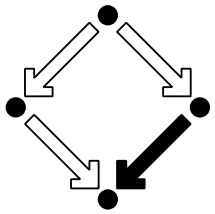
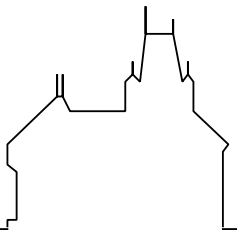


RISC-Linz
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe



The 28th International Workshop on

Unification

UNIF 2014

A FLoC workshop at Vienna Summer of Logic 2014

Workshop Proceedings

July 13, 2014
Vienna, Austria

Temur Kutsia and Christophe Ringeissen
(Editors)

RISC-Linz Report Series No. 14-06

Editors: RISC-Linz Faculty
B. Buchberger, R. Hemmecke, T. Jebelean, M. Kauers, T. Kutsia, G. Landsmann, F. Lichtenberger, P. Paule, V. Pillwein, N. Popov, H. Rolletschek, J. Schicho, C. Schneider, W. Schreiner, W. Windsteiger, F. Winkler.

Foreword

This volume collects contributions presented at the 28th International Workshop on Unification (UNIF 2014), held July 13, 2014, in Vienna, Austria. UNIF 2014 was organized as a FLoC workshop hosted by RTA-TLCA and IJCAR, in the context of the Vienna Summer of Logic.

UNIF is a well-established event with almost three decades of history. UNIF 2014 is the 28th event in a series of international meetings devoted to unification theory and its applications. Previous editions were held at:

Val D'Ajol, France, 1987
Val D'Ajol, France, 1988
Lambrecht, Germany, 1989
Leeds, UK, 1990
Barbizon, France, 1991
Schloß Dagstuhl, Germany, 1992
Boston, USA, 1993
Val D'Ajol, France, 1994
Sitges, Spain, 1995
Herrsching, Germany, 1996
Orléans, France, 1997
Rome, Italy, 1998
Frankfurt, Germany, 1999
Pittsburgh, USA, 2000
Siena, Italy, 2001
Copenhagen, Denmark, 2002
Valencia, Spain, 2003
Cork, Ireland, 2004
Nara, Japan, 2005
Seattle, USA, 2006
Paris, France, 2007
Hagenberg, Austria, 2008
Montreal, Canada, 2009
Edinburgh, UK, 2010
Wroclaw, Poland, 2011
Manchester, UK, 2012
Eindhoven, The Netherlands, 2013

Unification is concerned with the problem of identifying terms, finding solutions for equations, or making formulas equivalent. It is a fundamental process used in a number of fields of computer science, including automated reasoning, term rewriting, logic programming, natural language processing, program analysis, types, etc.

The International Workshop on Unification (UNIF) is a yearly forum for researchers in unification theory and related fields to meet old and new colleagues, to present recent (even unfinished) work, and to discuss new ideas and trends. It is also a good opportunity for young researchers and scientists working in related areas to get an overview of the current state of the art in unification theory.

The Program Committee selected 11 contributions. In addition, the program included two invited talks given by Jordi Levy, *On the Limits of Second-Order Unification*, and by Jose Meseguer on *Extensible Symbolic System Analysis*.

We would like to thank the authors for their contributions and presentations. We are grateful to the invited speakers for their talks and their contributions to the proceedings. We thank the members of the Program Committee and all the referees for their care and time spent in reviewing the submissions. We thank the members of the UNIF Steering Committee for their advice and support, and Andrei Voronkov for his EasyChair conference management system.

Workshop Chairs

Temur Kutsia — RISC, Johannes Kepler University Linz
Christophe Ringeissen — LORIA, INRIA Nancy Grand Est

Program Committee

Franz Baader	TU Dresden
Mnacho Echenim	University of Grenoble
Santiago Escobar	Universitat Politecnica de Valencia
Maribel Fernandez	King's College London
Temur Kutsia	RISC, Johannes Kepler University Linz
Christopher Lynch	Clarkson University
Mircea Marin	West University of Timisoara
Barbara Morawska	TU Dresden
Paliath Narendran	University at Albany–SUNY
Jan Otop	IST Austria
Christophe Ringeissen	LORIA INRIA Lorraine, Villers-les-Nancy
Manfred Schmidt-Schauss	Inst. for Informatik, Goethe-University Frankfurt
Ralf Treinen	PPS, Université Paris Diderot
Mateu Villaret	Departament d'Informàtica i Matemàtica Aplicada, Universitat de Girona

External Reviewers

Kimberly Gero
Silvio Ghilardi
Simon Kramer

Table of Contents

Invited Papers

Extensible Symbolic System Analysis 1
Jose Meseguer

On the Limits of Second-Order Unification 5
Jordi Levy

Contributed Papers

Unification Modulo Common List Functions 15
Peter Hibbs, Paliath Narendran and Shweta Mehto

Matching with respect to general concept inclusions in the Description Logic EL 22
Franz Baader and Barbara Morawska

Unification in the normal modal logic Alt1 26
Philippe Balbiani and Tinko Tinchev

On Asymmetric Unification and the Combination Problem in Disjoint Theories (Extended Abstract) 33
Serdar Erbatur, Deepak Kapur, Andrew M. Marshall, Catherine Meadows, Paliath Narendran and Christophe Ringeissen

Hierarchical Combination of Matching Algorithms (Extended Abstract) 36
Serdar Erbatur, Deepak Kapur, Andrew M. Marshall, Paliath Narendran and Christophe Ringeissen

From Admissibility to a New Hierarchy of Unification Types 41
Leonardo Manuel Cabrer and George Mectalfe

Constraint Manipulation in SGGS 47
Maria Paola Bonacina and David Plaisted

Two-sided unification is NP-complete 55
Tatyana Novikova and Vladimir Zakharov

Nominal Anti-Unification 62
Alexander Baumgartner, Temur Kutsia, Jordi Levy and Mateu Villaret

A Categorical Perspective on Pattern Unification (Extended Abstract) 69
Andrea Vezzosi and Andreas Abel

Towards a better-behaved unification algorithm for Coq 74
Beta Ziliani and Matthieu Sozeau

Extensible Symbolic System Analysis*

José Meseguer

University of Illinois at Urbana-Champaign, USA

Abstract

Unification and narrowing are a key ingredient not only to solve equations modulo an equational theory, but also to perform symbolic system analysis. The key idea is that a concurrent system can be naturally specified as a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory specifying the system's states as an algebraic data type, and R specifies the system's concurrent, and often non-deterministic, transitions. One can perform such symbolic analysis by describing sets of states as (the instances of) terms with logical variables, and using narrowing *modulo* E to symbolically perform transitions. Under reasonable conditions on $\mathcal{R}$, this idea can be applied not only for complete reachability analysis, but also for temporal logic model checking. This approach is quite flexible but has some limitations. Could it be possible to make symbolic system analysis techniques more *extensible* and more widely applicable by simultaneously combining the powers of rewriting, narrowing, SMT solving and model checking? We give a progress report on current work aiming at such a unified symbolic approach.

1 Introduction

The automatic analysis of systems through model checking is one of the most successful system verification methods. The standard approaches (see, e.g., [13]) assume a finite-state system whose state space is exhaustively explored to check whether a system satisfies a desired temporal logic property. However, systems are often *infinite-state* in two possible ways (or are simultaneously infinite in these two ways):

1. The number of *initial states* is infinite, even though the set of states reachable from each initial state may be finite. For example, systems *parametric* in the number of processes or objects are of this kind.

2. The number of states *reachable* from an initial state is infinite. This often happens because states contain *unbounded data structures*.

To cope with these two sources of infinity two complementary methods can be used. On the one hand, *state abstraction* and *parametric system* techniques (see, e.g., [13]) can reduce the verification of infinite-state systems to that of finite-state ones. One the other hand, *infinite-state model checking* methods can be used, based on various kinds of *symbolic techniques* such as: (i) automata and grammars, e.g., [1, 12, 10, 11, 20, 23, 4, 3, 2]; (ii) SMT solving, e.g., [5, 14, 18, 19, 22, 26, 27]; and (iii) narrowing [25, 16, 17, 8, 9, 7].

We can think of these various infinite-state symbolic analysis techniques a *niches*, so that: (i) if a system specification can be cast within one of them, and (ii) if the chosen symbolic method can deal with the temporal logic property of interest (some methods only support *reachability analysis*, not general temporal logic model checking), then symbolic analysis is possible.

A key open research issue limiting the applicability of current symbolic techniques is lack of, or limited support for, *extensibility*. That is, although certain classes of systems can be

*Research partially supported by NSF Grant CNS 13-19109.

formalized in ways that allow the application of specific symbolic analysis techniques, many other systems of interest fall outside the scope of some existing symbolic techniques. In such cases one would like to *extend and combine* the power of symbolic techniques to analyze the given system. Indeed, it seems fair to say that at present we lack *general extensibility techniques for symbolic analysis* that can *simultaneously* combine the power of SMT solving, rewriting- and unification-based analysis, and automata-based model checking; and we lack tools that can apply them *together* to analyze a wide variety of systems beyond the scope of each separate analysis technique.

2 Towards Extensible Symbolic System Analysis

Several of us at the University of Illinois at Urbana-Champaign, SRI International, the Universitat Politécnica de of Valéncia, the Escuela Colombiana de Ingeniería, NASA Langley, the Naval Research Laboratory, and the University of Waterloo in Canada (more on this in the Acknowledgments) are currently working on developing the foundations and implementations of techniques that can simultaneously support symbolic analysis using SMT solving, rewriting/narrowing methods, and automata-based model checking.

More precisely, a concurrent system can be naturally specified as a *rewrite theory* [21] $\mathcal{R} = (\Sigma, E_0 \cup E, R)$ where: (i) $(\Sigma, E_0 \cup E)$ is an equational theory describing the system's states as an algebraic data type; and (ii) R is a collection of rewrite rules specifying the *system transitions*. Furthermore, we can often identify an equational subtheory $(\Sigma_0, E_0) \subseteq (\Sigma, E_0 \cup E)$ such that initial algebra $\mathcal{T}_{\Sigma_0/E_0}$ of (Σ_0, E_0) has a *decidable* first-order theory, whose satisfiability can be decided by an SMT solver, and, furthermore, the subtheory $(\Sigma_0, E_0) \subseteq (\Sigma, E_0 \cup E)$ is *protected* by the inclusion (i.e., we have an isomorphism $\mathcal{T}_{\Sigma_0/E_0} \cong \mathcal{T}_{\Sigma/E_0 \cup E|\Sigma_0}$). The extensible symbolic methods sketched above are methods to reason symbolically about the initial model $\mathcal{T}_{\mathcal{R}}$ of the rewrite theory $\mathcal{R}$, which in general may be the model of an infinite-state system.

The technical steps we are taking to achieve the goal of extensible symbolic analysis can be visualized, and placed in the context of existing work, by considering Figure 1 below.

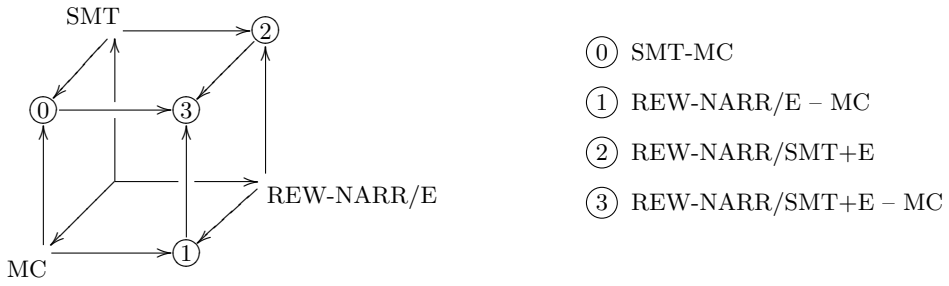


Figure 1: Combining techniques for extensible symbolic analysis

The three separate thrusts of symbolic analysis already mentioned, namely: (i) SMT solving, (ii) rewriting and unification-based techniques (modulo a theory E), and (iii) automata-based model checking are respectively abbreviated as the SMT, REW-NARR/E, and MC vectors in the cube. Vertices 0, 1, and 2 describe *pairwise combinations* obtainable as endpoints of vector additions for two of these three basic vectors. For example, vertex 0 describes SMT-based model checking, which is a very active area of research (see, e.g., [5, 14, 18, 19, 22, 26, 27]). Vertex 1 includes work on both rewriting-based model checking, e.g., [15, 6], and narrowing-based symbolic model checking , e.g., [25, 16, 17, 8, 9, 7]. Vertex 3 is the endpoint of adding the three

basic vectors, so that the joint power of the three symbolic analysis methods can be brought to bear on a much broader class of systems. A first, partial step towards reaching Vertex 3 is model checking based on *rewriting modulo SMT* [24], but the full power should be achieved through *narrowing modulo SMT* techniques currently under development.

Although such a combination of symbolic methods should make the analysis of systems much more extensible, there is already ample evidence from the work on narrowing-based model checking suggesting that symbolic techniques should be used in tandem with abstraction and other space state reduction techniques, which often remain necessary —or are in any case very useful even when not strictly needed— to make model checking decidable [17, 16, 8, 9, 7].

Acknowledgments

The development and implementation of these ideas, many of them ongoing, is joint work with various colleagues and Ph.D. students, including: Kyungmin Bae, Andrew Cholewa, Santiago Escobar, Steven Eker, Vijay Ganesh, Catherine Meadows, César Muñoz, Camilo Rocha, and Carolyn Talcott.

References

[1] P. Abdulla, B. Jonsson, P. Mahata, and J. dOrso. Regular tree model checking. In *Computer Aided Verification*, pages 452–466. Springer, 2002.

[2] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Workshop on Theory of Hybrid Systems*, pages 209–229. Springer LNCS 739, 1993.

[3] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[4] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009.

[5] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Model Checking Software*, pages 146–162, 2006.

[6] K. Bae and J. Meseguer. Model checking linear temporal logic of rewriting formulas under localized fairness. To appear in *Science of Computer Programming*, 2014.

[7] K. Bae and J. Meseguer. Predicate abstraction of rewrite theories. To appear in Proc. *RTA 2014*, Springer LNCS, 2014.

[8] Kyungmin Bae, Santiago Escobar, and Jose Meseguer. Abstract Logical Model Checking of Infinite-State Systems Using Narrowing. In *Rewriting Techniques and Applications (RTA'13)*, volume 21 of *LIPIcs*, pages 81–96. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.

[9] Kyungmin Bae, Santiago Escobar, and José Meseguer. Infinite-State Model Checking of LTLR Formulas Using Narrowing. In *Proc. WRLA 2014*. Springer LNCS, to appear, 2014.

[10] A. Bouajjani. Languages, rewriting systems, and verification of infinite-state systems. *Automata, Languages and Programming*, pages 24–39, 2001.

[11] A. Bouajjani and J. Esparza. Rewriting models of boolean programs. *Term Rewriting and Applications*, pages 136–150, 2006.

[12] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Computer Aided Verification*, pages 403–418. Springer, 2000.

[13] Edmund M. Clarke, Orna. Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2001.

[14] L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ansi-c software. In *ASE*, pages 137–148. IEEE, 2009.

[15] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2002.

[16] Santiago Escobar, Catherine Meadows, and José Meseguer. Maude-npa: Cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 2009.

[17] Santiago Escobar and José Meseguer. Symbolic model checking of infinite-state systems using narrowing. In *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 153–168, 2007.

[18] M.K. Ganai and A. Gupta. Accelerating high-level bounded model checking. In *ICCAD*, pages 794–801. ACM, 2006.

[19] M.K. Ganai and A. Gupta. Completeness in SMT-based BMC for software programs. In *DATE*, pages 831–836. IEEE, 2008.

[20] T. Genet and V. Tong. Reachability analysis of term rewriting systems with timbuk. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 695–706. Springer, 2001.

[21] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[22] A. Milicevic and H. Kugler. Model checking using SMT and theory of lists. *NASA Formal Methods*, pages 282–297, 2011.

[23] H. Ohsaki, H. Seki, and T. Takai. Recognizing boolean closed a-tree languages with membership conditional rewriting mechanism. In *Rewriting Techniques and Applications*, pages 483–498. Springer, 2003.

[24] Camilo Rocha, José Meseguer, and César Mu noz. Rewriting Modulo SMT and Open System Analysis. In *Proc. WRLA 2014*. Springer LNCS, to appear, 2014.

[25] P. Thati and J. Meseguer. Symbolic reachability analysis using narrowing and its application to the verification of cryptographic protocols. *J. Higher-Order and Symbolic Computation*, 20(1–2):123–160, 2007.

[26] M. Veanes, N. Bjørner, and A. Raschke. An SMT approach to bounded reachability analysis of model programs. In *FORTE*, pages 53–68. Springer, 2008.

[27] D. Walter, S. Little, and C. Myers. Bounded model checking of analog and mixed-signal circuits using an SMT solver. *Automated Technology for Verification and Analysis*, pages 66–81, 2007.

On the Limits of Second-Order Unification

Jordi Levy

Institut d'Investigació en Intel·ligència Artificia (IIIA-CSIC)
Barcelona, Spain
levy@iiia.csic.es

Abstract

Second-Order Unification is a problem that naturally arises when applying automated deduction techniques with variables denoting predicates. The problem is undecidable, but a considerable effort has been made in order to find decidable fragments, and understand the deep reasons of its complexity. Two variants of the problem, Bounded Second-Order Unification and Linear Second-Order Unification –where the use of bound variables in the instantiations is restricted–, have been extensively studied in the last two decades. In this paper we summarize some decidability/undecidability/complexity results, trying to focus on those that could be more interesting for a wider audience, and involving less technical details.

1 Introduction

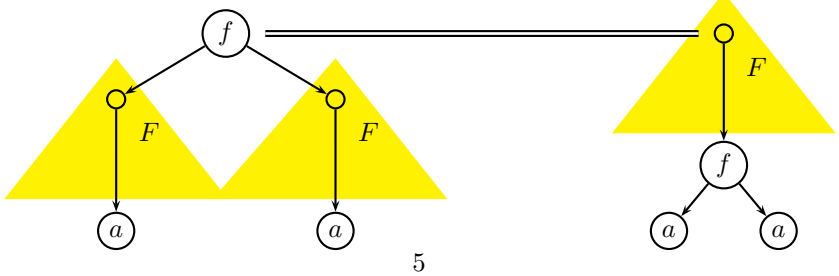
Unification consists on solving equations over expressions or terms. In the basic form of first-order unification, terms may contain (first-order) variables. When unifying, these variables may be replaced by terms, in order to make both sides of an equation $t \stackrel{?}{=} u$ syntactically equal. If we are not concerned with efficiency, we can decide first-order unifiability, and eventually compute the substitution or unifier, applying the following two transformations till we get an empty set of equations:

Simplification: $\{f(t_1, \ldots, t_n) \stackrel{?}{=} f(u_1, \ldots, u_n)\} \cup E \Rightarrow \{t_1 \stackrel{?}{=} u_1, \ldots, t_n \stackrel{?}{=} u_n\} \cup E$

Instantiation: $\{X \stackrel{?}{=} t\} \cup E \Rightarrow E\rho$
when X does not occur in t,
and the unifier is extended with the instantiation $\rho = [X \mapsto t]$

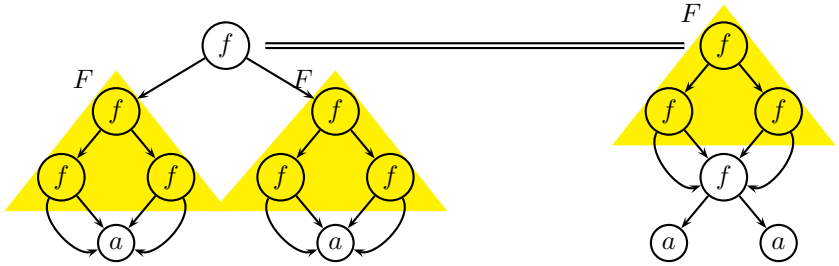
In second-order unification, variables may be applied to terms, and instances of variables may use these arguments, once or several times. Formally, variables are replaced by (second-order typed) lambda-terms, and then β-reduced. In general, Second-Order Unification allows the use of bound variables and binders also in equations. However, we know from experience that they do not make any remarkable difference with respect to the decidability or complexity of the problem. Therefore, for simplicity, we will avoid them in equations along this paper.

Consider the problem $f(F(a), F(a)) \stackrel{?}{=} F(f(a, a))$, where capital letters denote variables. One solution is $[F \mapsto \lambda x \,.\, x]$, that instantiates the equation as $f(a, a) = f(a, a)$.



But, there are other solutions like $[F \mapsto \lambda x\,.\,f(f(x,x),f(x,x))]$, that using the argument 4 times, instantiates the equation as:

$$f(f(f(a,a),f(a,a)),f(f(a,a),f(a,a))) = f(f(f(a,a),f(a,a)),f(f(a,a),f(a,a)))$$



In this example we can observe some of the properties that distinguishes second-order unification from first-order unification. First, the number of most general unifiers may be infinite. Second, equations of the form $F(t) \stackrel{?}{=} u$, where F occurs in u, that are unsolvable in first-order (occur-check), may be solvable in second-order (depending on the arguments). Notice also that, these type of equations, even when F does not occur in u, are not trivially solvable, because instances of F may use one or several times the argument t, and we must identify occurrences of t in u to find the value of F.

Depending on the number of times that the instance of a second-order variable may use their arguments, we distinguish three variants of second-order unification:

1. If there is no restriction on the number of times that a variable uses its arguments, we have *Second-Order Unification (SOU)*.
2. If the arguments are used exactly once, we have *Linear Second-Order Unification (LSOU)*. When, additionally, second-order variables are unary, we have *Context Unification (CU)*.
3. If the arguments may be used once or none, we have *Bounded Second-Order Unification (BSOU)*.

In the previous example $f(F(a),F(a)) \stackrel{?}{=} F(f(a,a))$, the substitution $[F \mapsto \lambda x\,.\,x]$ is a context unifier, and also a bounded second-order unifier, whereas $[F \mapsto \lambda x\,.\,f(f(x,x),f(x,x))]$ is not linear nor bounded. Despite this restriction, CU and BSOU are also infinitary, as the equation $F(f(a)) \stackrel{?}{=} f(F(a))$ and the infinite set of unifiers $\{[F \mapsto \lambda x\,.\,f(\stackrel{n}{\ldots}\,f(x)\ldots)]\}_{n\geq 0}$ show.

2 Some General Decidability and Undecidability Results

Second-Order Unification (SOU) was proved undecidable by Goldfarb (1981) by reducing the Hilbert's Tenth Problem. The more general case of Third-Order Unification was already proved undecidable, independently, by Lucchesi (1972) and Huet (1973). Pietrzykowski (1973) described the first complete second-order unification procedure, that was later extended to the higher-order case by Jensen and Pietrzykowski (1976). Gould (1966) was the first who found a complete second-order matching algorithm.

Context Unification (CU) was introduced independently by Comon (1993) and Schmidt-Schauß (1995). Comon (1993) studied the problem to solve membership constraints. He proved that context unification is decidable when any occurrence of the same context variable is always applied to the same term. Schmidt-Schauß (1995) was interested in reducing the

problem of unification modulo distributivity to a subset of such context unification problems. He proved that context unification is decidable when terms are *stratified*. By stratified we mean that the string of second-order variables we find going from the root of a term to any occurrence of the same variable, is always the same. Very recently, Jez (2014) has proved that CU is in PSPACE, answering a question that has remained open for 21 years.

Linear Second-Order Unification (LSOU) was introduced by Levy (1996). The generalization w.r.t. CU comes from two facts 1) we consider second-order terms, thus expressions may contain λ-bindings, and 2) second-order variables are not restricted to be unary like context variables. This generalization is motivated by the following example. The unification problem $F(a) \stackrel{?}{=} G(f(a))$ has two minimum linear second-order unifiers:

$$\sigma_1 = [F \mapsto \lambda x \,.\, G(f(x))]$$
$$\sigma_2 = [F \mapsto \lambda x \,.\, H(x, f(a))][G \mapsto \lambda x \,.\, H(a, x)]$$

However, if we restrict ourselves to unary second-order variables (context variables), we can not represent the second minimum unifier. Levy (1996) described a complete procedure for this problem and proved that the problem is decidable in the same cases studied in (Comon, 1993) and (Schmidt-Schauß, 1995), and also when no variable occurs more that twice. We will come back to this case later. Levy and Villaret (2000) proved that linear second-order unification can be reduced to context unification with tree-regular constraints, and commented on the possibility that linear second-order unification is decidable, if context unification is decidable (something that was unknown by that time, and should be revisited now). de Groote (2000) proved that Linear Higher-Order Matching is NP-complete.

Finally, Bounded Second-Order Unification was introduced and proved decidable by Schmidt-Schauß (2004). Later, Levy et al. (2006a) proved that BSOU is in fact NP-complete, using a similar technique as to prove that Monadic SOU is NP-complete (Levy et al., 2004).

Previous results analyze the general versions of these three variants of second-order unification. However, there are other papers where some restrictions on the classes of problems are studied. These results are also important because they shed light on the sources of complexity of these problems, and the possibility of finding (efficient) algorithms for some subclasses of problems. In the rest of this paper we will comment on some of these restrictions, focusing specially on those that could be interesting for a wide audience, and could help to understand the nature of this class of problems.

3 Pre-Unification

Huet (1975) introduced the notion *pre-unification*, a form of "lazy" unification, and found a non-redundant procedure for it. The idea is to forget about equations of the form $F(\dots) \stackrel{?}{=} G(\dots)$, called flex-flex, since SOU problems only containing such kind of equations are trivially solvable. SO pre-unification reminds first-order unification, but instead of instantiation rule, we have two new rules, imitation and projection.

Simplification: $\{f(t_1,\ldots,t_n) \stackrel{?}{=} f(u_1,\ldots,u_n)\} \cup E \Rightarrow \{t_1 \stackrel{?}{=} u_1,\ldots,t_n \stackrel{?}{=} u_n\} \cup E$

Imitation: $\{X(t_1,\ldots,t_n) \stackrel{?}{=} f(u_1,\ldots,u_m)\} \cup E \Rightarrow$
$\Big(\{X'(t_1,\ldots,t_n) \stackrel{?}{=} u_1,\ldots,X'(t_1,\ldots,t_n) \stackrel{?}{=} u_m\} \cup E\Big)\,\rho$
where $X'$are fresh variables, and the unifier is extended with the instantiation
$\rho = [X \mapsto \lambda y_1,\ldots,y_n\,.\,f(X'_1(y_1,\ldots,y_n),\ldots,X'_m(y_1,\ldots,y_n))]$

Projection: $\{X(t_1,\ldots,t_n) \stackrel{?}{=} f(u_1,\ldots,u_m)\} \cup E \Rightarrow$
$\Big(\{t_i \stackrel{?}{=} f(u_1,\ldots,u_m)\} \cup E\Big)\,\rho$
where $X'$are fresh variables, and the unifier is extended with the instantiation
$\rho = [X \mapsto \lambda y_1,\ldots,y_n\,.\,y_i]$

Pre-unification is complete for SOU, in the sense that a problem is solvable if, and only if, pre-unification leads to a set of flex-flex equations. However, some sequences of pre-unification transformations do not terminate. Pre-unification is also complete for BSOU, but we must re-adapt imitation rule in order to avoid repetition of bound variables. So, we instantiate $[X \mapsto \lambda y_1,\ldots,y_n\,.\,f(X'_1(y_{i_1},\ldots,y_{i_r}),\ldots,X'_m(y_{i_s},\ldots,y_{i_n}))]$, where $i_1,\ldots,i_n$ is a permutation of $1,\ldots,n$. Pre-unification is not complete for CU because some sets of flex-flex equations, like $\{X(a) \stackrel{?}{=} X(b)\}$, are not solvable (notice that instances of variables *must* use their arguments). Therefore, to get a complete procedure we must deal with flex-flex pairs, like is done in (Levy, 1996).

Word Unification is the problem of solving equations between words containing variables denoting words. The problem is decidable (Makanin, 1977), and in many aspects is quite similar to SOU. For instance, instances of variables may *overlap* with other occurrences of the same variable, and rise to an infinite number of most general solutions. For instance, the word unification equation $X \cdot a \stackrel{?}{=} a \cdot X$, has infinitely many solutions of the form $[X \mapsto a \cdot a \ldots a]$.

Non-termination of pre-unification is shown by the problem $X(f(a)) \stackrel{?}{=} f(X(a))$, that requires n imitation steps to generate the unifier $[X \mapsto \lambda y\,.\,f^n(y)]$. Notice that this problem is quite similar to the previous word unification problem $X \cdot a \stackrel{?}{=} a \cdot X$. To prove decidability of BSOU Schmidt-Schauß (2004) used the same technique as in word unification (see Makanin, 1977). He proved a *exponent of periodicity lemma* (see Schmidt-Schauß, 2004, Lemma 4.1), that states that we can (exponentially) bound the value of exponents in solutions without compromising solvability.

The possibility to choose between imitation and projection is also the responsible of NP-hardness of all variants of second-order unification. Typically (see Schmidt-Schauß, 2004), NP-hardness is proved by encoding 1-in-3SAT. We can interpret $[X \mapsto \lambda x\,.\,x]$ as false and $[X \mapsto \lambda x\,.\,f(x)]$ as true, and then encode clauses $X \vee Y \vee Z$ as $X(Y(Z(a))) \stackrel{?}{=} f(a)$, with the additional equation $X(f(a)) \stackrel{?}{=} f(X(a))$, in the case of BSOU and SOU, to ensure that all variables use their arguments.

4 Monadic Second-Order Unification

Farmer (1991) extended Golfarb's proof to prove that SOU is undecidable even if we restrict all second-order variables to be unary. Golfarb's proof requires the use of at least a binary function symbol, and this use is a crucial fact: Farmer (1988) also proved that SOU is decidable if all function symbols are at most unary (i.e. all constants are 0-ary or 1-ary), even if we do not restrict the arity of second-order variables. This fragment of SOU is call *Monadic Second-Order Unification (MSOU).*

Completeness of SOU (and BSOU and LSOU) decision procedures ensure that all constants occurring in a most general unifier, already occur in the original problem.[1] On the contrary, even if all original variables are unary, like in $X(a) \stackrel{?}{=} X(b)$, to represent the most general unifier $[X \mapsto \lambda x \,.\, Z(x,b), Y \mapsto \lambda y \,.\, Z(a,y)]$ we can require non-unary variables.

Levy and Villaret (2002, 2009) proved that the general forms of SOU, BSOU and LSOU can be reduced to a restricted form with just one binary function symbol. This is done by some sort of curryfication where terms like $f(g(a,b),c)$ are translated into $@(@(f,@(@(g,a),b)),c)$.

Levy et al. (2004, 2008) characterized the complexity of MSOU as NP-complete. To prove that MSOU is in NP, they showed how, for any solvable set of equations, we can represent at least one of the unifiers (in fact all size-minimal unifiers) in polynomial space. Then, they proved that we can check if a substitution (written in such representation) is a solution in polynomial time. There are two key in this proof: One is the result on the exponential upper bound on the exponent of periodicity of size-minimal unifiers. This upper bound allows us to represent exponents in linear space. The other key is a result of Plandowski (1994) where he proves that, given two context-free grammars with just one rule for every non-terminal symbol, we can check if they define the same language in polynomial time on the size of the grammars. This comprehension techniques, using context-free grammars generating singleton languages, and later using so called *tree context grammars* (Levy et al., 2006a), have been used to generalize matching and unification on compressed terms (see Gascón et al., 2008, 2009, for instance).

5 Two Occurrences per Variable

Word unification is trivially solvable when variables do not occur more than twice. In this case, when we apply a sort of imitation rule:

$$\textsf{Imitation:} \quad \{X \cdot w_1 \stackrel{?}{=} a \cdot w_2\} \cup E\} \Rightarrow \Big(\{X' \cdot w_1 \stackrel{?}{=} w_2\} \cup E\}\Big)\,\rho \quad \text{where } \rho = [X \mapsto a \cdot X']$$

the size of the problem does not increase. Notice that we remove an occurrence of a, and ρ introduces another occurrence of a, when instantiates the other occurrence of X. Similarly, Levy (1996) proves that the size of the problem does not increase during the execution of the LSOU procedure, when variables do not occur more than twice (see Levy, 1996, Theorem 3). Therefore, LSOU and CU are (trivially) decidable when no variable occurs more than twice. However, this is not the case for SOU.

Levy (1998) proved that SOU unification is undecidable even when no second-order variable occurs more than twice. The key in this proof is the observation that reachability can be encoded as SOU:

Given a ground term rewriting system $\{t_i \to u_i\}$, we can rewrite v into w, noted

$$t_1 \to u_1, \ldots, t_m \to u_m \vdash v \to w$$

if, and only if, the following SOU equation

$$X(f(a,v), u_1, \ldots, u_m) \stackrel{?}{=} f(X(a, t_1, \ldots, t_m), w)$$

is solvable, where f and a are symbols not used in the signature of the rewriting system.

[1] Alternatively, if a unifier contains constants not occurring in the original problem, we can replace them by fresh second-order variables with the same arity, obtaining a more general unifier.

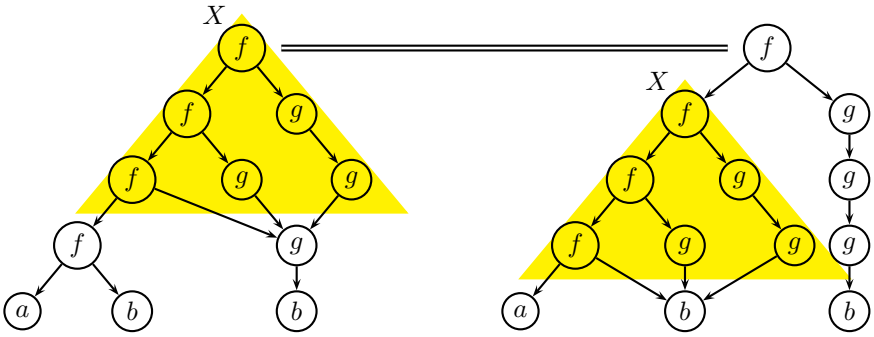
For instance, for the derivation:

$$b \to g(b) \vdash b \to g(b) \to g(g(b)) \to g(g(g(b)))$$

the equation would be:

$$X(f(a,b), g(b)) \stackrel{?}{=} f(X(a,b), g(g(g(b))))$$

and the instantiation of the equation as follow:



In this encoding, the number of times that a rewriting rule is used in the derivation corresponds to the number of times that the instance of X uses the corresponding argument. In fact, the unifier, if exists, encodes the sequence of rewriting steps. Ground reachability is a decidable problem, so this reduction does not proves undecidability of SOU.

If we allow the use of (first-order) variables in rewriting equations, but we restrict them to be instantiated always the same way, then we get a more general problem called *Rigid Reachability*. Notice that the rewriting rule $X \to a$ allows us to rewrite $f(b,c)$ into $f(a,a)$. However, $X \to a \vdash f(b,c) \to f(a,a)$ is an unsolvable rigid reachability problem (we cannot simultaneously instantiate X by b, and by c). If we allow the use of several rigid reachability equations we have *Simultaneous Rigid Reachability*. The undecidable of this problem was proved by Degtyarev and Voronkov (1996) by reduction of SOU. They consider a special kind of SOU equations, called *interpolation equations*, of the form $F(t_1, \ldots, t_n) \stackrel{?}{=} u$, where neither t_i nor u contain occurrences of second-order variables. By simplifying equations, and replacing subterms of the form $F(t_1, \ldots, t_n)$ by fresh first-order variables X and the equation $F(t_1, \ldots, t_n) \stackrel{?}{=} X$, we can express any SOU problem as a set of interpolation equations, without increasing the number of second-order variables, or its number of occurrences. Then, in Degtyarev and Voronkov's reduction, every interpolation equation $F(t_1, ..., t_n) \stackrel{?}{=} u$ is translated as $a_1 \to t_1, \ldots, a_n \to t_n \vdash X \to u$, where X is fresh.

We can reduce rigid reachability to SOU, like its is done for reachability. However, in this case we need some additional equations. For instance, the following rigid equation

$$c \to X \vdash g(c,c) \to g(d,e)$$

is unsolvable, whereas the corresponding SOU problem

$$F(f(a, g(c,c)), X) \stackrel{?}{=} f(F(a,c), g(d,e))$$

has a solution $[F \mapsto \lambda x, y \,.\, y][X \mapsto f(c, g(d,e))]$.

This problem can be easily fixed by adding a pair of equations $X \stackrel{?}{=} G_x(f_1(\overrightarrow{Y_x}), \ldots, f_N(\overrightarrow{Y_x})), b \stackrel{?}{=} G_x(b, \ldots, b)$, for all u_i that are variables X, where

$f_1, \ldots, f_N$ are all the function symbols of the rigid reachability signature. Notice that these additional equations disable X from being instantiated by a term with f in the head. Alternatively (see Levy and Veanes, 2000), we can add the reachability equation $f_1(a, \ldots, a) \to a, \ldots, f_N(a, \ldots, a) \vdash X \to a$, for any variable X, to ensure that instances of X do not contain f. This extension of Simultaneous Rigid Reachability is call *Guarded Simultaneous Rigid Reachability*. Guarded Simultaneous Rigid Reachability can be polynomially reduced to SOU where every second-order variable occurs twice. Levy and Veanes (1998) proved that Guarded Simultaneous Rigid Reachability is undecidable, even when restricted to systems of at most *two* reachability constraints. Therefore, SOU is undecidable under the following restrictions: there are at most two distinct second-order variables and two equations, every second-order variable occurs at most twice and in only one of the equations.

6 Just One Second-Order Variable

The number of different second-order variables in SOU plays a minor role compared to the total number of occurrences of second-order variables. Levy and Veanes (2000) present a straightforward reduction of arbitrary systems of second-order equations to equations using just one second-order variable and additional first-order variables.

Assume we have a system of SOU interpolation equations

$$\bigcup_{1 \le i \le m} \bigcup_{1 \le j \le k_i} F_i(t^1_{ij}, \ldots, t^n_{ij}) \stackrel{?}{=} u_{ij},$$

where $\{F_1, \ldots, F_m\}$ is the set of pair-wise distinct second-order variables, and every variable F_i has k_i occurrences. Assume without loss of generality that the arities of the F_i's are equal to n. If some F_i has smaller arity then increase the arity of F_i to the maximal arity replacing each $F_i(t^1_{ij}, \ldots, t^m_{ij})$ by $F_i(t^1_{ij}, \ldots, t^m_{ij}, \ldots, t^m_{ij})$, i.e. repeating the last argument as necessary. Then, we can reduce these equations to

$$\bigcup_{1 \le i \le m} \bigcup_{1 \le j \le k_i} G(t^1_{ij}, \ldots, t^n_{ij}) \stackrel{?}{=} g(\underbrace{\_, \ldots, \_}_{i-1}, u_{ij}, \underbrace{\_, \ldots, \_}_{m-i}),$$

where G is a fresh variable, g an appropriate constant, and "$\_$" denotes fresh and distinct first-order variables. If some of the t^k_{ij} is a variable, then we add also $G(c, \ldots, c) \stackrel{?}{=} g(\_, \ldots, \_)$. This reduction increases the number of occurrences of second-order variables in at most one. Notice also that the maximal arity of second-order variables and their set of arguments are preserved.

Combining this reduction with the results in previous section, we obtain that SOU is undecidable even for problems containing only one second-order variable occurring 4 times.

7 Restricting the Form of Arguments

One way to get a partial decidability result for SOU is to restrict the form of arguments of second-order variables. Levy and Veanes (2000, Corollary 15) proved that, even if we restrict arguments to be ground terms, SOU is undecidable. We have already seen that instances of second-order variables may "encode" sequences of transformations obtained from a rewriting system. The idea is to encode the execution steps of a universal Turing Machine, and reduce this way the Halting Problem. We represent the execution as a sequence of pairs of states

$$((v_1, v_1^+), (v_2, v_2^+), \ldots, (v_k, v_k^+)$$

where v^+ represents a successor state of v in the Turing Machine. Then, we use two equations, the first one has the form $F(\bar{t}, f(b,a)) \stackrel{?}{=} f(X, F(\overline{u}, a))$. It ensures that any solution satisfies $v_{i+i} = v_i^+$, because the instance of $F(\bar{t}, f(t', a))$ encodes $(v_1, \ldots, v_k, b)$, the instance of $f(X, F(\overline{u}, a))$ encodes $(X, v_1^+, \ldots, v_k^+)$, and the instance of X encodes the initial state. The second equation has the form $G(\bar{l}, f'(a, a')) \stackrel{?}{=} f'(F(\overline{v}, a), G(\overline{r}, a'))$. It ensures that v_i^+ is a valid successor of v_i in the TM. The sequence $\bar{l}$ and $\overline{r}$ encode the transitions of the Turing Machine, whereas $\bar{t}$, $\overline{u}$ and $\overline{v}$ only depends on the alphabet of the Turing Machine.

We obtain this way a system

$$\{F(\bar{t}, f(b,a)) \stackrel{?}{=} f(X, F(\overline{u}, a)), G(\bar{l}, f'(a, a')) \stackrel{?}{=} f'(F(\overline{v}, a), G(\overline{r}, a'))\}$$

that encodes the Halting Problem of a universal Turing Machine on input X. This proves undecidability of SOU for 5 occurrences of one second-order variable, even when the variable is only applied to ground terms.

A way to make SOU, or in general Higher-Order Unification, decidable is to restrict arguments of variables to be lists of pairwise distinct bound variables. This fragment is called *Higher-Order Patterns* and was proved decidable by Miller (1991). This restriction makes Higher-Order Pattern Unification equivalent to some sort of First-Order Unification with bindings and some kind of variable-capture restriction. In fact this extension of First-Order Unification has been studied extensively, and is called *Nominal Unification*. Urban et al. (2003) proved that Nominal Unification is decidable, and Levy and Villaret (2008) proved that it is in fact quadratic, by quadratic reduction to Higher-Order Patters Unification, that Qian (1996) proved decidable in linear time.

As we said, Jez (2014) decidability proof for CU closed a question open for more than 20 years. During this time, many other partial positive answers for CU have been found. For instance, the one-variable case (Gascón et al., 2010), the stratified case, used to prove decidability of unification modulo distributivity (Schmidt-Schauß, 2002; Levy et al., 2006b, 2011), the well-nested case, used in computational linguistics (Levy et al., 2005), the left-hole case, used to prove decidability of sequence unification (Kutsia et al., 2007, 2010), etc.

References

Comon, H., 1993. Completion of rewrite systems with membership constraints, part I: Deduction rules and part II: Constraint solving. Tech. rep., CNRS and LRI, Université de Paris Sud, (To appear in J. of Symbolic Computation).

de Groote, P., 2000. Linear higher-order matching is NP-complete. In: Proc. of the 11th Int. Conf. on Rewriting Techniques and Applications (RTA'00). Vol. 1833 of LNCS. pp. 127–140.

Degtyarev, A., Voronkov, A., 1996. The undecidability of simultaneous rigid E-unification. Theoretical Computer Science 166 (1-2), 291–300.

Farmer, W. M., 1988. A unification algorithm for second-order monadic terms. Annals of Pure and Applied Logic 39, 131–174.

Farmer, W. M., 1991. Simple second-order languages for wich unification is undecidable. Theoretical Computer Science 87, 173–214.

Gascón, A., Godoy, G., Schmidt-Schauß, M., 2008. Context matching for compressed terms. In: Proc. of the 23rd Annual IEEE Symposium on Logic in Computer Science (LICS'08). pp. 93–102.

Gascón, A., Godoy, G., Schmidt-Schauß, M., 2009. Unification with singleton tree grammars. In: Proc. of the 20th Int. Conf on Rewriting Techniques and Applications (RTA'09). Vol. 5595 of LNCS. pp. 365–379.

Gascón, A., Godoy, G., Schmidt-Schauß, M., Tiwari, A., 2010. Context unification with one context variable. J. Symb. Comput. 45 (2), 173–193.

Goldfarb, W. D., 1981. The undecidability of the second-order unification problem. Theoretical Computer Science 13, 225–230.

Gould, W. E., 1966. A matching procedure for ω-order logic. Ph.D. thesis, Princeton Univ.

Huet, G., 1973. The undecidability of unification in third-order logic. Information and Control 22 (3), 257–267.

Huet, G., 1975. A unification algorithm for typed λ-calculus. Theoretical Computer Science 1, 27–57.

Jensen, D. C., Pietrzykowski, T., 1976. Mechanizing ω-order type theory through unification. Theoretical Computer Science 3, 123–171.

Jez, A., 2014. Context unification is in PSPACE. In: Proc. of the 41st Int. Col. on Automata, Languages and Programming (ICAL'14).

Kutsia, T., Levy, J., Villaret, M., 2007. Sequence unification through currying. In: Proc. of the 18th Int. Conf. on Rewriting Techniques and Applications, RTA'07. Vol. 4533 of LNCS. Springer, pp. 288–302.

Kutsia, T., Levy, J., Villaret, M., 2010. On the relation between context and sequence unification. J. Symb. Comput. 45 (1), 74–95.

Levy, J., 1996. Linear second-order unification. In: Proc. of the 7th Int. Conf. on Rewriting Techniques and Applications (RTA'96). Vol. 1103 of LNCS. pp. 332–346.

Levy, J., 1998. Decidable and undecidable second-order unification problems. In: Proceedings of the 9th Int. Conf. on Rewriting Techniques and Applications (RTA'98). Vol. 1379 of LNCS. Tsukuba, Japan, pp. 47–60.

Levy, J., Niehren, J., Villaret, M., 2005. Well-nested context unification. In: Proc. of the 20th Int. Conf. on Automated Deduction, CADE-20. Vol. 3632 of LNCS. pp. 149–163.

Levy, J., Schmidt-Schauß, M., Villaret, M., 2004. Monadic second-order unification is *np*-complete. In: Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA'04). Vol. 3091 of LNCS. Aachen, Germany, pp. 55–69.

Levy, J., Schmidt-Schauß, M., Villaret, M., 2006a. Bounded second-order unification is NP-complete. In: Proceedings of the 17th International Conference on Rewriting Techniques and Applications (RTA'06). Vol. 4098 of LNCS. pp. 400–414.

Levy, J., Schmidt-Schauß, M., Villaret, M., 2006b. Stratified context unification is np-complete. In: Proceedings of the 3rd International Conference on Automated Reasoning (IJCAR 2006). Vol. 4130 of LNCS. Seattle, USA, pp. 82–96.

Levy, J., Schmidt-Schauß, M., Villaret, M., 2008. The complexity of monadic second-order unification. SIAM Journal on Computing 38 (3), 1113–1140.

Levy, J., Schmidt-Schauß, M., Villaret, M., 2011. On the complexity of bounded second-order unification and stratified context unification. Logic Journal of the IGPL 19 (6), 763–789.

Levy, J., Veanes, M., 1998. On unification problems in restricted second-order languages. In: Annual Conference of the European Association for Computer Science Logic (CSL'98).

Levy, J., Veanes, M., 2000. On the undecidability of second-order unification. Information and Computation 159, 125–150.

Levy, J., Villaret, M., 2000. Linear second-order unification and context unification with tree-regular constraints. In: Proc. of the 11th Int. Conf. on Rewriting Techniques and Applications (RTA'00). Vol. 1833 of LNCS. pp. 156–171.

Levy, J., Villaret, M., 2002. Currying second-order unification problems. In: Proc of the 13th Int. Conf. on Rewriting Techniques and Applications (RTA'02). Vol. 2378 of LNCS. pp. 326–339.

Levy, J., Villaret, M., 2008. Nominal unification from a higher-order perspective. In: Proc. of the 19$^{\text{th}}$ Int. Conf on Rewriting Techniques and Applications, RTA'08. Vol. 5117 of LNCS. pp. 246–260.

Levy, J., Villaret, M., 2009. Simplifying the signature in second-order unification. Appl. Algebra Eng. Commun. Comput. 20 (5-6), 427–445.

Lucchesi, C. L., 1972. The undecidability of the unification problem for third-order languages. Tech. Rep. CSRR 2059, Dept. of Applied Analysis and Computer Science, Univ. of Waterloo.

Makanin, G. S., 1977. The problem of solvability of equations in a free semigroup. Math. USSR Sbornik 32 (2), 129–198.

Miller, D., 1991. A logic programming language with lambda-abstraction, function variables, and simple unification. J. of Logic and Computation 1 (4), 497–536.

Pietrzykowski, T., 1973. A complete mechanization of second-order logic. J. of the ACM 20 (2), 333–364.

Plandowski, W., 1994. Testing equivalence of morphisms in context-free languages. In: ESA'94. Vol. 855 of LNCS. pp. 460–470.

Qian, Z., 1996. Unification of higher-order patterns in linear time and space. J. of Logic and Computation 6 (3), 315–341.

Schmidt-Schauß, M., 1995. Unification of stratified second-order terms. Tech. Rep. 12/94, Johan Wolfgang-Goethe-Universität, Frankfurt, Germany.

Schmidt-Schauß, M., 2002. A decision algorithm for stratified context unification. Journal of Logic and Computation 12, 929–953.

Schmidt-Schauß, M., 2004. Decidability of bounded second order unification. Information and Computation 188 (2), 143–178.

Urban, C., Pitts, A. M., Gabbay, M. J., 2003. Nominal unification. In: Proc. of the 17th Int. Work. on Computer Science Logic, CSL'03. Vol. 2803 of LNCS. pp. 513–527.

Unification Modulo Common List Functions

(Extended Abstract)

Peter Hibbs*, Paliath Narendran* and Shweta Mehto

Department of Computer Science
University at Albany–SUNY
Albany, NY 12222

1 Introduction

Reasoning about data types such as lists and arrays is an important research area with many applications, such as formal program verification [12, 10]. Early work on this [7] focused on proving inductive properties. Important outcomes of this work include the technique of *proof by consistency* or "inductionless induction" [13, 11] as well as *satisfiability modulo theories* (SMT) starting with the pioneering work of Nelson and Oppen [14] and of Shostak [16]. (See [1] for a recent syntactic, inference-rule based approach to developing SMT algorithms for lists and arrays.)

In our paper, we investigate the *unification* problem modulo various theories of lists. The constructors used in this paper are the usual nil and cons. The different theories are obtained by considering observer functions of increasing complexity. We first examine lists with only *right cons* (*rcons*) as an operator and propose a novel algorithm for the unification problem for this theory. We then introduce the theory of *reverse* (*rev*) and develop an algorithm to solve the unification problem over this theory. Lastly, we consider the unification problem modulo the theory of *fold right* or *reduce* which is of central importance in functional programming languages [9]. Note that in practice *reduce* is not a first-order function; we turn it into a first-order function by treating the binary function to be "folded" over the list as an uninterpreted function, i.e., as a constructor.

2 Definitions

The reader is assumed to be familiar with the concepts and notation used in [2]. For terminology and a more in-depth treatment of unification the reader is referred to [4]. Due to space constraints, we omit many proofs and details. Interested readers are referred to our technical report [8] in which much more detail is given.

The unification problems we consider are instances of *unification with constants* with some caveats. For instance, we only consider nil-terminated lists — this means that for any ground term X, the innermost element of X must be nil[1]. Constants of the list type are not permitted.

3 *rcons*

The equational axioms of this theory are

*supported in part by NSF grant CNS 09-05286.

[1] In LISP parlance, these are called *proper lists*.

$$\begin{aligned} rcons(\mathsf{nil}, x) &= \mathsf{cons}(x, \mathsf{nil}) \\ rcons(\mathsf{cons}(x, y), z) &= \mathsf{cons}(x, rcons(y, z)) \end{aligned}$$

We refer to this equational theory as RCONS and orient these from left to right to produce a convergent rewrite system.

In addition to requiring nil-terminated lists, we enforce the further restriction that the lists be *homogeneous* as in ML. We consider a *typed* theory here with a base sort of non-list elements. The lists we consider are of type list and may contain either list or non-list elements. Lists which do not themselves contain lists are called *flat lists*.

3.1 Complexity Analysis

Lemma 3.1. *Let s_1, s_2, t_1, t_2 be terms such that*

$$rcons(s_1, t_1) =_{\text{RCONS}} rcons(s_2, t_2).$$

Then $s_1 =_{\text{RCONS}} s_2$ and $t_1 =_{\text{RCONS}} t_2$.

Theorem 3.2. *Unifiability modulo* RCONS *is NP-hard.*

Proof Sketch. We will show this by reduction from monotone 1-in-3-SAT using the following device:

$$S_i : \ \mathsf{cons}(0, \mathsf{cons}(0, \mathsf{cons}(1, L_i))) =^? rcons(rcons(rcons(L_i, x_i), y_i), z_i)$$

Note that the solution to S_i must have exactly one of $\{x_i, y_i, z_i\}$ mapped to 1 and the others to 0. □

To show membership in NP we first consider the case where the lists are *flat*. Thus we only have two kinds of variables: variables of type list and variables of type non-list. We guess equivalence classes of all of the variables of type non-list. We consider these equivalence classes to be *discriminating*. That is, we fail on equations of the form $X =^? Y$ where X, Y are from different equivalence classes. All variables of type non-list may therefore be treated as constants.

Once this step is done, all equations are expressible in the following way:

$$\begin{aligned} &\mathsf{cons}(a_1, ...(\mathsf{cons}(a_n, rcons(rcons(...rcons(X, b_n)..., b_1))))) =^? \\ &\mathsf{cons}(c_1, ...(\mathsf{cons}(c_n, rcons(rcons(...rcons(Y, d_n), ..., d_1))))) \end{aligned}$$

with X and Y not necessarily distinct. We will denote the sequences $\{a_i\}$, $\{b_i\}$, $\{c_i\}$, $\{d_i\}$ with α, β, γ, δ respectively. We thus have to solve equations of the form $\alpha X \beta \ =^? \ \gamma Y \delta$.

Lemma 3.3. *The following algorithm can be used to solve the problem. Unification modulo* RCONS *is therefore NP-Complete*

1. For each equation in U, $\alpha X \beta =^? \gamma Y \delta$, remove all common prefixes and suffixes from both sides of that equation.

2. Select an arbitrary equation such that the variables on the right and left hand sides of the equation are distinct. If no such equation is available, proceed to Step 5.

3.a. If the equation is of the form $X =^? \alpha Y \beta$, replace all instances of X by $\alpha Y \beta$.

3.b. If the equation is of the form $\alpha X =^? Y\beta$, there is always a solution to X, Y of the form $X \mapsto Z\beta, Y \mapsto \alpha Z$, where Z is a fresh variable. If there exists some set of strings $\{u, v, w\}$ where $\alpha = uv$ and $\beta = vw$ where $v \neq \epsilon$ then there is an additional solution: $\{X \mapsto u, Y \mapsto w\}$. This solution is checked for validity. If there is more than one such set of strings, they are all checked. If no valid solution is found, replace all instances of X and Y with $Z\beta$ and αZ respectively. The number of variables is thereby reduced by 1.

4. Repeat from Step 1.

5. We are now left with only equations in solved form, and *independent* systems of equations each of which has only *one variable* in it. These can be checked for solvability using the algorithm in [5].

We omit the extension to *non-homogeneous, non-flat lists* here and direct the reader to [8].

4 *rev*

The term rewriting system we consider for nil-terminated lists is

$$\begin{array}{lrcl}
(1) & rcons(\mathsf{nil}, x) & \to & \mathsf{cons}(x, \mathsf{nil}) \\
(2) & rcons(\mathsf{cons}(x, y), z) & \to & \mathsf{cons}(x, rcons(y, z)) \\
(3) & rev(\mathsf{nil}) & \to & \mathsf{nil} \\
(4) & rev(\mathsf{cons}(x, y)) & \to & rcons(rev(y), x) \\
(5) & rev(rcons(x, y)) & \to & \mathsf{cons}(y, rev(x)) \\
(6) & rev(rev(x)) & \to & x
\end{array}$$

This system is convergent. We refer to this equational theory as REV. From this point on, we consider all terms to be in normal form modulo this term rewrite system.

Lemma 4.1. *Let s_1, s_2, t_1, t_2 be terms such that $rcons(s_1, t_1) =_{\text{REV}} rcons(s_2, t_2)$. Then $s_1 =_{\text{REV}} s_2$ and $t_1 =_{\text{REV}} t_2$.*

Lemma 4.2. *Let s_1, s_2 be terms such that $rev(s_1) =_{\text{REV}} rev(s_2)$. Then $s_1 =_{\text{REV}} s_2$.*

Lemma 4.3. *Unifiability modulo* REV *is NP-Complete.*

The NP-hardness proof given for unifiability modulo RCONS is equally valid for unifiability modulo REV. Membership in NP is shown by providing an algorithm to solve unification modulo REV which runs in NP time: we first guess equivalence classes of our variables as in the previous section. We then remove the 'highest' applications of *rev* in the dependency graph by applying the following inference rules:

(r1) $$\frac{\mathcal{EQ} \uplus \{X =^? rev(Y),\ X =^? \mathsf{cons}(W, Z)\}}{\mathcal{EQ} \cup \{X =^? \mathsf{cons}(W, Z),\ Y =^? rcons(Z', W),\ Z =^? rev(Z')\}}$$

(r2) $$\frac{\mathcal{EQ} \uplus \{X =^? rev(X),\ X =^? \mathsf{cons}(Y, Z)\}}{\mathcal{EQ} \cup \{X =^? \mathsf{cons}(Y, Z),\ Z =^? rcons(Z', Y),\ Z' =^? rev(Z')\}} \quad \text{if } Z \neq \mathsf{nil}$$

(r3) $$\frac{\mathcal{EQ} \uplus \{X =^? rev(X),\ X =^? \mathsf{cons}(Y, \mathsf{nil})\}}{\mathcal{EQ} \cup \{X =^? \mathsf{cons}(Y, \mathsf{nil})\}}$$

Each of the above rules (r1-r3) have analogous *rcons* rules which are very similar to the ones given here and are therefore omitted. After the above rules are applied to termination, the applications of *rev* exist only on the variables which correspond to leaf-nodes in the dependency graph. We now apply the rules of the flat case but once we have removed all equations of the form $\alpha X\beta =^? \alpha' Y\beta'$ where $X \neq Y$ and $Y \neq X^R$ where X^R denotes $rev(X)$, we then move on to *palindrome discovery*. In this step, we consider all equations of the form $\alpha X\beta =^? \alpha' X^R\beta'$. We maintain a list of variables that are known to be palindromes (i.e., where $X = X^R$) which is initially empty. We now have two cases:

> Case 1: $X =^? \alpha'' X^R\beta''$ in this case, if $|\alpha''\beta''| = 0$, then we conclude that X is a palindrome. Else, there can be no solution and we terminate with failure.
>
> Case 2: $\alpha'' X =^? X^R\beta''$. In this case, we check for the existence of a pair of strings $\{u, v\}$ such that $\alpha'' = u^R v, \beta = vu$. If such a pair exists, we check $X = u$ for consistency. If all such pairs are checked without finding a solution, then we default to the substitution $X = Z\beta''$, $X^R = \alpha'' Z$ where Z is known to be a palindrome. If $\beta'' \neq \alpha''^R$, then there is no solution and we fail. Otherwise we replace all occurrences of X with $Z\beta''$.

Once we have finished this, we only have equations of the form $\alpha X =^? X\beta$. If X is not a palindrome, then we may use the algorithm given in [5] to find a solution for it. If X is known to be a palindrome, then we may still run the algorithm given in [5] to check for a solution, but first check that the prefixes and suffixes of each equation (i.e., α, β) meet certain criteria:

Lemma 4.4. *Let α, β and A be non-empty strings such that A is a palindrome and $|\alpha| = |\beta| < |A|$. Then $\alpha A = A\beta$ if and only if there exist palindromes u, v, and a positive integer k such that $\alpha = uv$, $\beta = vu$ and $A = (uv)^k u$.*

Proof Sketch. This follows from the well-known result that for any equation $\alpha A = A\beta$ where $0 < |\alpha| = |\beta| < |A|$, α and β must be *conjugates* or there can be no solution. □

So, if the elements in the set of equations satisfy this constraint, then any solution must be a palindrome. Thus it is sufficient to check for the existence of appropriate u, v and then apply the algorithm of [5].

Lemma 4.5. *The above algorithm terminates*

Proof Sketch. The algorithm begins by applying inference rules (r1-r3) to termination. Each of these rules either lowers some application of rev further down in the dependency graph or deletes it outright. Because the set of input equations is finite, eventually all applications of rev must lie on the leaf-nodes of the graph and no further lowering can occur. The algorithm then removes all equations of the form $\alpha X\beta =^? \alpha' Y\beta'$ where $X \neq Y$ and $Y \neq X^R$ which terminates by the argument given in the statement of this procedure in Section 3.1. We then move on to the palindrome discovery step which removes an equation of the form $\alpha X\beta =^? \alpha' X^R\beta'$ in each iteration. Finally, we apply the algorithm given in [5] which terminates by assumption. □

5 *reduce*

The standard definition of *reduce* (for a particular two-argument function f) is given by the following rewrite rules:

$$\begin{aligned} reduce(\mathsf{nil}, x) &\rightarrow x \\ reduce(\mathsf{cons}(u, v), x) &\rightarrow f(u,\, reduce(v, x)) \end{aligned}$$

Since we consider only nil-terminated lists, we extend the signature of the theory with the *append* function @ and a monadic function g which creates *singleton* lists. This extended theory has the following convergent rewrite system:

$$\begin{array}{lrcl}
(1) & f(x,z) & \to & reduce(\mathsf{g}(x),z)\\
(2) & \mathsf{cons}(x,y) & \to & \mathsf{g}(x)\ @\ y\\
(3) & reduce(\mathsf{nil},z) & \to & z\\
(4) & reduce(x,reduce(y,z)) & \to & reduce(x\ @\ y,z)\\
(5) & \mathsf{nil}\ @\ x & \to & x\\
(6) & x\ @\ \mathsf{nil} & \to & x\\
(7) & (x\ @\ y)\ @\ z & \to & x\ @\ (y\ @\ z)
\end{array}$$

Note that $\mathsf{g}(x)$ is equivalent to $\mathsf{cons}(x, \mathsf{nil})$. We impose a type system for this equational theory. There are two types: `list` and `nonlist`. Under this type system the unification problem $\{reduce(X,Y) =^? \mathsf{cons}(U,V)\}$, for example, will result in a type-failure. Unification modulo this theory is at least as hard as the word equation problem, which is NP-hard and in PSPACE [15].

We now outline the algorithm to solve the unification problem modulo the extended theory. We assume the input equations are in standard form. We also assume that all instances of the function symbols f and cons are eliminated using the rewrite rules (1) and (2).

Let S be the set of `list` type variables. As in Section 3.1, we nondeterministically guess a partition of equivalence classes among all variables. We guess an ordering $\succ$ on the `list` type equivalence classes such that $X \succ Y$ if the length of X is larger than the length of Y, where the length of a variable Z refers to the number of instances of cons in Z. All list variables in the same equivalence class as nil must be equivalent to nil and clearly the partition containing nil must be a least element in the ordering $\succ$. We also nondeterministically guess an ordering $\gg$ on the `nonlist` variables, just as with the `list` variables, such that $X \gg Y$ if and only if the size of X after substitution is greater than the size of Y. This ordering is clearly acyclic and well-founded.

Lemma 5.1. *If $X = reduce(Y,Z)$ and Y is not equivalent to* nil *then $X \gg Z$.*

Proof Sketch. We prove this by induction on the length of the `list` variable Y. □

From this point on, if at any time in the algorithm an equation violates a type constraint or an ordering constraint, we terminate with failure. The inference rules for those failures are not included. We apply rewrite rules (3), (5) and (6) to remove equations involving nil. After this, once nils are eliminated, the problem boils down to unification modulo the rules

$$\begin{array}{lrcl}
(4) & reduce(x,reduce(y,z)) & \to & reduce(x\ @\ y,z)\\
(7) & (x\ @\ y)\ @\ z & \to & x\ @\ (y\ @\ z)
\end{array}$$

No rule has nil on the right-hand side (thus new instances of nil will not be produced) and, since g does not occur in these rewrite rules, the problem is now a *general* unification problem.

We construct a dependency graph for our unification problem U. If this graph contains a cycle, then clearly U is not unifiable unless the above theory is subterm-collapsing which it is not. Thus, if there is a cycle we terminate with failure. The main inference rule is

$$\textbf{(5)} \qquad \frac{\mathcal{EQ} \uplus \{X =^? reduce(Y,Z),\ X =^? reduce(V,W)\}}{\mathcal{EQ} \cup \{X =^? reduce(Y,Z),\ Y =^? V@Y',\ W = reduce(Y',Z)\}} \quad \text{if } Y \succ V$$

Note that we introduce a (possibly) new `list`-type variable Y' in rule (5). At that point we nondeterministically include Y' into the ordering $\succ$. (We omit failure rules here.) The only equations now left that are not in solved form are equations of the form $X =^? Y@Z$. Thus the set of equations we get is an instance of the *general* associative unification problem, which is decidable [3].

The termination and correctness of this algorithm is given in the technical report [8].

6 Conclusions

We have shown that unification of lists modulo the observer functions *rcons* and *rev* is NP-complete. Our algorithm for unification modulo *reduce* requires solving the general associative unification problem, and the algorithm for the latter makes use of an algorithm for the word equation problem with rational (regular) constraints. This problem (i.e., word equations with rational constraints) has been shown to be solvable in PSPACE [6] and our algorithm therefore requires no more than PSPACE complexity. The lower bound on the complexity of this last problem is an open question.

Acknowledgements: We wish to thank Dan DiTursi, Kim Gero, Wojciech Plandowski, Manfred Schmidt-Schauß and the referees for their helpful comments and suggestions.

References

[1] A. Armando, S. Ranise, M. Rusinowitch. Uniform Derivation of Decision Procedures by Superposition. *Lecture Notes in Computer Science* 2142: 513–527 (2001)

[2] F. Baader, T. Nipkow. *Term Rewriting and All That.* Cambridge Univ Press, 1999.

[3] F. Baader and K.U. Schultz. Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures. *Proceedings of the Eleventh Conference on Automated Deduction (CADE-11),* Saratoga Springs, New York, *Lecture Notes in Artificial Intelligence* **607** (Springer, Berlin, 1992) 50–65.

[4] F. Baader, W. Snyder. Unification Theory. In: John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.

[5] R. Dabrowski, W. Plandowski. On Word Equations in One Variable. *Algorithmica* 60(4): 819–828 (2011).

[6] V. Diekert, A. Je, W. Plandowski. Finding All Solutions of Equations in Free Groups and Monoids with Involution. *arXiv* :1405.5133 [cs.LO]

[7] J.V. Guttag, E. Horowitz, D.R. Musser. Abstract Data Types and Software Validation. *Commun. ACM* 21(12): 1048–1064 (1978).

[8] P. Hibbs, P. Narendran, S. Mehto. Unification Modulo Common List Functions. Technical Report SUNYA-CS-14-01, available at: `www.cs.albany.edu/~ncstrl/treports/Data/`

[9] G. Hutton. A tutorial on the universality and expressiveness of *fold*. *Journal of Functional Programming* 9(4): 355–372 (1999)

[10] D. Kapur. *Towards A Theory For Abstract Data Types.* Doctoral Dissertation, Massachusetts Institute of Technology, 1980.

[11] D. Kapur, D.R. Musser. Proof by Consistency. *Artif. Intell.* 31(2): 125–157 (1987).

[12] D.R. Musser. Abstract Data Type Specification in the AFFIRM System. *IEEE Trans. Software Eng.* 6(1): 24–32 (1980).

[13] D.R. Musser. On Proving Inductive Properties of Abstract Data Types. In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages* (POPL) 154–162 (1980).

[14] G. Nelson, D.C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems* 1(2): 245–257 (1979).

[15] W. Plandowski. An Efficient Algorithm For Solving Word Equations. In: *Proceedings of the ACM Symposium on the Theory of Computing '06*, 467–476 (2006).

[16] R.E. Shostak. Deciding Combinations of Theories. *J. ACM* 31(1): 1–12 (1984)

Matching with respect to general concept inclusions in the Description Logic $\mathcal{EL}$

Franz Baader and Barbara Morawska*
{baader,morawska}@tcs.inf.tu-dresden.de

Theoretical Computer Science, TU Dresden, Germany

Abstract

Matching concept descriptions against concept patterns was introduced as a new inference task in Description Logics (DLs) almost 20 years ago, motivated by applications in the Classic system. For the DL $\mathcal{EL}$, it was shown in 2000 that the matching problem is NP-complete. It then took almost 10 years before this NP-completeness result could be extended from matching to unification in $\mathcal{EL}$. The next big challenge was then to further extend these results from matching and unification without a TBox to matching and unification w.r.t. a general TBox, i.e., a finite set of general concept inclusions. For unification, we could show some partial results for general TBoxes that satisfy a certain restriction on cyclic dependencies between concepts, but the general case is still open. For matching, we were able to solve the general case: we can show that matching in $\mathcal{EL}$ w.r.t. general TBoxes is NP-complete. We also determine some tractable variants of the matching problem.

1 Introduction

The DL $\mathcal{EL}$, which offers the constructors conjunction ($\sqcap$), existential restriction ($\exists r.C$), and the top concept ($\top$), has recently drawn considerable attention since, on the one hand, important inference problems such as the subsumption problem are polynomial in $\mathcal{EL}$, even in the presence of general concept inclusions (GCIs) [11]. On the other hand, though quite inexpressive, $\mathcal{EL}$ can be used to define biomedical ontologies, such as the large medical ontology SNOMED CT.[1]

Matching of concept descriptions against concept patterns is a non-standard inference task in Description Logics, which was originally motivated by applications of the Classic system [8]. In [10], Borgida and McGuinness proposed matching as a means to filter out the unimportant aspects of large concept descriptions appearing in knowledge bases of Classic. Subsequently, matching (as well as the more general problem of unification) was also proposed as a tool for detecting redundancies in knowledge bases [7] and to support the integration of knowledge bases by prompting possible interschema assertions to the integrator [9].

All three applications have in common that one wants to search the knowledge base for concepts having a certain (not completely specified) form. This "form" can be expressed with the help of so-called *concept patterns*, i.e., concept descriptions containing variables (which stand for descriptions). For example, assume that we want to find concepts that are concerned with individuals having a son and a daughter sharing some characteristic. This can be expressed by the pattern $D := \exists\mathsf{has\text{-}child}.(\mathsf{Male} \sqcap X) \sqcap \exists\mathsf{has\text{-}child}.(\mathsf{Female} \sqcap X)$, where X is a variable standing for the common characteristic. The concept description $C := \exists\mathsf{has\text{-}child}.(\mathsf{Tall} \sqcap \mathsf{Male}) \sqcap \exists\mathsf{has\text{-}child}.(\mathsf{Tall} \sqcap \mathsf{Female})$ matches this pattern in the sense that, if we replace the variable X by the description Tall, the pattern becomes *equivalent* to the description. Thus, the substitution $\sigma := \{X \mapsto \mathsf{Tall}\}$ is a *matcher modulo equivalence* of the matching problem $C \equiv^? D$ since

*Supported by DFG under grant BA 1122/14-2
[1] see http://www.ihtsdo.org/snomed-ct/

$C \equiv \sigma(D)$. The original paper by Borgida and McGuinness actually considered matching modulo subsumption rather than matching modulo equivalence: such a problem is of the form $C \sqsubseteq^? D$, and a matcher τ is a substitution τ satisfying $C \sqsubseteq \tau(D)$. Obviously, any matcher modulo equivalence is also a matcher modulo subsumption, but not vice versa. For example, the substitution $\sigma_\top := \{X \mapsto \top\}$ is a *matcher modulo subsumption* of the matching problem $C \sqsubseteq^? D$, but it is not a matcher modulo equivalence.

The first results on matching in DLs were concerned with sublanguages of the Classic description language, which does not allow for existential restrictions of the kind used in our example. A polynomial-time algorithm for computing matchers modulo subsumption for a rather expressive DL was introduced in [10]. The main drawback of this algorithm was that it required the concept patterns to be in structural normal form, and thus it was not able to handle arbitrary matching problems. In addition, the algorithm was incomplete, i.e., it did not always find a matcher, even if one existed. For the DL $\mathcal{ALN}$, a polynomial-time algorithm for matching modulo subsumption and equivalence was presented in [5]. This algorithm is complete and it applies to arbitrary patterns. In [4], matching in DLs with existential restrictions was investigated for the first time. In particular, it was shown that in $\mathcal{EL}$ the matching problem (i.e., the problem of deciding whether a given matching problem has a matcher or not) is polynomial for matching modulo subsumption, but NP-complete for matching modulo equivalence.

Unification is a generalization of matching where both sides of the problem are patterns and thus the substitution needs to be applied to both sides. In [7] it was shown that the unification problem in the DL $\mathcal{FL}_0$, which offers the constructors conjunction ($\sqcap$), value restriction ($\forall r.C$), and the top concept ($\top$), is ExpTime-complete. In contrast, unification in $\mathcal{EL}$ is "only" NP-complete [6]. In the results for matching and unification mentioned until now, there was no TBox involved, i.e., equivalence and subsumption was considered with respect to the empty TBox. For unification in $\mathcal{EL}$, first attempts were made to take *general TBoxes*, i.e., finite sets of general concept inclusions (GCIs), into account. However, the results obtained so far, which are again NP-completeness results, are restricted to general TBoxes that satisfy a certain restriction on cyclic dependencies between concepts [2, 3].

For matching, we were able to solve the general case: matching in $\mathcal{EL}$ w.r.t. general TBoxes is NP-complete. The matching problems considered in this paper are actually generalizations of matching modulo equivalence and matching modulo subsumption. For the special case of matching modulo subsumption, we show that the problem is tractable also in the presence of GCIs. The same is true for the dual problem where the pattern is on the side of the subsumee rather than on the side of the subsumer.

Due to space constraints, we cannot provide proofs of our results. They can be found in [1].

2 The Description Logics $\mathcal{EL}$

The expressiveness of a DL is determined both by the formalism for describing concepts (the concept description language) and the terminological formalism, which can be used to state additional constraints on the interpretation of concepts and roles in a so-called TBox.

The *concept description language* considered in this paper is called $\mathcal{EL}$. Starting with a finite set N_C of *concept names* and a finite set N_R of *role names*, *$\mathcal{EL}$-concept descriptions* are built from concept names using the constructors *conjunction* ($C \sqcap D$), *existential restriction* ($\exists r.C$ for every $r \in N_R$), and *top* ($\top$). Since in this paper we only consider $\mathcal{EL}$-concept descriptions, we will sometimes dispense with the prefix $\mathcal{EL}$.

On the *semantic side*, concept descriptions are interpreted as sets. To be more precise, an *interpretation* $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ consists of a non-empty domain $\Delta^\mathcal{I}$ and an interpretation function

$\cdot^{\mathcal{I}}$ that maps concept names to subsets of $\Delta^{\mathcal{I}}$ and role names to binary relations over $\Delta^{\mathcal{I}}$. This function is inductively extended to concept descriptions as follows:

$$\top^{\mathcal{I}} := \Delta^{\mathcal{I}}, \quad (C \sqcap D)^{\mathcal{I}} := C^{\mathcal{I}} \cap D^{\mathcal{I}}, \quad (\exists r.C)^{\mathcal{I}} := \{x \mid \exists y : (x,y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$$

A *general concept inclusion axiom (GCI)* is of the form $C \sqsubseteq D$ for concept descriptions C, D. An interpretation $\mathcal{I}$ *satisfies* such an axiom $C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. A *general $\mathcal{EL}$-TBox* is a finite set of GCIs. An interpretation is a *model* of a general $\mathcal{EL}$-TBox if it satisfies all its GCIs.

A concept description C is *subsumed* by a concept description D w.r.t. a general TBox $\mathcal{T}$ (written $C \sqsubseteq_{\mathcal{T}} D$) if every model of $\mathcal{T}$ satisfies the GCI $C \sqsubseteq D$. We say that C is *equivalent* to D w.r.t. $\mathcal{T}$ ($C \equiv_{\mathcal{T}} D$) if $C \sqsubseteq_{\mathcal{T}} D$ and $D \sqsubseteq_{\mathcal{T}} C$. If $\mathcal{T}$ is empty, we also write $C \sqsubseteq D$ and $C \equiv D$ instead of $C \sqsubseteq_{\mathcal{T}} D$ and $C \equiv_{\mathcal{T}} D$, respectively. As shown in [11], subsumption w.r.t. general $\mathcal{EL}$-TBoxes is decidable in polynomial time.

3 Matching in $\mathcal{EL}$

In addition to the set N_C of concept names (which must not be replaced by substitutions), we introduce a set N_V of concept variables (which may be replaced by substitutions). *Concept patterns* are now built from concept names and concept variables by applying the constructors of $\mathcal{EL}$. A *substitution* σ maps every concept variable to an $\mathcal{EL}$-concept description. It is extended to concept patterns in the usual way:

- $\sigma(A) := A$ for all $A \in N_C \cup \{\top\}$,
- $\sigma(C \sqcap D) := \sigma(C) \sqcap \sigma(D)$ and $\sigma(\exists r.C) := \exists r.\sigma(C)$.

An $\mathcal{EL}$-concept pattern C is *ground* if it does not contain variables, i.e., if it is a concept description. Obviously, a ground concept pattern is not modified by applying a substitution.

Definition 3.1. *Let $\mathcal{T}$ be a general $\mathcal{EL}$-TBox.*[2] *An* $\mathcal{EL}$-matching problem *w.r.t. $\mathcal{T}$ is a finite set $\Gamma = \{C_1 \sqsubseteq^? D_1, \ldots, C_n \sqsubseteq^? D_n\}$ of subsumptions between $\mathcal{EL}$-concept patterns, where for each $i, 1 \leq i \leq n$, C_i or D_i is ground. A substitution σ is a* matcher *of Γ w.r.t. $\mathcal{T}$ if σ* solves *all the subsumptions in Γ, i.e. if $\sigma(C_1) \sqsubseteq_{\mathcal{T}} \sigma(D_1), \ldots, \sigma(C_n) \sqsubseteq_{\mathcal{T}} \sigma(D_n)$. We say that Γ is* matchable *w.r.t. $\mathcal{T}$ if it has a matcher.*

Matching problems modulo equivalence and subsumption are special cases of the matching problems introduced above:

- The $\mathcal{EL}$-matching problem Γ is a *matching problem modulo equivalence* if $C \sqsubseteq^? D \in \Gamma$ implies $D \sqsubseteq^? C \in \Gamma$. This coincides with the notion of matching modulo equivalence considered in [5, 4], but extended to a non-empty general TBox.
- The $\mathcal{EL}$-matching problem Γ is a *left-ground matching problem modulo subsumption* if $C \sqsubseteq^? D \in \Gamma$ implies that C is ground. This coincides with the notion of matching modulo subsumption considered in [5, 4], but again extended to a non-empty general TBox.
- The $\mathcal{EL}$-matching problem Γ is a *right-ground matching problem modulo subsumption* if $C \sqsubseteq^? D \in \Gamma$ implies that D is ground. To the best of our knowledge, this notion of matching has not been investigated before.

The general case of matching, as introduced in Definition 3.1, and thus also matching modulo equivalence, is NP-complete, whereas the two notions of matching modulo subsumption are tractable, even in the presence of GCIs.

[2] Note that the GCIs in $\mathcal{T}$ are built using concept descriptions, and thus do not contain variables.

Theorem 3.2. *Let Γ be an $\mathcal{EL}$-matching problem and $\mathcal{T}$ a general $\mathcal{EL}$-TBox. Deciding whether Γ has a matcher w.r.t. $\mathcal{T}$ is*

1. *polynomial if Γ is a left-ground or a right-ground matching problem modulo subsumption;*
2. *NP-complete in the general case.*

A detailed proof of this theorem can be found in [1]. Basically, the results for the case of matching modulo subsumption are proved as follows: in each case we define a specific substitution, and show that the matching problem has a matcher iff this substitution is a matcher. NP-hardness for the general case follows from the known NP-hardness result for matching modulo equivalence without a TBox. The NP-upper bound can be shown by introducing a goal-oriented matching algorithm that uses nondeterministic rules to transform a given matching problem into a solved form by a polynomial number of rule applications.

References

[1] Franz Baader, , and Barbara Morawska. Matching with respect to general concept inclusions in the description logic $\mathcal{EL}$. LTCS-Report 14-03, Chair of Automata Theory, Institute of Theoretical Computer Science, Technische Universität Dresden, Dresden, Germany, 2014. See http://lat.inf.tu-dresden.de/research/reports.html.

[2] Franz Baader, Stefan Borgwardt, and Barbara 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.

[3] Franz Baader, Stefan Borgwardt, and Barbara Morawska. A goal-oriented algorithm for unification in $\mathcal{ELH}_{R^+}$ w.r.t. cycle-restricted ontologies. In Michael Thielscher and Dongmo Zhang, editors, *Pro. of 25th Australasian Joint Conf. on Artificial Intelligence (AI'12)*, volume 7691 of *Lecture Notes in Artificial Intelligence*, pages 493–504. Springer-Verlag, 2012.

[4] Franz Baader and Ralf Küsters. Matching in description logics with existential restrictions. In *Proc. of the 7th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2000)*, pages 261–272, 2000.

[5] Franz Baader, Ralf Küsters, Alex Borgida, and Deborah L. McGuinness. Matching in description logics. *J. of Logic and Computation*, 9(3):411–447, 1999.

[6] Franz Baader and Barbara Morawska. Unification in the description logic $\mathcal{EL}$. *Logical Methods in Computer Science*, 6(3), 2010.

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

[8] Alexander Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Lori Alperin Resnick. CLASSIC: A structural data model for objects. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 59–67, 1989.

[9] Alexander Borgida and Ralf Küsters. What's not in a name? Initial explorations of a structural approach to integrating large concept knowledge-bases. Technical Report DCS-TR-391, Rutgers University, 1999.

[10] Alexander Borgida and Deborah L. McGuinness. Asking queries about frames. In *Proc. of the 5th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR'96)*, pages 340–349, 1996.

[11] Sebastian Brandt. Polynomial time reasoning in a description logic with existential restrictions, GCI axioms, and—what else? In Ramon López de Mántaras and Lorenza Saitta, editors, *Proc. of the 16th Eur. Conf. on Artificial Intelligence (ECAI 2004)*, pages 298–302, 2004.

Unification in the normal modal logic Alt_1

Philippe Balbiani[1] and Tinko Tinchev[2]

[1]Institut de recherche en informatique de Toulouse
CNRS — Université de Toulouse
[2]Department of Mathematical Logic and Applications
Sofia University

1 Introduction

The unification problem in a logical system L can be defined in the following way: given a formula $\phi(x_1, \ldots, x_\alpha)$, determine whether there exists formulas ψ_1, ..., ψ_α such that $\phi(\psi_1, \ldots, \psi_\alpha)$ is in L. The research on unification for modal logics was originally motivated by the admissibility problem for rules of inference: given a rule of inference $\phi_1(x_1, \ldots, x_\alpha), \ldots, \phi_m(x_1, \ldots, x_\alpha)/\psi(x_1, \ldots, x_\alpha)$, determine whether for all formulas χ_1, ..., χ_α, if $\phi_1(\chi_1, \ldots, \chi_\alpha)$, ..., $\phi_m(\chi_1, \ldots, \chi_\alpha)$ are in L then $\psi(\chi_1, \ldots, \chi_\alpha)$ is in L [1]. Within the context of description logics, the main motivation for investigating the unification problem was to propose new reasoning services in the maintenance of knowledge bases like, for example, the elimination of redundancies in the descriptions of concepts [2].

Combining algebraic and model-theoretic methods, Rybakov [7] demonstrated that the admissibility problem and the unification problem in intuitionistic propositional logic and modal logic $S4$ are decidable. Later on, Ghilardi [4], proving that intuitionistic propositional logic has a finitary unification type, yielded a new solution of the admissibility problem, seeing that determining whether a given rule of inference preserves validity in intuitionistic propositional logic is equivalent to checking whether the finitely many maximal unifiers of its premises are unifiers of its conclusion. These results incited researchers to determine whether there exists finitely many admissible rules of inference of intuitionistic propositional logic and modal logic $S4$ so that the remaining admissible rules of inference would be derivable from them [5].

With respect to the issue of computational complexity, the admissibility problem and the unification problem were mostly unexplored before the work of Jerábek [6] who established the $coNEXPTIME$-completeness of the admissibility problem for several intuitionistic and modal logics extending $K4$ such as $S4$ and GL, in contrast with the satisfiability problem for these logics which is usually $PSPACE$-complete and in contrast with the unification problem for modal logics contained in $K4$ which is undecidable if one considers a language with the universal modality [8]. One may ask whether the situation is getting better if the language is restricted in one way or another. Recently, the admissibility problem in the negation-implication fragment of intuition-

istic propositional logic was proved to be $PSPACE$-complete [3].
Nevertheless, very little is known about the unification problem in some of the most important description and modal logics considered in Computer Science and Artificial Intelligence. For example, the decidability of the unification problem for the following description and modal logics remains open: description logic $\mathcal{ALC}$, modal logic K, multimodal variants of K, sub-Boolean modal logics. In the ordinary modal language, the modal logic Alt_1 is the least normal logic containing the formula $\Diamond x \rightarrow \Box x$. It is also the modal logic determined by the class of all frames (W, R) such that R is functional on W, i.e. for all $s, t, u \in W$, if sRt and sRu, then $t = u$. In this paper, we demonstrate that the unification problem in Alt_1 is in $PSPACE$.

2 Definitions

Syntax Let AF be a countable set of atomic formulas (denoted x, y, etc). The set F of all formulas (denoted ϕ, ψ, etc) is inductively defined as follows:

- $\phi ::= x \mid \bot \mid \neg\phi \mid (\phi \vee \psi) \mid \Box\phi$.

We define the other Boolean constructs as usual. The formula $\Diamond\phi$ is obtained as an abbreviation: $\Diamond\phi ::= \neg\Box\neg\phi$. We adopt the standard rules for omission of the parentheses. The degree of a formula ϕ, in symbols $deg(\phi)$, and its atom-set, in symbols $var(\phi)$, are inductively defined as follows:

- $deg(x) = 0$, $var(x) = \{x\}$,
- $deg(\bot) = 0$, $var(\bot) = \emptyset$,
- $deg(\neg\phi) = deg(\phi)$, $var(\neg\phi) = var(\phi)$,
- $deg(\phi \vee \psi) = \max\{deg(\phi), deg(\psi)\}$, $var(\phi \vee \psi) = var(\phi) \cup var(\psi)$,
- $deg(\Box\phi) = deg(\phi) + 1$, $var(\Box\phi) = deg(\phi)$.

We shall say that a formula ϕ is atom-free iff $var(\phi) = \emptyset$. Let AFF be the set of all atom-free formulas.

Semantics For all $n \in \mathbb{N}$, an n-valuation is an $(n+1)$-tuple $(U_0, \ldots, U_n)$ of subsets of AF. We inductively define the truth of a formula ϕ in an n-valuation $(U_0, \ldots, U_n)$, in symbols $(U_0, \ldots, U_n) \models \phi$, as follows:

- $(U_0, \ldots, U_n) \models x$ iff $x \in U_n$,
- $(U_0, \ldots, U_n) \not\models \bot$,
- $(U_0, \ldots, U_n) \models \neg\phi$ iff $(U_0, \ldots, U_n) \not\models \phi$,
- $(U_0, \ldots, U_n) \models \phi \vee \psi$ iff $(U_0, \ldots, U_n) \models \phi$, or $(U_0, \ldots, U_n) \models \psi$,
- $(U_0, \ldots, U_n) \models \Box\phi$ iff if $n \neq 0$, then $(U_0, \ldots, U_{n-1}) \models \phi$.

Obviously, $(U_0, \ldots, U_n) \models \Diamond\phi$ iff $n \neq 0$ and $(U_0, \ldots, U_{n-1}) \models \phi$. A formula ϕ is said to be n-valid, in symbols $\models_n \phi$, iff for all n-valuations $(U_0, \ldots, U_n)$, $(U_0, \ldots, U_n) \models \phi$. The modal logic Alt_1 is the least normal logic containing the formula $\Diamond x \rightarrow \Box x$. It is also the modal logic determined by the class of all frames (W, R) such that R is functional on W, i.e. for all $s, t, u \in W$, if sRt and sRu, then $t = u$. Obviously, Alt_1 is equal to the set of all formulas ϕ such that for all $n \in \mathbb{N}$, $\models_n \phi$.

Unification In the sequel, we use $\phi(x_1, \ldots, x_\alpha)$ to denote a formula whose atomic formulas form a subset of $\{x_1, \ldots, x_\alpha\}$. We shall say that a formula $\psi(x_1, \ldots, x_\alpha)$ is unifiable iff there exists $\phi_1, \ldots, \phi_\alpha \in F$ such that $\psi(\phi_1, \ldots, \phi_\alpha) \in Alt_1$. The unification problem is the decision problem defined as follows: given a formula $\psi(x_1, \ldots, x_\alpha)$, determine whether $\psi(x_1, \ldots, x_\alpha)$ is unifiable.

3 Lemmas

Let $\psi(x)$ be a formula. The reader may easily verify that

Lemma 1 *For all $k \in \mathbb{N}$, the following conditions are equivalent:* (1) *$\psi(x)$ is unifiable;* (2) *there exists $\phi \in AFF$ such that $\psi(\phi) \in Alt_1$;* (3) *there exists $\phi \in AFF$ such that $\Box^k\bot \rightarrow \psi(\phi) \in Alt_1$ and $\Diamond^k\top \rightarrow \psi(\phi) \in Alt_1$.*

Remark that Lemma 1 still holds when one considers a formula $\psi(x_1, \ldots, x_\alpha)$ with more than one atomic formula. In this case, simply replace the "there exists $\phi \ldots$" by "there exists $\phi_1, \ldots, \phi_\alpha \ldots$". Concerning the remainder of the paper, the same remark is on as well. Hence, without loss of generality, we will always consider that ψ is a formula with at most one atomic formula. In this case, for all $n \in \mathbb{N}$, an n-valuation is comparable to an $(n+1)$-tuple of bits. Let $k \in \mathbb{N}$ be such that $deg(\psi(x)) \leq k$. For all $\phi \in AFF$ and for all $n \in \mathbb{N}$, if $k \leq n$, then let $V_k(\phi, n, i) =$ "if $\models_{n-k+i} \phi$, then 1, else 0" for each $i \in \mathbb{N}$ such that $i \leq k$.

Lemma 2 *For all $\phi \in AFF$ and for all $n \in \mathbb{N}$, if $k \leq n$, then $\models_n \psi(\phi)$ iff $(V_k(\phi, n, 0), \ldots, V_k(\phi, n, k)) \models \psi(x)$.*

Proof: By induction on $\psi(x)$. $\dashv$

Lemma 3 *For all $\phi \in AFF$, $\Diamond^k\top \rightarrow \psi(\phi) \in Alt_1$ iff for all $n \in \mathbb{N}$, if $k \leq n$, then $(V_k(\phi, n, 0), \ldots, V_k(\phi, n, k)) \models \psi(x)$.*

Proof: Let $\phi \in AFF$. The following conditions are equivalent: (1) $\Diamond^k\top \rightarrow \psi(\phi) \in Alt_1$; (2) for all $n \in \mathbb{N}$, $\models_n \Diamond^k\top \rightarrow \psi(\phi)$; (3) for all $n \in \mathbb{N}$, if $\models_n \Diamond^k\top$, then $\models_n \psi(\phi)$; (4) for all $n \in \mathbb{N}$, if $k \leq n$, then $(V_k(\phi, n, 0), \ldots, V_k(\phi, n, k)) \models \psi(x)$. The reasons for these equivalences to hold are the following: the equivalence between (1) and (2) follows from the definition of Alt_1, the equivalence between (2) and (3) follows from the fact that $\phi \in AFF$ and the equivalence between (3) and (4) follows from Lemma 2. $\dashv$

For all $\phi \in AFF$ and for all $n \in \mathbb{N}$, if $k \leq n$, then let $\vec{V}_k(\phi, n) = (V_k(\phi, n, 0), \ldots, V_k(\phi, n, k))$. For all $\phi \in AFF$, let $f_k(\phi) = \{\vec{V}_k(\phi, n)$: $n \in \mathbb{N}$ is such that $k \leq n\}$. The atom-free formulas ϕ' and ϕ'' are said to be k-equivalent, in symbols $\phi' \equiv_k \phi''$, iff $f_k(\phi') = f_k(\phi'')$.

Lemma 4 *$\equiv_k$ is an equivalence relation on AFF possessing finitely many equivalence classes.*

Proof: By definitions of $\equiv_k$ and f_k, knowing that for all $\phi \in AFF$, $f_k(\phi)$ is a nonempty set of $(k+1)$-tuples of bits. $\dashv$

Lemma 5 *For all $\phi', \phi'' \in AFF$, if $\phi' \equiv_k \phi''$, then $\Diamond^k \top \rightarrow \psi(\phi') \in Alt_1$ iff $\Diamond^k \top \rightarrow \psi(\phi'') \in Alt_1$.*

Proof: By definitions of $\equiv_k$ and f_k and Lemma 3. $\dashv$

For all $\phi \in AFF$ and for all $n \in \mathbb{N}$, let $\vec{a}_k(\phi, n) = \vec{V}_k(\phi, n \cdot (k+1) + k)$. For all $\phi \in AFF$, let $g_k(\phi) = \{(\vec{a}_k(\phi, n), \vec{a}_k(\phi, n+1))$: $n \in \mathbb{N}\}$. We shall say that the atom-free formulas ϕ' and ϕ'' are k-congruent, in symbols $\phi' \cong_k \phi''$, iff $g_k(\phi') = g_k(\phi'')$.

Lemma 6 *$\cong_k$ is an equivalence relation on AFF possessing finitely many equivalence classes.*

Proof: By definitions of $\cong_k$ and g_k, knowing that for all $\phi \in AFF$, $g_k(\phi)$ is a nonempty set of pairs of $(k+1)$-tuples of bits. $\dashv$

Lemma 7 *For all $\phi', \phi'' \in AFF$, if $\phi' \cong_k \phi''$, then $\phi' \equiv_k \phi''$.*

Proof: Let $\phi', \phi'' \in AFF$. Suppose $\phi' \cong_k \phi''$ and $\phi' \not\equiv_k \phi''$. Hence, $g_k(\phi') = g_k(\phi'')$ and $f_k(\phi') \neq f_k(\phi'')$. Thus, there exists $n' \in \mathbb{N}$ such that $k \leq n'$ and $\vec{V}_k(\phi', n') \notin f_k(\phi'')$, or there exists $n'' \in \mathbb{N}$ such that $k \leq n''$ and $\vec{V}_k(\phi'', n'') \notin f_k(\phi')$. Without loss of generality, assume there exists $n' \in \mathbb{N}$ such that $k \leq n'$ and $\vec{V}_k(\phi', n') \notin f_k(\phi'')$. By the division algorithm, there exists $m, l \in \mathbb{N}$ such that $n' = m \cdot (k+1) + l$ and $l < k+1$.
Case $m = 0$**.** Since $k \leq n'$, $n' = m \cdot (k+1) + l$ and $l < k+1$, then $n' = k$. Hence, $\vec{V}_k(\phi', n') = \vec{a}_k(\phi', 0)$. Since $g_k(\phi') = g_k(\phi'')$, then there exists $n'' \in \mathbb{N}$ such that $(\vec{a}_k(\phi', 0), \vec{a}_k(\phi', 1)) = (\vec{a}_k(\phi'', n''), \vec{a}_k(\phi'', n''+1))$. Since $\vec{V}_k(\phi', n') = \vec{a}_k(\phi', 0)$, then $\vec{V}_k(\phi', n') = \vec{V}_k(\phi'', n'' \cdot (k+1) + k)$.
Case $m \neq 0$**.** Since $g_k(\phi') = g_k(\phi'')$, then there exists $n'' \in \mathbb{N}$ such that $(\vec{a}_k(\phi', m-1), \vec{a}_k(\phi', m)) = (\vec{a}_k(\phi'', n''), \vec{a}_k(\phi'', n''+1))$. Hence, $V_k(\phi', (m-1) \cdot (k+1) + k, i) = V_k(\phi'', n'' \cdot (k+1) + k, i)$ and $V_k(\phi', m \cdot (k+1) + k, i) = V_k(\phi'', (n''+1) \cdot (k+1) + k, i)$ for each $i \in \mathbb{N}$ such that $i \leq k$. Since $n' = m \cdot (k+1) + l$ and $i \leq k - (l+1)$ and $V_k(\phi', m \cdot (k+1) + l, i) = V_k(\phi', (m-1) \cdot (k+1) + k, i + (l+1))$, or $k - l \leq i$ and $V_k(\phi', m \cdot (k+1) + l, i) = V_k(\phi', m \cdot (k+1) + k, i - (k-l))$ for each $i \in \mathbb{N}$ such that $i \leq k$, then $i \leq k - (l+1)$ and $V_k(\phi', n', i) = V_k(\phi'', n'' \cdot (k+1) + k, i + (l+1))$, or $k - l \leq i$ and $V_k(\phi', n', i) = V_k(\phi'', (n''+1) \cdot (k+1) + k, i - (k-l))$ for each

$i \in \mathbb{N}$ such that $i \leq k$. Thus, $V_k(\phi', n', i) = V_k(\phi'', (n''+1)\cdot(k+1)+l, i)$ for each $i \in \mathbb{N}$ such that $i \leq k$. Therefore, $\vec{V}_k(\phi', n') = \vec{V}_k(\phi'', (n''+1)\cdot(k+1)+l)$.
In both cases, $\vec{V}_k(\phi', n') \in f_k(\phi'')$: a contradiction. $\dashv$

Lemma 8 *For all $\phi', \phi'' \in AFF$, if $\phi' \cong_k \phi''$, then $\Diamond^k\top \rightarrow \psi(\phi') \in Alt_1$ iff $\Diamond^k\top \rightarrow \psi(\phi'') \in Alt_1$.*

Proof: By Lemmas 5 and 7. $\dashv$

We shall say that a nonempty set B of pairs of $(k+1)$-tuples of bits is modally definable iff there exists $\phi \in AFF$ such that $B = g_k(\phi)$. For all nonempty sets B of pairs of $(k+1)$-tuples of bits, let $\triangleright_B$ be the domino relation on B. A path in the directed graph $(B, \triangleright_B)$ is said to be weakly Hamiltonian iff it visits each vertex at least once. Let $\vec{1}_{k+1}$ be the $(k+1)$-tuple of 1 and $\vec{0}_{k+1}$ be the $(k+1)$-tuple of 0.

Lemma 9 *For all nonempty sets B of pairs of $(k+1)$-tuples of bits, B is modally definable iff the directed graph $(B, \triangleright_B)$ contains a weakly Hamiltonian path ending with $(\vec{1}_{k+1}, \vec{1}_{k+1})$, or ending with $(\vec{0}_{k+1}, \vec{0}_{k+1})$.*

Proof: Let B be a nonempty set of pairs of $(k+1)$-tuples of bits.
If. Suppose the directed graph $(B, \triangleright_B)$ contains a weakly Hamiltonian path ending with $(\vec{1}_{k+1}, \vec{1}_{k+1})$, or ending with $(\vec{0}_{k+1}, \vec{0}_{k+1})$. Hence, there exists $s \in \mathbb{N}$ and there exists $(b'_0, b''_0), \ldots, (b'_s, b''_s) \in B$ such that $((b'_0, b''_0), \ldots, (b'_s, b''_s))$ is a weakly Hamiltonian path ending with $(\vec{1}_{k+1}, \vec{1}_{k+1})$, or ending with $(\vec{0}_{k+1}, \vec{0}_{k+1})$. Let $(\beta_0, \ldots, \beta_{s\cdot(k+1)+k})$ be the sequence of bits determined by the sequence $(b'_0, , \ldots, b'_s)$ of $(k+1)$-tuples of bits.
Case $(b'_s, b''_s) = (\vec{1}_{k+1}, \vec{1}_{k+1})$**.** Let $\phi = \bigvee\{\Diamond^i\Box\bot\colon i \in \mathbb{N} \text{ is such that } i < s\cdot(k+1) \text{ and } \beta_i = 1\} \vee \Diamond^{s\cdot(k+1)}\top$.
Case $(b'_s, b''_s) = (\vec{0}_{k+1}, \vec{0}_{k+1})$**.** Let $\phi = \bigvee\{\Diamond^i\Box\bot\colon i \in \mathbb{N} \text{ is such that } i < s\cdot(k+1) \text{ and } \beta_i = 1\}$.
In both cases, the reader may easily verify that for all $n \in \mathbb{N}$, if $n \leq s$, then $V_k(\phi, n\cdot(k+1)+k, i) = \beta_{n\cdot(k+1)+i}$ for each $i \in \mathbb{N}$ such that $i \leq k$. Hence, for all $n \in \mathbb{N}$, if $n \leq s$, then $\vec{V}_k(\phi, n\cdot(k+1)+k) = b'_n$. Thus, for all $n \in \mathbb{N}$, if $n \leq s$, then $(\vec{a}_k(\phi, n), \vec{a}_k(\phi, n+1)) = (b'_n, b''_n)$. Therefore, $B = g_k(\phi)$.
Only if. Suppose B is modally definable. Hence, there exists $\phi \in AFF$ such that $B = g_k(\phi)$. Obviously, there exists $n_0 \in \mathbb{N}$ such that for all $n \in \mathbb{N}$, if $n_0 \leq n$, then $\vec{a}_k(\phi, n) = \vec{1}_{k+1}$, or for all $n \in \mathbb{N}$, if $n_0 \leq n$, then $\vec{a}_k(\phi, n) = \vec{0}_{k+1}$. Thus, $((\vec{a}_k(\phi, 0), \vec{a}_k(\phi, 1)), \ldots, (\vec{a}_k(\phi, n_0), \vec{a}_k(\phi, n_0+1)))$ is a weakly Hamiltonian path ending with $(\vec{1}_{k+1}, \vec{1}_{k+1})$, or ending with $(\vec{0}_{k+1}, \vec{0}_{k+1})$. $\dashv$

4 Algorithm

We are now in a position to formulate the main result of this paper.

Proposition 1 *The unification problem in Alt_1 is in $PSPACE$.*

Proof: Using the above Lemmas, when k is such that $deg(\psi(x)) \leq k$, the given formula $\psi(x)$ is unifiable iff there exists a modally definable set B of pairs of $(k+1)$-tuples of bits from which, by means of its domino relation, an infinite sequence of bits respecting $\psi(x)$ and ending with 1s, or ending with 0s can be constructed. Hence, it suffices to consider the following procedure:

procedure $UNI(\psi(x))$
begin
$k := deg(\psi(x))$
guess a tuple $(b(0), \ldots, b(k))$ of bits of size $k+1$
$bool := \top$
$i := 0$
while $bool \wedge i \leq k$ do
    begin
    $bool := MC(b(0), \ldots, b(i), \psi(x))$
    $i := i + 1$
    end
if $\neg bool$, then reject
while $(b(0), \ldots, b(k)) \neq \vec{0}_{k+1} \wedge (b(0), \ldots, b(k)) \neq \vec{1}_{k+1}$ do
    begin
    guess a tuple $(b(k+1), \ldots, b(2k+1))$ of bits of size $k+1$
    $bool := \top$
    $i := 0$
    while $bool \wedge i \leq k$ do
        begin
        $bool := MC(b(i+1), \ldots, b(i+k+1), \psi(x))$
        $i := i + 1$
        end
    if $\neg bool$, then reject
    $(b(0), \ldots, b(k)) := (b(k+1), \ldots, b(2k+1))$
    end
accept
end

The function $MC(\cdot)$ takes as input a tuple $(b(i), \ldots, b(i+j))$ of bits and a formula $\psi(x)$ and returns the Boolean value $MC(b(i), \ldots, b(i+j), \psi(x)) =$ "if $(b(i), \ldots, b(i+j)) \models \psi(x), \text{then } \top, \text{else } \bot$". It can be implemented as a deterministic Turing machine working in polynomial time. The procedure $UNI(\cdot)$ takes as input a formula $\psi(x)$ and accepts it iff, when $k = deg(\psi(x))$, there exists a modally definable set B of pairs of $(k+1)$-tuples of bits from which, by means of its domino relation, an infinite sequence of bits respecting $\psi(x)$ and ending with 1s, or ending with 0s can be constructed. By Lemma 9, the procedure $UNI(\cdot)$ accepts its input $\psi(x)$ iff $\psi(x)$ is unifiable. It can be implemented as a nondeterministic Turing machine working in polynomial space. Hence, the unification problem is in $NPSPACE$. Since $NPSPACE = PSPACE$, the unification problem is in $PSPACE$. $\dashv$

Still, we do not know whether the unification problem in Alt_1 is $PSPACE$-hard.

5 Conclusion

Much remains to be done. For example, one may consider the unification problem when the ordinary modal language is extended by a set AP of parameters (denoted p, q, etc). In this case, the unification problem is to determine, given a formula $\psi(p_1, \ldots, p_\alpha, x_1, \ldots, x_\beta)$, whether there exists formulas $\phi_1, \ldots, \phi_\beta$ such that $\psi(p_1, \ldots, p_\alpha, \phi_1, \ldots, \phi_\beta) \in Alt_1$. For each $k \geq 2$, one may also consider the unification problem in Alt_k, the least normal logic containing the formula $\Diamond(x_1 \wedge \neg x_2 \wedge \ldots \wedge \neg x_{k-1} \wedge \neg x_k) \wedge \ldots \wedge \Diamond(\neg x_1 \wedge \neg x_2 \wedge \ldots \wedge \neg x_{k-1} \wedge x_k) \rightarrow \Box(x_1 \vee x_2 \vee \ldots \vee x_{k-1} \vee x_k)$. In other respects, one may consider the unification problem when the ordinary modal language is replaced by its multimodal variant. Finally, what becomes of these problems when the ordinary modal language is extended by the universal modality?

Acknowledgements

Special acknowledgement is heartly granted to Çiğdem Gencer, Silvio Ghilardi, Rosalie Iemhoff and Emil Jeřábek. Philippe Balbiani was partially supported by the Bulgarian National Science Fund (contract DID02/32/209) and Tinko Tinchev was partially supported by the Centre international de mathématiques et d'informatique (contract ANR-11-LABX-0040-CIMI within the program ANR-11-IDEX-0002-02).

References

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

[2] Baader, F., Morawska, B.: *Unification in the description logic $\mathcal{EL}$.* In Treinen, R. (editor): *Rewriting Techniques and Applications.* Springer (2009) 350–364.

[3] Cintula, P., Metcalfe, G.: *Admissible rules in the implication-negation fragment of intuitionistic logic.* Annals of Pure and Applied Logic **162** (2010) 162–171.

[4] Ghilardi, S.: *Unification in intuitionistic logic.* Journal of Symbolic Logic **64** (1999) 859–880.

[5] Iemhoff, R.: *On the admissible rules of intuitionistic propositional logic.* The Journal of Symbolic Logic **66** (2001) 281–294.

[6] Jeřábek, E.: *Complexity of admissible rules.* Archive for Mathematical Logic **46** (2007) 73–92.

[7] Rybakov, V.: *Admissibility of Logical Inference Rules.* Elsevier (1997).

[8] Wolter, F., Zakharyaschev, M.: *Undecidability of the unification and admissibility problems for modal and description logics.* ACM Transactions on Computational Logic **9** (2008) 25:1–25:20.

On Asymmetric Unification and the Combination Problem in Disjoint Theories

(Extended Abstract)

Serdar Erbatur[1], Deepak Kapur[2] [*], Andrew M. Marshall[3][†], Catherine Meadows[3], Paliath Narendran[4] [‡]and Christophe Ringeissen[5]

[1] Università degli Studi di Verona, Italy
[2] University of New Mexico, Albuquerque, NM, USA
[3] Naval Research Laboratory, Washington, DC, USA
[4] University at Albany–SUNY, Albany, NY, USA
[5] LORIA – INRIA Nancy-Grand Est, Nancy, France

A full version of this work [5] appeared in the proceedings of the 17th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2014). In the following, we provide a summary of this work.

1 Introduction

We examine the disjoint combination problem in the newly developed paradigm of asymmetric unification. This new unification problem was developed based on newly identified requirements arising from symbolic cryptographic protocol analysis [4]. Its application involves unification-based exploration of a space in which the states obey rich equational theories that can be expressed as a decomposition $R \uplus E$, where R is a set of rewrite rules that are confluent, terminating and coherent modulo E. However, in order to apply state space reduction techniques, it is usually necessary for at least part of this state to be in normal form, and to remain in normal form even after unification is performed. This requirement can be expressed as an *asymmetric* unification problem $\{s_1 =^{\downarrow} t_1, \ \ldots, \ s_n =^{\downarrow} t_n\}$ where the $=^{\downarrow}$ denotes a unification problem with the restriction that any unifier leaves the right-hand side of each equation irreducible.

Although asymmetric unification has the potential of playing an important role in cryptographic protocol analysis, and possibly other unification-based state explorations as well, it is still not that well understood. Until the development of special-purpose algorithms for exclusive-or and free Abelian group theories, the only known asymmetric unification algorithm was variant narrowing. One important question is the problem of asymmetric unification in a combination of theories, in particular how to produce an algorithm for the combined theory by combining algorithms for the separate theories. This is particularly significant for cryptographic protocol analysis. Cryptographic protocols generally make use of more than one cryptoalgorithm. Often, these cryptoalgorithms can be described in terms of disjoint equational theories. In the case in which the algorithm used is variant narrowing, the problem is straightforward. If the combination of two theories with the finite variant property also has the finite variant property, then one applies variant narrowing. However, in attempting to combine theories with special-purpose algorithms, the path is less clear. This is an important point with respect to

[*]Partially supported by the NSF grant CNS-0905222
[†]ASEE postdoctoral fellowship under contract to the NRL.
[‡]Partially supported by the NSF grant CNS-0905286

efficiency since special-purpose asymmetric algorithms have the promise of being more efficient than variant narrowing.

In this work we take the first step to solving this problem, by showing that the combination method for the unification problem in disjoint equational theories developed by Baader and Schulz in [2] can be modified and extended to the asymmetric unification paradigm. The only restrictions on this new method are those inherited from the asymmetric unification problem and those inherited from Baader and Schulz.

2 Asymmetric Unification

We use the standard notation of equational unification [3] and term rewriting systems [1].

Definition 2.1. Let Γ be an E-unification problem, let $\mathcal{X}$ denote the set of variables occurring in Γ and $\mathcal{C}$ the set of free constants occurring in Γ. For a given linear ordering $<$ on $\mathcal{X} \cup \mathcal{C}$, and for each $c \in \mathcal{C}$ define the set V_c as $\{x \mid x \text{ is a variable with } x < c\}$. An *$E$-unification problem with linear constant restriction* (LCR) is an E-unification problem with constants, Γ, where each constant c in Γ is equipped with a set V_c of variables. A solution of the problem is an E-unifier σ of Γ such that for all c, x with $x \in V_c$, the constant c does not occur in $x\sigma$. We call σ an *E-unifier with linear constant restriction.*

Definition 2.2. We call $(\Sigma,\ E,\ R)$ a *decomposition* of an equational theory Δ over a signature Σ if $\Delta = R \uplus E$ and R and E satisfy the following conditions: (1) E is variable preserving, i.e., for each $s = t$ in E we have $Var(s) = Var(t)$. (2) E has a finitary and complete unification algorithm. That is, an algorithm that produces a finite complete set of unifiers. (3) For each $l \to r \in R$ we have $Var(r) \subseteq Var(l)$. (4) R is confluent and terminating modulo E, i.e., the relation $\to_{R/E}$ is confluent and terminating. (5) $\to_{R,E}$ is E-coherent, i.e., $\forall t_1, t_2, t_3$ if $t_1 \to_{R,E} t_2$ and $t_1 =_E t_3$ then $\exists\ t_4, t_5$ such that $t_2 \to^*_{R,E} t_4$, $t_3 \to^+_{R,E} t_5$, and $t_4 =_E t_5$.

This definition is inherited directly from [4]. The last restrictions ensure that $s \to^!_{R/E} t$ iff $s \to^!_{R,E} t$, therefore it is sufficient to consider R, E rather then R/E (see [4]).

Definition 2.3 (Asymmetric Unification)**.** Given a decomposition (Σ, E, R) of an equational theory, a substitution σ is an *asymmetric R, E-unifier* of a set $\mathcal{S}$ of asymmetric equations $\{s_1 =^{\downarrow} t_1,\ \ldots,\ s_n =^{\downarrow} t_n\}$ iff for each asymmetric equations $s_i =^{\downarrow} t_i$, σ is an $(E \cup R)$-unifier of the equation $s_i =^{?} t_i$ and $(t_i \downarrow_{R,E})\sigma$ is in R, E-normal form. A set of substitutions Ω is a *complete set of asymmetric R, E-unifiers* of $\mathcal{S}$ (denoted $CSAU_{R \cup E}(\mathcal{S})$ or just $CSAU(\mathcal{S})$ if the background theory is clear) iff: (i) every member of Ω is an asymmetric R, E-unifier of $\mathcal{S}$, and (ii) for every asymmetric R, E-unifier θ of $\mathcal{S}$ there exists a $\sigma \in \Omega$ such that $\sigma \leq_E^{Var(\mathcal{S})} \theta$.

Example 2.4. Let $R = \{x \oplus 0 \to x,\ x \oplus x \to 0,\ x \oplus x \oplus y \to y\}$ and E be the AC theory for $\oplus$. Consider the equation $y \oplus x =^{\downarrow} x \oplus a$, the substitution $\sigma_1 = \{y \mapsto a\}$ is an asymmetric solution but, $\sigma_2 = \{x \mapsto 0,\ y \mapsto a\}$ is not.

Definition 2.5 (Asymmetric Unification with Linear Constant Restriction)**.** Let $\mathcal{S}$ be a set of of asymmetric equations with some LCR. A substitution σ is an *asymmetric R, E-unifier* of $\mathcal{S}$ with LCR iff σ is an asymmetric solution to $\mathcal{S}$ and σ satisfies the LCR.

3 Combining Asymmetric Unification Algorithms

Let Δ_1 and Δ_2 denote two equational theories with disjoint signatures Σ_1 and Σ_2. Let Δ be the combination, $\Delta = \Delta_1 \cup \Delta_2$, of the two theories having signature $\Sigma_1 \cup \Sigma_2$. We assume Δ_i

admits a a decomposition (Σ_i, E_i, R_i), and an asymmetric Δ_i-unification with linear constant restriction algorithm is known for $i = 1, 2$. In [5], we show that the Baader-Schulz combination method [2] designed for unification can be reused for asymmetric unification. A slight adaptation is required to construct combined unifiers that are necessarily asymmetric.

Theorem 3.1. *([5]) Asymmetric $\Delta_1 \cup \Delta_2$-unification is decidable (resp. finitary) if asymmetric Δ_i-unification with LCR is decidable (resp. finitary), for $i = 1, 2$.*

As in [2], it can be shown that there exists an asymmetric Δ_i-unification algorithm with LCR if and only if there exists an asymmetric Δ_i-unification algorithm with free symbols. Therefore, the above theorem can be rephrased in terms of asymmetric unification with free symbols.

Example 3.2. Let $\Delta_1 = R_1 \cup E_1$, where $R_1 = \{e(x, d(x, y)) \to y,\ d(x, e(x, y)) \to y\}$ and $E_1 = \emptyset$. Let $\Delta_2 = R_2 \cup E_2$, where $R_2 = \{x \oplus 0 \to x,\ x \oplus x \to 0,\ x \oplus x \oplus y \to y\}$ and $E_2 = \{x \oplus y = y \oplus x,\ (x \oplus y) \oplus z = x \oplus (y \oplus z)\}\}$. Consider the set of equations $\{x_0 \oplus x_1 \oplus x_2 =^{\downarrow} x_3 \oplus x_4,\ e(x_1, d(0, x_5)) =^{\downarrow} x_2 \oplus x_0,\ e(x_1, d(x_0, e(x_2, x_6))) =^{\downarrow} e(x_7, x_5)\}$. After purification, we get Γ_2: $\{x_0 \oplus x_1 \oplus x_2 =^{\downarrow} x_3 \oplus x_4,\ e(x_1, d(z_0, x_5)) =^{\downarrow} z_1,\ 0 =^{\downarrow} z_0,\ z_1 =^{\downarrow} x_2 \oplus x_0,\ e(x_1, d(x_0, e(x_2, x_6))) =^{\downarrow} e(x_7, x_5)\}$. The next step considers the set of variable partitions, one of which is the following partition $\{\{x_0, x_3\}, \{x_2, x_4\}, \{x_5, z_1\}, \{x_1, z_0, x_7\}, \{x_6\}\}$ Choosing a representative for each set, we would produce the following Γ_3: $\{x_0 \oplus x_1 \oplus x_2 =^{\downarrow} x_0 \oplus x_2,\ e(x_1, d(x_1, x_5)) =^{\downarrow} x_5,\ 0 =^{\downarrow} x_1,\ x_5 =^{\downarrow} x_2 \oplus x_0,\ e(x_1, d(x_0, e(x_2, x_6))) =^{\downarrow} e(x_1, x_5)\}$. The next step considers the possible pairs of variable orderings and theory indexes. One pair that would be produced is the following: $x_6 > x_5 > x_2 > x_1 > x_0$, index-1 $= \{x_0, x_1, x_2, x_5\}$ and index-2 $= \{x_6\}$. Next Γ_4 is produced from that pair and split into pure sets to produce $\Gamma_{5,1}$ and $\Gamma_{5,2}$. Let us denote a variable, y, being treated as a constant as $\mathbf{y}$. Then, $\Gamma_{5,1}$ is the following set of equations: $\{x_0 \oplus x_1 \oplus x_2 =^{\downarrow} x_0 \oplus x_2,\ 0 =^{\downarrow} x_1,\ x_5 =^{\downarrow} x_2 \oplus x_0\}$ and $\Gamma_{5,2}$ is the following set of equations: $\{e(\mathbf{x}_1, d(\mathbf{x}_1, \mathbf{x}_5)) =^{\downarrow} \mathbf{x}_5,\ e(\mathbf{x}_1, d(\mathbf{x}_0, e(\mathbf{x}_2, x_6))) =^{\downarrow} e(\mathbf{x}_1, \mathbf{x}_5)\}$. Next $\Gamma_{5,i}$ is solved with LCR. The last step is to combine each pair of substitutions (σ_1, σ_2) into a substitution σ. One such pair is $\sigma_1 = \{x_1 \mapsto 0,\ x_5 \mapsto x_2 \oplus x_0\}$ and $\sigma_2 = \{x_6 \mapsto d(\mathbf{x}_2, e(\mathbf{x}_0, \mathbf{x}_5))\}$. Thus, we get the following asymmetric solution, $\{x_1 \mapsto 0,\ x_3 \mapsto x_0,\ x_4 \mapsto x_2,\ x_5 \mapsto x_2 \oplus x_0,\ x_6 \mapsto d(x_2, e(x_0, x_2 \oplus x_0)),\ x_7 \mapsto 0\}$, (existential variables z_0, z_1 are removed).

References

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

[2] Franz Baader and Klaus U. Schulz. Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures. *Journal of Symbolic Computation*, 21(2):211 – 243, 1996.

[3] 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.

[4] Serdar Erbatur, Santiago Escobar, Deepak Kapur, Zhiqiang Liu, Christopher A. Lynch, Catherine Meadows, José Meseguer, Paliath Narendran, Sonia Santiago, and Ralf Sasse. Asymmetric Unification: A New Unification Paradigm for Cryptographic Protocol Analysis. In Maria Paola Bonacina, editor, *Automated Deduction, CADE-24*, volume 7898 of *Lecture Notes in Computer Science*, pages 231–248. Springer Berlin Heidelberg, 2013.

[5] Serdar Erbatur, Deepak Kapur, Andrew M. Marshall, Catherine Meadows, Paliath Narendran, and Christophe Ringeissen. On asymmetric unification and the combination problem in disjoint theories. In Anca Muscholl, editor, *Foundations of Software Science and Computation Structures*, volume 8412 of *Lecture Notes in Computer Science*, pages 274–288. Springer Berlin Heidelberg, 2014.

Hierarchical Combination of Matching Algorithms

(Extended Abstract)

Serdar Erbatur[1], Deepak Kapur[2] *, Andrew M Marshall[3] †, Paliath Narendran[4] ‡, and Christophe Ringeissen[5]

[1] Università degli Studi di Verona (Italy)
[2] University of New Mexico (USA)
[3] Naval Research Laboratory (USA)
[4] University at Albany, SUNY (USA)
[5] LORIA – INRIA Nancy-Grand Est (France)

1 Introduction

A critical question in matching and unification is how to obtain an algorithm for the combination of non-disjoint equational theories when there exist algorithms for the constituent theories. In recent work ([4]) we were able to develop a new approach to the unification problem in the combination of non-disjoint theories. The approach is based on a new set of restrictions, for which we can identify a set of properties on the constituent theories, such that theories characterized by these properties satisfy the restrictions and thus can be combined using the new algorithm. The main properties of this class are: a hierarchical organization of E_1 and E_2, R_1 is a left-linear, convergent rewrite system corresponding to E_1, and the shared symbols are "inner constructors" of R_1.

Here we consider the matching problem in this new hierarchical framework. Due to the more restricted nature of the matching problem we obtain several improvements over the unification problem. One of the improvements is that we are able to relax several restrictions we assumed for the unification problem. Key among these discarded restrictions is a restriction on the type of new variables created by the unification algorithm for the first theory in the hierarchical organization. In the unification setting it was necessary to restrict variables which could cause reapplication of the first unification algorithm, denoted as "ping pong" variables. This tricky restriction can be avoided if most general solutions can be expressed without any new variable. Because matching problems in regular (variable-preserving) theories have only ground solutions, we can remove this assumption.

An additional improvement is obtained when constructing a general matching algorithm for the first theory which satisfies the restrictions of the hierarchical framework. In the unification case a general procedure was developed but due to the generality of unification problem, termination had to be checked for each theory. However, for the matching problem we are able to exploit an interesting relation to the work done on syntactic theories [5, 6, 3]. By assuming a newly defined *resolvent* property we are able to construct a terminating and thus general matching algorithm which can be used in the hierarchical framework for any theory satisfying the restrictions. The algorithm we present can be seen as an extension of the work done for matching in disjoint unions of regular/syntactic theories [7, 8, 9].

*Partially supported by the NSF grant CNS-0905222
†ASEE postdoctoral fellowship under contract to the NRL.
‡Partially supported by the NSF grant CNS-0905286

2 Preliminaries

We use the standard notation of equational unification [2] and term rewriting systems [1]. A term t is *linear* if each variable of t occurs only once in t. Given a first-order signature Σ, and a set E of Σ-axioms (i.e., pairs of Σ-terms, denoted by $l = r$), the *equational theory* $=_E$ is the congruence closure of E under the law of substitutivity. By a slight abuse of terminology, E will be often called an equational theory. An axiom $l = r$ is *variable-preserving* if $Var(l) = Var(r)$. An axiom $l = r$ is *linear* (resp. *collapse-free*) if l and r are linear (resp. non-variable terms). An equational theory is *variable-preserving* (resp. linear/collapse-free) if all its axioms are variable-preserving (resp. linear/collapse-free). An equational theory E is *finite* if for each term t, there are finitely many terms s such that $t =_E s$. A theory E is *subterm collapse-free* if and only if for all terms t it is not the case that $t =_E u$ where u is a strict subterm of t. Note that a subterm collapse-free theory is necessarily variable-preserving and collapse-free.

A Σ-equation is a pair of Σ-terms denoted by $s =^? t$. When t is ground, $s =^? t$ is denoted by $s \leq^? t$ and called a match-equation. A unification (resp. matching) problem P is a set of equations (resp. match-equations). An E-unifier of P is a substitution σ such that $s\sigma =_E t\sigma$ for each equation $s =^? t$ in P.

For a convergent rewrite system R we define a constructor of R to be a function symbol f which does not appear at the root on the left-hand side of any rewrite rule of R. We define an inner constructor to be a constructor f that satisfies the following additional restrictions: (1) f does not appear on the left-hand side on any rule in R. (2) f does not appear as the root symbol on the right-hand side of any rule in R. (3) there are no function symbols below f on the right-hand side of any rule in R. We consider two equational theories E_1 and E_2 built over the signatures Σ_1 and Σ_2. Let $\Sigma_{(1,2)} = \Sigma_1 \cap \Sigma_2$. In [4], we introduce a hierarchical framework for a union of equational theories $E_1 \cup E_2$ such that E_1 is given by a convergent rewrite system R_1 for which $\Sigma_{(1,2)}$-symbols are inner constructors. In [4], we study the unification problem in $E_1 \cup E_2$. In this work, we now consider the matching problem.

3 Hierarchical Combination for Matching

The key principle of the combination algorithm for matching is to purify only the left-hand sides of matching problems. Thus, this purification introduces a pending solved equation $X =^? t$. Since X occurs in a match-equation solved by A_1 or A_2, it will be instantiated by a ground term, say u, transforming eventually $X =^? t$ into a match-equation $t \leq^? u$. Hence, our rule-based procedures will generate equational problems involving also equations and not only match-equations. Fortunately, we assume the right properties to solve these equational problems by using only matching algorithms:

1. Properties of E_1: E_1 is finite, subterm collapse-free and R_1 is a left-linear, convergent term rewrite system corresponding to E_1.
2. Properties of E_2: E_2 is a linear, finite, collapse-free equational theory.
3. Properties of the shared symbols: If $f \in \Sigma_{(1,2)}$, then f is an *inner constructor* of R_1. If f and g are inner constructors of R_1, then f-rooted terms cannot be equated to g-rooted terms in E_2.

According to the above assumptions, we can show that $E_1 \cup E_2$ is finite, and so we could take a brute force approach to constructing a $E_1 \cup E_2$-matching algorithm [7]. However, we can use the constituent algorithms, A_1 and A_2 to construct a more efficient combination method.

We assume that A_1 and A_2 handle now left pure match-equations: A_1 handles match-equations whose left-hand sides are in $(\Sigma_1 \backslash \Sigma_{1,2})$, whilst A_2 handles match-equations whose left-hand sides are in Σ_2.

We first consider the question of constructing the A_1 algorithm. We show how such algorithms can be constructed for a family of theories related to the syntactic theories [5, 6, 3]. Therefore, we assume the following *resolvent* property for R_1.

Restriction 1. *(Algorithm A_1)*
Algorithm A_1 is a mutation-based algorithm as depicted in Figure 1, where R_1 is a resolvent *rewrite system; that is, any R_1-normal form can be reached by applying at most one rewrite step at the top position.*

Note, resolvent does not require that all paths from a term to its normal form use one topmost rewrite step, only that for each normal form there is at least one rewrite path with such a property. When R_1 is resolvent, the mutation-based A_1 algorithm presented in Figure 1 is sound and complete. For the second algorithm A_2, we simply assume it is a matching algorithm.

Restriction 2. *(Algorithm A_2)*
Algorithm A_2 is an E_2-matching algorithm.

Rule (i) **Mutate**

$$\{f(s_1, \ldots, s_m) \leq^? g(t_1, \ldots, t_n)\} \cup P$$
$$\longrightarrow \{s_1 =^? l_1, \ldots, s_m =^? l_m, r_1 \leq^? t_1, \ldots, r_n \leq^? t_n\} \cup P$$

If $f(l_1, \ldots, l_m) \to g(r_1, \ldots, r_n)$ is a fresh variant of a rule in R_1.

Rule (ii) **Matching Decomposition**

$$\{f(s_1, \ldots, s_m) \leq^? f(t_1, \ldots, t_m)\} \cup P$$
$$\longrightarrow \{s_1 \leq^? t_1, \ldots, s_m \leq^? t_m\} \cup P$$

Where $f \in \Sigma_1 \setminus \Sigma_{(1,2)}$.

Rule (iii) **Matching Clash**

$$\{f(s_1, \ldots, s_m) \leq^? g(t_1, \ldots, t_n)\} \cup P$$
$$\longrightarrow Fail$$

Where $f \in \Sigma_1 \setminus \Sigma_{(1,2)}, f \neq g$ and **Mutate** does not apply.

Figure 1: Mutation-based A_1 algorithm

3.1 The Matching Procedure - Hierarchical Combination

We give a new matching procedure for the hierarchical combination. It works as follows. First, we purify the left-hand sides of match-equations. After this purification step, we can easily distinguish which left-pure match-equations must be given to A_1 and A_2. Then, the solutions computed by A_1 and A_2 are combined using some replacement and merging rules.

$Solve_1$**: Run** A_1
We apply A_1 to match-equations having $\Sigma_1 \setminus \Sigma_{(1,2)}$-pure left-hand sides
$Solve_2$**: Run** A_2
We apply A_2 to match-equations having Σ_2-pure left-hand sides

RemEq:
$$\frac{\mathcal{P} \uplus \{t =^? t'\}}{\mathcal{P} \cup \{t \leq^? t'\}} \quad \text{if } t' \text{ is ground}$$

Rep:
$$\frac{\mathcal{P} \uplus \{t =^? t'[Y],\ Y \leq^? u\}}{\mathcal{P} \cup \{t =^? t'[u],\ Y \leq^? u\}}$$

Merge:
$$\frac{\mathcal{P} \uplus \{X \leq^? t,\ X \leq^? s\}}{\mathcal{P} \cup \{X \leq^? t\}} \quad \text{if } s =_{E_1 \cup E_2} t$$

Clash:
$$\frac{\mathcal{P} \uplus \{X \leq^? t,\ X \leq^? s\}}{\mathit{Fail}} \quad \text{if } s \neq_{E_1 \cup E_2} t$$

Figure 2: $\mathfrak{D}$: inference system for the combination of matching

The matching procedure is given as the inference system $\mathfrak{D}$ defined in Figure 2 by the set of inferences rules

$$\{Solve_1, Solve_2, \mathbf{RemEq}, \mathbf{Rep}, \mathbf{Merge}, \mathbf{Clash}\}.$$

We can easily verify that each rule in $\mathfrak{D}$ preserves the set of $E_1 \cup E_2$-solutions. This is clear for the rules in $\{\mathbf{RemEq}, \mathbf{Rep}, \mathbf{Merge}, \mathbf{Clash}\}$. Moreover, this is true by definition for $Solve_1$, and since E_2-matching is sound and complete for solving $E_1 \cup E_2$-matching problems whose left-hand sides are 2-pure, its is also true for $Solve_2$. Furthermore, it can be shown that normal forms with respect to $\mathfrak{D}$ are matching problems in solved form and that $\mathfrak{D}$ terminates for any input. This implies that the algorithm $\mathfrak{D}$ is sound and complete, which means that it provides an $E_1 \cup E_2$-matching algorithm.

Example 3.1. The following theory appears to be a good case-study for the above hierarchical combination method.

$$\mathcal{E}_{AC} = \left\{ \begin{array}{l} exp(exp(x,y),z) = exp(x, y \circledast z) \\ exp(x * y, z) = exp(x,z) * exp(y,z) \end{array} \right\} = E_1$$
$$\left\{ \begin{array}{l} (x \circledast y) \circledast z = x \circledast (y \circledast z) \\ x \circledast y = y \circledast x \end{array} \right\} = E_2$$

The theory $\mathcal{E}_{AC}$ has the following $AC(\circledast)$-convergent system:

$$R_1 = \left\{ \begin{array}{rcl} exp(exp(x,y),z) & \to & exp(x, y \circledast z) \\ exp(x * y, z) & \to & exp(x,z) * exp(y,z) \end{array} \right.$$

The main task is to construct an A_1 algorithm. It can be constructed from an instantiation of the mutation-based algorithm given in Figure 1. This leads to a set of matching inference rules dedicated to the particular case of R_1.

References

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

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

[3] Alexandre Boudet and Evelyne Contejean. On n-syntactic equational theories. In Hélène Kirchner and Giorgio Levi, editors, *Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 446–457. Springer Berlin Heidelberg, 1992.

[4] Serdar Erbatur, Deepak Kapur, Andrew M. Marshall, Paliath Narendran, and Christophe Ringeissen. Hierarchical combination. In Maria Paola Bonacina, editor, *Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 249–266. Springer Berlin Heidelberg, 2013.

[5] Jean-Pierre Jouannaud. Syntactic theories. In Branislav Rovan, editor, *Mathematical Foundations of Computer Science 1990*, volume 452 of *Lecture Notes in Computer Science*, pages 15–25. Springer Berlin Heidelberg, 1990.

[6] 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.

[7] T. Nipkow. Proof transformations for equational theories. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 278–288, Jun 1990.

[8] Tobias Nipkow. Combining matching algorithms: The regular case. *J. Symb. Comput.*, 12(6):633–654, 1991.

[9] Christophe Ringeissen. Combining decision algorithms for matching in the union of disjoint equational theories. *Inf. Comput.*, 126(2):144–160, 1996.

From Admissibility to a New Hierarchy of Unification Types

Leonardo Cabrer and George Metcalfe

[1] University of Florence, Italy
l.cabrer@disia.unifi.it
[2] University of Bern, Switzerland
george.metcalfe@math.unibe.ch

Abstract

Motivated by the study of admissible rules, a new hierarchy of "exact" unification types is introduced where a unifier is more general than another unifier if all identities unified by the first are unified by the second. A Ghilardi-style algebraic interpretation of this hierarchy is presented that features exact algebras rather than projective algebras. Examples of equational classes distinguishing the two hierarchies are also provided.

1 Introduction

It has long been recognized that the study of admissible rules is inextricably bound up with the study of equational unification (see, e.g., [23, 10, 11]). Indeed, from an algebraic perspective, admissibility in an equational class (variety) of algebras may be viewed as a generalization of unifiability in that class.[1] Let us fix an equational class of algebras $\mathcal{V}$ for a language $\mathcal{L}$ and denote by $\mathbf{Fm}_{\mathcal{L}}(X)$, the formula algebra (absolutely free algebra or term algebra) of $\mathcal{L}$ over a set of variables $X \subseteq \omega$. A substitution (homomorphism) $\sigma\colon \mathbf{Fm}_{\mathcal{L}}(X) \to \mathbf{Fm}_{\mathcal{L}}(\omega)$ is called a *$\mathcal{V}$-unifier (over X)* of a set of $\mathcal{L}$-identities Σ with variables in X if $\mathcal{V} \models \sigma(\varphi) \approx \sigma(\psi)$ for all $\varphi \approx \psi$ in Σ. A clause $\Sigma \Rightarrow \Delta$ (an ordered pair of finite sets of $\mathcal{L}$-identities Σ, Δ) is *$\mathcal{V}$-admissible* if every $\mathcal{V}$-unifier of Σ is a $\mathcal{V}$-unifier of a member of Δ. In particular, Σ is $\mathcal{V}$-unifiable if and only if $\Sigma \Rightarrow \emptyset$ is not $\mathcal{V}$-admissible.

In certain cases, $\mathcal{V}$-admissibility may also be reduced to $\mathcal{V}$-unifiability. Suppose that the unification type of $\mathcal{V}$ is at most finitary, meaning that every $\mathcal{V}$-unifier of a set of $\mathcal{L}$-identities Σ over the variables in Σ is a substitution instance of one of a finite set S of $\mathcal{L}$-unifiers of Σ. Then a clause $\Sigma \Rightarrow \Delta$ is $\mathcal{V}$-admissible if each member of S is an $\mathcal{L}$-unifier of a member of Δ. If there is an algorithm for determining the finite basis set S for Σ and the equational theory of $\mathcal{V}$ is decidable, then checking $\mathcal{V}$-admissibility is also decidable. This observation, together with the pioneering work of Ghilardi on equational unification for classes of Heyting and modal algebras [10, 11], has led to a wealth of decidability, complexity, and axiomatization results for admissibility in these classes and corresponding modal and intermediate logics [12, 13, 15, 7, 3, 2, 21, 18].

The success of this approach to admissibility appears to rely on considering varieties with at most finitary unification type. That this is not the case, however, is illustrated by the case of MV-algebras, the algebraic semantics of Łukasiewicz infinite-valued logic. Decidability, complexity, and axiomatization results for admissibility in this class have been established by Jeřábek [16, 17, 18] via a similar reduction of finite sets of identities to finite approximating sets of identities. On the other hand, it has been shown by Marra and Spada [20] that the variety of MV-algebras has nullary unification type, which means in particular that there are finite sets of identities for which no finite basis of unifiers exists. Further examples of this discrepancy

[1] We refer the reader to [4] and [19] for undefined notions of universal algebra and category theory, respectively.

may be found in [6], including the very simple example of the class of distributive lattices where admissibility and validity of clauses coincide but unification is nullary.

As mentioned above, it is possible to check the $\mathcal{V}$-admissibility of a clause $\Sigma \Rightarrow \Delta$ by checking that every $\mathcal{V}$-unifier of Σ in a certain "basis set" $\mathcal{V}$-unifies Δ. Such a basis set S typically has the property that every other $\mathcal{V}$-unifier of Σ is a substitution instance of a member of S. The starting point for this paper is the observation that a weaker condition on S suffices, leading potentially to smaller sets. What is really required for checking admissibility is the property that every $\mathcal{V}$-unifier of Σ $\mathcal{V}$-unifies all identities $\mathcal{V}$-unified by some member of S. Then $\Sigma \Rightarrow \Delta$ is $\mathcal{V}$-admissible if each member of S is a $\mathcal{V}$-unifier of a member of Δ. This leads to a new ordering of $\mathcal{V}$-unifiers and hierarchy of exact (unification) types. Moreover, we obtain a Ghilardi-style algebraic characterization making use of exact algebras rather than projective algebras. Crucially, we also show that an equational class can have an exact type that is "better than" its unification type. For example, MV-algebras have finitary exact type and distributive lattices have unitary exact type.

2 Equational Unification and Projective Algebras

Let us first recall some basic notions for equational unification, referring to [1] for further details. We then provide a short overview of the algebraic approach to equational unification developed by Ghilardi in [9].

Let $\mathbf{P} = \langle P, \leq \rangle$ be a preordered set. A *complete* set for $\mathbf{P}$ is a subset $M \subseteq P$ such that for every $x \in P$, there exists $y \in M$ such that $x \leq y$. A complete set M for $\mathbf{P}$ is called a *μ-set* for $\mathbf{P}$ if $x \not\leq y$ and $y \not\leq x$ for all distinct $x, y \in M$. It is easily seen that if $\mathbf{P}$ has a μ-set, then every μ-set of $\mathbf{P}$ has the same cardinality. $\mathbf{P}$ is said to be *nullary* if it has no μ-sets ($\mathrm{type}(\mathbf{P}) = 0$), *infinitary* if it has a μ-set of infinite cardinality ($\mathrm{type}(\mathbf{P}) = \infty$), *finitary* if it has a finite μ-set of cardinality greater than 1 ($\mathrm{type}(\mathbf{P}) = \omega$), and *unitary* if it has a μ-set of cardinality 1 ($\mathrm{type}(\mathbf{P}) = 1$). These types are ordered as follows: $1 < \omega < \infty < 0$.

Now let $\mathcal{L}$ be a language and $X \subseteq \omega$ a set of variables, and consider substitutions $\sigma_i \colon \mathbf{Fm}_{\mathcal{L}}(X) \to \mathbf{Fm}_{\mathcal{L}}(\omega)$ for $i = 1, 2$. We say that σ_1 is *more general* than σ_2 (written $\sigma_2 \preccurlyeq \sigma_1$) if there exists a substitution $\sigma' \colon \mathbf{Fm}_{\mathcal{L}}(\omega) \to \mathbf{Fm}_{\mathcal{L}}(\omega)$ such that $\sigma' \circ \sigma_1 = \sigma_2$. Let $\mathcal{V}$ be an equational class of algebras for $\mathcal{L}$ and Σ a finite set of $\mathcal{L}$-identities with variables denoted by $\mathrm{Var}(\Sigma)$. Then $\mathsf{U}_{\mathcal{V}}(\Sigma)$ is defined as the set of $\mathcal{V}$-unifiers of Σ over $\mathrm{Var}(\Sigma)$ preordered by $\preccurlyeq$. For $\mathsf{U}_{\mathcal{V}}(\Sigma) \neq \emptyset$, the *$\mathcal{V}$-unification type* of Σ is defined as $\mathrm{type}(\mathsf{U}_{\mathcal{V}}(\Sigma))$. The *unification type of* $\mathcal{V}$ is the maximal type of a $\mathcal{V}$-unifiable finite set Σ of $\mathcal{L}$-identities.

We now recall Ghilardi's algebraic account of equational unification. Let $\mathbf{F}_{\mathcal{V}}(X)$ denote the free algebra of $\mathcal{L}$ over a set of variables X and let $h_{\mathcal{V}} \colon \mathbf{Fm}_{\mathcal{L}}(X) \to \mathbf{F}_{\mathcal{V}}(X)$ be the canonical homomorphism. Given a finite set of $\mathcal{L}$-identities Σ and a finite $X \supseteq \mathrm{Var}(\Sigma)$, we denote by $\mathrm{Fp}_{\mathcal{V}}(\Sigma, X)$ the algebra in $\mathcal{V}$ finitely presented by Σ and X: that is, the quotient algebra $\mathbf{F}_{\mathcal{V}}(X)/\Theta_\Sigma$ where Θ_Σ is the congruence generated by $\{(h_{\mathcal{V}}(\varphi), h_{\mathcal{V}}(\psi)) : \varphi \approx \psi \in \Sigma\}$. The class of finitely presented algebras in $\mathcal{V}$ is denoted by $\mathsf{FP}(\mathcal{V})$.

Given $\mathbf{A} \in \mathsf{FP}(\mathcal{V})$, a homomorphism $u \colon \mathbf{A} \to \mathbf{B}$ is called a *unifier* for $\mathbf{A}$ if $\mathbf{B} \in \mathsf{FP}(\mathcal{V})$ is *projective* in $\mathcal{V}$ (i.e., there exist homomorphisms $i \colon \mathbf{B} \to \mathbf{F}_{\mathcal{V}}(\omega)$ and $j \colon \mathbf{F}_{\mathcal{V}}(\omega) \to \mathbf{B}$ such that $j \circ i$ is the identity map on B). Let $u_i \colon \mathbf{A} \to \mathbf{B}_i$ for $i = 1, 2$ be unifiers for $\mathbf{A}$. Then u_1 is *more general than* u_2, written $u_2 \leq u_1$, if there exists a homomorphism $f \colon \mathbf{B}_1 \to \mathbf{B}_2$ such that $f \circ u_1 = u_2$. Let $\mathsf{U}_{\mathcal{V}}(\mathbf{A})$ be the set of unifiers of $\mathbf{A}$ preordered by $\leq$. For $\mathsf{U}_{\mathcal{V}}(\mathbf{A}) \neq \emptyset$, the *unification type of* $\mathbf{A}$ *in* $\mathcal{V}$ is defined as $\mathrm{type}(\mathsf{U}_{\mathcal{V}}(\mathbf{A}))$ and the *algebraic unification type* of $\mathcal{V}$ is the maximal type of $\mathbf{A}$ in $\mathsf{FP}(\mathcal{V})$ such that $\mathsf{U}_{\mathcal{V}}(\mathbf{A}) \neq \emptyset$. In [9], Ghilardi proved that $\mathrm{type}(\mathsf{U}_{\mathcal{V}}(\Sigma))$ coincides with $\mathrm{type}(\mathsf{U}_{\mathcal{V}}(\mathrm{Fp}_{\mathcal{V}}(\Sigma, \mathrm{Var}(\Sigma))))$, for each $\mathcal{V}$-unifiable finite set of identities Σ. Hence

the algebraic unification type of $\mathcal{V}$ coincides with the unification type of $\mathcal{V}$.

3 A New Hierarchy

Let us begin by pointing out the relevance of finitely presented algebras for admissibility. We freely identify $\mathcal{L}$-identities with pairs of $\mathcal{L}$-formulas and recall that the *kernel* of a homomorphism $h\colon \mathbf{A} \to \mathbf{B}$ is defined as $\ker(h) = \{(a,b) \in A^2 : h(a) = h(b)\}$.

Lemma 1. *The following are equivalent for any $\mathcal{L}$-clause $\Sigma \Rightarrow \Delta$ with $X = \mathrm{Var}(\Sigma \cup \Delta)$:*

(1) *$\Sigma \Rightarrow \Delta$ is admissible in $\mathcal{V}$.*

(2) *If $\sigma\colon \mathbf{Fm}_{\mathcal{L}}(X) \to \mathbf{Fm}_{\mathcal{L}}(\omega)$ satisfies $\Sigma \subseteq \ker(h_{\mathcal{V}} \circ \sigma)$, then $\Delta \cap \ker(h_{\mathcal{V}} \circ \sigma) \neq \emptyset$.*

Let X be a set of variables and let $\sigma_i\colon \mathbf{Fm}_{\mathcal{L}}(X) \to \mathbf{Fm}_{\mathcal{L}}(\omega)$ be substitutions for $i = 1, 2$. We write $\sigma_2 \sqsubseteq \sigma_1$ if all identities $\mathcal{V}$-unified by σ_1 are $\mathcal{V}$-unified by σ_2. More precisely:

$$\sigma_2 \sqsubseteq \sigma_1 \quad \Leftrightarrow \quad \ker(h_{\mathcal{V}} \circ \sigma_1) \subseteq \ker(h_{\mathcal{V}} \circ \sigma_2).$$

Observe immediately that $\sigma_2 \preccurlyeq \sigma_1$ implies $\sigma_2 \sqsubseteq \sigma_1$.

Given an equational class of algebras $\mathcal{V}$ for $\mathcal{L}$ and a finite set Σ of $\mathcal{L}$-identities, $\mathsf{E}_{\mathcal{V}}(\Sigma, X)$ is defined as the set of $\mathcal{V}$-unifiers of Σ over $X \supseteq \mathrm{Var}(\Sigma)$ preordered by $\sqsubseteq$. For $X = \mathrm{Var}(\Sigma)$, we simply write $\mathsf{E}_{\mathcal{V}}(\Sigma)$ instead of $\mathsf{E}_{\mathcal{V}}(\Sigma, X)$.

We define the *exact type of Σ in $\mathcal{V}$* to be $\mathrm{type}(\mathsf{E}_{\mathcal{V}}(\Sigma)))$ (for $\mathsf{E}_{\mathcal{V}}(\Sigma) \neq \emptyset$). Note that, because $\sigma_2 \preccurlyeq \sigma_1$ implies $\sigma_2 \sqsubseteq \sigma_1$, every complete set for $\mathsf{U}_{\mathcal{V}}(\Sigma)$ is also a complete set for $\mathsf{E}_{\mathcal{V}}(\Sigma)$. Therefore, if $\mathrm{type}(\mathsf{U}_{\mathcal{V}}(\Sigma)) \in \{1, \omega\}$, we have

$$\mathrm{type}(\mathsf{E}_{\mathcal{V}}(\Sigma)) \leq \mathrm{type}(\mathsf{U}_{\mathcal{V}}(\Sigma)).$$

We observe also that the choice of $\mathsf{E}_{\mathcal{V}}(\Sigma) = \mathsf{E}_{\mathcal{V}}(\Sigma, \mathrm{Var}(\Sigma))$ to define the exact type of Σ, is not restrictive; that is, for each $X \supseteq \mathrm{Var}(\Sigma)$,

$$\mathrm{type}(\mathsf{E}_{\mathcal{V}}(\Sigma)) = \mathrm{type}(\mathsf{E}_{\mathcal{V}}(\Sigma, X)).$$

Using Lemma 1, we obtain the desired relationship with admissibility: namely, to check the $\mathcal{V}$-admissibility of an $\mathcal{L}$-clause $\Sigma \Rightarrow \Delta$, it suffices to find a complete set (preferably a μ-set) S for $\mathsf{U}_{\mathcal{V}}(\Sigma)$ then check that each $\sigma \in S$ is a $\mathcal{V}$-unifier of some $\mathcal{L}$-identity in Δ.

Let us now give the algebraic characterization of exact unification. We call an algebra $\mathbf{E}$ *exact* in $\mathcal{V}$ if it is isomorphic to a finitely generated subalgebra of $\mathbf{F}_{\mathcal{V}}(\omega)$ (see also [8] for a syntactic characterization). Given $\mathbf{A} \in \mathsf{FP}(\mathcal{V})$, an onto homomorphism $u\colon \mathbf{A} \to \mathbf{E}$ is called a *coexact unifier* for $\mathbf{A}$ if $\mathbf{E}$ is exact. Coexact unifiers are ordered in the same way as algebraic unifiers, that is, if $u_i\colon \mathbf{A} \to \mathbf{E}_i$ for $i = 1, 2$ are coexact unifiers for $\mathbf{A}$, then $u_1 \leq u_2$, if there exists a homomorphism $f\colon \mathbf{E}_1 \to \mathbf{E}_2$ such that $f \circ u_1 = u_2$.

Let $\mathsf{EU}_{\mathcal{V}}(\mathbf{A})$ be the set of coexact unifiers for $\mathbf{A}$ preordered by $\leq$. If $\mathsf{EU}_{\mathcal{V}}(\mathbf{A}) \neq \emptyset$, then the *exact type* of $\mathbf{A}$ is defined as the type of $\mathsf{EU}_{\mathcal{V}}(\mathbf{A})$. We obtain the following Ghilardi-style result.

Theorem 2. *Let $\mathcal{V}$ be an equational class and Σ a finite set of $\mathcal{V}$-unifiable $\mathcal{L}$-identities. Then for any $X \supseteq \mathrm{Var}(\Sigma)$,*

$$\mathrm{type}(\mathsf{E}_{\mathcal{V}}(\Sigma)) = \mathrm{type}(\mathsf{E}_{\mathcal{V}}(\Sigma, X)) = \mathrm{type}(\mathsf{EU}_{\mathcal{V}}(\mathrm{Fp}_{\mathcal{V}}(\Sigma, X)).$$

Class of Algebras	Unification Type	Exact Type
Boolean Algebras	Unitary	Unitary
Heyting Algebras	Finitary	Finitary
Semigroups	Infinitary	Infinitary or Nullary
Modal algebras	Nullary	Nullary
Distributive Lattices	Nullary	Unitary
Stone Algebras	Nullary	Unitary
Bounded Distributive Lattices	Nullary	Finitary
Idempotent Semigroups	Nullary	Finitary
MV-algebras	Nullary	Finitary

Table 1: Comparison of unification types and exact types

We define the *exact unification type of* $\mathcal{V}$ to be the maximal exact type of a $\mathcal{V}$-unifiable finite set Σ of $\mathcal{L}$-identities. Similarly, the *exact algebraic unification type* of $\mathcal{V}$ is the maximal exact type of $\mathbf{A}$ in $\mathcal{V}$ such that $\mathsf{EU}_{\mathcal{V}}(\mathbf{A}) \neq \emptyset$. By Theorem 2, the exact unification type and the exact algebraic unification type of $\mathcal{V}$ coincide.

The close connection between coexact unifiers and congruences has as a byproduct that if a finitely presented algebra $\mathbf{A}$ has a finite set of congruences, then type($\mathsf{E}_{\mathcal{V}}(\Sigma)$) is 1 or ω. Hence we obtain the following useful corollary.

Corollary 3. *If $\mathcal{V}$ is a locally finite variety, then $\mathcal{V}$ has exact unification type* 1 *or* ω.

4 Examples

Any unitary equational class such as the class of Boolean algebras also has exact unitary type, and any finitary equational class will have unitary or finitary exact type. For example, the class of Heyting algebras is finitary [10] and hence also has finitary exact type: consider the identity $x \vee y \approx \top$ and unifiers σ_1 with $\sigma_1(x) = \top$, $\sigma_1(y) = y$ and σ_2 with $\sigma_2(x) = x$, $\sigma_2(y) = \top$.

Minor changes to the original proofs that the class of semigroups has infinitary unification type [22] and that the class of modal algebras (for the logic K) has nullary unification type [14] establish that the former has infinitary or nullary exact type and the latter has nullary exact type. However, the class of distributive lattices, which is known to have nullary unification type [9], has unitary exact type as all finitely presented distributive lattices are exact. Similarly, the class of Stone algebras has nullary unification type but unitary exact type. The classes of bounded distributive lattices and idempotent semigroups are also both nullary, but because they are locally finite, they have at most – and indeed, it can be shown via suitable cases, precisely – finitary exact type.

In [20] it is proved that the equational class of MV-algebras has nullary unification type. This class is not locally finite so we cannot apply Corollary 3; however, combining results from [17] and [5], we can still prove that MV-algebras have finitary exact type. We observe that because finitely presented MV-algebras admit a presentation of the form $\{\alpha \approx \top\}$ and [5, Theorem 4.18] proves that every admissible saturated formula ([16, Definition 3.1]) is exact, [17, Theorem 3.8] effectively provides a bound on the exact type of a finitely presented algebra. Note, moreover, that each formula in the admissible saturated approximation (defined in [16, 17]) of a formula

α corresponds to an exact unifier of the identity $\alpha \approx \top$. Similarly in [6], the current authors present axiomatizations for admissible rules of several locally finite (and hence of finitary exact unification type) equational classes with classical unification type 0. In all these cases a complete description of exact algebras, and the finite exact unification type plays a central (if implicit) role. We therefore expect this approach to be useful for tackling other classes of algebras that have unitary or finitary exact type, independently of their unification type.

These examples are collected in Table 1, noting that we do not know if there are examples of equational classes of (i) finitary unification type that have unitary exact type, (ii) infinitary unification type that have unitary, finitary, or nullary exact type, (iii) nullary unification type that have infinitary exact type.

References

[1] F. Baader and W. Snyder. Unification theory. In *Handbook of Automated Reasoning*, volume I, chapter 8, pages 447–533. Elsevier Science B.V., 2001.

[2] S. Babenyshev and V. Rybakov. Linear temporal logic LTL: Basis for admissible rules. *Journal of Logic and Computation*, 21(2):157–177, 2011.

[3] S. Babenyshev and V. Rybakov. Unification in linear temporal logic LTL. *Annals of Pure and Applied Logic*, 162(12):991–1000, 2011.

[4] S. Burris and H. P. Sankappanavar. *A Course in Universal Algebra*, volume 78 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1981.

[5] L. M. Cabrer. Simplicial geometry of unital lattice ordered abelian groups. *Forum Mathematicum*, (in press, DOI:10.1515/forum-2011-0131).

[6] L. M. Cabrer and G. Metcalfe. Admissibility via natural duality. Submitted, 2013.

[7] P. Cintula and G. Metcalfe. Admissible rules in the implication-negation fragment of intuitionistic logic. *Annals of Pure and Applied Logic*, 162(10):162–171, 2010.

[8] D. H. J. de Jongh. Formulas of one propositional variable in intuitionistic arithmetic. In *Stud. Log. Found. Math. 110, The L. E. J. Brouwer Centenary Symposium, Proceedings of the Conference held in Noordwijkerhout*, pages 51–64. Elsevier, 1982.

[9] S. Ghilardi. Unification through projectivity. *Journal of Logic and Computation*, 7(6):733–752, 1997.

[10] S. Ghilardi. Unification in intuitionistic logic. *Journal of Symbolic Logic*, 64(2):859–880, 1999.

[11] S. Ghilardi. Best solving modal equations. *Annals of Pure and Applied Logic*, 102(3):184–198, 2000.

[12] R. Iemhoff. On the admissible rules of intuitionistic propositional logic. *Journal of Symbolic Logic*, 66(1):281–294, 2001.

[13] R. Iemhoff. Intermediate logics and Visser's rules. *Notre Dame Journal of Formal Logic*, 46(1):65–81, 2005.

[14] E. Jeřábek. Blending margins: the modal logic K has nullary unification type. To appear in *Journal of Logic and Computation*.

[15] E. Jeřábek. Admissible rules of modal logics. *Journal of Logic and Computation*, 15:411–431, 2005.

[16] E. Jeřábek. Admissible rules of Łukasiewicz logic. *Journal of Logic and Computation*, 20(2):425–447, 2010.

[17] E. Jeřábek. Bases of admissible rules of Łukasiewicz logic. *Journal of Logic and Computation*, 20(6):1149–1163, 2010.

[18] E. Jeřábek. The complexity of admissible rules of łukasiewicz logic. *Journal of Logic Computation*, 23(3):693–705, 2013.

[19] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, 1971.

[20] V. Marra and L. Spada. Duality, projectivity, and unification in Łukasiewicz logic and MV-algebras. *Annals of Pure and Applied Logic*, 164(3):192–210, 2013.

[21] S. Odintsov and V. Rybakov. Unification and admissible rules for paraconsistent minimal Johanssons' logic J and positive intuitionistic logic IPC^{+}. *Annals of Pure and Applied Logic*, 164(7-8):771–784, 2013.

[22] G. Plotkin. Building in equational theories. *Machine Intelligence*, 7:73–90, 1972.

[23] V. Rybakov. *Admissibility of Logical Inference Rules*, volume 136 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, Amsterdam, 1997.

Constraint Manipulation in SGGS

Maria Paola Bonacina[1] and David A. Plaisted[2]

[1] Dipartimento di Informatica, Università degli Studi di Verona, Italy
mariapaola.bonacina@univr.it

[2] Department of Computer Science, UNC at Chapel Hill, USA
plaisted@cs.unc.edu

Abstract

SGGS (Semantically-Guided Goal-Sensitive theorem proving) is a clausal theorem-proving method, with a seemingly rare combination of properties: it is first order, DPLL-style model based, semantically guided, goal sensitive, and proof confluent. SGGS works with *constrained clauses*, and uses a *sequence* of constrained clauses to represent a tentative model of the given set of clauses. A basic building block in SGGS inferences is *splitting*, which partitions a clause into clauses that have the same set of ground instances. Splitting introduces constraints and their manipulation, which is the subject of this paper. Specifically, splitting a clause with respect to another clause requires to compute their *difference*, which captures the ground instances of one that are not ground instances of the other. We give a set of inference rules to compute clause difference, and reduce SGGS constraints to *standard form*, and we prove that it is guaranteed to terminate, provided the standardization rules are applied within the clause difference computation.

Introduction

The SGGS theorem-proving method combines instance generation, resolution, and constraint solving in a *model-based* framework. It works with a set S of first-order clauses to be refuted and an *initial interpretation* I for *semantic guidance*. The features of SGGS can be seen as an attempt to build a model of S, distinct from I. The search for a model of S is done by constructing an SGGS derivation, which is a series $\Gamma_0 \vdash \Gamma_1 \vdash \Gamma_2 \vdash \ldots$ of objects Γ, called *SGGS clause sequences*. After Γ_0, which is empty, each Γ_i is obtained from the previous one by an SGGS inference rule.

An SGGS derivation terminates, if either a refutation is found, or no more inference rules can be applied. SGGS is *refutationally complete*: if S is unsatisfiable, there exist SGGS derivations from S that terminate with the generation of the empty clause. If S is satisfiable, the derivation may be infinite, and if so will in the limit represent a model of S. At each step the new clause sequence replaces the old one, so that only one clause sequence exists at any time, and SGGS is *proof confluent*: performing an inference will never prevent it from finding a refutation, so that *there is no need for backtracking*.

A key property of SGGS is that an SGGS clause sequence represents a candidate partial model. While in propositional logic, a model is represented by a sequence of literals (e.g., as in DPLL), in SGGS a first-order model is represented by a sequence of *constrained clauses*, each of which has a *selected literal*. The model $I[\Gamma]$ represented by a sequence Γ is given by the initial interpretation I modified to satisfy ground instances of selected literals. Informally, the literal L selected in the n-th clause C in Γ contributes to $I[\Gamma]$ its ground instances $L\sigma$ that are needed ($C\sigma$ is not already true in the model induced by the first $n-1$ clauses of Γ) and consistent ($\neg L\sigma$ is not true in the model induced by the first $n-1$ clauses of Γ). Formally, $I[\Gamma]$ is defined inductively on the length of the sequence [2, 1].

The inference rules of SGGS implement a search for a model thus represented. The main rule is *extension*, which adds to the current clause sequence an instance of a clause in S: the

objective is to find a model of all instances of all clauses in S, and if some are not satisfied, they must be added.

It may happen that selected literals have ground instances in common. If the literals have opposite sign, this would make the model *inconsistent*: SGGS features a restricted form of resolution, called *SGGS-resolution*, to remove such contradictions. SGGS-resolution represents an implicit sort of backtracking over the set of possible models of S. The resolvent is a *lemma*, that constrains the model, because the model must satisfy it, and intuitively captures a portion of the search space of models that has been explored. If resolution generates the empty clause, no model can be found.

If selected literals have ground instances in common, and have the same sign, there is *duplication*. SGGS features *splitting rules* that partition a clause with respect to another clause. The clause that gets partitioned, or split, is replaced by other clauses, that have its same set of ground instances, in such a way that the duplicated literals are isolated and can be removed.

The splitting rules of SGGS are the motivation for this paper. The splitting of a clause with respect to another clause can be computed by computing unification of selected literals, and the *difference* between two clauses. Intuitively, the splitting of C with respect to D replaces C by a set of clauses one of which captures the (constrained) ground instances of C that are also (constrained) ground instances of D, while the others capture the (constrained) ground instances of C that are not (constrained) ground instances of D. The latter form the difference between C and D, which is what matters in practice, since we want to remove the duplication.

In this paper, we illustrate the ingredients of SGGS that are relevant to constraint solving: SGGS constraints, constrained clauses, and the concepts of splitting of clauses and difference between clauses. We give a system of rules for constraint manipulation to compute clause differences, whence splittings, and reduce SGGS constraints to standard form. Then we discuss *termination*: while unrestricted applications of the standardization rules may not terminate, the computation of clause difference, and restoration of standard form during the computation of clause difference, are proved to terminate.

For the interested reader, a technical presentation of SGGS, including inference system, fairness, and proofs of refutational completeness and goal-sensitivity, is available in [2]. A non-technical exposition is offered in [3]. The representation of models by SGGS clause sequences is studied in its own right in [1].

Constrained Clauses and Splitting

We assume standard concepts and notations in clausal theorem proving. In addition, $\equiv$ is syntactic identity; $top(t)$ is the top symbol of term t; $at(L)$ is the atom of literal L; $at(T) = \{at(L) : L \in T\}$ for T a set of literals; $vars(C)$ is the set of variables in clause C, and the same notation applies to terms; clauses are *variants*, if made identical by a variable renaming, *similar*, if made identical by a substitution that replaces variables by variables, but may replace distinct variables by the same.

SGGS Constraints

In SGGS, an *atomic constraint* is either empty, denoted by *true*, or an expression of the form $x \equiv y$ or $top(t) = f$, where x and y are variables, f is a function symbol, and t is a term. Then, a *constraint* is either an atomic constraint, or the negation, conjunction, or disjunction of constraints.

SGGS constraints assume *Herbrand interpretations*: let $\models$ mean truth in all Herbrand interpretations; then, $\models t \equiv u$ for ground terms t and u if t and u are the same element of the

Herbrand universe; and $\models top(t) = f$ if the top symbol of ground term t is f. Thus, if $A\vartheta$ is a ground instance of a constraint A, either $\models A\vartheta$ or $\models \neg A\vartheta$.

SGGS constraints are a variant of *Herbrand constraints*: they are *Herbrand constraints* with the addition of atomic constraints of the form $top(t) = f$, which allow us to avoid existential quantifiers in the constraints, since $top(t) = f$ replaces $\exists x_1 \ldots \exists x_n.\ t \equiv f(x_1, \ldots, x_n)$.

An SGGS constraint is in *standard form*, if it is a conjunction of distinct atomic constraints of the form $x \not\equiv y$ and $top(x) \neq f$, where x and y are variables. A constraint $top(x) \neq f$ says that x cannot be replaced by a term whose top function symbol is f, while a constraint $x \not\equiv y$ specifies that x and y may not be replaced by identical terms.

A *constrained clause* is a formula $A \rhd C$, where A is a constraint and C is a clause; a variable that appears in A but not in C is implicitly existentially quantified. A constrained clause $A \rhd C$ may have a *selected* literal L, written $A \rhd C[L]$. $A \rhd L$ is called *constrained literal*. By convention, if L is selected in C, and $C' \equiv C\vartheta$, then $L' \equiv L\vartheta$ is selected in C'.

The *constrained ground instances* (cgi) of $A \rhd C$ are the ground instances of C that satisfy the constraints: $Gr(A \rhd C) = \{C\vartheta :\ \models A\vartheta,\ C\vartheta\ ground\}$, where $\models$ means truth in all Herbrand interpretations. Similarly, $Gr(A \rhd L) = \{L\vartheta :\ \models A\vartheta,\ L\vartheta\ ground\}$. For example, $P(a, b) \in Gr(x \not\equiv y \rhd P(x, y))$, but $P(b, b) \notin Gr(x \not\equiv y \rhd P(x, y))$. A constrained clause (literal) represents its constrained ground instances.

Partition, Splitting, and Difference

Since SGGS uses constrained literals and clauses to exhibit a partial model, it needs to know when constrained literals have instances in common: $A \rhd L$ and $B \rhd M$ *intersect* if $at(Gr(A \rhd L)) \cap at(Gr(B \rhd M)) \neq \emptyset$, and are *disjoint*, otherwise. Intersection does not require the literals to have the same sign, because it is defined based on atoms.

If $A \rhd L$ and $B \rhd M$ do not share variables, they intersect if and only if $at(L)$ and $at(M)$ unify and $(A \wedge B)\sigma$ is satisfiable, where σ is the mgu (most general unifier) of $at(L)$ and $at(M)$. The intersection is given by $at(Gr(A \rhd L)) \cap at(Gr(B \rhd M)) = at(Gr((A \wedge B)\sigma \rhd M\sigma)) = Gr((A \wedge B)\sigma \rhd at(M)\sigma)$.

A *partition* of $A \rhd C[L]$, where A is satisfiable, is a set $\{A_i \rhd C_i[L_i]\}_{i=1}^n$ such that $Gr(A \rhd C) = \bigcup_{i=1}^n \{Gr(A_i \rhd C_i[L_i])\}$, the $A_i \rhd L_i$'s are pairwise disjoint, the A_i's are satisfiable, and the L_i's are chosen consistently with L.

For example, $\{true \rhd P(f(z), y),\ top(x) \neq f \rhd P(x, y)\}$ is a partition of $true \rhd P(x, y)$ (which can of course be written simply $P(x, y)$). Similarly,

$$\{true \rhd [P(f(z), y)] \vee Q(f(z), y),\ top(x) \neq f \rhd [P(x, y)] \vee Q(x, y)\}$$

is a partition of $true \rhd [P(x, y)] \vee Q(x, y)$. On the other hand,

$$\{true \rhd P(f(z), y) \vee [Q(f(z), y)],\ top(x) \neq f \rhd P(x, y) \vee [Q(x, y)]\}$$

is not a partition of $true \rhd [P(x, y)] \vee Q(x, y)$, because selected literals are not chosen consistently.

If clauses $A \rhd C[L]$ and $B \rhd D[M]$ in an SGGS clause sequence have selected literals L and M that intersect, SGGS features inference rules that replace $A \rhd C[L]$ by $split(C, D)$, that is a partition of $C[L]$, where all cgi's of L that are also cgi's of M are isolated in one of the clauses of the partition. Formally, a *splitting* of $A \rhd C[L]$ by $B \rhd D[M]$, denoted $split(C, D)$, is a partition $\{A_i \rhd C_i\langle L_i\rangle\}_{i=1}^n$ of $A \rhd C[L]$ such that:

1. $\exists j$, $1 \leq j \leq n$, such that $at(Gr(A_j \rhd L_j)) \subseteq at(Gr(B \rhd M))$, and
2. $\forall i$, $1 \leq i \neq j \leq n$, $at(Gr(A_i \rhd L_i))$ and $at(Gr(B \rhd M))$ are disjoint.

The *difference* $C - D$ is $split(C, D)$ with C_j removed. Clause C_j is the *representative* of D in $split(C, D)$: $at(Gr(A_j \rhd L_j))$ is the intersection of $A \rhd L$ and $B \rhd M$, while $C - D$ captures the cgi's of L that are not cgi's of M. We write $Gr(C - D)$ for $\bigcup_{i=1, i\neq j}^{n} Gr(C_i)$.
For example, a splitting of $true \rhd P(x, y)$ by $true \rhd P(f(w), g(z))$ is

$$\{true \rhd P(f(w), g(z)),\ top(x) \neq f \rhd P(x, y),\ top(y) \neq g \rhd P(f(x), y)\}$$

and their difference is $\{top(x) \neq f \rhd P(x, y),\ top(y) \neq g \rhd P(f(x), y)\}$. On the other hand,

$$\{true \rhd P(f(w), g(z)),\ top(x) \neq f \rhd P(x, y),\ top(y) \neq g \rhd P(x, y)\}$$

is not a splitting of $true \rhd P(x, y)$ by $true \rhd P(f(w), g(z))$, because it is not a partition, since $top(x) \neq f \rhd P(x, y)$ and $top(y) \neq g \rhd P(x, y)$ intersect: for instance, $P(a, b)$ is a cgi of both. In the correct splitting, $P(a, b)$ is a cgi of $top(x) \neq f \rhd P(x, y)$, not of $top(y) \neq g \rhd P(f(x), y)$.

As this example shows, computing $split(C, D)$ and $C - D$ introduces constraints, including non-standard ones, even when C and D have empty constraints to begin with. This is precisely why SGGS works with constrained clauses.

If $at(L)$ and $at(M)$ do not unify, $Gr(C - D) = Gr(C)$; if they unify with mgu σ, then $split(C, D) = (C - D) \cup \{A\sigma \wedge B\sigma \rhd C[L]\sigma\}$, and $(C - D) = (C - (A\sigma \wedge B\sigma \rhd C[L]\sigma))$. Thus, if we have a way to compute $C - D$, we also have a way to compute $split(C, D)$, and we can restrict ourselves to compute $C - D$ under the assumption that D is an instance of C.

Rules to Compute Clause Difference and Standardize Constraints

The following rules are *sound*, as premise and conclusion represent the same set of cgi's. If a conclusion has the form $A_1 \rhd C_1, \ldots, A_n \rhd C_n$, it is a disjunction, and represents $\bigcup_{i=1}^{n} Gr(A_i \rhd C_i)$. We begin with rules to compute $C - D$ when $D \equiv C\sigma$.

Rules for Clause Difference and Disjunctive Normal Form

If $\{x \leftarrow f(x_1, \ldots, x_n)\} \subseteq \sigma$ for some $x \in vars(C)$ and new variables x_i, $1 \leq i \leq n$, the *DiffSim rule* applies $\{x \leftarrow f(x_1, \ldots, x_n)\}$ to make C similar to D and on the other hand adds $top(x) \neq f$ to make them different:

$$\frac{(A \rhd C) - (B \rhd D)}{(A \rhd C)\{x \leftarrow f(x_1, \ldots, x_n)\} - (B \rhd D),\ A \wedge (top(x) \neq f) \rhd C}$$

If C and D are similar, and $\{x \leftarrow y\} \subseteq \sigma$ for distinct variables $x, y \in vars(C)$, the *DiffVar rule* applies $\{x \leftarrow y\}$ to make C a variant of D and on the other hand adds $x \not\equiv y$ to make them different:

$$\frac{(A \rhd C) - (B \rhd D)}{(A \rhd C)\{x \leftarrow y\} - (B \rhd D),\ (x \not\equiv y \wedge A) \rhd C}$$

If C and D are variants but not identical, the *DiffId* rule makes them identical:

$$\frac{(A \rhd C) - (B \rhd D)}{(A \rhd C)\sigma - (B \rhd D)}$$

The *DiffElim rule* replaces difference by negation:

$$\frac{(A \rhd C) - (B \rhd C)}{(A \wedge \neg B) \rhd C}$$

Since B is a conjunction of constraints, $\neg B$ is a disjunction of their negations. The next rules restore disjunctive normal form (DNF). The *Equiv rule* replaces a constraint by its DNF:

$$\frac{A \rhd C}{dnf(A) \rhd C}$$

where $dnf(A)$ is the disjunctive normal form of A; and the *Div rule* subdivides disjunction:

$$\frac{(A \vee B) \rhd C}{A \rhd C,\ B \rhd C}$$

Rules for Reduction to Standard Form

The rules for reduction to standard form comprise rules for identity and rules for top symbol. The *rules for identity* eliminate or decompose all identity constraints, except those in standard form $x \not\equiv y$.
The *ElimId1 rule* eliminates a constraint between variable and term: if $x \notin vars(s)$, then:

$$\frac{(A \wedge x \equiv s) \rhd C}{(A \rhd C)\{x \leftarrow s\}}$$

if $x \in vars(s)$ and s is not a variable, then:

$$\frac{(A \wedge x \equiv s) \rhd C}{\bot}$$

$$\frac{(A \wedge x \not\equiv s) \rhd C}{(A \rhd C)}$$

The *ElimId2 rule* detects a conflict: if $f \neq g$, $m \geq 0$, $n \geq 0$, then:

$$\frac{(A \wedge f(s_1, \ldots, s_n) \equiv g(t_1, \ldots, t_m)) \rhd C}{\bot}$$

The *ElimId3 rule* eliminates a satisfied constraint: if $f \neq g$, $m \geq 0$, $n \geq 0$, then:

$$\frac{(A \wedge f(s_1, \ldots, s_n) \not\equiv g(t_1, \ldots, t_m)) \rhd C}{A \rhd C}$$

The *ElimId4 rule* decomposes an identity: if $n \geq 0$, then:

$$\frac{(A \wedge f(s_1, \ldots, s_n) \equiv f(t_1, \ldots, t_n)) \rhd C}{(A \wedge s_1 \equiv t_1 \wedge \ldots \wedge s_n \equiv t_n) \rhd C}$$

The *ElimId5 rule* decomposes a negated identity: if $n \geq 0$, then:

$$\frac{(A \wedge f(s_1, \ldots, s_n) \not\equiv f(t_1, \ldots, t_n)) \rhd C}{(A \wedge (s_1 \not\equiv t_1 \vee \ldots \vee s_n \not\equiv t_n)) \rhd C}$$

The *ElimId6 rule* eliminates a negated identity between variable and non-variable term:

$$\frac{(A \wedge x \not\equiv f(s_1, \ldots, s_n)) \rhd C}{A \wedge top(x) \neq f \rhd C,\ ((A \wedge f(s_1, \ldots, s_n) \not\equiv f(y_1, \ldots, y_n)) \rhd C)\{x \leftarrow f(y_1, \ldots, y_n)\}}$$

where $n \geq 0$, and the y_i's, $1 \leq i \leq n$, are new variables.
The *ElimId7 rule* detects a conflict: if s is a variable or constant, then:

$$\frac{(A \wedge s \not\equiv s) \rhd C}{\bot}$$

As an example, consider computing $split(C, D)$, where $A \rhd C[L]$ is $true \rhd P(x, f(x))$ and $B \rhd D[M]$ is $x \not\equiv y \rhd P(x, y)$. After renaming variables in the second clause, so that $B \rhd D[M]$ becomes $x' \not\equiv y \rhd P(x', y)$, the unification of $at(L) = P(x, f(x))$ and $at(M) = P(x', y)$, yields mgu $\sigma = \{x' \leftarrow x, y \leftarrow f(x)\}$, so that $A\sigma \wedge B\sigma \rhd C[L]\sigma$ is $x \not\equiv f(x) \rhd P(x, f(x))$. The ElimId1 rule reduces this clause to $true \rhd P(x, f(x))$, which is the same as $A \rhd C[L]$. Thus, $C - D = C - C$, or the difference is empty, because indeed $Gr(A \rhd C[L]) \subseteq Gr(B \rhd D[M])$. Accordingly, $split(C, D)$ is $A \rhd C[L]$ itself, or the splitting operation leaves the clause unchanged, because we tried to split a clause by a more general one.

The *rules for top symbol* eliminate all top symbol constraints, except those in standard form $top(x) \neq f$.
The *ElimTop1 rule* detects a conflict in a positive constraint: if $f \neq g$, $n \geq 0$, then:

$$\frac{A \wedge top(f(s_1, \ldots, s_n)) = g \rhd C}{\bot}$$

The *ElimTop2 rule* eliminates a satisfied positive constraint: if $n \geq 0$, then:

$$\frac{A \wedge top(f(s_1, \ldots, s_n)) = f \rhd C}{A \rhd C}$$

The *ElimTop3 rule* eliminates a satisfied negative constraint: if $f \neq g$, $n \geq 0$, then:

$$\frac{A \wedge top(f(s_1, \ldots, s_n)) \neq g \rhd C}{A \rhd C}$$

The *ElimTop4 rule* detects a conflict in a negated constraint: if $n \geq 0$, then:

$$\frac{A \wedge top(f(s_1, \ldots, s_n)) \neq f \rhd C}{\bot}$$

The *ElimTop5 rule* eliminates a positive constraint: if $n \geq 0$, then:

$$\frac{A \wedge top(x) = f \rhd C}{(A \rhd C)\{x \leftarrow f(x_1, \ldots, x_n)\}}$$

where for all i, $1 \leq i \leq n$, x_i is a new variable.
The combined effect of all rules is to standardize all constraints.

Termination

The application of the identity rules may not terminate in general. For example, consider a clause $(x \not\equiv f(y) \wedge y \not\equiv f(x) \rhd P(x, y))$: ElimId6 yields the two clauses $(top(x) \neq f \wedge y \not\equiv f(x)) \rhd P(x, y)$ and $(f(z) \not\equiv f(y) \wedge y \not\equiv f(f(z)) \rhd P(f(z), y))$. Using ElimId5, the latter clause becomes $(z \not\equiv y \wedge y \not\equiv f(f(z)) \rhd P(f(z), y))$, which by another application of ElimId6, yields the two clauses $(z \not\equiv y \wedge top(y) \neq f) \rhd P(f(z), y))$ and $(z \not\equiv f(w) \wedge f(w) \not\equiv f(f(z)) \rhd P(f(z), f(w)))$. Using ElimId5 again, the latter clause becomes $(z \not\equiv f(w) \wedge w \not\equiv f(z) \rhd P(f(z), f(w)))$, whose constraint is a variant of the original one.

Nonetheless, SGGS does not need that every series of applications of these rules terminate. It suffices that the computation of clause difference terminates:

Theorem 1. *Given $A \rhd C$ and $B \rhd D$, such that $D \equiv C\sigma$, and A and B are in standard form, any application of the clause difference rules to $C - D$, where (1) any application of DiffElim or ElimId5 is followed by conversion to DNF, and (2) all constraints are restored to standard form after every application of a clause difference rule, is guaranteed to terminate.*

Proof. First we show that the rules for clause difference do not cause non-termination. DiffId and DiffElim can be applied only once. DiffVar can be applied only a finite number of times, because each application decreases the number of variables in C. Each DiffSim step applies to C a substitution $\{x \leftarrow f(x_1, \ldots x_n)\}$ from σ: since σ contains finitely many such pairs, DiffSim can be applied only a finite number of times. Then we prove that standardization between an application of a clause difference rule and the next is guaranteed to terminate:

1. DiffId only renames variables, which does not enable any other rule.
2. DiffVar adds an $x \not\equiv y$, which is in standard form, and applies a substitution $\{x \leftarrow y\}$, whose only effect may be to replace an $x \not\equiv y$ by an $x \not\equiv x$, eliminated by ElimId7.
3. DiffSim adds a $top(x) \neq f$, which is in standard form, and applies a substitution $\{x \leftarrow f(x_1, \ldots, x_n)\}$, which may have two effects. One is to replace the occurrence of x in a constraint $top(x) \neq g$ by $f(x_1, \ldots, x_n)$. This enables either ElimTop3 or ElimTop4, which terminate. The other is to transform an $x \not\equiv y$ into an $f(x_1, \ldots, x_n) \not\equiv y$, enabling ElimId6. This rule adds a $top(x) \neq f$, which is in standard form, and applies another substitution of the same form, so that eventually a subset of the variables may be replaced by terms $f(x_1, \ldots, x_n)$ where the x_i's are new. This can only be done a finite number of times, because the new variables will never be replaced in this way. If two such substitutions are applied to a $z \not\equiv w$, an $f(x_1, \ldots, x_n) \not\equiv f(y_1, \ldots, y_n)$ may arise. ElimId5 applies to such a constraint, followed by conversion to DNF. The result is a disjunction of constrained clauses, each containing in its constraint an $x_i \not\equiv y_i$, for some i, which is in standard form.
4. DiffElim yields $(A \wedge \neg B) \rhd C$, followed by conversion to DNF. The effect may be to add $x \equiv y$ (negation of $x \not\equiv y$ in B) or $top(x) = f$ (negation of $top(x) \neq f$ in B). In the first case, ElimId1 applies $\{x \leftarrow y\}$, covered in Case (2) of this proof. In the second case, ElimTop5 applies $\{x \leftarrow f(x_1, \ldots, x_n)\}$, covered in Case (3) of this proof.

□

Discussion

We presented a set of inference rules to compute the difference between two constrained clauses, and to reduce to standard form SGGS constraints. We showed by a counter-example that it is not the case that any application of the inference rules for standardization is guaranteed to terminate. Then we proved that computation of clause difference is guaranteed to terminate, and that standardization in the context of computing clause differences is also guaranteed to terminate.

SGGS is a new reasoning method that uses sequences of constrained clauses to represent candidate partial models, during the search for a refutation, or a model, of a set of first-order clauses. When clauses in the sequence contribute to the candidate partial model sets of ground literals with non-empty intersection, there is a duplication. SGGS removes this duplication by inferences that split a clause with respect to another. Computing this splitting requires to compute unification of literals and differences of clauses, whence the interest for the difference operation.

SGGS constraints are a variant of Herbrand constraints (e.g., [6, 7, 5, 4]): they feature atomic constraints in the form $top(t) = f$, which allow one to avoid existential quantifiers in

constraints. If $top(t) = f$ is replaced with $\exists x_1 \ldots \exists x_n.\ t \equiv f(x_1, \ldots, x_n)$, SGGS constraints fit in the first-order logic of equations between trees.

Inference systems to decide the truth in the Herbrand universe of first-order formulæ with equality as the only predicate symbol were given independently in [6, 7] and [5]. Our inference system and termination result are tailored for the SGGS reasoning method; they capture what is needed precisely by SGGS, and therefore they are relevant to its understanding and implementation. More study may clarify a more precise relationship between our work in this paper and that in [6, 7] and [5]. Another possible topic for future investigation is the complexity of these procedures. The research in [6, 7] and [5] was motivated primarily by logic programming with constraints. It is interesting that those results may turn out to be useful for a theorem-proving method after several years.

References

[1] Maria Paola Bonacina and David A. Plaisted. Model representation by SGGS clause sequences. Submitted for publication (24 pages), 2014.

[2] Maria Paola Bonacina and David A. Plaisted. Semantically-guided goal-sensitive theorem proving. Technical Report 92/2014, Dipartimento di Informatica, Università degli Studi di Verona, Verona, Italy, January 2014. Revised April 2014, available at `http://profs.sci.univr.it/~bonacina/pub_tr.html` (56 pages).

[3] Maria Paola Bonacina and David A. Plaisted. SGGS theorem proving: an exposition. In Leonardo De Moura Boris Konev and Stephan Schulz, editors, *Notes of the Fourth Workshop on Practical Aspects in Automated Reasoning (PAAR), Seventh International Joint Conference on Automated Reasoning (IJCAR) and Sixth Federated Logic Conference (FLoC)*, EasyChair Proceedings in Computing (EPiC), pages 1–14, July 2014.

[4] Hubert Comon. Disunification: a survey. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 322–359. The MIT Press, 1991.

[5] Hubert Comon and Pierre Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.

[6] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. Technical report, IBM, Thomas J. Watson Research Center, Yorktown Heights, New York, USA, 1988.

[7] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 348–457. IEEE Computer Society Press, 1988.

Two-sided unification is NP-complete

Tatyana A. Novikova[1] and Vladimir A. Zakharov[2]

[1] Kazakhstan Branch of Lomonosov Moscow State University
[2] Lomonosov Moscow State University
(taniaelf@mail.ru, zakh@cs.msu.su)

Abstract

It is generally accepted that to unify a pair of substitutions θ_1 and θ_2 means to find out a pair of substitutions η' and η'' such that the compositions $\theta_1\eta'$ and $\theta_2\eta''$ are the same. Actually, unification is the problem of solving linear equations of the form $\theta_1 X = \theta_2 Y$ in the semigroup of substitutions. But some other linear equations on substitutions may be also viewed as less common variants of unification problem. In this paper we introduce a two-sided unification as the process of bringing a given substitution θ_1 to another given substitution θ_2 from both sides by giving a solution to an equation $X\theta_1 Y = \theta_2$. Two-sided unification finds some applications in software refactoring as a means for extracting instances of library subroutines in arbitrary pieces of program code. In this paper we study the complexity of two-sided unification and show that this problem is NP-complete by reducing to it the bounded tiling problem.

1 Introduction

To unify a pair of expressions E_1 and E_2 means to compute such instances of these expressions that are identical (syntactical unification) or have the same meaning (semantical unification). Such common instances of E_1 and E_2 can be obtained by replacing some variables in E_1 and E_2 by appropriate terms, i.e. by applying some substitutions to these expressions. Unification algorithms have found a wide utility in theorem proving, logic programming, term rewriting, type inference, language processing, etc. (see [1, 2]). In [9] it was shown that unification problem is also meaningful and efficiently decidable when expressions E_1 and E_2 are some formal models in imperative programs. If the programs are unifiable then their behaviors are somewhat similar; therefore, some results of the analysis of one program (proofs of its correctness, termination, etc.) can be easily adapted to the other.

But a similarity of programs can be formalized differently. Suppose that one has a library subroutine $\pi_0(\vec{x}\,{:}\,\mathit{input};\ \vec{y}\,{:}\,\mathit{output})$ with a set of formal input arguments $\vec{x}$ and a set of formal output parameters $\vec{y}$. Given some piece of program code π_1 one may wonder if it is possible to replace it with an appropriate subroutine call. Such a replacement would make the program both succinct and uniform which is very much helpful for program understanding and analysis. To this end one could try to find such instantiation η'' of input arguments $\vec{x}$ and such specialization η' of output parameters $\vec{y}$ as to make the composition of η', π_0, and η'' equivalent to π_1. In some formal models of programs (see [4, 8, 9]) a behavior of a program π can be specified by a substitution θ_π which assigns terms on input arguments $\vec{x}$ to output parameters $\vec{y}$. Thus, we can set up the following problem: given a pair of substitutions θ_{π_0} and θ_{π_1} find a pair of substitutions η'' (input instantiation) and η' (output specialization) such that the composition $\eta'\theta_{\pi_0}\eta''$ is equal to θ_{π_1}, or, in other words, solve the equation $X\theta_{\pi_0}Y = \theta_{\pi_1}$ in the semigroup of substitutions. It is worth noticing that the conventional unification problem may be regarded as that of solving linear equations of the form $\theta_1 X = \theta_2 Y$ in the semigroup of substitutions when both unknown substitutions are applied to θ_1 and θ_2 from the one side. Therefore, we call the solving of equations of the form $X\theta_0 Y = \theta_1$ when unknowns appear on both sides of

θ_0 *two-sided unification* of substitutions θ_0 and θ_1. In this paper we show that the problem of two-sided unification for first-order substitutions is NP-complete.

The paper is organized as follows. In Section 2 we recall briefly the basic notions concerning first-order substitutions, set up formally two-sided unification problem, and show that it is in NP. Afterward, in Section 3 we consider BOUNDED TILING problem which is widely used in complexity theory (see [3]) as an alternative to SATISFIABILITY. Finally, in Section 3 we prove that BOUNDED TILING is reducible to two-sided unifiability problem.

2 Preliminaries.

We deal with the first-order language over some fixed sets of functional symbols $\mathcal{F}$. Letters $\mathcal{U}, \mathcal{X}, \mathcal{Y}, \mathcal{Z} \ldots$ will be used for pairwise disjoint finite sets of variables. The set of terms $Term[\mathcal{X}]$ over a set of variables is defined as usual.

Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ and $\mathcal{Y} = \{y_1, y_2, \ldots\}$ be two sets of variables. A $\mathcal{X}$-$\mathcal{Y}$-*substitution* is any mapping $\theta : \mathcal{X} \to Term[\mathcal{Y}]$. Every such mapping can be represented as a set of bindings $\theta = \{x_1/\theta(x_1), \ldots x_n/\theta(x_n)\}$. We write $Subst[\mathcal{X}, \mathcal{Y}]$ for the set of all $\mathcal{X}$-$\mathcal{Y}$-substitutions. An *application* of a substitution θ to a term $t(x_1, \ldots, x_n)$ yields the term $t\theta = t(\theta(x_1), \ldots, \theta(x_n))$ obtained from t by replacing all occurrences of every variable $x_i,\ 1 \leq i \leq n$, with the term $\theta(x_i)$. A *composition* of a $\mathcal{X}$-$\mathcal{Y}$-substitution θ and a $\mathcal{Y}$-$\mathcal{Z}$-substitution η is a $\mathcal{X}$-$\mathcal{Z}$-substitution ξ such that the equality $x\xi = (x\theta)\eta$ (or, in other notation, $\xi(x) = (\theta(x))\eta$) holds for every $x,\ x \in \mathcal{X}$. To denote the composition of θ and η we use an expression $\theta\eta$; since $t(\theta\eta) = (t\theta)\eta$ holds for every term $t,\ t \in Term[\mathcal{X}]$, this notation makes it possible to skip parentheses when writing $t\theta\eta$ for the application of a composition of substitutions to a term. A $\mathcal{X}$-$\mathcal{X}$-substitution ρ is called a *renaming* iff ρ is a bijection on the set of variables $\mathcal{X}$. Two $\mathcal{X}$-$\mathcal{Z}$-substitutions θ_1 and θ_2 are *equivalent* if $\theta_1 = \theta_2\rho$ for some $\mathcal{X}$-$\mathcal{X}$-renaming ρ. If θ_1 is a composition of θ_2 and η then θ_1 is called an *instant* of θ_2, and θ_2 is called a *pattern* of θ_1.

Let θ_0 be a $\mathcal{X}$-$\mathcal{Y}$-substitution and θ_1 be a $\mathcal{Z}$-$\mathcal{U}$-substitution. Then a pair of substitutions η' and η'' from $Subst[\mathcal{Z}, \mathcal{X}]$ and $Subst[\mathcal{Y}, \mathcal{U}]$ respectively is called a *two-sided unifier* of (θ_0, θ_1) iff $\eta'\theta_0\eta'' = \theta_1$. Two-sided unification problem is that of finding, given a pair of substitutions (θ_0, θ_1), a two-sided unifier (η', η'') of (θ_0, θ_1). It must be noticed that two-sided unification, unlike usual unification, is asymmetric, since substitutions θ_0 and θ_1 play different roles in the equation $X\theta_0 Y = \theta_1$. Another important aspect of two-sided unification to be emphasized is that a substitution η'' does not affect directly the variables from $\mathcal{X}$ but only through the terms from θ_0 via the set of variables $\mathcal{Y}$. This is due to the software engineering application two-sided unification problem stems from: η'' only initializes input variables of θ_0 but does not interfere in the computation of θ_0.

Two-sided unification problem for a pair of substitutions (θ_0, θ_1) may have several solutions. For example, if $\theta_0 = \{x_1/f(y_1, y_2), x_2/y_3\}, \theta_1 = \{z/f(f(u, u), f(u, u))\}$ then two-sided unifiers of (θ_0, θ_1) are non-equivalent pairs $(\eta' = \{z/f(x_1, x_1)\}, \eta'' = \{y_1/u, y_2/u\})$, $(\eta' = \{z/f(f(x_2, x_2), x_1)\}, \eta'' = \{y_1/u, y_2/u\})$ and $(\eta' = \{z/x_1\}, \eta'' = \{y_1/f(u, u), y_2/f(u, u)\})$. Since the first component η' of every such pair is a pattern of θ_1 and the set of non-equivalent patterns of every substitution is finite, the set of non-equivalent two-sided unifiers of every pair of substitutions (θ_0, θ_1) is also finite.

When the complexity issues of decision problems for substitutions are concerned, the representation of terms in a set of bindings $\theta = \{x_1/t_1, \ldots x_n/t_n\}$ is of prime importance. We will assume that terms $t_1, \ldots, t_n$ in every substitution θ are represented by labeled trees. A representation of a composition $\theta\eta$ can be obtained from representations of θ and η just by attaching the terms from η to the corresponding leaves in the representations of terms from θ.

Lemma 1. *The problem of two-sided unifiability of pairs of substitutions represented by trees is in NP.*

Proof. It is easy to see that two-sided unifiability of (θ_0, θ_1) can be non-deterministically checked in polynomial time. It is sufficient

1. to guess a cut of tree representation T_{θ_1} of θ_1 into three pieces T_1, T_2, and T_3 in such a way that the leaves of every tree in T_1 and T_2 become the roots in the tree representations of T_2 and T_3 respectively.

2. to assign consistently variables from $\mathcal{X}$ and $\mathcal{Y}$ to all leaves of T_1 and T_2 respectively (the same variable can be assigned to different leaf nodes v_1, v_2 of a piece $T_i, i = 1, 2$, only if v_1 and v_2 are the roots of equal subtrees in T_{i+1}), and

3. to check that all trees from the middle piece T_2 represent only terms from θ_0.

Clearly, such cut of T_1, T_2, and T_3 of T_{θ_1} do exist iff $\theta_1 = \eta'\theta_0\eta''$ for some substitutions η' and η''. It is easy to see that the consistency of variable assignment and the inclusion of T_2 in T_{θ_0} can be checked in polynomial time. □

NP-hardness of two-sided unifiability problem follows from NP-completeness of BOUNDED TILING problem which is formally defined in the next section.

3 Bounded tiling problem

To define the bounded tiling problem imagine 1×1 square tiles whose edges are coloured. Suppose that only finitely many types of tiles are available. Consider a $n \times m$ rectangular area whose border is divided into segments of length 1 and assume that all such segments are also colored. The problem is to determine if it is possible to cover this area with the tiles (i.e. make a tiling) in such a manner that every pair of adjacent edges of two tiles has the same colour and every border segment has the same colour as the edge of a tile adjacent to it.

Formally BOUNDED TILING problem is specified as follows. Let $Colours = \{1, 2, \ldots, K\}$ be a finite set of *colours*. A *tile* is a quadruple $tile = \langle a_1, a_2, a_3, a_4\rangle$ of colours. The components of *tile* are denoted by $tile[0, -1], tile[-1, 0], tile[0, 1], tile[1, 0]$ respectively; they identify the colours of the top, right, bottom and left edges of the tile. A $n \times m$ *area* is the set of pairs $Area = \{(i, j) \ : \ 0 \le i \le n+1, 0 \le j \le m+1\}$; the elements of this set are called *squares*. The set of squares $Inter = \{(i, j) \ : \ 1 \le i \le n, 1 \le j \le m\}$ is the *interior* of the area. The *border* of the area is the set of squares $Border = Area \setminus Inter$. Two squares (i_1, j_1) and (i_2, j_2) in the *Area* are called *adjacent* iff $|i_1 - i_2| + |j_1 - j_2| = 1$. A *boundary constraint* is any mapping $B : Border \to Colours$. If $B(i, j) = a$ then this means that the "innermost" edge of a border square (i, j) is painted colour a. Let $Tiles = \{tile_1, \ldots, tile_L\}$ be a finite set of tiles. Then a *tiling* of an *Area* is any mapping $T : Inter \to Tiles$. Given a boundary constraint B, a tiling T is called *B-consistent* if the following two requirements are satisfied:

1. for every pair of adjacent interior squares (i_1, j_1) and (i_2, j_2) the equality $T(i_1, j_1)[i_1 - i_2, j_1 - j_2] = T(i_2, j_2)[i_2 - i_1, j_2 - j_1]$ holds; this equality means that the adjacent edges of the tiles inserted on these squares have the same colour;

2. for every interior square (i_1, j_1) which is adjacent to a border square (i_2, j_2) the equality $T(i_1, j_1)[i_1 - i_2, j_1 - j_2] = B(i_2, j_2)$ holds; this equality means that the colour of the border segment matches the colour of the adjacent edge of the tile.

An instance of the BOUNDED TILING problem is a tuple $BT = (n, m, Tiles, B)$; this instance is accepted iff there exists a B-consistent tiling of $n \times m$ $Area$ with tiles from $Tiles$. For the first time the TILING problem has been introduced in [10]. The complexity of this problem depends on the area to be tiled. Thus, in [11] it has been shown that if $Area$ is a quadrant of infinite plane then TILING problem is undecidable. In [3] it has been proved that BOUNDED TILING problem is NP-complete. We use this fact to prove NP-hardness of the two-sided unifiability problem.

4 NP-hardness of two-sided unification

Let $Colours = \{1, \ldots, K\}$. Consider an instance of the BOUNDED TILING problem $BT = (n, m, Tiles, B)$, where $Tiles = \{tile_1, \ldots, tile_L\}$. We show how to build such a pair of substitutions θ_0 and θ_1 that their two-sided unification (η', η''), if any, gives a solution to BT. The bindings of $\mathcal{X}$-$\mathcal{Y}$-substitution θ_0 represent the boundary constraint B and all possible insertions of tiles from $Tiles$ onto interior squares of the $Area$. The $\mathcal{Z}$-$\mathcal{U}$-substitution θ_1 represents the tiling of the same area with monochromatic tiles whose edges are painted colour K. The first component η' of a two-sided unifier specifies a choice of some possible tiling T of the $Area$, and the second component η'' checks the consistency of this tiling by simulating an attempt to "re-paint" consistently the edges of all tiles and border segments to achieve monochromatic tiling. The key feature of the substitution θ_0 is that the terms in its bindings share variables in such a manner that the colours of the adjacent edges of tiles can be changed only in common and by the same value. Therefore, a monochromatic "re-painting" η'' is possible only for B-consistent tilings.

To define θ_0 and θ_1 formally we introduce a set of functional symbols $\mathcal{F}$ which includes

- a binary function $g^{(2)}$ to build a $n \times m$ area,
- a 6-ary function $h^{(6)}$ to construct border constraints and instances of tiles,
- a unary function $f^{(1)}$ to enumerate colours and squares.

As for the sets of variables the substitutions θ_0 and θ_1 operate with, we assume that

- $\mathcal{X} = \mathcal{X}' \cup \mathcal{X}''$, where
  - $\mathcal{X}' = \{x'_{i,j} : (i,j) \in Border\}$: every variable $x'_{i,j}$ is associated with a border square (i,j) in the $Area$,
  - $\mathcal{X}'' = \{x''_{i,j,\ell} : (i,j) \in Interior, 1 \le \ell \le L\}$: every variable $x''_{i,j,\ell}$ is associated with an instant of a tile $tile_\ell$ inserted onto the square (i,j);
- $\mathcal{Y} = \{y_0\} \cup \mathcal{Y}'$, where
  - y_0 is a common "dummy" variable;
  - $\mathcal{Y}' = \{y_{i_1,j_1,i_2,j_2} : 0 \le i_1 \le i_2 \le n+1, 0 \le j_1 \le j_2 \le m+1, |i_1 - i_2| + |j_1 - j_2| = 1\}$: every variable y_{i_1,j_1,i_2,j_2} is associated with a pair of adjacent squares (i_1, j_1) and (i_2, j_2) in the $Area$;
- $\mathcal{Z} = \{z\}$, and $\mathcal{U} = \{u\}$.

By means of functional symbol $f^{(1)}$ we define recursively *numerals* $f_n(y)$ for every integer n as follows: $f_0(y) = y$ and $f_{n+1}(y) = f(f_n(y))$ for every n, $n \geq 0$. Numerals will be used to enumerate squares and colours. Clearly, $f_n(f_m(y)) = f_{n+m}(y)$ holds for every pair of integers n, m. We will say that a numeral $f_n(y)$ has a *rank* n.

The terms that represent the boundary constraint B and all possible insertions of individual tiles from $Tiles$ onto interior squares are defined as follows.

If a border square (i, j) is such that $(i, j) \in \{(0, 0), (n + 1, 0), (n + 1, m + 1), (0, m + 1)\}$ (i.e. (i, j) is a corner square of $Area$) then we assign the term

$$t_{i,j} = h(f_i(y_0), f_j(y_0), f_K(y_0), f_K(y_0), f_K(y_0), f_K(y_0))$$

to the variable $x_{i,j}$ associated with the square (i, j). This term indicates that all edges of this square are painted colour K.

Suppose that $(i, j) \in Border \setminus \{(0, 0), (n + 1, 0), (n + 1, m + 1), (0, m + 1)\}$ and $B(i, j) = k$. Then there exists the only interior square (i', j') which is adjacent to (i, j). Let y_{i_1,j_1,i_2,j_2} be the variable from $\mathcal{Y}'$ which is associated with the pair (i, j), (i', j'). Then we assign the term

$$t_{i,j} = h(f_i(y_0), f_j(y_0), f_K(y_0), f_K(y_0), f_K(y_0), f_k(y_{i_1,j_1,i_2,j_2}))$$

to the variable $x_{i,j}$ associated with the border square (i, j). This term indicates that the interior edge of this square (border segment) is painted colour $B(i, j)$, whereas all other edges are painted colour K.

Suppose that $(i, j) \in Interior$. Then there are exactly four squares in the $Area$ that are adjacent to the square (i, j) on the top, on the right, on the bottom, and on the left. Let $y_{i_1,j_1,i'_1,j'_1}, y_{i_2,j_2,i'_2,j'_2}, y_{i_3,j_3,i'_3,j'_3}$, and y_{i_4,j_4,i'_4,j'_4} be all those variables from $\mathcal{Y}'$ that are associated with these pairs of adjacent squares respectively. Then for every tile $tile_\ell = \langle k_1, k_2, k_3, k_4\rangle$, $1 \leq \ell \leq L$, from $Tiles$ we assign the term

$$t_{i,j,\ell} = h(f_i(y_0), f_j(y_0), f_{k_1}(y_{i_1,j_1,i'_1,j'_1}), f_{k_2}(y_{i_2,j_2,i'_2,j'_2}), f_{k_3}(y_{i_3,j_3,i'_3,j'_3}), f_{k_4}(y_{i_4,j_4,i'_4,j'_4}))$$

to the variable $x_{i,j,\ell}$ associated with the interior square (i, j) and the tile $tile_\ell$.

With terms $t_{i,j}$ and $t_{i,j,\ell}$ at hand, we define the substitution θ_0:

$$\theta_0 = \{x_{i,j}/t_{i,j} : (i, j) \in Border\} \cup \{x_{i,j,\ell}/t_{i,j,\ell} : (i, j) \in Interior, 1 \leq \ell \leq L\}.$$

It worth noticing that every variable from the set $\mathcal{Y}'$ occurred as an argument of numerals exactly in two terms from the range of substitution θ_0. We say that an occurrence of a variable y has a *depth* n iff n is the maximal rank of a numeral which includes this occurrence of y.

Using functional symbol $g^{(2)}$ we can build a (arbitrary) term t_{area} which has $(n+2)(m+2)$ argument positions (leaves in the tree representation of the term); every argument position in this term stands for a square in the $Area$. For every square (i, j) in the $Area$ we introduce the term $\widehat{t}_{i,j} = h(f_i(u), f_j(u), f_K(u), f_K(u), f_K(u), f_K(u))$ and define the substitution $\theta_1 = \{z/t_{area}(\widehat{t}_{0,0}, \widehat{t}_{0,1}, \ldots, \widehat{t}_{n+1,m+1})\}$ (monochromatic tiling of $Area$).

Lemma 2. *An instance of the BOUNDED TILING problem $BT = (n, m, Tiles, B)$ is acceptable iff the substitutions θ_0 and θ_1 defined above are two-sided unifiable.*

Proof. 1) Suppose that the instance BT is acceptable. Then there exists a B-consistent tiling T of $Area$ with the tiles from the set $Tiles$. For every pair of adjacent squares (i, j) and (i', j') in the interior of the area (assuming that $i \leq i'$, $j \leq j'$) denote by $c(i, j, i', j')$ the common colour of the adjacent edges of the tiles $T(i, j)$ and $T(i', j')$ installed onto these squares. The

same notation will be used for the common colour of a tile's edge and an adjacent segment of the boarder. By the definition of the terms $t_{i,j,\ell}$ both occurrences of a variable $y_{i,j,i',j'}$ in terms $t_{i,j,T(i,j)}$ and $t_{i',j',T(i',j')}$ have the same depth $c(i,j,i',j')$. Then a two-sided unification of (θ_0, θ_1) is a pair (η', η'') such that

$$\begin{array}{lcl} \eta' & = & \{z/t_{area}(x_{0,0}, x_{0,1}, \ldots, x_{0,m+1}, x_{1,0}, x_{1,1,T(1,1)}, \ldots, x_{1,m,T(1,m)}, x_{1,m+1}, \ldots, x_{n+1,m+1})\}, \\ \eta'' & = & \{y_0/u, y_{0,1,1,1}/f_{K-c(0,1,1,1)}(u), \ldots, y_{i,j,i',j'}/f_{K-c(i,j,i',j')}(u), \ldots\}. \end{array}$$

In substitution η' every argument of the term t_{area} corresponding to a square (i,j) is either a variable $x_{i,j}$ in the event that (i,j) is a boarder square, or a variable $x_{i,j,T(i,j)}$ in the event that (i,j) is an interior square. In the latter case the variable $x_{i,j,T(i,j)}$ indicates that a tile $tile_{T(i,j)}$ is placed onto the square (i,j). The substitution η'' assigns to every variable $y_{i,j,i',j'}$ associated with a pair of adjacent edges of two squares a numeral $f_{K-c(i,j,i',j')}(u)$ to complement the common colour $c(i,j,i',j')$ of the adjacent edges of the tiles $T(i,j)$ and $T(i',j')$ to the maximal colour K. By taking into account the fact that the tiling T is B-consistent we arrive at the conclusion that $\theta_1 = \eta'\theta_0\eta''$.

2) Suppose that $\theta_1 = \eta'\theta_0\eta''$ holds for a pair of substitutions (η', η''). Consider a sequence of functional symbols assigned to the nodes in an arbitrary branch in a tree representation of substitution θ_1. As it follows from the definition of θ_1, this sequence is $g, g, \ldots, g, h, f \ldots, f$. Moreover, for every square (i,j) the term in the range of θ_1 contains the only term of the form $h(f_i(y_0), f_j(y_0), \ldots)$. At the same time all terms in the range of θ_0 contain only functional symbols h and f. Thus, the substitution η' takes the form:

$$\eta' = \{z/t_{area}(x_{0,0}, x_{0,1}, \ldots, x_{0,m+1}, x_{1,0}, x_{1,1,\ell_{1,1}}, \ldots, x_{1,m,\ell_{1,m}}, x_{1,m+1}, , x_{n+1,m+1})\},$$

and η'' is a substitution of the form:

$$\eta'' = \{y_0/u, y_{0,1,1,1}/f_{k_{0,1,1,1}}(u), \ldots, y_{i,j,i',j'}/f_{k_{i,j,i',j'}}(u), \ldots\}.$$

Consider a tiling T such that $T(i,j) = \ell_{i,j}$ holds for every square (i,j) iff some term of the substitution η' includes a variable $x_{i,j,l_{i,j}}$. We show that this tiling is B-consistent.

Assume the contrary. Then there exists a pair of squares (i_1, j_1) and (i_2, j_2) in the *Area* such that either the adjacent edges of the tiles $T(i_1, j_1)$ and $T(i_2, j_2)$ inserted into these squares have different colours, or the adjacent edges of the tile $T(i_1, j_1)$ and the boarder square (i_2, j_2) are coloured differently. Without loss of generality we consider only the former case. Then the occurrences of the shared variable y_{i_1,j_1,i_2,j_2} in the terms $t_{i_1,j_1,\ell_{i_1 j_1}}$ and $t_{i_2,j_2,\ell_{i_2,j_2}}$ have different depths. Therefore, both occurrences of y_{i_1,j_1,i_2,j_2} in the terms of substitutions $\eta'\theta_0 = \{z/t_{area}(x_{0,0}, \ldots, x_{n+1,m+1})\theta_0\}$ also have different depths. Since η'' substitutes the same term instead of both occurrences of y_{i_1,j_1,i_2,j_2}, the numerals that indicate the colour of adjacent edges in the terms of composition $\eta'\theta_0\eta''$ have different ranks as well. In view of the fact that $\theta_1 = \eta'\theta_0\eta''$ the latter seems contrary to the definition of θ_1: all numerals that indicate the colour of edges must have the same rank K.

Thus, the tiling T defined above is B-consistent. □

Lemma 3. *BOUNDED TILING problem is log − space reducible to the problem of two-sided unifiability of first-order substitutions.*

Proof. Suppose that an instance of the BOUNDED TILING problem $BT = (n, m, Tiles, B)$ has a size N. As it can be seen from the definition of terms $t_{i,j}$, $t_{i,j,\ell}$, and $\widehat{t}_{i,j}$, a tree representation of every such term can be built by a deterministic procedure which operates on an auxiliary space of the size $O(\log N)$. Hence, tree representation of substitutions θ_0 and θ_1 as defined above can be also built within the same space. □

The main theorem follows from Lemmas 1 and 3.

Theorem 1. *Two-sided unifiability problem for first-order substitutions is NP-complete.*

5 Conclusion

This theorem completes the complexity picture in the study of solvability problem for equations of the form $X_1^{\sigma_1}\theta X_2^{\sigma_2} = X_1^{\sigma_3}\eta X_4^{\sigma_4}$ in the semigroup of first-order substitutions, where $\sigma_i \in \{0,1\}$, and X^σ is either X in the case of $\sigma = 1$, or empty substitution in the case of $\sigma = 0$. It is obvious that equations $X_1\theta X_2 = X_3\eta X_4$, $X_1\theta X_2 = \eta X_4$, and $X_1\theta X_2 = X_3\eta$ are trivially solvable for every pair of substitutions θ, η. Equations $\theta X_2 = \eta X_4$ and $\theta X_2 = \eta$ correspond to conventional unification problem; it is known that they are decidable in almost linear time (see [5, 6, 7]). Equations $X_1\theta = X_3\eta$ and $X_1\theta = \eta$ appeared in [12] with regard to equivalence checking problem in some class of sequential programs; they are decidable in polynomial time. Finally, in this paper we prove that only the solvability of equations of the form $X_1\theta X_2 = \eta$ (two-sided unification) is NP-complete problem.

References

[1] F. Baader, W. Snyder: Unification theory. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, 2001, v. 1, p. 447-533.

[2] K. Knight: Unification: a multidisciplinary survey. *ACM Computing Surveys*, 1989, v. 21, N 1, p. 93-124.

[3] C.H. Lewis: Complexity of solvable cases of the decision problem for predicate calculus. Proceedings of the 19-th Annual Symposium on Foundations of Computer Science, 1978, p. 35-47.

[4] D.C. Luckham, D.M. Park, M.S. Paterson: On formalized computer programs. *Journal of Computer and System Science*, 1970, v. 4, N 3, p. 220-249.

[5] Z. Manna, R. Waldinger: Deductive synthesis of the unification algorithm. *Science of Computer Programming*. 1981, v. 1, N 1-2, p. 5-48.

[6] A. Martelli, U. Montanari: An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 1982, v. 4, N 2, p. 258-282.

[7] M.S. Paterson, M.N. Wegman: Linear unification. *The Journal of Computer and System Science*, v. 16, N 2, 1978, p. 158–167.

[8] V.K. Sabelfeld: The logic-termal equivalence is polynomial-time decidable. *Information Processing Letters*, 1980, v. 10, N 2, p. 57-62.

[9] T.A. Novikova, V.A. Zakharov: Is it possible to unify programs?. The 27-th International Workshop on Unification, Epic Series, v. 19, 2013, p. 35-45.

[10] Wang Hao: Proving theorems by pattern recognition. *Bell System Technical Journal*. 1961, v. 40, N 1, p. 1-41.

[11] R. Berger: The undecidability of domino problem. *Memoirs of American Mathematical Society*, v. 66.

[12] V.A. Zakharov: On the decidability of the equivalence problem for orthogonal sequential programs. *Grammars*, v 2, N 3, p. 271-281.

Nominal Anti-Unification

Alexander Baumgartner[1], Temur Kutsia[1], Jordi Levy[2] and Mateu Villaret[3]

[1] Research Institute for Symbolic Computation, Johannes Kepler University Linz, Austria
[2] Artificial Intelligence Research Institute, Spanish Council for Scientific Research (IIIA-CSIC), Barcelona, Spain
[3] Departament d'Informàtica i Matemàtica Aplicada, Universitat de Girona, Spain

1 Introduction

Equation solving between nominal terms has been investigated by several authors, who designed and analyzed algorithms for nominal unification [3, 4, 15, 26], nominal matching [5], equivariant unification [6], permissive nominal unification [8]. However, in contrast to unification, its dual problem, anti-unification, has not been studied for nominal terms previously. In [18], it is referred to as "the as-of-yet undiscovered nominal anti-unification", which "could form a fundamental component of a refactoring tool" for αProlog [7] programs.

Software refactoring is one of possible applications of anti-unification. This method, formulated for different theories, has been successfully used in inductive logic programming [17], cognitive modeling [24], analogy making [13], inductive program synthesis [12], proof generalization [27], mathematical reasoning [10, 11], etc. Nominal anti-unification can play a role in extending some of these applications to the nominal setting. For instance, it can be useful to generalize proofs done in nominal logic, or in doing analogical reasoning in mathematics, or in adapting inductive program synthesis methods to αProlog programs, etc.

The anti-unification problem for two terms t_1 and t_2 is concerned with finding a generalization term t such that t_1 and t_2 are substitutive instances of t. The interesting generalizations are the least general ones (lgg). Plotkin [20] and Reynolds [22] initiated research on anti-unification in the 1970s, developing algorithms for first-order terms. Since then, anti-unification has been studied in various theories, including some of those with binding constructs: calculus of constructions [19], $M\lambda$ [9], second-order lambda calculus with type variables [16], simply-typed lambda calculus where generalizations are higher-order patterns [2], just to name a few.

In this paper we address the problem of computing lgg's for nominal *terms-in-context*, which are pairs of a freshness context and a nominal term. It turned out that without a restriction, there is no lgg for terms-in-context, in general. Therefore we restrict the set of atoms which are permitted in generalizations to be finite. In this case, there exists a single lgg (modulo $\simeq$) and we design an algorithm to compute it. Computation of nominal lgg's requires a solution to the equivariance problem which aims at finding a permutation of atoms π for given terms t_1 and t_2 such that π applied to t_1 is α-equivalent to t_2 (under a given freshness context).

Various anti-unification techniques, such as first-order, higher-order, or equational anti-unification have been used in inductive logic programming, logical and relational learning [21], reasoning by analogy [13], program synthesis [23], program verification [16], etc. Nominal anti-unification can, hopefully, contribute in solving similar problems in nominal setting.

The anti-unification algorithm has been implemented and is available from `www.risc.jku.at/projects/stout/software/nau.php`. The implementation of the equivariance algorithm is also accessible separately from `www.risc.jku.at/projects/stout/software/nequiv.php`.

2 Nominal Terms

Nominal terms contain *variables* and *atoms*. Variables can be instantiated and atoms can be bound. We have *sorts of atoms* ν and *sorts of data* δ as disjoint sets. *Atoms* $(a, b, \ldots)$ have one of the sorts of atoms. *Variables* $(X, Y, \ldots)$ have a sort of atom or data. Nominal function symbols $(f, g, \ldots)$ have an arity of the form $\tau_1 \times \cdots \times \tau_n \to \delta$, where δ is a sort of data and τ_i are sorts given by the grammar $\tau ::= \nu \mid \delta \mid \langle\nu\rangle\tau$. Abstractions have sorts of the form $\langle\nu\rangle\tau$.

A *swapping* $(a\,b)$ is a pair of atoms of the same sort. A *permutation* is a sequence of swappings. We use π, ρ to denote permutations. *Nominal terms* (t, s, r) are given by the grammar:

$$t ::= f(t_1, \ldots, t_n) \mid a \mid a.t \mid \pi{\cdot}X$$

The effect of swapping, and permutation application are defined in the standard way. The *inverse* of a permutation $\pi = (a_1\,b_1)\ldots(a_n\,b_n)$ is the permutation $(a_n\,b_n)\ldots(a_1\,b_1)$, denoted by π^{-1}. We use Id for the empty permutation and write X as the shortcut of $Id{\cdot}X$.

For a set A, we denote by $|A|$ its cardinality. The set of *atoms* of a term t or a permutation π is the set of all atoms which appear in it and is denoted by $\mathrm{Atoms}(t)$, $\mathrm{Atoms}(\pi)$ respectively. $\|t\|_{\mathsf{Abs}}$ stand for the number of abstraction occurrences in t.

Suspensions are uses of variables with a permutation of atoms waiting to be applied once a variable is instantiated. Occurrences of an atom a are said to be bound if they are in the scope of an abstraction of a, otherwise are said to be free. We denote by $\mathrm{FA}(t)$ the set of all atoms which occur freely in t: $\mathrm{FA}(f(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} \mathrm{FA}(t_i)$, $\mathrm{FA}(a) = \{a\}$, $\mathrm{FA}(a.t) = \mathrm{FA}(t) \setminus \{a\}$, and $\mathrm{FA}(\pi{\cdot}X) = \mathrm{Atoms}(\pi)$. $\mathrm{FA}^{\text{-s}}(t)$ is the set of all atoms which occur freely in t ignoring suspensions: $\mathrm{FA}^{\text{-s}}(f(t_1, \ldots, t_n))$, $\mathrm{FA}^{\text{-s}}(a)$, $\mathrm{FA}^{\text{-s}}(a.t)$ are defined like above but $\mathrm{FA}^{\text{-s}}(\pi{\cdot}X) = \emptyset$.

Substitutions, denoted by σ, are defined in the standard way, and their application allows atom capture, for instance, $a.X\{X \mapsto a\} = a.a$. The identity substitution is denoted by ε.

A *freshness constraint* is a pair of the form $a\#X$ stating that the instantiation of X cannot contain free occurrences of a. A *freshness context* is a finite set of freshness constraints. We will use ∇ and Γ for freshness contexts. $\mathrm{Atoms}(\nabla)$ denotes the set of atoms of ∇.

We say that a substitution σ *respects* ∇, if for all X, $\mathrm{FA}^{\text{-s}}(X\sigma) \cap \{a \mid a\#X \in \nabla\} = \emptyset$.

The predicate $\approx$ stands for α-equivalence and was defined in [25,26] by the following theory:

$$\frac{}{\nabla \vdash a \approx a} \qquad \frac{\nabla \vdash t \approx t'}{\nabla \vdash a.t \approx a.t'} \qquad \frac{a \neq a' \quad \nabla \vdash t \approx (a\,a'){\cdot}t' \quad \nabla \vdash a\#t'}{\nabla \vdash a.t \approx a'.t'}$$

$$\frac{a\#X \in \nabla \text{ for all } a \text{ such that } \pi{\cdot}a \neq \pi'{\cdot}a}{\nabla \vdash \pi{\cdot}X \approx \pi'{\cdot}X} \qquad \frac{\nabla \vdash t_1 \approx t'_1 \quad \cdots \quad \nabla \vdash t_n \approx t'_n}{\nabla \vdash f(t_1, \ldots t_n) \approx f(t'_1, \ldots, t'_n)}$$

where the freshness predicate $\#$ is defined by

$$\frac{a \neq a'}{\nabla \vdash a\#a'} \qquad \frac{(\pi^{-1}{\cdot}a\#X) \in \nabla}{\nabla \vdash a\#\pi{\cdot}X} \qquad \frac{\nabla \vdash a\#t_1 \quad \cdots \quad \nabla \vdash a\#t_n}{\nabla \vdash a\#f(t_1, \ldots t_n)} \qquad \frac{}{\nabla \vdash a\#a.t} \qquad \frac{a \neq a' \quad \nabla \vdash a\#t}{\nabla \vdash a\#a'.t}$$

Given a freshness context ∇ and a substitution σ, we define $\nabla\sigma$ as the *minimal* (with respect to $\subseteq$) freshness context such that for all $a\#X \in \nabla$ holds $\nabla\sigma \vdash a\#X\sigma$. It can easily be derived from the definition of the freshness predicate, if it exists. Otherwise it is undefined.

Theorem 1. *σ respects ∇ iff $\nabla\sigma$ is defined.*

A *term-in-context*, denoted by p, is a pair $\langle\nabla, t\rangle$ of a freshness context and a term. $\langle\nabla, t\rangle$ is *more general* than a term-in-context $\langle\Gamma, s\rangle$, written $\langle\nabla, t\rangle \preceq \langle\Gamma, s\rangle$, if there is a substitution σ, which respects ∇, such that $\nabla\sigma \subseteq \Gamma$ and $\Gamma \vdash t\sigma \approx s$. We write $\nabla \vdash t \preceq s$ if there exists a substitution σ such that $\nabla \vdash t\sigma \approx s$. We also write $\nabla \vdash t \simeq s$ iff $\nabla \vdash t \preceq s$ and $\nabla \vdash s \preceq t$.

Example 1. We give some examples to demonstrate the relations we have just defined:

- $\langle\{a\#X\}, f(a)\rangle \simeq \langle\emptyset, f(a)\rangle$. We can use $\{X \mapsto b\}$ as substitution applied to the first pair.
- $\langle\emptyset, f(X)\rangle \preceq \langle\{a\#Y\}, f(Y)\rangle$ with $\sigma = \{X \mapsto Y\}$, but not $\langle\{a\#Y\}, f(Y)\rangle \preceq \langle\emptyset, f(X)\rangle$.
- $\langle\{a\#X\}, f(X)\rangle \not\preceq \langle\{a\#X\}, f(a)\rangle$. Notice that $\sigma = \{X \mapsto a\}$ does not respect $\{a\#X\}$.
- $\langle\{b\#X\}, (a\,b)\cdot X\rangle \preceq \langle\{c\#X\}, (a\,c)\cdot X\rangle$ with the substitution $\sigma = \{X \mapsto (a\,b)(a\,c)\cdot X\}$.

A term-in-context $\langle\Gamma, r\rangle$ is called a *generalization* of two terms-in-context $\langle\nabla_1, t\rangle$ and $\langle\nabla_2, s\rangle$ if $\langle\Gamma, r\rangle \preceq \langle\nabla_1, t\rangle$ and $\langle\Gamma, r\rangle \preceq \langle\nabla_2, s\rangle$. It is the *least general generalization* (lgg) of $\langle\nabla_1, t\rangle$ and $\langle\nabla_2, s\rangle$ if there is no generalization $\langle\Gamma', r'\rangle$ of $\langle\nabla_1, t\rangle$ and $\langle\nabla_2, s\rangle$ which satisfies $\langle\Gamma, r\rangle \prec \langle\Gamma', r'\rangle$.

Note that if we have infinite number of atoms in the language, the relation $\prec$ is not well-founded: $\langle\emptyset, X\rangle \prec \langle\{a\#X\}, X\rangle \prec \langle\{a\#X, b\#X\}, X\rangle \prec \cdots$. As a consequence, two terms-in-context may not have an lgg and not even a minimal complete set of generalizations:[1]

Example 2. Let $p_1 = \langle\emptyset, a_1\rangle$ and $p_2 = \langle\emptyset, a_2\rangle$ be two terms-in-context. Then in any complete set of generalizations of p_1 and p_2 there is an infinite chain $\langle\emptyset, X\rangle \prec \langle\{a_3\#X\}, X\rangle \prec \langle\{a_3\#X, a_4\#X\}, X\rangle \prec \cdots$, where $\{a_1, a_2, a_3, \ldots\}$ is the set of all atoms of the language. Hence, p_1 and p_2 do not have a minimal complete set of generalizations.

Theorem 2. *The problem of anti-unification for terms-in-context is of nullary type.*

However, if we restrict the set of atoms which can be used in the generalizations to be finite, then the anti-unification problem becomes unitary.

We say that a term t (resp., a freshness context ∇) is *based* on a set of atoms A iff $\mathrm{Atoms}(t) \subseteq A$ (resp., $\mathrm{Atoms}(\nabla) \subseteq A$). A term-in-context $\langle\nabla, t\rangle$ is based on A if both t and ∇ are based on it. A permutation is A-based if it contains only atoms from A. An *A-based lgg* of A-based terms-in-context p_1 and p_2 is an A-based term-in-context p, which is a generalization of p_1 and p_2 and there is no A-based generalization p' of p_1 and p_2 which satisfies $p \prec p'$.

3 Nominal Anti-Unification Algorithm

Our anti-unification problem is parametric on the set of atoms we consider as the base, and finiteness of this set is essential to ensure the existence of an lgg. The problem we would like to solve is the following:

Given: Two nominal terms t and s of the same sort, a freshness context ∇, and a *finite* set of atoms A such that t, s, and ∇ are based on A.

Find: A term r and a freshness context Γ, such that the term-in-context $\langle\Gamma, r\rangle$ is an A-based least general generalization of the terms-in-context $\langle\nabla, t\rangle$ and $\langle\nabla, s\rangle$.

The triple $X : t \triangleq s$, where X, t, s have the same sort, is called the *anti-unification equation*, shortly AUE, and the variable X is called a *generalization variable*. We say that a set of AUEs P is based on a finite set of atoms A, if for all $X : t \triangleq s \in P$, the terms t and s are A-based.

The anti-unification algorithm is formulated in a rule-based way and depends on two global parameters, a finite set of atoms A and a freshness context ∇. It works on tuples of the form P; S; Γ; σ, where P and S are sets of AUEs, Γ is a freshness context and σ is a substitution. P, S, ∇, and Γ are A-based and ∇ does not constrain generalization variables. Furthermore if $X : t \triangleq s \in P \cup S$, then this is the sole occurrence of X in $P \cup S$. The rules are the following:

[1] The definition of minimal complete sets of generalizations is standard. For a precise definition, see, e.g. [1,14].

Dec: **Decomposition**

$\{X : \mathsf{h}(t_1, \ldots, t_m) \triangleq \mathsf{h}(s_1, \ldots, s_m)\} \uplus P;\ S;\ \Gamma;\ \sigma \Longrightarrow$
$\{Y_1 : t_1 \triangleq s_1, \ldots, Y_m : t_m \triangleq s_m\} \cup P;\ S;\ \Gamma;\ \sigma\{X \mapsto \mathsf{h}(Y_1, \ldots, Y_m)\}$,

where h is a function symbol or an atom, $Y_1, \ldots, Y_m$ are fresh variables of appropriate sorts.

Abs: **Abstraction**

$\{X : a.t \triangleq b.s\} \uplus P;\ S;\ \Gamma;\ \sigma \Longrightarrow \{Y : (c\ a)\cdot t \triangleq (c\ b)\cdot s\} \cup P;\ S;\ \Gamma;\ \sigma\{X \mapsto c.Y\}$,

where Y is fresh, $c \in A$ such that $\nabla \vdash c\#a.t$ and $\nabla \vdash c\#b.s$.

Sol: **Solving**

$\{X : t \triangleq s\} \uplus P;\ S;\ \Gamma;\ \sigma \Longrightarrow P;\ S \cup \{X : t \triangleq s\};\ \Gamma \cup \Gamma';\ \sigma$,

if neither Dec nor Abs is applicable, where $\Gamma' = \{a\#X \mid a \in A\ \wedge\ \nabla \vdash a\#t\ \wedge\ \nabla \vdash a\#s\}$.

Mer: **Merging**

$P;\ \{X : t_1 \triangleq s_1, Y : t_2 \triangleq s_2\} \uplus S;\ \Gamma;\ \sigma \Longrightarrow$
$P;\ \{X : t_1 \triangleq s_1\} \cup S;\ \Gamma\{Y \mapsto \pi\cdot X\};\ \sigma\{Y \mapsto \pi\cdot X\}$,

where π is an A-based permutation such that $\nabla \vdash \pi\cdot t_1 \approx t_2$, and $\nabla \vdash \pi\cdot s_1 \approx s_2$.

Given a finite set of atoms A, an A-based freshness context ∇, and two nominal A-based terms t and s, to compute an A-based generalization for $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$, we start with $\{X : t \triangleq s\}; \emptyset; \emptyset; \varepsilon$, where X is a fresh variable, and apply the rules (don't care) nondeterministically as long as possible. We denote this procedure by $\mathcal{N}$, and say that the *final state* is reached when no more rule is applicable. The final state is of the form $\emptyset; S; \Gamma; \sigma$, where Mer does not apply to S and we say that the *result computed* by $\mathcal{N}$ is $\langle \Gamma, X\sigma\rangle$.

Note that the Dec rule works also for the AUEs of the form $X : a \triangleq a$. In the Abs rule, it is important to have the corresponding c in A. If not then the Sol rule takes over.

Example 3. We illustrate $\mathcal{N}$ on a couple of examples:

- Let $t = f(a, b)$, $s = f(b, c)$, $\nabla = \emptyset$, and $A = \{a, b, c, d\}$. Then $\mathcal{N}$ performs the following transformations:

$$\{X : f(a,b) \triangleq f(b,c)\}; \emptyset; \emptyset; \varepsilon \Longrightarrow_{\mathsf{Dec}}$$
$$\{Y : a \triangleq b, Z : b \triangleq c\}; \emptyset; \emptyset; \{X \mapsto f(Y,Z)\} \Longrightarrow^{2}_{\mathsf{Sol}}$$
$$\emptyset; \{Y : a \triangleq b, Z : b \triangleq c\}; \{c\#Y, d\#Y, a\#Z, d\#Z\};\ \{X \mapsto f(Y,Z)\} \Longrightarrow_{\mathsf{Mer}}$$
$$\emptyset; \{Y : a \triangleq b\}; \{c\#Y, d\#Y\}; \{X \mapsto f(Y, (a\,b)(b\,c)\cdot Y)\}$$

  Hence, $p = \langle \{c\#Y, d\#Y\}, f(Y, (a\,b)(b\,c)\cdot Y)\rangle$ is the computed result. It generalizes the input pairs: $p\{Y \mapsto a\} \preceq \langle \nabla, t\rangle$ and $p\{Y \mapsto b\} \preceq \langle \nabla, s\rangle$. The substitutions $\{Y \mapsto a\}$ and $\{Y \mapsto b\}$ can be read from the final store. Note that $\langle \{c\#Y\}, f(Y, (a\,b)(b\,c)\cdot Y)\rangle$ would be also an A-based generalization of $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$, but it is strictly more general than p.
- Let $t = f(b, a)$, $s = f(Y, (a\,b)\cdot Y)$, $\nabla = \{b\#Y\}$, and $A = \{a, b\}$. Then $\mathcal{N}$ computes the term-in-context $\langle \emptyset, f(Z_1, (a\,b)\cdot Z_1)\rangle$ which generalizes the input pairs.
- Let $t = f(a.b, X)$, $s = f(b.a, Y)$, $\nabla = \{c\#X\}$, and $A = \{a, b, c, d\}$. Then $\mathcal{N}$ computes the term-in-context $p = \langle \{c\#Z_1, d\#Z_1\}, f(c.Z_1, Z_2)\rangle$ which generalizes the input: $p\{Z_1 \mapsto b, Z_2 \mapsto X\} = \langle \emptyset, f(c.b, X)\rangle \preceq \langle \nabla, t\rangle$ and $p\{Z_1 \mapsto a, Z_2 \mapsto Y\} = \langle \emptyset, f(c.a, Y)\rangle \preceq \langle \nabla, s\rangle$.

Theorem 3. *Let t, s be terms and ∇, Γ be freshness contexts, all based on a finite atoms set A.*

- Termination: *The procedure $\mathcal{N}$ terminates on any input.*

- Soundness: *If* $\{X : t \triangleq s\}; \emptyset; \emptyset; \varepsilon \Longrightarrow^+ \emptyset; S; \Gamma; \sigma$ *is a derivation obtained by an execution of* $\mathcal{N}$*, then* $\langle \Gamma, X\sigma\rangle$ *is an* A*-based generalization of* $\langle \nabla, t\rangle$ *and* $\langle \nabla, s\rangle$.
- Completeness: *If* $\langle \Gamma, r\rangle$ *is an* A*-based generalization of* $\langle \nabla, t\rangle$ *and* $\langle \nabla, s\rangle$*, then there exists a derivation* $\{X : t \triangleq s\}; \emptyset; \emptyset; \varepsilon \Longrightarrow^+ \emptyset; S; \Gamma'; \sigma$ *obtained by an execution of* $\mathcal{N}$*, such that* $\langle \Gamma, r\rangle \preceq \langle \Gamma', X\sigma\rangle$.
- Uniqueness: *Let* $\{X : t \triangleq s\}; \emptyset; \emptyset; \varepsilon \Longrightarrow^+ \emptyset; S_1; \Gamma_1; \sigma_1$ *and* $\{X : t \triangleq s\}; \emptyset; \emptyset; \varepsilon \Longrightarrow^+ \emptyset; S_2; \Gamma_2; \sigma_2$ *be two maximal derivations in* $\mathcal{N}$*. Then* $\langle \Gamma_1, X\sigma_1\rangle \simeq \langle \Gamma_2, X\sigma_2\rangle$.

Now we study how lgg's of terms-in-context depend on the set of atoms they are based on.

Lemma 1. *Let* A_1 *and* A_2 *be two finite sets of atoms with* $A_1 \subseteq A_2$ *such that the* A_1*-based terms-in-context* $\langle \nabla, t\rangle$ *and* $\langle \nabla, s\rangle$ *have an* A_1*-based lgg* $\langle \Gamma_1, r_1\rangle$ *and an* A_2*-based lgg* $\langle \Gamma_2, r_2\rangle$*. Then* $\Gamma_2 \vdash r_1 \preceq r_2$.

In general, we can not replace $\Gamma_2 \vdash r_1 \preceq r_2$ with $\Gamma_2 \vdash r_1 \simeq r_2$ in Lemma 1. Consider for instance the example $t = a.b$, $s = b.a$, $\nabla = \emptyset$, $A_1 = \{a, b\}$, and $A_2 = \{a, b, c\}$. Then for $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$, $\langle \emptyset, X\rangle$ is an A_1-based lgg and $\langle \{c\#X\}, c.X\rangle$ is an A_2-based lgg. Obviously, $\{c\#X\} \vdash X \preceq c.X$ but not $\{c\#X\} \vdash c.X \preceq X$.

We say that a set of atoms A is *saturated* for A-based t, s and ∇, if

$$|A \setminus (\mathrm{Atoms}(t) \cup \mathrm{Atoms}(s) \cup \mathrm{Atoms}(\nabla))| \geq \min\{\|t\|_{\mathsf{Abs}}, \|s\|_{\mathsf{Abs}}\}.$$

Lemma 2. *Under the conditions of Lemma 1, if* A_1 *is saturated for* t, s, ∇*, then* $\Gamma_2 \vdash r_1 \simeq r_2$.

4 Deciding Equivariance

Computation of π in the condition of the rule Mer above requires an algorithm that solves the following problem: Given a finite set of atoms A, terms t and s, and a freshness context ∇, all based on A, find an A-based permutation π such that $\nabla \vdash \pi{\cdot}t \approx s$. This is the problem of deciding whether t and s are equivariant with respect to ∇ and A.

We describe a rule-base algorithm, which we call $\mathcal{E}$, that solves this problem by effectively computing the corresponding permutation. It works on tuples of the form $E; \nabla; A; \pi$ (called systems). E is a set of equivariance equations of the form $t \approx s$ where t, s are nominal terms. ∇ is a freshness context, and A is a finite set of atoms which are available for computing π. The latter holds the permutation to be returned in case of success.

The algorithm is split into two phases. In phase 1, function applications, abstractions, and suspensions are decomposed as long as possible. Phase 2 is the permutation computation.

Phase 1 – Dec-E: **Decomposition**

$$\{f(t_1, \ldots, t_m) \approx f(s_1, \ldots, s_m)\} \cup E;\ \nabla;\ A;\ Id \Longrightarrow \{t_1 \approx s_1, \ldots, t_m \approx s_m\} \cup E;\ \nabla;\ A;\ Id.$$

Phase 1 – Alp-E: **Alpha Equivalence**

$$\{a.t \approx b.s\} \cup E;\ \nabla;\ A;\ Id \Longrightarrow \{(a\ c')\cdot t \approx (b\ c')\cdot s\} \cup E;\ \nabla;\ A;\ Id,$$

where c' is a fresh atom of the same sort as a and b.

Phase 1 – Sus-E: **Suspension**

$$\{\pi_1{\cdot}X \approx \pi_2{\cdot}X\} \cup E;\ \nabla;\ A;\ Id \Longrightarrow \{\pi_1{\cdot}a \approx \pi_2{\cdot}a \mid a \in A \wedge a\#X \notin \nabla\} \cup E;\ \nabla;\ A;\ Id.$$

Phase 2 – Rem-E: **Remove**

$$\{a \approx b\} \cup E;\ \nabla;\ A;\ \pi \Longrightarrow E;\ \nabla;\ A \setminus \{b\};\ \pi, \qquad \text{if } \pi{\cdot}a = b.$$

Phase 2 – Sol-E: **Solve**

$$\{a \approx b\} \cup E;\ \nabla;\ A;\ \pi \Longrightarrow E;\ \nabla;\ A \setminus \{b\};\ (\pi{\cdot}a\ b)\pi, \qquad \text{if } \pi{\cdot}a, b \in A \text{ and } \pi{\cdot}a \neq b.$$

The input for $\mathcal{E}$ is initialized in the Mer rule, which needs to compute an A-based permutation π for A-based context ∇ and two AUEs $X : t_1 \triangleq s_1$ and $Y : t_2 \triangleq s_2$. The system is initialized by $\{t_1 \approx t_2, s_1 \approx s_2\}$; ∇; A; Id. First we apply the rules of phase 1 exhaustively and afterwards Rem-E and Sol-E are applied as long as possible. If the final system is the *success state* $\emptyset$; ∇; A; π, then we say that $\mathcal{E}$ *computes* the permutation π. Otherwise it has the form E; ∇; A; π with $E \neq \emptyset$ to which no rule applies. It is transformed into $\bot$, called the *failure state*.

Example 4. We illustrate the algorithm $\mathcal{E}$ on examples and consider the equivariance problems:

- For $E = \{a \approx a,\ a.(a\,b)(c\,d)\cdot X \approx b.X\}$, $A = \{a, b, c, d\}$, and $\nabla = \{a\#X\}$, we derive
$$\{a \approx a, a.(a\,b)(c\,d)\cdot X \approx b.X\}; \{a\#X\};\ \{a, b, c, d\};\ Id \Longrightarrow_{\text{Alp-E}}$$
$$\{a \approx a, (a\,é)(a\,b)(c\,d)\cdot X \approx (b\,é)\cdot X\}; \{a\#X\};\ \{a, b, c, d\};\ Id \Longrightarrow_{\text{Sus-E}}$$
$$\{a \approx a, é \approx é, c \approx d, d \approx c\}; \{a\#X\};\ \{a, b, c, d\};\ Id \Longrightarrow^{2}_{\text{Rem-E}}$$
$$\{c \approx d, d \approx c\}; \{a\#X\};\ \{b, c, d\};\ Id \Longrightarrow^{\text{Sol-E}}_{\text{Rem-E}} \quad \emptyset;\ \{a\#X\};\ \{b\};\ (c\,d).$$
- For $E = \{a.b.(a\,b)(a\,c)\cdot X = b.a.(a\,c)\cdot X\}$, $A = \{a, b\}$, and $\nabla = \emptyset$, $\mathcal{E}$ returns Id.

Theorem 4. *Let t, s be terms and ∇ be a freshness context, all based on a finite set of atoms A.*

- Termination: *The procedure $\mathcal{E}$ terminates on any input.*
- Soundness: *Let $\{t \approx s\}$; ∇; A; $Id \Longrightarrow^* \emptyset$; ∇; B; π be a derivation in $\mathcal{E}$, then π is an A-based permutation such that $\nabla \vdash \pi\cdot t \approx s$.*
- Completeness: *If $\nabla \vdash \rho\cdot t \approx s$ for some A-based permutation ρ, then there is a derivation $\{t \approx s\}$; ∇; A; $Id \Longrightarrow^* \emptyset$; Γ; B; π, obtained by $\mathcal{E}$, such that $\pi\cdot a = \rho\cdot a$ for all $a \in \mathrm{FA}(t)$.*

Theorem 5. *The equivariance algorithm $\mathcal{E}$ has $O(n^2)$ space and time complexity and the anti-unification algorithm $\mathcal{N}$ has $O(n^4)$ time and $O(n^2)$ space complexity, where n is the input size.*

Acknowledgment

This research has been partially supported by the project HeLo (TIN2012-33042) and by the Austrian Science Fund (FWF) with the project SToUT (P 24087-N18).

References

[1] M. Alpuente, S. Escobar, J. Meseguer, and P. Ojeda. A modular equational generalization algorithm. In M. Hanus, editor, *LOPSTR*, volume 5438 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2008.

[2] A. Baumgartner, T. Kutsia, J. Levy, and M. Villaret. A variant of higher-order anti-unification. In F. van Raamsdonk, editor, *RTA*, volume 21 of *LIPIcs*, pages 113–127. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[3] C. Calvès. *Complexity and Implementation of Nominal Algorithms.* PhD thesis, Kings College London, 2010.

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

[5] C. Calvès and M. Fernández. Matching and alpha-equivalence check for nominal terms. *J. Comput. Syst. Sci.*, 76(5):283–301, 2010.

[6] J. Cheney. Equivariant unification. *JAR*, 45(3):267–300, 2010.

[7] J. Cheney and C. Urban. alpha-Prolog: A logic programming language with names, binding and a-equivalence. In B. Demoen and V. Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2004.

[8] G. Dowek, M. J. Gabbay, and D. P. Mulligan. Permissive nominal terms and their unification: an infinite, co-infinite approach to nominal techniques. *Logic Journal of the IGPL*, 18(6):769–822, 2010.

[9] C. Feng and S. Muggleton. Towards inductive generalization in higher order logic. In D. H. Sleeman and P. Edwards, editors, *ML*, pages 154–162. Morgan Kaufmann, 1992.

[10] M. Guhe, A. Pease, A. Smaill, M. Martínez, M. Schmidt, H. Gust, K.-U. Kühnberger, and U. Krumnack. A computational account of conceptual blending in basic mathematics. *Cognitive Systems Research*, 12(3-4):249–265, 2011.

[11] M. Guhe, A. Pease, A. Smaill, M. Schmidt, H. Gust, K.-U. Kühnberger, and U. Krumnack. Mathematical reasoning with higher-order anti-unifcation. In *Proceedings of the 32nd Annual Conference of the Cognitive Science Society*, pages 1992–1997, 2010.

[12] E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7:429–454, 2006.

[13] U. Krumnack, A. Schwering, H. Gust, and K.-U. Kühnberger. Restricted higher-order anti-unification for analogy making. In M. A. Orgun and J. Thornton, editors, *Australian Conference on Artificial Intelligence*, volume 4830 of *Lecture Notes in Computer Science*, pages 273–282. Springer, 2007.

[14] T. Kutsia, J. Levy, and M. Villaret. Anti-unification for unranked terms and hedges. In M. Schmidt-Schauß, editor, *RTA*, volume 10 of *LIPIcs*, pages 219–234. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[15] J. Levy and M. Villaret. Nominal unification from a higher-order perspective. *ACM Trans. Comput. Log.*, 13(2):10, 2012.

[16] J. Lu, J. Mylopoulos, M. Harao, and M. Hagiya. Higher order generalization and its application in program verification. *Ann. Math. Artif. Intell.*, 28(1-4):107–126, 2000.

[17] S. Muggleton. Inverse entailment and progol. *New Generation Comput.*, 13(3&4):245–286, 1995.

[18] D. Mulligan. *Extensions of Nominal Terms*. PhD thesis, School of Math. and Comp. Sci., Heriot-Watt University, Edinburgh, 2011.

[19] F. Pfenning. Unification and anti-unification in the calculus of constructions. In *LICS*, pages 74–85. IEEE Computer Society, 1991.

[20] G. D. Plotkin. A note on inductive generalization. *Machine Intel.*, 5(1):153–163, 1970.

[21] L. D. Raedt. *Logical and Relational Learning*. Springer, 2008.

[22] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intel.*, 5(1):135–151, 1970.

[23] U. Schmid. *Inductive Synthesis of Functional Programs, Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, volume 2654 of *Lecture Notes in Computer Science*. Springer, 2003.

[24] A. Schwering, U. Krumnack, K.-U. Kühnberger, and H. Gust. Syntactic principles of heuristic-driven theory projection. *Cognitive Systems Research*, 10(3):251–269, 2009.

[25] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. In M. Baaz and J. A. Makowsky, editors, *CSL*, volume 2803 of *Lecture Notes in Computer Science*, pages 513–527. Springer, 2003.

[26] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1–3):473–497, 2004.

[27] C. Walther and T. Kolbe. Proving theorems by reuse. *Artif. Intell.*, 116(1-2):17–66, 2000.

A Categorical Perspective on Pattern Unification (Extended Abstract)

Andrea Vezzosi and Andreas Abel

Department of Computer Science and Engineering
Chalmers University of Technology and Gothenburg University, Sweden
vezzosi@chalmers.se, andreas.abel@gu.se

Abstract

In 1991 Miller described a subset of the higher-order unification problem for the Simply Typed Lambda Calculus which admits most general unifiers, called the pattern fragment. This subset has been extended to more complex type theories and it is still used as the basis of modern unification algorithms in applications like proof search and type inference. Our contribution is a new presentation of the original unification algorithm that focuses on the abstract properties of the operations involved, using category theory as a structuring principle. These properties characterize a class of languages for which the algorithm can be reused.

1 Introduction

Pattern unification [Miller, 1991] is a restriction of higher-order unification where meta (unification) variables can only be applied to a list of distinct object (lambda bound) variables, called a *pattern*. This restriction is motivated by how such an unification problem with a meta variable at the head, $M\,x\,y\,z = t$, can essentially be read as a definition for the metavariable, $M := \lambda x\,y\,z.t$, as long as the resulting term is well-scoped.

The existence of most general unifiers guaranteed by the pattern restriction is important in applications like type inference for dependently typed languages or execution of higher order logic programs. In these cases, a common implementation strategy is to solve immediately those unification problems that fall into the pattern fragment and suspend the others, hoping that they will become tractable later when more meta variables have been solved [Reed, 2009, Abel and Pientka, 2011].

The basic intuition of our presentation, which is not new, is that patterns correspond to injective renamings and form a category; from there we go further and recognize how certain operations on patterns of the unification algorithm correspond to basic concepts of category theory like finite limits. The correctness of the resulting algorithm has been checked with a formalization [Vezzosi, 2012] using the proof assistant Agda. Category Theory has been used before to reason about first order unification by, for example, Goguen [Goguen, 1989].

2 The problem

We consider pattern unification in the Simply Typed Lambda Calculus (STLC) up to $\alpha\beta\eta$-equivalence. Without loss of generality, we can restrict our focus to terms in β-short η-long normal form by making use of the so-called spine formulation [Cervesato and Pfenning, 2003]. We use de Brujin [1972] indexes for object variables. As a consequence, unification is to be considered up to strict equality. Instead of a general application node $(t\,t)$ we have $(M\,p)$ and

$(x\,\vec{t})$ where the head is a meta or object variable. For application of meta variables $M\,p$ we represent the arguments by p, a list of distinct object variables, to ensure the pattern condition.

Terms	t	$::=$	λt	
		$\mid$	$x\,\vec{t}$	($x\,\vec{t}$ has base type ι)
		$\mid$	$M\,p$	($M\,p$ has base type ι)
Patterns	p	$::=$	$\vec{x}$	($x_i \neq x_j$ whenever $i \neq j$)
Object variables	x	$::=$	$0 \mid 1 + x$	

To match the way functions are always fully applied in terms, we define types in an uncurried style, i.e., as a (possibly empty) list of argument types and a base type ι for the result. We will simply write ι when there are no arguments.

Types	τ	$::=$	$\vec{\tau} \to \iota$	
Object contexts	Δ	$::=$	$\vec{\tau}$	
Meta contexts	Γ	$::=$	$. \mid M{:}\tau, \Gamma$	(Variables M in Γ are not repeated)

Typing contexts Δ for object variables are just lists of types. This identification between a typing context and argument types is exploited in the typing rules for metas and patterns. In the following, we list the rules for typing of variables $\Delta \vdash x : \tau$, patterns $\Delta \vdash p : \Delta'$, and normal terms $\Gamma; \Delta \vdash t : \tau$.

$$\frac{}{\tau, \Delta \vdash 0 : \tau} \qquad \frac{\Delta \vdash x : \tau}{\tau', \Delta \vdash 1 + x : \tau} \qquad \frac{}{\Delta \vdash . : .} \qquad \frac{\Delta \vdash x : \tau \qquad \Delta \vdash p : \Delta'}{\Delta \vdash (x, p) : (\tau, \Delta')}$$

$$\frac{\Gamma; \tau, \Delta \vdash t : \vec{\tau} \to \iota}{\Gamma; \Delta \vdash \lambda t : (\tau, \vec{\tau}) \to \iota} \qquad \frac{\Delta \vdash x : \Delta' \to \iota \qquad \Gamma; \Delta \vdash \vec{t} : \Delta'}{\Gamma; \Delta \vdash x\,\vec{t} : \iota} \qquad \frac{M : (\Delta' \to \iota) \in \Gamma \qquad \Delta \vdash p : \Delta'}{\Gamma; \Delta \vdash M\,p : \iota}$$

As hinted in the introduction we can think of $\Delta_2 \vdash p : \Delta_1$ as an injective renaming from variables in Δ_1 to variables in Δ_2, application to a variable $p\,x$ is performed by considering x from Δ_1 as an index to the position in p of the resulting variable in Δ_2. From this we form the category Pat with contexts as objects and patterns as morphisms, composition is given by $(p_1 \circ p_2)\,x = p_1\,(p_2\,x)$. We shall write $\Delta_2 \vdash p : \Delta_1$ as $p : \Delta_1 \to \Delta_2$.

We define *substitutions* σ as finite maps from meta variables to terms. *Update* $(\sigma, M := t)$ is defined as the substitution σ' such that $\sigma'\,M = t$ and $\sigma'\,N = \sigma\,N$ for $N \neq M$. Since we consider meta variables as living in a global scope, substitutions will produce terms without free object variables, hence they will be typed with an empty Δ. We write $\Gamma_2 \vdash \sigma : \Gamma_1$, or $\sigma : \Gamma_1 \to \Gamma_2$, iff $\sigma\,M = t$ for some $\Gamma_2; . \vdash t : \tau$ whenever $(M{:}\tau) \in \Gamma_1$.

Application of a substitution σ to a term, which we write $[\![\sigma]\!]\,t$, is done structurally as usual, except for nodes $(M\,p)$ where we need to normalize the generated beta-redex. Since p is merely a renaming, and our terms are η-long, normalizing amounts to stripping the outermost layer of λ abstractions from $(\sigma\,M)$ and applying p to their body. We do not need to apply σ to p because the latter does not contain meta variables.

$$[\![\sigma]\!]\,M\,p = [p]\,t \quad \text{where} \quad \sigma\,M = \vec{\lambda} t$$

In fact, from now on we will include the operation of stripping out the outer lambdas in the application of a substitution to a meta variable. Thus, given $\Gamma \vdash M : \Delta \to \iota$ we will have $\sigma\,M$ be a term t of type ι in the object context Δ, like the t in the equation above. This also allows us to express the *identity substitution* $\mathsf{id}_\Gamma : \Gamma \to \Gamma$ by simply $\mathsf{id}_\Gamma\,N = N\,\mathsf{id}_\Delta$ for $(N{:}\Delta \to \iota) \in \Gamma$. We write the *singleton substitution* $(\mathsf{id}, M := t)$ simply as $(M := t)$.

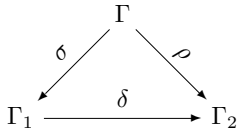
With the usual notion of composition we can also form the category Sub where meta variable contexts are the objects and substitutions are the morphisms.

Finally we observe that the set of terms with given type and contexts, $\mathsf{Tm}(\Gamma, \Delta, \tau)$, is functorial over both Sub and Pat, which is to say that application of renamings and substitutions commute with the respective compositions and identities, and between themselves. The category Type, is discrete, i.e., has types as objects but no morphisms except for identities.

$$\begin{array}{lcl} \mathsf{Tm} & : & \mathsf{Sub} \times \mathsf{Pat} \times \mathsf{Type} \to \mathsf{Set} \\ \mathsf{Tm}(\Gamma, \Delta, \tau) & = & \{t \in \mathrm{Terms} \mid \Gamma; \Delta \vdash t : \tau\} \end{array} \qquad \begin{array}{ll} [p_1 \circ p_2]\, t & = [p_1]\,[p_2]\, t \\ [\sigma_1 \circ \sigma_2]\, t & = [\![\sigma_1]\!]\,[\sigma_2]\, t \\ [\sigma]\,[p]\, t & = [p]\,[\sigma]\, t \end{array} \qquad \begin{array}{l} [\mathsf{id}_\Delta]\, t = t \\ [\![\mathsf{id}_\Gamma]\!]\, t = t \end{array}$$

Definition 1 (Unifier)**.** *A substitution $\sigma : \Gamma \to \Gamma_1$ is a* unifier *of two terms $t_1, t_2 \in \mathsf{Tm}(\Gamma, \Delta, \tau)$ whenever $[\![\sigma]\!]\, t_1 = [\![\sigma]\!]\, t_2$.*

Definition 2 (More General Substitution)**.** *A substitution $\sigma : \Gamma \to \Gamma_1$ is* more general *than $\rho : \Gamma \to \Gamma_2$, written $\sigma \leq \rho$, if there exists a substitution $\delta : \Gamma_1 \to \Gamma_2$ with $\rho = \delta \circ \sigma$.*



Definition 3 (Most General Unifier (MGU))**.** *A unifier $\sigma : \Gamma \to \Gamma_1$ of $t_1, t_2 \in \mathsf{Tm}(\Gamma, \Delta, \tau)$ is* most general *if $\sigma \leq \rho$ for every other unifier $\rho : \Gamma \to \Gamma_2$.*

3 Finding a solution

We will find the most general unifier of t_1 and t_2, or decide there cannot be one, by recursion on the terms themselves. In the following, we consider the possible cases.

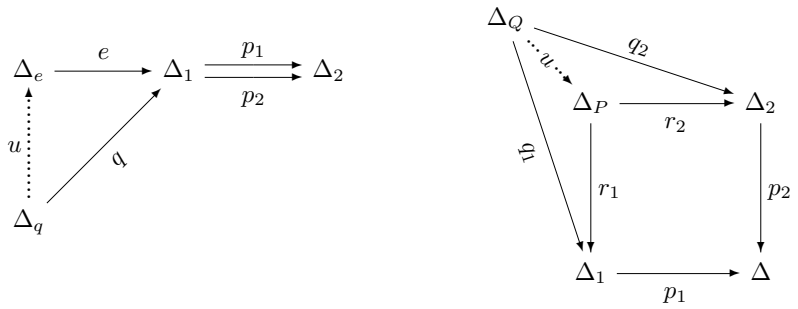
3.1 Rigid-Rigid

Since we have that $[\![\sigma]\!]\, \lambda t = \lambda([\![\sigma]\!]\, t)$, finding the unifier of λt_1 and λt_2 amounts to finding the one of t_1 and t_2. For variable applications, $x_1\, \vec{t_1}$ and $x_2\, \vec{t_2}$, it is almost the same, except that we need to check whether x_1 and x_2 are equal, and if so recurse over the subterms updating them with the unifier computed so far. In fact, we can abstract over both cases using a notion of operator $o ::= \lambda | x$ with decidable equality and arities, and such that $[\![\sigma]\!]\, (o\, \vec{t}) = o\, ([\![\sigma]\!]\, \vec{t})$.

3.2 Flex-Flex (Same Meta)

If the terms to unify are $M\, p_1$ and $M\, p_2$ things get more interesting. We can see that the most general unifier is $M := M'\, e$ where M' is a fresh meta variable and e is what is called the equalizer of p_1 and p_2. In fact, for σ to be an unifier it has to satisfy $[\![\sigma]\!]\, M\, p_1 = [\![\sigma]\!]\, M\, p_2$ which reduces to $M'\, (p_1 \circ e) = M'\, (p_2 \circ e)$ and the equalizer is the most general way to solve the equation $p_1 \circ e = p_2 \circ e$. What is meant by most general here is that for every other renaming q satisfying $p_1 \circ q = p_2 \circ q$ there is a unique u such that $q = e \circ u$. (See Figure 1.)

This property is all we need to show σ is most general, in fact, for any other unifier ρ we have $[p_1]\, (\rho\, M) = [p_2]\, (\rho\, M)$, and since the functor Tm preserves equalizers, i.e. $[e]$ is the equalizer of $[p_1]$ and $[p_2]$, we have a unique t such that $\rho\, M = [e]\, t$. That allows us to show $\sigma \leq \rho$ by $\delta := (\rho, M' := t)$. Now $(\delta \circ \sigma)\, M = [\![\delta]\!]\, M'\, e = [e]\, t = \rho\, M$ and $(\delta \circ \sigma)\, N = \delta\, N = \rho\, N$ for $N \neq M$.



Equalizer (Same Meta) Pullback (Different Meta)

Figure 1: Equalizer and Pullback diagrams

3.3 Flex-Flex (Different Meta)

As a stepping stone to the next case we consider the unification of the terms $M_1\, p_1$ and $M_2\, p_2$ for $M_1 \neq M_2$. In this case, the unifier is $\sigma = (M_1 := M'\, r_1, M_2 := M'\, r_2)$ for a fresh M' so that $[\![\sigma]\!]\, M_1\, p_1 = [\![\sigma]\!]\, M_2\, p_2$ reduces to $M_1\, (p_1 \circ r_1) = M_2\, (p_2 \circ r_2)$ and we can find r_1 and r_2 through another finite limit, the pullback of p_1 and p_2. (See Figure 1.)

3.4 Flex-Rigid

This is the main case to deal with, where we unify $M\, p$ with a t which does not contain M. Since $\sigma\, M$ will not be relevant for $[\![\sigma]\!]\, t$ we can decompose our candidate unifier σ into $(\pi, M := s)$ for a term s and another substitution π. The unification constraint $[\![\sigma]\!]\, M\, p = [\![\sigma]\!]\, t$ becomes $[p]\, s = [\![\pi]\!]\, t$. For this equation to be solvable the term $[\![\pi]\!]\, t$ may only have free object variables that appear in p. We distinguish free rigid variables, like x in $x\, \vec{t}$, from free pattern variables, like x in p. The substitution π can eliminate the free pattern variables which are not in p by so-called *pruning*.

Pruning. Pruning t with respect to p proceeds by recursion on t. In case $t = M'\, q$ we return the singleton substitution $\pi = (M' := M''\, r_2)$ for some fresh M'' and pattern r_2, so that $[\![\pi]\!]\, t = M''\, (q \circ r_2)$. The pattern $q \circ r_2$ must only contain free variables in p, which means there must exist a r_1 such that $p \circ r_1 = q \circ r_2$. The most general solution of this equation is again the pullback of p and q, and with an argument similar to the Flex-Flex case this leads to π being the most general pruning substitution. In case $t = \lambda t'$ we update the pattern p to handle the new bound variable, and recurse on t'. In case $t = x\, \vec{t}$ we recursively compute the pruning substitutions $\vec{\pi}$ for the subterms $\vec{t}$ and compose them by iteratively taking the pushout, i.e. the categorical dual of a pullback.

Inversion. Finding s such that $[p]\, s = [\![\pi]\!]\, t$, and thus uniquely solving the unification problem, is possible whenever the free rigid variables of t are contained in p. Specifically in the case $t = x\, \vec{t}$ the constraint reduces to $[p]\, s = x\, ([\![\pi]\!]\, \vec{t})$, this is solved by $s = y\, \vec{t}_1$ such that $p\, y = x$.

3.5 Failed Occurs Check

The remaining case is when we are tying to unify $M\,p$ with a term t where M appears in a nested position, e.g. $t = x\,(M\,q)$. Here we can conclude that there are no unifiers because the height of the two terms will never match.

4 Towards Generic Pattern Unification

Practical applications of higher-order unification need to handle languages more complex than STLC, e.g. with product and sum types, defined functions, and dependent types.

Our categorical view of the algorithm allows us to apply techniques from datatype generic programming to abstract away from the specific syntax and types of STLC. Instead of Pat we can consider a generic category Ctx having all the pullbacks and equalisers and whose arrows are monomorphisms. And instead of the grammar of STLC we can use an arbitrary one defined by a family of operators, as hinted in the Flex-Rigid section, as long as they are functorial with respect to Ctx, have decidable equality, and provide operations that attempt to invert the functorial action. From the functoriality of the operators we can derive the functoriality of the whole syntax. From a formalization point of view we would use an Indexed Container [Altenkirch and Morris, 2009] to describe the operators. It remains to be verified which language features of interest can fit into this generalization.

References

A. Abel and B. Pientka. Higher-order dynamic pattern unification for dependent types and records. In *Proc. of the 10th Int. Conf. on Typed Lambda Calculi and Applications, TLCA 2011*, volume 6690 of *Lect. Notes in Comput. Sci.*, pages 10–26. Springer, 2011.

T. Altenkirch and P. Morris. Indexed containers. In *Proc. of the 24nd IEEE Symp. on Logic in Computer Science (LICS 2009)*, pages 277–285. IEEE Computer Soc. Press, 2009.

I. Cervesato and F. Pfenning. A linear spine calculus. *J. Log. Comput.*, 13(5):639–688, 2003.

N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

J. A. Goguen. What is unification? – A categorical view of substitution, equation and solution. In *Resolution of Equations in Algebraic Structures, Volume 1: Algebraic Techniques*, pages 217–261. Academic, 1989.

D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991.

J. Reed. Higher-order constraint simplification in dependent type theory. In *4th Int. Wksh. on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP 2009)*, pages 49–56. ACM Press, 2009.

A. Vezzosi. Higher-order pattern unification in Agda, 2012. URL `http://github.com/Saizan/miller`.

Towards a better-behaved unification algorithm for Coq

Beta Ziliani[1] and Matthieu Sozeau[2]

[1] MPI-SWS (beta@mpi-sws.org)
[2] Inria Paris (matthieu.sozeau@inria.fr)

1 Introduction

The unification algorithm is at the heart of a proof assistant like Coq. In particular, it is a key component in the *refiner* (the algorithm that has to infer implicit terms and missing type annotations) and in the application of lemmas. In the first case, unification is in charge of equating the type of function arguments with the type of the elements to which the function is applied. In the second case, for instance when using the `apply` tactic, it is in charge of unifying the current goal with the conclusion of the lemma.

Despite playing a central role in proof development, there is no good source of documentation to understand Coq's unification algorithm. Since unification is inherently undecidable in Coq, as it must deal with higher-order problems up to conversion, some form of heuristic is desirable in order to solve problems that are trivial to the human eye. Otherwise, the proof developer will get easily frustrated when she finds two apparently equal terms not being unified. For instance, a desirable heuristic will equate the terms $?x +\!\!+ ?y \approx [] +\!\!+ (1 :: [])$, assigning $?x$ to the empty list and $?y$ to the singleton list $(1 :: [])$, where $?x$ and $?y$ are meta-variables and $+\!\!+$ is the list concatenation function. There exist other possible (convertible) solutions, like for instance assigning $(1 :: [])$ to $?x$ and $[]$ to $?y$, but in most of the cases preserving the structures of terms gives reasonable solutions.

The current approach in the source code of Coq includes this heuristic but also some harmful ones, like *Constraint Postponement*. In certain cases, when an equation has multiple solutions, it is delayed until more information is gather to solve the ambiguity. Constraint Postponement is commonly used (see *e.g.* [1]), and gives in practice reasonably good results. However, it also has its drawbacks. It may lead to extraneous error messages, since errors are reported at a later point in the unification process. More importantly, it does not mix well with other aspects of Coq's unification, like resolution of *Canonical Structures* [8]. Canonical Structures is an overloading mechanism similar to *type classes*, extensively used in the Mathematical Components library [4], on which the recent feat of proving the odd-order theorems [3] crucially relies. Supporting canonical structures resolution in unification makes the algorithm extremely sensitive to heuristics, since instance resolution depends heavily on the order in which unification problems are considered. Constraint Postponement also has an impact in performance, which became unpredictable as the unifier's complexity does not depend only on the size of the reduction paths of the two inputs.

In this talk we are going to present our work in progress on a new unification algorithm, built from scratch, which focuses on the following main properties:

Understandable: The algorithm can be described in full in a few pages, including canonical structures instance resolution.

Sound: The algorithm, when it succeeds, provides a well-typed substitution that equates both terms (up to conversion).

Predictable: The algorithm does not include heuristics that are hard to reason about.

In this paper, we will quickly introduce the language (§ 2) and delve into the delicate issues that come up in our setting, mainly due to unification up-to reduction, backtracking and dependencies (§ 3).

2 A primer on CIC with meta-variables

The *Calculus of Inductive Constructions* (CIC) is a dependently typed λ-calculus extended with inductive types. The terms of the language are defined as

$$\begin{aligned} t,T \mathrel{\widehat{=}} & \; x \mid c \mid i \mid k \mid s && x \in \mathscr{V}, c \in \mathscr{C}, i \in \mathscr{I}, k \in \mathscr{K}, s \in \mathscr{S} \\ & \mid \forall x : T.T \mid \lambda x : T.t \mid t\ t \mid \mathsf{let}\ x = t : T\ \mathsf{in}\ t \mid ?u[\sigma] && ?u \in \mathscr{M} \\ & \mid \mathsf{case}_T\ t\ \mathsf{of}\ k_1\ \overline{x} \Rightarrow t; \ldots; k_n\ \overline{x} \Rightarrow t\ \mathsf{end} \\ & \mid \mathsf{fix}_j\ \{x/n : T := t; \ldots; x/n : T := t\} \end{aligned}$$

where $\mathscr{V}$ is an enumerable set of variables, $\mathscr{M}$ of meta-variables, $\mathscr{C}$ of constants, $\mathscr{I}$ of inductive types, $\mathscr{K}$ of inductive constructors, and $\mathscr{S}$ is an infinite set of sorts defined as $\{\mathsf{Prop}, \mathsf{Type}(i) \mid i \in \mathbb{N}\}$.

Meta-variables are equipped with a substitution σ, which is nothing more than a list of terms.

In order to destruct an inductive type CIC provides a case constructor (match in vernacular) and a fixpoint. case is annotated with the return predicate T. In the term fix, the expression $x/n : T := t$ means that T is a type starting with at least n product types, and the n-th variable is the decreasing one in t. The subscript j of fix selects the j-th function as the main entry point.

The local context Γ, the meta-variables context Σ, and global environment E are defined as:

$$\begin{aligned} \Gamma, \Psi &\mathrel{\widehat{=}} \cdot \mid x : T, \Gamma \mid x := t : T, \Gamma \\ \Sigma &\mathrel{\widehat{=}} \cdot \mid ?u : T[\Psi], \Sigma \mid ?u := t : T[\Psi], \Sigma \\ E &\mathrel{\widehat{=}} \cdot \mid c : T, E \mid c := t : T, E \mid I, E \end{aligned}$$

Meta-variables have types T with all free variables bounded within a local context Ψ. In this work we borrow the notation $T[\Psi]$ from Contextual Modal Type Theory [6], while in [7] this is noted $\Psi \vdash T$.

Each possibly mutually recursive inductive type is stored in the environment E with the shape

$$\begin{aligned} I \mathrel{\widehat{=}} & \forall x_1 : T_1, \ldots, \forall x_h : T_h, \\ & \{\ \overline{i_m : A_m := \{k_1^m : C_1^m; \ldots; k_{n_1}^m : C_{n_1}^m\}}^m\} \end{aligned}$$

where every $i_m \in \mathscr{I}$, every $k_n^j \in \mathscr{K}$, every C_n^m has the shape $\forall \overline{x : T}, i_m\ t_1\ \ldots\ t_h$, and every A_m has the shape $\forall \overline{x : T}, s$. Inductive definitions are restricted to avoid circularity (each i_m can only appear strictly positive in every i_n depending on it). For the purpose of this work, we are not taking this restriction into consideration.

Reduction: Since the unification algorithm have to equate terms up to conversion, we need to present the reduction rules for CIC, listed in Figure 1. Besides the standard β rule, we have the ζ rule that expands let-definitions, three δ rules, that expand definitions from any of the contexts, and the ι rules for case destruction and fixpoint unfolding. The reduction rules depend on the contexts, which we assume as given. These rules rely on the standard multi-substitution of terms, noted $t\{\overline{t_n/x_n}\}$, which replaces each variable x_i with term t_i in term t.

3 Unification

The algorithm takes terms t_1 and t_2, a well-formed meta-variable context Σ and a well-formed local context Γ. We have a well-formed global environment E that is omnipresent. As precondition, the two terms should be well typed with types A_1 and A_2 respectively. Note that we do not require the types

$$
\begin{array}{rll}
(\lambda x : T.t)\ t' & \rightsquigarrow_\beta & t\{t'/x\} \\
\mathsf{let}\ x = t' : T\ \mathsf{in}\ t & \rightsquigarrow_\zeta & t\{t'/x\} \\
h & \rightsquigarrow_\delta & t \qquad \text{if } (h := t : T) \in E \text{ or } (h := t : T) \in \Gamma \\
?u[\sigma] & \rightsquigarrow_\delta & t\{\sigma/\Psi\} \qquad \text{if } (?u := t : T[\Psi]) \in \Sigma \\
\mathsf{case}_T\ (k_j\ \overline{a})\ \mathsf{of}\ \overline{k\ \overline{x} \Rightarrow t}\ \mathsf{end} & \rightsquigarrow_\iota & t_j\{\overline{a/x_j}\} \\
\mathsf{fix}_j\ \{F\}\ \overline{a} & \rightsquigarrow_\iota & t_j\{\overline{\mathsf{fix}_m\ \{F\}/x_m}\}\ \overline{a} \qquad F = \overline{x/n : T := t}
\end{array}
$$

Figure 1: Reduction rules in CIC.

to be equal. Upon success, the algorithm returns a new meta-variable context Σ' with instantiations for the meta-variables appearing in the terms or in Γ. The algorithm ensures that the terms t_1 and t_2 are convertible under this new meta-context. The unification judgment is noted as $\Sigma;\Gamma \vdash t_1 \approx t_2 \triangleright \Sigma'$.

In this abstract we will not present the unification rules, which the interested reader is invited to read from the accompanying appendix. Instead, we will focus on three points that make the design of the algorithm delicate: (1) type dependencies, (2) conversion, and (3) canonical structures resolution. In the following sections we explain these and motivate the need for a change in the current algorithm.

3.1 Type dependencies

Sometimes the unification algorithm is faced with an equation that has not one but many solutions, in a context where there should only be one possible candidate. For instance, consider the following term witnessing an existential quantification:

$$\mathsf{exist}\ \_\ 0\ (\mathsf{le\_n}\ 0) : \exists x.x \leq x$$

where exist is the constructor of the type $\exists x.P\ x$, with P a predicate over the (implicit) type of x. More precisely, exist takes a predicate P, an element x, and a proof that P holds for x, that is, $P\ x$. In the example above we are providing an underscore in place of P, since we want Coq to find out the predicate, and we annotate the term with a typing constraint (after the colon) to specify that we want the whole term to be a proof that there exists a number that is lesser or equal to itself. In this case, we provide 0 as such number, and the proof le_n 0 which has type $0 \leq 0$.

When typechecking the term, Coq first considers the term and then it checks that it is compatible to the typing constraint. More precisely, Coq will create a fresh meta-variable for the predicate P, let's call it $?P$, and unify $?P\ 0$ with $0 \leq 0$. Without any further information, Coq has four different (incomparable) options for P: $\lambda x.0 \leq 0, \lambda x.x \leq 0, \lambda x.0 \leq x, \lambda x.x \leq x$.

When faced with such an ambiguity, Coq delays the equation in the hope that further information will help disambiguate the problem. In this case, that information was given through the typing constraint, and Coq succeeds to typecheck the term. If there were no typing constraint, Coq would have picked an arbitrary solution. There are two direct consequences of these design decisions. On one hand, the algorithm is more "complete", in the sense that less typing annotations are required (in this case, we do not need to specify P). On the other hand, the arbitrary solution selected by Coq may not be the one expected by the proof developer. In the example above, for instance, when we remove the typing constraint Coq will decide that the term has type $\exists x.0 \leq x$. If, at a different point in the proof script, this term is used to prove $\exists x.x \leq x$, the proof developer will have to debug the proof script to find out where the problem originated from.

Moreover, performance-wise, constraint postponement can be disastrous as it might postpone unsolvable constraints and make failure exponentially slower, e.g. due to first-order unification (see 3.2) being applied to other unification problems before finally failing.

For all these reasons we decided to remove postponement from the algorithm. Actually, in most cases it is possible to achieve the same level of "completeness" without constraint postponement, by using *bidirectional typechecking*, that is, to use the typing information available (*e.g.*, from the typing constraint) to infer meta-variables in the term. Then, when typechecking exist _ 0 le_n_0 *under* the typing constraint $\exists x.0 \leq 0$, we can propagate a *unique* solution for P from the type to the term. Removing postponement also helps to get a simpler proof of type soundness for the unification algorithm, which we plan to mechanize.

3.2 First-order approximation

The algorithm includes a so-called first-order unification rule:

$$\frac{\Sigma_0; \Gamma \vdash u \approx u' \triangleright \Sigma_1}{\Sigma_0; \Gamma \vdash t\ u \approx t\ u' \triangleright \Sigma_1}\ \textsc{App-FO}$$

This rule applies when two applications are unified (here in a simplified binary application version), even if the head t might be unfoldable (i.e., a definition in some context). This rule clearly precludes the generation of most general unifiers, as u and u' do not need to be unified if, for example, t is $\lambda x.0$. However, it is very natural to add it as it can shorten many unifications that would in the end result in the unification of the arguments of t. So we must bear with it if we are to be compatible with the existing algorithm and keep in sync with its performance. Of course, a drawback of this rule is that we must *backtrack* on its application if the premise cannot be derived, and try instead to perform a step of reduction, like unfolding the head constant.

This behavior is problematic, especially in presence of fixpoint definitions which might generate repeated, almost identical applications of this rule which ultimately fail, when only the normal forms of the fixpoint applications can unify. We are experimenting with ideas to keep track of successes (and failures) of unifications to avoid an exponential blowup due to that behavior.

3.3 Canonical Structures resolution

A *structure* is a particular inductive type: it has only one constructor, and it generates one projector for each argument of the constructor. The syntax is

$$\mathsf{structure}\ i\ \overline{a} : s := k\ \{p_1 : A_1; \ldots; p_n : A_n\}$$

where $\overline{a}$ is a list of *arguments* of the type, of the form $x_1 : T_1, \ldots, x_m : T_m$. Each p_j is a *projector* name. This language construct generates an inductive type

$$\{\ i : \forall \overline{a}.s := \{\ k : \forall \overline{a}.\forall \overline{p : A}.i\ \overline{a}\ \}\ \}$$

and for each projector *name* p_j it generates a projector *function*:

$$\lambda \overline{a}.\lambda z.\mathsf{case}\ s\ \mathsf{with}\ k\ x_1\ \ldots\ x_j\ \ldots\ x_n \Rightarrow x_j\ \mathsf{end}\ :\ \forall \overline{a}.\forall z : i\ \overline{a}.A_j$$

An instance ι of the structure is created with the constructor k:

$$\iota := \forall \overline{x : B}.k\ t_1\ \ldots\ t_{m+n}$$

where m is the number of arguments of the structure. Terms t_1 to t_m corresponds to the arguments of the structure, and t_{m+1} to t_{m+n} to each of the values p_j.

The important aspect of structures is that their instances can be deemed as "canonical". A canonical instance instructs the unification algorithm to instantiate a structure meta-variable with the instance, if certain conditions holds. More precisely, a canonical instance populates the canonical instance database Δ_{db} with triples (p_j, h_j, ι), where h_j is the head constant appearing in value t_{m+j}. (h_j can also be an implication ($\rightarrow$) or a sort.) Then, whenever the unification algorithm have to solve a problem of the form $p_j\ \overline{a}\ ?s \approx h_j\ \overline{b}$, it instantiates $?s$ with ι. There cannot be two triples with the same projector and head constant. Coq enforces this invariant by shadowing previous triples with new overlapping triples.

An immediate consequence of using the head constant to determine the instance is that δ-expanding a term may expose a different constant, and therefore a different instance. In [5, 2], for instance, this fact is used to resolve overlapping instances. Similarly, an earlier δ-expansion may prevent the use of an instance, so δ-expansion is delayed as much as possible.

Another relevant aspect, as we mentioned in the introduction, is that constraint postponement in the presence of Canonical Structures resolution may lead to unexpected results.

4 Conclusion

We have presented the main features and design choices of our algorithm. It has been implemented and is being successfully tested on the Canonical Structures resolution part of the mathematical components library, which relies on all the expressive power of the unifier. With our collaborators, we are also working on a proof of soundness, mechanized in Coq, to provide a solid ground on which to build complex tactics. As a mid-term goal, we also plan to aggressively optimize the algorithm, making sure its semantics remains the same. Altogether, our work will give Coq users a fast, completely verified and documented unification algorithm.

References

[1] A. Abel and B. Pientka. Higher-order dynamic pattern unification for dependent types and records. In *TLCA*, pages 10–26, 2011.

[2] Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. Canonical big operators. In *TPHOLs*, volume 5170 of *LNCS*, pages 86–101, 2008.

[3] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *ITP 2013*, volume 7998 of *LNCS*. Springer, 2013.

[4] G. Gonthier, A. Mahboubi, and E. Tassi. A small scale reflection extension for the Coq system. Technical report, INRIA, 2008.

[5] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. *Journal of Functional Programming*, 23:357–401, 7 2013.

[6] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9:23:1–23:49, June 2008.

[7] B. Pientka and F. Pfenning. Optimizing higher-order pattern unification. In *19th International Conference on Automated Deduction*, pages 473–487. Springer-Verlag, 2003.

[8] A. Saïbi. Typing algorithm in type theory with inheritance. In *Proc of POPL'97*, pages 292–301, 1997.

5 Appendix: Unification rules

Given the complexity of the algorithm, we split the rules in different figures. Figures 2 and 3 show the subset of rules involving the CIC constructs without meta-variables; Figure 4 considers meta-variable instantiation; and Figure 8 considers canonical structures resolution.

The first three rules (PROP-SAME, TYPE-SAME and TYPE-SAME-LE) unifies two types, according to the restriction imposed on universe levels. For abstractions (LAM-SAME) and products (PROD-SAME) we first unify the types of the arguments and then the body of the binder, with the local context extended with the bound variable.

When unifying two lets the rule LET-SAME compares the definitions and then the body, augmenting the context with the definition. Note that we don't need to check the types of the definitions, since if the definitions are unifiable then their type is unifiable as well. If the rule fails to apply, then the rule LET-ZETA unfolds the definitions in both sides and tries again.

RIGID-SAME equates the same variable, constant, inductive type, or constructor.

The following two rules consider the rules for matching cases and fixpoints. In both cases we just unify pointwise every component of the constructors (case and fix respectively).

The last rule of Figure 2 considers two applications with the same number of arguments (n). It first compares the head element (t and t') and then proceeds to unify each of the arguments.

When the rules in Figure 2 fails to apply, then the algorithm tries to do one step reduction and try again, in the hope to find a solution. This process is described in Figure 3. The rules are pretty easy to read, and are labeled according to the reduction step they take.

One point should be made: when one of the terms is a case or a fix the algorithm tries weak-head reducing the term. We denote the weak head reduction of t under contexts Σ and Γ as $\Sigma;\Gamma \vdash t \downarrow^{\mathrm{w}}_{\beta\zeta\delta\iota} t'$. As a sanity check, we make sure that progress was made, by comparing the result of the weak head reduction with the original term.

Meta-variable instantiation is considered in figure 4. We proceed to describe each rule according to the case.

Same meta-variable: If both terms are the same meta-variable $?u$, we have two distinct cases: if their substitution is the exact same list of variables ξ, the rule META-SAME-SAME applies, in which the arguments of the meta-variable are compared point-wise. If, instead, their substitution is different, then the rule META-SAME is attempted. To better understand this rule, let's look at an example. Say $?u$ has type $T[x_1 : \mathsf{nat}, x_2 : \mathsf{nat}]$ and we have to solve the equation

$$?u[y_1, y_2] \approx ?u[y_1, y_3]$$

where y_1, y_2 and y_3 are defined in the local context. From this equation we cannot know yet what value $?u$ will hold, but at least we know it cannot refer to the second parameter, x_2, since that will render the equation above false. This reasoning is reflected in the rule META-SAME in the hypothesis

$$\Psi_1 \vdash \xi \cap \xi' \triangleright \Psi_2$$

This judgment performs an intersection of both substitutions, filtering out those positions from the context of the meta-variable Ψ_1 where the substitution disagree, resulting in Ψ_2. This judgment is defined in Figure 5.

Once we filter out the disagreeing positions of the substitution we need to create a new meta-variable $?v$ with same type of $?u$, but in the shorter context Ψ_2. We further instantiate $?u$ with $?v$. Both the creation of $?v$ and the instantiation of $?u$ in the context Σ is expressed in the fragment $\Sigma \cup \{?v : T[\Psi_2], ?u := ?v[\mathsf{id}_{\Psi_2}]\}$ of the last hypothesis. We use this new context to compare point-wise the arguments of the meta-variable.

$$\frac{}{\Sigma;\Gamma \vdash \mathsf{Prop} \approx \mathsf{Prop} \triangleright \Sigma}\ \textsc{Prop-Same}$$

$$\frac{i = j}{\Sigma;\Gamma \vdash \mathsf{Type}(i) \approx \mathsf{Type}(j) \triangleright \Sigma}\ \textsc{Type-Same}$$

$$\frac{i \leq j}{\Sigma;\Gamma \vdash \mathsf{Type}(i) \lesssim \mathsf{Type}(j) \triangleright \Sigma}\ \textsc{Type-Same-LE}$$

$$\frac{\Sigma;\Gamma \vdash A_1 \approx A_2 \triangleright \Sigma' \qquad \Sigma';\Gamma, x : A_1 \vdash t_1 \approx t_2 \triangleright \Sigma''}{\Sigma;\Gamma \vdash \lambda x : A_1.t_1 \approx \lambda x : A_2.t_2 \triangleright \Sigma''}\ \textsc{Lam-Same}$$

$$\frac{\Sigma;\Gamma \vdash A_1 \approx A_2 \triangleright \Sigma' \qquad \Sigma';\Gamma, x : A_1 \vdash B_1 \approx B_2 \triangleright \Sigma''}{\Sigma;\Gamma \vdash \forall x : A_1.B_1 \approx \forall x : A_2.B_2 \triangleright \Sigma''}\ \textsc{Prod-Same}$$

$$\frac{\Sigma;\Gamma \vdash t_2 \approx t_2' \triangleright \Sigma' \qquad \Sigma';\Gamma, x := t_2 \vdash t_1 \approx t_1' \triangleright \Sigma''}{\Sigma;\Gamma \vdash \mathsf{let}\ x = t_2 : T\ \mathsf{in}\ t_1 \approx \mathsf{let}\ x = t_2' : T'\ \mathsf{in}\ t_1' \triangleright \Sigma''}\ \textsc{Let-Same}$$

$$\frac{\Sigma;\Gamma \vdash t_1\{t_2/x\} \approx t_1'\{t_2'/x\} \triangleright \Sigma'}{\Sigma;\Gamma \vdash \mathsf{let}\ x = t_2 : T\ \mathsf{in}\ t_1 \approx \mathsf{let}\ x = t_2' : T'\ \mathsf{in}\ t_1' \triangleright \Sigma'}\ \textsc{Let-Zeta}$$

$$\frac{h \in \mathscr{V} \cup \mathscr{C} \cup \mathscr{I} \cup \mathscr{K}}{\Sigma;\Gamma \vdash h \approx h \triangleright \Sigma}\ \textsc{Rigid-Same}$$

$$\frac{\Sigma_0;\Gamma \vdash T \approx T' \triangleright \Sigma_1 \qquad \Sigma_1;\Gamma \vdash t \approx t' \triangleright \Sigma_2 \qquad \Sigma_2;\Gamma \vdash \overline{b} \approx \overline{b'} \triangleright \Sigma_3}{\Sigma_0;\Gamma \vdash \mathsf{case}_T\ t\ \mathsf{of}\ \overline{b}\ \mathsf{end} \approx \mathsf{case}_{T'}\ t'\ \mathsf{of}\ \overline{b'}\ \mathsf{end} \triangleright \Sigma_3}\ \textsc{Case-Same}$$

$$\frac{\Sigma_0;\Gamma \vdash \overline{T} \approx \overline{T'} \triangleright \Sigma_1 \qquad \Sigma_1;\Gamma \vdash \overline{t} \approx \overline{t'} \triangleright \Sigma_2}{\Sigma_0;\Gamma \vdash \mathsf{fix}_j\ \{\overline{x/n : T := t}\} \approx \mathsf{fix}_j\ \{\overline{x'/n' : T' := t'}\} \triangleright \Sigma_2}\ \textsc{Fix-Same}$$

$$\frac{\Sigma_0;\Gamma \vdash t \approx t' \triangleright \Sigma_1 \qquad \Sigma_1;\Gamma \vdash \overline{t_n} \approx \overline{t_n'} \triangleright \Sigma_2}{\Sigma_0;\Gamma \vdash t\ \overline{t_n} \approx t'\ \overline{t_n'} \triangleright \Sigma_2}\ \textsc{App-FO}$$

Figure 2: Unification algorithm: pure CIC constructs (part 1).

$$\frac{\Sigma;\Gamma \vdash t' \approx t\{t_1/x\}\ t_2\ \ldots\ t_n \triangleright \Sigma'}{\Sigma;\Gamma \vdash t' \approx (\lambda x : A.t)\ t_1\ \ldots\ t_n \triangleright \Sigma'}\ \textsc{Lam-BetaR}$$

$$\frac{\Sigma;\Gamma \vdash t\{t_1/x\}\ t_2\ \ldots\ t_n \approx t' \triangleright \Sigma'}{\Sigma;\Gamma \vdash (\lambda x : A.t)\ t_1\ \ldots\ t_n \approx t' \triangleright \Sigma'}\ \textsc{Lam-BetaL}$$

$$\frac{\Sigma;\Gamma \vdash t' \approx t_1\{t_2/x\}\ \overline{t_n} \triangleright \Sigma'}{\Sigma;\Gamma \vdash t' \approx (\mathsf{let}\ x = t_2 : T\ \mathsf{in}\ t_1)\ \overline{t_n} \triangleright \Sigma'}\ \textsc{Let-ZetaR}$$

$$\frac{\Sigma;\Gamma \vdash t_1\{t_2/x\}\ \overline{t_n} \approx t' \triangleright \Sigma'}{\Sigma;\Gamma \vdash (\mathsf{let}\ x = t_2 : T\ \mathsf{in}\ t_1)\ \overline{t_n} \approx t' \triangleright \Sigma'}\ \textsc{Let-ZetaL}$$

$$\frac{(x := t : A) \in \Gamma \qquad \Sigma;\Gamma \vdash t' \approx t\ \overline{t_n} \triangleright \Sigma'}{\Sigma;\Gamma \vdash t' \approx x\ \overline{t_n} \triangleright \Sigma'}\ \textsc{Rigid-Delta-VarR}$$

$$\frac{(x := t : A) \in \Gamma \qquad \Sigma;\Gamma \vdash t\ \overline{t_n} \approx t' \triangleright \Sigma'}{\Sigma;\Gamma \vdash x\ \overline{t_n} \approx t' \triangleright \Sigma'}\ \textsc{Rigid-Delta-VarL}$$

$$\frac{t' \text{ is fix or case} \qquad \Sigma;\Gamma \vdash t' \downarrow^{\mathrm{w}}_{\beta\zeta\delta\iota} t'' \qquad t' \neq t'' \qquad \Sigma;\Gamma \vdash t \approx t'' \triangleright \Sigma'}{\Sigma;\Gamma \vdash t \approx t' \triangleright \Sigma'}\ \textsc{WhdR}$$

$$\frac{t \text{ is fix or case} \qquad \Sigma;\Gamma \vdash t \downarrow^{\mathrm{w}}_{\beta\zeta\delta\iota} t'' \qquad t \neq t'' \qquad \Sigma;\Gamma \vdash t'' \approx t' \triangleright \Sigma'}{\Sigma;\Gamma \vdash t \approx t' \triangleright \Sigma'}\ \textsc{WhdL}$$

$$\frac{(c := t : A) \in E \qquad \Sigma;\Gamma \vdash t' \approx t\ \overline{t_n} \triangleright \Sigma'}{\Sigma;\Gamma \vdash t' \approx c\ \overline{t_n} \triangleright \Sigma'}\ \textsc{Rigid-Delta-ConsR}$$

$$\frac{(c := t : A) \in E \qquad \Sigma;\Gamma \vdash t\ \overline{t_n} \approx t' \triangleright \Sigma'}{\Sigma;\Gamma \vdash c\ \overline{t_n} \approx t' \triangleright \Sigma'}\ \textsc{Rigid-Delta-ConsL}$$

Figure 3: Unification algorithm: pure CIC constructs (part 2).

There are two other hypotheses that ensure that nothing goes wrong. Again, we explain them by means of example. The hypothesis

$$\mathsf{FV}(T) \subseteq \Psi_2$$

ensures that the type T is well formed in the new (shorter) context Ψ_2. This condition might sound redundant at first sight, and, in fact, it is not present in [1]. However, it is necessary since, unlike in [1], we don't have as premise that the type of both terms are the same. As example, consider the contexts

$$\Sigma = \{?u : x[x : \mathsf{Type}]\} \qquad \Gamma = \{y : \mathsf{Type}, z : \mathsf{Type}\}$$

and the equation

$$u[y] \approx u[z]$$

The intersection of both substitutions will return an empty context. But we cannot create a new meta-variable $?v$ with type x in the empty context! The problem comes from the fact that both terms have different types (y and z respectively). By ensuring that every free variable in the type of the meta-variable is in the context Ψ_2 we prevent this issue.

More subtle is the inclusion of the premise

$$\Sigma \vdash \Psi_2$$

Because of convertibility, it may happen that the two substitutions agree on a value whose type depends on a previous value not equal in both substitutions. As example, consider contexts

$$\Sigma = \{?v : \mathsf{Prop}[x : \mathsf{Type}, p : \mathsf{fst}(\mathsf{Prop}, x)]\} \qquad \Gamma = \{y : \mathsf{Type}, z : \mathsf{Type}, w : \mathsf{Prop}\}$$

and the equation

$$?v[y, w] \approx ?v[z, w]$$

After performing the intersection, we get the ill-formed context $[p : \mathsf{fst}(\mathsf{Prop}, x)]$.

Meta-variable instantiation: The rules META-INSTL (R) are in charge of instantiating a meta-variable. On the left (right) hand side it has meta-variable $?u$ applied to the (variable to variable) substitution ξ and with (only variables) arguments ξ'. On the right (left) hand side it has some term t. Assuming $?u$ has (contextual) type $T[\Psi]$, this rule must find a term t' such that $t'\{\xi/\hat{\Psi}\}\ \xi'$ is convertible to t, where $\hat{\cdot}$ is defined as

$$x_1 : T_1, \hat{\ldots}, x_n : T_n \mathrel{\widehat{=}} x_1, \ldots, x_n$$

In order to obtain t' the following steps are followed:

1. The meta-variables in t are *pruned*, as we are going to explain in the next section.
2. t' is constructed as a function taking arguments $\bar{x}$, one for each variable in ξ. The body of this function is the *inversion* of substitution $\xi, \xi'/\hat{\Psi}, \bar{x}$. More precisely, every variable in t appearing only once in the image of the substitution (ξ, ξ') is replaced by the corresponding variable in the domain of the substitution $(\hat{\Psi}, \bar{x})$. If a variable appears multiple times in the image and occur in term t, then inversion fails.
3. The type of t' is unified with the type of $?u$. We do this in order to ensure soundness of unification. Since we do not contemplate the types of the terms being unified, we need to obtain the type of t' in order to compare it with T. This introduces a penalty in the performance of the algorithm, but since we know t' is well typed (the unification algorithm requires both terms to be well typed, and the inversion process preserves the type), then we can perform a fast *retyping* of t'.
4. The term t' is *occur checked* to not contain meta-variable $?u$.

5.1 Pruning

The idea behind pruning can be understood with an example. Say we want to unify terms

$$?w[x, y] \approx c\ ?u[z, ?v[y]] \tag{1}$$

A solution exists, although z is a free variable in the rhs not appearing in the image of the substitution of the lhs. The solution has to restrict $?u$ so it does not depends on the first substitution. This can be done by meta-substituting $?u$ with a new $?u'$ with a smaller context. That is, if $?u : T[a : T_1, b : T_2]$, then

$$\frac{\Sigma;\Gamma \vdash \bar{t} \approx \overline{t'} \triangleright \Sigma'}{\Sigma;\Gamma \vdash ?u[\xi]\ \bar{t} \approx ?u[\xi]\ \overline{t'} \triangleright \Sigma'}\ \textsc{Meta-Same-Same}$$

$$\frac{\Psi_1 \vdash \xi \cap \xi' \triangleright \Psi_2 \qquad \Sigma \vdash \Psi_2 \qquad \mathsf{FV}(T) \subseteq \Psi_2 \qquad ?u : T[\Psi_1] \in \Sigma \qquad \Sigma \cup \{?v : T[\Psi_2], ?u := ?v[\mathsf{id}_{\Psi_2}]\};\Gamma \vdash \bar{t} \approx \overline{t'} \triangleright \Sigma'}{\Sigma;\Gamma \vdash ?u[\xi]\ \bar{t} \approx ?u[\xi']\ \overline{t'} \triangleright \Sigma'}\ \textsc{Meta-Same}$$

$$\frac{?u : T[\Psi] \in \Sigma_0 \qquad \Sigma_0 \vdash \mathsf{prune}(\xi, \xi'; t) \triangleright \Sigma_1 \qquad t' = \lambda \bar{x}.t\{\xi, \xi'/\hat{\Psi}, \bar{x}\}^{-1} \qquad \Sigma_1;\Psi \vdash t' : T' \qquad \Sigma_1;\Psi \vdash T' \lesssim T \triangleright \Sigma_2 \qquad ?u \notin t'}{\Sigma_0;\Gamma \vdash ?u[\xi]\ \xi' \approx t \triangleright \Sigma_2 \cup \{?u := t'\}}\ \textsc{Meta-InstL}$$

$$\frac{?u : T[\Psi] \in \Sigma_0 \qquad n \leq m \qquad n > 0 \qquad \Sigma_1;\Gamma \vdash ?u[\sigma] \approx t'\ \overline{t'_{m-n}} \triangleright \Sigma_2 \qquad \Sigma_0;\Gamma \vdash \overline{t_n} \approx \overline{t'_{m-n+1..m}} \triangleright \Sigma_1}{\Sigma_0;\Gamma \vdash ?u[\sigma]\ \overline{t_n} \approx t'\ \overline{t'_m} \triangleright \Sigma_2}\ \textsc{Meta-FOL}$$

Figure 4: Meta-variable instantiation.

$$\frac{}{\cdot \vdash \cdot \cap \cdot \triangleright \cdot}$$

$$\frac{\Psi \vdash \xi \cap \xi' \triangleright \Psi'}{\Psi, x : A \vdash \xi, y \cap \xi', y \triangleright \Psi', x : A}$$

$$\frac{\Psi \vdash \xi \cap \xi' \triangleright \Psi'}{\Psi, x := t : A \vdash \xi, y \cap \xi', y \triangleright \Psi', x := t : A}$$

$$\frac{\Psi \vdash \xi \cap \xi' \triangleright \Psi' \qquad z \neq y}{\Psi, x : A \vdash \xi, y \cap \xi', z \triangleright \Psi'}$$

$$\frac{\Psi \vdash \xi \cap \xi' \triangleright \Psi' \qquad z \neq y}{\Psi, x := t : A \vdash \xi, y \cap \xi', z \triangleright \Psi'}$$

Figure 5: Intersection of substitutions

a fresh unification variable $?u'$ is created with type $T[b : T_2]$, and $?u := ?u'[b]$. The result of this process in Equation 1 is

$$?w[x,y] \approx c\ ?u'[?v[y]]$$

which can now be easily solved. Instead, if z occurs inside the substitution of $?v$,

$$?w[x,y] \approx c\ ?u[x,?v[z]]$$

then it is not clear anymore, since a solution may exists by pruning z from $?v$, or by pruning $?v[z]$ from $?u$.

It is important to note that we can only prune offending variables that appear in the head of the term. Consider the following example:

$$?u[x] \approx c\ ?v[(x,y)] \tag{2}$$

One is tempted to prune the argument of $?v$, however this will prevent the unification algorithm from picking the following solution

$$?v[p] := \mathsf{fst}\ p$$

instantiating further $?u$ with the (convertible) term x. As example, considering the following problem:

$$\mathsf{let}\ p := (x,y)\ \mathsf{in}\ (?u[x], ?v[p]) \approx \mathsf{let}\ p := (x,y)\ \mathsf{in}\ (c\ ?v[(x,y)], \mathsf{fst}\ p)$$

After unifying the definition of the let, it introduces the definition $p := (x,y)$ in the local context and proceeds to pairwise unify the components of the pair. By unifying the first component we obtain Equation 2. If we (incorrectly) prune the argument from $?v$, this step succeeds instantiating $?u$ with $c\ ?v'[]$. The second component will try to unify (after expanding the new definition for $?v$)

$$?v'[] \approx \mathsf{fst}\ p \tag{3}$$

failing to unify. In this example it is easy to see where things went wrong, but in general it's a bad idea to fail at the wrong place, as the developer has to trace the algorithm to find that, actually, the problem was in another place.

Figure 6 shows the rules for pruning. Given a meta-context Σ, a list of variables ξ and a term t, the pruning of meta-variables in t is denoted

$$\Sigma \vdash \mathsf{prune}(\xi; t) \triangleright \Sigma'$$

where Σ' is a new meta-context extending Σ by instantiating the pruned meta-variables with new meta-variables, as we saw in the example above.

5.2 Canonical structures resolution

The scariest rules of this work are clearly the ones about canonical structures resolution, listed in Figure 8. But looking at them closely we can see they are not as scary as they look. The first rule CS-CONSTL shows the most common case of CS resolution. In this rule, on the left hand side we have projector p_j applied to the structure c, with structure parameters $\overline{a}$ and arguments $\overline{t}$. On the right hand side we have constant h applied to arguments $\overline{u}$ and $\overline{t'}$. That is, the j-th component of c should be a function taking arguments $\overline{t}$. In order to solve the equation we need an instance ι in the database relating p_j and h. This instance should be a function taking some arguments $\overline{x : B}$ and returning the application of the constructor of the structure k to parameters $\overline{a'}$, and with field values $\overline{v}$. The j-th value should have head constant h, applied to arguments $\overline{u'}$. The algorithm should find the right instantiation for the arguments of the instance. For this, it creates new meta-variables $?y$, one for each argument of ι, and

$$\frac{h \in \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}\} \cup \mathscr{C}}{\Sigma \vdash \mathsf{prune}(\xi; h) \triangleright \Sigma} \textsc{Prune-Constant}$$

$$\frac{x \in \xi}{\Sigma \vdash \mathsf{prune}(\xi; x) \triangleright \Sigma} \textsc{Prune-Var}$$

$$\frac{\Sigma \vdash \mathsf{prune}(\xi, x; t) \triangleright \Sigma'}{\Sigma \vdash \mathsf{prune}(\xi; \lambda x.t) \triangleright \Sigma'} \textsc{Prune-Lam}$$

$$\frac{\Sigma \vdash \mathsf{prune}(\xi, x; t) \triangleright \Sigma'}{\Sigma \vdash \mathsf{prune}(\xi; \forall x.t) \triangleright \Sigma'} \textsc{Prune-Prod}$$

$$\frac{\Sigma_0 \vdash \mathsf{prune}(\xi; t) \triangleright \Sigma_1 \qquad \Sigma_i \vdash \mathsf{prune}(\xi; t_i) \triangleright \Sigma_{i+1} \qquad i \in [1, n]}{\Sigma_0 \vdash \mathsf{prune}(\xi; t\ \overline{t_n}) \triangleright \Sigma_{i+1}} \textsc{Prune-App}$$

$$\frac{\Sigma_1 \vdash \mathsf{prune}(\xi; t_2) \triangleright \Sigma_2 \qquad \Sigma_2 \vdash \mathsf{prune}(\xi, x; t_1) \triangleright \Sigma_3}{\Sigma_1 \vdash \mathsf{prune}(\xi; \mathsf{let}\ x = t_2\ \mathsf{in}\ t_1) \triangleright \Sigma_3} \textsc{Prune-Let}$$

$$\frac{\Psi \vdash \mathsf{prune\_ctx}(\xi; \sigma) \triangleright \Psi}{\Sigma, u : A[\Psi], \Sigma' \vdash \mathsf{prune}(\xi; ?u[\sigma]) \triangleright \Sigma, u : A[\Psi], \Sigma'} \textsc{Prune-Meta-NoPrune}$$

$$\frac{u : A[\Psi] \in \Sigma \qquad \Psi \vdash \mathsf{prune\_ctx}(\xi; \sigma) \triangleright \Psi' \qquad \mathsf{FV}(A) \subseteq \Psi'}{\Sigma \vdash \mathsf{prune}(\xi; ?u[\sigma]) \triangleright \Sigma, ?v : A[\Psi'] \cup \{u := v[\mathsf{id}_{\Psi'}]\}} \textsc{Prune-Meta}$$

Figure 6: Pruning of meta-variables.

$$\frac{}{\cdot \vdash \mathsf{prune\_ctx}(\xi; \cdot) \triangleright \cdot} \textsc{PruneCtx-Nil}$$

$$\frac{\mathsf{FV}(t) \in \xi \qquad \Psi \vdash \mathsf{prune\_ctx}(\xi; \sigma) \triangleright \Psi'}{\Psi, x : A \vdash \mathsf{prune\_ctx}(\xi; \sigma, t) \triangleright \Psi', x : A} \textsc{PruneCtx-NoPrune}$$

$$\frac{y \notin \xi \qquad \Psi \vdash \mathsf{prune\_ctx}(\xi; \sigma) \triangleright \Psi'}{\Psi, x : A \vdash \mathsf{prune\_ctx}(\xi; \sigma, y\ \overline{t_n}) \triangleright \Psi'} \textsc{PruneCtx-Prune}$$

Figure 7: Pruning of contexts.

$$\frac{(p_j,h,\iota)\in\Delta_{\text{db}} \quad \iota := \lambda\overline{x:B}.k\ \overline{a'}\ \overline{v} \quad v_j = h\ \overline{u'} \quad \Sigma_1 = \Sigma_0, \overline{?y:B} \quad \Sigma_1;\Gamma\vdash \overline{a}\approx\overline{a'\{\overline{?y}/\overline{x}\}}\triangleright\Sigma_2 \quad \Sigma_2;\Gamma\vdash\overline{u}\approx\overline{u'\{\overline{?y}/\overline{x}\}}\triangleright\Sigma_3 \quad \Sigma_3;\Gamma\vdash c\approx\iota\ \overline{?y}\triangleright\Sigma_4 \quad \Sigma_4;\Gamma\vdash\overline{t}\approx\overline{t'}\triangleright\Sigma_5}{\Sigma_0;\Gamma\vdash p_j\ \overline{a}\ c\ \overline{t}\approx h\ \overline{u}\ \overline{t'}\triangleright\Sigma_5}\ \text{CS-ConstL}$$

$$\frac{\begin{array}{c}(p_j,\to,\iota)\in\Delta_{\text{db}}\\ \iota := \lambda\overline{x:B}.k\ \overline{a'}\ \overline{v} \quad v_j = u\to u' \quad \Sigma_1 = \Sigma_0, \overline{?y:B} \quad \Sigma_1;\Gamma\vdash \overline{a}\approx\overline{a'\{\overline{?y}/\overline{x}\}}\triangleright\Sigma_2\\ \Sigma_2;\Gamma\vdash t\approx u\{\overline{?y}/\overline{x}\}\triangleright\Sigma_3 \quad \Sigma_3;\Gamma\vdash t'\approx u'\{\overline{?y}/\overline{x}\}\triangleright\Sigma_4 \quad \Sigma_4;\Gamma\vdash c\approx\iota\ \overline{?y}\triangleright\Sigma_5\end{array}}{\Sigma_0;\Gamma\vdash p_j\ \overline{a}\ c\approx t\to t'\triangleright\Sigma_5}\ \text{CS-ProdL}$$

$$\frac{(p_j,s,\iota)\in\Delta_{\text{db}} \quad \iota := \lambda\overline{x:B}.k\ \overline{a'}\ \overline{v} \quad \Sigma_1 = \Sigma_0, \overline{?y:B} \quad \Sigma_1;\Gamma\vdash \overline{a}\approx\overline{a'\{\overline{?y}/\overline{x}\}}\triangleright\Sigma_2 \quad \Sigma_2;\Gamma\vdash c\approx\iota\ \overline{?y}\triangleright\Sigma_3}{\Sigma_0;\Gamma\vdash p_j\ \overline{a}\ c\approx s\triangleright\Sigma_3}\ \text{CS-SortL}$$

$$\frac{\begin{array}{c}(p_j,\_,\iota)\in\Delta_{\text{db}} \quad \iota := \lambda\overline{x:B}.k\ \overline{a'}\ \overline{v} \quad v_j = x_{j'} \quad \Sigma_1 = \Sigma_0, \overline{?y:B}\\ \Sigma_1;\Gamma\vdash \overline{a}\approx\overline{a'\{\overline{?y}/\overline{x}\}}\triangleright\Sigma_2 \quad \Sigma_2;\Gamma\vdash x_{j'}\{\overline{?y}/\overline{x}\}\approx t\triangleright\Sigma_3 \quad \Sigma_3;\Gamma\vdash c\approx\iota\ \overline{?y}\triangleright\Sigma_4\end{array}}{\Sigma_0;\Gamma\vdash p_j\ \overline{a}\ c\approx t\triangleright\Sigma_3}\ \text{CS-DefaultL}$$

Figure 8: Canonical structures resolution.

proceeds to unify the parameters of the projector with the parameters of the instance. Then, it unifies the arguments of the constant h encountered in the rhs with the ones in the field value. Is it after this point that it equates the structures with the instance. Finally, it unifies the arguments of the function defined by h on both sides of the equation.

The rule CS-PRODL considers the case when the value is a function type. It is similar to the previous one, except that the projector cannot have arguments. The same situation we have in rule CS-SORTL, where the right hand side is a sort (Prop or Type). The last rule CS-DEFAULTL considers the *default* instance, when the value of the j-th field of the instance is a variable.

For conciseness we have omitted the rules for when the projector is in the right hand side.

5.3 Algorithm

The rules shown does not precisely nail the way backtracking is handled, nor the priority of the rules.

First, the algorithm distinguishes three cases:

1. Any of the terms has a meta-variable in the head position. We have three subcases, where every attempt to use rules META-INSTL or META-INSTR is followed by an attempt to use rules META-FOL and META-FOR, respectively.
   (a) Both terms have the same meta-variable in the head position. Try rules META-SAME and META-SAME-SAME.
   (b) Both terms have different meta-variables. Try first META-INSTL and then META-INSTR if the variable on the left is the oldest one, or viceversa if it's the newest one.
   (c) Any other case: try META-INSTL and META-INSTR.

2. If both terms have no arguments, try the rules in Figure 2, except of course APP-FO. Special case if both are lets: first try LET-SAME and if that fails LET-ZETA.

3. In any other case the algorithm tries the following sequence:

   (a) If any of the sides is a projector of a structure it tries the rules in Figure 8. Except when both sides are the same projector.

   (b) If both sides have the same number of arguments, try APP-FO.

   (c) If any of the above failed, try rules in 3 in the order shown in the figure.

Author Index

Abel, Andreas 69

Baader, Franz 22
Balbiani, Philippe 26
Baumgartner, Alexander 62
Bonacina, Maria Paola 47

Cabrer, Leonardo Manuel 41

Erbatur, Serdar 33, 36

Hibbs, Peter 15

Kapur, Deepak 33, 36
Kutsia, Temur 62

Levy, Jordi 5, 62

Marshall, Andrew M. 33, 36
Meadows, Catherine 33
Mectalfe, George 41
Mehto, Shweta 15
Meseguer, Jose 1
Morawska, Barbara 22

Narendran, Paliath 15, 33, 36
Novikova, Tatyana 55

Plaisted, David 47

Ringeissen, Christophe 33, 36

Sozeau, Matthieu 74

Tinchev, Tinko 26

Vezzosi, Andrea 69
Villaret, Mateu 62

Zakharov, Vladimir 55
Ziliani, Beta 74