, τ) be a topological space. Let Int_τ and Cl_τ denote the *interior operator* and the *closure operator* in (X, τ). A subset A of X is *regular closed* iff $Cl_\tau(Int_\tau(A)) = A$. Regular closed subsets of X will also be called *regions.* It is well-known that the set $RC(X, \tau)$ of all regular closed subsets of X forms a Boolean algebra $(RC(X, \tau), 0_X, \star_X, \cup_X)$ where for all $A, B \in RC(X, \tau)$:

- $0_X = \emptyset$,
- $A^{\star_X} = Cl_\tau(X \setminus A)$,
- $A \cup_X B = A \cup B$.

As a result, for all $A, B \in RC(X, \tau)$, $1_X = X$ and $A \cap_X B = Cl_\tau(Int_\tau(A \cap B))$. Since regions are regular closed subsets of X, therefore two regions are *in contact* iff they have a nonempty intersection. For this reason, we define the relation $C^{(X,\tau)}$ on $RC(X, \tau)$ by

- $C^{(X,\tau)}(A, B)$ iff $A \cap B \neq \emptyset$.

The following conditions hold for all $A, B, A', B' \in RC(X, \tau)$:

- if $C^{(X,\tau)}(A,B)$ and $A \subseteq A'$ then $C^{(X,\tau)}(A',B)$,
- if $C^{(X,\tau)}(A,B)$ and $B \subseteq B'$ then $C^{(X,\tau)}(A,B')$,
- if $C^{(X,\tau)}(A \cup A',B)$ then either $C^{(X,\tau)}(A,B)$, or $C^{(X,\tau)}(A',B)$,
- if $C^{(X,\tau)}(A,B \cup B')$ then either $C^{(X,\tau)}(A,B)$, or $C^{(X,\tau)}(A,B')$,
- if $C^{(X,\tau)}(A,B)$ then $A \neq \emptyset$ and $B \neq \emptyset$,
- if $A \neq \emptyset$ then $C^{(X,\tau)}(A,A)$,
- if $C^{(X,\tau)}(A,B)$ then $C^{(X,\tau)}(B,A)$.

Mereotopologies Let (X,τ) be a topological space. A *mereotopology* over (X,τ) is a Boolean subalgebra M of $RC(X,\tau)$ such that for all $P \in X$ and for all $A \in \tau$, if $P \in A$ then there exists $B \in M$ such that $P \in B$ and $B \subseteq A$. A *mereotopological space* over (X,τ) is a structure (X,τ,M) where M is a mereotopology over (X,τ) [12]. Over the topological space $(\mathbb{R}^2, \tau_{\mathbb{R}^2})$, several mereotopologies can be considered. One can consider the mereotopology consisting of the set $RC(\mathbb{R}^2)$ of all regular closed subsets of $\mathbb{R}^2$. Nevertheless, as regions are supposed to be parts of the real plane occupied by concrete objects, it is clear that some of the regular closed subsets of $\mathbb{R}^2$ cannot count as regions. For this reason, one can consider the more concrete mereotopology consisting of the set $RCS(\mathbb{R}^2)$ of all regular closed semi-algebraic subsets of $\mathbb{R}^2$, i.e. those regular closed subsets of $\mathbb{R}^2$ definable by a first-order formula in the language of arithmetic interpreted over $\mathbb{R}$. The main property of this mereotopology is that any of its elements is a finite union of semi-algebraic cells, i.e. semi-algebraic subsets of $\mathbb{R}^2$ homeomorphic to a closed disc. But $RCS(\mathbb{R}^2)$ is not the only candidate for a region-based model of space. A simpler candidate is the mereotopology consisting of the set $RCP(\mathbb{R}^2)$ of all regular closed polygons of $\mathbb{R}^2$, i.e. those regular closed subsets of $\mathbb{R}^2$ definable by a finite union of finite intersections of closed half-planes. Although this mereotopology may seem overly simple, its study from the perspective of the unifiability problem will turn out to be relatively interesting.

Models Let (X,τ,M) be a mereotopological space. A *valuation* on (X,τ,M) is a map associating with every atom a regular closed subset of X in M. Given a valuation $\mathcal{V}$ on (X,τ,M), we define:

- $\bar{\mathcal{V}}(p) = \mathcal{V}(p)$,
- $\bar{\mathcal{V}}(x) = \mathcal{V}(x)$,
- $\bar{\mathcal{V}}(0) = \emptyset$,
- $\bar{\mathcal{V}}(a^\star) = Cl_\tau(X \setminus \bar{\mathcal{V}}(a))$,
- $\bar{\mathcal{V}}(a \cup b) = \bar{\mathcal{V}}(a) \cup \bar{\mathcal{V}}(b)$.

As a result, $\bar{\mathcal{V}}(1) = X$ and $\bar{\mathcal{V}}(a \cap b) = Cl_\tau(Int_\tau(\bar{\mathcal{V}}(a) \cap \bar{\mathcal{V}}(b)))$. Thus, $\mathcal{V}$ interprets every term as a regular closed subset of X in M. A *model* on (X,τ,M) is a structure $\mathcal{M} = (X,\tau,M,\mathcal{V})$ where $\mathcal{V}$ is a valuation on (X,τ,M). The connectives $\bot$, $\neg$ and $\vee$ being classically interpreted, the *satisfiability* of $\varphi \in FOR$ in $\mathcal{M}$ (in symbols $\mathcal{M} \models \varphi$) is defined as follows:

- $\mathcal{M} \models C(a,b)$ iff $C^{(X,\tau)}(\bar{\mathcal{V}}(a), \bar{\mathcal{V}}(b))$,
- $\mathcal{M} \models a \equiv b$ iff $\bar{\mathcal{V}}(a) = \bar{\mathcal{V}}(b)$.

As a result, $\mathcal{M} \models a \leq b$ iff $\bar{\mathcal{V}}(a) \subseteq \bar{\mathcal{V}}(b)$.

Validity Let (X, τ, M) be a mereotopological space. A formula φ is *valid* in (X, τ, M) iff for all valuations $\mathcal{V}$ on (X, τ, M), $(X, \tau, M, \mathcal{V}) \models \varphi$. Let $\mathcal{C}$ be a class of mereotopological spaces. A formula φ is *$\mathcal{C}$-valid* iff for all mereotopological spaces (X, τ, M) in $\mathcal{C}$, φ is valid in (X, τ, M). The *$\mathcal{C}$-validity problem* consists in determining whether a given formula is $\mathcal{C}$-valid. In this paper, we will be interested in the polygon-based mereotopological space $(\mathbb{R}^2, \tau_{\mathbb{R}^2}, RCP(\mathbb{R}^2))$ over $(\mathbb{R}^2, \tau_{\mathbb{R}^2})$. As a result, when we write "valid", we mean "valid in the mereotopological space $(\mathbb{R}^2, \tau_{\mathbb{R}^2}, RCP(\mathbb{R}^2))$".

Proposition 1. *For all $\varphi \in FOR^{eq}$, the following are equivalent:* $(\mathbf{1})$ *φ is valid;* $(\mathbf{2})$ *for all finite Boolean algebras $\mathcal{B}$ and for all valuations $\mathcal{V}_{\mathcal{B}}$ on $\mathcal{B}$, $(\mathcal{B}, \mathcal{V}_{\mathcal{B}}) \models \varphi$;* $(\mathbf{3})$ *for all Boolean algebras $\mathcal{B}$ and for all valuations $\mathcal{V}_{\mathcal{B}}$ on $\mathcal{B}$, $(\mathcal{B}, \mathcal{V}_{\mathcal{B}}) \models \varphi$.*

4 Unification

Substitutions A *substitution* is a function $\sigma : VAR \longrightarrow TER$ which moves at most finitely many variables. The *domain* of a substitution σ (in symbols $dom(\sigma)$) is the set of variables σ moves. Given a substitution σ, let $\bar{\sigma} : TER \cup FOR \longrightarrow TER \cup FOR$ be the endomorphism such that for all variables x, $\bar{\sigma}(x) = \sigma(x)$. The *composition* of the substitutions σ and τ is the substitution $\sigma \circ \tau$ such that for all $x \in VAR$, $(\sigma \circ \tau)(x) = \bar{\tau}(\sigma(x))$. For all $m, n \geq 0$, let $\Sigma_{m,n}$ be the set of all substitutions σ such that $dom(\sigma) \subseteq \{x_1, \ldots, x_n\}$ and for all positive integers $i \leq n$, $\sigma(x_i)$ is in $TER_{m,n}$. A substitution σ is *equivalent* to a substitution τ (in symbols $\sigma \simeq \tau$) iff for all variables x, $\sigma(x) \equiv \tau(x)$ is valid. Obviously, $\simeq$ is reflexive, symmetric and transitive on the set of all substitutions. A substitution σ is *more general* than a substitution τ (in symbols $\sigma \preceq \tau$) iff there exists a substitution υ such that $\sigma \circ \upsilon \simeq \tau$. Obviously, $\preceq$ is reflexive and transitive on the set of all substitutions. Moreover, it contains $\simeq$. A set of substitutions is *small* iff it contains finitely many non-pairwise equivalent substitutions modulo $\simeq$.

Proposition 2. *For all $m, n \geq 0$, $\Sigma_{m,n}$ is small.*

Unifiable formulas A formula φ is *unifiable* iff there exists a substitution σ such that $\bar{\sigma}(\varphi)$ is valid. In that case, we say that σ is a *unifier* of φ. The *unifiability problem* (in symbols $\mathcal{UNIF}$) consists in determining whether a given formula is unifiable [3]. A set of unifiers of $\varphi \in FOR$ is *complete* iff for all unifiers σ of φ, there exists a unifier τ of φ in that set such that $\tau \preceq \sigma$. An important question in unification theory is [6]: when a formula is unifiable, has it a minimal complete set of unifiers? When the answer is "yes", how large is this set?

Unification types A unifiable formula φ is *finitary* iff there exists a finite complete set of unifiers of φ but there exists no with cardinality 1. A unifiable formula φ is *unitary* iff there exists a unifier σ of φ such that for all unifiers τ of φ, $\sigma \preceq \tau$. In that case, we say that σ is a *most general unifier* of φ.

Proposition 3. *For all unifiable $\varphi \in FOR$, the following are equivalent:* $(\mathbf{1})$ *φ is either finitary, or unitary;* $(\mathbf{2})$ *there exists a small set Σ of substitutions such that for all unifiers σ of φ, there exists a unifier τ of φ in Σ such that $\tau \preceq \sigma$.*

Proposition 4. *Let $\varphi \in FOR$, $n \geq 2$ and $\sigma_1, \ldots, \sigma_n$ be substitutions. If the following hold then φ is finitary:* $(\mathbf{1})$ *for all positive integers $i \leq n$, σ_i is a unifier of φ;* $(\mathbf{2})$ *for all positive integers $i, j \leq n$, if $i \neq j$ then $\sigma_i \not\preceq \sigma_j$;* $(\mathbf{3})$ *$\sigma_1, \ldots, \sigma_n$ form a complete set of unifiers of φ.*

For all a in TER, when we write "a^0", we mean "$a^\star$" and when we write "a^1", we mean "a".

5 Examples

For some formulas, if they are unifiable then they are finitary. Luckily, in many cases, this can be easily proved. For example, let us consider the formula

$$\varphi_{01} \quad := \quad x \equiv 0 \vee x \equiv 1.$$

Let σ_0 and σ_1 be the substitutions such that $\sigma_0(x) = 0$, $\sigma_1(x) = 1$ and for all variables y, if $x \neq y$ then $\sigma_0(y) = y$ and $\sigma_1(y) = y$.

Proposition 5.

- *σ_0 and σ_1 are unifiers of φ_{01},*
- *neither $\sigma_0 \preceq \sigma_1$, nor $\sigma_1 \preceq \sigma_0$,*
- *σ_0 and σ_1 form a complete set of unifiers of φ_{01},*
- *φ_{01} is finitary.*

Unfortunately, there are unifiable formulas for which the proof that they are finitary can be more involved. For example, let us consider the formula

$$\varphi_{pq} \quad := \quad C(p, q) \to x \not\equiv 0 \wedge x \leq p \cup q.$$

Let σ_p and σ_q be the substitutions such that $\sigma_p(x) = p \cup (q \cap x)$, $\sigma_q(x) = q \cup (p \cap x)$ and for all variables y, if $x \neq y$ then $\sigma_p(y) = y$ and $\sigma_q(y) = y$.

Proposition 6.

- *σ_p and σ_q are unifiers of φ_{pq},*
- *if $p \neq q$ then neither $\sigma_p \preceq \sigma_q$, nor $\sigma_q \preceq \sigma_p$,*
- *if $p \neq q$ then σ_p and σ_q form a complete set of unifiers of φ_{pq},*
- *if $p \neq q$ then φ_{pq} is finitary.*

6 Monomials

The purpose of this section is to introduce definitions and properties about terms. These definitions and properties are purely Boolean. They will be used later in Sections 7 and 8. From now on, when we write "**CPL**", we mean "Classical Propositional Logic". Let $k, m, n \geq 0$ be such that $n \leq k$. An *m-vector* is a map $\vec{s}$ associating with every positive integer $i \leq m$ an element $\vec{s}(i)$ of $\{0, 1\}$. A *(k, m, n)-correspondence* is a map f associating with every m-vector $\vec{s}$ a surjective function $f_{\vec{s}} : \{0, 1\}^k \longrightarrow \{0, 1\}^n$. An *$n$-monomial* is a term of the form $x_1^{\beta_1} \cap \ldots \cap x_n^{\beta_n}$ where $(\beta_1, \ldots, \beta_n) \in \{0, 1\}^n$. For all m-vectors $\vec{s}$, considering a term a in $TER_{m,n}$ as a formula in **CPL**, let $mon_{\vec{s}}(n, a)$ be the set of all n-monomials $x_1^{\beta_1} \cap \ldots \cap x_n^{\beta_n}$ such that a is a tautological consequence of $p_1^{\vec{s}(1)} \cap \ldots \cap p_m^{\vec{s}(m)} \cap x_1^{\beta_1} \cap \ldots \cap x_n^{\beta_n}$.

Proposition 7. *Let $a \in TER_{m,n}$. Considered as formulas in **CPL**, the terms a and $\bigcup\{p_1^{\vec{s}(1)} \cap \ldots \cap p_m^{\vec{s}(m)} \cap x_1^{\alpha_1} \cap \ldots \cap x_n^{\alpha_n} : \vec{s}$ is an m-vector and $x_1^{\alpha_1} \cap \ldots \cap x_n^{\alpha_n} \in mon_{\vec{s}}(n, a)\}$ are equivalent.*

For all positive integers $i \leq n$, let $\pi_i : \{0, 1\}^n \longrightarrow \{0, 1\}$ be the function such that for all $(\beta_1, \ldots, \beta_n) \in \{0, 1\}^n$, $\pi_i(\beta_1, \ldots, \beta_n) = \beta_i$. Let f be a (k, m, n)-correspondence. For all m-vectors $\vec{s}$, for all $(\beta_1, \ldots, \beta_n) \in \{0, 1\}^n$ and for all positive integers $i \leq n$, let $f_{\vec{s}}^{-1}(\beta_1, \ldots, \beta_n)$ be the set of all $(\alpha_1, \ldots, \alpha_k) \in \{0, 1\}^k$ such that $f_{\vec{s}}(\alpha_1, \ldots, \alpha_k) = (\beta_1, \ldots, \beta_n)$, $\Delta_{\vec{s},i}$ be the set of all $(\alpha_1, \ldots, \alpha_k) \in \{0, 1\}^k$ such that $\pi_i(f_{\vec{s}}(\alpha_1, \ldots, \alpha_k)) = 1$ and $c_{\vec{s},i}$ be the term $\bigcup\{x_1^{\alpha_1} \cap \ldots \cap x_k^{\alpha_k} : (\alpha_1, \ldots, \alpha_k) \in \Delta_{\vec{s},i}\}$. Remark that $\Delta_{\vec{s},i}$ and $c_{\vec{s},i}$ depend on f — more precisely, on $f_{\vec{s}}$ — too.

Proposition 8. *For all m-vectors $\vec{s}$ and for all $(\beta_1,\ldots,\beta_n)\in\{0,1\}^n$, considered as formulas in* **CPL**, *the terms $\bigcup\{x_1^{\alpha_1}\cap\ldots\cap x_k^{\alpha_k}:\ (\alpha_1,\ldots,\alpha_k)\in f_{\vec{s}}^{-1}(\beta_1,\ldots,\beta_n)\}$ and $c_{\vec{s},1}^{\beta_1}\cap\ldots\cap c_{\vec{s},n}^{\beta_n}$ are equivalent.*

7 Tuples of terms

Let $k,m,n\geq 0$ be such that $n\leq k$. Let $(a_1,\ldots,a_n)\in TER_{m,k}^n$. For all m-vectors $\vec{s}$, we define on $\{0,1\}^k$ the equivalence relation $\sim_{(a_1,\ldots,a_n)}^{k,\vec{s}}$ by $(\alpha_1,\ldots,\alpha_k)\sim_{(a_1,\ldots,a_n)}^{k,\vec{s}}(\alpha_1',\ldots,\alpha_k')$ iff for all positive integers $i\leq n$, $x_1^{\alpha_1}\cap\ldots\cap x_k^{\alpha_k}\in mon_{\vec{s}}(k,a_i)$ iff $x_1^{\alpha_1'}\cap\ldots\cap x_k^{\alpha_k'}\in mon_{\vec{s}}(k,a_i)$.

Proposition 9. *For all m-vectors $\vec{s}$, $\sim_{(a_1,\ldots,a_n)}^{k,\vec{s}}$ has at most 2^n equivalence classes on $\{0,1\}^k$.*

Proposition 10. *There exists a (k,m,n)-correspondence f such that for all m-vectors $\vec{s}$ and for all $(\alpha_1,\ldots,\alpha_k),(\alpha_1',\ldots,\alpha_k')\in\{0,1\}^k$, if $f_{\vec{s}}(\alpha_1,\ldots,\alpha_k)=f_{\vec{s}}(\alpha_1',\ldots,\alpha_k')$ then $(\alpha_1,\ldots,\alpha_k)\sim_{(a_1,\ldots,a_n)}^{k,\vec{s}}(\alpha_1',\ldots,\alpha_k')$.*

A (k,m,n)-correspondence f is *balanced* iff for all m-vectors $\vec{s}$ and for all $(\alpha_1,\ldots,\alpha_k),(\alpha_1',\ldots,\alpha_k')\in\{0,1\}^k$, if $f_{\vec{s}}(\alpha_1,\ldots,\alpha_k)=f_{\vec{s}}(\alpha_1',\ldots,\alpha_k')$ then $(\alpha_1,\ldots,\alpha_k)\sim_{(a_1,\ldots,a_n)}^{k,\vec{s}}(\alpha_1',\ldots,\alpha_k')$. By Proposition 10, let f be a balanced (k,m,n)-correspondence. For all m-vectors $\vec{s}$, by means of f — more precisely, of $f_{\vec{s}}$ —, we define the n-tuple $(b_{\vec{s},1},\ldots,b_{\vec{s},n})$ of terms by setting for all positive integers $i\leq n$, $b_{\vec{s},i}=\bigcup\{x_1^{\beta_1}\cap\ldots\cap x_n^{\beta_n}:\ x_1^{\alpha_1}\cap\ldots\cap x_k^{\alpha_k}\in mon_{\vec{s}}(k,a_i)$ and $f_{\vec{s}}(\alpha_1,\ldots,\alpha_k)=(\beta_1,\ldots,\beta_n)\}$. An n-tuple $(b_1,\ldots,b_n)\in TER_{m,n}^n$ of terms is *properly obtained* from $(a_1,\ldots,a_n)$ iff for all positive integers $i\leq n$, $b_i=\bigcup\{p_1^{\vec{s}(1)}\cap\ldots\cap p_m^{\vec{s}(m)}\cap b_{\vec{s},i}:\ \vec{s}$ is an m-vector$\}$. For all m-vectors $\vec{s}$, for all $(\beta_1,\ldots,\beta_n)\in\{0,1\}^n$ and for all positive integers $i\leq n$, let $f_{\vec{s}}^{-1}(\beta_1,\ldots,\beta_n)$, $\Delta_{\vec{s},i}$ and $c_{\vec{s},i}$ be as in Section 6. A substitution υ is *properly obtained* from $(a_1,\ldots,a_n)$ iff for all variables y, if $y\not\in\{x_1,\ldots,x_n\}$ then $\upsilon(y)=y$ and for all positive integers $i\leq n$, $\upsilon(x_i)=\bigcup\{p_1^{\vec{s}(1)}\cap\ldots\cap p_m^{\vec{s}(m)}\cap c_{\vec{s},i}:\ \vec{s}$ is an m-vector$\}$.

Proposition 11. *Let $(b_1,\ldots,b_n)\in TER_{m,n}^n$ and υ be a substitution. If $(b_1,\ldots,b_n)$ and υ are properly obtained from $(a_1,\ldots,a_n)$ then for all positive integers $i\leq n$, considered as formulas in* **CPL**, *the terms a_i and $\bar{\upsilon}(b_i)$ are equivalent.*

Proposition 12. *Let σ be the substitution such that for all variables y, if $y\not\in\{x_1,\ldots,x_n\}$ then $\sigma(y)=y$ and for all positive integers $i\leq n$, $\sigma(x_i)=a_i$. Let $(b_1,\ldots,b_n)\in TER_{m,n}^n$ and τ be the substitution such that for all variables y, if $y\not\in\{x_1,\ldots,x_n\}$ then $\tau(y)=y$ and for all positive integers $i\leq n$, $\tau(x_i)=b_i$. Let υ be a substitution. If $(b_1,\ldots,b_n)$ and υ are properly obtained from $(a_1,\ldots,a_n)$ then $\tau\circ\upsilon\simeq\sigma$.*

Proposition 13. *Let $(b_1,\ldots,b_n)\in TER_{m,n}^n$. If $(b_1,\ldots,b_n)$ is properly obtained from $(a_1,\ldots,a_n)$ then for all valuations $\mathcal{V}$ on $RCP(\mathbb{R}^2)$, there exists a valuation $\mathcal{V}'$ on $RCP(\mathbb{R}^2)$ such that for all positive integers $i\leq n$, $\bar{\mathcal{V}}(b_i)=\bar{\mathcal{V}}'(a_i)$.*

Proposition 14. *Let σ be the substitution such that for all variables y, if $y\not\in\{x_1,\ldots,x_n\}$ then $\sigma(y)=y$ and for all positive integers $i\leq n$, $\sigma(x_i)=a_i$. Let $\varphi\in FOR$. Let $(b_1,\ldots,b_n)\in TER_{m,n}^n$ and τ be the substitution such that for all variables y, if $y\not\in\{x_1,\ldots,x_n\}$ then $\tau(y)=y$ and for all positive integers $i\leq n$, $\tau(x_i)=b_i$. If $(b_1,\ldots,b_n)$ is properly obtained from $(a_1,\ldots,a_n)$ then σ is a unifier of φ only if τ is a unifier of φ.*

8 Unification type

Now, we are ready to prove the main results of this paper.

Proposition 15. *Let $\varphi \in FOR$. Let $m, n \geq 0$ be such that φ's constants form a subset of $\{p_1, \ldots, p_m\}$ and φ's variables form a subset of $\{x_1, \ldots, x_n\}$. For all unifiers σ of φ, there exists a unifier τ of φ in $\Sigma_{m,n}$ such that $\tau \preceq \sigma$.*

Proof. Let σ be a unifier of φ. Let σ' be the substitution defined by $\sigma'(x_i) = \sigma(x_i)$ for all $i = 1 \ldots n$ and $\sigma'(y) = y$ for all y not in $\{x_1, \ldots, x_n\}$. Obviously, σ' is a unifier of φ too. Now, it may happen that for some $i \in \{1, \ldots, n\}$, $\sigma'(x_i)$ contains extra constants which do not appear in φ. If it is, then let $q_1, \ldots, q_l$ be the list of these extra constants. Take new variables $z_1, \ldots, z_l$ and define σ'' by uniformly replacing in $\sigma'(x_1), \ldots, \sigma'(x_n)$ each occurrence of $q_1, \ldots, q_l$ by, respectively, $z_1, \ldots, z_l$. Obviously, σ'' is a unifier of φ too. As a result, for all constants q, if $q \notin \{p_1, \ldots, p_m\}$ then for all positive integers $i \leq n$, q does not occur in $\sigma''(x_i)$ and for all variables y, if $y \notin \{x_1, \ldots, x_n\}$ then $\sigma''(y) = y$. Let $k \geq 0$ and $(a_1, \ldots, a_n) \in TER^n_{m,k}$ be such that $n \leq k$ and for all positive integers $i \leq n$, $\sigma''(x_i) = a_i$. For all m-vectors $\vec{s}$, let $\sim^{k,\vec{s}}_{(a_1,\ldots,a_n)}$ be as in Section 7. By Proposition 10, let f be a balanced (k, m, n)-correspondence. For all m-vectors $\vec{s}$, for all $(\beta_1, \ldots, \beta_n) \in \{0, 1\}^n$ and for all positive integers $i \leq n$, let $f^{-1}_{\vec{s}}(\beta_1, \ldots, \beta_n)$, $\Delta_{\vec{s},i}$ and $c_{\vec{s},i}$ be as in Section 6. Let $(b_1, \ldots, b_n) \in TER^n_{m,n}$ be an n-tuple of terms properly obtained from $(a_1, \ldots, a_n)$. Let τ be the substitution such that for all variables y, if $y \notin \{x_1, \ldots, x_n\}$ then $\tau(y) = y$ and for all positive integers $i \leq n$, $\tau(x_i) = b_i$. Remark that τ is in $\Sigma_{m,n}$. Moreover, by Proposition 14, τ is a unifier of φ. Let υ be a substitution properly obtained from $(a_1, \ldots, a_n)$. By Proposition 12, $\tau \circ \upsilon \simeq \sigma''$. Hence, $\tau \preceq \sigma''$. By the construction of τ, one can deduce that $\tau \preceq \sigma$. □

Proposition 16. *Let $\varphi \in FOR$. If φ is unifiable then φ is either finitary, or unitary.*

Proof. By Propositions 2, 3 and 15. □

Proposition 17. *$\mathcal{UNIF}$ is in $EXPSPACE$.*

Proof. Let $\varphi \in FOR$. Let $m, n \geq 0$ be such that φ's constants form a subset of $\{p_1, \ldots, p_m\}$ and φ's variables form a subset of $\{x_1, \ldots, x_n\}$. By Proposition 15, the reader may easily verify that φ is unifiable iff there exists a unifier σ of φ in $\Sigma_{m,n}$. Each σ in $\Sigma_{m,n}$ is completely described by the terms $\sigma(x_i) \in TER_{m,n}$, i ranging over $\{1, \ldots, n\}$. Hence, by Proposition 7, each σ in $\Sigma_{m,n}$ is completely described by the disjunctions of conjunctions $\bigcup\{p_1^{\vec{s}(1)} \cap \ldots \cap p_m^{\vec{s}(m)} \cap x_1^{\alpha_1} \cap \ldots \cap x_n^{\alpha_n} : \vec{s}$ is an m-vector and $x_1^{\alpha_1} \cap \ldots \cap x_n^{\alpha_n} \in mon_{\vec{s}}(n, \sigma(x_i))\}$, i ranging over $\{1, \ldots, n\}$. Obviously, the size of these disjunctions of conjunctions is at most exponential in $m + n$. Since the validity problem is in $PSPACE$ [Kontchakov *et al.* (2008), Kontchakov *et al.* (2010), Kontchakov *et al.* (2014)], therefore $\mathcal{UNIF}$ is in $EXPSPACE$. □

9 Conclusion

In this paper, we have adapted to the problem of unifiability with constants in TLs the line of reasoning developed by Balbiani and Gencer [4] within the simpler context of the problem of unifiability without constants in Boolean Region Connection Calculus. Much remains to be done. Firstly, about the choice of the mereotopological space $RCP(\mathbb{R}^2)$. It remains to see whether the line of reasoning developed in this paper will still apply to $RC(\mathbb{R}^2)$ and $RCS(\mathbb{R}^2)$. What happens if we consider mereotopological spaces over the topological spaces $(\mathbb{R}^n, \tau_{\mathbb{R}^n})$, i.e. the real space $\mathbb{R}^n$ of dimension n together with its ordinary topology $\tau_{\mathbb{R}^n}$, when $n \geq 3$? Secondly, about the computability of the unifiability problem in TLs. By Proposition 17, this problem is decidable. Nevertheless, its exact complexity is still unknown. In this respect, we believe that arguments developed in [1] could be used. Thirdly, about adding to the language the predicate of connectedness or the predicate of internal connectedness considered in [8, 9, 10]. The line of reasoning developed in this paper up to Proposition 16 will still apply to these extended languages. Nevertheless, in that case, as proved in [8, 9, 10], the validity problem becomes undecidable.

Acknowledgements

Special acknowledgement is heartly granted to Philippe Balbiani and Tinko Tinchev who contributed to the development of the work we present today. We also make a point of thanking the anonymous referees for their feedback.

References

[1] Baader, F.: *On the complexity of Boolean unification.* Information Processing Letters **67** (1998) 215–220.

[2] Baader, F., Borgwardt, S., Morawska, B.: *Extending unification in $\mathcal{EL}$ towards general TBoxes.* In: Principles of Knowledge Representation and Reasoning. AAAI Press (2012) 568–572.

[3] Baader, F., Ghilardi, S.: *Unification in modal and description logics.* Logic Journal of the IGPL **19** (2011) 705–730.

[4] Balbiani, P., Gencer, Ç.: *Finitariness of elementary unification in Boolean Region Connection Calculus.* In: Frontiers of Combining Systems. Springer (2017) 281–297.

[5] Balbiani, P., Tinchev, T., Vakarelov, D.: *Modal logics for region-based theories of space.* Fundamenta Informaticæ **81** (2007) 29–82.

[6] Dzik, W.: *Unification Types in Logic.* Wydawnicto Uniwersytetu Slaskiego (2007).

[7] Ghilardi, S.: *Best solving modal equations.* Annals of Pure and Applied Logic **102** (2000) 183–198.

[8] Kontchakov, R., Nenov, Y., Pratt-Hartmann, I., Zakharyaschev, M.: *Topological logics with connectedness over Euclidean spaces.* ACM Transactions on Computational Logic **14** (2013) DOI: 10.1145/2480759.2480765.

[9] Kontchakov, R., Pratt-Hartmann, I., Wolter, F., Zakharyaschev, M.: *Spatial logics with connectedness predicates.* Logical Methods in Computer Science **6** (2010) 1–43.

[10] Kontchakov, R., Pratt-Hartmann, I., Zakharyaschev, M.: *Spatial reasoning with $RCC8$ and connectedness constraints in Euclidean spaces.* Artificial Intelligence **217** (2014) 43–75.

[11] Martin, U., Nipkow, T.: *Boolean unification — the story so far.* Journal of Symbolic Computation **7** (1989) 275–293.

[12] Pratt-Hartmann, I.: *First-order mereotopology.* In: Handbook of Spatial Logics. Springer (2007) 13–97.

[13] Tinchev, T., Vakarelov, D.: *Logics of space with connectedness predicates: complete axiomatizations.* In: *Advances in Modal Logic.* College Publications (2010) 434–453.

[14] Vakarelov, D.: *Region-based theory of space: algebras of regions, representation theory, and logics.* In: *Mathematical Problems from Applied Logic. Logics for the XXIst Century. II.* Springer (2007) 267–348.

[15] Wolter, F., Zakharyaschev, M.: *Spatio-temporal representation and reasoning based on RCC-8.* In: Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning. Morgan Kaufmann (2000) 3–14.

Rewriting with Generalized Nominal Unification

Yunus D. K. Kutz[1] and Manfred Schmidt-Schauß[2*]

[1] Goethe-University Frankfurt am Main, Germany, `kutz@ki.informatik.uni-frankfurt.de`
[2] Goethe-University Frankfurt am Main, Germany, `schauss@ki.informatik.uni-frankfurt.de`

Abstract

We consider rewriting, critical pairs and confluence tests on rewrite rules using nominal notation. Computing critical pairs is done using nominal unification, and rewriting using nominal matching. The progress is that we permit atom variables in the notation and in the unification algorithm, which generalizes previous approaches using usual nominal unification

Keywords: nominal unification, atom variables, nominal rewriting, Knuth-Bendix criterion,

1 Introduction

The goal of this paper is to demonstrate the expressive power of nominal unification with atom variables [14] also in applications, where we consider rewriting and critical pairs ala Knuth-Bendix [7] in a higher-order language with alpha-equivalence and nominal modeling, where in the nominal unification algorithm also atom-variables are permitted in addition to expression-variables and where the rewriting is done using a corresponding form of nominal matching with atom variables. This generalizes the approach in [5, 2]. The application of nominal unification with atom variables avoids guessing of (dis-)equality of atom, which is necessary not only as a pre-procedure by the previous uses of nominal unification in rewriting, but also in rewriting sequences in every single rewriting step.

Nominal techniques [10, 9] support machine-oriented reasoning on the syntactic level for higher-order languages and support alpha-equivalence. An algorithm for nominal unification was first described in [17], which outputs unique most general unifiers. More efficient algorithms are given in [3, 8], also exhibiting a quadratic algorithm. The approach is also used in higher-order logic programming [4] and in automated theorem provers like nominal Isabelle [15, 16]. Nominal unification was generalized to permit also atom variables [14] where also in the generalization, unique most general unifiers are computed, while the decision problem is NP-complete.

As an extended example, illustrating also the ideas and potentials of the nominal modelling and unification in rewriting, in particular with atom variables, we consider the monad laws [18]. An informal explanation is that monads are an implementation of sequential actions, as extension of lambda calculus, where $a_1 \gg= a_2$ means a sequential combination of actions: a_1 is executed before a_2, and the return-value v of a_1 is used in the next action, written in lambda notation as $(a_2\ v)$. Besides the operational behavior, there is a set of monad laws, describing the desired behavior of monadic combination as an equational theory (see below). Hamana[6] used second-order unification to show confluence. However, second-order unification is undecidable, and thus the extension of this idea will in general lead to undecidable algorithmic questions. Thus, we use (decidable) nominal unification with atom-variables to obtain the same result, however, for a finer notion of unification and of equivalence.
We will use the following encoding: `return` is a function symbol of arity 0, `app` $\gg=$ are function

*supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SCHM 986/11-1.

symbols of arity 2, where we write `>>=` as infix, and `app` as juxtaposition. A, B, C denote atom-variables, and other upper-case letters expression-variables.

The three monad laws are encoded as follows:

$$\begin{array}{llll}
(Id_l) & \emptyset & \vdash (\texttt{return}\ X) \gg\!= F & \rightarrow F\ X \\
(Id_r) & \emptyset & \vdash M \gg\!= \texttt{return} & \rightarrow M \\
(A) & A\#F, G & \vdash (M \gg\!= F) \gg\!= G & \rightarrow M \gg\!= (\lambda A.(F\ A) \gg\!= G)
\end{array}$$

Additionally we add η-reduction and as a very weak version of β-reduction, we add $B\beta$. Note that $B\beta$ is a consequence of the monad laws as equations (w.r.t. α-equivalence), see Fig. 1.

$$\begin{array}{llll}
(\eta) & A\#F & \vdash (\lambda A.F\ A) & \rightarrow F \\
(B\beta) & A\#F, G & \vdash (\lambda A.(F\ A) \gg\!= G)\ X & \rightarrow (F\ X) \gg\!= G
\end{array}$$

Note that the combined rewriting system is terminating, which is a prerequisite for applying the technique of local confluence for showing confluence of rewrite system.

Our rewrite system is higher-order, but we only use nominal unification for computing the critical pairs and nominal matching for rewriting, where we permit atom variables in every case.

There are the following critical pairs, using nominal unification with atom variables:

1. $(B\beta)$ in (η).
2. (Id_l) in $(B\beta)$.
3. (Id_l) equal to Id_r.
4. (Id_l) in A.
5. (Id_r) in $(B\beta)$.
6. (Id_r) in A.
7. A in A

The critical pairs (2), (3), and (5) are trivial or easily joinable. The remaining ones are treated separately (see Fig. 1). The pair arising from the overlap of the associativity rule with itself needs a check if two freshness environment-expression pairs are equivalent (which indeed they are), which is done comparing the set of ground instances, respecting the freshness constraints.

The next example, also motivating the use of atom-variables, is a rule in the concurrent calculus CHF [12]. It permits rewriting $\texttt{let}\ y = c\ x_1 \ldots x_n\ \texttt{in}\ C[y] \rightarrow \texttt{let}\ y = c\ x_1 \ldots x_n\ \texttt{in}\ C[c\ x_1 \ldots x_n]$, and can be applied to $C_1[\texttt{let}\ z = c\ y_1 \ldots y_n\ \texttt{in}\ C_2[z]]$ even if the y_i are not pairwise different variables, which is in contrast to usual nominal rewriting using atoms instead of atom variables, since a unique unifier covers all possibilities of equal/unequal atoms.

2 Nominal Rewriting

We first introduce some notation [14].

Let $\mathcal{F}$ be a set of function symbols $f \in \mathcal{F}$, s.t. each f has a fixed arity $ar(f) \geq 0$. Let $\mathcal{A}t$ be *the set of atoms* ranged over by a, b, c. The ground language NL_a is defined by the grammar:

$$e \quad ::= \quad a \mid (f\ e_1 \ldots e_{ar(f)}) \mid \lambda a.e$$

(1)
$$\begin{tikzcd}
\begin{array}{c} A\#F,G \\ C\#(\lambda A.(F\ A) \gg= G)\ X \\ (\lambda C.(\lambda A.(F\ A) \gg= G)\ X)\ C \end{array} \arrow[r, "\eta"'] \arrow[d, "B\beta"] & (\lambda A.(F\ A) \gg= G)\ X \arrow[d, "B\beta"] \\
\lambda C.((F\ X) \gg= G)\ C \arrow[r, "\eta"] & ((F\ X) \gg= G)
\end{tikzcd}$$

(4)
$$\begin{tikzcd}
\begin{array}{c} A\#F,G \\ (\mathtt{return}\ X \gg= F) \gg= G \end{array} \arrow[r, "A"] \arrow[d, "Id_l"'] & \mathtt{return}\ X \gg= (\lambda A.F\ A \gg= G) \arrow[d, "Id_l"] \\
F\ X \gg= G & (\lambda A.F\ A \gg= G)\ X \arrow[l, "B\beta"]
\end{tikzcd}$$

(6)
$$\begin{tikzcd}
\begin{array}{c} A\#G \\ (M \gg= \mathtt{return}) \gg= G \end{array} \arrow[r, "A"] \arrow[d, "Id_r"'] & M \gg= (\lambda A.\mathtt{return}\ A \gg= G) \arrow[d, "Id_l"] \\
M \gg= G & M \gg= \lambda A.G\ A \arrow[l, "\eta"]
\end{tikzcd}$$

(7)
$$\begin{tikzcd}
\begin{array}{c} A\#F,G \\ A'\#G,G' \\ ((M \gg= F) \gg= G) \gg= G' \end{array} \arrow[r, "A"] \arrow[d, "A"'] & ((M \gg= (\lambda A.F\ A \gg= G)) \gg= G' \arrow[d, "A"] \\
(M \gg= F) \gg= (\lambda A'.G\ A' \gg= G') \arrow[dd, "A"'] & \begin{array}{c} B\#\lambda A.F\ A \gg= G \\ B\#G' \\ M \gg= (\lambda B.((\lambda A.F\ A \gg= G)\ B \gg= G')) \end{array} \arrow[d, "B\beta"] \\
 & M \gg= (\lambda B.(F\ B \gg= G) \gg= G') \arrow[d, "A"] \\
\begin{array}{c} B'\#F \\ B'\#\lambda A'.G\ A' \gg= G' \\ M \gg= (\lambda B'.F\ B' \gg= (\lambda A'.G\ A' \gg= G')) \end{array} \arrow[r, leftrightarrow, "\alpha{-}inst"] & \begin{array}{c} C\#G,G' \\ M \gg= (\lambda B.F\ B \gg= (\lambda C.G\ C \gg= G')) \end{array}
\end{tikzcd}$$

Figure 1: Joining the nontrivial critical pairs of Monad Theory

where λ is a binder for atoms. The basic constraint $a\#e$ is valid if a is not free in e and a set of constraints ∇ is valid if all constraints are valid.

We will use the following definition of α-equivalence on NL_a:

Definition 2.1. *Syntactic α-equivalence $\sim$ in NL_a is inductively defined:*

$$\frac{}{a \sim a} \qquad \frac{\forall i : e_i \sim e'_i}{(f\ e_1 \ldots e_{ar(f)}) \sim (f\ e'_1 \ldots e'_{ar(f)})} \qquad \frac{e \sim e'}{\lambda a.e \sim \lambda a.e'} \quad \frac{a\#e' \wedge e \sim (a\ b)\cdot e'}{\lambda a.e \sim \lambda b.e'}$$

Note that $\sim$ is identical to the equivalence relation generated by α-equivalence by renaming binders, which can be proved in a simple way by arguing on the (binding-)structure of expressions (using deBruijn-indices) and hence $\sim$ is an equivalence relation on NL_a. It is also a congruence on NL_a, i.e., for a context C, we have $e_1 \sim e_2$ implies $C[e_1] \sim C[e_2]$.

Let $\mathcal{S}$ be a set of expression-variables ranged over by S, T and let $\mathcal{A}$ be the set of atom-variables ranged over by A, B. The grammar of the nominal language NL_{AS} with atom-variables is:

$$\begin{array}{lcl} e & ::= & A \mid S \mid \pi{\cdot}A \mid \pi{\cdot}S \mid (f\ e_1 \ldots e_{ar(f)}) \mid \lambda\pi{\cdot}A.e \\ \pi & ::= & \emptyset \mid ((\pi{\cdot}A)\ (\pi'{\cdot}A')) \cdot \pi'' \end{array}$$

where π is a permutation and $\emptyset$ denotes the identity. Note that we permit nested permutations. The expression $((\pi{\cdot}A)\ (\pi'{\cdot}A'))$ is a single nested swapping. We assume that permutation application is done as simplification if possible. The inverse π^{-1} of a permutation $\pi = sw_1{\cdot}\ldots{\cdot}sw_n$ with swappings sw_i is the expression $sw_n{\cdot}\ldots{\cdot}sw_1$.
$AtVar(e)$ are the atom-variables contained in e, $ExVar(e)$ the expression-variables contained in e and $Var(e) = AtVar(e) \cup ExVar(e)$.

The ground language of NL_{AS} is NL_a, i.e. a ground substitution replaces atom variables by atoms and expression variables by expression in NL_a.
A *freshness constraint* ∇ is a set (a Boolean conjunction) of constraints of the form $A\#e$. Note that constraints of the form $\pi{\cdot}A\#e$ are equivalent to $A\#\pi^{-1}{\cdot}e$, hence we omit them from the syntax. ∇ is valid under a ground substitution γ, if $\nabla\gamma$ is valid; this is written as: $\gamma \models \nabla$.

Definition 2.2. *A* rewrite rule *is of the form* $(\nabla, l \to r)$*, where* ∇ *is a freshness constraint and* l, r *are expressions of* NL_{AS} *and* l *is not an atom- nor an expression-variable, and* $ExVar(r) \subseteq ExVar(l)$*. The rewrite rule is also written as* $\nabla \vdash l \to r$*.*

We will illustrate the ideas for rewriting by two examples.

Example 2.3. *Let the rewrite rule (for garbage collection) be* $(\{A\#S'\}, \texttt{let}\ A = S\ \texttt{in}\ S' \to S')$*. Then a rewrite step on the ground expression* $(\texttt{let}\ x = a\ \texttt{in}\ \lambda b.b)$ *without any atom- nor expression-variables can be done as follows: We have to compute a nominal matcher of* $(\texttt{let}\ A = S\ \texttt{in}\ S') \preceq (\texttt{let}\ x = a\ \texttt{in}\ \lambda b.b)$*, which results in a substitution* $\sigma = \{A \mapsto x; S \mapsto a; S' \mapsto \lambda b.b\}$ *and the resulting freshness constraint* $(A\#S')\sigma$ *is valid, since* $x\#\lambda b.b$ *is valid. The resulting expression of the rewriting step is* $\lambda b.b$*.*

This form of application appears to be too restricted, since we also want to rewrite expressions containing atom- and expression-variables, perhaps restricted by freshness constraints. In doing so we gain the ability to rewrite a multitude of related ground expressions (all instances of the expression-constraint pair) and are able to join critical pairs. We generalize the example and permit atom- and expression-variables in the target expression.

Example 2.4. *We use the same rewrite rule as above:* $(\{A\#S'\}, \texttt{let}\ A = S\ \texttt{in}\ S' \to S')$*. Then let the abstract expression be* $(\texttt{let}\ B = S_3\ \texttt{in}\ S_4)$ *with the additional freshness constraint* $B\#S_4$*, which represents a set of ground expressions. Rewriting can informally be done as follows: we have to compute a matcher of* $(\texttt{let}\ A = S\ \texttt{in}\ S') \preceq (\texttt{let}\ B = S_3\ \texttt{in}\ S_4)$*. This is done by treating* B, S_3 *and* S_4 *like constants. The freshness constraint* $B\#S_4$ *is interpreted as a part of the description of the input.*

The matching substitution is $\sigma = \{A \mapsto B; S \mapsto S_3; S' \mapsto S_4\}$*. The freshness constraint* $\{A\#S'\}$ *of the rule is instantiated to* $\{B\#S_4\}$*, which is identical to the input constraint. Thus the result of rewriting is* $(S_4, \{B\#S_4\})$*.*

$$\text{(M1)}\ \frac{(\Gamma \mathbin{\dot\cup} \{e \preceq e\}, \nabla, \theta)}{(\Gamma, \nabla, \theta)} \qquad \text{(M2)}\ \frac{(\Gamma \mathbin{\dot\cup} \{\pi \cdot S \preceq e\}, \nabla, \theta)}{(\Gamma \mathbin{\dot\cup} \{S \preceq \pi^{-1} \cdot e\}, \nabla, \theta)}$$

$$\text{(M3)}\ \frac{(\Gamma \mathbin{\dot\cup} \{S \preceq e\}, \nabla, \theta)}{(\Gamma[e/S], \nabla[e/S], \theta \cup \{S \mapsto e\})} \qquad \text{(M4)}\ \frac{(\Gamma \mathbin{\dot\cup} \{\pi_1{\cdot}A \preceq \pi_2{\cdot}B\}, \nabla, \theta)}{(\Gamma, \nabla \cup \{A =_{\#} \pi_1^{-1}{\cdot}\pi_2{\cdot}B\}, \theta)}$$

$$\text{(M5)}\ \frac{(\Gamma \mathbin{\dot\cup} \{(f\ e_1 \ldots e_{ar(f)}) \preceq (f\ e'_1 \ldots e'_{ar(f)})\}, \nabla, \theta)}{(\Gamma \mathbin{\dot\cup} \{e_1 \preceq e'_1, \ldots, e_{ar(f)} \preceq e'_{ar(f)}\}, \nabla, \theta)}$$

$$\text{(M6)}\ \frac{(\Gamma \mathbin{\dot\cup} \{\lambda\pi_1{\cdot}A_1.e_1 \preceq \lambda\pi_2{\cdot}A_2.e_2\}, \nabla, \theta)}{(\Gamma \mathbin{\dot\cup} \{e_1 \preceq ((\pi_1{\cdot}A_1)\ (\pi_2{\cdot}A_2)){\cdot}e_2\}, \nabla \cup \{(A_1 \# \pi_1^{-1}{\cdot}(\lambda\pi_2{\cdot}A_2.e_2))\}, \theta)}$$

Figure 2: Rules of *NomMatchAS*

2.1 Nominal Match

The goal of matching (in a first approximation) is to find out for given expressions e, e' whether there is a substitution θ such that $e\theta$ represents e'. Below we will use (Δ, e') as targets, i.e., expressions e' that are restricted by a freshness constraint Δ. The semantics of a pair (Δ, s) is a set of ground expressions: $\{s\sigma \mid s\sigma \text{ is ground}, \Delta\sigma \text{ is valid}\}$.

The basic components of a freshness constraint ∇ are single constraints $A\#e$. Certain basic constraints can be written in different notation or more explicitly: (i) $A_1 =_{\#} \pi{\cdot}A_2$ means equality and abbreviates $A_1\#\lambda(\pi{\cdot}A_2).A_1$; (ii) $A_1 \neq_{\#} \pi{\cdot}A_2$ means disequality and is another way to write $A_1\#\pi{\cdot}A_2$; and (iii) $A_1\#\lambda\pi{\cdot}A_2.e$ could be written as a disjunction $(A_1 =_{\#} \pi{\cdot}A_2 \vee A_1\#e)$ (but we will not do this explicitely).

The rules for computing a match $e \preceq e'$ (ignoring the Δ-constraints) in NL_{AS} are in Fig. 2. The rules operate on a triple (Γ, ∇, θ) of a set of match-equations Γ, freshness constraints ∇ and a substitution θ.

Definition 2.5. *A matcher for a matching problem $(\Delta, e) \preceq (\Delta', e')$ is a tuple (∇, θ) such that:*

- $\nabla \models e\theta \sim e'$*, i.e.* $\forall\gamma : \nabla\gamma$ *is valid and* $e\gamma, e'\gamma$ *ground* $\implies e\sigma\gamma \sim e'\gamma$
- *For every ground substitution γ with domain $AtVar(\Delta')$, such that $\gamma \models \Delta'$, there is an extension γ' of γ, such that $\gamma' \models \nabla \cup \Delta\sigma$. This means the following formula must hold: $\forall\mathcal{B}.\exists\mathcal{A}.(\Delta' \implies \nabla \cup \Delta\sigma)$, where $\mathcal{B} = Var(\Delta', e')$ and $\mathcal{A} = Var(\nabla \cup \Delta\sigma) \setminus \mathcal{B}$*

Let $FA(.)$ denote the free atom variables in a constraint or an expression.

Definition 2.6. *Let the input of NomMatchAS be (Δ, e) and (Δ', e'), where $FA(\Delta') \subseteq FA(e')$ must hold,*
The matching algorithm NomMatchAS starts with $\Gamma = (\{e \preceq e'\}, \Delta, \emptyset)$. Then it performs the rules in Fig. 2 until the triple is $(\emptyset, \nabla, \theta)$, i.e. Γ is empty.
If the process gets stuck, then there is no match.
If Γ is empty, then the second matching condition needs to be tested, i.e. the formula $\forall\mathcal{B}.\exists\mathcal{A}.(\Delta' \implies \nabla)$, where $\mathcal{B} = Var(\Delta', e')$ and $\mathcal{A} = Var(\nabla) \setminus \mathcal{B}$, must hold.

The condition $\forall\mathcal{B}.\exists\mathcal{A}.(\Delta' \implies \nabla)$ can be made algorithmic by only looking for equivalence relations on $\mathcal{B}$ (that may be induced by the substitutions γ). I.e. for every equivalence relation $\sim$

on $\mathcal{B}$, let $EQ(\sim)$ be the freshness constraint that exactly describes the equations and disequations (of atom variables) for $\sim$: Then $(\Delta' \cup EQ(\sim)) \implies (\nabla \cup EQ(\sim))$ must be valid, which can be checked in polynomial time.

Proposition 2.7. *The complexity of the final test of the matching algorithm is in* Π_2^P.

Proof. The quantifiers have the effect of adding an equivalence relation on the atom-variables. If the constraint Δ is instantiated with atom variables by a ground substitution σ, then very single freshness constraint $A\#e$ can be decided in polynomial time in the size of the constraint by simply checking $A\sigma\#e\sigma$. □

We are working on determining lower complexity bounds for a single rewriting step. The same techniques as in [14] permit to show:

Theorem 2.8. *NomMatchAS is sound and complete and computes at most one match.*

2.2 Rewriting and Overlap

We define nominal rewriting of expression with atom- and expression-variables on targets $(\Delta, C[s])$ where s is the sub-expression that is to be modified and Δ is a freshness constraint.

Definition 2.9. *Let* $(\nabla, l \to r)$ *be a rewrite rule and let* $(\Delta, C[s])$ *be the object to be rewritten, where we assume that* $Var(\nabla, l \to r) \cap Var(\Delta, C[s]) = \emptyset$. *(The condition can be achieved by a renaming of* $\nabla, l \to r$.*) A rewrite step is defined as follows:*

Let (∇', σ) *be a nominal matcher of* $(\nabla, l) \preceq (\Delta, s)$ *computed with NomMatchAS and let* $\nabla'' = \nabla' \wedge \nabla\sigma$.

Then the result of rewriting is $(\Delta \cup \nabla'', C[r\sigma])$.

Now we define overlap, join and critical pairs and the Knuth Bendix-criterion in our setting.

Definition 2.10. *An* overlap *of two (variable-disjoint) rewrite rules* $(\nabla_1, l_1 \to r_1)$ *and* $(\nabla_2, l_2 \to r_2)$ *is computed by the following algorithm. Select a non-variable position* p *in* l_1*, represented by a context* C*, such that* $C[l_1'] = l_1$ *and the hole of* C *is at expression-position* p*. Apply the unification algorithm in [14] to the equation* $l_1' \doteq l_2$ *and constraint* $\nabla_1 \wedge \nabla_2$*. If there is an overlap, then the (unique) result of the unification algorithm is a constraint and a substitution* (∇', σ)*, where we assume that* $Dom(\sigma) \cap Var(\nabla') = \emptyset$*. The resulting overlap expression is* $(l_1\sigma, \nabla')$.

The critical pair *consists of the corresponding rewriting results:* $((r_1\sigma, \nabla'), (C\sigma[r_2\sigma], \nabla'))$.

For the final join, we have to check for the equivalence of targets (Δ_1, e_1) and (Δ_2, e_2). In general this cannot be done by purely syntactic means. A correct method is to compute whether these match each other also respecting all the freshness constraints. This test is decidable.

3 Conclusion

Future work is extend the method to equational theories that are defined in more general ways, for example using descriptions of infinite sets of equations by context variables in rules, and applying the nominal unification algorithm as described in [13].

A potential application are some reduction rules in the call-by-need calculus of [1] or in CHF [11, 12], like `let` $y = v$ `in` $C[y] \to$ `let` $y = v$ `in` $C[v]$, where v is a value, or similar rules.

Further applications are other higher-order theories of data structures like the monad theory.

References

[1] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *POPL'95*, pages 233–246, San Francisco, CA, 1995. ACM Press.

[2] M. Ayala-Rincón, M. Fernández, M. J. Gabbay, and A. C. Rocha-Oliveira. Checking overlaps of nominal rewriting rules. *ENTCS*, 323:39–56, 2016.

[3] C. Calvès and M. Fernández. A polynomial nominal unification algorithm. *Theor. Comput. Sci.*, 403(2-3):285–306, 2008.

[4] J. Cheney. *Nominal Logic Programming.* PhD thesis, Cornell University, Ithaca, NY, August 2004.

[5] M. Fernández and A. Rubio. Nominal completion for rewrite systems with binders. In A. Czumaj, K. Mehlhorn, A. M. Pitts, and R. Wattenhofer, editors, *Proc. 39th ICALP Part II*, volume 7392 of *LNCS*, pages 201–213. Springer, 2012.

[6] M. Hamana. How to prove your calculus is decidable: practical applications of second-order algebraic theories and computation. *PACMPL*, 1(ICFP):22:1–22:28, 2017.

[7] D. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, 1970.

[8] J. Levy and M. Villaret. An efficient nominal unification algorithm. In C. Lynch, editor, *Proc. 21st RTA*, volume 6 of *LIPIcs*, pages 209–226. Schloss Dagstuhl, 2010.

[9] A. Pitts. Nominal techniques. *ACM SIGLOG News*, 3(1):57–72, Feb. 2016.

[10] A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science.* Cambridge University Press, New York, NY, USA, 2013.

[11] D. Sabel and M. Schmidt-Schauß. A contextual semantics for concurrent Haskell with futures. In P. Schneider-Kamp and M. Hanus, editors, *Proc. 13th ACM PPDP 2011*, pages 101–112. ACM, 2011.

[12] M. Schmidt-Schauß and N. Dallmeyer. Space improvements and equivalences in a functional core language. *CoRR*, abs/1802.06498, 2018.

[13] M. Schmidt-Schauß and D. Sabel. Nominal unification with atom and context variables. In H. Kirchner, editor, *Proc. 3rd FSCD 2018).* Schloss Dagstuhl, 2018. accepted for publication.

[14] M. Schmidt-Schauß, D. Sabel, and Y. Kutz. Nominal unification with atom-variables. *Journal of Symbolic Computation*, 2018. article in press.

[15] C. Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reasoning*, 40(4):327–356, 2008.

[16] C. Urban and C. Kaliszyk. General bindings and alpha-equivalence in nominal Isabelle. *Log. Methods Comput. Sci.*, 8(2), 2012.

[17] C. Urban, A. M. Pitts, and M. Gabbay. Nominal unification. In *17th CSL, 12th EACSL, and 8th KGC*, volume 2803 of *LNCS*, pages 513–527. Springer, 2003.

[18] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

32nd International Workshop on Unification (UNIF 2018), Informal Proceedings
Oxford, 7th July, 2018

$ACUI$ Unification modulo Ground Theories

Franz Baader[1], Pavlos Marantidis[1]*, and Antoine Mottet[2]*
firstname.lastname@tu-dresden.de

[1] Theoretical Computer Science, TU Dresden, Germany
[2] Institute for Algebra, TU Dresden, Germany

Abstract

It is well-known that the unification problem for a binary associative-commutative-idempotent function symbol with a unit ($ACUI$-unification) is polynomial for unification with constants and NP-complete for general unification. We prove that the same is true if we add a finite set of ground identities. To be more precise, we first show that not only unification with constants, but also unification with linear constant restrictions is in P for any extension of $ACUI$ with a finite set of ground identities. Using well-known combination results for unification algorithms, this then yields an NP-upper bound for general unification modulo such a theory. The matching lower bound can be shown as in the case without ground identities.

1 Introduction

As shown by Kapur and Narendran [7], general $ACUI$-unification is NP-complete, while elementary unification and unification with free constants is in P. In particular, this also implies that the word problem in $ACUI$ is decidable in polynomial time. Marché proved in [8] that the word problem remains decidable if $ACUI$ is extended with a finite set of ground identities, but no complexity bounds are given. This result actually holds for a signature possibly containing several $ACUI$ symbols and free function symbols.

We are interested in whether decidability of unification for $ACUI$ is also stable under adding ground identities. In this paper, we answer this question affirmatively for the case of a single $ACUI$ symbol. The ground identities may contain this symbol and additional constant symbols, but no additional function symbols of arity greater than 0. In this setting, we can actually prove that not only decidability of unification, but also the complexity results transfer. To this purpose, we show that unification with linear constant restrictions, a notion that generalizes unification with free constants, is decidable in P. Then, using known combination results by Baader and Schulz [4], we can conclude that general unification is decidable in NP.

Our interest in $ACUI$-unification modulo an additional ground theory stems from the fact that the theory $ACUI$ is a common subtheory of the equational theories corresponding to the Description Logics $\mathcal{FL}_0$ and $\mathcal{EL}$: for $\mathcal{FL}_0$, $ACUI$ is extended with unary function symbols that behave like homomorphisms and for $\mathcal{EL}$ the additional unary function symbols behave like

*Supported by DFG Graduiertenkolleg 1763 (QuantLA)

monotone operators. Unification in $\mathcal{FL}_0$ is known to be ExpTime-complete [3] and NP-complete in $\mathcal{EL}$ [2]. However, it is not known how to extend these decidability results to unification in the presence of so-called general TBoxes, though for $\mathcal{EL}$ there are some positive results for a restricted form of TBoxes [1]. Since, from an equational theory point of view, general TBoxes correspond to finite sets of ground identities, we are interested in equational theories for which decidability of unification is stable under adding finite sets of ground identities. We will show in this paper that $ACUI$ is such a theory.

2 $ACUIG$-unification with linear constant restriction

We assume in the following that the reader is familiar with basic notions of unification theory, and in particular the difference between unification with constants, unification with linear constant restriction, and general unification. Detailed definitions and a discussion of this difference can be found in [5].

Let $\Sigma = \{+, \mathbf{0}\}$ for a binary function symbol $+$ and a constant symbol $\mathbf{0}$. We denote the equational theory that states that $+$ is an associative, commutative, and idempotent symbol with unit $\mathbf{0}$ by $ACUI$. Furthermore, let F be a countably infinite set of constants and V a countably infinite set of variables. The set of terms built from Σ, F and V is denoted by $T_\Sigma(V, F)$, and the set of ground terms, i.e., terms that do not contain variables, by $T_\Sigma(F)$. If G is a finite set of ground identities using terms in $T_\Sigma(F)$, then we denote the equational theory $ACUI \cup G$ with $ACUIG$. The constants from F not occurring in G are called *free constants*. For example, if $a, b, c \in F$ and $x, y \in V$, then $x + y$ and $a + x$ belong to $T_\Sigma(V, F)$, and $a + b + b$ and $b + a + a$ are elements of $T_\Sigma(F)$. The latter two terms are actually equivalent modulo $ACUI$. If G contains the identity $a + b = c$, then these terms are also equivalent to $a + b + c$. In fact, $b + a + a =_{ACUI} a + b + b =_{ACUI} a + b =_{ACUI} a + b + a + b =_{ACUIG} a + b + c$.

A substitution is a mapping $\sigma : V \to T_\Sigma(V, F)$, which is the identity for all but finitely many variables. It can be homomorphically extended to a mapping from $T_\Sigma(V, F)$ to $T_\Sigma(V, F)$ in the obvious way.

Definition 1 ($ACUIG$-unification problem with linear constant restriction)**.**
Input: *A finite system $\Gamma = \{s_1 =^? t_1, \ldots, s_k =^? t_k\}$ of equations between terms in $T_\Sigma(V, F)$, a finite set of ground identities $G = \{g_1 = h_1, \ldots, g_\ell = h_\ell\}$ between terms of $T_\Sigma(F)$, and a linear order $\prec$ on $X \cup D$, where $X \subseteq V$ is the set of variables occurring in Γ and $D \subseteq F$ is the set of* free *constants occurring in Γ.*

Question: *Is there a substitution σ such that $\sigma(s_i) =_{ACUIG} \sigma(t_i)$ for every $i = 1, \ldots, k$ and for every $x \in X$ and $d \in D$ we have that d does not occur in $\sigma(x)$ if $x \prec d$. Such a substitution is called an $ACUIG$-unifier of Γ w.r.t. $\prec$.*

Now, let Γ be an $ACUIG$-unification problem with linear constant restriction $\prec$, and $C = \{a_1, \ldots, a_n\}$ be the finite set of constants occurring in Γ and in G, and $X = \{x_1, \ldots, x_m\}$ the finite set of variables occurring in Γ. In order to check whether Γ has a unifier w.r.t. $\prec$ it is sufficient to consider substitutions that are the identity on $V \setminus X$ and replace every $x \in X$ by a term in $T_\Sigma(C)$, i.e., a ground term containing (in addition to $\mathbf{0}$) only constants from C. In fact, any $ACUIG$-unifier of Γ w.r.t. $\prec$ can be turned into one satisfying this property by replacing variables and constants in $F \setminus C$ with $\mathbf{0}$. If we apply such a substitution to the terms s_i, t_i occurring in Γ, then we obtain terms in $T_\Sigma(C)$.

Modulo $ACUI$, terms in $T_\Sigma(C)$ can be represented as subsets of C. These sets just consist of the constants occurring in the terms. Two ground terms are equivalent modulo $ACUI$ iff the corresponding sets are equal. For this reason, we often assume in the following that ground terms are represented as such sets. However, in the presence of a set of ground identities G, different sets may represent terms that are equivalent modulo $ACUIG$. In our above example, the terms $a + b + b$ and $b + a + a$ are both represented by the set $\{a, b\}$, whereas $a + b + c$ is represented by $\{a, b, c\}$. Intuitively, the identity $a + b = c$ can be used to add c to the set $\{a, b\}$.

We will now show how we can decide whether two sets represent terms that are equivalent modulo $ACUIG$. For this purpose we saturate the sets by using the identities in G to add constants to them, as we have done with c in our example. For each identity $g_i = h_i$ in G, let $G_i, H_i \subseteq C$ be the sets corresponding to g_i, h_i.

Given a set $A \subseteq C$, its *saturation* is obtained by iteratively applying the identities of G as follows: begin with setting $A^* := A$; as long as there is an identity $g_i = h_i$ in G such that $G_i \subseteq A^*$ and $H_i \not\subseteq A^*$ (or $H_i \subseteq A^*$ and $G_i \not\subseteq A^*$), extend A^* by setting $A^* := A^* \cup H_i$ (respectively, by setting $A^* := A^* \cup G_i$). This saturation process terminates after a number of iterations that is bounded by the cardinality of G. In fact, once an identity $g_i = h_i$ was applied in the saturation process, it is no longer applicable since the set A^* then contains $G_i \cup H_i$. It is also easy to see that the result of the saturation does not depend on the order in which rules are applied. Thus, each set A has a unique saturation A^*, which can be computed in polynomial time.

As an example, consider the set of ground identities

$$G = \{a + b + c = d, \quad b + c + e = f\}$$

and the term $s = a + f$. The saturation process for s starts with setting $A_s^* := A_s = \{a, f\}$. For the second identity, we have that $A_f = \{f\} \subseteq A_s^*$, but $A_{b+c+e} = \{b, c, e\} \not\subseteq A_s^*$. Hence, we can extend A_s^* to $A_s^* := A_s^* \cup A_{b+c+e} = \{a, b, c, e, f\}$. Now, for the first identity, we have that $A_{a+b+c} = \{a, b, c\} \subseteq A_s^*$, but $A_d = \{d\} \not\subseteq A_s^*$, and thus we obtain $A_s^* := A_s^* \cup A_d = \{a, b, c, d, e, f\}$. This is the final saturated set since it cannot be further extended using the identities in G.

The following lemma is an easy consequence of the definition of saturation.

Lemma 2. *Let $A, B \subseteq C$. Then the following holds:*

$$A \subseteq A^*, \quad A^{**} = A^*, \quad A \subseteq B \Rightarrow A^* \subseteq B^*, \quad A^* \cup B^* \subseteq (A \cup B)^*.$$

Proposition 3. *Let $A_s, A_t \subseteq C$ be sets respectively representing the terms $s, t \in T_\Sigma(C)$. Then $s =_{ACUIG} t$ iff $A_s^* = A_t^*$. In particular this implies that the word problem for ACUIG is decidable in polynomial time.*

Proof. Decidability in polynomial time obviously follows from the equivalence in the first statement since the saturation A^* of a set $A \subseteq C$ can be computed in polynomial time.

To show the equivalence, first assume that $A_s^* = A_t^*$. To conclude from this that $s =_{ACUIG} t$, it is sufficient to show that saturation steps correspond to rewrite steps in $ACUIG$. Thus, assume that $l \in T_\Sigma(C)$ has the corresponding set A_l, and that $g_i = h_i$ is an identity in G such that $G_i \subseteq A_l$. Then l is of the form $l = g_i + l'$. We now have $l = g_i + l' =_{ACUIG} g_i + g_i + l' =_{ACUIG} h_i + l$, and the set corresponding to the term $h_i + l$ is $A_l \cup H_i$.

Second, assume that $A_s^* \neq A_t^*$. To show that this implies $s \neq_{ACUIG} t$, we construct a model $\mathcal{A}$ of $ACUIG$ in which this identity does not hold. As interpretation domain, we use all saturated sets, i.e., $\Delta := \{A^* \mid A \subseteq C\}$. The binary symbol $+$ is interpreted as union followed by saturation, i.e., $A^* + B^* := (A^* \cup B^*)^*$, $\mathbf{0}$ as $\emptyset^*$, and $c \in C$ as $\{c\}^*$. Given a term $u \in T_\Sigma(C)$

with corresponding set A_u, its interpretation in this algebra is A_u^*. This can easily be shown by induction on the structure of u, where the induction step uses the fact that

$$(A^* \cup B^*)^* = (A \cup B)^*, \tag{1}$$

which is an easy consequence of Lemma 2. Thus, $A_s^* \neq A_t^*$ implies that the terms s, t have different interpretations in $\mathcal{A}$. To show $s \neq_{ACUIG} t$, it is thus sufficient to show that $\mathcal{A}$ satisfies all identities of $ACUIG$. For the identities in $ACUI$ this is an easy consequence of (1) and the fact that set union is associative, commutative and idempotent and has $\emptyset$ as unit. Now consider an identity $g_i = h_i \in G$. When saturating the corresponding sets G_i and H_i, one can in a first step go both from G_i and from H_i to $G_i \cup H_i$ (unless this step is void due to an inclusion). Saturating further, one thus obtains identical saturated sets, which shows that g_i and h_i are interpreted in $\mathcal{A}$ by the same saturated set. □

Continuing our example, recall that the term $s = a + f$ has the saturated set $A_s^* = \{a, b, c, d, e, f\}$. It is easy to see that for $t = b + d + e$ saturation produces the sequence of

$$A_t = \{b, d, e\} \rightarrow \{a, b, c, d, e\} \rightarrow \{a, b, c, d, e, f\} = A_t^*,$$

where in the first step the identity $d = a + b + c$ is applied, and in the second the identity $b + c + e = f$. Thus, we have $A_s^* = A_t^*$, which shows that $s = a + f =_{ACUIG} b + d + e = t$.

Next, we introduce an algorithm that solves $ACUIG$-unification with linear constant restriction in polynomial time. Intuitively, it starts with a maximal substitution that respects the linear order $\prec$. Next, whenever an equation is not satisfied, that is, when an element appears on one side but not on the other, we trim the substitution, so that it no longer introduces this violation. Upon termination, the algorithm provides a solution if one exists, or outputs **Fail** otherwise. By a slight abuse of notation we assume that the substitutions σ considered in the algorithm actually map to sets of constants rather than ground terms. In addition, for a term $t \in T_\Sigma(X, C)$ we use $\sigma(t)$ to denote also the set corresponding to the term $\sigma(t)$.

Algorithm 1: Computation of unifier

```
Input: An ACUIG-unification problem with linear constant restriction $\prec$, as introduced
       in Definition 1.
Output: A unifier $\sigma : X \rightarrow 2^C$ or Fail.
Set $\sigma(x) := \{c \in C \mid c \prec x$ or $c$ occurs in $G\}$ for all $x \in X$
while some equation $s =^? t$ in $\Gamma$ is not satisfied by $\sigma$ do
    if there is a variable $x$ in $s$ such that $\sigma(x) \not\subseteq \sigma(t)^*$ or $y$ in $t$ such that $\sigma(y) \not\subseteq \sigma(s)^*$
     then
        Set $\sigma(x) := \sigma(x) \cap \sigma(t)^*$ for all variables $x$ in $s$
        Set $\sigma(y) := \sigma(y) \cap \sigma(s)^*$ for all variables $y$ in $t$
    else
        return Fail
    end
end
return $\sigma$
```

Before proving correctness of this algorithm, we give an example that illustrates how it tests for the existence of an $ACUIG$-unifier w.r.t. a linear constant restriction. Consider the system of equations

$$\Gamma = \{g + x_2 =^? a + x_1,\ b + x_1 =^? c + f + g,\ c + x_2 =^? a + c + e\},$$

the set of ground identities

$$G = \{a + b + c = d,\ b + c + e = f\}$$

considered in our previous examples, and the linear order

$$x_2 \prec g \prec x_1.$$

Note that g is the only *free* constant occurring in Γ, and thus it is the only constant occurring in this linear constant restriction. Also note that without $g \prec x_1$, the second equation of Γ would not be solvable. In addition, without G this second equation would not be solvable either: b belongs to the left-hand side, but could never belong to the right-hand side without additional ground identities.

The algorithm begins by setting

$$\sigma(x_1) := \{a, b, c, d, e, f, g\} \text{ and } \sigma(x_2) := \{a, b, c, d, e, f\}.$$

Next, the algorithm enters the while loop and picks in each iteration an equation that is not satisfied:

- The second equation is not satisfied by σ. In fact, we have that $\sigma(c+f+g)^* = \{b, c, e, f, g\}$, and hence $\sigma(x_1) \not\subseteq \sigma(c+f+g)^*$. The algorithm proceeds to set $\sigma(x_1) := \{a, b, c, d, e, f, g\} \cap \{b, c, e, f, g\} = \{b, c, e, f, g\}$.
- The third equation is not satisfied by σ. We have that $\sigma(a+c+e)^* = \{a, c, e\}$, and hence $\sigma(x_2) \not\subseteq \sigma(a+c+e)^*$. The algorithm proceeds to set $\sigma(x_2) := \{a, b, c, d, e, f\} \cap \{a, c, e\} = \{a, c, e\}$.
- The first equation is not satisfied by σ. We have that $\sigma(x_1) = \{b, c, e, f, g\} \not\subseteq \sigma(g+x_2)^* = \{a, c, e, g\}$. The algorithm proceeds to set $\sigma(x_1) := \{b, c, e, f, g\} \cap \{a, c, e, g\} = \{c, e, g\}$.

The algorithm then terminates since all equations are satisfied, and yields the substitution $\sigma = \{x_1 \mapsto c + e + g, x_2 \mapsto a + c + e\}$ as output.

Proposition 4. *Algorithm 1 terminates in polynomial time. If Γ has a unifier, then it provides a unifier as output, and otherwise it fails.*

Proof. Termination in polynomial time is an easy consequence of the fact that in each iteration of the while-loop, at least one constant is removed from the image of a variable, or the loop is exited.

Since the algorithm only returns a substitution if the while-loop is exited regularly, this substitution satisfies all the equations of Γ. It satisfies the linear constant restriction due to the fact that the original substitution satisfies it and that constants are only removed from, but never added to, the image of variables during the run of the algorithm. Consequently, if the algorithm returns a substitution, then this substitution is a unifier of Γ w.r.t. $\prec$. This shows that the algorithm must return **Fail** in case Γ has no unifier w.r.t. $\prec$.

To prove the completeness of the algorithm, assume that $\widehat{\sigma}$ is a unifier of Γ, and that the algorithm terminates during the rth iteration of the while-loop. Let $\sigma^{(0)}$ be the substitution σ before the first iteration of the while-loop. For $i \in \{1, \ldots, r-1\}$, let $\sigma^{(i)}$ be the substitution obtained at the end of the ith iteration of the while-loop.

We extend $\subseteq$ to substitutions in a natural way, by using pointwise comparison. We prove by induction on i that $\widehat{\sigma} \subseteq \sigma^{(i)}$, for all $i \in \{0, \ldots, r-1\}$. Since $\widehat{\sigma}$ satisfies the linear constant restriction, we have $\widehat{\sigma} \subseteq \sigma^{(0)}$. Let now $i \in \{0, \ldots, r-2\}$, and assume that we already know that $\widehat{\sigma} \subseteq \sigma^{(i)}$. We must prove $\widehat{\sigma} \subseteq \sigma^{(i+1)}$.

Since the algorithm does not exit the while-loop at this stage, there is an equation $s =^? t$ in Γ that is not satisfied by $\sigma^{(i)}$. In addition, since the algorithm does not fail at iteration i, there exists a variable x in s such that $\sigma(x) \not\subseteq \sigma(t)^*$ or y in t such that $\sigma(y) \not\subseteq \sigma(s)^*$. Clearly, for every $x \in X$ that does not appear in this equation, we have $\widehat{\sigma}(x) \subseteq \sigma^{(i)}(x) = \sigma^{(i+1)}(x)$. Let now x be a variable occurring in s (variables in t can be treated analogously). To prove that $\widehat{\sigma}(x) \subseteq \sigma^{(i+1)}(x)$, it suffices to prove that $\widehat{\sigma}(x) \subseteq \sigma^{(i)}(x)$ and that $\widehat{\sigma}(x) \subseteq \sigma^{(i)}(t)^*$. The first statement is true by the induction hypothesis. Now, we have

$$\widehat{\sigma}(x) \overset{(1)}{\subseteq} \widehat{\sigma}(s) \overset{(2)}{\subseteq} \widehat{\sigma}(s)^* \overset{(3)}{=} \widehat{\sigma}(t)^* \overset{(4)}{\subseteq} \sigma^{(i)}(t)^*,$$

where (1) holds because x occurs in s, (2) by Lemma 2, (3) because $\widehat{\sigma}$ is a unifier of Γ, and (4) by Lemma 2 since $\widehat{\sigma} \subseteq \sigma^{(i)}$. This finishes the induction proof.

Therefore, we now know that $\widehat{\sigma} \subseteq \sigma^{(r-1)}$. There are two possible reasons for the algorithm terminating in the rth iteration. Either the while-loop is exited regularly or the algorithm returns **Fail**. In the first case, $\sigma^{(r-1)}$ is a unifier and the algorithm returns this substitution.

It remains to show that the second case cannot occur. In this case, we have $\sigma^{(r-1)}(s)^* \neq \sigma^{(r-1)}(t)^*$ for some equation $s =^? t$ in Γ, but $\sigma^{(r-1)}(x) \subseteq \sigma^{(r-1)}(t)^*$ for all variables x in s and $\sigma^{(r-1)}(y) \subseteq \sigma^{(r-1)}(s)^*$ for all variables y in t. This can only be the case if there is a constant $c \in C$ such that c occurs in s, but $c \notin \sigma^{(r-1)}(t)^*$; or c occurs in t, but $c \notin \sigma^{(r-1)}(s)^*$. We show that this is impossible.

Thus assume that c occurs in s (the case where c occurs in t can be treated symmetrically). We have

$$c \overset{(1)}{\in} \widehat{\sigma}(s) \overset{(2)}{\subseteq} \widehat{\sigma}(s)^* \overset{(3)}{=} \widehat{\sigma}(t)^* \overset{(4)}{\subseteq} \sigma^{(r-1)}(t)^*,$$

where (1) holds since c occurs in s, (2) by Lemma 2, (3) since $\widehat{\sigma}$ is a unifier of Γ, and (4) by Lemma 2 since $\widehat{\sigma} \subseteq \sigma^{(r-1)}$. □

The following theorem is an immediate consequence of Proposition 4.

Theorem 5. *Let ACUI be the equational theory that states that the binary function symbol $+$ is associative, commutative, and idempotent, and has the constant symbol **0** as unit, and let ACUIG be an extension of ACUI by finitely many ground identities built using $+$, **0**, and additional constant symbols. Then the ACUIG-unification problem with linear constant restrictions is decidable in polynomial time.*

3 General $ACUIG$-unification

General ACUIG-unification problems differ from the ones we have considered until now in that the terms used in Γ may contain “free” function symbols, i.e., function symbols not occurring in the identities of *ACUIG*. For example, $\{f(x + a, a + b) =^? f(b + y, x)\}$ is such a general *ACUIG*-unification problem since it contains the additional function symbol f that does not occur in the identities of *ACUIG*.

The following was proved by Baader and Schulz [4] and provides an upper bound for general *ACUIG*-unification.

Theorem 6 ([4]). *If solvability of E_i-unification problems with linear constant restrictions is decidable in NP for $i = 1, 2$, then unifiability in the combined theory $E_1 \cup E_2$ is also decidable in NP.*

In particular, this implies that, if E-unification with linear constant restriction is decidable in NP, the same is true for general unification, by choosing as the second theory the empty theory.

Theorem 7. *For every finite set of ground identities G, general $ACUIG$-unification is NP-complete.*

Proof. Membership in NP is an immediate consequence of Theorem 5 together with Theorem 6.

NP-hardness can be shown by the same reduction from the set-matching problem as used in [6] to show that general ACI-unification is NP-hard. To be more precise, this reduction yields ACI-unification problems of the form

$$\Gamma = \{g(s_1) + \ldots + g(s_m) =^? g(t_1) + \ldots + g(t_m)\},$$

where $+$ is an associative, commutative, and idempotent function symbol, g is a unary free function symbol and the terms $s_1, \ldots, s_m, t_1, \ldots, t_n$ contain only free function symbols and variables. The presence of a unit and of ground identities in $ACUIG$ do not change solvability of such problems compared to ACI since

- in the top-level sum the additional identities cannot be used due to the fact that all terms on this level start with the free function symbol g;
- while the variables occurring in the terms $g(s_1), \ldots, g(s_m), g(t_1), \ldots, g(t_n)$ may be replaced by terms containing $+$ and constant symbols from G, these "alien" subterms can be abstracted away by free constants.

This shows that such a problem Γ is solvable modulo ACI iff it is solvable modulo $ACUIG$, which completes our proof of NP-hardness of general $ACUIG$-unification. □

4 Conclusion

We have shown that $ACUIG$-unification with linear constant restrictions is decidable in P, and general $ACUIG$-unification is NP-complete. Note, however, that according to our definition of $ACUIG$ this result holds for a single $ACUI$-symbol $+$ and ground identities G built using only $+$ and free constant symbols. Due to the combination results of Baader and Schulz [4], we can also deal with several $ACUI$-symbols $+_1, \ldots, +_n$ and sets of ground identities $G_1, \ldots, G_n$, where the identities in G_i are built using only $+_i$ and free constant symbols not occurring in any of the other sets G_j $(i \neq j)$. The combination results show that unification in the union of such theories can be decided in NP. However, this result cannot deal with situations where the ground identities share constants, or contain several $ACUI$-symbols, or contain free function symbols. It is an open problem whether unification in such "mixed" theories remains decidable. It is only known from Marché's results [8] that the word problem is decidable in this setting, and it would be interesting to see whether the same is true for unification.

Motivated by the applications in Description Logics mentioned in the introduction, we also intend to investigate what effect adding ground identities to extensions of $ACUI$ has on the unification problem.

References

[1] F. Baader, S. Borgwardt, and B. Morawska. Extending unification in $\mathcal{EL}$ towards general TBoxes. In *Proc. of the 13th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2012)*, pages 568–572. AAAI Press/The MIT Press, 2012.

[2] F. Baader and B. Morawska. Unification in the description logic $\mathcal{EL}$. In R. Treinen, editor, *Proc. of the 20th Int. Conf. on Rewriting Techniques and Applications (RTA 2009)*, volume 5595 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2009.

[3] F. Baader and P. Narendran. Unification of concept terms in description logics. *J. of Symbolic Computation*, 31(3):277–305, 2001.

[4] F. Baader and K. U. Schulz. General A- and AX-unification via optimized combination procedures. In H. Abdulrab and J.-P. Pécuchet, editors, *Word Equations and Related Topics*, volume 677 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 1993.

[5] F. Baader and W. Snyder. Unification theory. In J. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 447–533. Elsevier Science Publishers, 2001.

[6] D. Kapur and P. Narendran. NP-completeness of the set unification and matching problems. In J. H. Siekmann, editor, *Proceedings of the 8th International Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 489–495, Oxford, UK, 1986. Springer.

[7] D. Kapur and P. Narendran. Complexity of unification problems with associative-commutative operators. *Journal of Automated Reasoning*, 9(2):261–288, Oct 1992.

[8] C. Marché. Normalized rewriting: an alternative to rewriting modulo a set of equations. *Journal of Symbolic Computation*, 21(3):253 – 288, 1996.

32nd International Workshop on Unification (UNIF 2018), Informal Proceedings
Oxford, 7th July, 2018

Knowledge Problems in Equational Extensions of Subterm Convergent Theories

(Extended Abstract)

Serdar Erbatur[1], Andrew M. Marshall[2], and Christophe Ringeissen[3]*

[1] IMDEA Software Institute (Spain)
serdar.erbatur@imdea.org
[2] University of Mary Washington (USA)
marshall@umw.edu
[3] Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
Christophe.Ringeissen@loria.fr

Abstract

We study decision procedures for two knowledge problems critical to the verification of security protocols, namely the intruder deduction and the static equivalence problems. These problems can be related to particular forms of context matching and context unification. Both problems are defined with respect to an equational theory and are known to be decidable when the equational theory is given by a subterm convergent term rewrite system. In this note we extend this to consider a subterm convergent equational term rewrite system defined modulo an equational theory, like Commutativity or Associativity-Commutativity. We show that for certain classes of such equational theories, namely the shallow classes, the two knowledge problems remain decidable.

1 Introduction

Verifying the security of protocols requires the development of specific decision procedures to reason about the knowledge of an intruder. Two important measures of this knowledge are *(intruder) deduction* [17,18] and *static equivalence* [2]. The deduction problem is the question of whether an intruder, given his deductive capability and a sequence of messages representing their knowledge, can obtain some secret. This is a critical measure of the capability of the protocol to maintain secrets. Deducibility is needed for many questions about the security of protocols. However, there are some questions for which we need to be able to decide more than deducibility. For some protocols, in addition to deducibility, we would like to determine whether an intruder can distinguish between different runs of the protocol. For example, in protocols which attempt to transmit encrypted votes we would like to know if, to the attacker, two different votes are indistinguishable. Static equivalence measures this property.

*This work has received funding from the European Research Council (ERC) under the H2020 research and innovation program (grant agreement No 645865-SPOOC).

Much work has gone into investigating and developing decision procedures for the deduction and the static equivalence problems [2,6,8,10,12]. In this work the security protocols are often represented by equational theories with the theories of interest usually defined as unions of several simpler sub-theories. In this paper, we focus on decision procedures for the deduction problem and the static equivalence problem in equational theories $E = E_1 \cup E_2$ where E_1 and E_2 are possibly non-disjoint. Until now, the following scenarios have been successfully investigated: E_1 is given by a subterm convergent term rewrite system, and E_2 is empty [2]; E_1 and E_2 are disjoint [9]; E_1 and E_2 share only constructors [12]. In this paper, we consider the case where E_1 is given by a term rewrite system which is both subterm and convergent modulo E_2. We then show that the methods of [2] can be extended to subterm convergent rewrite systems for a significant class of E_2 theories.

2 Preliminaries

We assume the reader is familiar with equational unification and term rewrite systems. We review some critical definitions below but a more complete overview can be found in [5].

A finite convergent *term rewrite system* (TRS) R is said to be *subterm convergent* if for any $l \to r \in R$, r is either a strict subterm of l or a ground term. An equational theory is *subterm convergent* if it is presented by a subterm convergent TRS.

The size of a term t is denoted by $|t|$ and defined in the usual way as follows: $|f(t_1, \dots, t_n)| = 1+\Sigma_{i=1}^{n}|t_i|$ if f is a n-ary function symbol with $n \geq 1$, $|c| = 1$ if c is a constant, and $|x| = 0$ if x is a variable. The size of a TRS R is denoted by $|R|$ and defined as follows: $|R| = \max_{\{l \to r \in R\}} |l|$. Since a variable cannot occur as the left-hand side of any rule in R, we have that $|R| \geq 1$ for any non-empty TRS R. When R is empty, we define $|R| = 1$.

Equational Theories. Let us introduce the different classes of theories considered in the paper. An equational theory E is *finite* if for each term t, there are only finitely many terms s such that $t =_E s$. Matching in finite theories is finitary. A sufficient condition to get a finite theory is to assume that E is permutative. An equational theory E is *permutative* if for each axiom $l = r$ in E, l and r contain the same symbols with the same number of occurrences. Well-known theories such as Associativity (A), Commutativity (C), and Associativity-Commutativity (AC) are permutative theories. Unification in permutative theories is undecidable in general [19].

A theory E is *syntactic* if it has a finite *resolvent presentation* S, that is a finite set of equational axioms S such that each equality $t =_E u$ has an equational proof $t \leftrightarrow_S^* u$ with at most one step $\leftrightarrow_S$ applied at the root position. A theory E is *shallow* if variables can only occur at a depth at most 1 in axioms of E. Shallow theories are syntactic theories for which unification is finitary [7]. The theory AC is permutative and syntactic, while C is permutative and shallow.

Notions of Knowledge. The applied pi calculus and frames are used to model attacker knowledge [3]. In this model, the set of messages or terms which the attacker knows, and which could have been obtained from observing one or more protocol sessions, are the set of terms in $Ran(\sigma)$ of the frame $\phi = \nu\tilde{n}.\sigma$, where σ is a substitution ranging over ground terms. We also need to model cryptographic concepts such as nonces, keys, and publicly known values. We do this by using names, which are essentially free constants. Here also, we need to track the names which the attacker knows, such as public values, and the names which the attacker does not know a priori, such as freshly generated nonces. $\tilde{n}$ consists of a finite set of restricted names, these names represent freshly generated names which remain secret from the attacker. The set of names occurring in a term t is denoted by $fn(t)$.

Definition 1 (Deduction Problem [2])**.** *Let $\phi = \nu\tilde{n}.\sigma$ be a frame, and t a ground term. We say that t is* deduced from ϕ modulo E*, denoted by $\phi \vdash_E t$, if there exists a term s such that $s\sigma =_E t$ and $fn(s) \cap \tilde{n} = \emptyset$. The term s is called a* recipe *of t in ϕ modulo E.*

Another form of knowledge is the ability to tell if two frames are *statically equivalent* modulo E, sometimes also called *indistiguishability.*

Definition 2 (Static Equivalence [2])**.** *Two terms s and t are* equal *in a frame $\phi = \nu\tilde{n}.\sigma$ modulo an equational theory E, denoted $(s =_E t)\phi$, iff $s\sigma =_E t\sigma$, and $\tilde{n} \cap (fn(s) \cup fn(t)) = \emptyset$. Two frames $\phi = \nu\tilde{n}.\sigma$ and $\psi = \nu\tilde{n}.\tau$ are* statically equivalent modulo E*, denoted as $\phi \approx_E \psi$, if $Dom(\sigma) = Dom(\tau)$ and for all terms s and t, we have $(s =_E t)\phi$ iff $(s =_E t)\psi$.*

Both deduction and static equivalence are known to be decidable in subterm convergent theories [2]. In the following, we lift this result to term rewrite systems that are subterm convergent modulo some equational theory.

3 Subterm Equational Convergent TRS

Consider $(\Sigma, E) = (\Sigma_1 \cup \Sigma_2, R_1 \cup E_2)$ where $(\Sigma_1 \cup \Sigma_2, R_1)$ is a TRS modulo a finite theory (Σ_2, E_2) (for instance $\Sigma_2 = \{+\}$ and $E_2 = AC(+)$). The rewrite relation of R_1 modulo E_2 is defined as usual: $s \to_{R_1,E_2} t$ if there exist some position p in s, some rule $l \to r \in R_1$ and a substitution μ such that $s|_p =_{E_2} l\mu$ and $t = s[r\mu]_p$. We assume that $\to_{R_1,E_2}$ is convergent modulo E_2 [15]. This implies the uniqueness of normal forms modulo E_2 and the decidability of the word problem modulo E: for any terms s and t, we have $s =_E t$ iff $(s\downarrow_{R_1,E_2}) =_{E_2} (t\downarrow_{R_1,E_2})$. In the following, we say that a term or a substitution is normalized if it is normalized w.r.t $\to_{R_1,E_2}$. A frame $\phi = \nu\tilde{n}.\sigma$ is said to be normalized if σ is normalized.

Definition 3. *Let Σ_1 and Σ_2 be two disjoint signatures, and (Σ_2, E_2) a finite theory. A* subterm E_2-convergent *TRS $(\Sigma_1 \cup \Sigma_2, R_1)$ is a TRS such that $\to_{R_1,E_2}$ is convergent modulo E_2 and for any $l \to r$ in R_1, l is not Σ_2-rooted and r is a strict subterm of l or a ground term.*

Example 1. *The following TRSs are subterm $AC(+)$-convergent:*

$\{occ(x+k,k) \to ok\}$	$\{rm(x+k,k) \to x\}$
$\{dec(enc(x,k+y),k) \to x\}$	$\{dec(enc(x,k),k+y) \to x\}$

In the case of subterm convergent TRSs (modulo the empty theory), the decision procedure for the deduction problem computes deducible terms among the set of subterms occurring in the frame. When considering a non-empty theory E_2, we have to introduce an extended notion of subterm to capture the fact that matching modulo E_2 is now performed when applying a rewrite step modulo E_2.

In the rest of this section we assume that E_2 is both permutative and syntactic. While this may seem somewhat restrictive it allows for the consideration of theories such as AC and C which are found in a large number of security protocols. Both AC and C are indeed syntactic theories [16].

Given a term t, $St(t)$ is the finite set of terms in t inductively defined as follows:

$$St(t) = \{t' \mid t' =_{E_2} t\} \cup \left\{ t' \;\middle|\; \begin{array}{l} t' \in St(x_i\sigma), f(x_1,\ldots,x_m)\sigma =_{E_2} t, f \in \Sigma_1 \cup \Sigma_2 \\ x_1,\ldots,x_m \text{ are pairwise disjoint variables} \end{array} \right\}$$

This definition is well-founded since E_2 is permutative. There exists a mutation-based E_2-matching algorithm [11] since E_2 is syntactic, and so $St(t)$ is computable.

For a set of terms T, $St(T) = \bigcup_{t \in T} St(t)$, and for a substitution σ, $St(\sigma) = St(Ran(\sigma))$.

Proposition 1. *For any terms t, t', $t =_{E_2} t'$ implies $St(t) = St(t')$, and for any position p in t, $St(t|_p) \subseteq St(t)$.*

The following result states that we cannot generate a new term outside $St(t)$ by rewriting terms in $St(t)$ (except the ground right-hand sides of R_1).

Lemma 1. *If $l\sigma =_{E_2} t$, then for any position p of l, $(l|_p)\sigma \in St(t)$.*

Proof. By structural induction on l.

If l is a variable, this is trivial since the only possible position is ϵ and $l|_\epsilon = l$.

Assume l is a term $f(l_1, \ldots, l_m)$ and σ is a substitution such that $f(l_1, \ldots, l_m)\sigma =_{E_2} t$.

If there is an equational step at the root position, then there exist some terms $g_1, \ldots, g_m$ such that $l_1\sigma =_{E_2} g_1, \ldots, l_m\sigma =_{E_2} g_m$ and $f(g_1, \ldots, g_m) =_{E_2} t$. By definition of $St(t)$ and Proposition 1, the terms $g_1, \ldots, g_m$ are in $St(t)$, and so $l_1\sigma, \ldots, l_m\sigma \in St(t)$.

If there is no equational step at the root position, then t is of the form $f(t_1, \ldots, t_m)$ and $l_1\sigma =_{E_2} t_1, \ldots, l_m\sigma =_{E_2} t_m$. By definition of $St(t)$ and Proposition 1, the terms $t_1, \ldots, t_m$ are in $St(t)$, and so $l_1\sigma, \ldots, l_m\sigma \in St(t)$. □

4 Decision Procedures

From now on, we assume that E_2 is a shallow permutative theory, e.g., Commutativity.

Deduction. The decision procedure for the deduction problem requires the computation of some finite deducible terms defining the so-called *completion* of a given frame.

Definition 4. *Let $\phi = \nu\tilde{n}.\sigma$ be a normalized frame. The set of* local deducible terms in ϕ *is the smallest set D such that:*

- *$Ran(\sigma) \subseteq D$,*
- *if $t_1, \ldots, t_n \in D$ and $f(t_1, \ldots, t_n) \in St(\sigma)$ then $f(t_1, \ldots, t_n) \in D$,*
- *if $t \in D$, $t' \in St(\sigma)$, $t =_{E_2} t'$, then $t' \in D$,*
- *if there is a root reduction $s[\bar{r}] \rightarrow^{\epsilon}_{R_1,E_2} t$ where $|s| \leq |R_1|$, $fn(s) \cap \tilde{n} = \emptyset$, $\bar{r} \in D$ and $t \in St(\sigma)$, then $t \in D$.*

Let $\sigma_* = \sigma\{\chi_u \mapsto u \mid u \in D \backslash Ran(\sigma)\}$ where χ_u is a fresh variable. The frame $\phi_* = \nu\tilde{n}.\sigma_*$ is called the completion of ϕ with respect to R_1. *The* recipe substitution *of ϕ is $\zeta_\phi = \{\chi_u \mapsto \zeta_u \mid u \in D\backslash Ran(\sigma)\}$ where ζ_u denotes an arbitrary recipe of u w.r.t. ϕ.*

The decision procedure is based on the following reduction lemma, using the facts that the completion is computable and the deduction problem is decidable in the empty equational theory.

Lemma 2. *Let $E = R_1 \cup E_2$ where R_1 is any subterm E_2-convergent TRS and E_2 is any shallow permutative theory. For any normalized frame ϕ and any normalized term t, we have that $\phi \vdash_E t$ if and only if $\phi_* \vdash t$.*

Static Equivalence. The decision procedure for the static equivalence is based on the computation of small equalities bounded by the size of R_1.

Definition 5. *Let $\phi = \nu\tilde{n}.\sigma$ be a normalized frame. The set $Eq(\phi)$ is the set of equalities $t\zeta_\phi = t'\zeta_\phi$ such that $(t\zeta_\phi)\sigma =_E (t'\zeta_\phi)\sigma$ where t, t' are Σ-terms, $(fn(t) \cup fn(t')) \cap \tilde{n} = \emptyset$, $|t|, |t'| \leq |R_1|$. Given any frame $\psi = \nu\tilde{n}.\tau$, the fact that $t\tau =_E t'\tau$ for any $t = t' \in Eq(\phi)$ is denoted by $\psi \models Eq(\phi)$.*

To get a decision procedure, it remains to show that checking small equalities defined by Eq are sufficient to prove the static equivalence of the two input frames. Note that the check of each of these equalities is effective since the E-equality is decidable.

Lemma 3. *Let $E = R_1 \cup E_2$ where R_1 is any subterm E_2-convergent TRS and E_2 is any shallow permutative theory. For any normalized frames ϕ and ψ, we have that $\phi \approx_E \psi$ iff $\psi \models Eq(\phi)$ and $\phi \models Eq(\psi)$.*

Main result. According to the above reduction lemmas, we get the following result.

Theorem 1. *Let $E = R_1 \cup E_2$ where R_1 is any subterm E_2-convergent TRS and E_2 is any shallow permutative theory. Then, deduction and static equivalence are decidable in E.*

To prove both reduction lemmas (Lemma 2 and Lemma 3) and so Theorem 1, we reuse the same approach as in [1, 2] by applying two technical lemmas.

The first lemma in the appendix of [1] can be generalized as follows.

Lemma 4. *Let $E = R_1 \cup E_2$ where R_1 is any subterm E_2-convergent TRS and E_2 is any shallow permutative theory. For any terms s and t satisfying the name restriction, if $s\phi_* =_{E_2} t\phi_*$ and $\psi \models Eq(\phi)$ then $(s\zeta_\phi)\psi =_E (t\zeta_\phi)\psi$.*

Then, the second lemma in the appendix of [1] is generalized in the following way.

Lemma 5. *Let $E = R_1 \cup E_2$ where R_1 is any subterm E_2-convergent TRS and E_2 is any shallow permutative theory. For any term s satisfying the name restriction and for any term t such that $s\phi_* \rightarrow_{R_1,E_2} t$, there exists a term u satisfying the name restriction such that $t =_{E_2} u\phi_*$ and for any frame ψ such that $\psi \models Eq(\phi)$, $(s\zeta_\phi)\psi =_E (u\zeta_\phi)\psi$.*

The proofs of Lemma 4 and Lemma 5 can be found in [13]. The assumption that E_2 is shallow permutative allows us to get simple proofs.

We are working on generalizing Theorem 1 to syntactic permutative theories E_2 like for instance Associativity-Commutativity. In this general case, the related reduction lemmas for the deduction problem and the static equivalence should be more complicated to express. Indeed, we may have to integrate a deduction procedure modulo E_2 in the construction of the completion, and a static equivalence procedure modulo E_2 to get a reduction lemma for static equivalence in $R_1 \cup E_2$.

5 Related Work and Conclusion

The intruder deduction problem corresponds to the general cap problem studied in [4]. Among other results, it is shown in [4] that the general cap problem is in NP for *dwindling* convergent rewrite systems, which are indeed subterm convergent theories. The NP procedure is given by a saturation procedure used to complete the knowledge given by the input frame. In the conclusion of [4], the extension to AC-rewrite systems is mentioned as an interesting future work.

Currently we assume in Definition 3 that the Σ_2-symbols are constructors, i.e., not appearing at the root of the left-hand sides of the rewrite system. However, this appears to be more restrictive than needed. Indeed, it should be possible to remove this restriction and consider a more relaxed definition where the Σ_2-symbols are not necessarily constructors. This would allow us to solve the deduction and static equivalence problem in a larger class of theories. For example, we could then consider the theory of Abelian Pre-Group (APG) defined by the following C-convergent TRS:

$$R_{APG} = \{x * e \rightarrow x,\ x * i(x) \rightarrow e,\ i(i(x)) \rightarrow x,\ i(e) \rightarrow e\}$$

where $C = \{x * y = y * x\}$. In [20], $APG = R_{APG} \cup C$ was considered as an approximation to deal with unification in homomorphic encryption over Abelian groups. Theorem 1 would then allow us to also solve the problems of deduction and static-equivalence in APG.

The next step of our work is to go beyond the class of shallow permutative theories, in order to take into account a larger class including AC. Due to the potential interest of AC in protocol analysis, it is useful to be able to handle some AC-rewrite systems and to study the AC-extension of saturation procedures that have been developed for the intruder deduction problem, the static equivalence [6], and the static inclusion [14].

Another challenging problem is to investigate the equational extension of the combination procedure developed in [12] for the deduction and the static equivalence in unions of theories sharing absolutely free constructors. This would permit us to consider shared AC-constructors.

Acknowledgments: We would like to thank Véronique Cortier for helpful comments and discussions.

References

[1] Martín Abadi and Véronique Cortier. Deciding knowledge in security protocols under equational theories. Research Report RR-5169, INRIA, 2004.

[2] Martín Abadi and Véronique Cortier. Deciding knowledge in security protocols under equational theories. *Theor. Comput. Sci.*, 367(1-2):2–32, 2006.

[3] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'01, pages 104–115, New York, NY, USA, 2001. ACM.

[4] Siva Anantharaman, Paliath Narendran, and Michaël Rusinowitch. Intruders with caps. In Franz Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4533 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2007.

[5] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[6] Mathieu Baudet, Véronique Cortier, and Stéphanie Delaune. YAPA: A generic tool for computing intruder knowledge. *ACM Trans. Comput. Log.*, 14(1):4, 2013.

[7] Hubert Comon, Marianne Haberstrau, and Jean-Pierre Jouannaud. Syntacticness, cycle-syntacticness, and shallow theories. *Inf. Comput.*, 111(1):154–191, 1994.

[8] Bruno Conchinha, David A. Basin, and Carlos Caleiro. FAST: an efficient decision procedure for deduction and static equivalence. In Manfred Schmidt-Schauß, editor, *Proceedings of RTA 2011,*

Novi Sad, Serbia, volume 10 of *LIPIcs*, pages 11–20. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[9] Véronique Cortier and Stéphanie Delaune. Decidability and combination results for two notions of knowledge in security protocols. *Journal of Automated Reasoning*, 48(4):441–487, 2010.

[10] Ştefan Ciobâcă, Stéphanie Delaune, and Steve Kremer. Computing knowledge in security protocols under convergent equational theories. *J. Autom. Reasoning*, 48(2):219–262, 2012.

[11] Serdar Erbatur, Deepak Kapur, Andrew M. Marshall, Paliath Narendran, and Christophe Ringeissen. Unification and matching in hierarchical combinations of syntactic theories. In Carsten Lutz and Silvio Ranise, editors, *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland. Proceedings*, volume 9322 of *LNCS*, pages 291–306. Springer, 2015.

[12] Serdar Erbatur, Andrew M. Marshall, and Christophe Ringeissen. Notions of knowledge in combinations of theories sharing constructors. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, Proceedings*, volume 10395 of *LNCS*, pages 60–76. Springer, 2017.

[13] Serdar Erbatur, Andrew M. Marshall, and Christophe Ringeissen. Computing knowledge in equational extensions of subterm convergent theories. Available at `https://hal.inria.fr`, 2018.

[14] Kimberly A. Gero. *Deciding Static Inclusion for Delta-strong and Omega Delta-strong Intruder Theories: Applications to Cryptographic Protocol Analysis.* PhD thesis, State University of New York aty Albany, 2015.

[15] Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM J. Comput.*, 15(4):1155–1194, 1986.

[16] C. Kirchner and F. Klay. Syntactic theories and unification. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 270–277, Jun 1990.

[17] Jonathan Millen and Vitaly Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, CCS'01, pages 166–175, New York, NY, USA, 2001. ACM.

[18] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Computer Security*, 6:85128, 1998.

[19] Manfred Schmidt-Schauß. Unification in permutative equational theories is undecidable. *J. Symb. Comput.*, 8(4):415–421, 1989.

[20] Fan Yang, Santiago Escobar, Catherine Meadows, José Meseguer, and Paliath Narendran. Theories of homomorphic encryption, unification, and the finite variant property. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*, PPDP '14, pages 123–133, New York, NY, USA, 2014. ACM.

UNIFICATION BASED ON GENERALIZED EMBEDDING

PETER SZABO
KURT-SCHUMACHER-STR. 13, D-75180 PFORZHEIM
JÖRG SIEKMANN
SAARLAND UNIVERSITY/DFKI
STUHLSATZENHAUSWEG, D-66123 SAARBRÜCKEN

Abstract. In this paper we introduce a concept of a minimal and complete set of E-unifiers $\lambda U\Sigma_E(\Gamma)$ for a unification problem Γ, based on homeomorphic embedding modulo an equational theory E. We propose a definitional framework based on notions and definitions of standard unification theory of first order terms extended to the (homeomorphic) embedding order modulo E. The main result is that the set $\lambda U\Sigma_E(\Gamma)$ always exists for a finite signature Σ and it is always finite.

1. Introduction

Ordering is a well established concept in mathematics and it plays an important role in many areas of computer science too. *Quasi orderings* (*qo*) and most noteably *well founded quasi orderings* (*wfo)* and *well quasi orderings (wqo)* in particular are of great general interest, see [13]. Probably the most popular application within our own field is the use of certain quasi orders and well quasi orders on first order terms to prove the termination of rewriting rules, see [3, 4].

In the theory of E-unification of terms based on a signature Σ and an equational theory E, the set $\mathcal{U}\Sigma_E(\Gamma)$ denotes the set of all E-unifiers of a unification problem Γ. Of great interest is now to find a complete and minimal subset of $\mathcal{U}\Sigma_E(\Gamma)$, denoted as $\mu\mathcal{U}\Sigma_E(\Gamma)$, from which all other E-unifiers can be obtained.

Equality on terms induced by the equational theory E will be denoted as $=_E$ and the *E-subsumption order on terms* is denoted as $\leq_E$. So, if there are two unifiers τ and σ for the terms s and t, such that $s\tau =_E t\tau$ and $s\sigma =_E t\sigma$ and there is a substitution λ, such that $\tau =_E \sigma\lambda$, then τ is an instance of σ, or σ *subsumes* τ, denoted $\sigma \leq_E \tau$. This led to the notion of *a most general E-unifier* (mgu), that is an E-unifier, which is not an instance of any other E-unifier. The set of most general unifiers is denoted as $\mu\mathcal{U}\Sigma_E(\Gamma)$ and every E-unifier is E-subsumed by some element of $\mu\mathcal{U}\Sigma_E(\Gamma)$, that is, it can be obtained by instantiaton in an automated reasoning process, such as *resolution* [17]. Often we shall drop the E in E-unifiers if it is understood from the context.

To illustrate the role of orderings in E-unification, consider the equational theory A for free semigroups with the axiom of associativity for terms built over a binary function symbol f with $A = \{f(x, f(y, z)) = f(f(x, y), z)\}$. This is also known as the word (or string) algebra and the notation is that of words (strings), where we just drop the function symbol f and have concatenation of symbols.
For example the string unification problem $\Gamma_1 = \{ax =^? xa\}$ has most general unifiers of the form $\sigma_n = \{x \mapsto a^n : n \geqq 1\}$. Because the σ_n are ground substitutions, they are incomparable with respect to the subsumption order, so $\mu\mathcal{U}\Sigma_A(\Gamma_1) = \{\sigma_n : n \geqq 1\}$ is an infinite set and therefore Γ is of unification type *infinitary*. Furthermore, since the subsumption order is not a well quasi order, there are equational theories such that the set of mgus does not exist (see[2][18][9]).

In order to address these problems, we proposed the *encompassment* of terms (see e.g.[11]) to be generalized to the notion of encompassment of unifers and introduced the notion of an

Key words and phrases. Universal algebra, equational theory, ordering, unification theory, E-unification, essential unification, (homeomorphic) embedded E-unifiers, pure E-unifier.

essential unifier. We say σ is encompassed by τ, $\sigma \sqsubseteq_E \tau$, iff each domain variable x of τ is also a domain variable of σ and $x\tau$ has an instance of $x\sigma$ as a subterm (modulo E). E-unifiers, which do not encompass any other unifier are then called *essential* unifiers and the complete set of essential unifiers is denoted as $e\mathcal{U}\Sigma_E(\Gamma)$ for a unification problem Γ. If $\mu\mathcal{U}\Sigma_E(\Gamma)$ exists, we have $e\mathcal{U}\Sigma_E(\Gamma) \subseteq \mu\mathcal{U}\Sigma_E(\Gamma)$, that is, the encompassment order generalizes the subsumption order and there are even cases where an E-unification problem with an infinite set of mgus reduces to a finite set of essential unifiers [10, 18]. Moreover it can happen, that an equational theory E, for which $\mu\mathcal{U}\Sigma_E(\Gamma)$ does not exist may have a minimal and complete set of essential unifiers $e\mathcal{U}\Sigma_E(\Gamma)$.

For example the unification type of Γ_1 from above changes drastically using the encompassment order: the *essential* unifier $\sigma_1 = \{x \mapsto a\}$ encompasses all the other most general unifiers $\sigma_n = \{x \mapsto a^n\}$, $n > 1$, because $\sigma_1 \sqsubseteq_A \sigma_n$, $n > 1$. More precisely encompassment allows the decomposition $\sigma_n = \lambda_n\sigma_1$, where $\lambda_n = \{x \mapsto a^n x\}$, $n \geqq 0$. So the minimal and complete set of essential unifiers for Γ_1 is $e\mathcal{U}\Sigma_E(\Gamma_1) = \{\sigma_1\}$, that is, it is unitary instead of infinitary as it is under the subsumption ordering.

Nevertheless there are still essentially infinitary string unification problems, as the following example shows. Let $\Gamma_2 = \{xby =^? ayayb\}$ be the string unification problem, which has $eU\Sigma_A(\mathit{\Gamma}_2) = \{\{x \mapsto ab^n a, y \mapsto b^n\} : n > 0\}$ as its minimal and complete set of essential unifiers. The unifiers are incomparable with respect to encompassment, because $ab^n a$ can not be a substring of $ab^m a$ for $m \neq n$. Forthermore, as the encompassment order on unifying substitutions is not a wqo, unfortunately again, there are theories with a solvable unification problem Γ, for which $e\mathcal{U}\Sigma_E(\Gamma)$ does not exist (see [2][8][18]).

This paper deals with a third approach, the extension of the well known *homeomorphic embedding of terms to a homeomorphic embedding modulo E of terms* (also used in [1] with a different definition) and to *a homeomorphic embedding modulo E of substitutions*, called E-embedding of terms or substitutions respectively. Informally, the homeomorphic embedding of terms is understood as follows: Let s and t be terms, then s is syntactically embedded into t, denoted as $s \trianglelefteq t$ iff s=t, or $s \trianglelefteq t_i$ for $t = f(t_1, ..., t_n)$ and some i or $s_i \trianglelefteq t_i$ for $s = f(s_1, ..., s_n)$ and all i. For example $f(x, b) \trianglelefteq \boldsymbol{f}(g(a, \boldsymbol{x}), f(x, \boldsymbol{b}))$ and also $f(x, b) \trianglelefteq \boldsymbol{f}(f(a, h(\boldsymbol{x})), f(\boldsymbol{b}, a))$ and $f(a, x) \trianglelefteq \boldsymbol{f}(g(\mathbf{a}, b), \boldsymbol{x})$, but $f(a, b) \ntrianglelefteq \boldsymbol{f}(g(\boldsymbol{a}, b), x)$.

The E-embedding order for terms, denoted as $\trianglelefteq_E$, will be lifted to an E-embedding order for substitutions similar to the encompassment order in [18]. We define $\sigma \trianglelefteq_E \tau$ iff each domain variable x of τ is also a domain variable of σ and $x\tau$ homeomorphically E-embeds $x\sigma$, that is if $\tau = \{x_i \mapsto t_i\}$ and $\sigma = \{x_i \mapsto s_i\}$, $1 \leqq i \leqq n$, then $\sigma \trianglelefteq_E \tau$ iff $s_i \trianglelefteq_E t_i$. To illustrate the effect of this E-embedding order, take $\mathit{\Gamma}_2$ from above as an example, where E is the equational theory A for strings. In this case $aba \trianglelefteq_A ab......ba$ and $b \trianglelefteq_A b.....b$, hence with $\sigma_1 = \{x \mapsto aba, y \mapsto b\}$ we have $\sigma_1 \trianglelefteq_A \sigma_n$ for all $n > 1$. Consequently σ_1 is the only unifier and the set of embedment free unifiers for Γ_2 is $\lambda\mathcal{U}\Sigma_A(\Gamma_2) = \{\sigma_1\}$ and it is finite. In fact it can be shown that the theory is even *unitary* instead of infinitary under the subsumption and encompassment order [8, 9].

But in order to generalize the encompassment order for terms to the embedment order for unification problems, we need a more general notion of embedment. This is achieved by defining, that a term s is *E-instance-embedded* into a term t iff an instance of s, say $s\lambda$, is E-embedded into t, which we call $\lambda-$*embedded modulo E* or λ_E*-embedding*. This is denoted as $s \leqslant_E t$. Furthermore E-unifiers, which have no E-instance embedded unifier are called *embedment free E-unifiers* or *free* λ_E*-unifiers* and the complete set of free λ_E-unifiers is denoted as $\lambda U\Sigma_E(\Gamma)$ for a unification problem Γ.

In the following we introduce the concept of a minimal and complete set of E-unifiers based on λ_E-embedding and propose a definitional framework based on notions and definitions of standard unification theory extended to the (homeomorphic) E-embedding order.

2. Notions and Notation

Notation and basic definitions in unification theory are well known and have found their way into many and diverse academic fields and most monographs and textbooks on automated reasoning have sections on unification. In the full paper to be published in a journal we unify the various presentations of the necessary concepts for unification towards a concise notation which serves our purpose and we show how the additional concepts for ordering E-unifiers based on homeomorphic embedding can be built upon these definitions. These sections as well as several proofs and details are deleted in this workshop paper.

Our main interest is to show that the set of free λ_E-unifiers always exists and the main technique to show this result is based on orderings, in particular on well quasi orderings.

Definition 1. A *quasi order* is a relation that is *reflexive* and *transitive.*

A term t is an *instance* of a term s, denoted $s \leq t$, if $s\sigma = t$ for some substitution σ

$$s \leq t \iff \exists\sigma : s\sigma = t$$

We also say *s (syntactically) subsumes t* and this relation is a quasi order (or preorder as it is sometimes called). We call it the *subsumption order* on terms.
A term t (syntactically) *encompasses* a term s, denoted $s \sqsubseteq t$, if an instance of s is a subterm of t. With **Sub(t)**, the set of all subterms of t, we have:

$$s \sqsubseteq t \iff \exists\sigma : s\sigma \in \mathbf{Sub}(t)$$

Encompassment conveys the notion that s appears in t with a context "above" and a substitution instance "below". We say t *encompasses* s or s is *encompassed* by t and $\sqsubseteq$ is the *encompassment order*. In particular $s \sqsubset t$ is called *strict encompassment*, if $s\sigma$ is a proper subterm of t.

A term s is *homeomorphically embedded* into t iff s can be obtained from t by erasing some parts in t. We abbreviate "homeomorphical embedding" just to "embedding". Embedment conveys the notion that the structure of s and some corresponding symbols appear within t. A term s is *instance-embedded* into t, we also say it is λ*-embedded* into t, iff an instance of s, i.e. $s\lambda$, is embedded into t. This is the main notion of this paper, which we will generalize to embedment of substitutions later on.[1]
Using $s \trianglelefteq t$ to denote that s is a subterm of t, we have the following *orders on terms,* extended to equality modulo E for the congruences induced by the equations in E. :

Definition 2. (ordering terms modulo E)

(1) A term s is an *E-subterm of t, denoted* $s \trianglelefteq_E t$, iff there is an $s' =_E s$ and a term $t' =_E t$ such that $s' \trianglelefteq t'$.
(2) A term s *E-subsumes t,* $s \lessdot_E t$, iff there exists a substitution σ with $s\sigma =_E t$.
(3) A term s is *E-encompassed by t*, $s \sqsubseteq_E t$ iff there is a substitution σ such that $s\sigma \trianglelefteq_E t$.
(4) A term s is *E-embedded* into a term t, denoted $s \unlhd_E t$, if $s =_E t$, or there is a term $s' =_E s$ and a term $t' =_E t$ such that s' is syntactically embedded into t':

$$s \unlhd_E t \iff \begin{cases} s =_E t,\ or \\ t = f(t_1, \dots, t_n)\ and\ \exists s' \in [s]_E\ and\ \exists t'_i \in [t_i]_E : s' \unlhd t'_i,\ or \\ t = f(t_1, \dots t_n), s = f(s_1, \dots s_n), \\ \quad and\ \forall i : s'_i \unlhd t'_i,\ where\ s' \in [s_i]_E,\ t' \in [t_i]_E. \end{cases}$$

(5) A term s is *E-instance-embedded* into t, denoted $s \lessdot_E t$, if an instance of s is E-embedded into t , that is $s\lambda \unlhd_E t$ for a substitution λ. We say s is λ_E-embedded.

Theorem 3. *The E-embedment order,* $\unlhd_E$*, is a quasi order on terms.*

Proof. Let r, s, t be terms.
reflexivity: $r \unlhd_E r$ is obvious, because terms embed themselves.
transitivity: $r \unlhd_E s \unlhd_E t \Longrightarrow r \unlhd_E t$.
By Definition 2.(4) $r \unlhd_E s \Longrightarrow \exists r' \in [r]_E\ and\ \exists s' \in [s]_E : r' \unlhd s'$ and
$s \unlhd_E t \Longrightarrow \exists s'' \in [s]_E$ and $\exists t'' \in [t]_E : s'' \unlhd t''$.
Now $r' \unlhd s' =_E s''\ and\ s'' \unlhd t'' \Longrightarrow r' \unlhd_E t''$.
Hence $r =_E r' \unlhd_E t'' =_E t \Longrightarrow r \unlhd_E t$.

□

Definition 4. (ordering of substitutions modulo E restricted to a set of variables)
Let V be some set of variables.

[1]Signs and notation are still not uniform in all related fields; our notation is used more often in the literature on automated theorem proving and unification theory, whereas term rewriting systems usually prefer notational conventions as proposed in [5] and [6].

(1) A substitution σ is a *sub-substitution modulo E* of τ, denoted as $\sigma \preceq_E^V \tau$, if $\mathbf{Dom}(\sigma) = \mathbf{Dom}(\tau)$ and these are variables in V and for all x in this domain $x\sigma$ is an E-subterm of $x\tau$, i.e. $x\sigma \preceq_E x\tau$.
(2) A substitution σ *E-subsumes* a substitution τ *restricted to* V, denoted as $\sigma \leq_E^V \tau$, if there exists a substitution λ such that $\sigma\lambda =_E^V \tau$. The relation $\leq_E^V$ is called the *E-subsumption order for substitutions* restricted to V.
We denote *E-subsumption equivalence* as $\sigma \sim_E^V \tau$, if $\sigma \leq_E^V \tau$ *and* $\tau \leq_E^V \sigma$.
(3) A substitution σ is *E-encompassed* by τ *restricted to* V, denoted $\sigma \sqsubseteq_E^V \tau$, if there exists λ, such that $(\sigma\lambda)\mid_V$ is a sub-substitution of τ modulo E restricted to V. We denote *E-encompassment equivalence* as $\sigma \approx_E^V \tau$ if $\sigma \sqsubseteq_E^V \tau$ *and* $\tau \sqsubseteq_E^V \sigma$.
(4) A substitution σ is *E-embedded* into a substitution τ, denoted as $\sigma \trianglelefteq_E^V \tau$, iff $\mathbf{Dom}(\sigma) = \mathbf{Dom}(\tau)$ and for all x in this domain we have $x\sigma \trianglelefteq_E^V x\tau$.
(5) A substitution σ is λ_E*-embedded* into a substitution τ, denoted as $\sigma \lessdot_E^V \tau$, iff $\mathbf{Dom}(\sigma) = \mathbf{Dom}(\tau)$ and there is a substitution λ, such that $\forall x \in V : x(\sigma\lambda)\mid_V$ is E-embedded into $x\tau$.

The encompassment and embedment order on terms are well known as quasi orderings, but the *modulo E* extension to substitutions requires verification.

Theorem 5. *The E-encompassment order is a quasi order on substitutions.*

For a proof see [18] and the early proof in [10].

Theorem 6. *The E-embedment order is a quasi order on substitutions.*

Proof. This is shown by lifting Theorem 3 for terms componentwise to substitutions.

□

Theorem 7. *The* λ_E*-embedment order* $\lessdot_E$ *is a quasi order on terms.*

Proof. *Let* r, s, t *be terms:*
reflexivity: is obvious because every term λ-embeds itself.
transitivity: we show $r \lessdot_E s \lessdot_E t$ implies $r \lessdot_E t$.
By Definition 2.(5) we have:
$r \lessdot_E s$ implies $\exists\sigma : r\sigma \trianglelefteq_E s$ and
$s \lessdot_E t$ implies $\exists\tau : s\tau \trianglelefteq_E t$. Furthermore:
$r\sigma \trianglelefteq_E s \Longrightarrow \exists\widetilde{r} \in [r\sigma]_E$ and $\exists s' \in [s]_E : \widetilde{r} \trianglelefteq s'$ and
$s\tau \trianglelefteq_E t \Longrightarrow \exists\widetilde{s} \in [s\tau]_E$ and $\exists t' \in [t]_E : \widetilde{s} \trianglelefteq t'$.
Since it can be shown that $\trianglelefteq$ is substitution-composable from the right we have
$\widetilde{r} \trianglelefteq s' \Longrightarrow \widetilde{r}\tau \trianglelefteq s'\tau$ and $s'\tau =_E \tilde{s} \trianglelefteq t' \Longrightarrow s'\tau \trianglelefteq_E t'$.
Now: $\widetilde{r}\tau \trianglelefteq s'\tau \trianglelefteq_E t'$ and transitivity of $\trianglelefteq_E$ implies $\widetilde{r}\tau \trianglelefteq_E t'$
and $r(\sigma\tau) \trianglelefteq_E s'\tau \trianglelefteq_E t' =_E t$ and transitivity of $\trianglelefteq_E$
implies $r(\sigma\tau) \trianglelefteq_E t$. But this means $r \lessdot_E t$.

□

Theorem 8. *The* λ_E*-embedment order is a quasi order on substitutions.*

Proof. Similar to Theorem 7 by lifting it componentwise to substitutions.

□

Our interest in this paragraph is on quasi orderings and the next definition lists some well known notions, see [12, 16].

Definition 9. Let $\leq$ be a quasi ordering on a set S, then:

(1) An infinite sequence of elements of S, $a_1, a_2, a_3, \ldots$ is called a $\leq$*−chain* if $a_i \leq a_{i+1}$ for all $i \geq 1$. The sequence $a_1, a_2, a_3, \ldots$ is said to *contain a chain* if it has a subsequence that is a chain.
(2) The infinite sequence $a_1, a_2, a_3, \ldots$ is called an *antichain* if neither $a_i \leq a_j$ nor $a_j \leq a_i$, for all $1 \leq i, j$ and i≠j.
(3) The quasi ordering $\leq$ is *well-founded* (wfo) if it contains no infinite strictly descending <-chain; that is, there is no infinite sequence $a_1, a_2, a_3, \ldots$ of elements of S such that $a_i > a_{i+1}$ for every i *in* $\mathbb{N}$.

(4) A *well-quasi-ordering on S* (wqo), $\leq$, is a quasi-ordering which is well-founded and it has no infinite antichains in S with respect to $\leq$.

The following ***Tree Theorem*** due to Kruskal states that the set of finite trees over a well-quasi-ordered set of labels is itself well-quasi-ordered under homeomorphic embedding. He uses a notation where T(Y) denotes the collection of all (structured) trees over an alphabet Y.

Theorem 10. ***The Tree Theorem.***
If Y is well quasi ordered then T(Y) is well quasi ordered too.

Proof. See Joseph B. Kruskal [12] and the more elegant proof by Crispin Nash-William [16]

□

The following theorem is a consequence of the tree theorem for the set of first order terms $T(F,X)$, built over a finite set of function symbols F and a finite set of variable symbols X. Hereby we refer to the work of Jean H. Gallier and M. Leuschel [7, 15]. They discuss the proof that "Given a finite alphabet $\Sigma = F \cup X$ which is well quasi ordered (in our case by equality) then $\trianglelefteq$ is also a well quasi order on $T(F, X)$". The next theorem is a genaralisation to "modulo E".

Theorem 11. *Let E be an equational theory. The E-embedding relation , $\trianglelefteq_E$, is a well quasi order on the set of terms built over a* ***finite*** *signature.*

Proof. (Sketch)
(i) $\trianglelefteq_E$ is *well founded.*
If not, then there exists an infinite strictly descending $\triangleright_E$ –chain: $t_1 \triangleright_E t_2 \triangleright_E t_3 \triangleright_E ....$
which has the more detailed form: $t'_1 \triangleright t'_2 =_E t''_2 \triangleright t'_3 =_E t''_3 \triangleright t'_4 ....$ Now take the following infinite sub sequence of terms t'_1, t'_2, t'_3, ... Because of Theorem 10 there are two indices i, j, i<j such that t_i is embedded into t_j, hence the chain above can not be infinite. Thus $\trianglelefteq_E$ is well founded.
(ii) There are *no antichains* with respect to $\trianglelefteq_E$.
Otherwise there is an $\trianglelefteq_E$-antichain $s_1, s_2, s_3, ...$ with respect to $\trianglelefteq_E$ and it can be shown that in this case there exists a corresponding infinite sequence of terms s'_1, s'_2,s'_3, ... , where $s'_i \in [s_i]_E, i \geq 1$, which are incomparable and this again contradicts Kruskal's theorem.

□

E-unification of first order terms is based on an *infinite* set of variable symbols and it is well known, that the embedding order of terms with an infinite set of variable symbols is not a well quasi order, since we have the antichain $x_1, x_2, x_3, ..$ Of course the same is the case then for embedment modulo E.
But fortunately *well foundedness* of the embedding ordering is still valid, since the number of symbols decreases in a strictly descending $\triangleright$ –chain .

Theorem 12. *Let E be an equational theory. E-embedding, $\trianglelefteq_E$, is a well founded quasi order on the set of terms.*

Proof. The proof is based on the fact that E-equivalent terms do not have new variable symbols.

□

The next Theorem is similar and shows that λ_E-embedding is well foundned too.

Theorem 13. *Let E be an equational theory. λ_E–embedding, $\preccurlyeq_E$, is a well founded quasi order on the set of terms.*

3. Ordering E-unifiers under homeomorphic embedding

We shall now look at unification under λ_E-embedding, which is our main interest in this paper, and we start with a recapitulation of the standard notions of E-unification.

3.1. **E-Unification.** Let E be an equational theory and let Σ be the signature of the term algebra. An *E-unification problem* is a finite set of equations

$$\Gamma = \{s_1 \overset{?}{=}_E t_1, \ldots, s_n \overset{?}{=}_E t_n\}$$

An *E-unifier* for Γ is a substitution σ such that

$$s_1\sigma =_E t_1\sigma, \ldots, s_n\sigma =_E t_n\sigma$$

The set of all E-unifiers of Γ is denoted $\mathcal{U}\Sigma_E(\Gamma)$. A *complete* set of E-unifiers $c\mathcal{U}\Sigma_E(\Gamma)$ for Γ is a set of E-unifiers, such that for every E-unifier τ there exists $\sigma \in c\mathcal{U}\Sigma_E(\Gamma)$ with $\sigma \leq_E \tau$. The set $\mu\mathcal{U}\Sigma_E(\Gamma)$ is called a *minimal complete set* of E-unifiers for Γ, if it is complete and for all distinct elements σ and σ' in $\mu\mathcal{U}\Sigma_E(\Gamma)$ if $\sigma \leq_E \sigma'$ then $\sigma =_E \sigma'$.

3.2. **E-Unifiers ordered by Homeomorphic Embedding.** This paper is based on the observation that certain solutions *embed* the instances of other solutions. This then leads to the notion of *(embedment-) free E-unifiers,* where free E-unifiers are the elements of our new minimal and complete set of E-unifiers, which we denote as $\lambda\mathcal{U}\Sigma_E(\Gamma)$.

Definition 14. Let E be an equational theory, Γ a solvable E-unification problem and let $U\Sigma_E(\Gamma)$ be the set of E-unifiers for Γ. If an E-unifier σ in $U\Sigma_E(\Gamma)$ does *not* have any instance E-embedded unifier (any λ_E-embedded unifier), then σ is called a *free λ_E-unifier*. The minimal and complete set of free λ_E-unifiers will be denoted as $\lambda U\Sigma_E(\Gamma)$.

Theorem 15. *For first order terms built over a **finite** signature Σ and a solvable unification problem Γ and an equational theory E: The set of free λ-unifiers, $\lambda U\Sigma_E(\Gamma)$, exists and it is minimal, complete and finite.*

Proof. The proof is based on Theorem 11

□

It is well known that the set of terms can not be well quasi ordered since we usually have an infinite set of variables and they form an antichain. That is, we can not use Theorem 11. But it may be possible for an automated deduction system to set a limit to the number of variables involved in the search for a proof and the above theorem would be still useful.

Unfortunately we do not know if the relation $\leqslant_E$ is wqo[2]. But we know it is wellfounded and hence we have:

Theorem 16. *For a signature Σ, a solvable unification problem Γ and an equational theory E: The set of free λ_E-unifiers, $\lambda U\Sigma_E(\Gamma)$, exists and is minimal and complete. But it is not necessarily finite .*

Finally there is a standard trick used in logic programming [14] [15] as well as in termination research for term rewriting systems [5], namely to disregard the name of a variable and simply view all variables as the same entity. This observation led to the notion of *pure embedding,* which we abbreviate to *π-embedding* in the following and it will be denoted as $s \trianglelefteq^{\pi} t$. As before we generalize embedding to instance embedding or π-embedding by saying a term s is *π-embedded* into a term t, if it is λ-embedded and in addition the names of the variable symbols are ignored. It is defined as follows:

Definition 17. (*Pure E-embedding*)

(1) A term s is *π_E-embedded* into a term t, denoted $s \trianglelefteq^{\pi}_E t$, if s and t are variables, or $s =_E t$ or there is a term $s'_i =_E s_i$ and a term $t'_i =_E t_i$ such that s'_i is π_E- embedded into t'_i :

$$s \trianglelefteq^{\pi}_E t \iff \begin{cases} s =_E t, \textit{ or s and t are variables or} \\ t = f(t_1, \ldots, t_n) \textit{ and } \exists s' \in [s]_E \textit{ and } \exists t'_i \in [t_i]_E : s' \trianglelefteq^{\pi}_E t'_i \textit{ or} \\ t = f(t_1, \ldots t_n) \textit{ and } s = f(s_1, \ldots s_n) \\ \qquad \textit{and } \forall i : s'_i \trianglelefteq^{\pi}_E t'_i, \textit{ where } s' \in [s_i]_E,\ t' \in [t_i]_E. \end{cases}$$

(2) A term s is instance π_E-embedded into a term t, denoted $s \leqslant^{\pi}_E t$, if an instance of s is π_E-embedded into t , that is $s\lambda \trianglelefteq^{\pi}_E t$ for a substitution λ.

[2]We have not been able to prove it (yet) nor to disprove it.

(3) A substitution σ is *instance* π_E*−embedded* into a substitution τ for a set of variables V, denoted as $\sigma \leqslant_E^{\pi V} \tau$, iff $\mathbf{Dom}(\sigma) = \mathbf{Dom}(\tau)$ and there is a substitution λ, such that $\forall x \in V : x(\sigma\lambda) |_V$ is π-embedded into $x\tau$.

Since instance π-embedding is a special case of λ-embedding we have (using theorem 11) that π-embedding, $s \trianglelefteq^{\pi} t$, is a well quasi order on the set of terms and it is now easy to show that π-embedding is a well quasi order on the set of substitutions as well.

Theorem 18. *π_E-embedding is a well quasi order on the set of substitutions.*

The final step is now to extend π_E-embedding of substitutions to instance π_E-embedding.

Theorem 19. *Instance π_E-embedding is a well quasi order on the set of substitutions.*

E-unifiers which do not contain any instance π_E-embedded unifiers are called *free π_E unifiers* and this set is denoted as $\pi U\Sigma_E(\Gamma)$.

Our main result now follows from these theorems, but note the completeness proof is more complex than ususal, because we need a generator to compute all unifiers from $\pi U\Sigma_E(\Gamma)$.

Theorem 20. *For first order terms built over a signature Σ, a solvable unification problem Γ and an equational theory E: The set of free π_E-unifiers, $\pi U\Sigma_E(\Gamma)$, exists and is minimal, complete and finite.*

4. Conclusion

These results do not imply that we have a general way of **efficiently** generating $\lambda U\Sigma_E(\Gamma)$ nor $\pi U\Sigma_E(\Gamma)$, which is unlikely to be found in general. We need to look for an appropriate algorithm for each specific theory E, just as in standard unification theory and this has not been done yet.

References

[1] M. Alpuente, A. Cuenca-Ortega, S. Escobar, and J. Meseguer. Partial evaluation of order-sorted equational programs modulo axioms. In *Hermenegildo M., Lopez-Garcia P. (eds) Logic-Based Program Synthesis and Transformation. LOPSTR 2016.* Lecture Notes in Computer Science, vol 10184. Springer, 2017.

[2] F. Baader. A note on unification type zero. *Information Processing Letters*, 27:91–93, 1988.

[3] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[4] N. Dershowitz. Termination of rewriting. *Journal of symbolic computation*, 3 (1):69–115, 1987.

[5] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 244–320. Elsevier Science Publishers (North-Holland), 1990.

[6] Nachum Dershowitz and Jean-Pierre Jouannaud. Notations for rewriting. *Bulletin of the EATCS*, 43:162–174, 1991.

[7] J.H. Gallier. What's so special about kruskal's theorem and the ordinal gamma0 ?: A survey of some results in proof theory. *Annals of Pure and Applied Logic*, 53(3):199–260, 1991.

[8] M. Hoche, J. Siekmann, and P. Szabo. String unification is essentially infinitary. In Mircea Marin, editor, *The 22nd International Workshop on Unification (UNIF'08)*, pages 82–102, Hagenberg, Austria, 2008.

[9] M. Hoche, J. Siekmann, and P. Szabo. String unification is essentially infinitary. *IFCoLog Journal of Logics and their Applications*, 2016.

[10] M. Hoche and P. Szabo. Essential unifiers. *Journal of Applied Logic*, 4(1):1–25, 2006.

[11] G. Huet. A complete proof of correctness of the knuth-bendix completion algorithm. *Journal of Computer and System Sciences*, 23 (1):11–21, 1981.

[12] J. B. Kruskal. Well-quasi-ordering, the tree theorem and Vázsonyi's conjecture. *Trans. Amer. Math. Soc.*, 95:210–225, 1960.

[13] J. B. Kruskal. The theory of well-quasi-ordering: A frequently discovered concept. *Journal of Combinatorial Theory*, 13:297–305, 1972.

[14] M. Leuschel. Improving homeomorphic embedding for online termination. In *International Workshop on Logic Programming Synthesis and Transformation.* Springer, Berlin, Heidelberg, 1998.

[15] M. Leuschel. Homeomorphic embedding for online termination of symbolic methods. In *The Essence of Computation*. Lecture Notes in Computer Science, vol 2566. Springer, Berlin, Heidelberg, 2002.

[16] C. St. J. A. Nash-Williams. On well-quasi-ordering finite trees. In B. J. Green, editor, *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 59 of *04*, pages 833–835. Cambridge Philosophical Society, 1963.

[17] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[18] P. Szabo, J. Siekmann, and M. Hoche. What is essential unification? In *Martin Davis on Computability, Computation, and Computational Logic*. Springer's Series "Outstanding Contributions to Logic", 2016.

Formalization of First-Order Syntactic Unification

Kasper Fabæch Brandt   Anders Schlichtkrull   Jørgen Villadsen
DTU Compute, AlgoLoG, Technical University of Denmark, 2800 Kongens Lyngby, Denmark
{andschl,jovi}@dtu.dk

Abstract

We present a new formalization in the Isabelle proof assistant of first-order syntactic unification, including a proof of termination. Our formalization follows, almost down to the letter, the ML-code from Baader and Nipkow's book "Term Rewriting and All That" (1998). Correctness is implied by the formalization's similarity to Baader and Nipkow's ML-code, but we have yet to formalize the correctness of the unification algorithm.

1 Introduction

We present a new and concise formalization in the Isabelle proof assistant [1] of a quite simple first-order syntactic unification algorithm. In particular we provide a formal proof of termination. The Isabelle2017 formalization is available here:

https://github.com/logic-tools/unification

We follow the succinct presentation of the unification algorithm in Baader and Nipkow's book "Term Rewriting and All That" [2], Section 4.7 Unification and term rewriting in ML, with Isabelle/HOL types as follows.

```
type_synonym vname = "string × int"

datatype "term" = V vname | T "string × term list"

type_synonym subst = "(vname × term) list"
```

A variable name consists of a name and an index (as mentioned in the book the index is not used in the current situation but it can simplify renaming). A term is either a variable or a compound term with a list of subterms. A substitution is a list of pairs, each with a variable name and a term, hence a so-called association list.

2 The Unification Algorithm

We first define a few auxiliary functions as in Baader and Nipkow's book [2].

```
definition indom :: "vname ⇒ subst ⇒ bool" where
  "indom x s = list_ex (λ(y, _). x = y) s"

fun app :: "subst ⇒ vname ⇒ term" where
  "app ((y,t)#s) x = (if x = y then t else app s x)"
| "app [] _ = undefined"

fun lift :: "subst ⇒ term ⇒ term" where
  "lift s (V x) = (if indom x s then app s x else V x)"
| "lift s (T(f,ts)) = T(f, map (lift s) ts)"

fun occurs :: "vname ⇒ term ⇒ bool" where
  "occurs x (V y) = (x = y)"
| "occurs x (T(_,ts)) = list_ex (occurs x) ts"
```

We then define the unification algorithm using the option type rather than ML exceptions.

```
function (sequential) solve :: "(term × term) list × subst ⇒ subst option"
  and elim :: "vname × term × (term × term) list × subst ⇒ subst option"
where
  "solve([], s) = Some s"
| "solve((V x, t) # S, s) = (
    if V x = t then solve(S,s) else elim(x,t,S,s))"
| "solve((t, V x) # S, s) = elim(x,t,S,s)"
| "solve((T(f,ts),T(g,us)) # S, s) = (
    if f = g then solve((zip ts us) @ S, s) else None)"

| "elim(x,t,S,s) = (
    if occurs x t then None
    else let xt = lift [(x,t)]
      in solve(map (λ (t1,t2). (xt t1, xt t2)) S,
        (x,t) # (map (λ (y,u). (y, xt u)) s)))"
```

The unification algorithm is called as `solve([(t1,t2)],[])` for terms `t1` and `t2` to unify and has exponential time and space complexity but performs well on practical examples, as pointed out in Section 4.7 of Baader and Nipkow's book [2].

3 The Termination Proof

The termination proof is derived from the one given on the "Unification (computer science)" Wikipedia page [3]. The idea is to define a measure on the arguments of the function that gets smaller and smaller for each recursive call with respect to the lexicographic ordering.

The measure is $<n_{var}, n_{fun}, |S|>$ with

- n_{var}: number of unsolved variables in S,
- n_{fun}: number of constants and functions in S, and
- $|S|$: number of equations in S.

This differs from the one given on the Wikipedia page because that algorithm defines the set of solved equations as the subset of equations that have a variable on their left-hand side while Baader and Nipkow have an explicit set S representing these. For the same reason, Wikipedia's algorithm reorients equations with a lone variable on the right hand side, while Baader and Nipkow remove them from the set of unsolved equations and add them to the set of variable assignments that makes up the return value.

The majority of the proof consists of showing basic theory about the three quantities, more specifically how they are reduced for each of the operations performed.

Let us look at the proof and its formalization in more detail. We want to define the relation on arguments defined by lexicographical ordering on $<n_{var}, n_{fun}, |S|>$. To do this we define functions which given the arguments to the `solve` function calculates n_{var}, n_{fun} and $|S|$. Let us consider n_{var} first. The function is as follows:

```
(λXX. case XX of Inl(l,_) ⇒ n_var l | Inr(x,t,S,_) ⇒ n_var ((V x, t)#S))
```

Here, `XX` represents the tuple consisting of the arguments given to the function call. We split on the pair-case `Inl(l,_)` representing a call to the `solve` function and the quadruple-case `Inr(x,t,S,_)` representing a call to the `elim` function. In both cases we use the function `n_var` to count the number of variables occurring in the set of unsolved equations. In the case for a call to `elim` we consider `x = t` an unsolved equation and therefore call `n_var` on `((V x, t)#S)`. The other components are defined in a similar way. We now compose the three measures to the desired relation. It is done using Isabelle's `<*mlex*>` operator. The operator is defined such that m `<*mlex*>` R takes a measure m and a relation R, and then turns the measure m into a relation and composes it lexicographically with the relation R. The measure is turned into a relation by comparing the size of the measure on given elements and letting this determine which element the relation considers larger.

The relation we get is as follows:

```
(λXX. case XX of Inl(l,_) ⇒ n_var l | Inr(x,t,S,_) ⇒ n_var ((V x, t)#S)) <*mlex*>
(λXX. case XX of Inl(l,_) ⇒ n_fun l | Inr(x,t,S,_) ⇒ n_fun ((V x, t)#S)) <*mlex*>
(λXX. case XX of Inl(l,_) ⇒ size l | Inr(x,t,S,_) ⇒ size ((V x, t)#S)) <*mlex*>
(λXX. case XX of Inl(l,_) ⇒ 1 | Inr(x,t,S,_) ⇒ 0) <*mlex*> {}
```

The last two components are `(λXX. case XX of Inl(l,_) ⇒ 1 | Inr(x,t,S,_) ⇒ 0)` and `{}`. The former is the measure which gives calls to `solve` size `1` and calls to `elim` size `0`, and the latter is the empty relation.

Our formal termination proof does an application of Isabelle's “relation” proof method which leaves us to prove subgoals expressing that the considered relation is wellfounded and that calls are related as desired by the relation.

Let R represent the above relation. For the readers with the courage to read Isabelle syntax we state the 6 subgoals.

1.
```
wf R
```
2.
```
V x = t ⟹ (Inl (S, s), Inl ((V x, t) # S, s)) ∈ R
```
3.
```
V x ≠ t ⟹ (Inr (x, t, S, s), Inl ((V x, t) # S, s)) ∈ R
```
4.
```
(Inr (x, T v, S, s), Inl ((T v, V x) # S, s)) ∈ R
```
5.
```
f = g ⟹
          (Inl (zip ts us @ S, s),
           Inl ((T (f, ts), T (g, us)) # S, s)) ∈ R
```
6.
```
 ¬ occurs x t ⟹ xa = lift [(x, t)] ⟹
          (Inl (map (λ(t1, t2). (xa t1, xa t2)) S, (x, t) #
                map (λ(y, u). (y, xa u)) s),
           Inr (x, t, S, s)) ∈ R
```

Comparing subgoals 2-6 with the definition of `solve` and `elim` it is not hard to see where the goals come from. For instance in subgoal 2 we have that `V x = t` and the two `Inl` expressions indicate that we are looking at calls to `solve`. In one call with arguments `((V x, t) # S, s))` and in the other with `(S, s)`. All this corresponds exactly to what is happening in the if-case of the second defining equation of `solve`.

We use Isabelle's automation to prove subgoal 1 (wellfoundedness) and to simplify subgoals 2-6. We then prove the subgoals using appropriate lemmas that we have also formalized.

4 Code Generation

We illustrate Isabelle's code generation with a few minimal examples. It is possible to generate and load `SML` (Standard ML) code directly in the Isabelle formalization (called code reflection).

```
code_reflect Unification
  datatypes
    "term" = V | T
    and
    char = zero_char_inst.zero_char | Char
  functions
    solve
```

For the mininal examples we only need the `solve` function in addition to the `term` and `char` datatypes.

This gives the following output displayed in Isabelle (`inta` is a generated datatype for integers used instead of the usual `int` datatype in `SML` and `num` is a generated datatype for natural numbers).

```
structure Unification:
  sig
    datatype char = Char of num | Zero_char
    type inta
    type num
    val solve:
       (term * term) list * ((char list * inta) * term) list ->
         ((char list * inta) * term) list option
    datatype term = T of char list * term list | V of char list * inta
  end
```

Alternatively one can export the generated `SML` code and for example save it in a separate file.

```
export_code solve in SML
```

Below we show a fragment of the generated `SML` code (besides `SML` it is possible to export to the functional programming languages `Haskell`, `Ocaml` and `Scala`).

```
fun elim (x, (t, (sa, s))) =
  (if occurs x t then NONE
    else let
           val xt = lift [(x, t)];
         in
           solve (List.map (fn (t1, t2) => (xt t1, xt t2)) sa,
                   (x, t) :: List.map (fn (y, u) => (y, xt u)) s)
         end)
and solve ([], s) = SOME s
  | solve ((V x, t) :: sa, s) =
    (if equal_terma (V x) t then solve (sa, s) else elim (x, (t, (sa, s))))
  | solve ((T v, V x) :: sa, s) = elim (x, (T v, (sa, s)))
  | solve ((T (f, ts), T (g, us)) :: sa, s) =
    (if List.equal_lista String.equal_char f g
      then solve (List.zip ts us @ sa, s) else NONE);
```

Back to the code reflection, we open the `Unification` structure in Isabelle (note the use of so-called cartouches ‹ … › around the `SML` code in the Isabelle formalization).

```
ML ‹ open Unification ›
```

First we consider unification of identical constants. We formally prove the expected result.

```
lemma "solve([(T([],[]),T([],[]))],[]) = Some []"
  by code_simp
```

We consider the same example in Isabelle using the generated code.

```
ML_val ‹ solve([(T([],[]),T([],[]))],[]) ›
```

Output displayed in Isabelle: `val it = SOME []: ((char list * inta) * term) list option`

We then consider unification of different constants. We again formally prove the expected result.

```
lemma "solve([(T([zero_char_inst.zero_char],[]),T([],[]))],[]) = None"
  by code_simp
```

We also consider the same example in Isabelle using the generated code.

```
ML_val ‹ solve([(T([Zero_char],[]),T([],[]))],[]) ›
```

Output displayed in Isabelle: `val it = NONE: ((char list * inta) * term) list option`

For both examples the output is as expected. More advanced examples are of course possible.

5 Related Work

There are several other formalizations of unification algorithms. An early result is Paulson's formalization [4] in LCF of an algorithm by Manna and Waldinger [5]. This formalization was used as the basis of a formalization by Coen, Slind and Krauss [6] of the same in Isabelle. In this formalization they represent terms as binary trees.

Urban, Pitts and Gabbay [7] also formalize unification in Isabelle, McBride [8] formalizes unification in LEGO, Kumar and Norrish [9] formalize unification in HOL4 and recently Avelar, Galdino, de Moura and Ayala-Rincón [10] formalize unification in PVS.

Most similar to our formalization is probably the formalization in Isabelle by Sternagel and Thiemann [11] since they formalize the same algorithm from Baader and Nipkow's book. A difference is that they have merged the functions solve and elim into one and thus our formalization is strictly more faithful to the original code. Another difference is that they do a different termination proof.

The above formalizations are of worst-case exponential running time algorithms, however, Ruiz-Reina, Martín-Mateos, Alonso and Hidalgo [12] formalize in ACL2 a worst-case quadratic running time algorithm which also stems from Baader and Nipkow's book. Brandt [13] has done preliminary work on formalizing the same algorithm in Isabelle's Imperative HOL. Except for this preliminary work all mentioned formalizations prove termination and correctness.

6 Conclusion

We have formalized first-order syntactic unification. In contrast to other formalizations from the literature we follow the ML-code from Baader and Nipkow's book undeviatingly. We formalize a proof of termination and run the unification algorithm on a number of examples using Isabelle's code-generation.

It is our hope that the stand-alone formalization can be useful for teaching unification algorithms. We did not find any mistakes in the presentation of the unification algorithm in Baader and Nipkow's book. The Isabelle proof assistant allows for quite readable formal proofs of the theorems and the 400 lines are checked in a few seconds.

Future work includes proving correctness but our proof of termination for the rather direct formalization of a succinct unification algorithm should be of interest in itself.

References

[1] Tobias Nipkow, Lawrence C. Paulson and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic.* Springer 2002.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press 1998.

[3] Wikipedia page. *Unification (computer science).* Retrieved from Wikipedia on 25 May 2018: `https://en.wikipedia.org/wiki/Unification_(computer_science)`

[4] Lawrence C. Paulson. *Verifying the unification algorithm in LCF.* Science of Computer Programming 5(2) 143–169 1985.

[5] Zohar Manna and Richard Waldinger. *Deductive synthesis of the unification algorithm.* Science of Computer Programming 1(1–2) 5–48 1981.

[6] Martin Coen, Konrad Slind and Alexander Krauss. *Theory Unification.* Isabelle2017. `https://isabelle.in.tum.de/library/HOL/HOL-ex/Unification.html`

[7] Christian Urban, Andrew Pitts and Murdoch Gabbay. *Nominal Unification.* Theoretical Computer Science 323(1–3) 473–497 2004.

[8] Conor McBride. *First-order unification by structural recursion*. Journal of Functional Programming 13(6) 1061–1075 2003.

[9] Ramana Kumar and Michael Norrish. *(Nominal) Unification by Recursive Descent with Triangular Substitutions.* In Matt Kaufmann and Lawrence C. Paulson, editors: Interactive Theorem Proving (ITP 2010), Lecture Notes in Computer Science 6172, Springer 2010.

[10] Andréia Borges Avelar, André Luiz Galdino, Flávio Leonardo Cavalcanti de Moura and Mauricio Ayala-Rincón. *First-order unification in the PVS proof assistant.* Logic Journal of the IGPL, 22(5) 758–789 2014.

[11] Christian Sternagel and René Thiemann. *First-Order Terms.* Archive of Formal Proofs 2018. `https://www.isa-afp.org/entries/First_Order_Terms.html`

[12] José-Luis Ruiz-Reina, Francisco-Jesús Martín-Mateos, José-Antonio Alonso and María-José Hidalgo. *Formal Correctness of a Quadratic Unification Algorithm*. Journal of Automated Reasoning 37(1–2) 67–92 2006.

[13] Kasper Fabæch Brandt. *Formalization of a near-linear time algorithm for solving the unification problem.* Master's thesis, DTU Compute, AlgoLoG, Technical University of Denmark, 2018. `http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=7091`

Efficiency of a good but not linear nominal unification algorithm

Weixi Ma[1], Jeremy G. Siek[2], David Thrane Christiansen[3], and Daniel P. Friedman[4]

[1] Indiana University, Bloomington, Indiana, U.S.A.
`mvc@iu.edu`
[2] Indiana University, Bloomington, Indiana, U.S.A.
`jsiek@indiana.edu`
[3] Galois, Inc., Portland, Oregon, U.S.A.
`dtc@galois.com`
[4] Indiana University, Bloomington, Indiana, U.S.A.
`dfried@indiana.edu`

Abstract

We present a nominal unification algorithm that runs in $O(n \times log(n) \times G(n))$ time, where G is the functional inverse of Ackermann's function. Nominal unification generates a set of variable assignments if there exists one, that makes terms involving binding operations α-equivalent. We preserve names while using special representations of de Bruijn numbers to enable efficient name management. We use Martelli-Montanari style multi-equation reduction to generate these name management problems from arbitrary unification terms.

1 Introduction and background

Equational theories over terms, such as the α, β, and η in the λ-calculus [Church, 1941], are a critical component of programming languages and formal systems. As users of logic programming languages and theorem provers, we desire such rules to be available out of the box. Two theories provide this convenience: Miller's higher-order pattern unification [Miller, 1989] and Urban et al.'s nominal unification [Urban et al., 2004]. Higher-order pattern unification, the foundation of Isabelle [Paulson, 1986], λProlog [Nadathur et al., 1988], and Twelf [Pfenning and Schürmann, 1999], handles a fragment of the $\beta\eta$-rules. Nominal unification, the unification modulo the α-rule, has inspired extensions of logic programming languages such as αProlog [Cheney and Urban, 2004] and αKanren [Byrd and Friedman, 2007], as well as theorem provers such as Nominal Isabelle [Urban and Tasson, 2005] and αLeanTAP [Near et al., 2008]. Although these two theories can be reduced to one another [Cheney, 2005, Levy and Villaret, 2012], implementing higher-order pattern unification is more complicated because it has to deal with β-reduction and capture-avoiding substitution. An implementation of nominal unification, in which unification does not involve explicit β-reduction, is more straightforward and easier to formalize.

Concerning time complexity, Qian [1996] has proven that higher-order pattern unification is decidable in linear time. Still, it has been an open problem whether there exists a nominal unification algorithm that can do better than $O(n^2)$. Levy and Villaret [2012] give a quadratic-time reduction from nominal unification to higher-order pattern unification. Meanwhile, algorithmic advances by Paterson and Wegman [1978] and Martelli and Montanari [1982] for unification have inspired many improvements to the efficiency of nominal unification. Ideas like applying swappings lazily and composing swappings eagerly and sharing subterms have also been explored. Calvès [2010] describes quadratic algorithms that extend the Paterson-Wegman and Martelli-Montanari's algorithms with name (atom) handling; Levy and Villaret [2010] describe a quadratic algorithm that reduces unification problems to a sequence of freshness and equality constraints and then solves the constraints.

The inefficiency of these nominal unification algorithms comes from the swapping actions. To decide the α-equivalence of two names, we need to linearly traverse a list whose length is the number of binders. One might try to replace these lists with some structures of better lookup efficiency, such as hashtables, but then composing two swappings would take linear time, and that operation is also rather frequent. Here, we present an algorithm that does not use swappings but instead represents names with de Bruijn numbers. De Bruijn numbers enable the use of persistent hashtables, in particular, Bagwell's Hash Array Mapped Trie (HAMT). HAMTs provide efficient lookup and they use sharing to avoid the linear-time costs that would normally be associated with duplicating a hashtable [Bagwell, 2001].

We organize this paper as follows. In section 2, we provide an alternative representation of de Bruijn numbers that is suitable for unification. In section 3, we describe the abstract machines for name management and unification. In section 4, we discuss the time complexity of this algorithm. The proofs of our claims are in progress and are available at the authors' Github,[1] formalized in Agda.

2 De Bruijn numbers should coexist with names

De Bruijn numbers are a technique for representing syntax with binding structure [de Bruijn, 1972]. A *de Bruijn number* is a natural number that indicates the distance from a name's occurrence to its corresponding binder. When all names in an expression are replaced with their corresponding de Bruijn numbers, a direct structural equality check is sufficient to decide α-equivalence. Many programming languages use de Bruijn numbers in their internal representations for machine manipulation during operations such as type checking. The idea of using names for free variables and numbers for bound variables, known as the locally nameless approach [Charguéraud, 2012], is employed for formalizing programming language metatheory [Aydemir et al., 2006, 2008]. Also, de Bruijn numbers, combined with explicit substitution, have been introduced in higher-order unification [Dowek et al., 2000] to improve the efficiency of unification.

Figure 1: Terms

$$\begin{array}{llll} t, l, r & ::= & a & \text{name} \\ & \mid & \lambda a.t & \text{abstraction} \\ & \mid & (l\, r) & \text{combination} \end{array}$$

Figure 2: Free and bound

$$\frac{a \notin \Phi}{\Phi \vdash \texttt{Fr}\ a}\ [\textsc{Free}]$$

$$\frac{\begin{array}{l}(\texttt{name}{\to}\texttt{idx}\ \Phi\ a) = i \\ (\texttt{idx}{\to}\texttt{name}\ \Phi\ i) = a\end{array}}{\Phi \vdash \texttt{Bd}\ a\ i}\ [\textsc{Bound}]$$

Figure 3: $\approx$-rules

$$\frac{\begin{array}{l}a_1 = a_2 \\ \Phi_1 \vdash \texttt{Fr}\ a_1 \quad \Phi_2 \vdash \texttt{Fr}\ a_2\end{array}}{\langle a_1; \Phi_1\rangle \approx \langle a_2; \Phi_2\rangle}\ [\textsc{Same-Free}]$$

$$\frac{\begin{array}{l}i_1 = i_2 \\ \Phi_1 \vdash \texttt{Bd}\ a_1\ i_1 \quad \Phi_2 \vdash \texttt{Bd}\ a_2\ i_2\end{array}}{\langle a_1; \Phi_1\rangle \approx \langle a_2; \Phi_2\rangle}\ [\textsc{Same-Bound}]$$

Despite the convenience when implementing α-equivalence, programs written with de Bruijn numbers are notoriously difficult for humans

[1] https://github.com/mvcccccc/UNIF2018

to read and understand. What's worse, as pointed out by Berghofer and Urban [2007], translating pencil-and-paper style proofs to versions using de Bruijn numbers is surprisingly involved: such a translation may alter the structure of proofs. Thus, recovering proofs with explicit names from proofs that use de Bruijn numbers is difficult or even impossible. Thus, for the sake of both readers and writers of proofs, it is worth providing an interface with names.

If our concern is simply deciding α-equivalence between expressions, an easy way to use de Bruijn numbers while preserving names is to traverse the expressions, annotate each name with its de Bruijn number, then read back the expressions without numbers. This approach, however, does not work for unification, because it only contains the mapping *from names to numbers.* In unification modulo α-equivalence, one frequently needs the mapping *from numbers to names* to decide what name to assign to a unification variable.

We represent de Bruijn numbers by *static closures.* Such closures preserve the mappings in both directions: names to numbers and numbers to names.

Definition 2.1. *A* closure *is an ordered pair $\langle t; \Phi\rangle$ of a term t, defined in Figure 1, and a scope Φ, where the scope is an ordered list of names for the binders in the enclosing context. The name of the innermost binder is written first in Φ.*

When the term of a closure is a name, the closure itself represents a de Bruijn number. Consider the term $\lambda\, a.\lambda\, b.a$. The de Bruijn number of the name a is 1 and the closure-representation of this number is $\langle a; (b\, a)\rangle$. We can retrieve the number-representation by finding the position of the first appearance of the name in the scope. In this case, the position of a in the scope $(b\, a)$ is 1, its de Bruijn number. The de Bruijn number of b would have been 0 if the closure had been $\langle b; (b\, a)\rangle$

We define three operations on scopes: `ext` extends the scope by `consing` a name to the front of the scope; `idx`→`name` yields the name of a given index counting from the leftmost in the scope; and `name`→`idx` yields the location of the first appearance of a given name counting from the front. When repeated names appear, the first appearance in a scope shadows the others.

Figure 4: Unification terms and problems

X			vars
a, b			names
xs	$::=$	ϵ	list of vars
	$\mid$	X, xs	
t, l, r	$::=$	a	
	$\mid$	$\lambda a.t$	abstractions
	$\mid$	$(l\, r)$	combinations
	$\mid$	X	
e_ν	$::=$	$\langle a; \Phi\rangle = \langle a; \Phi\rangle$	ν-equation
	$\mid$	$\langle a; \Phi\rangle = \langle X; \Phi\rangle$	
p_ν	$::=$	ϵ	ν-problems
	$\mid$	e_ν, p_ν	
e_δ	$::=$	$\langle X; \Phi\rangle = \langle X; \Phi\rangle$	δ-equation
p_δ	$::=$	ϵ	δ-problems
	$\mid$	e_δ, p_δ	

Figure 2 defines the free and bound relations "constructively," with de Bruijn numbers serving as evidence that variables are well-scoped. When a name, a, does not appear in the scope, Φ, we say, "a is free with respect to Φ," written as $\Phi \vdash \texttt{Fr}\, a$; when a's first appearance in Φ is the position i, we say, "a is bound at i with respect to Φ," written as $\Phi \vdash \texttt{Bd}\, a\, i$. The BOUND rule needs two premises to be algorithmic for either a name or index input, and prevent incorrect results caused by shadowing. For example, given the index 1 and the scope $(a\, a)$, the relation, $(a\, a) \vdash \texttt{Bd}\, a\, 1$, does **not** hold. Figure 3 defines the rules that decide α-equivalent of two names w.r.t. their scopes, written as $\langle a; \Phi\rangle \approx \langle a; \Phi\rangle$.

3 Unification

In Figure 4, we introduce unification variables, abbreviated as *var*s. Now, let's consider a simplified unification problem: a variable can only be instantiated by a name, that is, finding the unifier of two terms that share the same structure but differ in names and variables. A unifier consists of two parts: σ and δ.

A *substitution*, σ, is a partial finite function from unification variables, X_i, to terms, t_i. For readability, we write σ as a set, $\{X_1/t_1, ..., X_j/t_j\}$ and we write $\{X/t\} \cup \sigma$ for extending σ with X/t. For the simplified problems, we restrict t to a name.

A *closure equation* is a pair of two closures that are α-equivalent. Δ stands for a set of closure equations. We write Δ as $\{(\langle t_1; \Phi_1\rangle\ \langle t'_1; \Phi'_1\rangle), ..., (\langle t_i; \Phi_1\rangle\ \langle t'_i; \Phi'_1\rangle)\}$ and we write $\{(\langle t; \Phi\rangle\ \langle t'; \Phi'\rangle)\} \cup \Delta$ for extending Δ with $(\langle t; \Phi\rangle\ \langle t'; \Phi'\rangle)$. δ is a special form of Δ: for each equation in δ, the terms on both sides are variables. Given a variable X, $\delta(X)$ yields the list of closure equations where X appears at least once.

The simplified problem is about solving three kinds of problems: unifying a closure equation that has a name term on one side and a var term on the other side, abbreviated to N-V, and similarly N-N and V-V. We refer to an N-N or N-V equation as an e_ν and refer to a V-V equation as an e_δ. Given two lists of these closure equations, p_ν and p_δ, we first run the ν-machine, defined in Figure 5, on p_ν to generate a substitution. The δ-machine, defined in Figure 6, then computes the final unifier on three inputs: the substitution resulting from the ν-machine, δ, and a list of known variables, initialized by the domain of the substitution. If no transitions apply, the machine fails and the unification problem has no unifier.

Lemma 3.1. *For all finite inputs, the ν-machine and the δ-machine terminate; for all finite inputs, the ν-machine and the δ-machine succeed with the most general unifier if and only if one exists.*

Proof. By structural induction on the transitions of the machines. $\square$

Figure 5: ν-machine

$$\boxed{\sigma \vdash p_\nu \Rightarrow_\nu \sigma}$$

$$\frac{}{\sigma_0 \vdash \epsilon \Rightarrow_\nu \sigma_0}\ [\textsc{Empty}]$$

$$\frac{\begin{array}{c}\langle a_1; \Phi_1\rangle \approx \langle a_2; \Phi_2\rangle \\ \sigma_0 \vdash p \Rightarrow_\nu \sigma_1\end{array}}{\sigma_0 \vdash \langle a_1; \Phi_1\rangle = \langle a_2; \Phi_2\rangle,\ p \Rightarrow_\nu \sigma_1}\ [\text{N-N}]$$

$$\frac{\begin{array}{c}\langle a_1; \Phi_1\rangle \approx \langle a_2; \Phi_2\rangle \\ \{X_2/a_2\} \cup \sigma_0 \vdash p \Rightarrow_\nu \sigma_1\end{array}}{\sigma_0 \vdash \langle a_1; \Phi_1\rangle = \langle X_2; \Phi_2\rangle,\ p \Rightarrow_\nu \sigma_1}\ [\text{N-V}]$$

Figure 6: δ-machine and the pull operation

$$\boxed{\begin{array}{l}\sigma; p_\delta \vdash xs \Rightarrow_\delta \sigma; p_\delta \\ \sigma; xs \vdash p_\delta \Rightarrow_{\text{pull}} \sigma; xs\end{array}}$$

$$\frac{}{\sigma; \delta \vdash \epsilon \Rightarrow_\delta \sigma; \delta}\ [\textsc{Empty-Xs}]$$

$$\frac{}{\sigma; \epsilon \vdash xs \Rightarrow_\delta \sigma; \epsilon}\ [\textsc{Empty-D}]$$

$$\frac{\begin{array}{c}\sigma_0; xs_0 \vdash \delta_0(X) \Rightarrow_{\text{pull}} \sigma'_0; xs_1 \\ \sigma'_0; \delta_0 \setminus \delta_0(X) \vdash xs_1 \Rightarrow_\delta \sigma_1; \delta_1\end{array}}{\sigma_0; \delta_0 \vdash X, xs_0 \Rightarrow_\delta \sigma_1; \delta_1}\ [\textsc{Pull}]$$

$$\frac{}{\sigma; xs \vdash \epsilon \Rightarrow_{\text{pull}} \sigma; xs}\ [\textsc{Empty}]$$

$$\frac{\begin{array}{l}\langle a_1; \Phi_1\rangle \approx \langle a_2; \Phi_2\rangle \qquad [\text{N-N}] \\ \sigma_0(X_1) = a_1 \quad \sigma_0(X_2) = a_2 \\ \sigma_0; xs_0 \vdash p \Rightarrow_{\text{pull}} \sigma_1; xs_1\end{array}}{\sigma_0; xs_0 \vdash \langle X_1; \Phi_1\rangle = \langle X_2; \Phi_2\rangle, p \Rightarrow_{\text{pull}} \sigma_1; xs_1}$$

$$\frac{\begin{array}{l}\langle a_1; \Phi_1\rangle \approx \langle a_2; \Phi_2\rangle \qquad [\text{N-V}] \\ \sigma_0(X_1) = a_1 \quad X_2 \notin dom(\sigma_0) \\ \{X_2/a_2\} \cup \sigma_0; (X_2, xs_0) \vdash p \Rightarrow_{\text{pull}} \sigma_1; xs_1\end{array}}{\sigma_0; xs_0 \vdash \langle X_1; \Phi_1\rangle = \langle X_2; \Phi_2\rangle, p \Rightarrow_{\text{pull}} \sigma_1; xs_1}$$

Now the question is how to generalize the previous algorithm, that is, given two arbitrary terms, where a variable may be instantiated by any term besides names, can we re-shape the two terms to create a proper input to the two machines?

Here we use the idea of Martelli and Montanari [1982]: finding the shared shape of two terms by computing the common parts and frontiers over a multi-equation. They define the *common part* of two terms to be a term obtained by superimposing, and the *frontier* to be the substitution that captures the differences between each term and the common part. For example, given distinct names a, b, and c, distinct vars X and Y, and two terms $(a\,X)$ and $(Y\,(b\,c))$, the common part is the term $(Y\,X)$, and the frontier is the substitution $\{Y/a,\ X/(b\,c)\}$. A multi-equation, defined in Figure 7, groups together many closures to be unified, where the variable closures are on the left-hand side, and the non-variable closures are on the right-hand side.

The ρ-machine, defined in Figure 8, reduces an arbitrary nominal unification problem to p_ν, p_δ, and a substitution where the codomain is unrestricted. Each $\Rightarrow_s$ transition computes the common part and the frontier of a multi-equation. For readability, the sketch only shows the rules for multi-equations with two closures. A multi-equation with more than two closures is handled by simultaneously applying the rule to all closures. Unlike the Martelli-Montanari algorithm, the ρ-machine finds the maximum common part instead of the minimum. Thus, in the V-C, V-A, and V-A$'$ rules, we need two operators, `new-name` and `new-var`, to create new names and new variables for the shapes that fit with combinations and abstractions. The ordering of multi-equation is the same with Martelli-Montanari: for each multi-equation, we count the appearances of its left-hand side variables in other multi-equations of U and select the multi-equation associated with the smallest counter each time.

Conjecture 3.1. *Given a unification problem, we run the ρ-machine, the ν-machine, and the δ-machine in sequence. The algorithm terminates; if the algorithm fails, i.e. no transitions apply, the problem has no solution; if the algorithm terminates, then the result of the δ-machine is the mgu.*

Figure 7: Multi-equations

$$
\begin{array}{llll}
e & ::= & (\langle t;\Phi\rangle\;\langle t;\Phi\rangle) & \text{multi-equation}\\
 & | & \langle t;\Phi\rangle, e & \\
 & & & \\
U & ::= & \epsilon & \text{list of multi-equations}\\
 & | & e, U &
\end{array}
$$

Figure 8: ρ-machine

$$
\boxed{\begin{array}{l}
p_\nu; p_\delta; \sigma \vdash U \Rightarrow_\rho p_\nu; p_\delta; \sigma\\
p_\nu; p_\delta; \sigma \vdash e \Rightarrow_s p_\nu; p_\delta; \sigma
\end{array}}
$$

$$
\frac{}{p_0;\delta_0;\sigma_0 \vdash \epsilon \Rightarrow_\rho p_0;\delta_0;\sigma_0}\ [\textsc{Empty}]
$$

$$
\frac{\begin{array}{c}
p_0;\delta_0;\sigma_0 \vdash e \Rightarrow_s p_0';\delta_0';\sigma_0'\\
p_0';\delta_0';\sigma_0' \vdash U \Rightarrow_\rho p_1;\delta_1;\sigma_1
\end{array}}{p_0;\delta_0;\sigma_0 \vdash (e,U) \Rightarrow_\rho p_1;\delta_1;\sigma_1}\ [\textsc{Step}]
$$

$$
\frac{p_1 = (\langle a_1;\Phi_1\rangle\;\langle a_2;\Phi_2\rangle) \cup p_0}{p_0;\delta_0;\sigma_0 \vdash (\langle a_1;\Phi_1\rangle\;\langle a_2;\Phi_2\rangle) \Rightarrow_s p_1;\delta_0;\sigma_0}\ [\text{N-N}]
$$

$$
\frac{p_1 = (\langle a_1;\Phi_1\rangle\;\langle X_2;\Phi_2\rangle) \cup p_0}{p_0;\delta_0;\sigma_0 \vdash (\langle a_1;\Phi_1\rangle\;\langle X_2;\Phi_2\rangle) \Rightarrow_s p_1;\delta_0;\sigma_0}\ [\text{N-V}]
$$

$$
\frac{\delta_1 = (\langle X_1;\Phi_1\rangle\;\langle X_2;\Phi_2\rangle) \cup \delta_0}{p_0;\delta_0;\sigma_0 \vdash (\langle X_1;\Phi_1\rangle\;\langle X_2;\Phi_2\rangle) \Rightarrow_s p_0;\delta_1;\sigma_0}\ [\text{V-V}]
$$

$$
\frac{\begin{array}{c}
p_0;\delta_0;\sigma_0 \vdash (\langle l_1;\Phi_1\rangle\;\langle l_2;\Phi_2\rangle) \Rightarrow_s p_0';\delta_0';\sigma_0'\\
p_0';\delta_0';\sigma_0' \vdash (\langle r_1;\Phi_1\rangle\;\langle r_2;\Phi_2\rangle) \Rightarrow_s p_1;\delta_1;\sigma_1
\end{array}}{p_0;\delta_0;\sigma_0 \vdash (\langle (l_1\,r_1);\Phi_1\rangle\;\langle (l_2\,r_2);\Phi_2\rangle) \Rightarrow_s p_1;\delta_1;\sigma_1}\ [\text{C-C}]
$$

$$
\frac{\begin{array}{c}
\Phi_1' = (\texttt{ext}\;\Phi_1\;a_1) \quad \Phi_2' = (\texttt{ext}\;\Phi_2\;a_2)\\
p_0;\delta_0;\sigma_0 \vdash (\langle t_1;\Phi_1'\rangle\;\langle t_2;\Phi_2'\rangle) \Rightarrow_s p_1;\delta_1;\sigma_1
\end{array}}{p_0;\delta_0;\sigma_0 \vdash (\langle \lambda\,a_1.t_1;\Phi_1\rangle\;\langle \lambda\,a_2.t_2;\Phi_2\rangle) \Rightarrow_s p_1;\delta_1;\sigma_1}\ [\text{A-A}]
$$

$$
\frac{\begin{array}{l}
X_l = (\texttt{new-var}) \quad X_r = (\texttt{new-var})\\
p_0;\delta_0;\{X_1/(X_l,X_r)\} \cup \sigma_0 \vdash (\langle X_l;\Phi_1\rangle\;\langle l_2;\Phi_2\rangle) \Rightarrow_s p_0';\delta_0';\sigma_0'\\
p_0';\delta_0';\sigma_0' \vdash (\langle X_r;\Phi_1\rangle\;\langle r_2;\Phi_2\rangle) \Rightarrow_s p_1;\delta_1;\sigma_1
\end{array}}{p_0;\delta_0;\sigma_0 \vdash (\langle X_1;\Phi_1\rangle\;\langle (l_2\,r_2);\Phi_2\rangle) \Rightarrow_s p_1;\delta_1;\sigma_1}\ [\text{V-C}]
$$

$$
\frac{\begin{array}{l}
\Phi_1' = (\texttt{ext}\;\Phi_1\;a_1) \quad \Phi_2' = (\texttt{ext}\;\Phi_2\;a_2) \quad \Phi_1 \vdash \texttt{Fr}\;a_1 \quad a_2 = (\texttt{new-name}) \quad X_t = (\texttt{new-var})\\
p_0;\delta_0;\{X_1/\lambda\,a_1.X_t\} \cup \sigma_0' \vdash (\langle X_t;\Phi_1'\rangle\;\langle t_2;\Phi_2'\rangle) \Rightarrow_s p_1;\delta_1;\sigma_1
\end{array}}{p_0;\delta_0;\sigma_0 \vdash (\langle X_1;\Phi_1\rangle\;\langle \lambda\,a_2.t_2;\Phi_2\rangle) \Rightarrow_s p_1;\delta_1;\sigma_1}\ [\text{V-A}]
$$

$$
\frac{\begin{array}{l}
\Phi_1' = (\texttt{ext}\;\Phi_1\;a_1) \quad \Phi_2' = (\texttt{ext}\;\Phi_2\;a_2) \quad \Phi_1 \vdash \texttt{Bd}\;a_1\;i \quad \Phi_2 \vdash \texttt{Bd}\;a_2\;i \quad X_t = (\texttt{new-var})\\
p_0;\delta_0;\{X_1/\lambda\,a_1.X_t\} \cup \sigma_0' \vdash (\langle X_t;\Phi_1'\rangle\;\langle t_2;\Phi_2'\rangle) \Rightarrow_s p_1;\delta_1;\sigma_1
\end{array}}{p_0;\delta_0;\sigma_0 \vdash (\langle X_1;\Phi_1\rangle\;\langle \lambda\,a_2.t_2;\Phi_2\rangle) \Rightarrow_s p_1;\delta_1;\sigma_1}\ [\text{V-A}']
$$

4 A note on time complexity

In the previous sections, we represent scopes by lists for simplicity, but lists are inefficient for variable lookup. To have better time complexity, we represent a scope with a counter and two persistent hashtables. One hashtable maps from names to numbers, the other maps from numbers to names, and the counter is used to track the de Bruijn number. When we extend a scope with a name, we extend the two hashtables with the corresponding maps and add one to the counter. A persistent hashtable, in practice, has constant time for update and lookup, although the worst case scenario could be $O(log(n))$. Thus, `ext`, `idx`$\rightarrow$`name`, and `name`$\rightarrow$`idx` are all logarithmic time. In addition, using persistent structures avoids copying the entire data-structure when branching, in particular, during the C-C rule of the ρ-machine. Also, we implement δ with a hashtable that maps from a variable to the list that contains its closure equations, i.e., the equation $\langle X_1; \Phi_1\rangle \approx \langle X_2; \Phi_2\rangle$ exists in both X_1's entry and Y_2's entry in the hashtable.

Now the ν-machine and the δ-machine are both worst case $O(n \times log(n))$, where n is the sum of name and variable occurrences. The algorithm of Martelli-Montanari is $O(n \times G(n))$, when representing sets with UNION-FIND [Tarjan, 1975], where n is the number of variable occurrences in the original terms. The ρ-machine is similar except that two new factors are involved: the update operation of HAMT and the generation of names and variables. We consider the former one to have $O(log(n))$ complexity, and we implement name and variable creation with state monads [Moggi, 1991] to have constant time. Thus reducing an arbitrary unification problem to the input of the ν and δ machines becomes $O(n \times log(n) \times G(n))$.

Acknowledgment

We thank Christian Urban for the discussion and inspiration and the anonymous reviewers for their comments and suggestions.

References

Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal Reasoning Techniques in Coq. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '06, Seattle, WA, USA, August 2006.

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 3–15, New York, NY, USA, 2008.

Phil Bagwell. Ideal Hash Trees. *Technical Report EPFL-REPORT-169879, Ecole polytechnique fédérale de Lausanne*, November 2001.

Stefan Berghofer and Christian Urban. A Head-to-Head Comparison of De Bruijn Indices and Names. *ENTCS 174*, (5):53–67, June 2007.

William E. Byrd and Daniel P. Friedman. αKanren A Fresh Name in Nominal Logic Programming. *Université Laval Technical Report*, (DIUL-RT-0701, Scheme Workshop '07, editor Danny Dubé):79 – 90, 2007.

Christophe Calvès. *Complexity and Implementation of Nominal Algorithms.* PhD thesis. King's College of London, 2010.

Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, October 2012.

James Cheney. Relating nominal and higher-order pattern unification. In *Proceedings of UNIF 2005*, pages 104–119, 2005.

James Cheney and Christian Urban. αProlog: A logic programming language with names, binding and α-equivalence. In *Logic Programming*, LNCS 3132, pages 269–283. Springer, Berlin, Heidelberg, September 2004.

Alonzo Church. *The Calculi of Lambda-conversion.* Princeton University Press, Humphrey Milford Oxford University Press, 1941.

Nicolaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, January 1972.

Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher Order Unification via Explicit Substitutions. *Information and Computation*, 157(1):183–235, February 2000.

Jordi Levy and Mateu Villaret. An Efficient Nominal Unification Algorithm. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, RTA '10, pages 209–226, Edinburgh, Scottland, UK, 2010.

Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. *ACM Transactions on Computational Logic*, 13(2):1–31, April 2012.

Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, April 1982.

Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Extensions of Logic Programming*, LNCS 475, pages 253–281. Springer, Berlin, Heidelberg, December 1989.

Eugenio Moggi. Notions of Computation and Monads. *Inf. Comput. 93*, 93(1):55–92, July 1991.

G. Nadathur, D. Miller, University of Pennsylvania Department of Computer, and Information Science. *An Overview of Lambda Prolog*, volume 116 of *LINC LAB*. University of Pennsylvania, Department of Computer and Information Science, 1988.

Joseph P. Near, William E. Byrd, and Daniel P. Friedman. αleanTAP: A declarative theorem prover for first-order classical logic. In *Logic Programming*, LNCS 5366, pages 238–252. Springer, Berlin, Heidelberg, December 2008.

M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978.

Lawrence C. Paulson. Natural deduction as higher-order resolution. *The Journal of Logic Programming*, 3(3):237–258, October 1986.

Frank Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *CADE-16*, LNCS 1632, pages 202–206. Springer, Berlin, Heidelberg, July 1999.

Z. Qian. Unification of Higher-order Patterns in Linear Time and Space. *Journal of Logic and Computation*, 6(3):315–341, June 1996.

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2): 215–225, April 1975.

Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. In *CADE-20*, LNCS 3632, pages 38–53. Springer, Berlin, Heidelberg, July 2005.

Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497, September 2004.