The 34th International Workshop on UNIFICATION

UNIF 2020

WORKSHOP PROCEEDINGS

Temur Kutsia and Andrew M. Marshall (Editors)

(June 2020)

RISC-Linz Report Series No. 20-10

Supported by:

Preface

This volume collects contributions presented at the 34th International Workshop on Unification (UNIF 2020), held on June 29, 2020. It was a part of "Paris Nord Summer of LoVe 2020", a joint event on LOgic and VErification at Université Paris 13, made of IJCAR 2020, FSCD 2020, Petri Nets 2020, and over 20 satellite workshops. Due to COVID-19 pandemic, all these events, including UNIF 2020, were held online.

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.

UNIF is a well-established event with more than three decades of history. It is a yearly forum, where researchers in unification theory and related fields meet old and new colleagues, present recent (even unfinished) work, and 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 UNIF 2020 Program Committee selected 11 contributions. Each paper was reviewed by at least three reviewers. In addition, the program included two invited talks given by Stéphanie Delaune on *Rewriting in Protocol Verification* and by Manfred Schmidt-Schauß on *Nominal Algorithms: Applications and Extensions.*

Many people helped to make UNIF 2020 a successful event. We would like to thank the Conference Chairs of FSCD and IJCAR: Stefano Guerrini and Kaustuv Chaudhuri, the FSCD/ IJCAR Workshops Chairs Giulio Manzonetto and Andrew Reynolds, and the UNIF Steering Committee for their support in the preparation of the workshop. The work of the Program Committee was greatly helped by Andrei Voronkov's EasyChair system.

Temur Kutsia
June 2020 Andrew M. Marshall

Organization

Workshop Chairs

Temur Kutsia	RISC, Johannes Kepler University Linz
Andrew M. Marshall	University of Mary Washington

Program Committee

Mauricio Ayala Rincón	Universidade de Brasília
Franz Baader	TU Dresden
Alexander Baumgartner	University of Chile
Evelyne Contejean	LRI, CNRS, Univ Paris-Sud, Orsay
Daniel Dougherty	Worcester Polytechnic Institute
Besik Dundua	Ivane Javakhishvili Tbilisi State University
Serdar Erbatur	University of Texas at Dallas
Santiago Escobar	Universitat Politècnica de València
Maribel Fernández	King's College London
Silvio Ghilardi	Università degli Studi di Milano
Pascual Julián-Iranzo	University of Castilla-La Mancha
Temur Kutsia	RISC, Johannes Kepler University Linz, co-chair
Jordi Levy	IIIA - CSIC
Christopher Lynch	Clarkson University
Andrew M. Marshall	University of Mary Washington, co-chair
Barbara Morawska	Ahmedabad University
Daniele Nantes-Sobrinho	Universidade de Brasília
Paliath Narendran	University at Albany–SUNY
Veena Ravishankar	University of Mary Washington
Christophe Ringeissen	INRIA

Additional Reviewers

Du, Wei
Grigolia, Revaz
Pulver, Andrew
Suchy, Ashley

Table of Contents

Rewriting in Protocol Verification

Stéphanie Delaune

University of Rennes, CNRS, IRISA
Stephanie.Delaune@irisa.fr

Abstract

In this talk, we will review some results and techniques that allow automatic analysis of security protocols. The focus will be on privacy-type security properties (e.g. anonymity, unlinkability, ...) which are usually expressed using a notion of equivalence, and are actually more difficult to analyse than secrecy and authentication properties. In particular, we will discuss some advances that have been done in protocol verification thanks to techniques originally developed for rewriting and unification theory.

Nominal Algorithms: Applications and Extensions

Manfred Schmidt-Schauß

Deot. Computer Science and Mathematics,
Goethe-university Frankfurt, Germany
schauss@ki.informatik.uni-frankfurt.de

Abstract

Nominal unification was introduced by Urban, Pitts, and Gabbay [8]. It turned out to be a smart abstraction which nicely supports reasoning in higher-order languages: It is more powerful than first-order unification and has far better computational properties than higher-order unification. In particular, reasoning on correctness and influence on resource-usage of program transformations can be supported, since overlaps of transformation and of reductions rules of the operational semantics can be computed by nominal unification. Nominal matching, nominal rewriting and solving nominal constraints support the reasoning on transformation and reduction sequences.

However, for a wider application to more expressive higher-order languages, the algorithms have to be adapted to language extensions like recursive let or name restrictions as in pi-calculus. There are further extensions: atom-variables to improve the computational coverage, environment-variables to abstract sets of (recursive) bindings, and context variables to abstract positions within expressions. I will motivate and sketch the approaches to algorithms adapted for these extensions, and justify some restrictions or modifications to enable good computational properties of the nominal algorithms, in particular unification and matching. Selected work on extensions is discussed and potential future work is sketched.

1 Introduction: Classical Nominal Unification

1.1 Classical nominal unification

The work in [8] presents an algorithm to unify abstract expressions for a ground language defined by $e ::= a \mid \lambda a.e \mid f(a_1, \ldots, a_n)$ where a represents atoms (i.e. names), e the expressions, λ is the usual lambda abstraction, and $f(a_1, \ldots, a_n)$ permits to form terms, where application $(e\ e)$ can be represented using a binary function symbol. The abstract language also has unification variables (or expression variables X), where expressions can be substituted. The language for solutions has in addition permutations of ground atoms, and two extra terms in the grammar for e: X, and $\pi{\cdot}e$. A solution usually consists of a substitution and a set of constraints $a\#e$. Nominal unification solves equations w.r.t. α-equivalence on the ground language. The introduction of permutations is, however, not only technical, but really adds to the power of the method. The main reason is that $e_1 \sim_\alpha e_2 \Leftrightarrow \pi{\cdot}e_1 \sim_\alpha \pi{\cdot}e_2$, and that argueing on permutations is smoother than argueing on renamings. Nominal unification in its basic form is unitary and a polynomial (i.e.quadratic) algrithm is known for computing a unifier as well as for deciding nominal unifiability [8, 4, 1].

2 Extension by Atom Variables

The first extension which I want to mention is the addition of atom variables in nominal unification. This is discussed already in [8, 9] where an alleged counter argument is that the inclusion

of atom variables leads to equations like: $(A\ B)\cdot C \doteq C$, which has two incomparable solutions:
1. $A = B$; 2. $A\#B, A\#C, B\#C$

A remedy to regain flexibility in using names is in [2, 3] where equivariance of relations are proposed, which permit arbitrary global atom permutations.

However, equivariance cannot express "$a = b \vee a \neq b$", i.e. it is restricted to permuting names, whereas our proposal is to use atom-variables, which can also express solutions σ for atom variables A, B, where $\sigma(A) = \sigma(B)$ may be a valid as well as $\sigma(A) \neq \sigma(B)$.

The two incomparable solutions of $(A\ B)\cdot C \doteq C$ are 1. $A = B$, and 2. $\{A\#B, A\#C, B\#C\}$, which can be represented by one solution using atom-variables using a single freshness constraint: $C\#\lambda(A\ B)C.C$ [7].

Hence the general idea in using atom-variables is to keep the unique solutions property, and to put the alternatives into the constraint system. This is successfully worked out in [7] for a nominal unification algorithm.

An essential rule in the nominal unification algorithm in [8, 9] is the decomposition rule for lambda-expression:

$$\frac{\lambda a.X \doteq \lambda b.Y}{\{a\#Y\}, X \doteq (a\ b)\cdot Y}$$

where our proposal with atom-variables is the rule:

$$\frac{\lambda A.X \doteq \lambda B.Y}{\{A\#\lambda B.Y\}, X \doteq (A\ B)\cdot Y}$$

The results in [7] for an appropriately designed unification algorithm are:

- Nominal unification with atom-variables is unitary.
- Its complexity is polynomial, where sharing has to be used for representing permutations.
- Solvability is NP-complete. The reason is that the constraints of an outputted solution may represent an empty set of ground solutions, and that this satisfiability problem is NP-complete.

As a summary:

- Profit: Polynomial time computation of a single unifier, and increased flexibility in choosing names.
- Costs: nested permutations, which in general can not be simplified, for example $((A\ B)\cdot C\ \ (A\ C)\cdot B))$ which requires a sharing structure for permutations;
  and an NP-hard decision problem.

3 Application in Reasoning on Programming Languages

One motivation for studying nominal algorithms are higher-order equational theories where a decision algorithm for equations modulo the equational theory and modulo α-equivalence is of use. Therefore Higher-Order Rewriting is of interest and also decidable and efficient algorithms for Nominal Unification, Nominal Matching, Nominal Rewriting, and Nominal Constraint Solving, such that a Nominal Knuth Bendix confluence check can be proved correct and then applied

The **main motivation** for us is correctness (w.r.t. semantics) of transformations in higher-order programming languages.

As a starting point the setting is: a declarative higher-order programming language and its operational semantics as a set of small-step rules. Examples of typical rules are (in a core variant of Haskell):

$R[(\lambda x.s_1)\ s_2] \to R[(\texttt{letr}\ x = s_2\ \texttt{in}\ s_1)]$
$R[(\texttt{letr}\ E_1\ \texttt{in}\ (\texttt{letr}\ E_2\ \texttt{in}\ s))] \to R[\texttt{letr}\ E_1, E_2\ \texttt{in}\ s]$
$R[(\texttt{letr}\ x = (\texttt{letr}\ E_2\ \texttt{in}\ s_2), E_1\ \texttt{in}\ s_1)] \to R[\texttt{letr}\ x = s_2, E_2, E_1\ \texttt{in}\ s_1]$

where R is a reduction context, and E_1, E_2 are any sub-environments.

Some typical transformations are

$(\lambda x.e_1)\ e_2 \to (\texttt{letr}\ x = e_2\ \texttt{in}\ e_1)$
$(\texttt{letr}\ E_1\ \texttt{in}\ (\texttt{letr}\ E_2\ \texttt{in}\ e)) \to (\texttt{letr}\ E_1, E_2\ \texttt{in}\ e)$
$(\texttt{letr}\ E_1\ \texttt{in}\ e_1)\ (\texttt{letr}\ E_2\ \texttt{in}\ e_2) \to (\texttt{letr}\ E_1, E_2\ \texttt{in}\ (e_1\ e_2))$

The correctness of transformations requires the following proof tasks: if $e_1 \xrightarrow{trans} e_2$, then $e_1\!\downarrow \iff e_2\!\downarrow$, where $e\!\downarrow$ means that there is a reduction sequence from e to a value, using the operational semantics. Note that the operational small-step semantics is usually equivalent to nominal rewriting of expressions by reduction rules, but in general without constraints.

$$\begin{tikzcd}
e_1 \arrow[r, "\text{trans}"] \arrow[d, "\text{red}_i"'] & {e'}_1 \arrow[d, dashed, "\text{red},*"] \\
e_2 \arrow[r, dashed, "\text{trans},*"] \arrow[d, "\text{red}_j"'] & {e'}_2 \arrow[d, dashed] \\
\dots \arrow[d] & \dots \arrow[d, dashed] \\
e_i\ (\text{value}) & e'_j\ (\text{value})
\end{tikzcd}$$

Necessary subtasks:
Computation of overlap possibilities by *nominal unification*;
Computation of the corresponding joins also using *nominal matching and rewriting.*
Proof Goal: Show that $e_1\!\downarrow$ if and only if $e'_1\!\downarrow$.
Induction arguments by *nominal rewriting*, *nominal matching* and *nominal constraint solving* and the computed join patterns.

An example for correctness of transformations is the reduction and transformation rule (cp).

$$\texttt{let}\ x = v\ \texttt{in}\ C[x] \quad \to \quad \texttt{let}\ x = v\ \texttt{in}\ C[v]$$

where C is any context, even with binders, C does not bind x, and v is a value (abstraction, data).

In a core-language formulation of Haskell with recursive let, the rule (cp) is as follows:

$$\texttt{letr}\ x = v, E\ \texttt{in}\ C[x] \quad \to \quad \texttt{letr}\ x = v, E\ \texttt{in}\ C[v]$$

where C is any context, even with binders, and C does not bind x; E is a set of bindings (environment); and v is a value (abstraction, data).

The requirement is to adapt nominal algorithms to further syntactic constructs: Let-constructs: recursive let: `letr`, and name-binder ν. context-variables C, environment variables E, and the full treatment requires that also generalized constraints are used.

Currently it is unknown how to combine all these generalizations in one setting. So first the extensions are checked one-by-one: `letr` (recursive let as in Haskell); context-variables C, environment variables E, and perhaps more.

4 Extension By Recursive Let

First investigation: `letr`, but no atom variables [5].

The language extension is $\ldots \mid (\texttt{letr}\ a_1.e_1,\ldots,a_n.e_n\ \texttt{in}\ e)$, where the scope of a_i is in every e_j and e. Syntactically, $(\texttt{letr}\ a_1.e_1,\ldots,a_n.e_n\ \texttt{in}\ e)$ is considered as the same as $(\texttt{letr}\ \texttt{env}\ \texttt{in}\ e)$ where `env` is a permutation of $(a_1.e_1,\ldots,a_n.e_n)$

The following extra problems have to be tackled: A unification rule decomposing an equation between two `letr`-expressions, and fixpoint equations $X = \pi{\cdot}X$, which have trivial solutions in classic nominal unification.

An extra unification rule for decomposing `letr` is:

$$\frac{\Gamma\dot\cup\{\texttt{letr}\ a_1.s_1;\ldots,a_n.s_n\ \texttt{in}\ r \doteq \texttt{letr}\ b_1.t_1;\ldots,b_n.t_n\ \texttt{in}\ r'\}}{\mid_{\forall\rho}\ \Gamma\dot\cup \mathit{flat}(\lambda a_1.\ldots a_n.(s_1,\ldots,s_n,r) \doteq \lambda b_{\rho(1)}.\ldots b_{\rho(n)}.(t_{\rho(1)},\ldots,t_{\rho(n)},r'))}$$ where ρ is guessed as a permutation on $\{1,\ldots,n\}$

Notice that $\{a_1,\ldots,a_n\} \cap \{b_1,\ldots,b_n\}$ maybe nonempty. This rule is the only non-deterministic one in this setting. The equality defined by permuting environments has as consequence that there are nontrivial expressions as fixpoints of (atom-)permutations. For usual nominal unification: $(a\ b){\cdot}e \sim e$ implies a, b not free in e! In the language with recursive let: a, b may be free in e:

$$(a\ b)\cdot(\texttt{letr}\ c.a; d.b\ \texttt{in}\ \mathit{True}) = (\texttt{letr}\ c.b; d.a\ \texttt{in}\ \mathit{True}) \sim (\texttt{letr}\ c.a; d.b\ \texttt{in}\ \mathit{True}).$$

The observation is that there are expressions t in the letrec-language with $t \sim (a\ b)\cdot t$ where $\{a, b\}$ are free atoms in t. The consequence for the unification algorithm is that substitutions together with freshness constraints are insufficient to represent solutions.

Other central rules for letrec nominal unification are:

(MMS) $$\frac{\Gamma\dot\cup\{X \doteq e_1, X \doteq e_2\},\nabla}{\Gamma\dot\cup\{X \doteq e_1, e_1 \doteq e_2\},\nabla}$$ if e_1, e_2 are neither variables nor suspensions.

(FPS) $$\frac{\Gamma\dot\cup\{X \doteq \pi_1{\cdot}X,\ldots,X \doteq \pi_n{\cdot}X, X \doteq e\},\theta}{\Gamma\dot\cup\{e \doteq \pi_1{\cdot}e,\ldots,e \doteq \pi_n{\cdot}e\},\theta \cup \{X \mapsto e\}}$$ if X is maximal w.r.t. $>_{vd}$, $X \notin \mathit{Var}(\Gamma)$, and e is neither a variable nor a suspension, and no failure rule is applicable.

(ElimFP) $$\frac{\Gamma\dot\cup\{X \doteq \pi_1{\cdot}X,\ldots,X \doteq \pi_n{\cdot}X, X \doteq \pi{\cdot}X\},\theta}{\Gamma\dot\cup\{X \doteq \pi_1{\cdot}X,\ldots,X \doteq \pi_n{\cdot}X\},\theta}$$ if $\pi \in \langle\pi_1,\ldots,\pi_n\rangle$., i.e. π is in the subgroup generated by $\pi_1,\ldots.\pi_n$.

(Output) $$\frac{\Gamma,\nabla,\theta}{\theta,\nabla,\{\text{“}X \in \mathit{Fix}(\pi)\text{“} \mid X \doteq \pi \cdot X \in \Gamma\}}$$ if Γ only consists of fixpoint-equations.

ElimFP removes redundant fixpoint-equations if π is in the generated subgroup $\langle\pi_1,\ldots,\pi_n\rangle$. Finally, the minimized fixpoint equations are parts of the result. The extra techniques for efficiency for nominal unification in the letrec-language are: Using flattened expressions, and priorities to control application of MMS and FPS, Compressed (i.e. shared) representation of the output substitution, avoiding redundant fixpoint equations by (ElimFP) exploiting efficient (i.e. polynomial) algorithms in permutation groups to minimize the set of generators of a permutation group.

The results for letrec nominal unification are:

Theorem 4.1. *Nominal unifiability of letrec-expressions together with freshness constraints is NP-complete.*

The number of most general solutions is at most exponential, and the size of a single one is polynomial.

We also generalized the letrec-nominal unification algorithm to permit atom variables with similar results, yet unpublished.

5 Extension By Context Variables

We report on research extending nominal unification for input equations containing atom-variables and context-variables [6]. The input of the algorithm NomUnifyASD is (Γ, ∇): equations Γ and freshness and DVC-constraints ∇, where the latter intend to enforce that the variable condition holds: that bound variables are different and also different from free variables. This condition is justified for higher-order PL, since the semantics is modulo α-equivalence. A DVC-constraint $\textsc{dvc}(e)$ is solved by ρ, iff in $e\rho$ all bound variables are distinct and distinct from free variables where ρ uses fresh names for bound variables in every instantiation $x\rho$.

An extra condition is that for every input equation $e_1 \doteq e_2$, the constraints $\textsc{dvc}(e_1), \textsc{dvc}(e_2)$ must hold, resp., are in the input constraints.
The justification for higher-order programming languages is that (i) semantic properties are modulo α-equivalence, (ii) for expressions it is always legal to rename them such that the variable condition holds. For example: after beta-reduction, the result is usually renamed such that the variable condition holds.

The **nominal unification algorithm NomUnifyASD** requires as additional constraint that every context variable occurs at most once. As a justification, most all left-and right-hand sides of rules and transformations are linear in context variables.

The variable condition is required for the context decomposition lemma in the ground language: Let C_1, C_2 be (ground) contexts with the same hole positions, and $\textsc{dvc}[C_1[e_1]]$ and $\textsc{dvc}[C_2[e_2]]$ holds. Then

$$\begin{gathered} C_1[e_1] \sim_\alpha C_2[e_2] \\ \iff \\ \exists \text{ permutation } \pi \text{ with } C_1 \sim_\alpha \pi{\cdot}C_2 \text{ and } e_1 \sim_\alpha \pi{\cdot}e_2, \end{gathered}$$

(plus some conditions on $dom(\pi)$)

The consequences for the unification algorithm are that this enables systematic solutions of $D_1[e_1] \doteq D_2[e_2]$, however, the unification rule introduces permutation variables.

Theorem 5.1. *(Main Result for NomUnifyASD) NomUnifyASD is sound and complete for ASD1 unification problems and runs in NEXPTIME.*

The collecting version returns an at most exponential set of polynomial-sized unifiers.

6 Final Remarks

To apply nominal algorithms future research maybe to construct algorithms for the combined extensions, such that proofs of correctness and other properties of translations can be supported by automated techniques. However, our conjecture is that the most general case is undecidable, hence restrictions like linearity appear justified.

References

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

[2] James Cheney. The complexity of equivariant unification. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*, volume 3142 of *LNCS*, pages 332–344. Springer-Verlag, 2004.

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

[4] Jordi Levy and Mateu Villaret. An efficient nominal unification algorithm. In Christopher Lynch, editor, *Proc. 21st RTA*, volume 6 of *LIPIcs*, pages 209–226. Schloss Dagstuhl, 2010.

[5] Manfred Schmidt-Schauß, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal unification of higher order expressions with recursive let. In Manuel V. Hermenegildo and Pedro López-García, editors, *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*, volume 10184 of *LNCS*, pages 328–344. Springer, 2016.

[6] Manfred Schmidt-Schauß and David Sabel. Nominal unification with atom and context variables. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*, pages 28:1–28:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[7] Manfred Schmidt-Schauß, David Sabel, and Yunus Kutz. Nominal unification with atom-variables. *J. Symb. Comput.*, pages 42–64, 2019.

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

[9] Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1–3):473–497, 2004.

An Investigation into Nominal Equational Problems (Work in progress)

Mauricio Ayala-Rincón[1], Maribel Fernández[2], Daniele Nantes-Sobrinho[3], and Deivid Vale[4*]

[1] Departamentos de Matemática e Ciência da Computação, Universidade de Brasília, Brazil
`ayala@unb.br`

[2] Department of Informatics, King's College London, London, UK
`maribel.fernandez@kcl.ac.uk`

[3] Departamento de Matemática, Universidade de Brasília, Brazil
`dnantes@mat.unb.br`

[4] Department of Software Science, Radboud University, The Netherlands
`deividvale@cs.ru.nl`

Abstract

We consider *nominal equational problems* of the form $\exists W \forall Y : P$, where P consists of conjunctions and disjunctions of equations of the form $s \approx_\alpha t$ (read: "s is α-equivalent to t"), freshness constraints of the form $a\#t$ (read: "a is fresh for t") and their negations: $s \not\approx_\alpha t$ and $a \not\# t$, where a is an atom and s, t are nominal terms. When dealing with general nominal equational problems we face the challenge of properly defining their semantics to take into account the interaction between negative freshness constraints and the existential and universal quantifiers. Here we propose a discussion regarding two different approaches: (i) adopting the usual freshness and equational contraints; (ii) the use of the "new" quantifier (И) and fixed point equations instead of freshness constraints; in both cases being careful to obtain the correct meaning.

1 Introduction

Disunification problems have been extended to the nominal setting [2], using a restricted form of constraints called *nominal (disunification) constraints*: equations (judgments $\Delta \vdash s \approx_\alpha{}^? t$) enriched with disequations, i.e., negated equations of the form $s \not\approx_\alpha{}^? t$. In that setting, a nominal constraint problem $\mathcal{P}$ is equivalent to the existentially closed formula:

$$\mathcal{P} := \exists \overline{X} \left(\left(\bigwedge \Delta_i \vdash s_i \approx_\alpha t_i \right) \wedge \left(\bigwedge \nabla_j \vdash p_j \not\approx_\alpha q_j \right) \right).$$

This problem is solved in the nominal term-algebra $\mathcal{T}(\Sigma, \mathbb{A}, \mathbb{X})$ by constructing suitable representation to the witnesses for the variables in $\mathcal{P}$ [2].

Comon and Lescanne [6] investigated the so-called *equational problem*, in their words: "an equational problem is any first-order formula whose only predicate symbol is =", that is, it has the form $\exists w_1, \ldots, w_n \forall y_1, \ldots, y_m : P$ where P is a *system*, i.e., an equation $s = t$, or a disequation $s \neq t$, or a disjunction of systems $\bigvee P_i$, or a conjunction of systems $\bigwedge P_i$, or a failure $\bot$, or success $\top$. The motivation to study such problems was the applicability in pattern-matching for functional languages, sufficient completeness for term rewriting systems, dealing with negation in logic programming languages, etc.

*The author is supported by the NWO TOP project "Implicit Complexity through Higher Order Rewriting" (ICHOR), NWO 612.001.803/7571.

With the development of nominal techniques, including nominal logic [10], nominal unification and rewriting [7], nominal logic programming [4], and nominal (universal) algebra [8], it is natural to extend equational problems into the "nominal world" and consider *nominal equational problems*. Based on Comon and Lescanne's work, the expected form of a nominal extension to the first-order equational problem would be

$$\mathcal{P} ::= \exists W_1 \dots W_n \forall Y_1 \dots Y_m : P$$

with P being a *nominal system*, i.e., a formula consisting of conjunctions and disjunctions of freshness, equality constraints, and their negations.

In this paper, we discuss alternative formulations of nominal equational problems taking into account the kind of constraints used and the model on which they are interpreted. We also discuss a preliminary rule based strategy to solve such problems. This work is a first step towards the generalisation of nominal disunification constraint problems (introduced in [2]) which consist of equations and disequations without universally-quantified variables.

2 Background

We assume the reader is familiar with nominal techniques and recall some concepts and notations that shall be used in the paper; for more details the reader is referred to [7, 11].

Fix countable infinite, pairwise disjoint, sets of *atoms* $\mathbb{A} = \{a, b, c, \dots\}$ and *variables* $\mathbb{X} = \{X, Y, Z, \dots\}$. Atoms follow the permutative convention, i.e., names a, b, and c run permutatively over $\mathbb{A}$, therefore they represent different names. As usual, we form nominal terms with a finite set Σ of *term-formers* — disjoint from $\mathbb{A}$ and $\mathbb{X}$ — such that for each $f \in \Sigma$, a unique non-negative integer n (the arity of f, written as $f : n$) is assigned.

A *permutation* π is a bijection $\mathbb{A} \to \mathbb{A}$ with finite domain, i.e., the set $\mathtt{supp}(\pi) := \{a \in \mathbb{A} \mid \pi(a) \neq a\}$ is finite. Write $\mathtt{id}$ for the identity permutation and $\pi \circ \pi'$ for the composition of π and π'. The difference set of π and γ is defined by $\mathtt{ds}(\pi, \gamma) = \{a \in \mathbb{A} \mid \pi(a) \neq \gamma(a)\}$.

Nominal terms are given by the following grammar: $s, t := a \mid \pi \cdot X \mid [a]t \mid f(t_1, \dots, t_n)$ where a is an *atom*, $\pi \cdot X$ is a moderated variable, $[a]\,t$ is the *abstraction* of a in the term t, and $f(t_1, \dots, t_n)$ is a *function application* with $f \in \Sigma$ and $f : n$. We abbreviate a ordered sequence $t_1, \dots, t_n$ of terms by $\tilde{t}$.

Example 1. *Let $\Sigma_\lambda := \{\mathtt{lam} : 1, \mathtt{app} : 2\}$ be a signature for the λ-calculus. Using atoms to represent λ-calculus variables, λ-expressions are generated by the grammar: $e := a \mid \mathtt{lam}([a]\,e) \mid \mathtt{app}(e, e)$. As usual, we write $\mathtt{app}(s, t)$ as $s\,t$ and $\mathtt{lam}([a]\,s)$ as $\lambda\,[a]\,s$. The following are examples of nominal terms: $(\lambda\,[a]\,a)\,X$ and $(\lambda\,[a]\,(\lambda\,[b]\,b\,a)\;\,c)\,d$.*

The *action of a permutation* π on a term t is inductively defined by: $\pi \cdot a = \pi(a)$, $\pi \cdot (\pi' \cdot X) = (\pi \circ \pi') \cdot X$, $\pi \cdot ([a]t) = [\pi(a)](\pi \cdot t)$, and $\pi \cdot f(t_1, \dots, t_n) = f(\pi \cdot t_1, \dots, \pi \cdot t_n)$. *Substitutions*, ranging over $\sigma, \gamma, \tau \dots$, are maps (with finite domain) from variables to terms. The *action of a substitution* σ on a term t, denoted $t\sigma$, is inductively defined by: $a\sigma = a, (\pi \cdot X)\sigma = \pi \cdot (X\sigma)$, $([a]t)\sigma = [a](t\sigma)$ and $f(t_1, \dots, t_n)\sigma = f(t_1\sigma, \dots, t_n\sigma)$. Note that $t(\sigma\gamma) = (t\sigma)\gamma$.

Equality and Freshness Constraints A *nominal equation* (disequation) is the symbol $\top$ ($\bot$) or an expression of the form $s \approx_\alpha t$ ($s \not\approx_\alpha t$) where s and t are nominal terms. A *trivial equation* is either of the form $s \approx_\alpha s$ or $\top$. Similarly, a *trivial disequation* is either $s \not\approx_\alpha s$ or $\bot$.

A finite set of *primitive freshness constraints* of the form $a\#X$ is called a *freshness context*, we use Δ, ∇, and Γ to denote them. Equality and freshness constraints are defined by the derivation rules in Figure 1 below.

$$\frac{}{\nabla \vdash a \approx_\alpha a}\text{(ax)} \qquad \frac{\nabla \vdash t_1 \approx_\alpha t'_1 \quad \cdots \quad \nabla \vdash t_n \approx_\alpha t'_n}{\nabla \vdash f(t_1, \ldots t_n) \approx_\alpha f(t'_1, \ldots, t'_n)}\text{(f)} \qquad \frac{\nabla \vdash t \approx_\alpha t'}{\nabla \vdash [a]t \approx_\alpha [a]t'}\text{(abs-a)}$$

$$\frac{\nabla \vdash t \approx_\alpha (a\,a') \cdot t' \qquad \nabla \vdash a \# t'}{\nabla \vdash [a]t \approx_\alpha [a']t'}\text{(abs-b)} \qquad \frac{a\#X \in \nabla \text{ for all } a \text{ s.t. } \pi \cdot a \neq \pi' \cdot a}{\nabla \vdash \pi \cdot X \approx_\alpha \pi' \cdot X}\text{(var)}$$

$$\frac{}{\nabla \vdash a\#b}\text{(\#-ax)} \qquad \frac{(\pi^{-1} \cdot a\#X) \in \nabla}{\nabla \vdash a\#\pi \cdot X}\text{(\#-var)} \qquad \frac{}{\nabla \vdash a\#[a]t}\text{(\#-abs-a)}$$

$$\frac{\nabla \vdash a\#t}{\nabla \vdash a\#\,[b]\,t}\text{(\#-abs-b)} \qquad \frac{\nabla \vdash a\#t_1 \qquad \cdots \qquad \nabla \vdash a\#t_n}{\nabla \vdash a\#f(t_1, \ldots t_n)}\text{(\#-f)}$$

Figure 1: Equality and Freshness Rules

3 (NEP) Nominal Equational Problems

Definition 1. *A* nominal system P *is a formula defined by the following grammar*

$$P, P' ::= \top \mid \bot \mid s \approx_\alpha t \mid s \not\approx_\alpha t \mid a\#t \mid a\not\#t \mid P \wedge P' \mid P \vee P'.$$

Although not usual, the negation of freshness — denoted as $a\not\#X$ — means that *a is not fresh for X*, that is, there exists an instance $t = X\sigma$ of X with at least one free occurrence of a.

Definition 2 (NEP-First Version)**.** *A* NEP *is a formula of the form*

$$\mathcal{P} ::= \exists W_1 \ldots W_n \forall Y_1 \ldots Y_m : P$$

where P is a nominal system and $\overline{W} = \{W_1, \ldots, W_n\}, \overline{Y} = \{Y_1, \ldots, Y_m\}$, are sets of mutually distinct variables called respectivley auxiliary unknowns *and* parameters. *$fv(\mathcal{P})$ denotes the set of free variables occurring in $\mathcal{P}$ also called* principal unknowns.

Example 2 (Nominal Disunification Constraints)**.** *Nominal disunification constraints [2] have the form $\mathcal{P} := \exists\overline{X}\langle E \mid\mid D\rangle$, where E is a finite set of nominal equations in context, i.e., $E = \bigcup_{0\leq i\leq n}\{\Delta_i \vdash s_i \approx_\alpha t_i\}$ and D is a finite set of nominal disequations in context, $D = \bigcup_{0\leq j\leq m}\{\nabla_j \vdash u_j \not\approx_\alpha v_j\}$. This problem is a particular case of* NEP*: if one takes the judgment $\Delta \vdash s \approx_\alpha t$ as $\Delta \Rightarrow s \approx_\alpha t$, or yet as $\neg\Delta \vee s \approx_\alpha t$[1], we obtain the following formula:*

$$\mathcal{P} := \exists\overline{X}(\bigwedge_{i=0}^{n}(\neg[\Delta_i] \vee s_i \approx_\alpha t_i)) \wedge (\bigwedge_{j=0}^{m}(\neg[\nabla_j] \vee u_j \not\approx_\alpha v_j)), \tag{1}$$

where $[\Delta_i], [\nabla_j]$ are conjunctions of freshness constraints contained in Δ_i, ∇_j, respectively.

3.1 Solutions of Equational Problems

Let $\mathcal{P} = \exists\overline{W}\forall\overline{Y} : P$ be a NEP. Let $\mathcal{A}$ be an algebra that provides an interpretation for the symbols in the signature. An *$\mathcal{A}$-solution* for $\mathcal{P}$ is a pair $\langle\Gamma, \sigma\rangle$, consisting of a freshness context Γ and a substitution σ, such that $\langle\Gamma, \sigma\rangle$ *$\mathcal{A}$-validates* the system P (as defined below). We will assume that $\mathcal{A}$ is the nominal algebra of terms $\mathcal{T}(\Sigma, \mathbb{A}, \mathbb{X})$, but one could use the ground algebra $\mathcal{T}(\Sigma, \mathbb{A}, \emptyset)$ or a quotient algebra, say $\mathcal{T}(\Sigma, \mathbb{A}, \mathbb{X})/ =_E$ for a given equational theory E.

[1]Similarly, for disequations

Definition 3. *Let C range over equality and freshness constraints and D range over negative constraints (disequations and negated freshness). We denote by $C\sigma$ (resp. $D\sigma$) the constraint obtained by instantiating the terms in C (resp. D) with the substitution σ and by $\neg D$ the positive constraint obtained by negating D.*

*A pair $\langle \Gamma, \sigma \rangle$ $\mathcal{A}$-*validates *a system P iff*

1. *$P = \top$; or* 2. *$P = C$; $\Gamma \vdash C\sigma$ holds in $\mathcal{A}$; or* 3. *$P = D$; $\Gamma \not\vdash \neg D\sigma$ in $\mathcal{A}$; or*
4. *$P = P_1 \wedge \ldots \wedge P_n$ and $\langle \Gamma, \sigma \rangle$ $\mathcal{A}$-validates each P_i, $1 \leq i \leq n$; or*
5. *$P = P_1 \vee \ldots \vee P_m$ and $\langle \Gamma, \sigma \rangle$ $\mathcal{A}$-validates at least one P_i, $1 \leq i \leq m$.*

The definition of solution relies on a pair $\langle \Gamma, \gamma \rangle$ being *away from a set of variables.*

Definition 4. *A pair $\langle \Gamma, \gamma \rangle$ of a freshness context and a substitution is* away from a set of variables *$\mathbb{V} \subset \mathbb{X}$ iff Γ does not contain any $a\#X$ with $X \in \mathbb{V}$ and γ is away from $\mathbb{V}$, i.e., no variable from $\mathbb{V}$ occurs in $\langle \Gamma, \gamma \rangle$.*

Definition 5. *A pair $\langle \Gamma, \gamma \rangle$ is an $\mathcal{A}$-*solution *of the* NEP *$\mathcal{P} = \exists \overline{W} \forall \overline{Y} : P$ if, and only if, the following conditions hold:*

1. *$\langle \Gamma, \gamma \rangle$ is away from $\overline{W} \cup \overline{Y}$ and $\mathtt{dom}(\gamma) = \overline{X} = fv(\mathcal{P})$;*
2. *there is a pair $\langle \Delta, \delta \rangle$ away from $\overline{Y} \cup \overline{X}$ ($\mathtt{dom}(\delta) = \overline{W}$) such that for all pairs $\langle \Lambda, \lambda \rangle$ away from $\overline{W} \cup \overline{X}$ ($\mathtt{dom}(\lambda) = \overline{Y}$), $\langle \Gamma\Delta\Lambda, \gamma\delta\lambda \rangle$ $\mathcal{A}$-validates P.*

3.2 Nominal Equational Solved Forms

The future goal is to develop a procedure to solve NEP based on applications of *simplification rules*, as proposed in [6], that transform problems into simpler ones preserving the set of solutions. Successive applications of such rules lead to a *solved form* from which we know how to extract a solution from. We consider three first-order solved forms: *parameterless*, *unification*, and *definition with constraints*. Below we extend those notions to the nominal setting.

Definition 6 (Solved Forms)**.**

1. *A* NEP *$\mathcal{P}$ is in* unification solved form *if it is equivalent to a nominal unification problem of the form $\langle \Gamma, X_1 \approx_\alpha t_1 \wedge \ldots \wedge X_n \approx_\alpha t_n \rangle$ where all the unknowns $X_1, \ldots, X_n$ are distinct and do not occur in the t_i's and Γ is a freshness context;*
2. *A* NEP *$\mathcal{P}$ is in* parameterless solved form *if it contains no universal quantifiers.*
3. *A* NEP *is a* definition with constraints *if it is either $\top, \bot$, or a problem of the form $\mathcal{P} := \exists \overline{X} (\bigwedge_{i=0}^{n} (\neg[\Delta_i] \vee X_i \approx_\alpha t_i)) \wedge (\bigwedge_{j=0}^{m} (\neg[\nabla_j] \vee X'_j \not\approx_\alpha v_j))$, where variables $X_1, \ldots, X_n$ occur only once in the equational part (left conjunction). Variables X'_j is different from v_j, for $1, \leq j \leq m$. $[\Delta_i]$ and $[\nabla_j]$ are defined as in Example 2.*

It is essential to remark that as in [6] the *definition with constraints* solved form is equivalent to the *disunification problem* introduced in [3], and its extension to the nominal setting is the disunification constraints problem [2] described in Example 2. We discuss rules in the Appendix.

3.3 Discussion: a strict equational approach to (NEP)

It is natural to try to define NEP as nominal formulas whose only predicate symbol is $\approx_\alpha$. For that, we can explore existing results relating freshness and equality constraints. Initially, the freshness predicate was defined using a quantified fixed point equation, in [11]: $a\#t$ iff $\text{И}a'.(a\ a')\cdot t \approx t$.

In another work, a freshness constraint was shown to have a tight relation with a specific equation between abstracted atoms:

Lemma 1 (Lemma 3.1 in [9]). *$P \cup \{a\#^? t\}$ and $P \cup \{[a][b]t \approx^? [b][b]t\}$ have the same solutions.*

The above approaches motivate the following definition for NEP:

Definition 7 (NEP-Second Version). *A NEP is a formula of the form*

$$\mathcal{P} ::= \exists\overline{W}\forall\overline{Y}\text{И}\overline{a} : P$$

where P is a system generated by the grammar $P, P' ::= \top \mid \bot \mid s \approx_\alpha t \mid s \not\approx_\alpha t \mid P \wedge P' \mid P \vee P'$, and $\overline{W} = \{W_1, \dots, W_n\}, \overline{Y} = \{Y_1, \dots, Y_m\}$, and $\boldsymbol{fv}(\mathcal{P})$, as in Definition 2, are the auxiliary unknowns*,* parameters *and* principal unknowns.

As shown in [1], solving equations via freshness constraints is equivalent to following the approach via fixed point equations when the equational theory is empty. However, when dealing with equational theories that include commutativity, it seems to be more convenient to use a purely equational approach. Therefore, we conjecture that this second approach would be more convenient when dealing quotient algebras, such as $\mathcal{T}(\Sigma, \mathbb{A}, \mathbb{X})/=_C$, or $\mathcal{T}(\Sigma, \mathbb{A}, \mathbb{X})/=_{AC}$, among others.

4 Conclusion

We have considered two approaches to define NEPs as a straightforward nominal version of the problem introduced in [6, 5], using (negated) freshness constraints in addition to $\approx_\alpha$, or using purely $\approx_\alpha$ as predicate but with the "new" quantifier И. As future work we plan to investigate which approach is more convenient when defining the rules for simplifying the NEPs in order to obtain a correct procedure to solve such problems, besides we also intend to investigate NEPs modulo equational theories.

References

[1] Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. Fixed-point constraints for nominal equational unification. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD*, pages 7:1–7:16, 2018. URL: https://doi.org/10.4230/LIPIcs.FSCD.2018.7, doi:10.4230/LIPIcs.FSCD.2018.7.

[2] Mauricio Ayala-RincÃşn, Maribel FernÃąndez, Daniele Nantes-Sobrinho, and Deivid Vale. On Solving Nominal Disunification Constraints. *ENTCS*, 348:3 – 22, 2020. Proc. 14th Int. Workshop on Logical and Semantic Frameworks, with Applications LSFA 2019. doi:10.1016/j.entcs.2020.02.002.

[3] Wray L. Buntine and Hans-Jürgen Bürckert. On solving equations and disequations. *J. ACM*, 41(4):591–629, July 1994. URL: http://doi.acm.org/10.1145/179812.179813, doi:10.1145/179812.179813.

[4] James Cheney and Christian Urban. *alpha*-Prolog: A Logic Programming Language with Names, Binding and α-Equivalence. In *Proc. 20th International Conference on Logic Programming ICLP*, volume 3132 of *LNCS*, pages 269–283. Springer, 2004. URL: https://doi.org/10.1007/978-3-540-27775-0_19, doi:10.1007/978-3-540-27775-0_19.

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

[6] Hubert Comon and Pierre Lescanne. Equational problems and disunification. *J. Symb. Comput.*, 7(3/4):371–425, 1989. URL: https://doi.org/10.1016/S0747-7171(89)80017-3, doi:10.1016/S0747-7171(89)80017-3.

[7] Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting. *Information and Computation*, 205(6):917 – 965, 2007. doi:10.1016/j.ic.2006.12.002.

[8] Murdoch J. Gabbay and Aad Mathijssen. Nominal (Universal) algebra: equational logic with names and binding. *J. of Logic and Computation*, 19(6):1455–1508, 2009. doi:10.1093/logcom/exp033.

[9] Jordi Levy and Mateu Villaret. An efficient nominal unification algorithm. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA*, volume 6 of *LIPIcs*, pages 209–226, 2010. URL: https://doi.org/10.4230/LIPIcs.RTA.2010.209, doi:10.4230/LIPIcs.RTA.2010.209.

[10] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165 – 193, 2003. doi:10.1016/S0890-5401(03)00138-X.

[11] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science.* Cambridge University Press, New York, NY, USA, 2013. doi:10.1017/CBO9781139084673.

A Rule based procedure

In this section we present a preliminary set of simplification rules which will be used in the algorithm for solving `NEP`. They are still under investigation and proofs of correctness and termination are ongoing work.

Intuitively, a set of transformation rules transforms a problem $\mathcal{P}$ into a problem $\mathcal{P}'$ (denoted as $\mathcal{P} \implies \mathcal{P}'$), which are simpler in some sense. Transformation rules may have conditions (rule controls) in order to be applied. The goal is to reach one of the normal forms defined above. Different strategies can possible lead to different normal forms. Strategies can also vary according to the model where the problem is being solved.

The primary control gives priority for application of rules: we split rules into sets $\mathcal{R}_i$ using the index i as a priority stack, i.e., a rule $R \in \mathcal{R}_i$ can only be applied if no rules from $\mathcal{R}_j$, with $j < i$ can be applied. A *procedure* on a `NEP` $\mathcal{P}$ is a strategy $R_0, R_1, \ldots, R_k$ for application of rules such that $\mathcal{P} = \mathcal{P}_0 \Longrightarrow_{R_0} \mathcal{P}_1 \Longrightarrow_{R_2} \ldots \Longrightarrow_{R_k} \mathcal{P}_{k+1}$, where $R_i \in \mathcal{R}_j$, for $0 \leq i \leq k$ and some $0 \leq j \leq 6$, satisfying the primary control give above. A problem $\mathcal{P}$ is rewritten in a *pattern-matching* fashion, i.e, rules give the pattern occurring in the problem. Before the application of each rule $\mathcal{P}$ is reduced to its conjunctive normal form.

The explosion rule creates a new problem for each $f \in \Sigma$. Given some explosion equation (disequation), all possible constructions with $f \in \Sigma$ must be considered for completeness' sake. Therefore, our procedure will build a finitely branching tree of problems to be solved.

Example 3. *Let $\mathcal{P}$ be a* `NEP`*, using the signature from Example 1, as follows:*

$$\mathcal{P} = \forall Y : \lambda\,[a]\,X \not\approx_\alpha \lambda\,[a]\,\lambda\,[a]\,Y \overset{dec}{\Longrightarrow} \forall Y : [a]\,X \not\approx_\alpha [a]\,\lambda\,[a]\,Y \overset{abs}{\Longrightarrow} \forall Y : X \not\approx_\alpha \lambda\,[a]\,Y$$

Notice that more rules can be applied and the explosion rule results in two parallel problems $\mathcal{P}_1 = \exists W_1 \forall Y : X \not\approx_\alpha \lambda\,[a]\,Y \wedge X \approx_\alpha \lambda W_1$ *and* $\mathcal{P}_2 = \exists W_1, W_2 \forall Y : X \not\approx_\alpha [a]\,Y \wedge X = W_1 W_2$.

$\mathcal{R}_0$: Trivial Rules

(T_1)	$t \approx_\alpha t \implies \top$	(T_2)	$t \not\approx_\alpha t \implies \bot$	(T_3)	$a \approx_\alpha b \implies \bot$	(T_4)	$a\#b \implies \top$
(T_5)	$a\#a \implies \bot$	(T_6)	$a\not\#a \implies \top$	(T_7)	$a\not\#b \implies \bot$		

$\mathcal{R}_1$: Clash and Occurrence Check Rules

(CL_1)	$f(\tilde{t}) \approx_\alpha g(\tilde{u}) \implies \bot$	(CL_2)	$f(\tilde{t}) \not\approx_\alpha g(\tilde{u}) \implies \top$	where $f \neq g$
(CL_3)	$s \not\approx_\alpha t \implies \top$	(CL_4)	$s \approx_\alpha t \implies \bot$	$s\vert_\epsilon \neq t\vert_\epsilon$ and neither is a moderated variable
(O_1)	$Z \approx_\alpha t \implies \bot$	(O_2)	$Z \not\approx_\alpha t \implies \top$	$Z \notin \mathtt{vars}(t)$ and $Z \neq t$

$\mathcal{R}_2$: Elimination of parameters and auxiliary unknowns.

$$(C_1)\ \forall \overline{Y}, Y : P \implies \forall \overline{Y} : P \quad (C_2)\ \exists \overline{W}, W : P \implies \exists \overline{W} : P \quad (C_3) \exists \overline{W}, W : W \approx_\alpha t \wedge P \implies \exists \overline{W} : P$$

$W \notin \mathtt{vars}(P, t)$ and $Y \notin \mathtt{vars}(P)$.

$\mathcal{R}_3$: Equality and freshness simplification

(E_1)	$\pi \cdot X \approx_\alpha \gamma \cdot X \implies \wedge\, \mathtt{ds}(\pi, \gamma)\#X$	(F_1)	$a\#\pi \cdot X \implies \pi^{-1}(a)\#X$
(E_2)	$[a]\,t \approx_\alpha [a]\,u \implies t \approx_\alpha u$	(F_2)	$a\#\,[a]\,t \implies \top$
(E_3)	$[a]\,t \approx_\alpha [b]\,u \implies (b\ a) \cdot t \approx_\alpha u \wedge b\#t$	(F_3)	$a\#\,[b]\,t \implies a\#t$
(E_4)	$f(\tilde{t}) \approx_\alpha f(\tilde{u}) \implies \wedge_i\, t_i \approx_\alpha u_i$	(F_4)	$a\#f(t_1, \ldots, t_n) \implies \wedge_i a\#t_i$

$\mathcal{R}_4$: Instantiation Rules

(I_1) $Z \approx_\alpha t \wedge P \implies Z \approx_\alpha t \wedge P[Z/t]$, where $Z \notin \mathtt{vars}(t)$ and Z is not a parameter.

(I_2) $\pi \cdot Z \approx_\alpha t \implies Z \approx_\alpha \pi^{-1} \cdot t$ (I_3) $\pi \cdot Z \not\approx_\alpha t \implies Z \not\approx_\alpha \pi^{-1} \cdot t, t$ is not a suspension.

$\mathcal{R}_5$: Simplification of Parameters

$$\begin{array}{ll}
(U_1) & \forall \overline{Y}, Y : P \wedge Y \not\approx_\alpha t \implies \bot \\
(U_2) & \forall \overline{Y} : P \wedge (Y \not\approx_\alpha t \vee Q) \implies \forall \overline{Y} : P \wedge Q[Y/t],\ \text{if } Y \notin \mathtt{vars}(t),\ Y \in \overline{Y} \\
(U_3) & \forall \overline{Y}, Y : P \wedge Y \approx_\alpha t \implies \bot,\ \text{if } Y \not\equiv t \\
(U_4) & \forall \overline{Y} : P \wedge (Z_1 \approx_\alpha t_1 \vee \cdots \vee Z_n \approx_\alpha t_n \vee Q) \implies \forall \overline{Y} : P \wedge Q \\
(U_5) & \forall \overline{Y}, Y : P \wedge a\#Y \implies \bot \\
(U_6) & \forall \overline{Y}, Y : P \wedge a\not\#Y \implies \bot
\end{array}$$

Conditions for (U_4)**:** (i) Z_i is a variable and $Z_i \not\equiv t_i$; (ii) each equation in the disjunction contains at least one occurrence of a parameter; (iii) Q does not contain any parameter.

$\mathcal{R}_6$: Terms Disunification

(DC)	$f(\tilde{t}) \not\approx_\alpha f(\tilde{u}) \implies \vee_i\, t_i \not\approx_\alpha u_i$	(NF_1)	$a\not\#\pi \cdot X \implies \pi^{-1}(a)\not\#X$
(D_1)	$\pi \cdot X \not\approx_\alpha \gamma \cdot X \implies \vee_i\, \mathtt{ds}(\pi, \gamma)\not\#X$	(NF_2)	$a\not\#\,[a]\,t \implies \bot$
(D_2)	$[a]\,t \not\approx_\alpha [a]\,u \implies t \not\approx_\alpha u$	(NF_3)	$a\not\#\,[b]\,t \implies a\not\#t$
(D_3)	$[a]\,t \not\approx_\alpha [b]\,u \implies (b\ a) \cdot t \not\approx_\alpha u \vee b\not\#t$	(NF_4)	$a\not\#f(\tilde{t}) \implies \vee_i a\not\#t_i$

$\mathcal{R}_7$: Explosion Rule

$$\exists \overline{W} \forall \overline{Y} : P \implies \exists W_1, \ldots, W_n, \overline{W} \forall \overline{Y} : P \wedge X \approx_\alpha f(W_1, \ldots, W_n)$$

Rule Conditions: (i) X is a free or existential variable occurring in P, $W_1, \ldots, W_n$ are newly chosen auxiliary variables not occurring anywhere in the problem, and $f \in \Sigma$; (ii) there exists an equation $X = u$ (or disequation $X \not\approx_\alpha u$) in P such that u is not a variable and contains at least one parameter; (iii) no other rule can be applied.

Successive application of rules gives:

$$\begin{aligned}
\mathcal{P}_1 &\overset{inst}{\Longrightarrow} \exists W_1 \forall Y : \lambda W_1 \not\approx_\alpha \lambda\, [a]\, Y \wedge X \approx_\alpha \lambda W_1 \\
&\overset{dec}{\Longrightarrow} \exists W_1 \forall Y : W_1 \not\approx_\alpha [a]\, Y \wedge X \approx_\alpha \lambda W_1 \\
&\overset{expl}{\Longrightarrow} \exists W_1 W_2 \forall Y : W_1 \not\approx_\alpha [a]\, Y \wedge X \approx_\alpha \lambda W_1 \wedge W_1 \approx_\alpha \lambda W_2 \\
&\overset{inst}{\Longrightarrow} \exists W_1 W_2 \forall Y : \lambda W_2 \not\approx_\alpha [a]\, Y \wedge X \approx_\alpha \lambda W_1 \wedge W_1 \approx_\alpha \lambda W_2 \\
&\overset{dis}{\Longrightarrow} \exists W_1 W_2 \forall Y : X \approx_\alpha \lambda W_1 \wedge W_1 \approx_\alpha \lambda W_2 \\
&\overset{pl,inst}{\overset{*}{\Longrightarrow}} \exists W_1 W_2 : X \approx_\alpha \lambda\lambda W_2 \wedge W_1 \approx_\alpha \lambda W_2.
\end{aligned}$$

Similarly, $\mathcal{P}_2 \overset{*}{\Longrightarrow} \exists W_1, W_2 : X = W_1 W_2$*. Notice that from this point one reaches a parameterless normal form. Solutions to* $\mathcal{P}$ *can be easily obtained by instantiating* W_2 *to any ground term in* $\mathcal{P}_1$ *and* W_1, W_2 *to any term in* $\mathcal{P}_2$ *since* X *only needs to be instantiated to a term headed by an application. It is easy to check that this choice indeed generates solutions for* $\mathcal{P}$*.*

About the Unification Type of $\mathbf{K}+\Box\Box\bot$

Philippe Balbiani[a] Çiğdem Gencer[a,b] Maryam Rostamigiv[a] Tinko Tinchev[c]

[a]Toulouse Institute of Computer Science Research
CNRS — Toulouse University, Toulouse, France
[b]Faculty of Arts and Sciences
Istanbul Aydın University, Istanbul, Turkey
[c]Faculty of Mathematics and Informatics
Sofia University St. Kliment Ohridski, Sofia, Bulgaria

1 Introduction

The unification problem in a propositional logic is to determine, given a formula φ, whether there exists a substitution σ such that $\sigma(\varphi)$ is in that logic [1]. In that case, σ is a unifier of φ. When a unifiable formula has minimal complete sets of unifiers, it is either infinitary, finitary, or unitary, depending on the cardinality of its minimal complete sets of unifiers. Otherwise, it is nullary. Within the context of elementary unification, it is known that $\mathbf{Alt}_1$ is nullary [8], $\mathbf{S}5$ and $\mathbf{S}4.3$ are unitary [10, 11, 12], transitive modal logics like $\mathbf{K}4$ and $\mathbf{S}4$ are finitary [13, 15], $\mathbf{KD}45$, $\mathbf{K}45$ and $\mathbf{K}4.2^+$ are unitary [14, 16], $\mathbf{K}$ is nullary [17] and $\mathbf{K}4\mathbf{D}1$ is unitary [18]. The unification types of the description logics $\mathcal{E}L$ and $\mathcal{F}L_0$ are known too: both of them are nullary [2, 3]. In this paper, we prove that in modal logic $\mathbf{K}+\Box\Box\bot$ — the least normal modal logic containing the formula $\Box\Box\bot$ — unifiable formulas are either unitary, or finitary[1].

2 Preliminaries

Let S be a finite set. We will write $\|S\|$ for the cardinality of S. If S is non-empty then for all equivalence relations $\sim$ on S and for all $T\subseteq S$, $T/\sim$ will denote the quotient set of T modulo $\sim$.

Proposition 1. *Let T be a finite set. If S is non-empty then for all equivalence relations $\sim$ on S, $\|S/\sim\|\leq\|T\|\leq\|S\|$ iff there exists a surjective function f from S to T such that for all $\alpha,\beta\in S$, if $f(\alpha)=f(\beta)$ then $\alpha\sim\beta$.*

Proposition 1 will be used twice in the proof of Proposition 11.

3 Syntax

Let $\mathbf{VAR}$ be a countably infinite set of *variables* (with typical members denoted x, y, etc). Let $(x_1,x_2,\ldots)$ be an enumeration of $\mathbf{VAR}$ without repetitions. Let $n\geq1$. The set $\mathbf{FOR}_n$ of all n*-formulas* (with typical members denoted φ, ψ, etc) is inductively defined by:

- $\varphi,\psi ::= x_i \mid \bot \mid \neg\varphi \mid (\varphi\vee\psi) \mid \Box\varphi$ where $i\in\{1,\ldots,n\}$.

We adopt the standard rules for omission of the parentheses. The connectives $\top$, $\wedge$, $\rightarrow$ and $\leftrightarrow$ are defined by the usual abbreviations. We have also a connective $\Diamond$ which is defined by $\Diamond\varphi ::= \neg\Box\neg\varphi$. For all $\varphi\in\mathbf{FOR}_n$, we respectively write "φ^0" and "φ^1" to mean "$\neg\varphi$" and "φ". From now on,

[1]**Acknowledgements:** The preparation of this paper has been supported by *Bulgarian Science Fund* (Project *DN02/15/19.12.2016*) and *Université Paul Sabatier* (Programme *Professeurs invités 2018*). We are indebted to Silvio Ghilardi for his suggestion to consider the question of the unification type of $\mathbf{K}+\Box\Box\bot$.

$\boxed{\textbf{we write "}\mathbf{L}_2\textbf{" to mean "}\mathbf{K}+\Box\Box\bot\textbf{".}}$

Let $\equiv_n$ be the equivalence relation on $\mathbf{FOR}_n$ defined by:

- $\varphi{\equiv_n}\psi$ iff $\varphi\leftrightarrow\psi{\in}\mathbf{L}_2$.

Proposition 2. *$\equiv_n$ possesses finitely many equivalence classes.*

An *n-substitution* is a couple (k,σ) where $k{\geq}1$ and σ is a homomorphism from $\mathbf{FOR}_n$ to $\mathbf{FOR}_k$. Let $\mathbf{SUB}_n$ be the set of all n-substitutions. The equivalence relation $\simeq_n$ on $\mathbf{SUB}_n$ is defined by:

- $(k,\sigma){\simeq_n}(l,\tau)$ iff for all $i{\in}\{1,\ldots,n\}$, $\sigma(x_i)\leftrightarrow\tau(x_i){\in}\mathbf{L}_2$.

The preorder $\preccurlyeq_n$ on $\mathbf{SUB}_n$ is defined by:

- $(k,\sigma){\preccurlyeq_n}(l,\tau)$ iff there exists a k-substitution (m,υ) such that for all $i{\in}\{1,\ldots,n\}$, $\upsilon(\sigma(x_i))\leftrightarrow\tau(x_i){\in}\mathbf{L}_2$.

4 Semantics

Let $n{\geq}1$. An *n-tuple of bits* (denoted α, β, etc) is a function from $\{1,\ldots,n\}$ to $\{0,1\}$. Such function should be understood as a propositional valuation of the variables $x_1,\ldots,x_n$: for all $i{\in}\{1,\ldots,n\}$, if $\alpha_i{=}0$ then it is interpreted to mean "x_i is false" else it is interpreted to mean "x_i is true". Let $\mathbf{BIT}_n$ be the set of all n-tuples of bits. An *n-model* is a structure of the form (α,S) where $\alpha{\in}\mathbf{BIT}_n$ and $S{\subseteq}\mathbf{BIT}_n$. Such structure should be understood as a tree-like Kripke model of depth at most 1: α is the valuation of its root node and S is the set of the valuations of its non-root nodes. Let $\mathbf{MOD}_n$ be the set of all n-models. We shall say that an n-model (α,S) is *degenerated* if $S{=}\emptyset$. Let $\mathbf{MOD}_n^{\mathsf{deg}}$ be the set of all degenerated n-models. Notice that $\|\mathbf{MOD}_n^{\mathsf{deg}}\|{=}2^n$. Notice also that for all sets S of n-tuples of bits, $S\times\{\emptyset\}$ is a set of degenerated n-models. The binary relation $\models_n$ of *n-satisfiability* between $\mathbf{MOD}_n$ and $\mathbf{FOR}_n$ is defined as expected. In particular,

- $(\alpha,S){\models_n}x_i$ iff $\alpha_i{=}1$ where $i{\in}\{1,\ldots,n\}$,
- $(\alpha,S){\models_n}\Box\varphi$ iff for all $\beta{\in}S$, $(\beta,\emptyset){\models_n}\varphi$.

As a result, $(\alpha,S){\models_n}\Diamond\varphi$ iff there exists $\beta{\in}S$ such that $(\beta,\emptyset){\models_n}\varphi$.

Proposition 3. *For all $\varphi{\in}\mathbf{FOR}_n$, $\varphi{\in}\mathbf{L}_2$ iff for all $(\alpha,S){\in}\mathbf{MOD}_n$, $(\alpha,S){\models_n}\varphi$.*

For all $\alpha{\in}\mathbf{BIT}_n$, the n-formula

- $\bar{x}^\alpha{=}\bigwedge\{x_i^{\alpha_i}:\ i{\in}\{1,\ldots,n\}\}$

exactly characterizes the propositional valuation represented by α. For all $(\alpha,S){\in}\mathbf{MOD}_n$, the n-formula

- $\mathbf{for}_n(\alpha,S){=}\bar{x}^\alpha\wedge\Box\bigvee\{\bar{x}^\gamma:\ \gamma{\in}S\}\wedge\bigwedge\{\Diamond\bar{x}^\gamma:\ \gamma{\in}S\}$

exactly characterizes the tree-like Kripke model of depth at most 1 represented by (α,S).

Proposition 4. *Let $(\alpha,S),(\beta,T){\in}\mathbf{MOD}_n$. The following conditions are equivalent: **(i)** $(\alpha,S){=}(\beta,T)$; **(ii)** $(\alpha,S){\models_n}\mathbf{for}_n(\beta,T)$.*

Proposition 5. *Let $(k,\sigma){\in}\mathbf{SUB}_n$. For all $(\alpha,S){\in}\mathbf{MOD}_k$, there exists $(\beta,T){\in}\mathbf{MOD}_n$ such that $(\alpha,S){\models_k}\sigma(\mathbf{for}_n(\beta,T))$.*

Proposition 6. *Let $(k,\sigma)\in\mathbf{SUB}_n$. Let $(\alpha,S)\in\mathbf{MOD}_k$. For all $(\beta,T),(\gamma,U)\in\mathbf{MOD}_n$, if $(\alpha,S)\models_k\sigma(\mathbf{for}_n(\beta,T))$ and $(\alpha,S)\models_k\sigma(\mathbf{for}_n(\gamma,U))$ then $(\beta,T)=(\gamma,U)$.*

For all $k\geq 1$, a (k,n)-*morphism* is a function f from $\mathbf{MOD}_k$ to $\mathbf{MOD}_n$ such that for all $(\alpha,S)\in\mathbf{MOD}_k$ and for all $(\beta,T)\in\mathbf{MOD}_n$, if $f(\alpha,S)=(\beta,T)$ then[2]

forward condition: for all $\gamma\in S$, there exists $\delta\in T$ such that $f(\gamma,\emptyset)=(\delta,\emptyset)$,

backward condition: for all $\delta\in T$, there exists $\gamma\in S$ such that $f(\gamma,\emptyset)=(\delta,\emptyset)$.

Proposition 7. *Let $k\geq 1$. Let f be a (k,n)-morphism. Let $(\beta,T)\in\mathbf{MOD}_k$ and $(\gamma,U)\in\mathbf{MOD}_n$. If $f(\beta,T)=(\gamma,U)$ then the following conditions hold: **(i)** the image by f of $T\times\{\emptyset\}$ is equal to $U\times\{\emptyset\}$; **(ii)** $T=\emptyset$ iff $U=\emptyset$.*

Proposition 8. *Let $k\geq 1$. Let f be a (k,n)-morphism. Let $(\beta,T)\in\mathbf{MOD}_k$ and $(\gamma,U)\in\mathbf{MOD}_n$. If the following conditions hold then $f(\beta,T)=(\gamma,U)$: **(i)** $f(\beta,T)\models_n\bar{x}^{\gamma}$; **(ii)** for all $\delta\in T$, there exists $\epsilon\in U$ such that $f(\delta,\emptyset)=(\epsilon,\emptyset)$; **(iii)** for all $\epsilon\in U$, there exists $\delta\in T$ such that $f(\delta,\emptyset)=(\epsilon,\emptyset)$.*

5 Unification

Let $n\geq 1$. An n-*unifier* of $\varphi\in\mathbf{FOR}_n$ is an n-substitution (k,σ) such that $\sigma(\varphi)\in\mathbf{L}_2$. We shall say that $\varphi\in\mathbf{FOR}_n$ is n-*unifiable* if there exists an n-unifier of φ. We shall say that a set Σ of n-unifiers of an n-unifiable $\varphi\in\mathbf{FOR}_n$ is n-*complete* if for all n-unifiers (k,σ) of φ, there exists $(l,\tau)\in\Sigma$ such that $(l,\tau)\preccurlyeq_n(k,\sigma)$. As is well-known, for all $\varphi\in\mathbf{FOR}_n$, if φ is n-unifiable then for all minimal n-complete sets Σ,Δ of n-unifiers of φ, Σ and Δ have the same cardinality. Then, an important question is the following: when $\varphi\in\mathbf{FOR}_n$ is n-unifiable, is there a minimal n-complete set of n-unifiers of φ? When the answer is "yes", how large is this set? For all n-unifiable $\varphi\in\mathbf{FOR}_n$, we shall say that:

- φ is n-*nullary* if there exists no minimal complete set of unifiers of φ,
- φ is n-*infinitary* if there exists a minimal complete set of unifiers of φ with infinite cardinality,
- φ is n-*finitary* if there exists a minimal complete set of unifiers of φ with finite cardinality ≥ 2,
- φ is n-*unitary* if there exists a minimal complete set of unifiers of φ with cardinality 1.

Proposition 9. *The n-unifiable n-formula $\Diamond x_1\rightarrow\Box x_1$ is n-finitary.*

For all n-unifiable $\varphi\in\mathbf{FOR}_n$ and for all $\pi\geq 1$, we shall say that φ is n-π-*reasonable* if for all n-unifiers (k,σ) of φ, if $k\geq\pi$ then there exists an n-unifier (l,τ) of φ such that $(l,\tau)\preccurlyeq_n(k,\sigma)$ and $l\leq\pi$.

Proposition 10. *Let $\varphi\in\mathbf{FOR}_n$ be n-unifiable and $\pi\geq 1$. If φ is n-π-reasonable then φ is either n-finitary, or n-unitary.*

6 Main results

Let $n\geq 1$.

Proposition 11. *Let $k\geq n$. For all (k,n)-morphisms g, there exists a surjective (k,n)-morphism f such that for all $(\alpha,S),(\beta,T)\in\mathbf{MOD}_k$, if $f(\alpha,S)=f(\beta,T)$ then $g(\alpha,S)=g(\beta,T)$.*

[2]The morphisms described here should not be mistaken for the bounded morphisms usually considered in modal logic [9, Definition 2.10]. In particular, in the above definition, there is no condition related to the propositional valuation of the variables.

Proof. Let g be a (k,n)-morphism. Let $\sim_k$ be the equivalence relation on $\mathbf{MOD}_k$ defined by:

- $(\alpha,S)\sim_k(\beta,T)$ iff $g(\alpha,S)=g(\beta,T)$.

Lemma 1. *1.* $\|\mathbf{MOD}_k^{\mathsf{deg}}/\sim_k\|\leq\|\mathbf{MOD}_n^{\mathsf{deg}}\|$,

2. $\|\mathbf{MOD}_n^{\mathsf{deg}}\|\leq\|\mathbf{MOD}_k^{\mathsf{deg}}\|$.

Hence, by Proposition 1, there exists a surjective function f^{deg} from $\mathbf{MOD}_k^{\mathsf{deg}}$ to $\mathbf{MOD}_n^{\mathsf{deg}}$ such that for all $(\alpha,\emptyset),(\beta,\emptyset)\in\mathbf{MOD}_k^{\mathsf{deg}}$, if $f^{\mathsf{deg}}(\alpha,\emptyset)=f^{\mathsf{deg}}(\beta,\emptyset)$ then $(\alpha,\emptyset)\sim_k(\beta,\emptyset)$.

Lemma 2. *For all non-empty sets S,T of k-tuples of bits, if the images by f^{deg} of $S\times\{\emptyset\}$ and $T\times\{\emptyset\}$ are equal then the images by g of $S\times\{\emptyset\}$ and $T\times\{\emptyset\}$ are equal.*

For all non-empty sets E of n-tuples of bits, let

- $f^{\circ}(E)$ be the set of all $(\alpha,S)\in\mathbf{MOD}_k\setminus\mathbf{MOD}_k^{\mathsf{deg}}$ such that the image by f^{deg} of $S\times\{\emptyset\}$ is equal to $E\times\{\emptyset\}$,
- $f^{\bullet}(E)$ be the set of all $(\alpha,S)\in\mathbf{MOD}_n\setminus\mathbf{MOD}_n^{\mathsf{deg}}$ such that $S=E$.

Notice that since f^{deg} is surjective, therefore $\|f^{\circ}(E)\|\geq2^k$. Notice also that $\|f^{\bullet}(E)\|=2^n$.

Lemma 3. *For all non-empty sets E of n-tuples of bits,*

1. $\|f^{\circ}(E)/\sim_k\|\leq\|f^{\bullet}(E)\|$,

2. $\|f^{\bullet}(E)\|\leq\|f^{\circ}(E)\|$.

Thus, for all non-empty sets E of n-tuples of bits, by Proposition 1, there exists a surjective function f^E from $f^{\circ}(E)$ to $f^{\bullet}(E)$ such that for all $(\alpha,S),(\beta,T)\in f^{\circ}(E)$, if $f^E(\alpha,S)=f^E(\beta,T)$ then $(\alpha,S)\sim_k(\beta,T)$. Let f be the function from $\mathbf{MOD}_k$ to $\mathbf{MOD}_n$ such that for all $(\alpha,\emptyset)\in\mathbf{MOD}_k^{\mathsf{deg}}$,

- $f(\alpha,\emptyset)=f^{\mathsf{deg}}(\alpha,\emptyset)$

and for all $(\alpha,S)\in\mathbf{MOD}_k\setminus\mathbf{MOD}_k^{\mathsf{deg}}$, E being the non-empty set of n-tuples of bits such that the image by f^{deg} of $S\times\{\emptyset\}$ is equal to $E\times\{\emptyset\}$,

- $f(\alpha,S)=f^E(\alpha,S)$.

Lemma 4. *f is surjective.*

Lemma 5. *f is a (k,n)-morphism.*

Lemma 6. *For all $(\alpha,S),(\beta,T)\in\mathbf{MOD}_k$, if $f(\alpha,S)=f(\beta,T)$ then $g(\alpha,S)=g(\beta,T)$.*

This finishes the proof of Proposition 11. □

Proposition 12. *For all $\varphi\in\mathbf{FOR}_n$, if φ is n-unifiable then φ is n-n-reasonable.*

Proof. Let $\varphi\in\mathbf{FOR}_n$. Suppose φ is n-unifiable. Let (k,σ) be an n-unifier of φ such that $k\geq n$. Hence, $\sigma(\varphi)\in\mathbf{L}_2$. Let g be the function from $\mathbf{MOD}_k$ to $\mathbf{MOD}_n$ such that for all $(\alpha,S)\in\mathbf{MOD}_k$,

- $g(\alpha,S)$ is the $(\beta,T)\in\mathbf{MOD}_n$ such that $(\alpha,S)\models_k\sigma(\mathbf{for}_n(\beta,T))$.

Notice that by Propositions 5 and 6, g is well-defined.

Lemma 7. *g is a (k,n)-morphism.*

Lemma 8. *For all* $(\alpha,S),(\beta,T)\in\mathbf{MOD}_k$*, if* $g(\alpha,S)=g(\beta,T)$ *then for all* $i\in\{1,\ldots,n\}$*,* $(\alpha,S)\models_k\sigma(x_i)$ *iff* $(\beta,T)\models_k\sigma(x_i)$*.*

Since $k\geq n$ therefore by Proposition 11 and Lemma 7, let f be a surjective (k,n)-morphism such that for all $(\alpha,S),(\beta,T)\in\mathbf{MOD}_k$, if $f(\alpha,S)=f(\beta,T)$ then $g(\alpha,S)=g(\beta,T)$. Let (n,τ), (k,ν) be the n-substitutions defined by:

- $\tau(x_i)=\bigvee\{\mathbf{for}_n(f(\alpha,S)):\ (\alpha,S)\in\mathbf{MOD}_k$ is such that $(\alpha,S)\models_k\sigma(x_i)\}$ where $i\in\{1,\ldots,n\}$,
- $\nu(x_i)=\bigvee\{\mathbf{for}_k(\alpha,S):\ (\alpha,S)\in\mathbf{MOD}_k$ is such that $f(\alpha,S)\models_n x_i\}$ where $i\in\{1,\ldots,n\}$.

Lemma 9. *Let* $\psi\in\mathbf{FOR}_n$*. For all* $(\beta,T)\in\mathbf{MOD}_n$*, the following conditions are equivalent:* ***(i)*** *there exists* $(\alpha,S)\in\mathbf{MOD}_k$ *such that* $f(\alpha,S)=(\beta,T)$ *and* $(\alpha,S)\models_k\sigma(\psi)$*;* ***(ii)*** *for all* $(\alpha,S)\in\mathbf{MOD}_k$*, if* $f(\alpha,S)=(\beta,T)$ *then* $(\alpha,S)\models_k\sigma(\psi)$*;* ***(iii)*** $(\beta,T)\models_n\tau(\psi)$*.*

Lemma 10. *For all* $(\beta,T)\in\mathbf{MOD}_k$ *and for all* $i\in\{1,\ldots,n\}$*,* $(\beta,T)\models_k\nu(x_i)$ *iff* $f(\beta,T)\models_n x_i$*.*

Lemma 11. *Let* $(\beta,T)\in\mathbf{MOD}_k$ *and* $(\gamma,U)\in\mathbf{MOD}_n$*. The following conditions are equivalent:* ***(i)*** $f(\beta,T)=(\gamma,U)$*;* ***(ii)*** $(\beta,T)\models_k\nu(\mathbf{for}_n(\gamma,U))$*.*

Lemma 12. *For all* $(\beta,T)\in\mathbf{MOD}_k$ *and for all* $i\in\{1,\ldots,n\}$*,* $(\beta,T)\models_k\nu(\tau(x_i))$ *iff* $(\beta,T)\models_k\sigma(x_i)$*.*

Since $\sigma(\varphi)\in\mathbf{L}_2$, therefore by Proposition 3, for all $(\alpha,S)\in\mathbf{MOD}_k$, $(\alpha,S)\models_k\sigma(\varphi)$. Thus, by Lemma 9, for all $(\beta,T)\in\mathbf{MOD}_n$, $(\beta,T)\models_n\tau(\varphi)$. Consequently, by Proposition 3, $\tau(\varphi)\in\mathbf{L}_2$. Hence, (n,τ) is an n-unifier of φ. Since by Lemma 12, $(n,\tau)\preccurlyeq_n(k,\sigma)$, therefore φ is n-n-reasonable. $\Box$

Theorem 1. *For all* $\varphi\in\mathbf{FOR}_n$*, if* φ *is* n*-unifiable then* φ *is either* n*-finitary, or* n*-unitary.*

7 Conclusion

In this paper, within the context of elementary unification, we have proved Theorem 1 asserting that in $\mathbf{K}+\Box\Box\bot$, unifiable formulas are either finitary, or unitary. We believe that in our line of reasoning, the main properties of $\mathbf{K}+\Box\Box\bot$ are the ones given in Propositions 2, 5 and 6. Proposition 2 says that $\mathbf{K}+\Box\Box\bot$ is locally tabular[3] — it is used in the proof of Proposition 10. Propositions 5 and 6 give us the possibility to define the function g — they are used in the proof of Proposition 12. Notice that Theorem 1 is an immediate consequence of Propositions 10 and 12. Here are open questions:

1. determine the unification type of the locally tabular modal logic $\mathbf{K}+\Box^d\bot$ for each $d\geq3$,
2. determine the unification types of other locally tabular modal logics like the ones studied in [19, 20, 21],
3. determine the unification types of the modal logics $\mathbf{KB}$, $\mathbf{KD}$ and $\mathbf{KT}$.

We conjecture that within the context of elementary unification, the modal logics mentioned in Items 1 and 2 are either finitary, or unitary. As for the modal logics considered in Item 3, it is only known that $\mathbf{KD}$ and $\mathbf{KT}$ are not unitary within the context of elementary unification and $\mathbf{KB}$, $\mathbf{KD}$ and $\mathbf{KT}$ are nullary within the context of unification with parameters [4, 5, 6].

[3]A modal logic $\mathbf{L}$ is *locally tabular* if for all $n\geq1$, the equivalence relation $\equiv_n$ on $\mathbf{FOR}_n$ defined by

- $\varphi\equiv_n\psi$ iff $\varphi\leftrightarrow\psi\in\mathbf{L}$

possesses finitely many equivalence classes. The most popular of all locally tabular modal logics is probably $\mathbf{S}5$. See [19, 20, 21] for other examples of locally tabular modal logics.

References

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

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

[3] BAADER, F. and P. NARENDRAN, 'Unification of concept terms in description logics', *Journal of Symbolic Computation* **31** (2001) 277–305.

[4] BALBIANI, P., 'Remarks about the unification type of several non-symmetric non-transitive modal logics', *Logic Journal of the IGPL* **27** (2019) 639–658.

[5] BALBIANI, P. and Ç. GENCER, '$\mathbf{KD}$ is nullary', *Journal of Applied Non-Classical Logics* **27** (2017) 196–205.

[6] BALBIANI, P. and Ç. GENCER, 'About the unification type of modal logics between $\mathbf{KB}$ and $\mathbf{KTB}$', Studia Logica (2019) https://doi.org/10.1007/s11225-019-09883-0.

[7] BALBIANI, P., Ç. GENCER, M. ROSTAMIGIV and T. TINCHEV, 'About the unification types of the modal logics determined by classes of deterministic frames', arXiv:2004.07904v1 [cs.LO].

[8] BALBIANI, P. and T. TINCHEV, 'Unification in modal logic $\mathbf{Alt}_1$', In: *Advances in Modal Logic,* College Publications (2016) 117–134.

[9] BLACKBURN, P., M. DE RIJKE and Y. VENEMA, *Modal Logic,* Cambridge University Press (2001).

[10] DZIK, W., 'Unitary unification of $\mathbf{S}5$ modal logics and its extensions', *Bulletin of the Section of Logic* **32** (2003) 19–26.

[11] DZIK, W., *Unification Types in Logic,* Wydawnicto Uniwersytetu Slaskiego (2007).

[12] DZIK, W. and P. WOJTYLAK, 'Projective unification in modal logic', *Logic Journal of the IGPL* **20** (2012) 121–153.

[13] GHILARDI, S., 'Best solving modal equations', *Annals of Pure and Applied Logic* **102** (2000) 183–198.

[14] GHILARDI, S. and L. SACCHETTI, 'Filtering unification and most general unifiers in modal logic', *Journal of Symbolic Logic* **69** (2004) 879–906.

[15] IEMHOFF, R., 'A syntactic approach to unification in transitive reflexive modal logics', *Notre Dame Journal of Formal Logic* **57** (2016) 233–247.

[16] JEŘÁBEK, E., 'Logics with directed unification', In: *Algebra and Coalgebra meet Proof Theory,* Workshop at Utrecht University (2013).

[17] JEŘÁBEK, E., 'Blending margins: the modal logic $\mathbf{K}$ has nullary unification type', *Journal of Logic and Computation* **25** (2015) 1231–1240.

[18] KOST, S., 'Projective unification in transitive modal logics', *Logic Journal of the IGPL* **26** (2018) 548–566.

[19] MIYAZAKI, Y., 'Normal modal logics containing $\mathbf{KTB}$ with some finiteness conditions', In: *Advances in Modal Logic,* College Publications (2004) 171–190.

[20] NAGLE, M. and S. THOMASON, 'The extensions of the modal logic $\mathbf{K}5$', *Journal of Symbolic Logic* **50** (1985) 102–109.

[21] SHAPIROVSKY, I. and V. SHEHTMAN, 'Local tabularity without transitivity', In: *Advances in Modal Logic,* College Publications (2016) 520–534.

A,C and AC Nominal Anti-Unification

Alexander Baumgartner[1] and Daniele Nantes-Sobrinho[2]

[1] Instituto de Ciencias de la Ingeniería, Universidad de O'Higgins, Chile
`alexander.baumgartner@uoh.cl`
[2] Departamento de Matemática, Universidade de Brasília, Brazil
`dnantes@mat.unb.br`

Abstract

This is a work in progress on nominal anti-unification modulo associative (A), commutative (C) and associative-commutative (AC) equational theories. A rule-based algorithm to solve nominal anti-unification problems is obtained by extending the existing algorithm with rules dedicated to the equational part. Also, we have extended the algorithm which deals with nominal equivariance to take into account the theories involved. This is a first step towards the investigation of nominal anti-unification problems modulo equational theories as well as its relation to equational higher-order pattern anti-unification.

1 Introduction

In general, the anti-unification problem is concerned with finding a generalization of two given input terms. More specifically, the nominal anti-unification problem for two nominal terms t_1 and t_2 and a freshness context ∇ is concerned with finding a term t that is more general than the original ones under consideration of ∇. That is, there exist substitutions σ_1 and σ_2 such that $\nabla \vdash t_1 \approx t\sigma_1$ and $\nabla \vdash t_2 \approx t\sigma_2$. The interesting generalizations are the least general ones, which retain the common structure of t_1 and t_2 as much as possible. In [1] the authors investigated the problem of computing least general generalizations (lgg) for nominal terms-in-context, i.e., pairs of the form $\langle \nabla, s \rangle$, where ∇ is a freshness context and s is a nominal term. The authors have identified that, in general, without restricting the set of atoms permitted in the generalization to be finite, there is no lgg for terms-in-context. Even more, a minimal complete set of generalizations does not exist.

In the context of nominal equation solving, extensions of nominal unification with recursive let [2], with atom variables [3], modulo equational theories [4, 5, 6, 7] were investigated. In particular, for equational theories involving commutative operators it was identified that the nominal unification type was not unitary anymore and an approach using fixed point constraints was proposed in [7]. Therefore it is natural to investigate such extensions for nominal anti-unification and also check how equational theories can affect the existing nominal anti-unification framework.

In this work we investigate the extension of the current nominal anti-unification algorithm [1], by proposing rules that can deal with the equational theories for associativity (A), commutativity (C) and associtivity-commutativity (AC). The preliminary results were obtained exploring the fact that there is a reduction from nominal unification to higher-order pattern unification (HOPU) [8], where higher-order patterns [9] are λ-terms whose free variables may only be applied to a sequence of distinct bound variables. And also the existing extension of the HOPU algorithm [10] which takes into account associative (A), commutative (C) and associative-commutative (AC) symbols [11]. Therefore, it was not surprising that their rules could be adapted to our problem. In addition, we had to extend the algorithm for solving the *equivariance problem* with rules dedicated to the aforementioned equational theories.

2 Preliminaries

We assume the reader is familiar with the nominal syntax and only recall the main concepts and notations that are needed in the paper; for more details we refer the reader to [12, 13].

Nominal Syntax. Fix a countable infinite set of *variables* $\mathcal{X} = \{X, Y, Z, \ldots\}$ and a countable infinite set of *atoms* $\mathcal{A} = \{a, b, c, \ldots\}$, such that $\mathcal{X} \cap \mathcal{A} = \emptyset$. Variables represent meta-level *unknowns* and atoms *object-level* variable symbols. Atoms are identified by their name, so it will be redundant to say two atoms a and b are different. A signature Σ is a set of *term-formers* (disjoint from $\mathcal{A}$ and $\mathcal{X}$) such that to each $f \in \Sigma$ is assigned a unique non-negative integer n, called the *arity* of f , written as $f : n$. A *permutation* π is a bijection $\mathcal{A} \to \mathcal{A}$ with finite domain, i.e., the set $supp(\pi) := \{a \in A \mid \pi(a) \neq a\}$ is finite. Id denotes the identity permutation. The composition of two permutations π and π' will be denoted as $\pi \circ \pi'$.

Nominal terms are given by the following grammar: $s, t ::= a \mid \; \pi \cdot X \mid \; a.t \mid \; f^E(t_1 \ldots t_n)$, where a is an *atom*, $\pi \cdot X$ is a *suspension* (or moderated variable), $a.t$ denotes the *abstraction* of atom a in the term t, and $f^E(t_1, \ldots t_n)$ is a *function application*, where $f^E \in \Sigma$ and $f^E : n$. The parameter E describes the equational theory assigned to the function symbol f, which can be $\emptyset$, A,C and AC. In the case we have $f^{\emptyset}$, we will omit the superscript, writing only f.

The *action of a permutation* π on a term t is defined by the structure of t as follows: $\pi \bullet a = \pi(a)$, $\pi \cdot (\pi' \cdot X) = (\pi \circ \pi') \cdot X$, ,$\pi \bullet (a.t) = \pi(a).(\pi \bullet t)$ and $\pi \bullet f(t_1, \ldots, t_n) = f(\pi \bullet t_1, \ldots, \pi \bullet t_n)$. The difference set of π and π' is defined as $ds(\pi, \pi') = \{a \mid \pi \bullet a \neq \pi' \bullet a\}$.

Substitutions, range over $\sigma, \gamma, \ldots$, are defined as maps from variables to nominal terms with finite domain. The *action of a substitution* σ on a term t, denoted as $t\sigma$, is inductively defined by: $a\sigma = a, (\pi \cdot X)\sigma = \pi \cdot (X\sigma)$, $(a.t)\sigma = a.(t\sigma)$ and $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$. Substitutions are written in postfix notation: $t(\sigma\gamma) = (t\sigma)\gamma$.

Equality and Freshness Constraints. In the nominal theory freshness constraints are used to avoid the problem of capturing free atoms. A *freshness constraint* is a pair of the form $a\#X$ and its intended meaning is that a cannot occur as a free atom in an instantiation of X. A finite set of freshness constraints is called a *freshness context*, and we use ∇ and Γ to denote them. An *α-equality constraint* is a pair of the form $s \approx t$ and means that s is α-equivalent to t. The notion of α-equivalence between two nominal terms, denoted by $\approx$, and the freshness predicate $\#$ are defined by the rules in Figure 1.

$$\frac{}{\nabla \vdash a \approx a}\,(\text{at}) \qquad \frac{ds(\pi, \pi')\#X \in \nabla}{\nabla \vdash \pi \cdot X \approx \pi' \cdot X}\,(\text{var}) \qquad \frac{\nabla \vdash t \approx t'}{\nabla \vdash a.t \approx a.t'}\,(\text{abs-a})$$

$$\frac{\nabla \vdash t \approx (a\,a') \bullet t' \quad \nabla \vdash a\#t'}{\nabla \vdash a.t \approx a'.t'}\,(\text{abs-b}) \qquad \frac{\nabla \vdash t_i \approx t'_i}{\nabla \vdash f(t_1, \ldots t_n) \approx f(t'_1, \ldots, t'_n)}\,(\text{f}) \qquad \frac{}{\nabla \vdash a\#a'}\,(\#\text{at})$$

$$\frac{(\pi^{-1} \bullet a\#X) \in \nabla}{\nabla \vdash a\#\pi \cdot X}\,(\#\text{var}) \qquad \frac{}{\nabla \vdash a\#a.t}\,(\#\text{abs-a}) \qquad \frac{\nabla \vdash a\#t}{\nabla \vdash a\#a'.t}\,(\#\text{abs-b}) \qquad \frac{\nabla \vdash a\#t_i}{\nabla \vdash a\#f(t_1, \ldots t_n)}\,(\#\text{f})$$

Figure 1: Equality and Freshness Rules. In rules f and $\#f$: $i = 1 \ldots n$

Equality Constraints Modulo A, C, and AC. We define the predicates, $\approx_A$, $\approx_C$, and $\approx_{AC}$, for α-equivalence modulo A, C, and AC, respectively. The set of rules for each theory $E \in$ {A,C,AC} is obtained by adding the specific rule (E) in Figure 2, to the set of equality rules in Figure 1,replacing $\approx$ by the corresponding $\approx_E$. Note that we will omit the f^A and f^{AC} from the right premise in the rules (A) and (AC), if $n = 2$. For instance, instead of $f^A(t_2) \approx_A f^A(s_2)$ we simply get $t_2 \approx_A s_2$. For more details, we refer the reader to [4].

$$\frac{\nabla \vdash t_1 \approx_A s_1 \quad \nabla \vdash f^A(t_2,\dots,t_n) \approx_A f^A(s_2,\dots,s_n)}{\nabla \vdash f^A(t_1,\dots,t_n) \approx_A f^A(s_1,\dots,s_n)}\text{(A)} \qquad \frac{\nabla \vdash t_1 \approx_C s_i \quad \nabla \vdash t_2 \approx_C s_j}{\nabla \vdash f^C(t_1,t_2) \approx_C f^C(s_1,s_2)}\text{(C)}$$

$$\frac{\nabla \vdash t_1 \approx_{AC} s_k \quad \nabla \vdash f^{AC}(t_2,\dots,t_n) \approx_{AC} f^{AC}(s_1,\dots,s_{k-1},s_{k+1},\dots,s_n)}{\nabla \vdash f^{AC}(t_1,\dots,t_n) \approx_{AC} f^{AC}(s_1,\dots,s_n)}\text{(AC)}$$

Figure 2: Equational Rules. Where $n \geq 2$, $i = 1,2$ and $j = (i \mod 2) + 1$

3 Nominal Anti-Unification

In this section we recall the anti-unification algorithm proposed in [1] and propose an extension to it which takes into account the equational theories A, C and AC.

A (nominal) *anti-unification equation* is a triple of the form $X : t \triangleq s$ where X is a variable, called the *generalization variable*, and t, s are nominal terms. A (nominal) *anti-unification problem* P is a finite set of anti-unification equations.

3.1 NAU: nominal anti-unification algorithm

The nominal anti-unification algorithm for the empty theory was introduced in [1] and operates on tuples of the form $P;\ S;\ \Gamma;\ \sigma$ where: P denotes the set of unsolved anti-unification equations; S denotes the set of solved anti-unification equations; Γ denotes the freshness context computed so far; and σ is a substitution that holds the generalized term. Furthermore, there are two global parameters, a finite set $\mathcal{A}$ of atoms (the ones that are allowed in the generalization) and a freshness context ∇. The symbol $\uplus$ denotes the disjoint union operation.

Dec : **Decomposition** $\{X : \mathsf{h}(t_1,...,t_m) \triangleq \mathsf{h}(s_1,...,s_m)\} \uplus P; S; \Gamma; \sigma \Longrightarrow \bigcup_{i=1}^{m}\{Y_i : t_i \triangleq s_i\} \cup P; S; \Gamma; \sigma\{X/\mathsf{h}(Y_1,\dots,Y_m)\}$ where $\mathsf{h} \in \Sigma \cup A, Y_1,\dots,Y_m$ are fresh variables of the corresponding sorts, $m \geq 0$.
Abs : **Abstraction** $\{X : a.t \triangleq b.s\} \uplus P; S; \Gamma; \sigma \Longrightarrow \{Y : (c\,a) \bullet t \triangleq (c\,b) \bullet s\} \cup P; S; \Gamma; \sigma\{X/c.Y\}$, where Y is fresh , $c \in \mathcal{A}, \nabla \vdash c\#a.t, \nabla \vdash c\#b.s$
Sol : **Solving** $\{X : t \triangleq s\} \uplus P; S; \Gamma;\ \sigma \Longrightarrow P; S \cup \{X : t \triangleq s\}; \Gamma \cup \Gamma'; \sigma$, if none of the previous rules is applicable. $\Gamma' := \{a\#X \mid a \in \mathcal{A} \land \nabla \vdash a\#t \land \nabla \vdash a\#s\}$.
Mer : **Merging** $P; \{X : t_1 \triangleq s_1, Y : t_2 \triangleq s_2\} \uplus S; \Gamma; \sigma \Longrightarrow P; \{X : t_1 \triangleq s_1\} \cup S; \Gamma\{Y/\pi\cdot X\}; \sigma\{Y/\pi\cdot X\}$, where π is an $\text{Atoms}(t_1,s_1,t_2,s_2)$ – based permutation s.t. $\nabla \vdash \pi \bullet t_1 \approx t_2, \nabla \vdash \pi \bullet s_1 \approx s_2$.

To compute a generalization of two given terms t, s and a freshness context ∇ w.r.t. a set of atoms $\mathcal{A}$, we start with $\{X : t \triangleq s\};\ \emptyset;\ \Gamma;\ \varepsilon$ and apply the rules exhaustively. The generalization variable X does neither occur in ∇, nor in t, nor in s. Intuitively, it represents the generalization that becomes less and less general as the algorithm advances (by applying some rules). The result of this procedure is of the form $\emptyset;\ S;\ \Gamma;\ \sigma$ were $\langle \Gamma, X\sigma \rangle$ is the computed generalization and S contains the differences of t and s.

3.2 Equational Nominal Anti-Unification

Associative function applications are *flattened*, i.e., for any associative function symbol, say f^A, all subterms of the form $f^A(t_1,\dots,f^A(s_1\dots,s_m),\dots,t_n)$ are rewritten as $f^A(t_1,\dots,s_1\dots,s_m,\dots,t_n)$. Furthermore, $f^A(t)$ (f^A applied to a single argument) stands for t. The decomposition rule defined for the empty theory needs to be extended for associative and/or commutative function symbols. We consider each case separately in the style of [11]. Below Y_1, Y_2 are fresh variables of the corresponding sorts, $j = (i \mod 2) + 1$, $1 \leq k < n$, $1 \leq l < m$ and $n, m \geq 2$.

DecA : Associative Decomposition $\{X : f^A(t_1,\dots,t_n) \triangleq f^A(s_1,\dots,s_m)\} \cup P; S; \Gamma; \sigma \Longrightarrow \{Y_1 : f^A(t_1,\dots,t_k) \triangleq f^A(s_1,\dots,s_l),$ $Y_2 : f^A(t_{k+1},\dots,t_n) \triangleq f^A(s_{l+1},\dots,s_m)\} \cup P; S; \Gamma; \sigma\{X/f^A(Y_1,Y_2)\}$
DecC : Commutative Decomposition $\{X : f^C(t_1,t_2) \triangleq f^C(s_1,s_2)\} \cup P; S; \Gamma; \sigma \Longrightarrow \{Y_1 : t_1 \triangleq s_i, Y_2 : t_2 \triangleq s_j\} \cup P; S; \Gamma; \sigma\{X/f^C(Y_1,Y_2)\}$
DecAC : Associative-Commutative Decomposition $\{X : f^{AC}(t_1,\dots,t_n) \triangleq f^{AC}(s_1,\dots,s_m)\} \cup P; S; \Gamma; \sigma \Longrightarrow \{Y_1 : f^{AC}(t_{i_1},\dots,t_{i_k}) \triangleq f^{AC}(s_{j_1},\dots,s_{j_l}),$ $Y_2 : f^{AC}(t_{i_{k+1}},\dots,t_{i_n}) \triangleq f^{AC}(s_{j_{l+1}},\dots,s_{j_m})\} \cup P; S; \Gamma; \sigma\{X/f^{AC}(Y_1,Y_2)\}$ where $\{i_1\dots,i_n\} \equiv \{1,\dots,n\}, \{j_1\dots,j_m\} \equiv \{1,\dots,m\}$

Example 1. *Find a lgg of $f^{AC}((a\,b)\cdot X', g(a,a,b), g(b,b,a)) \triangleq f^{AC}(h^{AC}(b,a,b), h^{AC}(a,a,b), X')$. The initial freshness context is empty and the set of allowed atoms is $\{a,b\}$. The example involves branching. We will only illustrate one particular branch:*

$$\{X : f^{AC}((a\,b)\cdot X', g(a,a,b), g(b,b,a)) \triangleq f^{AC}(h^{AC}(b,a,b), h^{AC}(a,a,b), X')\}; \emptyset; \emptyset; Id$$
$$\Longrightarrow^{k=l=1,i_1=1,i_2=2,i_3=3,j_1=3,j_2=1,j_3=2}_{\mathsf{DecAC}}$$
$$\{Y_1 : (a\,b)\cdot X' \triangleq X', Y_2 : f^{AC}(g(a,a,b), g(b,b,a)) \triangleq f^{AC}(h^{AC}(b,a,b), h^{AC}(a,a,b))\}; \emptyset; \emptyset; \{X/f^{AC}(Y_1,Y_2)\}$$
$$\Longrightarrow^{k=l=1,i_1=j_1=1,i_2=j_2=2}_{\mathsf{DecAC}}$$
$$\{Y_1 : (a\,b)\cdot X' \triangleq X', Z_1 : g(a,a,b) \triangleq h^{AC}(b,a,b), Z_2 : g(b,b,a) \triangleq h^{AC}(a,a,b)\}; \emptyset; \emptyset; \{X/f^{AC}(Y_1,Z_1,Z_2)\}$$
$$\Longrightarrow^{3}_{\mathsf{Sol}}$$
$$\emptyset; \{Y_1 : (a\,b)\cdot X' \triangleq X', Z_1 : g(a,a,b) \triangleq h^{AC}(b,a,b), Z_2 : g(b,b,a) \triangleq h^{AC}(a,a,b)\}; \emptyset; \{X/f^{AC}(Y_1,Z_1,Z_2)\}$$
$$\Longrightarrow_{\mathsf{Mer}} \emptyset; \{Y_1 : (a\,b)\cdot X' \triangleq X', Z_1 : g(a,a,b) \triangleq h^{AC}(b,a,b)\}; \emptyset; \{X/f^{AC}(Y_1,Z_1,(a\,b)\cdot Z_1)\}$$

Note that in the last step we need to solve an equivariance problem. The computed term in context is $\langle\emptyset, f^{AC}(Y_1,Z_1,(a\,b)\cdot Z_1)\rangle$ and we can obtain the original terms by applying, resp., $f^{AC}(Y_1, Z_1,(a\,b)\cdot Z_1)\{Y_1/(a\,b)\cdot X', Z_1/g(a,a,b)\}$, and $f^{AC}(Y_1,Z_1,(a\,b)\cdot Z_1)\{Y_1/X', Z_1/h^{AC}(b,a,b)\}$.

4 Equivariance

In the merging rule of `NAU` we need to decide the (nominal) *equivariance problem,* that is: Given a set E of equations of nominal terms, is there a permutation π of the atoms that appear in E, such that all the equations become true (under consideration of E) when applying π to all the left-hand side terms, under the consideration of a given freshness context? That is, $\nabla \vdash \pi \bullet t \approx s$ for each equation $t \approx s \in E$. To solve that problem we extend the nominal equivariance algorithm that was introduced for the empty theory in [1]. It operates on tuples of the form E; ∇; $\mathcal{A}$; π where E is a set of equations of nominal terms $t \approx s$; ∇ is a freshness context; $\mathcal{A}$ is a finite set of atoms; and π is the permutation which is returned in case of success.

Equivariance algorithm (E). The algorithm works in two phases: **Phase 1** decomposes and simplifies the given equivariance problem. **Phase 2** tries to compute a permutation. The first phase involves branching. The entire procedure succeeds if one of the branches succeeds and, in that case, it returns a permutation computed by a successful branch. Note that there might be more than one successful branches. Again, we assume that associative function applications are flattened. Below $k \geq 0$, $n \geq 2$, $1 \leq i \leq n$, $i' \in \{1, 2\}$, $(s'_1, \ldots, s'_{n-1}) = (s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n)$ and c is an auxiliary atom of the corresponding sort that does not occur in $\mathcal{A}$.

Phase 1. Simplifying the equivariance problem.
`Dec-E` : $\{f(t_1, \ldots, t_k) \approx f(s_1, \ldots, s_k)\} \uplus E; \nabla; \mathcal{A}; Id \Longrightarrow \bigcup_{j=1}^{k} \{t_j \approx s_j\} \cup E; \nabla; \mathcal{A}; Id.$
`DecC-E` : $\{f^C(t_1, t_2) \approx f^C(s_1, s_2)\} \uplus E; \nabla; \mathcal{A}; Id \Longrightarrow \{t_1 \approx s_{i'}, t_2 \approx s_{(i' \mod 2)+1}\} \cup E; \nabla; \mathcal{A}; Id.$
`DecAC-E` : $\{f^{AC}(t_1, \ldots, t_n) \approx f^{AC}(s_1, \ldots, s_n)\} \uplus E; \nabla; \mathcal{A}; Id \Longrightarrow$ $\{t_1 \approx s_i, f^{AC}(t_2, \ldots, t_n) \approx f^{AC}(s'_1, \ldots, s'_{n-1})\} \cup E; \nabla; \mathcal{A}; Id.$
`Alp-E` : $\{a.t \approx b.s\} \uplus E; \nabla; \mathcal{A}; Id \Longrightarrow \{(c\ a) \bullet t \approx (c\ b) \bullet s\} \cup E; \nabla; \mathcal{A}; Id.$
`Sus-E` : $\{\pi_1 \cdot X \approx \pi_2 \cdot X\} \uplus E; \nabla; \mathcal{A}; Id \Longrightarrow \{\pi_1 \bullet a \approx \pi_2 \bullet a \mid a \in \mathcal{A} \wedge a\#X \notin \nabla\} \cup E; \nabla; \mathcal{A}; Id.$
Phase 2. Computing the permutation.
`Rem-E` : $\{a \approx b\} \uplus E;\ \nabla;\ \mathcal{A};\ \pi \Longrightarrow E;\ \nabla;\ \mathcal{A} \setminus \{b\};\ \pi, \quad$ if $\pi \bullet a = b$.
`Sol-E` : $\{a \approx b\} \uplus E;\ \nabla;\ \mathcal{A};\ \pi \Longrightarrow E;\ \nabla;\ \mathcal{A} \setminus \{b\};\ (\pi \bullet a\ \ b)\pi, \quad$ if $b \in \mathcal{A}, \pi \bullet a \neq b$.

Note that the equivariance algorithm doesn't contain a separate decomposition rule for associative symbols, since, in that case, flattening of associative function applications and simply applying `Dec-E` suffices.

Example 2 (Cont. Example 1)**.** *To obtain the permutation in the last step of the previous example, we proceeded as following:*

$$\begin{aligned}
&\{g(a, a, b) \approx g(b, b, a),\ h^{AC}(b, a, b) \approx h^{AC}(a, a, b)\}; \emptyset; \{a, b\}; Id \Longrightarrow_{\mathsf{Dec\text{-}E}} \\
&\{a \approx b,\ b \approx a,\ h^{AC}(b, a, b) \approx h^{AC}(a, a, b)\}; \emptyset; \{a, b\}; Id \Longrightarrow^{i=1}_{\mathsf{DecAC\text{-}E}} \\
&\{a \approx b,\ b \approx a,\ h^{AC}(a, b) \approx h^{AC}(a, b)\}; \emptyset; \{a, b\}; Id \Longrightarrow^{i=2}_{\mathsf{DecAC\text{-}E}} \\
&\{a \approx b,\ b \approx a\}; \emptyset; \{a, b\}; Id \Longrightarrow^{\mathsf{Sol\text{-}E}}_{\mathsf{Rem\text{-}E}} \qquad \emptyset; \emptyset; \emptyset; (a\, b)
\end{aligned}$$

The example illustrates a successful branch and the computed permutation is $(a\, b)$*. Note that, in the third step,* $h^{AC}(t)$ *(*h^{AC} *applied to a single argument) stands for* t*.*

5 Conclusion

We have investigated the extension of the nominal anti-unification problem to theories A,C and AC. The extended algorithm is obtained by adding specific rules to deal with each particular theory as well as the adaptation of the equivariance algorithm. In the nominal unification modulo commutativity scenario one knows that equations of the form $\pi \cdot X \approx_C X$ are the cause for the type of the nominal unification problem to be non-unitary, differently from the first order approach. However, in the nominal anti-unification setting modulo commutativity, a fixed-point *anti-equation* $\{X : \pi \cdot Y_1 \triangleq_C Y_1\}$, is trivially solvable, and causes no unexpected problems. Besides, since there is a reduction from nominal unification to higher-order pattern unification, we plan to investigate whether there is a simple reduction from our setting to the setting of equational HOPAU from [11]. It would also be interesting to check how the nominal anti-unification algorithm deals with recursive let as in[2].

References

[1] Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal anti-unification. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, Warsaw, Poland*, volume 36 of *LIPIcs*, pages 57–73. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[2] Manfred Schmidt-Schauß, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal unification of higher order expressions with recursive let. In Manuel V. Hermenegildo and Pedro López-García, editors, *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*, volume 10184 of *Lecture Notes in Computer Science*, pages 328–344. Springer, 2016.

[3] Manfred Schmidt-Schauß, David Sabel, and Yunus D. K. Kutz. Nominal unification with atom-variables. *J. Symb. Comput.*, 90:42–64, 2019.

[4] Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, Daniele Nantes-Sobrinho, and Ana Cristina Rocha Oliveira. A formalisation of nominal α-equivalence with A, C, and AC function symbols. *Theor. Comput. Sci.*, 781:3–23, 2019.

[5] Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, and Daniele Nantes-Sobrinho. Nominal c-unification. In Fabio Fioravanti and John P. Gallagher, editors, *Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers*, volume 10855 of *Lecture Notes in Computer Science*, pages 235–251. Springer, 2017.

[6] Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, and Daniele Nantes-Sobrinho. On solving nominal fixpoint equations. In Clare Dixon and Marcelo Finger, editors, *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings*, volume 10483 of *Lecture Notes in Computer Science*, pages 209–226. Springer, 2017.

[7] Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. On nominal syntax and permutation fixed points. *CoRR*, abs/1902.08345, 2019.

[8] Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. *ACM Trans. Comput. Log.*, 13(2):10:1–10:31, 2012.

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

[10] Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. Higher-order pattern anti-unification in linear time. *J. Autom. Reasoning*, 58(2):293–310, 2017.

[11] David M. Cerna and Temur Kutsia. Higher-order equational pattern anti-unification. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*, pages 12:1–12:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[12] Maribel Fernández and Murdoch Gabbay. Nominal rewriting. *Inf. Comput.*, 205(6):917–965, 2007.

[13] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, 2013.

On the Unification of Term Schemata

David Cerna[1,2], Alexander Leitsch[3], and Anela Lolic[4]

[1] Institute for Formal Methods and Verification, JKU Linz, Austria
[2] Research Institute for Symbolic Computation, JKU Linz, Austria
david.cerna@jku.at,david.cerna@risc.jku.at
[3] Institute of Logic and Computation, TU Wien, Vienna, Austria
leitsch@logic.at
[4] Institute of Logic and Computation, TU Wien, Vienna, Austria
anela@logic.at

Abstract

Term schemata are infinite sequences of terms which are defined inductively. We investigate the unification problem for term schemata and formulate some open problems. The solution of these problems is relevant to the analysis of proof schemata, in particular to schematic cut-elimination.

1 Introduction

Recursive definitions of functions play a central role in computer science, particularly in functional programming. While recursive definitions of formulas and proofs are less common, they are of increasing importance in automated proof analysis. Proof schemata, i.e. recursively defined infinite sequences of proofs, serve as an alternative formulation of induction. Prior to the formalization of the concept, an analysis of Fürstenberg's proof of the infinitude of primes [1] suggested the need for a formalism quite close to the type of proof schemata defined in [6]. The underlying method for this analysis was CERES [2] (cut-elimination by resolution) which, unlike reductive cut-elimination, can be applied to recursively defined proofs by extracting a schematic unsatisfiable universal formula (the characteristic schema) and constructing a recursively defined refutation. Moreover, Herbrand's theorem can be extended to an expressive fragment of proof schemata, that is those formalizing k-induction [4, 6]. Unfortunately, the construction of recursively defined refutations is a highly complex task. In previous work [6] a superposition calculus for certain types of formulas was used for the construction of refutation schemata, but only works for a weak fragment of arithmetic and is hard to use interactively. In [3] the schematic approach was substantially generalized; the new method is capable of handling several recursion parameters and thus can deal with nested inductions. To refute the corresponding characteristic schemata a new resolution calculus for universal formula schemata was developed (see also [3]). A crucial part of this calculus is unification which is used to define single resolution steps. Unlike ordinary first-order unification the problem here consists in *unifying term schemata*, i.e. syntactic expressions representing infinite (recursively defined) sequences of terms. In [3] we introduced the new concept of s-unification (*schematic*-unification) which replaces ordinary unification in case of schemata. But s-unification is just one possibility to approach unification of term schemata. In this paper we describe the general problem of unifying term schemata and characterize different subclasses of schematic unification problems. Several models for unification of term schemata were developed in the 1990ties; we just mention [5] and [8]. However, our approach differs from those mentioned above; it is based on primitive recursive definitions and the unification problem is undecidable in general.

2 Unification problems for term schemata

Below we define the general problem, give some basic definitions and formulate decision problems for schematic unification. In the most general sense a term schema is an arbitrary infinite sequence of first-order terms s_n. Formally we work with syntactic expressions $\hat{s}(n)$ where n is a parameter (a number variable). For any assignment $\sigma = \{n \to \alpha\}$, for α being a numeral, $\hat{s}(\alpha)$ evaluates to a term s_α; for the evaluation via σ we write $\hat{s}(n)\downarrow_\sigma = s_\alpha$. We use a similar framework for substitution schemata. If $\hat{\lambda}(n)$ is a syntactic expression representing a substitution schema and σ is an assignment (as defined above) then $\hat{\lambda}(n)\downarrow_\sigma$ is a substitution λ_α.

Definition 1 (unification of term schemata)**.** *Given two term schemata $\hat{s}(n)$ and $\hat{t}(n)$ we define $\hat{s}(n), \hat{t}(n)$ as* unifiable *if there exists a substitution schema $\hat{\lambda}(n)$ such that $\hat{s(n)}\hat{\lambda}(n)\downarrow_\sigma = \hat{t(n)}\hat{\lambda}(n)\downarrow_\sigma$ for all assignments σ.*

For a fully formal definition of term schemata on which this paper is based see [3]. Here we define two different types of schematic unification problems, simple and global ones. We use basic definitions from [3], especially defined and undefined symbols and an ordering $<$ of the defined symbols. For applications in computational logic only computable term schemata makes sense. Here we consider only schemata defined by primitive recursion. The so-called *simple term schemata* are schemata with a fixed number of variables defined via primitive recursion:

Definition 2 (simple term schema)**.** *Let $\vec{x}$ be a tuple of first-order variables (variables of type ι) and n be a parameter (a variable of type ω). A* simple term schema *is defined by primitive recursive definitions of the form*

$$\begin{aligned}\hat{f}(\vec{x}, \mathbf{0}) &= g(\vec{x}),\\ \hat{f}(\vec{x}, s(n)) &= h(\vec{x}, n, z)\{z \leftarrow \hat{f}(\vec{x}, n)\}\end{aligned}$$

where $g(\vec{x})$ is a term over the variables $\vec{x}$ and $h(\vec{x}, n, z)$ is a term over the variables $\vec{x}, z$ and the parameter n. If $\hat{f}$ is not a minimal defined symbol then both $g(\vec{x})$ and $h(\vec{x}, n, z)$ may contain defined symbols $\hat{u}$ with $\hat{u} < \hat{f}$.

Note that the general unfication problem for simple term schemata is undecidable (the equivalence problem of loop-programs can be reduced to it).

Example 1. *Consider the following simple term schema:*

$$\hat{f}(x, \mathbf{0}) = h(a, a) \qquad \hat{f}(x, s(n)) = h(x, \hat{f}(x, n))$$

$$\begin{aligned}\hat{f}_1(x, y, \mathbf{0}) &= h(a, a) \qquad & \hat{f}_1(x, y, s(n)) &= h(x, \hat{f}(y, n))\\ \hat{g}(x, y, \mathbf{0}) &= h(a, a) \qquad & \hat{g}(x, y, s(n)) &= h(\hat{g}(x, y, n), y)\end{aligned}$$

Using these simple term schemata we can form the following four unification problems:

$$\begin{aligned}&\hat{f}(x, s(n)) \overset{?}{=} \hat{g}(x, x, s(n)), \qquad &&\hat{f}(x, s(n)) \overset{?}{=} \hat{g}(x, y, s(n)),\\ &\hat{f}(x, s(n)) \overset{?}{=} \hat{g}(y, y, s(n)), \qquad &&\hat{f}_1(x, y, s(n)) \overset{?}{=} \hat{g}(z, z, s(n)).\end{aligned}$$

Notice that the first three problems fail due to the occurs-check while the fourth problem does not and is unifiable. Let us consider the first and the last problem in more detail:

$\hat{f}(x, s(n)) \overset{?}{=} \hat{g}(x, x, s(n))$ is solvable iff for all subsitutions of the number variable n by a numeral

k the normal forms of $\hat{f}(x, s(k))$ and $\hat{g}(x, x, s(k))$ are unifiable (they are ordinary first-order terms). Therefore the first unification problem is unsolvable because $\hat{f}(x, s(1))$ and $\hat{g}(x, x, s(1))$ are not unifiable; note that

$$\hat{f}(x, 2) = h(x, \hat{f}(x, 1)) = h(x, h(x, \hat{f}(x, 0))) = h(x, h(x, h(a, a))),$$
$$\hat{g}(x, x, 2) = h(\hat{g}(x, x, 1), x) = h(h(\hat{g}(x, x, 0), x), x) = h(h(h(a, a), x), x).$$

The fourth problem is solvable. The infinite sequence of unifiers is given by the substitution schema

$$\hat{\vartheta}(n) = \{x \leftarrow \hat{g}(\hat{f}(y, n), \hat{f}(y, n), n),\ z \leftarrow \hat{f}(y, n)\}.$$

Even though simple term schemata allow for recursive definitions, recursive variable occurrence causes unification to fail in most cases, thus, like the fourth example, unification is usually decided by analyzing the structure of the terms.

In contrast to simple term schemata *global term schemata* are based on primitive recursive definitions using *global* variables instead of ordinary first-order variables (see [3]). These schemata may contain an increasing numbers of variables depending on the assignment of the parameter n.

Definition 3 (global term schema)**.** *Let $\vec{X}$ be a tuple of global variables (variables of type $\omega \to \iota$) and n be a parameter (a variable of type ω). A* global term schema *is defined by primitive recursive definitions of the form*

$$\begin{array}{rcl} \hat{f}(\vec{X}, \mathbf{0}) & = & t(\vec{X}), \\ \hat{f}(\vec{X}, s(n)) & = & s(\vec{X}, n, z)\{z \leftarrow \hat{f}(\vec{X}, n)\} \end{array}$$

where $t(\vec{X})$ is a term over the global variables $\vec{X}$ and $s(\vec{X}, n, z)$ is a term over the global variables $\vec{X}$, the individual variable z and the parameter n. If $\hat{f}$ is not a minimal defined symbol then both $t(\vec{X})$ and $s(\vec{X}, n, z)$ may contain defined symbols $\hat{u}$ with $\hat{u} < \hat{f}$.

A formal definition of (the semantics of) objects of the form $\hat{f}(\vec{X}, n)$ can be found in [3]. While for simple term schemata the domain variables of the unification schema form a fixed finite set, the unifiers in global term schemata have domains which may depend on the parameter n. Still it is possible that there exists a unification schema of the form

$$\hat{\vartheta}(n) \colon \{X(s_1) \leftarrow t_1, \ldots, X(s_k) \leftarrow t_k\}$$

where k is a fixed number. Note that, for every assignment σ, $\hat{\vartheta}(n){\downarrow_\sigma}$ has a domain which varies with σ but is always of a fixed size k. We refer to such unifiers as *s-unification schemata* [3]. The following examples illustrates such an s-unification.

Example 2. *Consider the following global term schema:*

$$\begin{array}{rclcrcl} \hat{f}(X, \mathbf{0}) & = & h(a, X(0)) & \qquad & \hat{f}(X, s(n)) & = & h(X(s(n)), \hat{f}(X, n)) \\ \hat{g}(X, \mathbf{0}) & = & h(X(0), a) & \qquad & \hat{g}(X, s(n)) & = & h(\hat{g}(X, n), X(s(n))) \end{array}$$

Using these schemata we can define the unification problem

$$\hat{f}(X, s(n)) \overset{?}{=} \hat{g}(X, s(n))$$

which has as a unification schema, $\hat{\vartheta}(n)\colon \left\{X(n) \leftarrow \hat{h}(n)\right\}$ *where* $\hat{h}(n)$ *is as follows:*

$$\hat{h}(\mathbf{0}) \;=\; a \qquad \hat{h}(s(n)) \;=\; h(\hat{h}(n), \hat{h}(n))$$

$\hat{\vartheta}(n)$ *is an s-unifier; its domain is different for every assignment of the parameter* n *but the domain size is always* 1*. Like in the previous example the bindings may contain schematically defined terms.*

In the next example we define a global term schema and a corresponding unification problem which is solvable (i.e. there is a formal expression representing the substitution schema) but unsolvable via s-unifiers, i.e. if $\hat{\vartheta}(n)$ is an unification schema of Example 3, then there exist two assignments σ, σ' such that $|dom(\hat{\vartheta}(n){\downarrow_\sigma})| \neq |dom(\hat{\vartheta}(n){\downarrow_{\sigma'}})|$.

Example 3. *Consider the following two global schemata (where the second schema could also be defined as a simple one):*

$$\begin{array}{ll} \hat{f}(X,0) = X(0) & \hat{f}(X,n+1) = h(X(n+1), \hat{f}(X,n)), \\ \hat{g}(X,0) = X(0) & \hat{g}(X,n+1) = h(X(0), \hat{g}(X,n)). \end{array}$$

Note that $\hat{f}(X,0), \hat{f}(X,1), \hat{f}(X,2) \ldots$ *evaluate to*

$$X(0),\; h(X(1), X(0)),\; h(X(2), h(X(1), X(0))), \ldots$$

so the number of different first-order variables is increasing. The unification problem

$$\hat{f}(X,n) \stackrel{?}{=} \hat{g}(X,n)$$

is solvable. The schematic unifier has the following recursive definition

$$\hat{\vartheta}(0) = \{\},\;\; \hat{\vartheta}(n+1) = \{X(n+1) \leftarrow X(0)\} \cup \vartheta(n).$$

Note that, for $\sigma = \{n \to \alpha\}$ *and* $\sigma' = \{n \to \alpha+1\}$*,* $dom(\vartheta(n){\downarrow_{\sigma'}}) =$ $\{X(\alpha+1)\} \cup dom(\vartheta(n){\downarrow_\sigma})$ *and so the size of the domain is not invariant under assignments. Obviously there exists no s-unifier (unifier with fixed domain size) for this unification problem.*

Additionally, it is crucial to distinguish *free schemata* containing no equations between terms - except primitive recursive definitions - and *theory schemata*. In order to define the class of primitive recursive functions we need a theory schema containing equations defining projections and constant functions. We call such a theory schema the *standard schema*.

Definition 4 (standard schema)**.** *A term schema (simple or global) is called a* standard schema *if it contains*

- *equations of the form* $\hat{g}[\alpha, i](x_1, \ldots, x_\alpha) = x_i$ *(where* $1 \leq i \leq \alpha$*) for every projection function* $I^\alpha_i\colon \iota^\alpha \to \iota$ *where* $I^\alpha_i(\beta_1, \ldots, \beta_n) = \beta_i$ *and*

- *equations of the form* $\hat{h}[\alpha, c](x_1, \ldots, x_\alpha) = c$ *for every constant function of type* $\iota^\alpha \to \iota$*.*

Here $\hat{g}[\alpha, i], \hat{h}[\alpha, c]$ *are* α*-ary defined function symbols.*

It is well known that the equivalence problem of standard schemata is undecidable and equivalent to the equivalence problem of LOOP programs. The problem is even undecidable when the recursion depth is ≤ 2 (correponding to the equivalence problem of LOOP-2 programs). As a consequence the unification problem for standard term schemata is undecidable as well. However, when we only consider standard term schemata of recursion depth ≤ 1 (corresponding to the LOOP-1 class) the equivalence problem becomes decidable. Based on the definitions above we can define the following 3 problems:

1. *Is the unification problem for simple standard schemata of recursion depth* ≤ 1 *decidable?* In this case both the domain size and the variables occurring in the domain are fixed. For a proof of decidability, one would need to show that unification for simple standard schemata of recursion depth ≤ 1 is reducible to the equivalence problem of LOOP-1 programs.

2. *Is the unification problem for global standard schemata of recursion depth* ≤ 1 *decidable?* This question is similar to the previous question, but for more general notions of unifiers and variables, i.e. where the domain size varies and the variable indexing is part of the object language.

3. *Is the unification problem for simple and/or global free schemata decidable?* When we restrict ourselves to free schemata, we are restricting the types of functions which can be represented. For example, without projections there is not much which can be done with the arguments to a defined symbol within a term schema. However, such simple recursive structures may turn up in the resolution calculus discussed in [3] which motivates our interest in them. The situation is similar for global free schemata.

As a final remark, our investigation has thus far ignored equational unification. We are interested in studying E-unification in the term schema setting, however, due to the complexity of such an investigation we have left it to future work. Concerning the relationship to higher-order unification, one may notice that our use of higher-order variables (global variables) is very restricted, essentially a weakened form of higher-order patterns [7]. While it may be interesting to investigate the relationship between the two formalisms, it is not clear if free term schema can be easily expressed in a language handled by existing higher-order unification algorithms, nor if it would be beneficial to address it in such a setting.

References

[1] Matthias Baaz, Stefan Hetzl, Alexander Leitsch, Clemens Richter, and Hendrik Spohr. Ceres: An analysis of Fürstenberg's proof of the infinity of primes. *Theoretical Computer Science*, 403(2-3):160–175, August 2008.

[2] Matthias Baaz and Alexander Leitsch. Cut-elimination and redundancy-elimination by resolution. *Journal of Symbolic Computation*, 29:149–176, 2000.

[3] David Cerna, Alexander Leitsch, and Anela Lolic. Schematic refutations of formula schemata. *CoRR*, abs/1902.08055, 2019.

[4] Cvetan Dunchev, Alexander Leitsch, Mikheil Rukhaia, and Daniel Weller. Cut-elimination and proof schemata. In *TbiLLC*, volume 8984 of *Lecture Notes in Computer Science*, pages 117–136. Springer, 2013.

[5] Miki Hermann and Roman Galbavý. Unification of infinite sets of terms schematized by primal grammars. *Theoretical Computer Science*, 176(1):111 – 158, 1997.

[6] Alexander Leitsch, Nicolas Peltier, and Daniel Weller. CERES for first-order schemata. *J. Log. Comput.*, 27(7):1897–1954, 2017.

[7] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming*, pages 253–281, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[8] Gernot Salzer. *Unification of Meta-Terms.* PhD thesis, Technical University of Vienna, 1991.

Terminating Non-Disjoint Combined Unification

(Extended Abstract)

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

[1] University of Texas at Dallas, USA
serdar.erbatur@utdallas.edu
[2] University of Mary Washington, USA
marshall@umw.edu
[3] Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
Christophe.Ringeissen@loria.fr

1 Introduction

Unification is a critical tool in many fields such as automated reasoning, logic programming, declarative programming, and the formal analysis of security protocols. For many of these applications we want to consider equational unification, where the problem is defined modulo an equational theory E, such as Associativity-Commutativity. Since equational unification is undecidable in general, specialized techniques have been developed to solve the problem for particular classes of equational theories, many of high practical interest. For instance, when the equational theory E has the Finite Variant Property (FVP) [3, 7], there exists a reduction from E-unification to syntactic unification via the computation of finitely many variants of the unification problem.

Another ubiquitous scenario is given by an equational theory E involved in a union of theories $F \cup E$. To solve this case, it is quite natural to proceed in a modular way by reusing the unification algorithms available for F and for E. There are terminating and complete combination procedures for signature-disjoint unions of theories [10, 2]. However, the non-disjoint case remains a challenging problem. One approach to the non-disjoint combination problem that has been successful in some cases is the hierarchical approach [5]. In this approach, $F \cup E$-unification can be considered as a conservative extension of E-unification. Then, a new inference system related to F, say U_F, can be combined with an E-unification algorithm to obtain a $F \cup E$ unification algorithm. While this hierarchical approach won't work for every $F \cup E$ it can be a very useful tool when applicable. However, up to now it could be complex to know if a combination $F \cup E$ could be solved via the hierarchical approach. For example, there is no general method for obtaining the inference system U_F, and the resulting hierarchical unification procedure may not terminate.

In this paper, we consider "syntactic" theories $F \cup E$ where U_F can be defined as a system of mutation rules, and we present new terminating instances of the hierarchical unification procedure.

2 Preliminaries

We use the standard notation of equational and term rewriting systems [1]. An equational theory E is *regular* if for any axiom $l = r \in E$, l and r have the same set of variables. An equational theory E is *collapse-free* if for any axiom $l = r \in E$, l and r are non-variable terms. An equational theory E is *subterm collapse-free* if for all terms t it is not the case that $t =_E u$ where u is a strict subterm of t. A subterm collapse-free theory is necessarily regular and collapse-free. An equational term rewrite system, equational TRS for short, is a pair (R, E) where R is a set of rewrite Σ-rules and E is an equational Σ-theory, Σ being a signature. An equational TRS (R, E) is said to be E-convergent if the rewrite relation $\rightarrow_{R,E}$, defined via E-matching, is *E-convergent*, meaning that $=_E \circ \rightarrow_{R,E} \circ =_E$ is terminating and $\rightarrow_{R,E}$ is Church-Rosser modulo E. A function symbol that does not occur in $\{l(\epsilon) \mid l \rightarrow r \in R\}$ is called a *constructor* for R. Let Σ_0 be the subsignature of Σ that consists of function symbols occurring in the axioms of E. An E-convergent TRS (R, E) is said to be *E-constructed* if all symbols in Σ_0 are constructors for R. Given two rewrite rules $g \rightarrow d$ and $l \rightarrow r$, the E-**Forward** inference generates a new rewrite rule when l and d overlap. It is formally defined as follows:

E-**Forward** $\quad g \rightarrow d[l'],\ l \rightarrow r \ \vdash\ (g \rightarrow d[r])\sigma$
where $g \rightarrow d[l'], l \rightarrow r \in R, l'$ is not a variable, $\sigma \in CSU_E(l' =^? l)$.

An E-convergent TRS (R, E) is *forward-closed* if any application of E-**Forward** generates a rule which is redundant in (R, E) when the premises are rules in (R, E), following an appropriate definition of redundancy [8]. It can be shown that for any E-constructed TRS (R, E) where E is regular, collapse-free and E-unification is finitary, (R, E) has the FVP if and only if it has a finite closure by E-**Forward**.

An *alien* subterm of a Σ_0-rooted term t is a $\Sigma\backslash\Sigma_0$-rooted subterm s such that all superterms of s are Σ_0-rooted. A set of equations $G = \{x_1 = t_1,\ \ldots,\ x_n = t_n\}$ is said to be in *tree solved form* if each x_i is a variable occurring once in G.

3 Hierarchical Unification

Consider now a union of theories $R \cup E$ where E is regular and collapse-free and (R, E) is assumed to be E-constructed. Thanks to this assumption, R and E are "sufficiently separated" and thus we can envision the problem of building a $R \cup E$-unification algorithm using an approach based on combination. A hierarchical unification procedure is parameterized by an E-unification algorithm and a mutation-based reduction procedure U. It applies some additional rules given in Figure 1: **Coalesce**, **Split**, **Flatten**, and **VA** are used to separate the terms, U is used to simplify the $\Sigma\backslash\Sigma_0$-equations, and finally, **Solve** calls the E-unification algorithm.

Coalesce $\quad \{x = y\} \cup G \ \vdash\ \{x = y\} \cup (G\{x \mapsto y\})$
where x and y are distinct variables occurring both in G.

Split $\quad \{f(\bar{v}) = t\} \cup G \ \vdash\ \{x = f(\bar{v}), x = t\} \cup G$
where $f \in \Sigma\backslash\Sigma_0$, t is a non-variable term and x is a fresh variable.

Flatten $\quad \{v = f(\ldots, u, \ldots)\} \cup G \ \vdash\ \{v = f(\ldots, x, \ldots), x = u\} \cup G$
where $f \in \Sigma\backslash\Sigma_0$, v is a variable, u is a non-variable term, and x is a fresh variable.

VA $\quad \{s = t[u]\} \cup G \ \vdash\ \{s = t[x], x = u\} \cup G$
where t is Σ_0-rooted, u is an alien subterm of t, and x is a fresh variable.

Solve $\quad G \cup G_0 \ \vdash\ \bigvee_{\sigma_0 \in CSU_E(G_0)} G \cup \hat{\sigma}_0$
where G is a set of $\Sigma\backslash\Sigma_0$-equations, G_0 is a set of Σ_0-equations, G_0 is E-unifiable and not in tree solved form, $\hat{\sigma}_0$ is the tree solved form associated with σ_0, and w.l.o.g for any $x \in Dom(\sigma_0)$, $x\sigma_0 \in Var(G_0)$ if $x\sigma_0$ is a variable.

Figure 1: H_E rules

Definition 1 (Hierarchical unification procedure)**.** *Assume a Σ_0-theory E for which an E-unification algorithm is known to compute a finite $CSU_E(G_0)$ for all E-unification problems G_0, a Σ-theory $F \cup E$ for which E-unification is complete for solving the Σ_0-fragment of $F \cup E$-unification, and an inference system U satisfying the following assumptions: U transforms only equations of the form $x_0 = f(x_1, \dots, x_n)$ where $x_0, x_1, \dots, x_n$ are variables and f is a function symbol in $\Sigma \backslash \Sigma_0$; and U is parameterized by some finite set S of $F \cup E$-equalities such that the soundness and completeness of each inference $\vdash_U$ follows from at most one equality in S. Under these assumptions, the $H_E(U)$ inference system is defined as the repeated application of some inference from H_E (cf. Figure 1) or U, using the following order of priority:* ***Coalesce**, **Split**, **Flatten**, **VA**, U, **Solve**. A $F \cup E$-unification problem is in* separate form *if it is a normal form with respect to $H_E \backslash \{$**Solve**$\}$.*

Note, that when we speak of an inference system, U, this is not just a set of rules but also a strategy for applying those rules. This could include, as in the $\mathcal{E}_{AC}$ case of Proposition 3, methods for detecting errors such as occur-checks and non-termination [6].

Proposition 1. *Let (R, E) be any E-constructed TRS such that an inference system U following Definition 1 is known for the equational theory $R \cup E$, in addition to an existing E-unification algorithm. Then E, $R \cup E$ and U satisfy the assumptions of Definition 1, and the $H_E(U)$ inference system provides a sound and complete $R \cup E$-unification procedure if the normal forms w.r.t $H_E(U)$ are either the dag solved forms or problems that are not $R \cup E$-unifiable. If $H_E(U)$ is terminating, then it is a $R \cup E$-unification algorithm.*

3.1 Subterm Collapse-Free Theories

Hierarchical unification algorithms are known for particular subterm collapse-free theories of particular interest for protocol analysis.

Proposition 2. *([11, 6]) Let E be the empty Σ_0-theory where Σ_0 only consists of a binary function symbol $*$. Consider $R_{\mathcal{D}} = \{h(x * y) \to h(x) * h(y)\}$ and $R_{\mathcal{D}1} = \{f(x * y, z) \to f(x, z) * f(y, z)\}$. The equational TRSs $(R_{\mathcal{D}}, E)$ and $(R_{\mathcal{D}1}, E)$ are E-constructed. Moreover, $R_{\mathcal{D}} \cup E$ (resp., $R_{\mathcal{D}1} \cup E$) is a subterm collapse-free theory admitting a unification algorithm of the form $H_E(U_{\mathcal{D}})$ (resp., $H_E(U_{\mathcal{D}1})$).*

Proposition 3. *([6]) Let $AC = AC(\circledast)$, $R_{\mathcal{E}} = \{exp(exp(x, y), z) \to exp(x, y \circledast z), exp(x * y, z) \to exp(x, z) * exp(y, z)\}$ and $R_{\mathcal{F}} = \{enc(enc(x, y), z) \to enc(x, y \circledast z)\}$. The equational TRSs $(R_{\mathcal{E}}, AC)$ and $(R_{\mathcal{F}}, AC)$ are AC-constructed. Moreover, $\mathcal{E}_{AC} = R_{\mathcal{E}} \cup AC$ (resp., $\mathcal{F}_{AC} = R_{\mathcal{F}} \cup AC$) is a subterm collapse-free theory admitting a unification algorithm of the form $H_{AC}(U_{\mathcal{E}})$ (resp., $H_{AC}(U_{\mathcal{F}})$).*

3.2 Forward-Closed E-Constructed TRSs

For any forward-closed E-constructed TRS (R, E) such that E is regular and collapse-free, a $R \cup E$-unification algorithm of the form $H_E(U)$ can be obtained by defining some inference system U based on the *Basic Syntactic Mutation* approach initiated for the class of theories saturated by paramodulation [9], and already applied in [4] to a particular class of forward-closed equational TRSs.

Let BSM_R be the inference system given in Figure 2. One can notice that each inference rule in BSM_R generates some boxed terms. This particular annotation of terms, detailed in [9, 4], allows us to control the rules application in such a way that BSM_R is terminating.

Imit $\bigcup_i \{x = f(\bar{v}_i)\} \cup G \ \vdash \ \{x = \boxed{f(\bar{y})}\} \cup \bigcup_i \{\bar{y} = \bar{v}_i\} \cup G$
where $f \in \Sigma \backslash \Sigma_0$, $i > 1$, $\bar{y}$ are fresh variables and there are no more equations $x = f(\dots)$ in G.

MutConflict$_R$ $\{x = f(\bar{v})\} \cup G \ \vdash \ \{x = \boxed{t}, \boxed{\bar{s}} = \bar{v}\} \cup G$
where $f \in \Sigma \backslash \Sigma_0$, $f(\bar{s}) \rightarrow t$ is a fresh instance of a rule in R, $f(\bar{v})$ is unboxed, and (there is another equation $x = u$ in G with a non-variable term u or $x = f(\bar{v})$ occurs in a cycle).

ImitCycle $\{x = f(\bar{v})\} \cup G \ \vdash \ \{x = \boxed{f(\bar{y})}, \bar{y} = \bar{v}\} \cup G$
where $f \in \Sigma \backslash \Sigma_0$, $f(\bar{v})$ is unboxed, $\bar{y}$ are fresh variables and $x = f(\bar{v})$ occurs in a cycle.

Figure 2: BSM_R rules

Lemma 1. *Assume E is any regular and collapse-free theory such that an E-unification algorithm is known. Let (R, E) be a forward-closed E-constructed TRS and BSM_R the inference system given in Fig. 2. Then $H_E(BSM_R)$ is a $R \cup E$-unification algorithm.*

Example 1. *Consider $R = \{\pi_1(x.y) \rightarrow x, \pi_2(x.y) \rightarrow y, dec(enc(x, y), y) \rightarrow x\}$ and $E = \{enc(x.y, z) = enc(x, z).enc(y, z)\}$. E-unification algorithms are know for this type of one-sided distributivity [11] and can be used in a hierarchical unification procedure of the form $H_E(BSM_R)$. Since (R, E) is forward-closed and E-constructed, $H_E(BSM_R)$ provides an $R \cup E$-unification algorithm.*

4 Combined Hierarchical Unification

We are now interested in combining hierarchical unification algorithms known for E-constructed TRSs. Given two E-constructed TRSs, say (R_1, E) and (R_2, E), the problem is to study the possible construction of a (combined) hierarchical unification algorithm for $(R_1 \cup R_2, E)$ using the two hierarchical unification algorithms known for (R_1, E) and (R_2, E).

4.1 Combining Subterm Collapse-Free Theories

Let us first consider a technical lemma which is useful to get a hierarchical unification procedure.

Lemma 2. *Let (R_1, E) and (R_2, E) be two E-constructed TRSs sharing only symbols in E such that, for $i = 1, 2$, $R_i \cup E$ admits a sound and complete unification procedure of the form $H_E(U_i)$. Assume that $R_1 \cup R_2 \cup E$ is subterm collapse-free, and for any $\Sigma_1 \backslash \Sigma_0$-rooted term t_1 and any $\Sigma_2 \backslash \Sigma_0$-rooted term t_2, t_1 cannot be equal to t_2 modulo $R_1 \cup R_2 \cup E$. Then, $H_E(U_1 \cup U_2)$ is a sound and complete $R_1 \cup R_2 \cup E$-unification procedure.*

We study below a possible way to satisfy the assumptions of Lemma 2.

Definition 2 (Layer-preservingness)**.** *Let (R, E) be an E-constructed TRS over the signature Σ. A Σ-term t is said to be* Σ_0-capped *if there exist a constant-free Σ_0-term u and a substitution σ such that $t = u\sigma$, $Dom(\sigma) = Var(u)$ and $Ran(\sigma)$ is a set of $\Sigma \backslash \Sigma_0$-rooted terms. The TRS (R, E) is said to be* layer-preserving *if $R \cup E$ is subterm collapse-free and any normal form of any $\Sigma \backslash \Sigma_0$-rooted term is Σ_0-capped.*

Remark 1. *The assumption that rules in R are $\Sigma \backslash \Sigma_0$-rooted was used in [5], and layer-preservingness generalizes this assumption.*

The property of being E-constructed and layer-preserving is modular.

Lemma 3. *Assume E is subterm collapse-free, for $i = 1, 2$, (R_i, E) is an E-constructed layer-preserving TRS whose signature is Σ_i, and $\Sigma_1 \cap \Sigma_2 = \Sigma_0$. If $=_E \circ \rightarrow_{R_1 \cup R_2} \circ =_E$ is terminating, then $(R_1 \cup R_2, E)$ is an E-constructed layer-preserving TRS, and for any $\Sigma_1 \backslash \Sigma_0$-rooted term t_1 and any $\Sigma_2 \backslash \Sigma_0$-rooted term t_2, t_1 cannot be equal to t_2 modulo $R_1 \cup R_2 \cup E$.*

By Lemma 3, the two assumptions of Lemma 2 can be satisfied, and this leads to a hierarchical unification procedure for the combined TRS. In the following, we consider a notion of decreasingness in order to study the termination of this unification procedure.

Definition 3 (Decreasingness)**.** *Consider a complexity measure defined as a mapping C from separate forms to natural numbers. A $H_E(U)$ inference system is said to be C-*decreasing *if for any separate form $G \cup G_0$ we have that (1) for any G' such that $G \cup G_0 \vdash_U G' \cup G_0$, the separate form of $G' \cup G_0$ does not increase C; (2) for any G'_0 such that $G \cup G_0 \vdash_{\textbf{\textit{Solve}}} G \cup G'_0$, then either the separate form of $G \cup G'_0$ is in normal form w.r.t $H_E(U)$, or it decreases C.*

Consequently, $H_E(U)$ is terminating if there exists some C such that $H_E(U)$ is C-decreasing.

Theorem 1. *Assume E is a subterm collapse-free theory such that an E-unification algorithm is known, and C is a complexity measure defined on separate forms. Let (R_1, E) and (R_2, E) be two E-constructed TRSs sharing only symbols in E such that, for $i = 1, 2$, (R_i, E) is layer-preserving, and $R_i \cup E$ admits a C-decreasing unification algorithm of the form $H_E(U_i)$. If $=_E \circ \rightarrow_{R_1 \cup R_2} \circ =_E$ is terminating, then $(R_1 \cup R_2, E)$ is an E-constructed TRS such that $(R_1 \cup R_2, E)$ is layer-preserving, and $R_1 \cup R_2 \cup E$ admits a C-decreasing unification algorithm of the form $H_E(U_1 \cup U_2)$.*

Example 2. *Consider the theories $\mathcal{E}_{AC}$ and $\mathcal{F}_{AC}$ introduced in Proposition 3 and the corresponding hierarchical unification algorithms $H_{AC}(U_\mathcal{E})$ and $H_{AC}(U_\mathcal{F})$ where the mutation rules defining $U_\mathcal{E}$ and $U_\mathcal{F}$ can be found in [6]. Let SVC be the complexity measure defined as follows: given a $R \cup E$-unification problem in separate form $G \cup G_0$, $SVC(G \cup G_0)$ is the number of equivalence classes of variables shared by G and G_0 that are variables abstracting $\Sigma \backslash \Sigma_0$-rooted terms.*

One can check that the unification algorithms $H_{AC}(U_\mathcal{E})$ and $H_{AC}(U_\mathcal{F})$ are both SVC-decreasing. By Theorem 1, we get that $\mathcal{E}_{AC} \cup \mathcal{F}_{AC}$ admits a SVC-decreasing unification algorithm of the form $H_{AC}(U_\mathcal{E} \cup U_\mathcal{F})$. We suspect that this complexity measure, SVC, could be useful for proving termination in other theories.

4.2 Combining Forward-Closed E-Constructed TRSs

The union of two forward-closed E-constructed TRSs remains a forward-closed E constructed TRS. Thus, a hierarchical unification algorithm can be constructed in a modular way in unions of forward-closed E-constructed TRSs.

Theorem 2. *Assume E is a regular and collapse-free theory such that an E-unification algorithm is known. Let (R_1, E) and (R_2, E) be two forward-closed E-constructed TRSs sharing only symbols in E. Then $R_1 \cup R_2 \cup E$ admits a unification algorithm of the form $H_E(BSM_{R_1} \cup BSM_{R_2})$.*

References

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

[2] F. Baader and K. U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *Journal of Symbolic Computation*, 21(2):211 – 243, 1996.

[3] H. Comon-Lundh and S. Delaune. The finite variant property: How to get rid of some algebraic properties. In J. Giesl, editor, *Rewriting Techniques and Applications*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005.

[4] A. K. Eeralla, S. Erbatur, A. M. Marshall, and C. Ringeissen. Rule-based unification in combined theories and the finite variant property. In C. Martín-Vide, A. Okhotin, and D. Shapira, editors, *Language and Automata Theory and Applications - 13th International Conference, LATA 2019, St. Petersburg, Russia, March 26-29, 2019, Proceedings*, volume 11417 of *Lecture Notes in Computer Science*, pages 356–367. Springer, 2019.

[5] S. Erbatur, D. Kapur, A. M. Marshall, P. Narendran, and C. Ringeissen. Hierarchical combination. In M. P. Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 249–266. Springer, 2013.

[6] S. Erbatur, A. M. Marshall, D. Kapur, and P. Narendran. Unification over distributive exponentiation (sub)theories. *Journal of Automata, Languages and Combinatorics (JALC)*, 16(2–4):109–140, 2011.

[7] S. Escobar, R. Sasse, and J. Meseguer. Folding variant narrowing and optimal variant termination. *J. Log. Algebr. Program.*, 81(7-8):898–928, 2012.

[8] D. Kim, C. Lynch, and P. Narendran. Reviving basic narrowing modulo. In A. Herzig and A. Popescu, editors, *Frontiers of Combining Systems - 12th International Symposium, FroCoS 2019, London, UK, September 4-6, 2019, Proceedings*, volume 11715 of *Lecture Notes in Computer Science*, pages 313–329. Springer, 2019.

[9] C. Lynch and B. Morawska. Basic syntactic mutation. In A. Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 2002.

[10] M. Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. *Journal of Symbolic Computation*, 8:51–99, July 1989.

[11] E. Tidén and S. Arnborg. Unification problems with one-sided distributivity. *Journal of Symbolic Computation*, 3(1/2):183–202, 1987.

Unification of Drags

Jean-Pierre Jouannaud[1] and Fernando Orejas[2]

[1] Université de Paris-Saclay, Laboratoire de Spécification et Vérification, France
[2] Universitat Politècnica de Catalunya, Barcelona, Spain

1 Introduction

Our work is based on a recent, purely combinatorial view of graphs [1]. Drags are labelled graphs equipped with roots and sprouts. Roots are vertices without predecessors that can be seen as input ports, while sprouts are vertices without successors labelled by variables that can be seen as output ports. Drags appear as a generalization of terms which admit many roots, arbitrary sharing, and cycles. Rewrite rules are then pairs of drags that preserve variables and roots, hence avoiding the creation of dangling edges when rewriting. A key aspect of drags is that they can be equipped with a composition operator so that matching a left-hand side of rule L w.r.t. an input drag D amounts to write D as the composition of a context graph C with L, and rewriting D with the rule L $\rightarrow$ R amounts to replace L with R in that composition. Since substitutions cannot be separated from context in presence of cycles, composition must play both rôles of plugging a context and a substitution.

To assess our claim that drags are a natural generalization of terms, it is our program to extend the most useful term rewriting techniques to drags: here, unification; in a paper to be presented at IWC 2020, checking confluence by means of critical pairs.

After generalizing the term subsumption order to drags thanks to composition, we show that unifiable drags admit a most general unifier which can be computed in quadratic time.

2 The Drag Model [1]

Drags are finite ***d***irected ordered ***r***ooted l***a***beled multi-***g***raphs. Vertices with no outgoing edges are designated *sprouts*. Other vertices are *internal*. We presuppose: a set of function symbols Σ, whose elements, equipped with a fixed arity, are used as labels for internal vertices; and a set of nullary variable symbols Ξ, disjoint from Σ, used to label sprouts.

Definition 1 (Drags)**.** A *drag* D is a tuple $\langle V, R, L, X, S\rangle$, where

1. V is a finite set of *vertices*;
2. $R : [p \mathinner{..} p + |R|] \rightarrow V$ is a finite list of vertices, called *roots*; $R(p+n)$ refers to the nth root in R; unless otherwise stated, $p = 1$; we denote R by $\mathcal{R}(D)$;
3. $S \subseteq V$ is a set of *sprouts*, leaving $V \setminus S$ to be the *internal* vertices;
4. $L : V \rightarrow \Sigma \cup \Xi$ is the *labeling* function, mapping internal vertices $V \setminus S$ to labels from the vocabulary Σ and sprouts S to labels from the vocabulary Ξ, writing $v : f$ for $f = L(v)$;
5. $X : V \rightarrow V^*$ is the *successor* function, also used relationaly, mapping each vertex $v \in V$ to a list of vertices in V whose length equals the arity of its label (that is, $|X(v)| = |L(v)|$).

The reflexive-transitive closure X^* of the relation X is called *accessibility*. Vertex v is *accessible* if it is accessible from some root. A drag is *clean* if all its vertices are accessible.

Terms as ordered trees, sequences of terms (forests), terms with shared subterms (dags) and sequences of dags (jungles) are all particular kinds of clean rooted drags.

Any vertex may be a root, we do not assume that roots have no predecessors.

We use $\mathcal{V}ar(D)$ for the set of variables labeling the sprouts of D. A drag is *linear* if no two sprouts have the same label, in which case variables and sprouts can be identified.

Given two drags U, V that share no vertices, we denote by $U \oplus V$ their juxtaposition defined as expected, so that $\mathcal{R}(U \oplus V) = \mathcal{R}(u)\,\mathcal{R}(v)$.

2.1 Drag composition

A variable in a drag should be understood as a potential connection to a root of another drag, as specified by a connection device called a *switchboard*. A switchboard ξ is a pair of partial injective functions, one for each drag, whose *domain* $\mathcal{D}om(\xi)$ and *image* $\mathcal{I}m(\xi)$ are a set of sprouts of one drag and a set of positions in the list of roots of the other, respectively.

Definition 2 (Switchboard)**.** Let $D = \langle V, R, L, X, S\rangle$ and $D' = \langle V', R', L', X', S'\rangle$ be drags. A *switchboard* ξ for D, D' is a pair $\langle \xi_D : S \to [1\,..\,|R'|]; \xi_{D'} : S' \to [1\,..\,|R|]\rangle$ of partial injective functions such that

1. $s \in \mathcal{D}om(\xi_D)$ and $L(s) = L(t)$ imply $t \in \mathcal{D}om(\xi_D)$ and $\xi_D(s) = \xi_D(t)$ for all sprouts $s, t \in S$;
2. $s \in \mathcal{D}om(\xi_{D'})$ and $L'(s) = L'(t)$ imply $t \in \mathcal{D}om(\xi_{D'})$ and $\xi_{D'}(s) = \xi_{D'}(t)$ for all $s, t \in S$;
3. ξ is *well-behaved*: it does not induce any cycle among sprouts, using ξ, R, R' relationally: $\nexists\ n > 0, s_1, \ldots, s_{n+1} \in S, t_1, \ldots, t_n \in S', s_1 = s_{n+1}.\ \forall i \in [1\,..\,n].\ s_i\ \xi_D R' X'^*\ t_i\ \xi_{D'} R X^*\ s_{i+1}$

The pair $\langle D', \xi\rangle$ is an *extension* of D, a *rewriting extension* if ξ_D is surjective and $\xi_{D'}$ total.

Sprouts labelled by the same variable should be connected to the same vertex, as required by conditions (1,2). These conditions are of course automatically satisfied by switchboards ξ, called *linear*, defined for sprouts whose variables are all different. It follows that $\xi_D(\mathcal{D}om(\xi_D))$ must be a set, making the set difference $[1\,..\,|R'|] \setminus \xi_D(\mathcal{D}om(\xi_D))$ well defined.

We now move to the composition operation on drags induced by a switchboard. The essence of this operation is that the (disjoint) union of the two drags is formed, but with sprouts in the domain of the switchboards merged with the roots to which the switchboard images refer. Merging sprouts with their images requires one to worry about the case where multiple sprouts are merged successively, when the switchboards map sprout to rooted-sprout to rooted-sprout, until, eventually, an internal vertex of one of the two drags must be reached because a switchboard is well-behaved. That vertex is called *target*:

Definition 3 (Target)**.** Let $D = \langle V, R, L, X, S\rangle$ and $D' = \langle V', R', L', X', S'\rangle$ be drags such that $V \cap V' = \varnothing$, and ξ be a switchboard for D, D'. The *target* $\xi^*(s)$ is a mapping from sprouts in $S \cup S'$ to vertices in $V \cup V'$ defined as follows:

Let $v = R'(n)$ if $s \in S$, and $v = R(n)$ if $s \in S'$, where $n = \xi(s)$.

1. If $v \in (V \cup V') \setminus (S \cup S')$, then $\xi^*(s) = v$.
2. If $v \in (S \cup S') \setminus \mathcal{D}om(\xi)$, then $\xi^*(s) = v$.
3. If $v \in \mathcal{D}om(\xi)$, then $\xi^*(s) = \xi^*(v)$.

The target mapping $\xi^*(\_)$ is extended to all vertices of D and D' by letting $\xi^*(v) = v$ when $v \in (V \setminus S) \cup (V' \setminus S')$.

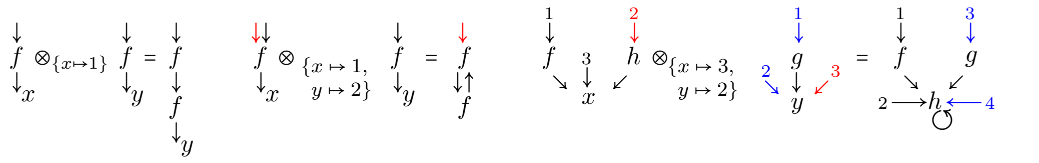
Figure 1: Different forms of composition: substitution, formation of a cycle, and transfer of roots.

Example 1. Consider the last of the three examples in Figure 1, in which a drag D, whose list of roots is $R = [f\,h\,x]$ (identifying vertices with their label) is composed with a second drag whose list of roots is $R' = [g\,y\,y]$, via the switchboard $\{x \mapsto 3, y \mapsto 2\}$. We calculate the target of the two sprouts: $x\,\xi\,3\,R'\,y\,\xi\,2\,R\,h$; hence $\xi^*(x) = \xi^*(y) = h$.

We are now ready for defining the composition of two drags. Its set of vertices will be the union of two components: the internal vertices of both drags, and their sprouts which are not in the domain of the switchboard. The labeling is inherited from that of the components.

Definition 4 (Composition)**.** Let $D = \langle V, R, L, X, S\rangle$ and $D' = \langle V', R', L', X', S'\rangle$ be drags such that $V \cap V' = \varnothing$, and let ξ be a switchboard for D, D'. Their *composition* is the drag $D \otimes_\xi D' = \langle V'', R'', L'', X'', S''\rangle$, with interface (R'', S'') denoted $(R, S) \otimes_\xi (R', S')$, where

1. $V'' = (V \cup V') \setminus \mathcal{D}om(\xi)$;
2. $S'' = (S \cup S') \setminus \mathcal{D}om(\xi)$;
3. $R'' = \xi^*(R([1\,..\,|R|] \setminus \xi_{D'}(\mathcal{D}om(\xi_{D'})))) \cup \xi^*(R'([1\,..\,|R'|] \setminus \xi_D(\mathcal{D}om(\xi_D))))$;
4. $L''(v) = L(v)$ if $v \in V \cap V''$; and $L''(v) = L'(v)$ if $v \in V' \cap V''$;
5. $X''(v) = \xi^*(X(v))$ if $v \in V \setminus S$; and $X''(v) = \xi^*(X'(v))$ if $v \in V' \setminus S$

If $\langle \xi_D, \xi\rangle$ is a rewriting extension of D', then *all* roots and sprouts of D' disappear in the composed drag. The drag D can then be seen as the context of the left-hand side of a rule $D' \to R$, where R must have the same number of roots as D' (and $\mathcal{V}ar(R) \subseteq \mathcal{V}ar(D')$.)

Example 2. We show in Figure 1 three examples of compositions. The first is a substitution of term. The second uses a bi-directional switchboard, which induces a cycle. In that example, the remaining root is the first (red) root of the first drag which has two roots, the first red, the other black. The third example shows how sprouts that are also roots connect to roots in the composition (colors black and blue indicate roots' origin, while red indicates a root that disappears in the composition). Since x points at y and y at the second root of the first drag, a cycle is created on the vertex of the resulting drag which is labelled by h. Further, the third root of the first drag has become the second root of the result, while the first (resp., second) root of the second drag has become the third (resp., fourth) root of the result. This agrees of course with the definition, as shown by the following calculations (started in Example 1): $\xi^*([1, 2, 3] \setminus [2]) = \xi^*([1, 3]) = [f, h]$; and $\xi^*([1, 2, 3] \setminus [2]) = \xi^*([1, 2]) = [g, h]$, hence the list of roots of the resulting drag is $[f, h, g, h]$.

A clean linear drag all of whose vertices are its sprouts, whose set of edges is empty, and whose list of roots is a list of its sprouts, is called an *identity*. We denote it by 1_X^Y, where X is its set of sprouts and Y is its list of roots. We use $\varnothing$ for the identity empty drag $1_\varnothing^\varnothing$.

Composition has important algebraic properties, associativity and the above identities, that play a key rôle in the construction of most general unifiers and its justification.

3 Unification

The purpose of this section is to *identify* two clean drags U, V by composing them with the same *minimal* rewriting context $\langle C, \xi \rangle$, resulting in the same drag W. An identification corresponds to the fact that we want the same drag to be rewritten by two different rewrite rules whose left-hand sides are U and V. In order for $C \otimes_\xi U$ and $C \otimes_\xi V$ to both make sense, we assume that U, V are renamed apart (variables and root numbers).

Definition 5. Given drags U, V, we call *partner vertices* two lists L_U, L_V of equal length of internal vertices of U and V, respectively, such that no two vertices $u, u' \in L_U$ (resp., $v, v' \in L_V$) are in relationship with X_U (resp., X_V).

Definition 6. Two drags U, V are *identified* with a drag W at *partner vertices* $(\overline{u}, \overline{v})$ by an injective function $o : \mathcal{V}er(U) \cup \mathcal{V}er(V) \to \mathcal{V}er(W)$ called *identification*, written $U[\overline{u}] =_o V[\overline{v}]$, iff:

1. $o(\overline{u}) = o(\overline{v})$;
2. $\forall w \in \mathcal{V}er(U), w' \in \mathcal{V}er(V)$ such that $o(w) = o(w')$, $w : f$ iff $w' : f$ iff $o(w) = o(w') : f$;
3. $\forall w \in \mathcal{V}er(U), w' \in \mathcal{V}er(V)$ such that $o(w) = o(w')$, $o(X_U(w)) = o(X_V(w))$.

While two terms u, v are unified *at their root*, the solution being a substitution σ such that $u\sigma = v\sigma$, two drags U, V are unified at partner vertices $(\overline{u}, \overline{v})$, the solution being an extension $\langle C, \xi \rangle$ of both U and V that identifies $C \otimes_\xi U$ and $C \otimes_\xi V$ at these partner vertices:

Definition 7. A *unification problem* is a pair of clean drags (U, V) that are renamed apart, together with partner vertices $P = \{(\overline{u}, \overline{v})\}$, which we write $U[\overline{u}] = V[\overline{v}]$. A *solution* (or *unifier*) to the unification problem $U[\overline{u}] = V[\overline{v}]$ is a clean rewriting extension $\langle C, \xi \rangle$ such that the *overlap* drags $C \otimes_\xi U$ and $C \otimes_\xi V$ are identified at P. A unification problem $U[\overline{u}] = V[\overline{v}]$ is *solvable* if it has a solution.

Example 3. Let $U = f(h(x))$ and $V = f(h(a))$, in which U has two roots, f and h in this order, and V has two roots h and f in this order. These roots are numbered $1, 2, 3, 4$. Let the partner vertices be $\{(h, h)\}$. Then, the corresponding unification problem has for solution the rewriting extension $\langle C, \xi \rangle$ such that $C = a \oplus f(y) \oplus f(z)$, which has three roots, a, f, f in this order, and $\xi = \{x \mapsto 1, y \mapsto 4, z \mapsto 2\}$. The overlap is the drag which has two roots labelled f and f in this order with the drag $h(a)$ being their common successor. Note that flipping the two roots of V would give another solvable unification problem, since the predecessors of a vertex are not ordered (unless they are also successors). Note also that the two root vertices of U and V, which are both labelled by the same function symbol f, remain distinct in the obtained overlap. □

We want unification to be minimal, that is, to capture all possible extensions that identify U and V, without useless identifications occuring above or below partner vertices.

Definition 8. We say that a drag U is an *instance* of a drag V, or that V *subsumes* U, and write $U \geq V$, if there exists a clean context extension $\langle C, \xi \rangle$ such that $U = C \otimes_\xi V$.

The (of course well-founded) subsumption quasi-order for drags, corresponds to *encompassment* of terms. On the other hand, its equivalence generalizes the case of terms, since encompassment and subsumption for terms have the same equivalence.

Propagate

$$U \oplus V \begin{bmatrix} & u : f \cdot \textcolor{red}{\mathbf{i}} & \\ \swarrow & & \searrow \\ s_1 & \dots & s_n \end{bmatrix}\begin{bmatrix} & v : f \cdot \textcolor{red}{\mathbf{i}} & \\ \swarrow & & \searrow \\ t_1 & \dots & t_n \end{bmatrix} \Rightarrow$$

$$U \oplus V \begin{bmatrix} & f \cdot \textcolor{blue}{\mathbf{i}} & \\ \swarrow & & \searrow \\ s_1 \cdot \textcolor{red}{\mathbf{c+1}} & \dots & s_n \cdot \textcolor{red}{\mathbf{c+n}} \end{bmatrix}\begin{bmatrix} & f \cdot \textcolor{blue}{\mathbf{i}} & \\ \swarrow & & \searrow \\ t_1 \cdot \textcolor{red}{\mathbf{c+1}} & \dots & t_n \cdot \textcolor{red}{\mathbf{c+n}} \end{bmatrix}$$

Variable case $\quad U \oplus V[s : x \cdot \textcolor{red}{\mathbf{c}}][u : f \cdot \textcolor{red}{\mathbf{c}}] \Rightarrow U \oplus V[s : x \cdot \textcolor{blue}{\mathbf{c}}][u : f \cdot \textcolor{blue}{\mathbf{c}}]$

Merge $\quad U \oplus V[s : x][t : x] \Rightarrow U \oplus V[s \cdot \textcolor{red}{\mathbf{c+1}}][t \cdot \textcolor{red}{\mathbf{c+1}}]$

Transitivity $\quad U \oplus V[u \cdot \textcolor{blue}{\mathbf{i}} \cdot \textcolor{blue}{\mathbf{j}}][v \cdot \textcolor{blue}{\mathbf{i}}][w \cdot \textcolor{blue}{\mathbf{j}}] \Rightarrow U \oplus V[v \cdot \textcolor{red}{\mathbf{c+1}}][w \cdot \textcolor{red}{\mathbf{c+1}}]$

Symbol conflict $\quad U \oplus V[u : f \cdot \textcolor{red}{\mathbf{i}}][v : g \cdot \textcolor{red}{\mathbf{i}}] \Rightarrow \bot \quad$ **if** $\quad f \neq g$

Internal conflict $\quad U \oplus V[u \cdot i][v \cdot i] \Rightarrow \bot$
if internal vertices u, v belong both either to U or to V

Occur check
$U \oplus V[s_1 : x_1 \cdot i_1][w_1 \cdot i_1] \dots [s_n : x_n \cdot i_n][w_n \cdot i_n] \Rightarrow \bot$

$$\textbf{if} \begin{cases} \forall i \in [1..n]\ s_i \text{ is a sprout and } w_i \text{ is an internal vertex} \\ \forall i \in [1..n]\ s_{i+1} \text{ (convention: } s_{n+1} = s_1\text{) is accessible from } w_i \\ \exists i \in [1..n]\ w_i \text{ is not rooted} \end{cases}$$

Figure 2: Drag unification rules

3.1 Unification algorithm

The unification algorithm is described by a set of transformation rules operating on the drag $U \oplus V$. We single out a vertex w in the drag $U \oplus V$ by writing $U \oplus V[w]$, a notation that extends as expected to several vertices that are pairwise different.

Identifying $C \otimes_\xi U$ and $C \otimes_\xi V$ at a pair of vertices (u, v) requires that u and v have the same label, and that the property can be recursively propagated to their corresponding pairs of successors. To organize the propagation, the initial partner vertices will hold marks $\textcolor{red}{\mathbf{1}}, \dots \textcolor{red}{|\overline{\mathbf{u}}|}$). Assuming now that the the pair (u, v) is marked with a red natural number, propagation will turn this mark into blue, while marking the pairs of succesors with fresh red marks. In case one of u, v is a sprout, no propagation occurs, it is enough to turn the red mark into blue. Our syntax for marking a vertex is as in $u : f \cdot \mathbf{i_1} \cdots \mathbf{i_n}$ if u has label f and (blue or red) marks $\mathbf{i_1}, \dots, \mathbf{i_n}$. Vertex u, label f or marks may be omitted when convenient.

Propagation stops when there are no more pairs of internal vertices holding a red mark, unless two marked sprouts hold the same variable, in which case they must be marked red.

We call c the number of already used marks, initialized to 0, and incremented by one at each use of a mark, including the initial marking of the partner vertices. The rules are reminiscent from the unification rules for terms, although we don't use the same rule names except for *Merge* and *Occur check*. For example, we use *Propagate* rather than *Decompose* to stress the fact that drags cannot be treated as terms.

The procedure described at Figure 2 computes an equivalence relation between the vertices of two drags to be unified from which their most general unifier will be extracted. It consists in a set of transformation rules operating on the unification problem $U = V$ (actually, on the drag

$U \oplus V$) by marking pairs of vertices with elements of an initial segment of the natural numbers, figuring out new edges of a specific kind between vertices of $U \oplus V$, in the style of Patterson and Wegman unification algorithm, as well as Huet's.

Example 4. In our example of Figure 3, unification of the initial two drags proceeds in eleven steps. *Propagation* steps are labelled by the red mark processed while *Transitivity* steps are labelled by the generated mark. This explains why some steps have the same label.

Soundness of these rules is based on the notion of *generated equivalence* between the vertices of the generated drag, which corresponds to a congruence on terms. Completeness requires specific generated equivalences, called solved form which allow an occur-check, that is a vertex v marked n accessible from some vertex u marked n, iff u is rooted.

Example 5 (Continued)**.** Figure 4 shows the context drag, switchboard, and overlapping drag obtained by composition of the inputs drags of Figure 3. The equivalence on vertices considered here is defined by having equal markings, which is in solved form. Note that the resulting drag has two roots. Unifying at vertices $(r{:}1, r{:}2)$ instead would give a single root.

Theorem 1. *Unification is unitary, and has quadratic worst case complexity.*

References

[1] Nachum Dershowitz and Jean-Pierre Jouannaud. Drags: A compositional algebraic framework for graph rewriting. *Theor. Comput. Sci.*, 777:204–231, 2019.

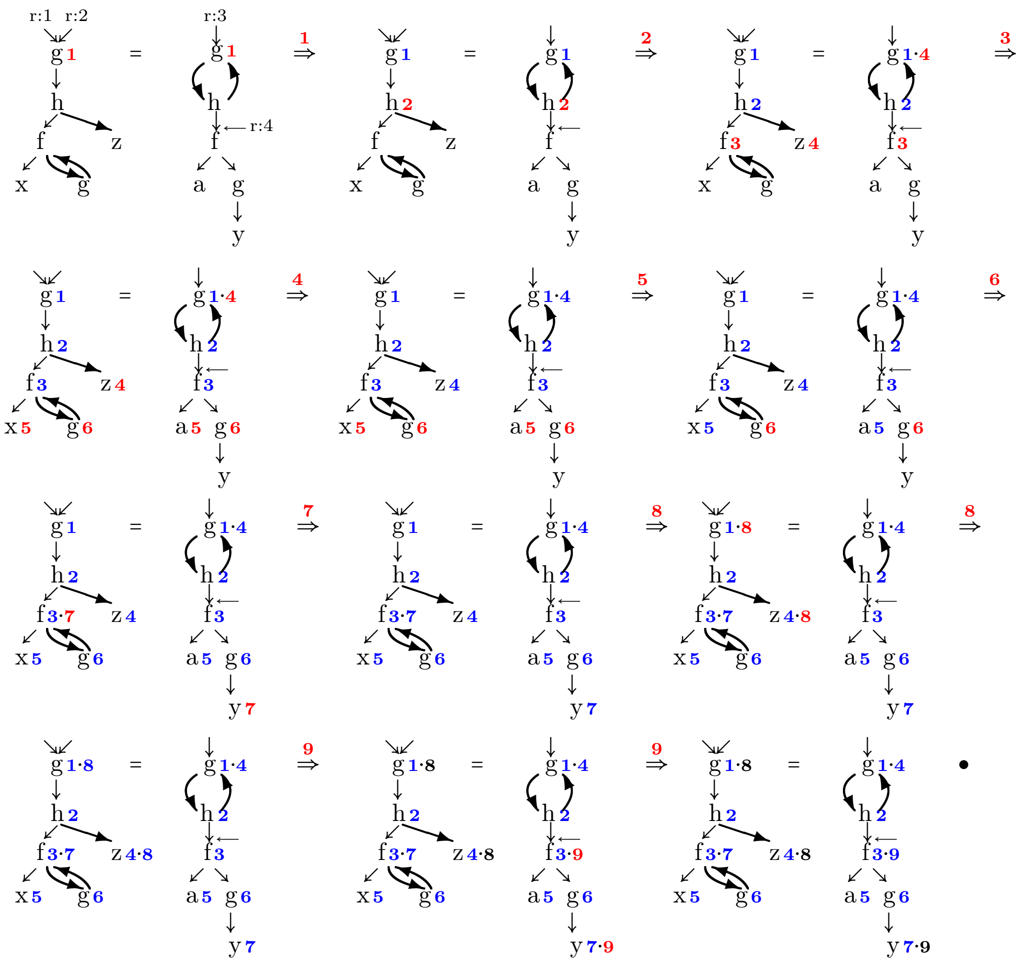
Figure 3: Successful unification of two patterns.

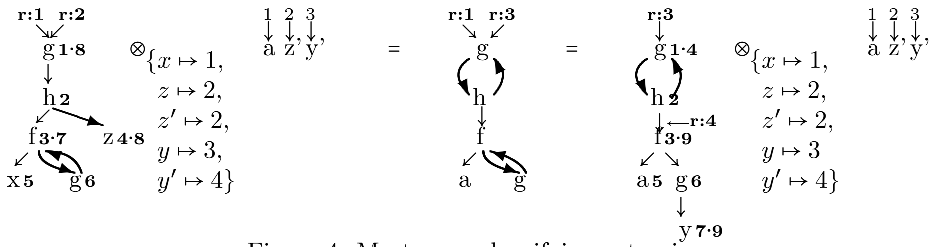
Figure 4: Most general unifying extension

Proximity-Based Unification with Arity Mismatch

Temur Kutsia and Cleo Pau

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

Abstract

Proximity relations are binary fuzzy relations, which are reflexive and symmetric, but not transitive. They can thus define distances between symbols and, by extension, to terms. We propose an algorithm that finds all the substitutions that bring close to each other two terms whose signatures tolerate mismatches in function symbol names, arity, and in the arguments order (so called full fuzzy signatures). This work generalizes on the one hand, proximity-based unification to full fuzzy signatures, and on the other hand, similarity-based unification over a full fuzzy signature by extending similarity to proximity.

1 Introduction

The classical unification fails when there is no match between two corresponding function symbols of the terms to be unified. While in most situations this is the desired outcome, there are cases when some tolerance regarding the mismatches would offer a better result. The type of the accepted differences can vary, and some mismatches were already explored in previous researches.

Mismatch between symbol names under similarity and proximity. The work of Sessa [8] covers the case of unification with similar symbols with the same arity, where similarity is a fuzzy equivalence relation. Julián-Iranzo and Rubio-Manzano [3–5] consider a proximity (fuzzy reflexive, symmetric, non-transitive) relation between symbols of the same arity, with the restriction that once a symbol is considered as being a neighbor to another symbol, it cannot become a proximal candidate in a different neighborhood class. We treated the unrestricted proximity unification with mismatches between same arity symbols in [7].

Mismatch between symbol names and arities under similarity. Aït-Kaci and Pasi [1] explored unification in a class of similarity relations over symbols with different arities, by introducing injective mappings between the arguments of the similar functions, specifying which argument pairs should be considered similar. The mappings are defined over all the arguments of the symbol with the smaller arity.

In this paper we generalize the previous contributions by considering proximity relations over function symbols with different arity, while also extending the parameters mismatches to allow mappings on subsets of arguments. Hence, we allow mismatch between symbol names and arities under unrestricted proximity.

2 Preliminaries

Proximity relations. We define the basic notions about proximity relations according to [3]. Given a set S, a mapping from $S \times S$ to the real interval $[0, 1]$ is called a binary *fuzzy relation* on S. By fixing a number λ, $0 \leq \lambda \leq 1$, we can define the crisp counterpart of $\mathcal{R}$, named the λ-*cut* of $\mathcal{R}$ on S, as $\mathcal{R}_\lambda := \{(s_1, s_2) \mid \mathcal{R}(s_1, s_1) \geq \lambda\}$. We take the minimum as the T-norm $\wedge$. A *proximity relation* on a set S is a reflexive and symmetric fuzzy relation $\mathcal{R}$ on S.

Terms and substitutions. We consider first-order *terms* defined as usual: $t := x \mid f(t_1, \dots, t_n)$, where $x \in \mathcal{V}$ is a variable and $f \in \mathcal{F}$ is an n-ary function symbol with $n \geq 0$. We denote arbitrary function symbols by f, g, h, constants by a, b, c, variables by x, y, z, v, and terms by s, t, r. The *head* of a term is defined as $head(x) := x$ and $head(f(t_1, \dots, t_n)) := f$. For a term t, we denote with $var(t)$ the set of all variables appearing in t.

A *substitution* is a mapping from variables to terms, which is the identity almost everywhere. We use the Greek letters $\sigma, \vartheta, \varphi$ to denote substitutions, except for the identity substitution which is written as Id. We represent substitutions with the usual set notation. The restriction of a substitution σ on a set of variables V is denoted by $\sigma|_V$ and defined in the usual way. The notion of more generality for substitutions is defined with the help of syntactic equality: σ is more general than ϑ, written $\sigma \preceq \vartheta$, if there exists φ such that $\sigma\varphi = \vartheta$.[1]

Position mappings. Given two sets $\{1, \dots, n\}$ and $\{1, \dots, m\}$, a *position mapping* is an injective function $\pi : I_n \mapsto I_m$, where $I_n \subseteq \{1, \dots, n\}$, $I_m \subseteq \{1, \dots, m\}$ and $|I_n| = |I_m|$. Note that it can be also the empty mapping: $\pi : \emptyset \mapsto \emptyset$. Usually, for $\pi : I_n \mapsto I_m$ we write $\pi = \{i \mapsto \pi(i) \mid i \in I_n\}$.

Given a proximity relation $\mathcal{R}$ over $\mathcal{F}$, we assume that to any pair of function symbols f and g with $\mathcal{R}(f, g) = \lambda > 0$, there is an attached position mapping π such that if f is n-ary and g is m-ary, then π is a mapping from $\{1, \dots, n\}$ to $\{1, \dots, m\}$. (π is the identity if $f = g$, and is the empty mapping if f or g is a constant.) We use the notation $f \sim^{\pi}_{\mathcal{R},\lambda} g$.

Proximity relations over terms. Each proximity relation $\mathcal{R}$ considered in this paper is defined on $\mathcal{F} \cup \mathcal{V}$ such that $\mathcal{R}(f, x) = 0$ for all $f \in \mathcal{F}$ and $x \in \mathcal{V}$, and $\mathcal{R}(x, y) = 0$ for all $x \neq y$, and $x, y \in \mathcal{V}$.

We extend such an $\mathcal{R}$ to terms as follows: (i) $\mathcal{R}(s, t) := 0$ if $\mathcal{R}(head(s), head(t)) = 0$; (ii) $\mathcal{R}(s, t) := 1$ if $s = t$ and $s, t \in \mathcal{V}$; (iii) $\mathcal{R}(s, t) := \mathcal{R}(f, g) \wedge \mathcal{R}(s_{i_1}, t_{j_1}) \wedge \cdots \wedge \mathcal{R}(s_{i_k}, t_{j_k})$, if $s = f(s_1, \dots, s_n)$, $t = g(t_1, \dots, t_m)$, $f \sim^{\pi}_{\mathcal{R},\lambda} g$, and $\pi = \{i_1 \mapsto j_1, \dots, i_k \mapsto j_k\}$.

Unification problems, unifiers. We write $(\mathcal{R}, \lambda)$-*equations* between terms as $t \simeq^?_{\mathcal{R},\lambda} s$, with the question mark indicating that they are supposed to be solved (i.e., the terms t and s to be $(\mathcal{R}, \lambda)$-*unified*). A *solution* (a *unifier*) of such an equation is a substitution σ such that $t\sigma \simeq_{\mathcal{R},\lambda} s\sigma$. We say that $\mathcal{R}(t\sigma, s\sigma) \geq \lambda$ is the *approximation degree* of $(\mathcal{R}, \lambda)$-unifying t and s by σ (or, equivalently, the *approximation degree* of solving $t \simeq^?_{\mathcal{R},\lambda} s$ by σ).

An $(\mathcal{R}, \lambda)$-*unification problem* (or, briefly, a *unification problem*) is a finite set of $(\mathcal{R}, \lambda)$-equations. A *solution* (*unifier*) of a unification problem P is a substitution that solves all the equations in P. The approximation degree of the unification of P by σ is obtained by $\wedge_{eq \in P}\, deg(eq\sigma)$, where $deg(eq\sigma)$ is the approximation degree of solving $eq \in P$ by σ.

3 Unification rules

The unification rules work on triples $P; \sigma; \alpha$, called configurations, where P is a unification problem, σ is a substitution computed so far, and α is the approximation degree, also computed so far. The symbol $\perp$ is a special configuration. The rules transform configurations into configurations ($\mathcal{R}$ and λ are assumed to be given):

[1]Note that we did not use proximity in the definition of more generality, in order to guarantee that $\preceq$ is a quasi-order, preserving good properties of unifiers. See Remark 1 in [7].

Tri: **Trivial**

$\{x \simeq^?_{\mathcal{R},\lambda} x\} \uplus P; \sigma; \alpha \Longrightarrow P; \sigma; \alpha.$

Dec: **Decomposition**

$\{f(t_1,\dots,t_n) \simeq^?_{\mathcal{R},\lambda} g(s_1,\dots,s_m)\} \uplus P; \sigma; \alpha \Longrightarrow P \cup \{t_{i_1} \simeq^?_{\mathcal{R},\lambda} s_{j_1}, \dots, t_{i_k} \simeq^?_{\mathcal{R},\lambda} s_{j_k}\}; \sigma; \alpha \wedge \beta,$
if $f \sim^{\pi}_{\mathcal{R},\lambda} g$ with $\pi = \{i_1 \mapsto j_1, \dots, i_k \mapsto j_k\}$ and $\mathcal{R}(f,g) = \beta \geq \lambda$, $n, m, k \geq 0$.

Cla: **Clash**

$\{f(t_1,\dots,t_n) \simeq^?_{\mathcal{R},\lambda} g(s_1,\dots,s_m)\} \uplus P; \sigma; \alpha \Longrightarrow \bot, \qquad$ if $\mathcal{R}(f,g) < \lambda$.

Ori: **Orient**

$\{t \simeq^?_{\mathcal{R},\lambda} x\} \uplus P; \sigma; \alpha \Longrightarrow P \cup \{x \simeq^?_{\mathcal{R},\lambda} t\}; \sigma; \alpha, \qquad$ if t is not variable.

Occ: **Occurrence check**

$\{x \simeq^?_{\mathcal{R},\lambda} g(s_1,\dots,s_n)\} \uplus P; \sigma; \alpha \Longrightarrow \bot,$
if for each $f \sim^{\pi}_{\mathcal{R},\lambda} g$, there exists $i \mapsto j \in \pi$ such that $x \in var(s_j)$.

$\mathsf{Var\text{-}E}$: **Variable elimination**

$\{x \simeq^?_{\mathcal{R},\lambda} g(s_1,\dots,s_n)\} \uplus P; \sigma; \alpha \Longrightarrow \Big(P \cup \{x \simeq^?_{\mathcal{R},\lambda} g(s_1,\dots,s_n)\}\Big)\vartheta; \sigma\vartheta; \alpha,$
where $n \geq 0$, $f \sim^{\pi}_{\mathcal{R},\lambda} g$, f is m-ary, $\vartheta = \{x \mapsto f(v_1,\dots,v_m)\}$ where $v_1,\dots,v_m$ are fresh variables, and for each $i \mapsto j \in \pi$ we have $x \notin var(s_j)$.

Given a unification problem P, we create the initial system $P; Id; 1$ and start applying the unification rules in all possible ways, generating a complete tree of derivations in the standard way. The $\mathsf{Var\text{-}E}$ rule causes branching, since there can be multiple f's satisfying the condition there. No rule applies to $\bot$ (indicating failure) or to a configuration of the form $\{x_1 \simeq^?_{\mathcal{R},\lambda} v_1, \dots, x_n \simeq^?_{\mathcal{R},\lambda} v_n\}; \sigma; \alpha$, $n \geq 0$, called *variables-only* configuration. In the latter case we say that α is the computed approximation degree, $\sigma|_{var(P)}$ is the computed substitution and $\{x_1 \simeq^?_{\mathcal{R},\lambda} v_1, \dots, x_n \simeq^?_{\mathcal{R},\lambda} v_n\}$ is the computed constraint. We denote the obtained unification algorithm by $\mathcal{U}$.

In the examples below it is assumed that $\mathcal{R}(sym_1, sym_2) = 0$ for any pair of symbols sym_1 and sym_2 except those for which the proximity is explicitly given.

Example 1. Let f be a binary function symbol with $f \sim^{\{1\mapsto 1\}}_{\mathcal{R},0.6} g$, $f \sim^{\{2\mapsto 1\}}_{\mathcal{R},0.7} h$, and a, b, c be constants such that $b \sim_{\mathcal{R},0.4} c$. Let the unification problem be $P = \{f(x,x) \simeq_{\mathcal{R},0.4} f(g(a), h(c))\}$. Then the algorithm $\mathcal{U}$ starts with decomposition:

$$\{f(x,x) \simeq^?_{\mathcal{R},0.4} f(g(a),h(c))\}; Id; 1 \Longrightarrow_{\mathsf{Dec}}$$
$$\{x \simeq^?_{\mathcal{R},0.4} g(a),\ x \simeq^?_{\mathcal{R},0.4} h(c)\}; Id; 1.$$

From here there are two ways to proceed by $\mathsf{Var\text{-}E}$ on $x \simeq^?_{\mathcal{R},0.4} g(a)$: by choosing the variable eliminating substitution either $\{x \mapsto g(v)\}$ or $\{x \mapsto f(v_1, v_2)\}$. The former one leads to failure, since $\mathcal{R}(g,h) = 0$. Therefore, we show here only the second derivation:

$$\{x \simeq^?_{\mathcal{R},0.4} g(a),\ x \simeq^?_{\mathcal{R},0.4} h(c)\}; Id; 1 \Longrightarrow_{\mathsf{Var\text{-}E}}$$
$$\{f(v_1,v_2) \simeq^?_{\mathcal{R},0.4} g(a),\ f(v_1,v_2) \simeq^?_{\mathcal{R},0.4} h(c)\}; \{x \mapsto f(v_1,v_2)\}; 1 \Longrightarrow_{\mathsf{Dec}}$$

$\{v_1 \simeq^?_{\mathcal{R},0.4} a,\ f(v_1,v_2) \simeq^?_{\mathcal{R},0.4} h(c)\}; \{x \mapsto f(v_1,v_2)\}; 0.6 \Longrightarrow_{\mathsf{Var\text{-}E,\,Dec}}$

$\{f(a,v_2) \simeq^?_{\mathcal{R},0.4} h(c)\}; \{x \mapsto f(a,v_2), v_1 \mapsto a\}; 0.6 \Longrightarrow_{\mathsf{Dec}}$

$\{v_2 \simeq^?_{\mathcal{R},0.4} c\}; \{x \mapsto f(a,v_2), v_1 \mapsto a\}; 0.6.$

Here we have two alternatives by Var-E, both leading to success. The first alternative chooses $\{v_2 \mapsto b\}$ and gives the final configuration $\emptyset; \{x \mapsto f(a,b), v_1 \mapsto a, v_2 \mapsto b\}; 0.4$. The substitution computed in this derivation is $\sigma_1 = \{x \mapsto f(a,b)\}$. It solves P, because $f(f(a,b),f(a,b)) \simeq_{\mathcal{R},0.4} f(g(a),h(c))$.

The other alternative is to take $\{v_2 \mapsto c\}$. In this branch we get $\sigma_2 = \{x \mapsto f(a,c)\}$ and $\alpha = 0.6$. It also solves P, because $f(f(a,c),f(a,c)) \simeq_{\mathcal{R},0.4} f(g(a),h(c))$.

If we took the $\lambda = 0.6$, then σ_2 would be the only solution of P. For $\lambda > 0.6$, there would be no solution.

The previous example is, in fact, a matching problem since variables did not appear in the right side. As we saw, the computed constraint was empty. Now we consider a case when variables appear in both sides.

Example 2. Let f,g,h,a,b,c be as in Example 1 and consider the problem $P = \{f(x,x) \simeq^?_{\mathcal{R},0.4} f(g(y),h(z))\}$. Then the algorithm stops with the final configuration $S;\sigma;\alpha$ where $S = \{v_1 \simeq^?_{\mathcal{R},0.4} y, v_2 \simeq^?_{\mathcal{R},0.4} z\}$, $\sigma = \{x \mapsto f(v_1,v_2)\}$, and $\alpha = 0.6$. For illustration, we take three unifiers of P: ϑ_1, ϑ_2, and ϑ_3 together with their approximation degrees and show how they can be obtained from $S;\sigma$:

1.
   - $\vartheta_1 = \{x \mapsto f(y,z)\}$ and the approximation degree $\beta = 0.6$.
   - The instance of $S;\sigma$ under $\varphi = \{v_1 \mapsto y, v_2 \mapsto z\}$: $S\varphi = \{y \simeq^?_{\mathcal{R},0.4} y, z \simeq^?_{\mathcal{R},0.4} z\}$ and $\sigma\varphi = \{x \mapsto f(y,z), v_1 \mapsto y, v_2 \mapsto z\}$.
   - $S\varphi$ is solved, and $(\sigma\varphi)|_{var(P)} = \vartheta_1$. Besides, $\alpha \geq \beta$.
2.
   - $\vartheta_2 = \{x \mapsto f(a,b), y \mapsto a, z \mapsto b\}$ and the approximation degree $\beta = 0.6$.
   - The instance of $S;\sigma$ under $\varphi = \{v_1 \mapsto a, v_2 \mapsto b, y \mapsto a, z \mapsto b\}$: $S\varphi = \{a \simeq^?_{\mathcal{R},0.4} a, b \simeq^?_{\mathcal{R},0.4} b\}$ and $\sigma\varphi = \{x \mapsto f(a,b), v_1 \mapsto a, v_2 \mapsto b, y \mapsto a, z \mapsto b\}$.
   - $S\varphi$ is solved, and $(\sigma\varphi)|_{var(P)} = \vartheta_2$. Besides, $\alpha \geq \beta$.
3.
   - $\vartheta_3 = \{x \mapsto f(a,c), y \mapsto a, z \mapsto b\}$ and the approximation degree $\beta = 0.4$.
   - Instance of $S;\sigma$ under $\varphi = \{v_1 \mapsto a, v_2 \mapsto c, y \mapsto a, z \mapsto b\}$: $S\varphi = \{a \simeq^?_{\mathcal{R},0.4} a, c \simeq^?_{\mathcal{R},0.4} b\}$ and $\sigma\varphi = \{x \mapsto f(a,c), v_1 \mapsto a, v_2 \mapsto c, y \mapsto a, z \mapsto b\}$.
   - $S\varphi$ is solved, and $(\sigma\varphi)|_{var(P)} = \vartheta_3$. Besides, $\alpha \geq \beta$.

This example explains why the algorithm stops at variables-only configuration. If it went further from $S;\sigma;\alpha$ in the usual way, eliminating v_1 and v_2 by the substitution $\{v_1 \mapsto y, v_2 \mapsto z\}$, we would end up with the final configuration $\emptyset; \{x \mapsto f(y,z), v_1 \mapsto y, v_2 \mapsto z\}$, but the computed substitution $\{x \mapsto f(y,z)\}$ would not be more general than the unifier ϑ_3. (Recall that more generality is defined by syntactic equality, not by proximity.)

Example 3. This example shows why we can not simply have $x \in var(g(s_1,\dots,s_n))$ in the condition of Occ. Assume f and g are as in Example 2. Then $\{x \simeq^?_{\mathcal{R},0.4} f(a,x)\}$ has a unifier $\{x \mapsto g(a)\}$. If Occ is applied to this problem, we would get $\perp$ and, hence, would lose solutions.

Below we state the properties of the algorithm $\mathcal{U}$.

Theorem 1. *$\mathcal{U}$ terminates for any input.*

Theorem 2. *Let $P; Id; 1 \Longrightarrow^* S; \sigma; \alpha$ be a derivation in $\mathcal{U}$, and φ be a $(\mathcal{R}, \lambda)$-unifier of S with the approximation degree β. Then $\sigma\varphi$ is a $(\mathcal{R}, \lambda)$-unifier of P with the approx. degree $\alpha \wedge \beta$.*

Theorem 3. *Let P be a $(\mathcal{R}, \lambda)$-unification problem and ϑ be its unifier with the approximation degree β. Then there exists a derivation $P; Id; 1 \Longrightarrow^* S; \sigma; \alpha$ in $\mathcal{U}$ with $\alpha \geq \beta$ and a unifier φ of S such that $(\sigma\varphi)|_{var(P)} = \vartheta|_{var(P)}$.*

The NP-hardness of the decision problem of $(\mathcal{R}, \lambda)$-unifiability with arity mismatch can be shown by reduction from positive 1-in-3-SAT. Let $\pi_1 = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3\}$, $\pi_2 = \{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 2\}$, $\pi_3 = \{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 1\}$, $h_i \simeq^{\pi_1}_{\mathcal{R},\lambda} f$, $h_i \simeq^{\pi_i}_{\mathcal{R},\lambda} g$ for $1 \leq i \leq 3$. Then each positive 3-SAT clause $x_1 \vee x_2 \vee x_3$ can be encoded as two proximity equations $y \simeq^?_{\mathcal{R},\lambda} f(x_1, x_2, x_3)$, and $y \simeq^?_{\mathcal{R},\lambda} g(1, 0, 0)$, where 1 and 0 are constants. Their unifiers force exactly one x to be mapped to 1, and the other two to 0 ($\{y \mapsto h_1(1, 0, 0), x_1 \mapsto 1, x_2 \mapsto 0, x_3 \mapsto 0\}$, $\{y \mapsto h_2(0, 1, 0), x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 0\}$, and $\{y \mapsto h_3(0, 0, 1), x_1 \mapsto 0, x_2 \mapsto 0, x_3 \mapsto 1\}$). The reduction is polynomial and preserves solvability in both directions.

Reducing nondeterminism. The Var-E rule blindly chooses f from the proximity class of g. Eventually, the whole class will be explored. If x appears nowhere else except the equation transformed by Var-E, then this approach is reasonable as it generates all necessary unifiers. However, if there is another occurrence of x, it may happen that some of those f's do not lead to success. In such cases, we can cut the number of alternatives doing more informed selection.

The idea is simple: (1) postpone variable elimination as much as possible; (2) when all equations have a variable in the left hand side, collect all those with the same variable in the left and a non-variable terms in the right; (3) apply variable elimination only with those f's that belong to the intersection of proximity classes of the head symbols in the right hand side. Hence, the modified Var-E rule will have the form

Var-E-Mod: **Variable elimination modified**

$$\{x \simeq^?_{\mathcal{R},\lambda} g_1(s^1_1, \dots, s^1_{k_1}), \dots, x \simeq g_n(s^n_1, \dots, s^n_{k_n})\} \uplus P; \sigma; \alpha \Longrightarrow$$
$$\Big(P \cup \{x \simeq^?_{\mathcal{R},\lambda} g_1(s^1_1, \dots, s^1_{k_1}), \dots, x \simeq g_n(s^n_1, \dots, s^n_{k_n})\}\Big)\vartheta; \sigma\vartheta; \alpha,$$

where $n \geq 1$, $k_i \geq 0$ for all $1 \leq i \leq n$, P does not contain an equation of the form $x \simeq^?_{\mathcal{R},\lambda} g(\tilde{s})$, $\vartheta = \{x \mapsto f(v_1, \dots, v_m)\}$ where $v_1, \dots, v_m$ are fresh variables and $f \sim^{\pi_1}_{\mathcal{R},\lambda} g_1, \dots, f \sim^{\pi_n}_{\mathcal{R},\lambda} g_n$, and for each $1 \leq l \leq n$, if $i \mapsto j \in \pi_l$, then $x \notin var(s^l_j)$.

4 Concluding remarks

The problem considered in this paper generalizes proximity-based unification [7] and matching [6], permitting arity mismatch between symbols (in addition to mismatches between symbol names of the same arities). It also generalizes similarity-based unification where such a mismatch has been studied, see [2], since similarity is a special case of proximity.

In [7], we proposed an algorithm to solve proximity equations. In our opinion, that was an elegant two-staged approach. In the first stage, unification rules are applied which either leads to failure or returns neighborhood (proximity class) constraints, which are solved in the second stage. For the problem studied in this paper, such a separation of the algorithm into two stages is quite challenging, since we need a concrete element of a proximity class to perform decomposition. This is an interesting issue we would like to look into in more detail.

Acknowledgments

This work is supported by Austrian Science Fund (FWF) under project 28789-N32. The authors are grateful to the anonymous referees for useful remarks.

References

[1] H. Aït-Kaci and G. Pasi. Fuzzy unification and generalization of first-order terms over similar signatures. In F. Fioravanti and J. P. Gallagher, editors, *Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers*, volume 10855 of *Lecture Notes in Computer Science*, pages 218–234. Springer, 2017.

[2] H. Aït-Kaci and Y. Sasaki. An axiomatic approach to feature term generalization. In L. D. Raedt and P. A. Flach, editors, *ECML*, volume 2167 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2001.

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

[4] P. Julián-Iranzo and F. Sáenz-Pérez. An efficient proximity-based unification algorithm. In *2018 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2018*, pages 1–8. IEEE, 2018.

[5] P. Julián-Iranzo and F. Sáenz-Pérez. Proximity-based unification: an efficient implementation method. *IEEE Transactions on Fuzzy Systems*, 2020.

[6] T. Kutsia and C. Pau. Matching and generalization modulo proximity and tolerance. RISC Report Series 19-07, Research Institute for Symbolic Computation, Johannes Kepler University Linz, 2019.

[7] T. Kutsia and C. Pau. Solving proximity constraints. In M. Gabbrielli, editor, *Logic-Based Program Synthesis and Transformation - 29th International Symposium, LOPSTR 2019, Porto, Portugal, October 8-10, 2019, Revised Selected Papers*, volume 12042 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2019.

[8] M. I. Sessa. Approximate reasoning by similarity-based SLD resolution. *Theor. Comput. Sci.*, 275(1-2):389–426, 2002.

Algorithmic Problems in Synthesized Cryptosystems (Extended Abstract)

Andrew M. Marshall[1], Catherine A. Meadows[2], Paliath Narendran[3], Veena Ravishankar[1], and Brandon Rozek[1]

[1] University of Mary Washington, Fredericksburg, VA, USA
[2] Naval Research Laboratory, Washington, DC, USA
[3] University at Albany–SUNY, Albany, NY, USA

1 Introduction

A relatively new and promising direction in generating secure cryptosystems is to synthesize and verify them automatically. Generally, the cryptosystems are first generated, and then symbolic techniques that have been proven sound and/or complete with respect to cryptographic security are applied, in order to identify secure cryptosystems and/or weed out insecure ones. (See for example [3, 7]). In this work we apply a technique we are developing for the synthesis of *cryptographic modes of operation*, that is, algorithms that use block ciphers, cryptosystems that, given a plaintext block of fixed length η, return an η-length block of ciphertext. In this approach we analyze the synthesized cryptographic algorithms via a type of protocol modeling the interaction between an adversary and an encryptor/oracle. We denote these protocols as Cryptographic Modes of Operation programs or MOO-programs for short. In this model the adversary sends messages to the oracle which then encrypts those messages according to some pre-determined method. In cases that there are choices as to action the oracle can take, that choice is made by the adversary. The encrypted blocks are then sent back to the adversary based on some schedule. As shown in [6, 9], both the encryption method and the schedule are relevant to the security of the cryptosystem. One can then reduce certain security questions about the cryptographic algorithm to the question as to whether the adversary can identify a set of ciphertext blocks such that it can force the oracle to return instantiations of the blocks whose $\oplus$ sum to 0, under the assumption that the adversary is not able to compute the block cipher function itself. We are therefore interested in the class of algorithmic questions that ask whether there exists an algorithm to decide if, given a MOO-program with a particular schedule, can the adversary force the cryptosystem to produce cipher blocks that are equal modulo some theory. The most basic form of this question, where the numeric length and number of independent protocol interactions is finitely bounded, and the ciphertext terms are rooted in the encryption symbol, has been proven decidable for the *xor*-equational theory [8]. Here we investigate some of the remaining cases for that theory.

A related algorithmic question asks if given a MOO-term and assuming that the block encryption function can be inverted (decrypted), can one unwind the encryption and other function applications to retrieve the original message. This "invertibility" property of course is a requirement for any useful encryption algorithm. We investigate the following algorithmic question of given a MOO-program: is there an algorithm that will decide if the cipher blocks are invertible?

2 Preliminaries

We assume the reader is familiar with equational unification and term rewriting systems [2].

In this paper we will primarily be concerned with the equational theory of *xor*, $E_{xor} = R_{\oplus} \cup E_{\oplus}$, where $R_{\oplus} = \{x \oplus x \to 0,\ x \oplus 0 \to x\}$ and $E_{\oplus} = AC(\oplus)$ over the signature, $\Sigma_{\oplus} = \{\oplus, f, 0\}$.

MOO-Program: This describes an interaction between the adversary and the oracle in which the adversary sends blocks of plaintext to be encrypted and the oracle sends back blocks of ciphertext according to some fixed *schedule* defined by the mode of operation. Messages with multiple blocks are encrypted block by block. In a block-wise schedule an encrypted block is sent to the adversary immediately after it is generated by the oracle. In a message-wise schedule, all the encrypted blocks are sent to the adversary after the entire message is encrypted. The blocks sent between the adversary and the oracle are modeled by terms. These $MOO_{\oplus}$-terms can be 0, variables representing plain-text blocks, bounded variables representing a random string, or any term built up using the signature $\Sigma = \{\oplus, 0, f\}$. The terms of the frame are further restricted depending on the method of encryption being modeled. The method of encryption dictates how the oracle constructs cipher blocks. For example, in Cipher Block Chaining, CBC, the i^{th} cipher block, C_i, is modeled by $f(C_{i-1} \oplus x_i)$, where x_i is the i^{th} plaintext sent by the adversary.

A *MOO*-Program will be modeled by a list of $MOO_{\oplus}$-terms of the form $[t_1,\ t_2, \ldots,\ t_n]$. All $MOO_{\oplus}$-terms are listed in the order that they are sent. For example, the following *MOO*-Program models the CBC mode of encryption with three cipher blocks using the block-wise schedule. $[IV, x_1, f(IV \oplus x_1), x_2, f(x_2 \oplus f(IV \oplus x_1))]$. Here IV is a bound variable representing an initial nonce. Each x_i models a plain-text block sent by the adversary and each f-rooted term is a cipher block returned by the oracle.

Each *MOO*-Program models a single *session* between the adversary and oracle, where a session is a program that encrypts a single message consisting of a sequence of plaintext blocks. The adversary has the ability to execute multiple simultaneous sessions with the oracle. In this case the initial nonces, the IV, will be fresh for each session. Each session can then be modeled by its own list of terms or *MOO*-Program.

We are interested in the problem of deciding if the adversary is able to find two or more ground cipher blocks $C_1, \ldots, C_k$ whose $\oplus$ sum is zero. We can define several instances of the problem based on the combination of the following factors: The equational theory E, the method of encryption, the schedule, and bounds on the session length and number of sessions.

3 Decision Problems on Detecting Possible Collisions

Here we are interested in whether the problem of identifying sets of ciphertext blocks summing to zero will turn undecidable if we release the boundedness condition on either the number of sessions or their lengths.

3.1 Non-deterministic Decision Problem in Unbounded Sessions

In this section we prove that the non-deterministic version of the decision problem for MOO-programs of unbounded session lengths and bounded number of sessions is undecidable. Here, non-deterministic means that more than one encryption method is available to the oracle, and the choice of which one is used is made by the adversary.

Definition 1. *Let α be a string $a_0 a_1 \ldots a_m$ and C be a block. Then, $F(\alpha \bigoplus C) = f(a_0 \oplus f(a_1 \oplus \ldots f(a_m \oplus C) \ldots))$.*

The cipher block construction below encodes possible solutions to the Post Correspondence Problem (PCP).

Definition 2. *Let Γ be an alphabet s.t., $\Gamma = \{a, b\}$. The PCP consists of a finite list of blocks,*

$$\left(\frac{\alpha_1}{\beta_1}\right), \left(\frac{\alpha_2}{\beta_2}\right), \ldots, \left(\frac{\alpha_n}{\beta_n}\right)$$

Each α_i and β_i is a word over Γ. A solution to the PCP is a sequence of indices $i_j, 1 \leq j \leq M$, where $M > 1$ and $1 \leq i_j \leq n$ such that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_M} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_M}$

The PCP is a classical undecidable problem. We use to define how cipher block terms are constructed by the oracle.

Definition 3. *Let $PCP = (\frac{\alpha_0}{\beta_0}), (\frac{\alpha_1}{\beta_1}), \ldots, (\frac{\alpha_n}{\beta_n})$, and C_i the i^{th} cipher block output.*
For $i > 0$ let $C_i = E_{i_0}$ or, E_{i_2}, or $\ldots$, or E_{i_n} where $E_{i_j} = [f(r_i \oplus C_{i,1}), f(r_i \oplus C_{i,2})]$, $0 \leq j \leq n$, $C_{i,1} = F(\alpha_j \bigoplus C_{i-1,1})$, $C_{i,2} = F(\beta_j \bigoplus C_{i-1,2})$ $C_{0,1} = F(\alpha_j \bigoplus 0)$, $C_{0,2} = F(\beta_j \bigoplus 0)$.

Based on the non-deterministic system of Definition 3 we can define the following *MOO*-program which produces two equal cipher blocks iff the adversary finds a solution to the PCP.

Definition 4. *Denote the following MOO-program as PCP_{NDMOO_1}. The program works as follows: The adversary non-deterministically picks a possible solution to the PCP, $i_0, i_1, i_2, \ldots, i_k$. Each turn the adversary sends an index in the solution, starting with i_k and proceeding each turn until i_0 is reached. At each step the oracle encodes a pair of cipher blocks E_j, according to Definition 3 and returns them to the adversary. After receiving each E_j, the adversary attempts to check if any two cipher blocks are equal. The program stops if the adversary finds two equal pairs or it sends a stop session command.*

Lemma 1. *Given a PCP problem the corresponding PCP_{NDMOO_1} program produces a set of cipher blocks whose $\oplus$ sum is zero iff there is a solution to the given PCP.*

Proof Sketch Assume there is a set of cipher blocks whose $\oplus$ sum is zero. Since all cipher blocks are rooted in the free function symbol f, this can only be the case if the program returns two equal cipher blocks. In addition, due to the random r_i only blocks from the same step, $C_{i,1}$ and $C_{i,2}$ can sum to zero. The encoding of the PCP solution follows from Definition 3. Assume there is a solution to the PCP. Let $i_1, i_2, \ldots, i_m$ be the solution. Notice that during the m^{th} step the blocks $C_{m,1}$ and $C_{m,2}$ will fully encode this solution. Thus both the encoded strings and the unique random nonces will be equal, leading to a zero sum.

Example 1. *Consider the following PCP: $\overbrace{\left(\frac{ba}{baa}\right)}^{\textit{block 1}}, \overbrace{\left(\frac{ab}{ba}\right)}^{\textit{block 2}}, \overbrace{\left(\frac{aaa}{aa}\right)}^{\textit{block 3}}$. A solution to this problem is $1, 3$. Let's trace a run of the PCP_{NDMOO_1} program where the adversary guesses the solution $1, 3$. In the first step the adversary sends the number 3 to the oracle and receives the following cipher block in return. $C_0 = E_{0_3}$ where $E_{0_3} = [f(r_0 \oplus C_{0,1}), f(r_0 \oplus C_{0,2})]$, $C_{0,1} = F(\alpha_3 \bigoplus 0) = (f(a \oplus f(a \oplus f(a \oplus 0))))$, $C_{0,2} = F(\beta_3 \bigoplus 0) = f(a \oplus f(a \oplus 0))$ Notice that now after step 2 the adversary has two cipher blocks, $C_{1,1}$ and $C_{1,2}$, which are equal. $C_{1,1} = f(b \oplus f(a \oplus f(a \oplus f(a \oplus f(a \oplus 0)))))$, $C_{1,2} = f(b \oplus f(a \oplus f(a \oplus f(a \oplus f(a \oplus 0)))))$*

Theorem 1. *Assume M is an arbitrary non-deterministic MOO-program. The problem of determining if M, executing with a bounded number of sessions and unbounded session lengths, ever produces two equal cipher blocks is undecidable.*

3.2 Additional Undecidability Results

Due to space several additional undecidability results are not included but can be proven using a similar reduction. These cases include deterministic unbounded session length, both deterministic and non-deterministic unbounded number of sessions with bounded session length.

4 The Invertibility Problem

A natural requirement of any cryptographic algorithm is that it be *invertible*; that is, one can find the original plaintext using the ciphertext and decryption key. In the case of modes of encryption that leads to two different questions. The first is, given a set S of MOO-terms with subterms designated as plain text, can we tell if S is invertible? The second is: given a MOO-program, can we tell if the projection of any MOO frame on to the oracle is invertible? The second question is our ultimate goal, but to answer it we need to answer the first. We present an answer to the first, and we are currently working on answering the second.

Let $\mathcal{C} = \{C_0, C_1, \ldots, C_n\}$ represent the cipher blocks, C_i, produced by the oracle in the MOO-program. We instantiate the variables representing plaintext in $\mathcal{C}$ to constants p_i. Let $P = \{p_0, p_1, \ldots, p_n\}$ be the set representing the plaintext messages during a run of the MOO-program. We first define an *invertibility* relation, $\phi \vdash_E p$, s.t. $p \in P$, axiomatized by a set of inference rules introducing a new symbol, f^{-1}. f is the symbolic encryption function, i.e., $f = enc(\_, K)$, for some key K, and let the model decryption function $f^{-1} = dec(\_, K)$, s.t. $f^{-1}(f(p)) = p$. In this case the set of rules axiomatizing the invertibility problem are exactly those axiomatizing the deduction problem [1, 4]. The deduction problem is undecidable in general, but is decidable for some theories (see for example [1,4]). Thus, it allows us to obtain a general decidability result for the theory of interest.

4.1 General Invertibility

Here we limit our investigation to signature $\Sigma = \{\oplus, 0, f, f^{-1}\}$ and MOO-programs over this signature. The equational theories can be presented as a combination. $R_\oplus = \{x \oplus x \to 0,\ x \oplus 0 \to x\}$, $R_f = \{f(f^{-1}(x)) \to x,\ f^{-1}(f(x)) \to x\}$, $E_\oplus = AC(\oplus)$. We are interested in the invertibility problem for the theory $E^{-1} = R_f \cup R_\oplus \cup E_\oplus$. Note that f^{-1} is not always necessary to obtain invertibility but is if the plain-text appears below an f.

First consider the deduction problem for $R_\oplus \cup E_\oplus$, shown decidable in [1]. Furthermore, the theory R_f is subterm convergent, each rule's right-hand side is a subterm of the left-hand side, and thus the deduction problem is also decidable due to [5]. Therefore we have the following.

Lemma 2. *([1, 5]) The deduction problem is decidable in both $R_\oplus \cup E_\oplus$ and R_f.*

We now consider the combination problem for the disjoint combination theory $R_f \cup (R_\oplus \cup E_\oplus)$. From the disjoint combination result of [4] we get the following.

Lemma 3. *The deduction problem is decidable in the theory $R_f \cup (R_\oplus \cup E_\oplus)$.*

Corollary 1. *The invertibility problem for the theory $R_f \cup (R_\oplus \cup E_\oplus)$ is decidable.*

While we know that an algorithm exists for solving the invertibility problem for the theory $R_f \cup (R_\oplus \cup E_\oplus)$, a more efficient algorithm is possible, taking advantages of the properties of MOO-programs. The first step is to compute a set representing the knowledge of the adversary.

Definition 5. *Let $\mathcal{C} = \{C_0, C_1, \ldots, C_m\}$ be the set of cipher-blocks from a MOO-program. Let N denote any initial nonces, random strings, known by the adversary, and let $K = range(\mathcal{C}) \cup N$ Let $S = \max\{|t|, s.t.\ t \in \mathcal{C}\}$. The set of* saturated *knowledge, K^*, is computed as follows:*

1. *Initially $K^* = K$.*
2. *Three closure operations are applied until there is no change to the set K^*:*
   (a) *If $t \in K^*$, $t(\epsilon) = f$, $f^{-1}(t) \to t'$, then $K^* = K^* \cup \{t'\}$.*
   (b) *If $t_1, t_2 \in K^*$, $t_1 \oplus t_2 = t'$, and $|t'| \leq S$ then $K^* = K^* \cup \{t'\}$.*
   (c) *If $t \in K^*$, and $|f(t)| \leq S$ then $K^* = K^* \cup \{f(t)\}$.*

We now present the invertibility algorithm in Algorithm 1.

Algorithm 1 Invertibility for $MOO_\oplus$ terms

```
Require: Set $K$, a plain-text goal $p$, and a set $K_p \subseteq K$ of terms containing $p$ as a subterm.
  if $K_p = \emptyset$ then
    Exit with Failure.
  else
    Compute $K^*$
  end if
  if $p \in K^* \vee \exists t \in K^*\ s.t.\ t =_{R_f \cup R_\oplus \cup E_\oplus} p$ then
    Return success.
  else
    Exit with Failure
  end if
```

Lemma 4. *Consider a MOO-program, M, over the signature $\Sigma = \{f, \oplus, 0\}$. Let K^* be the set of saturated knowledge constructed from the cipher blocks of M and nonces. Then $\phi \vdash_{E^{-1}} t$ iff $\exists t' \in K^*\ s.t.\ t' =_{E^{-1}} t$.*

Theorem 2. *Algorithm 1 is terminating, sound, and complete for the theory $R_f \cup (R_\oplus \cup E_\oplus)$,*

Proof Sketch Notice that K^* is finite since for any $t \in K^*$, $|t| \leq |t'|$, where t' is the largest term in $\mathcal{C}$. Since checking equality module $R_f \cup (R_\oplus \cup E_\oplus)$ is decidable, termination follows. The remainder of the proof follows from Lemma 4.

Example 2. *Let $C_0 = p_0 \oplus f(IV)$, $C_1 = p_1 \oplus f(p_0) \oplus f(IV)$, $C_3 = p_2 \oplus f(p_1) \oplus f(p_2)$. If IV is known, $IV \in N$, then $f(IV) \in K^*$ and from C_0 we obtain, p_0. Once we have p_0, $f(p_0) \in K^*$ and from C_1 we get p_1. Likewise, we can also obtain p_2. Thus we have invertibility.*

However, if we don't know the IV, $IV \notin N$, then we can't know $f(IV)$ and thus not p_0.

5 Conclusions

We report on two algorithmic questions important to the synthesis of cryptographic modes of operation. Future work includes identifying decidable cases of the decision problem, considering the second form of the invertibility problem, and implementing the decidable case into a new tool for the automatic synthesis and security verification of certain cryptosystems.

References

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

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

[3] Brent Carmer and Mike Rosulek. Linicrypt: A model for practical cryptography. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*, pages 416–445, 2016.

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

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

[6] Antoine Joux, Gwenaëlle Martinet, and Frédéric Valette. Blockwise-adaptive attackers: Revisiting the (in)security of some provably secure encryption models: Cbc, gem, IACBC. In *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, pages 17–30, 2002.

[7] Alex J Malozemoff, Jonathan Katz, and Matthew D Green. Automated analysis and synthesis of block-cipher modes of operation. In *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*, pages 140–152. IEEE, 2014.

[8] Catherine Meadows. Symbolic security criteria for blockwise adaptive secure modes of encryption. Cryptology ePrint Archive, Report 2017/1152, 2017. https://eprint.iacr.org/2017/1152.

[9] Phillip Rogaway. Nonce-based symmetric encryption. In *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, pages 348–359, 2004.

An Improved Algorithm for Testing Whether a Special String Rewriting System is Confluent

Paliath Narendran, Saumya Arora, and Yu Zhang

University at Albany–SUNY (USA),
pnarendran@albany.edu
{saumya.arora22295, yuzh1987}@gmail.com

Abstract

String rewriting systems is an important research area with many applications. One important desired property of string rewriting systems is *confluence*, which ensures that any two equivalent strings have the same "normal form".

In this paper, we propose an algorithm to check confluence of *special* string rewriting systems, i.e., those where the right-hand side of every rule is the empty string. Our goal is to improve the worst-case complexity of the algorithm of Kapur, Krishnamoorthy, McNaughton and Narendran [6]. The key improvement that we suggest is the use of *generalized suffix trees* to check for overlaps between the left-hand sides of the string rewriting system.

Keywords: special string rewriting system, confluence, generalized suffix trees.

1 Introduction

String rewriting systems have been studied very extensively in the last four decades. These systems are basically collections of rewrite rules of the form $l \to r$ where l and r are strings over an alphabet of symbols Σ. One of the important properties that one usually desires for a string rewriting system is *confluence* which means that any two equivalent strings can be rewritten in a finite number of steps to a common string. If the rewriting process can also be shown to always terminate, then equivalence of strings with respect to the system (the *word problem*) is decidable.

A suffix tree is a data structure that stores all the suffixes of a given string in a tree format. Suffix trees have proven to be useful in reducing the time complexity of many important string algorithms.

A generalized suffix tree (GST) [10, 11] is a variation on suffix tree. It is a suffix tree of a *set* of strings and is a combination of suffix trees of all the strings in the set. This is done by concatenating these strings by ending them each with a different end marker (as shown in Figure 2). We are using generalized suffix trees in our algorithm in order to improve the speed of implementation of string operations. Figure 1 is an example of a suffix tree for the string *abbaab* which illustrates how the above-mentioned string arrangement can be represented by a suffix tree:

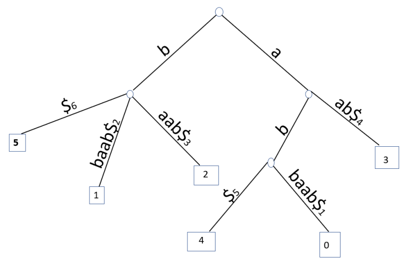

Figure 1: Suffix tree for string *abbaab*

In Figure 2 below, we represent a set of two strings $\{abac, caba\}$ as a generalized suffix tree:

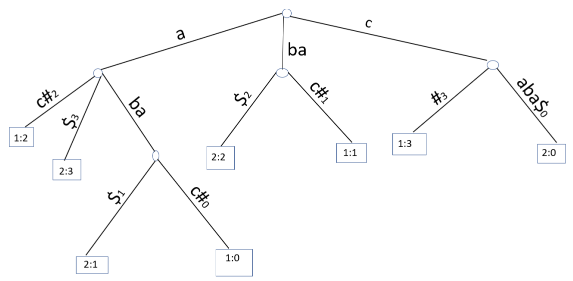

Figure 2: Generalized suffix tree for a set containing two strings $\{abac, caba\}$

We focus here on the problem of confluence for *special* string rewriting systems where every rule is of the form $w \to \epsilon$ where w is a nonempty string over some alphabet. Our main goal is to suggest an improvement of the algorithm of Kapur, Krishnamoorthy, McNaughton and Narendran [6], which has a worst case complexity of $O(k|T|)$ where k is the number of rules in the system T and $|T|$ is the *size* of T, i.e., the sum of the lengths of all its left-hand sides. We formulate the algorithm a little differently and propose the use of *generalized suffix trees* to check for overlaps between left-hand sides.

We want to point out that here we only discuss checking confluence of a *residual* string rewriting system, where a system is residual if and only if no left-hand side is a substring of another. It is shown in [6] that this is the crux of the problem, since extracting a residual system from a given system and checking the extra rules for joinability can be done in linear time.

2 Definitions

The reader is referred to the books [1] and [4] for term rewriting and string rewriting respectively. A string α overlaps with a string β if and only if a *nonempty* prefix of α is a suffix of β: in other words, there are strings u, v, w such that $\alpha = uv$ and $\beta = wu$ where $u \neq \epsilon$. Two strings x and y are *conjugate* (or *cyclically equal*) if and only if there exist two strings u and v such that $x = uv$ and $y = vu$. For instance, the strings *abaab* and *ababa* are conjugates. Note that conjugacy is an equivalence relation. A nonempty string is *primitive* if and only if it is not the power of any other string. For instance, the string *abaab* is primitive, while the string ϵ and $bababa = (ba)^3$ are not. If x is a nonempty string, and z is a primitive string such that $x = z^n$, then we call z the *root* of x, and n the exponent of x [5].

Lemma 2.1. *Let* $\{\beta_1, \ldots, \beta_n\}$ *be a set of* primitive *strings that are conjugate to one another. Then*

$$T \;=\; \{\beta_1 \to \epsilon, \ldots, \beta_n \to \epsilon\}$$

is confluent if and only if

1. *no* β_i *overlaps with itself,* and
2. *no two strings* β_i *and* β_j $(j \neq i)$ *overlap in more than one way.*

Proof. Suppose T is not confluent. Then there must be nonempty strings u, v, w such that $\beta_i \;=\; uv$ and $\beta_j \;=\; vw$ for some i, j, with $u \neq w$. Two cases have to be considered.

1. $i = j$: Straightforward, since $v \neq \varepsilon$.
2. $i \neq j$: Since β_i and β_j are conjugates, there must be strings x, y such that $\beta_i \;=\; xy \;=\; uv$ and $\beta_j \;=\; yx \;=\; vw$. If $y = v$ then $x = u = w$. Thus either y is a proper prefix and a proper suffix of v, or vice versa. Thus β_i and β_j overlap in more than one way.

For the "only if" part, we need to consider the cases:

1. Some β_i overlaps with itself, i.e., $\beta_i \;=\; uvu$ for some nonempty strings u, v. But uv cannot be the same as vu since this would mean that β_i is not primitive. Thus T is not confluent ($uv \;\longleftrightarrow\; uvuvu \;\longleftrightarrow\; vu$).
2. Some β_i and β_j $(j \neq i)$ overlap in more than one way. Let x be the smallest prefix of β_i that is a suffix of β_j. Since there is another such overlap, it must be that

$$\beta_i \;=\; xyxw \;\text{ and }\; \beta_j \;=\; uxyx$$

If $u \neq w$ then T is not confluent. If $u = w$, then $yxu \;=\; uxy$. This equation has the following general solution:

$$u \;=\; (v_1v_2)^p v_1, \; x \;=\; (v_2v_1)^q v_2, \;\text{ and }\; y \;=\; (v_1v_2)^r v_1$$

for some v_1, v_2 and $p, q, r \geq 0$. But then

$$\begin{aligned} uxyx \;&=\; (v_1v_2)^p v_1 (v_2v_1)^q v_2 (v_1v_2)^r v_1 (v_2v_1)^q v_2 \\ &=\; (v_1v_2)^{p+2q+r+2} \end{aligned}$$

which is not a primitive string. □

Lemma 2.2. *Let* $\{\alpha_1, \ldots, \alpha_n\}$ *be a set of conjugate strings and* $\{\gamma_1, \ldots, \gamma_n\}$ *be their primitive roots. Then*

$T = \{\alpha_1 \to \epsilon, \ldots, \alpha_n \to \epsilon\}$ is confluent if and only if $T' = \{\gamma_1 \to \epsilon, \ldots, \gamma_n \to \epsilon\}$ is confluent.

Proof. Since primitive roots of conjugate strings are conjugates themselves, the strings $\gamma_1, \ldots, \gamma_n$ are conjugates to one another. Let $k > 0$ be such that $\alpha_i = \gamma_i^k$ for all i.

Suppose T' is not confluent. Then either

1. Some γ_i overlaps with itself, i.e., $\gamma_i = uvu$ for some nonempty strings u, v where $uv \neq vu$. Since $\alpha_i = \gamma_i^k$, $(uvu)^{k-1}uvuvu(uvu)^{k-1} \to_T (uvu)^{k-1}uv$ and $(uvu)^{k-1}uvuvu(uvu)^{k-1} \to_T vu(uvu)^{k-1}$. But $(uvu)^{k-1}uv$ and $vu(uvu)^{k-1}$ cannot be equal since $uv \neq vu$.
2. There are nonempty strings u, v, w such that $\gamma_i = uv$ and $\gamma_j = vw$ for some $i \neq j$, with $u \neq w$. Then $(uv)^{k-1}uvw(vw)^{k-1} \to_T (uv)^{k-1}u$ and $(uv)^{k-1}uvw(vw)^{k-1} \to_T w(vw)^{k-1} = (wv)^{k-1}w$. Again, $(uv)^{k-1}u$ and $(wv)^{k-1}w$ cannot be equal unless $u = w$.

In the other direction, suppose T' is confluent and T is not. We can now use Lemma 5.7 in [6]: if T is not confluent, there must be rules $(xy)^k \to \epsilon$, $(yx)^k \to \epsilon$ in T, and strings u, v, w such that

$$(xy)^k = uv, \ (yx)^k = wu, \ \text{ and } v \neq w.$$

By Lemma 5.7 in [6] we get that x or y must have a self-overlap, or, in other words, xy and yx overlap in more than one way. This contradicts the assumption that T' is confluent. (The case where the rules are the same is the case $y = \epsilon$.) □

3 An Improved Algorithm for Testing Whether a Special String Rewriting System in Confluent

The key new idea behind the algorithm is the use of generalized suffix trees (GST). This is also the major difference between our algorithm and that of [6]. The advantage is that this makes overlaps between strings easier to locate. For instance, if we build a GST for the set of strings S, then, for any string $w \in S$, the strings it overlaps with (including itself) can be found in $|w|$ steps. This avoids the pairwise comparison of strings and hence the nested loops in the algorithm of [6] (page 131).

The outline of our proposed algorithm is as follows:

Let $T = \{\alpha_1 \to \epsilon, \ldots, \alpha_n \to \epsilon\}$ be a special string rewriting system[1]. Let $\mathcal{L}$ denote the set of left-hand sides, i.e., $\{\alpha_1, \ldots, \alpha_n\}$. We can build a GST for $\mathcal{L}$ in time $O(|T|)$.

1. Identify all the left-hand sides that have self-overlaps and check whether their smallest self-overlap is also a primitive root. If not, then terminate with failure. If yes, save the primitive root.

   Example: Consider the rule $aba \to \epsilon$. The smallest self-overlap of aba is a, but a is not a primitive root of aba. Hence the system is not confluent ($ba \longleftrightarrow ababa \longleftrightarrow ab$).

[1]As mentioned earlier, we assume that T is residual.

This step can be done in $O(|T|)$ time by using GST.

2. For every string α_i identify strings that it overlaps within $\mathcal{L}$. If α_i has self-overlap and one of the strings it overlaps with has not, then terminate with failure. Similarly if α_i has no self-overlap and one of the strings it overlaps with has, then terminate with failure.

   Example: Let $\alpha_1 = abab$ and $\alpha_2 = bcba$. α_1 has a self-overlap (ab) whereas α_2 does not. $bcb \longleftrightarrow bcbabab \longleftrightarrow bab$.

   Similarly, if α_i and one of the strings it overlaps with have different lengths, then terminate with failure.

   If α_i and a string α_j that it overlaps with are cyclically equal, then put them in the same conjugate class. To check whether they are cyclically equal, since we already know the overlap part, we only need to check whether the rest of the strings are the same. From the Lemma 5.1 in [6], we know that if α_i and α_j are not cyclically equal, then we can terminate with failure. Therefore if $\alpha_{i_1}, \ldots, \alpha_{i_k}$ are the strings that α_i overlaps with, then

   For $j = 1, \ldots, k$ check whether α_{i_j} is cyclically equal to α_i.
   Mark α_i and all of $\alpha_{i_1}, \ldots, \alpha_{i_k}$.

   Note again that since conjugacy is an equivalence relation, we don't need to consider all pairs of strings. We only need to check that the initially considered string, α_i, is conjugate to all the other strings.

   Now process the other strings in the conjugacy class: if any of them has an overlap with an unmarked string or a string in another conjugacy class, then terminate with failure.

   Example: Consider the strings *abc*, *bca* and *ccb*. *abc* overlaps with *bca*, and they are cyclically equal. *abc* does not overlap with *ccb*, but *bca* overlaps with *ccb*. So if *abc* is processed first, then only *bca* will be marked as conjugate. Now when *bca* is processed the additional overlaps will be discovered and the algorithm will terminate with failure.

3. Separate strings in $\mathcal{L}$ into equivalence classes of conjugates.

4. For each equivalence class, we already know their primitive roots from Step 1. We want to check whether any of the primitive roots overlaps with another primitive root in more than one way. This can be done by using a GST for the primitive roots and in $O(|T|)$.

   Example: Consider the rules $abac \to \epsilon$ and $caba \to \epsilon$. *abac* and *caba* are cyclically equal. But *abac* overlaps with *caba* in more than one way. Non-confluence is not hard to see, since $bac \longleftrightarrow cababac \longleftrightarrow cab$ and *bac* and *cab* are irreducible.

4 Conclusion and Future Work

We have developed an algorithm of complexity $\big(O(|T|)\big)$ to improve the worst-case complexity $\big(O(k|T|)\big)$ of the algorithm of Kapur, Krishnamoorthy, McNaughton and Narendran [6]. Implementing this algorithm may not be arduous because generalized suffix trees have already been implemented in many programming languages. It will also be interesting to see whether generalized suffix trees can be used for other kinds of systems as well.

For future work, we plan to look into computational problems for special confluent systems. The word matching problem was shown to be undecidable in [7]. Unification — as defined and shown in [9] — is decidable. We plan to look into the matching and unification problems modulo special confluent systems, starting with the matching problem. We hope the structural properties we proved and used for special confluent systems will be helpful in devising efficient algorithms.

References

[1] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1999.

[2] Ronald V. Book. A note on special Thue systems with a single defining relation. *Mathematical Systems Theory* 16(1):57–60, 1983.

[3] Ronald V. Book and Colm O'Dunlaing. Testing for the Church-Rosser property. *Theoretical Computer Science* 16:223–229, 1981.

[4] Ronald V. Book and Friedrich Otto. *String-rewriting systems.* Springer, 1993.

[5] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings.* Cambridge University Press, 2007

[6] Deepak Kapur, Mukkai S. Krishnamoorthy, Robert McNaughton, and Paliath Narendran. The Church-Rosser property and special Thue systems. *Theoretical Computer Science* 39:123–133, 1985.

[7] Paliath Narendran and Friedrich Otto. The word matching problem is undecidable in general even for finite special string-rewriting systems that are confluent. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP'97),* Bologna, Italy, July 1997, Lecture Notes in Computer Science 1256, pages 638–648, Springer.

[8] Maurice Nivat. On some families of languages related to the Dyck language. In *Proceedings of the Second Annual ACM Symposium on Theory of Computing*, STOC '70, pages 221–225, New York, NY, USA, 1970. ACM.

[9] Friedrich Otto, Paliath Narendran, and Daniel J. Dougherty. Equational unification, word unification, and 2nd-order equational unification. *Theoretical Computer Science*, 198(1-2):1–47, 1998.

[10] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica* 14(3):249–260, 1995.

[11] Bálint Márk Vásárhelyi. Suffix trees and their applications. PhD dissertation, *Faculty of Science*, Budapest, 2013.

Some Results on Prefix Grammars

Paliath Narendran, Ashley Suchy, and Yu Zhang

University at Albany–SUNY (USA),
e-mail: {pnarendran,asuchy,yzhang20}@albany.edu

Abstract

In this paper, we show some results about prefix grammars, also known as prefix rewriting systems. In particular, we investigate the complexity of some problems regarding prefix grammars.

This work was inspired by the paper of Michael Frazier and C. David Page Jr.. We show that their algorithm for obtaining a right-linear grammar equivalent to a prefix grammar can be made to run in polynomial time. Additionally, we show that we can check, in polynomial time, whether a prefix string rewriting system generates every string (the universality problem), and whether a prefix grammar is deterministic or ambiguous.

We also prove that, given a deterministic finite-state automaton (DFA) and a prefix grammar, whether the languages that they represent are equal is a PSPACE-hard problem.

Keywords: prefix grammar, right-linear grammar, ambiguity, determinism, universality.

1 Introduction

In this paper we explore prefix grammars and the equivalent right-linear grammars that can be obtained from the Frazier-Page algorithm [1]. We study the complexity of the Frazier-Page algorithm and show that it can be made to run in polynomial time. We also show that checking the properties of universality, determinism and ambiguity can be done in polynomial time. We show that the restricted equivalence problem of whether the language of a given DFA is equal to the language of a given prefix grammar is PSPACE-hard.

2 Definitions

Let L be a language over an alphabet Σ and $w \in \Sigma^*$. The left quotient of L by w is defined as $w\backslash L = \{x \mid wx \in L\}$. Note also that $\epsilon\backslash L = L$ for all L.

A prefix grammar G_P is the triple $\big(\Sigma, S = \{\alpha_i \mid 1 \leq i \leq m\}, P = \{\beta_j \to \gamma_j \mid 1 \leq j \leq n\}\big)$, where Σ is a finite set of symbols, $S \subset \Sigma^*$ is a finite set of base strings, and P is a finite set of productions where β and γ are strings over Σ. (We use this notation throughout the paper.) The productions may only be applied to rewrite the prefix of a string. Let $\mathcal{L}(G_P)$ be the language of the prefix grammar G_P.

A right-linear grammar G_R is the tuple $G_R = (\Sigma, S, N, R)$, where Σ is a finite set of terminal symbols, S is the start symbol, N is a finite set of non-terminal symbols, and R is a finite set of productions of the form $A \to xB$ or $A \to x$ where $A, B \in N$ and $x \in \Sigma \cup \{\epsilon\}$. Let $\mathcal{L}(G_R)$ denote the language of the right-linear grammar G_R.

3 On the Complexity of the Frazier-Page Algorithm

Let $G_P = \big(\Sigma, \{\alpha_i \mid 1 \leq i \leq n\}, \{\beta_j \to \gamma_j \mid 1 \leq j \leq m\}\big)$ be a prefix grammar. The Frazier-Page algorithm constructs a right-linear grammar G_R with $m+1$ nonterminals: the start symbol S_1 and $\{V_\beta \mid (\beta \to \gamma) \in P\}$. It starts with the initial set of productions

$$S_1 \to \alpha_1 \mid \ \dots \ \mid \alpha_n \mid \gamma_1 V_{\beta_1} \mid \ \dots \ \mid \gamma_m V_{\beta_m}.$$

Let us denote this initial grammar as G_{init}. Note that $|G_{init}| \ \leq \ |G_P|$.

The algorithm then adds new rules as follows until no more rules can be generated: at any stage if there is a derivation sequence from S_1 to a string $\beta_i w V_{\beta_j}$ by the grammar at that point where the last rule applied is of the form $V_{\beta_k} \to uwV_{\beta_j}$ with $u \neq \varepsilon$, then add $V_{\beta_i} \to wV_{\beta_j}$.

All the new rules that are added to this will be of the form $V_{\beta_i} \ \to \ wV_{\beta_j}$, where the right-hand side wV_{β_j} is the suffix of the right-hand side of some rule in G_{init}. Thus there will be *at most* $m * |G_{init}|$ new rules. The size of G_R, the final grammar, will be $O(|G_{init}|^3)$. To obtain a new rule of the form $V_{\beta_i} \ \to \ wV_{\beta_j}$, we have to find a derivation $S \ \Rightarrow^* \ \beta_i w V_{\beta_j}$.

Lemma 3.1. [2] (pages 154-155) *Given a right-linear grammar G and a string w, we can check whether $w \in \mathcal{L}(G)$ in $O(|w| * |G|^2)$ time.*

Thus finding a derivation for $\beta_i w V_{\beta_j}$ can be done in $O(|\beta_i w V_{\beta_j}| * |G_R|^2)$ time. The length of each such $\beta_i w V_{\beta_j}$ has a (weak) upper bound of $|G_P|$. Assuming the worst case where each iteration only produces one new rule, we see that such derivations may have to be done $m{*}|G_{init}|$ times in each iteration. The bound on the overall number of iterations is also $m * |G_{init}|$. Thus the overall complexity is $O(|G_P| * |G_R|^2 * m^2 * |G_{init}|^2)$, or $O(|G_P|^{11})$.

4 Computational Problems

We have proved the following theorem in [6].

Theorem 4.1. *Let $G_P \ = \ \big(\Sigma, \{\alpha_i \ \mid \ 1 \ \leq \ i \ \leq \ n\}, \{\beta_j \ \to \ \gamma_j \ \mid \ 1 \ \leq \ j \ \leq \ m\}\big)$ be a prefix grammar and $G_R \ = \ \big(\{V_{\beta_i}\}, \Sigma, R, S_1\big)$ be the right-linear grammar obtained from a G_P using the Frazier-Page algorithm.*

(a) For any i, j, if $\alpha_i \to_P^* \beta_j x$, then $V_{\beta_j} \Rightarrow_R^+ x$.

(b) (Conversely) For any j, if $V_{\beta_j} \Rightarrow_R^+ x$, then $\beta_j x \ \in \ \mathcal{L}(G_P)$

4.1 Universality

Lemma 4.1. *Let G_R be a right-linear grammar obtained from a G_P by using the Frazier-Page algorithm. If $\mathcal{L}(G_P)$ is Σ^*, then* every *non-terminal in G_R can generate Σ^*.*

Proof. Let $L(G_P) = \Sigma^*$ and suppose one of the non-terminals of G_R, say V_β, cannot generate a string w. But $\beta w \in L(G_P)$. So $V_\beta \not\Rightarrow_R^+ w$. By Theorem 4.1 this is a contradiction. □

We now address the following question:

Can we check, in polynomial time, whether a prefix string rewriting system generates Σ^*?

We first consider the following problem:

Let $A = \{x_1, \ldots, x_m\}$ and $B = \{y_1, \ldots, y_n\}$ be two sets of strings over an alphabet Σ such that B does not contain the empty string, i.e., none of the y_i's is empty. We say (A, B) *covers* a language L if and only if $L \subseteq A \cup B\,\Sigma^*$.

The computational problem is this: given a DFA for a regular language L and sets A and B as explained above, can we check whether (A, B) covers L in polynomial time?

The algorithm works as follows:

First construct a DFA $M = (\Sigma, q_0, Q, Q_F, \delta)$ that recognizes the set difference $L \setminus A$. This can be done in polynomial time given a DFA for L. For each $y_i \in B$, run M and mark the state $q_j \in Q$ that the machine ends in. Let Q_{marked} be the set of these states. For each $q_{f_k} \in Q_F$, run depth-first-search to find all paths from q_0 to q_{f_k}. If there is a path that does not contain any state from Q_{marked} then $L \nsubseteq \mathcal{L}(G_P)$. Otherwise $L \subseteq \mathcal{L}(G_P)$.

Theorem 4.2. *We can check, given a prefix grammar G_P, whether $\mathcal{L}(G_P) = \Sigma^*$ in polynomial time.*

Proof. By Theorem 4.1 and the algorithm for converting a G_P to a G_R , we can consider G_R which accepts the same language as G_P. We can also assume, without loss of generality, that G_R does not have any *unit productions,* i.e., rules of the form $V_{\beta_i} \rightarrow V_{\beta_j}$. We can assume that rules with the same left-hand side are grouped together in the usual way, e.g.,

$$V_{\beta_i} \rightarrow x_{i1} \mid \ldots \mid x_{im} \mid y_{i1}V_{\beta_{i_1}} \mid \ldots \mid y_{in}V_{\beta_{i_n}}$$

For each such β_i, let $A_{\beta_i} = \{x_{i1}, \ldots, x_{im}\}$ and $B_{\beta_i} = \{y_{i1}, \ldots, y_{in}\}$. We also define A_ϵ and B_ϵ for the start symbol S of the G_R, which can be considered as V_ϵ in the Frazier-Page construction[1]. We can check whether each $\beta_i \backslash L$ is covered by the corresponding $\big(\{x_{i1}, \ldots, x_{im}\}$, $\{y_{i1}, \ldots, y_{in}\}\big)$.

Now let G_R be the right-linear grammar derived from a G_P using the Frazier-Page algorithm, and let $\beta_0 = \epsilon$.

Claim 4.2.1. *Let $\widehat{L}$ be any language. If $\widehat{L} \subseteq \mathcal{L}(G_p)$, then $\beta_i \backslash \widehat{L}$ is covered by (A_{β_i}, B_{β_i}) for all β_i, $0 \le i \le n$.*

Proof. Suppose there is a left-hand side β such that $\beta \backslash \widehat{L}$ is not covered by (A_β, B_β). But $\beta \backslash \widehat{L} \subseteq \beta \backslash \mathcal{L}(G_p)$. Thus every string in $\beta \backslash \widehat{L}$ can be derived from V_β. It is not hard to see that all such strings belong to $A_\beta \cup B_\beta \Sigma^*$. □

Claim 4.2.2. $\mathcal{L}(G_p) = \Sigma^*$ *if and only if Σ^* is covered by (A_{β_i}, B_{β_i}) for all β_i, $0 \le i \le n$.*

Proof. In order to show the "if" direction, suppose $\mathcal{L}(G_p) \neq \Sigma^*$. Let w be the shortest string with the following property: there exists a β_i such that w cannot be generated from V_{β_i} in one or more steps. Since Σ^* is covered by (A_{β_i}, B_{β_i}) for all β_i, there must be strings y and w' such

[1]See [1], at the beginning of page 70.

that $w = yw'$, $w' \in \Sigma^*$ and $y \in B_{\beta_i}$. Suppose the group of rules with V_{β_i} as the left-hand side is

$$V_{\beta_i} \rightarrow x_{i1} \mid \ldots \mid x_{im} \mid y_{i1}V_{\beta_{i_1}} \mid \ldots \mid y_{in}V_{\beta_{i_n}}$$

Then $y = y_{ij}$ for some j, and w' is a string shorter than w which cannot be generated from $V_{\beta_{i_j}}$. This is a contradiction. The "only if" direction follows straightforwardly from Claim 4.2.1. □

Thus the result follows. □

4.2 Restricted Equivalence Problem

We show that the following problem is PSPACE-hard:

Input: A DFA M and a prefix grammar $G_p = (\Sigma, S, P)$.

Question: Is $\mathcal{L}(\mathsf{M})$ equal to $\mathcal{L}(G_P)$?

The reduction is from the following problem:

Input: DFAs $\mathsf{M}_1, \ldots, \mathsf{M}_k$ over an alphabet Σ.

Question: $\mathcal{L}(\mathsf{M}_1) \cup \ldots \cup \mathcal{L}(\mathsf{M}_k) \overset{?}{=} \Sigma^*$

Let $\{c, c_1, \ldots, c_k\}$ be a set of new symbols and let $\Sigma' = \Sigma \cup \{c, c_1, \ldots, c_k\}$. A DFA M for the regular language

$$\bigcup_{i=1}^{k} \{c_i\} \circ \mathcal{L}(\mathsf{M}_i) \;\cup\; \{c\} \circ \Sigma^*$$

can be constructed in polynomial time. For each $1 \leq i \leq k$, construct prefix grammars $(\Sigma',\, S_i. P_i)$ for $\{c_i\} \circ \mathcal{L}(\mathsf{M}_i)$ using the Ravikumar-Quan algorithm [4]. Now let

$$\Pi \;=\; \Big(\Sigma',\, \bigcup_{i=1}^{k} S_i,\, P'\Big) \;\text{ where }\; P' \;=\; \bigcup_{i=1}^{k} P_i \;\cup\; \{c_i \rightarrow c \mid 1 \leq i \leq k\}$$

In other words, we take the union of all sets of base strings as the new base, and the union of all productions along with the extra productions that change every c_i to a c. The languages $\mathcal{L}(\mathsf{M})$ and $\mathcal{L}(\Pi)$ are equivalent if and only if $\{c\} \circ \Sigma^*$ is a subset of $\mathcal{L}(\Pi)$ if and only if $\mathcal{L}(\mathsf{M}_1) \cup \ldots \cup \mathcal{L}(\mathsf{M}_k) = \Sigma^*$. A similar result is proved by Lohrey and Petersen [3] where both inputs are prefix rewriting systems.

4.3 Determinism and Ambiguity

Theorem 4.3. *Checking whether a prefix grammar G_P is deterministic is decidable in polynomial time.*

Proof-idea: It can be shown that G_P is not deterministic if and only if there are left-hand sides β_i and β_j ($i \neq j$) such that β_i is a prefix of β_j and $\beta_j \setminus \mathcal{L}(G_P)$ is nonempty. □

Lemma 4.2. *A prefix grammar G_P is ambiguous if and only if one of following conditions holds:*

(a) There are base strings α_i and α_j such that $\alpha_i \to^+ \alpha_j$,

(b) There is a string w and distinct rules $\beta_i \to \gamma_i$ and $\beta_j \to \gamma_j$ such that $w = \gamma_i w' = \gamma_j w''$ for some w', w'' and both $\beta_i w'$ and $\beta_j w''$ belong to $\mathcal{L}(G_P)$. (In other words, there are two distinct derivations of w where the last steps are distinct.)

Proof. The "if" part is straightforward.

For the "only if" direction, if G_P is ambiguous, there must exist a string w, and at least two distinct ways to derive it.

w can either be a base string or a string derived from the base string. The case where w is a base string is covered by case (a).

If w is not a base string, then there must be two non-trivial (i.e., not of length zero) derivation sequences for w. If the last steps are the same for w, since the two derivations are distinct, we always can trace the steps of generating w, go back and find another string with shorter steps and the last steps are distinct. □

Theorem 4.4. *Ambiguity of prefix grammars can be checked in polynomial time.*

Proof. To prove this result, we will show the two conditions in Lemma 4.2 can be checked in polynomial time. First, we can apply the Frazier-Page algorithm to G_P, and get a G_R in polynomial time. For each language $\beta_j \backslash \mathcal{L}(G_R)$, we can create an NFA in linear time.

Case (a): For each α_i and γ_j in G_P, if γ_j is a prefix of α_i, then check whether the remaining suffix can be accepted by the NFA of $\beta_j \backslash \mathcal{L}(G_R)$.

Case (b): Suppose $\gamma_i w' = \gamma_j w''$, $i \neq j$. Then either γ_i is a prefix of γ_j or the other way. Without loss of the generality, let us assume that $\gamma_j = \gamma_i u$, then $\gamma_j w'' = \gamma_i u w'' = \gamma_i w'$. Thus $w' = uw''$.

We now need to check whether $\beta_i w' = \beta_i u w''$ and $\beta_j w''$ both belong to $\mathcal{L}(G_P)$. In other words, we need to check whether there are strings u, z such that $uz \in \beta_i \backslash \mathcal{L}(G_R)$ and $z \in \beta_j \backslash \mathcal{L}(G_R)$, or, equivalently, $uz \in u \cdot \beta_j \backslash \mathcal{L}(G_R)$.

NFAs for $\beta_i \backslash \mathcal{L}(G_R)$ and $u \cdot \beta_j \backslash \mathcal{L}(G_R)$ can be obtained in linear time from G_R. Now getting an NFA M for $\beta_i \backslash \mathcal{L}(G_R) \cap u \cdot \beta_j \backslash \mathcal{L}(G_R)$ can be done in $O(mn)$ time, where m and n are the number of states in the two NFAs. The cost of checking whether M accepts any string or not can be done in linear time.

This has to be repeated for all pairs of right-hand sides γ_i and γ_j where $i \neq j$ and one is a prefix of the other. Thus if there are p rules in G_P, we have to check at most $\frac{p(p-1)}{2}$ pairs of right-hand sides. The whole process is in polynomial time. □

5 Conclusion and Future work

The following table contrasts our complexity results with the results for the other formalisms for specifying regular languages.

	universality	ambiguity	equivalence to a DFA
DFA	P	N/A	P
NFA	PSPACE-complete	P	PSPACE-complete
Prefix grammars	P	P	PSPACE-complete

As is well-known, prefix rewriting systems can be viewed as ground term rewriting systems by considering each symbol as a monadic (unary) function. We plan to work on the unification, matching and related problems for ground unary term rewriting systems.

One immediate step is to investigate the two generalizations introduced by Greibach [5]. Both have the flavor of matching. Please refer to our full paper [6] for more details.

References

[1] M. Frazier and C. David Page Jr. Prefix grammars: an alternative characterization of the regular languages. *Information Processing Letters* 51(2): 67–71, July 26, 1994.

[2] J.E. Hopcroft, R. Motwani, J. Ullman. Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley, USA, 2006.

[3] M. Lohrey and H. Petersen. Complexity results for prefix grammars. *Informatique Théorique et Applications* 39(2):391–401, 2005.

[4] B. Ravikumar and L. Quan. Efficient algorithms for prefix grammars. Unpublished report, 2002.

[5] S.A. Greibach. A note on pushdown store automata and regular systems. *Proc. of the American Mathematical Society* 18 (1967) 263–268.

[6] https://www.cs.albany.edu/ dran/reports/prefix-grammars-report-4.pdf

Unification on the Run

Thomas Prokosch and François Bry

Institute for Informatics, Ludwig-Maximilian University of Munich, Germany
prokosch@pms.ifi.lmu.de bry@lmu.de

1 Introduction

Since Robinson introduced unification [6], many variations of Robinson's unification algorithm [6] have been proposed [5, 4, 1, 2, 3]. Indeed, "[t]he unification algorithm as originally proposed can be extremely inefficient" [4, page 259]. Improving over Robinson's original unification algorithm has been attempted in three manners:

- by changing when the occurs check is performed (as done for example in [4]),
- by sharing instead of copying subexpressions to which variables are bound (as first suggested by [5, 1]),
- by simplifying the already computed substitution or the expressions still to unify (as done with the rules "variable elimination," "reduction" and "compactification" of [4]).

An issue which has received little attention is whether a potential improvement of a unification algorithm is realizable in run-time systems. Suggestions of the afore-mentioned third kind seem hardly realizable at reasonable costs in a runtime system. The article [3] stresses that many suggestions for improving Robinson's unification algorithm are not successful in practice. That article reports on an empirical evaluation of the performance of the unification algorithms by Robinson [6], Martelli-Montanari [4], Escalada-Ghallab [2], and a formerly unpublished improvement of Robinson's algorithm showing that, unexpectedly, Robinson's unification algorithm is the most efficient!

This article reports on a refinement of Robinson's original unification algorithm based on

- an in-memory representation of expressions, an issue not considered by Robinson,
- single left-to-right runs through, or traversals, of the expressions tested for unifiability,
- keeping track of the matching or unification mode of the sub-expressions so far run through, an approach so far not considered,
- and exploiting the afore-mentioned in-memory representation for detecting when occurs checks are unnecessary.

2 Preliminaries

Finitely many non-variable symbols and infinitely many variables are considered. In the following, the lower case letters $a, b, c, \ldots, z$ with the exception of v denote the non-variable symbols, $v_0, v_1, v_2, \ldots$ (with subscripts) denote the variables. v^i (with a superscript) denotes an arbitrary variable.

An *expression* is either a first-order term or a first-order atomic formula. Expressions are defined from constructors as follows. A *constructor* is a pair s/a with s a non-variable symbol and a one of finitely many arities associated with the symbol s. There are finitely many constructors. An *expression* is either a *variable* or a *non-variable expression.* A non-variable expression is either a constructor s of arity 0, $s/0$, or it has the form $s(e_1, \ldots, e_n)$ where s/n is a constructor of arity $n \geq 1$ and $e_1, \ldots, e_n$ are expressions. $e_1, \ldots,$ and e_n are the *direct subexpressions* of $s(e_1, \ldots, e_n)$. Two expressions are *variable-disjoint* if none of the variables occurring in the one expression occurs in the other.

An expression $s(e_1, \ldots, e_n)$ is in *standard notation.* It can also be written without parentheses in *prefix notation* (or *Polish* or *Lukasiewicz notation*) as $s/n\ e_1/a_1\ \ldots\ e_n/a_n$ where a_i is the arity of expression e_i $(1 \leq i \leq n)$. While the standard notation is easier to read the (parenthesis-free) prefix form is necessary for linear (or parenthesis-free) processor languages.

3 In-memory representations and dereferencing

The representation of an expression in the memory of a run-time system is based on the expression's prefix notation. Assuming that a constructor and a variable are stored in 4 bytes and storage begins at address 0, the representation of $f(a, v_1, b, v_1)$ is:

0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19
$f/4$	$a/0$	nil	$b/0$	8

The leftmost, or first, occurrence of the variable v_1 is represented by the value nil which indicates that the variable is unbound. The second occurrence of the variable v_1 is represented by an offset: The address of this second occurrence's representation, 16, minus the offset, 8, is the address of the representation of the variable's first occurrence, 8. Occurrences of (the representation of) a variable like the second occurrence of v_1 in $f(a, v_1, b, v_1)$ and the cell representing such variables like the cell at address 16 in the above representation of $f(a, v_1, b, v_1)$ are called a *locally bound variables* or *offset variables.*

Two properties of an expression representation are worth stressing:

1. Variables' names are irrelevant to expression representations, that is, variant expressions have the same representation except for the memory addresses.
2. Two distinct expression representations do not share variables.

Representation of substitutions An elementary substitution $\{v^i \mapsto e\}$ can be seen as a pair (address of v^i, address of the representation of e). If the representation of $p(a, v_1, v_1)$ is stored at address 0 and the representation of $q(b, v_3)$ at address 23, the substitution application $p(a, v_1, v_1)\{v_1 \mapsto q(b, v_3)\}$ is represented before substitution application as:

0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19 20 21 22	23 24 25 26	27 28 29 30	31 32 33 34
$p/3$	$a/0$	nil	4		q/2	b/0	nil

and after the application of $p(a, v_1, v_1)\{v_1 \mapsto q(b, v_3)\}$ as:

0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19 20 21 22	23 24 25 26	27 28 29 30	31 32 33 34
$p/3$	$a/0$	23	4		q/2	b/0	nil

Observe that the cell representing the second occurrence of the variable v_1 (cell at 12) keeps its offset (4) unchanged. Thus, binding a variable v which occurs in an expression e to an expression e' consists in storing at the leftmost occurrence of v in the representation of e the address of the representation of e', leaving unchanged further occurrences of v in the representation of e. This approach to binding variables makes the representation of a substitution application unique.

Dereferencing Consider the following two representations of $f(a, v_1, v_2, v_1, g(v_1))$:

0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27
$f/5$	$a/0$	nil	nil	8	g/1	16

0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27 28 29 30 31 32 33 34 35	36 37 38 39	40 41 42 43
$f/5$	$a/0$	nil	nil	8	36		g/1	8

The first representations of $f(a, v_1, v_2, v_1, g(v_1))$ is dereferenced because, except for the representations of the second and third occurrences of the variable v_1, the variables' value are nil. The second and third occurrences of v_1 cannot be dereferenced like a pointer because this would result in the following representation of $f(a, v_1, v_2, v_3, g(v_4))$, not of $f(a, v_1, v_2, v_1, g(v_1))$:

0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27 28 29 30 31 32 33 34 35	36 37 38 39	40 41 42 43
$f/5$	$a/0$	nil	nil	nil	36		g/1	nil

A dereferenced representation of an expression e is generated from any representation R_e of e as follows: While traversing R_e from left to right, if the cell reached contains a constructor or nil, or the offset of a locally bound variable, then copy the cell's content to a new cell. Otherwise (the token reached is a non-locally bound variable v storing the address E of an expression representation), recursively dereference the expression representation at address E.

In dereferencing, care must be given not to trespass expression representations' ends in recursive calls. This is cared for by using as follows the constructors' arities during a left-to-right traversal of an expression representation E: Let R denote the number of remaining (sub-)expression representations; set $R := 1$ before traversing E, at each constructor s/n perform the update $R := R-1+n$ (-1 for the (sub-)expression beginning at that constructor, and $+n$ for the n subexpression representations now to be traversed), and at each variable perform the update $R := R - 1$. The expression representation's end is reached when $R = 0$.

4 A matching-unification algorithm

A call to `unif(e1, e2)` performs a left-to-right run through, or traversal, of the representations of expressions e_1 and e_2 stored at the addresses `e1` and `e2` respectively. The algorithm makes uses of the variables

- `A`: One of `VR` (variant), `SI` (strict instance), `SG` (strict generalisation), `OU` (only unifiable, i.e. unifiable but none of `VR`, `SI`, `SG`), or `NU` (not unifiable). Initialisation: `A := VR`
- `R1` and `R2`: The end of the expression representation at address `e1` (`e2`, respectively) is reached when `R1 = 0` (`R2 = 0`, respectively). Initialisation: `R1 := arity(e1); R2 := arity(e2)`
- `S1` and `S2`: Substitutions for variables in the expression representations at addresses `e1` and `e2` respectively. Initialisation: `S1 := []; S2 := []`

`S+R` denotes the list obtained by appending `R` to the list `S`. `a += b` is shorthand for `a := a+b`. The algorithm make uses of the functions:

- `type(e)`: Type of the value stored in the cell with address `e`: `cons` if that value is a constructor `s/n`, `novar` if it is a non-offset variable, or `ofvar` if it is an offset variable (i.e. referring to a local non-offset variable).
- `value(e)`: Value stored in the cell with address `e` (possibly `nil`).

- `arity(e)`: Arity of the constructor or variable stored at address `e`, the arity of a variable being `0`.
- `deref(e, S)`: The application of a substitution `S` to the expression representation at address `e`.
- `occurs-in(e1, e2)`: Checks whether a variable at address `e1` occurs-in the expression representation at address `e2`.

The algorithm consists of 16 cases given in 4 tables. Each case is characterised by `type(e1)`, `type(e2)` and the (dereferenced) expression representations at addresses `e1` and `e2`.

	type(e2) = cons value(e2) = s2/a2	type(e2) = novar value(e2) = nil
type(e1) = cons value(e1) = s1/a1	if value(e1) = value(e2) then R1 += arity(e1) R2 += arity(e2) else A := NU	S2 += (e2, deref(e1, S1)) R1 += arity(e1) if A = VR then A := SI if A = SG then A := OU
type(e1) = novar value(e1) = nil	S1 += (e1, deref(e2,S2)) R2 += arity(e2) if A = VR then A := SG if A = SI then A := OU	S2 += (e2, e1)

If both expressions are unbound variables then the variable at address `e2` is bound to that at address `e1` what avoids generating cyclic substitutions.

	type(e2) = ofvar deref(e2,S2) != nil	type(e2) = ofvar deref(e2,S2) = nil
type(e1) = cons value(e1) = s1/a1	unif(deref(e1, S1), deref(e2, S2))	de1 := deref(e1, S1) de2 := deref(e2, S2) if occurs-in(de2, de1) then A := NU else R1 += arity(e1) S1 += (de2, de1) S2 += (de2, de1) A := OU
type(e1) = novar value(e1) = nil	S1 += (e1, deref(e2,S2)) if A = VR then A := SG if A = SI then A := OU	S1 += (e1, deref(e2,S2)) if A = VR then A := SG if A = SI then A := OU

If the representation at address `e1` starts with a constructor and the address `e2` is a bound variable, then the algorithm is recursively called. Otherwise, a binding is only generated if the occurs check fails. Unbound variables can be bound to any variable whether bound or unbound.

	type(e2) = cons value(e2) = s2/a2	type(e2) = novar value(e2) = nil
type(e1) = ofvar deref(e1,S1) != nil	unif(deref(e1, S1), deref(e2, S2))	S2 += (e2, deref(e1, S1)) if A = VR then A := SI if A = SG then A := OU
type(e1) = ofvar deref(e1,S1) = nil	de1 := deref(e1, S1) de2 := deref(e2, S2) if occurs-in(de1, de2) then A := NU else R2 += arity(e2) S1 += (de1, de2) S2 += (de1, de2) A := OU	S2 += (e2, deref(e1, S1)) if A = VR then A := SI if A = SG then A := OU

The four cases above are symmetrical to the preceding four cases.

	`type(e2) = ofvar` `deref(e2,S2) != nil`	`type(e2) = ofvar` `deref(e2,S2) = nil`
`type(e1) = ofvar` `deref(e1,S1) != nil`	`unif(deref(e1, S1),` `deref(e2, S2))`	`de1 := deref(e1, S1)` `de2 := deref(e2, S2)` `if occurs-in(de2, de1)` `then A := NU` `else A := OU` `S1 += (de2, de1)` `S2 += (de2, de1)`
`type(e1) = ofvar` `deref(e1,S1) = nil`	`de1 := deref(e1, S1)` `de2 := deref(e2, S2)` `if occurs-in(de1, de2)` `then A := NU` `else A := OU` `S1 += (de1, de2)` `S2 += (de1, de2)`	`de1 := deref(e1, S1)` `de2 := deref(e2, S2)` `S1 += (de2, de1)` `S2 += (de2, de1)`

Two offset variables pointing to bound variables result in a recursive call after applying substitutions `S1` and `S2`. Two offset variables only one of which points to an unbound variable require an occurs check. However, two offset variables both pointing to unbound variables make an occurs check unnecessary.

The time complexity of the algorithm is dominated by both the occurs check and the compatibility check both of which depend on the lengths of the expressions and the number of offset variables (ofvar-nb) bound to non-offset variables. Thus, the time complexity of the algorithm given above is in $O(\max(\text{length}(e_1), \text{length}(e_2)) \times \max(1, \text{ofvar-nb}(e_1) + \text{ofvar-nb}(e_2)))$.

To sum up, keeping track of the matching mode, as long as the expression prefixes traversed match, and dinstinguishing between locally bound, or offset, variables, and non-locally bound variables makes it possible to avoid unnecessary occurs check.

Further work will be devoted to a experimental comparison of the algorithm given above with formerly proposed unification algorithms.

References

[1] Dennis de Champeaux. About the Paterson-Wegman linear unification algorithm. *Journal of Computer and System Sciences*, 32(1):79–90, 1986.

[2] Gonzalo Escalada-Imaz and Malik Ghallab. A practically efficient and almost linear unification algorithm. *Artificial Intelligence*, 36(2):249—263, September 1988.

[3] Krystof Hoder and Andrei Voronkov. Comparing unification algorithms in first-order theorem proving. In Bärbel Mertsching, Marcus Hund, and Muhammad Zaheer Aziz, editors, *Proceedings of KI 2009 – Advances in Artificial Intelligence, 32nd Annual German Conference on AI*, number 5803 in LNCS, pages 435–443. Springer, 2009.

[4] Alberti Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transaction on Programming Language Systems (TOPLAS)*, 4(2):258–282, April 1982.

[5] Michael Stewart Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.

[6] John Allan Robinson. A machine-oriented logic based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.

Author Index