– Informal Proceedings –

33rd International Workshop on Unification

Dortmund, June 24, 2019

Preface

This volume contains the papers presented at UNIF 2019: 33rd International Workshop on Unification held on June 24, 2019 in Dortmund, Germany.

UNIF 2019 was affiliated with the **4th International Conference on Formal Structures for Computation and Deduction FSCD 2019**. There were 9 submissions, each one was reviewed by at least 3 Program Committee members and external reviewers. The committee decided to accept the 9 papers for presentation.

The program also includes 2 invited talks: Jörg Siekmann on "**Rethinking Unification Theory**", and Narciso Martí Oliet "**Maude Strategies for Narrowing**".

We would like to thank all members of the Program Committee and the subreviewers for their high quality reviews and discussion held on the EasyChair platform that assured thescientific standard of UNIF in its 33rd edition. Finally, we would like to thank the UNIF Steering Committee and the FSCD scientific and organising team for the support before and during the workshop.

December 12, 2019
Dallas
Brasília

Serdar Erbartur
Daniele Nantes-Sobrinho

Program Committee

Serdar Erbatur	LMU Munich
Daniele Nantes	Universidade de Brasília
Takahito Aoto	Niigata University
Alexander Baumgartner	University of Chile
Mauricio Ayala Rincón	Universidade de Brasília
Evelyne Contejean	LRI, CNRS, Univ Paris-Sud, Orsay
Kimberly Cornell	The College of Saint Rose
Santiago Escobar	Universitat Politècnica de València
Maribel Fernández	King's College London
Temur Kutsia	RISC- Johannes Kepler University Linz
Jordy Levy	IIIA - CSIC
Hai Lin	Shenyang Normal University
Christopher Lynch	Clarkson University
Andrew M. Marshall	University of Mary Washington
Catherine Meadows	US Naval Research Laboratory
Paliath Narendran	University at Albany–SUNY
Christophe Ringeissen	INRIA
David Sabel	Ludwig Maximilian University of Munich
René Thiemann	University of Innsbruck
Manfred Schmidt-Schauss	Goethe-University Frankfurt am Main
Ralf Treinen	IRIF- Université Paris-Diderot
Daniel Lima Ventura	Universidade Federal de Goiás

Additional Reviewers

Veena Ravishankar	University of Mary Washington

List of Accepted Papers

Rethinking UnificationTheory

Jörg Siekmann
Saarland University/DFKI
Stuhlsatzenhausweg 3, D-66123 Saarbrücken

June 5, 2019

Let us lean back for a minute and reflect on the motivation for our field. Apart from its theoretical interest, i.e. the structural relationships among and within equational theories, there is the practical motivation, most clearly expressed in Gordon Plotkin´s seminal paper from 1972 [10]: we want to take certain troublesome axioms, like associativity or commutativity, out of the axiom set for an automated deduction system that may lead the system to go astray. Instead - so the proposal - they should be "built-in".

Now the past 30 years - the first workshop in Val D'Ajol was in 1987 - have revealed an astonishing complexity even for those simple axioms - not so astonishing after all, for someone familiar with semigroup theory [3] and more generally the results about equational theories [7, 6].

Historically, the development of unification theory began with the central notion of *a most general unifier* based on the *subsumption order*. A unifier σ is most general, if it subsumes any other unifier τ, that is, if there is a substitution λ with $\tau =_E \sigma\lambda$, where E is an equational theory and $=_E$ denotes equality under E. Since there is in general more than one most general unifier for a unification problem under an equational theory E, called *E-Unification*, we have the notion of a complete and minimal set of unifiers under E for a unification problem Γ, denoted as $\mu\mathcal{U}\Sigma_E(\Gamma)$. This set is still the basic notion in unification theory today.

But, unfortunately, the subsumption quasi order is not a well founded quasi order, which is the reason why for certain equational theories there are solvable E-unification problems, but the set $\mu\mathcal{U}\Sigma_E(\Gamma)$ does not exist. We say these problems are of type nullary in the unification hierarchy [11]. In order to overcome this problem and also to substantially reduce the number of most general unifiers in nonnullary theories, we introduced the notion of *essential unification.* An *essential unifier*, as introduced by Hoche and Szabo [2], generalizes the notion of a most general unifier with a most pleasant effect: the set of essential unifiers is often much smaller than the set of most general unifiers. Essential unification may even reduce an infinitary theory to an essentially finitary theory. For example the one variable string unification problem is essentially finitary whereas it is infinitary in the usual sense [1]. A most drastic reduction is obtained for idempotent semigroups, or bands as they are called in computer science, which

are of type nullary: there exist two unifiable terms s and t, but the set of most general unifiers does not exist. This is in stark contrast to essential unification: the set of essential unifiers for bands always exists and it is finite [2].

The key idea for essential unification is to base the notion of generality not on the standard subsumption order for terms with the associated subsumption order for substitutions, but on the well known *encompassment order* for terms. We also extended this ordering for terms to an order for substitutions and proposed the encompassment order as a more natural relation for minimal and complete sets of E-unifiers, calling them *essential unifiers*, denoted as $e\mathcal{U}\Sigma_E(\Gamma)$. If $\mu\mathcal{U}\Sigma_E(\Gamma)$ exists, then $e\mathcal{U}\Sigma_E(\Gamma) \subseteq \mu\mathcal{U}\Sigma_E(\Gamma)$, i.e. it is always a subset. An interesting effect is, that there are cases of an equational theory E, for which the complete set of most general unifiers does not exist, the *minimal and complete set of essential unifiers* however does exist.

Unfortunately again, the encompassment order is not a well founded quasi ordering, that is, there are still theories with a solvable unification problem, for which a minimal and complete set of essential unifiers can not be obtained.

In a more recent paper [8][9] we therefore proposed a third approach, namely the extension of the well known *homeomorphic embedding of terms* to a *homeomorphic embedding of substitutions (modulo E),* known as equational embedding in the literature, and examined the set of E-unifiers under this ordering using the seminal *tree embedding theorem* or Kruskal's Theorem [4, 5] as it is called.

The main result of this latest approach is, that for any solvable E-unification problem the minimal and complete set of E-unifiers always *exists* and it is even smaller than the set of essential unifiers. Under some additional conditions, called *pure equational embedding*, it is **always finite**.

Our main observation is that for unification theory *subsumption* is just a special case of *encompassment*, which in turn is a special case of *homeomorphic embedding*.

References

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

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

[3] J. M. Howie. *An Introduction to Semigroup Theory.* Academic Press, 1976.

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

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

[6] M. Lothaire. *Combinatorics on words*, volume 17 of *Encyclopedia of Mathematics*. Addison-Wesley1997, reprinted in: Cambridge University Press, Cambridge mathematical library, 1983.

[7] M. Lothaire. *Algebraic combinatorics on words*. Cambridge University Press, 2002.

[8] Szabo P. and Siekmann J. Unification based on generalised embedding. In *Proceedings of UNIF 18*, 2018.

[9] Szabo P. and Siekmann J. Unification based on generalized embedding. *Mathematical Structures in Computer Science*, submitted 2019.

[10] G. Plotkin. Building-in equational theories. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 73–90. Edinburgh University Press, 1972.

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

Maude strategies for narrowing

Narciso Martí-Oliet

Universidad Complutense Madrid,
Department of Informatic Systems and Programming,
Madrid, Spain
narciso@sip.ucm.es

Abstract

Strategies allow modular separation between the rules that specify a system and the way that these rules are applied. They can be used both to implement and test different algorithms over a given specification or to drive the search of solutions to reachability problems reducing the state space. A strategy language for rewriting using Maude has been developed and implemented during the last years. The language controls when a basic step of rewriting is taken, by using sequences, tests, and other combinators. The next Maude release will include full support for this strategy language.

The ongoing work that we present extends the use of this strategy language to narrowing, which is a more general method than rewriting and may have a much larger state space for a given problem, because whereas rewriting uses matching for solving reachability problems, narrowing uses unification. The application of a basic narrowing step is controlled in this case by means of a subset of the strategy combinators defined for rewriting. Narrowing strategies can turn an infinite state space into a finite one, as it has already been shown in an example using the prototype that we have developed. This prototype is a proof of concept to settle the basis of what can be achieved using strategies in a narrowing environment.

A Coq Formalization of Boolean Unification

Daniel J. Dougherty

Worcester Polytechnic Institute, Worcester, MA, U.S.A.
dd@wpi.edu

Abstract

We report on a verified implementation of two (well-known) algorithms for unification modulo the theory of Boolean rings: Löwenheim's method and the method of Successive Variable Elimination. The implementations and proofs of correctness were done in the Coq proof assistant; we view this contribution as an early step in a larger project of developing a suite of verified implementations of equational unification algorithms.

1 Introduction

There is a significant emerging body of work devoted to formalizing mathematics and to specifying and verifying algorithms and systems using proof assistants such as Coq, Isabelle, HOL, and others. In this paper we report on first steps in a project of developing—in the proof assistant Coq—a library of verified E-unification algorithms. Our long-term goal is to build a resource of reusable data structures, algorithms, and theorems that will be a resource to researchers in mathematics and computer science.

Boolean unification is an attractive choice for a first theory to formalize. Boolean rings arise in many contexts, in connection with satisfiability, with circuit synthesis, and with logic programming, [BS87], and play a role in general topology, algebra, lattice theory. Boolean unification has a well-behaved theory (it is decidable and unitary) and there are interesting algorithms suitable for "native" implementation (as opposed to, for example, AC-unification, which naturally involves passing to reasoning about Diophantine equations).

To date we have implemented and proved correct two well-known algorithms for unification modulo the theory of Boolean rings, in the absence of free function symbols: Löwenheim's method and the method of Successive Variable Elimination. The code comprises about 1600 lines of definitions (of concepts and algorithms) and about 2900 lines of proof. It is available at http://web.cs.wpi.edu/~dd/unif.pdf. We have not made the Coq source available because, as described below, our development is being actively refined.

Our main goal in the present paper is to generate feedback from the unification community about the usefulness of a project like this and a discussion about future directions to pursue.

2 Related Work

There have been several verified implementations of *syntactic* unification, too many to attempt to list here. But we do note that in last year's UNIF (2018) Kasper Fabæch Brandt, Anders Schlichtkrull, and Jøorgen Villadsen present [BSV18] an Isabelle coding of syntactic unification, with a formal proof of termination though not of correctness. The related work section of that paper cites a selection of other verified developments of syntactic unification in various tools. The problem of formalizing unification modulo theories has received much less attention. A notable exception is the Coq formalization of associative-commutative matching by Evelyne Contejean in [Con04].

There are many examples of formal development of algebraic theories, again, far too many to summarize here, but there are relatively few treatments of equational logic in a general, universal-algebra sense.

Two early examples are [Cap99, Dom08], but the example most relevant to the current project is the CoLoR system [BK11], part of a family of tools also including CiME [CCF+11] and Coccinelle [Con]. CoLoR is primarily a (Coq) library for analyzing termination of rewrite relations. Although it does not treat semantic unification, it has an extensive infrastructure for reasoning about term-rewriting generally, and has been a valuable resource for the current project.

During the 2018-19 academic a WPI undergraduate project team (Spyridon Antonatos, Matthew McDonald, Dylan Richardson, and Joseph St. Pierre) worked on Boolean unification in parallel with the author, with some differences in data structures and outcomes.

3 Preliminaries

We assume familiarity with standard notions of equational logic and unification [BS01, BN98].

A *Boolean ring* is a ring in which every element x is idempotent, that is, $x^2 = x$. The specific set $\mathbb{B}$ of axioms we use in our development is the following.

$$\begin{array}{lll} 0+x=x & 1*x=x & (x+y)+z=x+(y+z) \\ (x*y)*z=x*(y*z) & x+y=y+x & x*(y+z)=(x*y)+(x*z) \\ (y+z)*x=(y*x)+(z*x) & x+x=0 & x*x=x \end{array}$$

In what follows we will use $s =_B t$ to mean that s and t are provably equal modulo $\mathbb{B}$; we use $s \stackrel{?}{=}_B t$ to denote a unification problem modulo $\mathbb{B}$.

The axioms other than the last are precisely the axioms for a ring, with the omission of an additive inverse and the inclusion of the axiom $x+x=0$. But in the presence of an additive inverse, the omnipotence axiom $x^2 = x$ entails $x+x=0$, and so the additive inverse operator is in fact the identity. [Proof: expand $x+x$ as $(x+x)(x+x) =_B xx+xx+xx+xx =_B x+x+x+x$. Cancellation yields $0 =_B x+x$.] Presenting the theory as above, with $x+x=0$ as an axiom, rather than using a separate additive inverse operator, thus simplifies a formal development: there is one fewer function symbol in the signature.

Two easy but significant consequences of the axioms are (i) commutativity of multiplication: $xy =_B yx$ [Proof: $x+y =_B (x+y)(x+y) =_B xx+xy+yx+yy =_B x+xy+yx+y$. Cancellation yields $0 =_B xy+yx$] and (ii) the fact that every element is a 0-divisor [since $x(1+x) =_B x+xx =_B x+x =_B 0$].

Since an equation $u =_B v$ is equivalent to the equation $u+v =_B 0$,

As usual in the presence of a "inverse" operator, unification problems $u \stackrel{?}{=}_B v$ reduce to matching problems of the form $t =_B 0$. So in the rest of the paper, instead of speaking of *unifying* two terms u and v, we will often speak of *solving* a term t, meaning finding a substitution σ such that $\sigma t =_B 0$.

There is a well-known translation between Boolean rings and Boolean algebras, under which multiplication corresponds to conjunction and addition to exclusive-or. Any field of sets yields a Boolean ring under intersection and symmetric difference. In fact, by the Stone Representation Theorem (and the relationship between Boolean rings and Boolean algebras) every Boolean ring is isomorphic to a field of sets [Sto36].

Unification Of course unification modulo the theory of rings is undecidable (by reduction to Hilbert's 10th problem). By contrast, there are several well-known algorithms for $\mathbb{B}$-unification. The decision problem for $\mathbb{B}$-unification (without free constants) is essentially the satisfiability problem, and so *NP*-complete, since, as noted in [BN98], $s =_B t$ under $\mathbb{B}$ if and only if s and t are equal terms over the boolean ring $\mathcal{B}_2$. The decision problem for $\mathbb{B}$-unification is Π_2^p-complete for a signature with free constants and is PSPACE-complete for a signature with free non-constant function symbols [Baa98] (by reduction to the validity of sentences of quantified boolean formulas).

Space considerations preclude a full discussion here of the history of algorithms for $\mathbb{B}$-unification (see [BN98] for an introduction) but we do note that $\mathbb{B}$-unification over the basic signature even in the presence of free constants is unitary; it is finitary when arbitrary free function symbols are allowed. Algorithms for constructing unifiers in the basic case can be found in [Löw08, BS87]; Martin and Nipkow [MN89] treat the case of free constants. Our formalizations treat the methods of Löwenheim [Löw08] and of Variable Elimination [Boo47, BS87]. It is not hard to show that simultaneous $\mathbb{B}$-unification reduces to $\mathbb{B}$-unification of single equation, even in the absence of auxiliary function symbols.

4 The Development

Coq is simultaneously a functional programming language and an environment for constructing proofs. The programming language of Coq closely resembles OCaml; the chief differences are the richer type system and the requirement that every function be terminating.

We do not have space here to discuss both algorithms in detail, and even so we can only hint at what the implementation and verification look like.

We represent terms by the following inductive data type. T0, T1, A, and M denote 0, 1, addition, and multiplication, respectively.

```
Inductive bterm : Type :=
| T0 : bterm | T1 : bterm | V : var → bterm
| A : bterm → bterm → bterm | M : bterm → bterm → bterm
```

Using Coq's Notation facility we use x +' y and x ∗' y as syntactic sugar for A x y and M x y and (the quotes are added because symbols + and ∗ have standard-library meanings in Coq).

The equational theory $\mathbb{B}$ is captured by the inductive relation eqv below; we introduce a Coq Notation (s == t) as infix syntactic sugar for (eqv s t). To save space, only a selection of the axioms are given here.

```
Inductive eqv : bterm → bterm → Prop :=
| assocA : forall x y z, x +' y == y +' x
...
|eqv_ref : forall x , x == x
...
| A_compat : forall x x', x == x' → forall y y', y == y' → x +' y == x' +' y'
```

The cases not shown capture the other rules of $\mathbb{B}$ and a set of equations expressing that eqv is an equivalence relation compatible with respect to the ring operations.

4.1 Löwenheim's Method

The algorithm has two stages. To solve $t \stackrel{?}{=}_B 0$, (i) first search for any substitution γ with $\gamma(t) = 0$; (ii) if such a γ is found, return the substitution

$$\sigma \stackrel{\text{def}}{=} \{v := (t+1) * v + (t) * \gamma(v) \mid v \in Vars(t)\}.$$

Remarkably, this will be a most general solution. To ensure that part (i) terminates, we search through the finitely many substitutions γ defined on $Vars(t)$ that only take on values $T0$ or $T1$. Of course, a key part of the verification is that this restriction is complete. If no such γ is found we report that t is not solvable.

For example, to solve $xy \stackrel{?}{=}_B 0$, we can start with $\gamma = [x := 0; y := 1]$; then σ is, after simplification,

$$x := xy + x \qquad y := y$$

Note that σ is more general than γ, indeed, $\gamma \circ \sigma = \gamma$. It is, perhaps, not obvious that σ is actually most general.

Löwenheim's method in Coq The most we can do in this limited space is to show three functions that correspond to the algorithm outlined above, and the corresponding shape of the correctness proof. The functions below, respectively (i) compute an initial solution (returning "None" if none found); (ii) build a new substitution, the "lowenheim lift," out of a given substitution and a given term, and (iii) bundle these together as a constructive decision procedure.

```
Definition ground_soln (t: bterm) : option sub :=
find (fun s ⇒ eqvb (s ' t) T0) (all_01subs_bterm t).

Definition lowenheim_lift (t: bterm) (tau: sub) (x: var) : bterm :=
  if inb x (vars_bterm t)
  then ((t +' T1) *' (V x) +' t *' (apply_sub tau (V x)) )
  else (V x).

Definition solve_lowenheim (t: bterm) :=
  option_map (lowenheim_lift t) (ground_soln t).
```

To prove correctness we must show that if ground_soln t returns None then t is not solvable, and otherwise, lowenheim_lift constructs a most general solution from the substitution returned by ground_soln t. The structure of the correctness claim is the same as the structure of solve_lowenheim itself.

```
Theorem lowenheim_correct (t: bterm) :
match (solve_lowenheim t) with
| None ⇒ ∼(solvable t)
| Some sigma ⇒ mgu_strong sigma t
end .
```

4.2 Successive Variable Elimination

This algorithm depends the fact that for any bterm t and any variable x we can compute quotient term q and remainder term r such that $(t =_B qx + r)$. Using this, the variable elimination algorithm for solvability of a term t is as follows:

1. choose a variable x from among the variables of t, and compute q and r such that t == q x + r.
2. set t' to be $(q+1)r$
3. compute a most general substitution σ' solving t'
4. then the following substitution is a most general solution for t

$$\sigma \stackrel{\text{def}}{=} \{x \mapsto x * ((\sigma' q) + 1) + (\sigma r) \mid x \in vars\, t\}$$

Correctness Argument The correctness argument proceeds by induction on the number of variables in the original term t. Then each of the claims below is captured by a lemma in the formalization.

- Any ground term can be reduced to either 1 or 0, and so is either unsolvable or has the identity as a most general solution.
- As we go from t down through the recursive calls, if t is solvable then each derived bterm $t' \stackrel{\text{def}}{=} (q+1)r$ is solvable.
- If σ' is a most general solution for a derived bterm t', then the updated substitution computed from σ' is a most general solution for the original bterm .

A few words about the factorization of a term t into the form $qx+r$ give some glimpse into the kind of considerations that arise in a formalization in a proof assistant. The construction as described in [BN98] relies on the fact that any term can be represented in polynomial form, as a set of monomials, each of which is a set a variables. Extracting q and r from t is very simple under this representation. But as described in the next section, polynomial form is not the most convenient representation for us. So we construct recursive functions to compute q and r given x and the standard representation of t. For example when t is t_1t_2 (the most interesting case) and is recursively factored as q_1x+r_1 and q_2x+r_2, then

$$\begin{aligned} t_1t_2 &=_B (q_1x+r_1)(q_2x+r_2) \\ &=_B q_1q_2xx+q_1r_2x+q_2r_1x+r_1r_2 \\ &=_B (q_1q_2+q_1+q_2)x+(r_1r_2) \end{aligned}$$

This is mathematically straightforward, and as a programming task it is easy to code a pair of mutually recursive functions handling cases on the form of t. But there is a wrinkle: in Coq all functions must provably terminate, and Coq's automatic techniques for verifying termination do not suffice for the mutual recursion suggested by the above. There are techniques for supplying Coq with a termination justification developed by the user, though this is sometimes a non-trivial task. But luckily in the present case, we can disentangle the calculations: a function computing the remainder r of a term t after division by x can be computed directly, first, and then used in computing the quotient q.

5 Design Choices and Lessons Learned

$\mathbb{B}$-equality as an Inductive Predicate We might have, naively, captured each of the $\mathbb{B}$-equations as a Coq Axiom, such as

```
Axiom invA : forall x, x +' x = T0
```

where we have used the primitive Coq equality predicate = instead of eqv. But this would be a mistake: the resulting theory would be logically inconsistent. A fundamental principle of inductively defined data (such as terms) is that if C_1 and C_2 are distinct constructors (such as +' and T0), then any equation $C_1\vec{s} = C_2\vec{t}$ entails falsehood.

This is why we represent equality under $\mathbb{B}$ as a *Inductive Predicate*. When eqv is defined inductively it means that (eqv s t) holds *precisely* when it follows from the rules in the inductive definition. (And Coq provides tactics, such as inversion, to exploit this fact.) It follows that eqv really does precisely capture provable equality under $\mathbb{B}$.

By the way, this inductive-predicate approach differs from systems such as CoLoR or treatments of algebraic structure in, e.g., the Mathematical Component Library. The difference is that we are not interested in deriving just those theorems that hold in all $\mathbb{B}$-structures, we are interested in one particular structure, the term model for $\mathbb{B}$. So for example, we can *prove* the proposition $\sim$(T0 == T1), which is not a theorem of $\mathbb{B}$.

Terms and Polynomials (The discussion in this paragraph is specific to the theory $\mathbb{B}$.) For any term t there is a $\mathbb{B}$-equivalent term p in polynomial form: monomials are sets of variables and polynomials are set of monomials. If we totally order variables and—then, lexicographically—monomials, every term can be reduced to a unique polynomial normal form modulo $\mathbb{B}$. This is very convenient theoretically, and is exploited in several places in [BN98], and so it suggests a representation for terms in the formalization. But as my undergraduate team grew to realize, this is not a particularly easy representation to work with. Arithmetic operations are not easily *directly* described as operations on these sets-of-sets; there is post-processing to be done to restore the required invariants. In turn, these repairs must be proven correct, for each operation, and the verification effort grows burdensome out of proportion with the intellectual convenience of having unique normal forms. Eschewing this approach meant that certain techniques and results, such as (i) the fact that $s =_B t$ is $\mathbb{B}$-provable if and only if s and t are equal polynomials over 2-element Boolean rings, and (ii) the factoring lemma crucial to the Variable Elimination algorithm, had to be defined and verified without recourse to the most mathematically-natural data structure, which is polynomial form.

Terms as a Signature-specific Inductive Data Type The data structure shown in Section 4 is what the current development uses, and it has much to recommend it. It is direct, it supports programming by pattern-matching, and Coq automatically computes an induction principle that makes tactic-based reasoning convenient. But a significant drawback to this approach is the fact that it is not generic across signatures. Much of the infrastructure of the verification effort comprises standard results about variable occurrences, substitutions, and so forth. Conceptually the definitions of, and proofs about, these notions are the same for any signature, but with the signature-specific approach this infrastructure would, in a formal development, have to be constructed anew for each signature. To see a very simple example of this lack of robustness, note that for a theory E we often want to explore E-unification between terms that may have free function symbols, symbols not occurring in the signature of E. Specifying terms as a signature-specific type obviously precludes anything but an ad-hoc approach to such an analysis.

A Generic Inductive Data Type for Terms A better approach, the object of the work on the project at the time of this writing, is to work with a single definition of term that can be *parameterized* by signatures. Indeed this is the approach taken by Coccinelle and CoLoR.

Given a definition of Signature (a set of function symbols with arities) and a declaration of Sig, a variable of type Signature, the following (taken from the CoLoR distribution) is the definition of term:

```
Inductive term : Type :=
| Var : variable → term
| Fun : forall f : Sig, vector term (arity f) → term .
```

A term is given a function symbol f from Sig and a vector of terms whose length is the arity of f. In fact there is a design choice reflected here: Coccinelle, for example, uses a *list* of terms to hold the arguments to f. Lists have better library support than do vectors in Coq, but we then incur either the stain of accepting that ill-formed terms can be built or the burden of maintaining and verifying well-formedness after each operation, reminiscent of the polynomial-form discussion above.

One small cost to this choice is that, because term occurs recursively inside the vector data structure, the induction principle automatically generated by Coq for these terms is not as strong as it could be. (This drawback applies to the list-based approach as well.) This is a remark about the current strength of Coq's algorithm for constructing induction principles, not about the existence of a useful induction principle. In fact it is straightforward to generate by hand the proper induction principles and tell Coq to use these: this is exactly what Coccinelle and CoLoR do. A slightly greater cost to this approach is

simply that dependent types such as vectors can be awkward to work with and reason about compared to direct inductive types.

6 Future Work

The current development is rather naive, from a Coq perspective: there are doubtless many unrealized opportunities for proof automation (including an adaptation of Coq's built-in ring tactic to incorporate Boolean-ring specific simplifications). The most immediate future, indeed ongoing, work is to address the challenge described in Section 5 of finding the best data structure for terms. This will be crucial to maintaining a flexible set of libraries that admit free function symbols and work across a variety of equational theories. Coq has facilities for extraction of functional programs; we expect this to be straightforward to incorporate into our libraries. Of course the most significant goal for the future is to treat other equational theories. Any number of specific theories E are natural candidates but it will perhaps be even more interesting to formalize "generic" E-unification approaches such as those based on transformations, combining algorithms, and narrowing (see [Mes18] for a discussion)

Acknowledgments It was very helpful to do this work in parallel with the work of the WPI undergraduate project team of Spyridon Antonatos, Matthew McDonald, Dylan Richardson, and Joseph St. Pierre. Although the development described here differs in many ways from theirs, their project led to many useful discussions about design choices for implementation and verification. It also generated a useful succession of deadlines to be met. The extremely clear introduction to Boolean unification in chapter 4 of [BN98] was a crucial element in making this topic suitable for introducing Coq to an undergraduate project team.

We benefited greatly from utility code in the library "Base Library for ICL" written by Gert Smolka. https://www.ps.uni-saarland.de/courses/cl-ss15/coq/ICL.Base.html

References

[Baa98] Franz Baader. On the complexity of boolean unification. *Information Processing Letters*, 67(4):215–220, 1998.

[BK11] Frédéric Blanqui and Adam Koprowski. Color: a coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 21(4):827–859, 2011.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[Boo47] George Boole. *The mathematical analysis of logic*. Philosophical Library, 1847.

[BS87] Wolfram Buttner and Helmut Simonis. Embedding boolean expressions into logic programming. *Journal of Symbolic Computation*, 4(2):191–205, 1987.

[BS01] Franz Baader and Wayne Snyder. Unification theory. *Handbook of automated reasoning*, 1:445–532, 2001.

[BSV18] Kasper Fabæch Brandt, Anders Schlichtkrull, and Jørgen Villadsen. Formalization of first-order syntactic unification. In *32nd International Workshop on Unification (UNIF), Informal Proceedings*, 2018.

[Cap99] V. Capretta. Universal algebra in type theory. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *TPHOLs 1999*, volume 1690 of *LNCS*, pages 131–148. Springer, 1999.

[CCF+11] Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Automated certified proofs with CiME3. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, volume 10 of *LIPIcs*, pages 21–30. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[Con] Evelyne Contejean. The Coccinelle library for rewriting. http://www.lri.fr/~contejea/Coccinelle/coccinelle.html.

[Con04] Evelyne Contejean. A certified AC matching algorithm. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2004.

[Dom08] C. Domınguez. Formalizing in Coq Hidden Algebras to Specify Symbolic Computation Systems. In *AISC*, pages 270–284. Springer, 2008.

[Löw08] Leopold Löwenheim. Über das Auflösungsproblem im logischen Klassenkalkul. *Sitzungsberichte Berliner Math. Gesell.*, 7:89–94, 1908.

[Mes18] José Meseguer. Variant-based satisfiability in initial algebras. *Science of Computer Programming*, 154:3–41, 2018.

[MN89] Urusula Martin and Tobias Nipkow. Boolean unification—the story so far. *Journal of Symbolic Computation*, 7(3-4):275–293, 1989.

[Sto36] Marshall H Stone. The theory of representation for boolean algebras. *Transactions of the American Mathematical Society*, 40(1):37–111, 1936.

Nominal Unification with Letrec and Environment-Variables*

Manfred Schmidt-Schauß[1] and Yunus Kutz[2]
Goethe-University, Frankfurt am Main, Germany

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

Abstract

Abstract. We extend nominal unification to higher-order expressions including letrec such that besides expression-variables, also atom- and environment-variables are permitted. In unification problems, the occurrences of environment- and expression-variables are restricted such that they occur at most once in the equations. A terminating and complete unification algorithm is described that computes at most an exponential number of unifiers of polynomial size. A restricted variant of the unification algorithm is provided as decision algorithm. The complexity of the problem is determined to be NP-complete.

Keywords: nominal unification, letrec-expressions, abstract environments

1 Introduction

The goal of this paper is to extend the expressive power of nominal unification such that it can be used in reasoning algorithms in call-by-need functional programming languages that employ letrec-environments as for example Haskell [5, 3]. Nominal techniques [7, 6] support machine-oriented reasoning on the syntactic level for higher-order languages and support alpha-equivalence. An algorithm for nominal unification was first described in [15], which outputs unique most general representations. More efficient algorithms are given in [1, 4], also exhibiting a quadratic algorithm. The approach is also used in higher-order logic programming [2] and in automated theorem provers like nominal Isabelle [13, 14]. Nominal unification was generalized to permit also atom-variables [11] where also in the generalization, unique most general representations are computed, while the decision problem is NP-complete.

An extension of nominal unification to languages with a recursive let was worked out in [9], however, without atom-variables, and where it was shown that the nominal unification and matching problems are NP-complete. The nominal unification algorithm for letrec was extended to atom-variables in [10]. Also, a nominal matching algorithm for letrec with environment variables, but without atom-variables is proposed in [10], however, how to extend or adapt the matching algorithm with environment-variables to nominal unification was left open.

A simple example to motivate the use of environment-variables is the rule (cp) in the calculus LR [12] $(\texttt{letr}\ x = \lambda z.S, Env\ \texttt{in}\ (x\ \lambda y.y)) \to (\texttt{letr}\ x = \lambda z.S, Env\ \texttt{in}\ ((\lambda z.S)\ \lambda y.y))$, which has an abstract environment Env. Nominal unification of an overlap with a transformation like (lbeta) would require that environment-variables are permitted in the nominal unification algorithm. In this rule the occurrence of atom-variables is not linear, which is not a problem for the presented nominal unification algorithm. Another example is a variant of a rule for common subexpression elimination, which could be written as $(\texttt{letr}\ x = S, y = S, Env\ \texttt{in}\ S') \to (\texttt{letr}\ x = S, y = x, Env\ \texttt{in}\ S')$, and which requires the same expression-variable twice in the

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

left hand side. An equation between the left hand side and another expression with linear occurrences could be the input of our nominal unification algorithm.

The results in this paper are a nominal unification algorithm for a higher-order calculus with letrec and atom-, expression- and environment-variables where expression- and environment-variables occur linearly in the set of equations (Theorem 3.5). The complexity of the problem is shown to be NP-complete (see Theorem 4.7).

2 Nominal Terms

We first introduce some notation [11]. Let $\mathcal{F}$ be a set of function symbols $f \in \mathcal{F}$, s.t. each f has a fixed arity $ar(f) \geqslant 0$. Let $\mathcal{A}t$ be *the set of atoms* ranged over by a, b, c. The ground language NL_a is defined by the grammar:

$e \quad ::= \quad a \mid (f\ e_1 \dots e_{ar(f)}) \mid \lambda a.e \mid \texttt{letr}\ a_1.e_1, \dots, a_n.e_n\ \texttt{in}\ e$

where λ is a binder for atoms, and $\texttt{letr} \dots \texttt{in} \dots$ is the recursive let, where the atoms a_i in the letrec-environment must be pairwise different.

The basic constraint $a\#e$ is valid if a is not free in e and a set of constraints ∇ is valid if all constraints are valid. $LV(Env)$ is the multiset of binders in a letrec-environment Env.

As a reminder, the α-equivalence relation $\sim$ on NL_a is defined as the equivalence closure of iterated (capture-free) renaming of atoms. For a better treatment of α-equivalence with letrec, we show a decomposition principle for letrec-expression modulo α:

Lemma 2.1. *Let $e_1 = \texttt{letr}\ a_1.s_1, \dots, a_n.s_n\ \texttt{in}\ r$ and $e_2 = \texttt{letr}\ b_1.t_1, \dots, b_n.t_n\ \texttt{in}\ r'$ be NL_a-expressions. Then $e_1 \sim e_2$ is equivalent to the following conditions:*

1. *φ is a permutation on atoms, such that $dom(\varphi) \subseteq \{a_1, \dots, a_n\} \cup \{b_1, \dots, b_n\}$, and it extends the mapping $\{b_i \mapsto a_{\rho(i)} \mid i = 1, \dots, n\}$, where ρ is a permutation on the set $\{1, \dots, n\}$.*
2. *For $M := \{a_1, \dots, a_n\} \backslash \{b_1, \dots, b_n\}$, we have $\varphi(M) = \{b_1, \dots, b_n\} \backslash \{a_1, \dots, a_n\}$.*
3. *$M\#(\texttt{letr}\ b_1.t_1, \dots, b_n.t_n\ \texttt{in}\ r')$.*
4. *$r \sim \varphi(r')$ and $s_i \sim \varphi(t_{\rho(i)})$ for $i = 1, \dots, m$ hold.*

This leads to the following decomposition principle for all constructs w.r.t $\sim$, which can also be seen as a definition of α-equivalence on NL_a:

Lemma 2.2. *Syntactic α-equivalence $\sim$ in NL_a is characterized by the following rules:*

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

$$\frac{\textit{The conditions of Lemma 2.1 hold.}}{\texttt{letr}\ a_1.s_1, \dots, a_n.s_n\ \texttt{in}\ r \sim \texttt{letr}\ b_1.t_1, \dots, b_n.t_n\ \texttt{in}\ r'}$$

Definition 2.3 (Extensions of NL_a.)**.** *Let $\mathcal{S}$ be a set of expression-variables ranged over by S, T; let $\mathcal{A}$ be the set of atom-variables ranged over by A, B, let $\mathcal{E}$ be a set of variables standing for letrec-environments ranged over by E, and let $\mathcal{P}$ be a set of permutation-variables ranged over*

by P. The grammar of the nominal language NL_{aASPE} with atoms, atom-variables, expression-variables, permutation-variables and environment-variables is:

$$\begin{array}{lcl} e & ::= & W \mid \pi{\cdot}S \mid (f\ e_1 \ldots e_{ar(f)}) \mid \lambda W.e \mid \texttt{letr}\ env\ \texttt{in}\ e \\ \pi & ::= & \varnothing \mid (W\ W') \cdot \pi' \mid P \cdot \pi' \\ env & ::= & \varnothing \mid \pi{\cdot}env \mid (\pi{\cdot}a.e; env) \mid (\pi{\cdot}A.e; env) \mid (\pi{\cdot}E; env) \\ W & ::= & \pi{\cdot}a \mid \pi{\cdot}A \end{array}$$

where π is a permutation and $\varnothing$ denotes the identity.

We will also use sublanguages like $NL_{AS} \subset NL_{aASPE}$ and $NL_{ASE} \subset NL_{aASPE}$, where only the components mentioned in the index are used in the grammar.

Note that we permit nested permutation expressions. The expression $((\pi{\cdot}A)\ (\pi'{\cdot}A'))$ is a single nested swapping. The inverse π^{-1} of a permutation $\pi = sw_1{\cdot}\ldots{\cdot}sw_n$ with swappings sw_i is the expression $sw_n{\cdot}\ldots{\cdot}sw_1$.
$AtVar(e)$ are the atom-variables contained in e, $ExVar(e)$ the expression-variables contained in e and $Var(e) = AtVar(e) \cup ExVar(e)$.

The language of interest in this paper is NL_{ASPE}. The ground language of NL_{ASPE} is NL_a, i.e. a ground substitution replaces atom-variables by atoms, expression-variables by ground expressions; permutation variables by permutations, and environment-variables by ground environments. The language NL_{aASPE} serves as an intermediate language during the interpretation of NL_{ASPE} expressions.

We will use further constraint primitives and constraint expressions in sets of constraints, like $LV(env)$, $dom(\pi)$, and $\#LV(env))$, where the latter means that an instance is only valid, if all atom expressions as binders in a let-binding in env are different.

3 Nominal Unification with Environments

In this section we construct a unification algorithm for expressions of NL_{ASE} where E-variables and S-variables occur linearly.

As data structure we use a set Γ of (symmetric) equations between expressions, general constraints ∇, and a substitution θ.

Definition 3.1 (Constraints)**.** *A freshness constraint has the form $A\#e$. A general constraint is either a freshness constraint or is of the form:*

$$\begin{array}{lcc} \#env \mid & dom \subseteq LV(env_1) \cup LV(env_2) & \mid \quad P \cdot LV(env_2) = LV(env_1) \\ & P{\cdot}(LV(env_1) \backslash LV(ienv_2)) = (LV(ienv_2) \backslash LV(ienv_1)) & \mid \quad LV(ienv_1) \backslash LV(ienv_2) \# e \end{array}$$

A solution of set of constraints is a ground substitution γ, s.t. all constraints hold in NL_a.

For technical reasons we restrict the scope of unification problems. This may not be necessary, especially the linearity of expression-variables looks like it could be relaxed. However, we do not have complete proofs for general problems at the moment.

Definition 3.2. *A set of equations Γ over NL_{ASPE} is* admissible*, if environment-variables and expression-variables occur at most once in Γ and it does not contain permutation-variables.*

Definition 3.3. *[Decomposing letrec.] Let $(\texttt{letr}\ env_1\ \texttt{in}\ e_1) \doteq (\texttt{letr}\ env_2\ \texttt{in}\ e_2)$ be the equation to be decomposed, where env_j for $j = 1, 2$ consists of a list of bindings $b_{j,i}$ and environment-variables $E_{j,i}$. The decomposition is non-deterministic and proceeds as follows:*
There is a single guess of a relation R consisting of pairs (k_1, k_2) where k_j is a component of env_j for $j = 1, 2$, such that

$$\text{(E1)}\ \frac{(\Gamma \cup \{e \doteq e\}, \nabla, \theta)}{(\Gamma, \nabla, \theta)} \qquad \text{(E2)}\ \frac{(\Gamma \cup \{\pi{\cdot}S \doteq e\}, \nabla, \theta)}{(\Gamma[S \mapsto \pi^{-1}{\cdot}e], \nabla[S \mapsto \pi^{-1}{\cdot}e], \theta \cup \{S \mapsto \pi^{-1}{\cdot}e\})}$$

$$\text{(E3)}\ \frac{(\Gamma \cup \{\pi_1{\cdot}A \doteq \pi_2{\cdot}B\}, \nabla, \theta)}{(\Gamma, \nabla \cup \{A =_{\#} \pi_1^{-1}{\cdot}\pi_2{\cdot}B\}, \theta)} \qquad \text{(E4)}\ \frac{(\Gamma \cup \{(f\ e_1 \dots e_{ar(f)}) \doteq (f\ e'_1 \dots e'_{ar(f)})\}, \nabla, \theta)}{(\Gamma \cup \{e_1 \doteq e'_1, \dots, e_{ar(f)} \doteq e'_{ar(f)}\}, \nabla, \theta)}$$

$$\text{(E5)}\ \frac{(\Gamma \cup \{(W_1.e_1 \doteq W_2.e_2\}, \nabla, \theta)}{(\Gamma \cup \{W_1 \doteq W_2, e_1 \doteq e_2\}, \nabla, \theta)}$$

$$\text{(E6)}\ \frac{(\Gamma \cup \{\lambda\pi_1{\cdot}A_1.e_1 \doteq \lambda\pi_2{\cdot}A_2.e_2\}, \nabla, \theta)}{(\Gamma \cup \{((\pi_1{\cdot}A_1)\ (\pi_2{\cdot}A_2)){\cdot}e_1 \doteq e_2\}, \nabla \cup \{(A_1 \# \pi_1^{-1}{\cdot}(\lambda\pi_2{\cdot}A_2.e_2))\}, \theta)}$$

$$\text{(E7)}\ \frac{(\Gamma \cup \{\texttt{letr}\ env_1\ \texttt{in}\ e_1 \doteq \texttt{letr}\ env_2\ \texttt{in}\ e_2\}, \nabla, \theta)}{(\Gamma \cup \Gamma_{res}, \nabla \cup \nabla_{res}, \theta \circ \theta_{res})}\ \text{guess according to Def. 3.3}$$

Figure 1: Rules of NomEnv1

1. *Every binding $b_{1,j}$ is related to exactly one component of the right hand side.*
2. *Every binding $b_{2,j}$ is related to exactly one component of the left hand side.*

Let P be a fresh permutation-variable. The resulting equations Γ_{res} consist of $e_1 \doteq P \cdot e_2$ and equations $K \doteq P{\cdot}K'$ for single-binding components K, K', if these are related by R .
The resulting substitution components θ_{res} are as follows:
Create fresh environment-variables $E_{i,j,k,h}$ which (roughly) represent the intersection of $E_{i,j}$ and $E_{k,h}$. There are substitution components for every environment-variable on the left hand side:
$E_{1,j} \mapsto P{\cdot}(E_{1,j,2,1}, \dots E_{1,j,2,m}, B_{2,j})$ for an appropriate m and where $B_{2,j}$ are the single bindings on the right hand side that are related by R to $E_{1,j}$.
There are substitution components for every environment-variable on the right hand side:
$E_{2,j} \mapsto P^{-1}{\cdot}(E_{1,1,2,j}, \dots E_{1,m',2,j}, B_{1,j})$ for an appropriate m' and where $B_{1,j}$ are the single bindings on the left hand side that are related by R to $E_{2,j}$.

The resulting constraints ∇_{res} are:

1. *$dom(P) \subseteq LV(env_1) \cup LV(env_2)$.*
2. *$P{\cdot}(LV(env_2)) = LV(env_1)$*
3. *$(LV(env_1) \backslash LV(env_2)) \# \texttt{letr}\ env_2\ \texttt{in}\ e_2$*
4. *$P{\cdot}(LV(env_1) \backslash LV(env_2)) = (LV(env_2) \backslash LV(env_1))$.*

The effects of applying the rule is to remove the equation, and to add Γ_{res}; ∇_{res} and θ_{res}.

Definition 3.4. *The algorithm* NomEnv1 *is defined by the rules in Fig. 1 on Γ, ∇, where the input Γ_0, ∇_0 is admissible.*

The set ∇ may contain constraints. Since binders in a letrec must be different, ∇ must contain constraints which ensure for every environment env that all variables bound by the environment are different, i.e. $\#LV(env)$.

$$(\texttt{RemoveEl})\ \frac{(\Gamma \cup \left\{ \begin{array}{l} \texttt{letr}\ W_1.s_1;\ldots;W_k.s_k;E_1\ \texttt{in}\ s \\ \doteq \texttt{letr}\ W'_1.s'_1;\ldots;W'_{k'}.s'_{k'}\ \texttt{in}\ s' \end{array} \right\}, \nabla, \theta)}{(\Gamma \cup \left\{ \begin{array}{l} \texttt{letr}\ W_1.s_1;\ldots;W_k.s_k;A_1.S_1;\ldots;A_{k'-k}.S_{k'-k}\ \texttt{in}\ s \\ \doteq \texttt{letr}\ W'_1.s'_1;\ldots;W'_{k'}.s'_{k'}\ \texttt{in}\ s' \end{array} \right\}, \nabla\theta_E, \theta \circ \theta_E)}$$
where $\theta_E = \{E_1 \mapsto A_1.S_1;\ldots;A_{k'-k}.S_{k'-k},\}$. A_i, S_i are fresh.

$$(\texttt{RemoveEr})\ \frac{(\Gamma \cup \left\{ \begin{array}{l} \texttt{letr}\ W_1.s_1;\ldots;W_k.s_k\ \texttt{in}\ s \\ \doteq \texttt{letr}\ W'_1.s'_1;\ldots;W'_{k'}.s'_{k'};E'_1\ \texttt{in}\ s' \end{array} \right\}, \nabla, \theta)}{(\Gamma \cup \left\{ \begin{array}{l} \texttt{letr}\ W_1.s_1;\ldots;W_k.s_k\ \texttt{in}\ s \\ \doteq \texttt{letr}\ W'_1.s'_1;\ldots;W'_{k'}.s'_{k'};A_1.S_1;\ldots;A_{k-k'}.S_{k-k'}\ \texttt{in}\ s' \end{array} \right\}, \nabla\theta_E, \theta \circ \theta_E)}$$
where $\theta_E = \{E'_1 \mapsto A'_1.S'_1;\ldots;A'_{k-k'}.S'_{k-k'}\}$. A_i, S_i are fresh.

$$(\texttt{RemoveElr})\ \frac{(\Gamma \cup \left\{ \begin{array}{l} \texttt{letr}\ W_1.s_1;\ldots;W_k.s_k;E_1\ \texttt{in}\ s \\ \doteq \texttt{letr}\ W'_1.s'_1;\ldots;W'_{k'}.s'_{k'};E'_1\ \texttt{in}\ s' \end{array} \right\}, \nabla, \theta)}{(\Gamma \cup \left\{ \begin{array}{l} \texttt{letr}\ W_1.s_1;\ldots;W_k.s_k;A_1.S_1;\ldots;A_h.S_h\ \texttt{in}\ s \\ \doteq \texttt{letr}\ W'_1.s'_1;\ldots;W'_{k'}.s'_{k'};A'_1.S'_1;\ldots;A'_h.S'_{h'}\ \texttt{in}\ s' \end{array} \right\}, \nabla\theta_E, \theta \circ \theta_E)}$$
where $\theta_E = \{E_1 \mapsto A_1.S_1;\ldots;A_h.S_h, E'_1 \mapsto A'_1.S'_1;\ldots;A'_{h'}.S'_{h'}\}$
and A_i, S_i, A'_i, S'_i are fresh
and $k' + h' = k + h = k + k' + N_2$; where $N_2 = |AtPos(s_1,\ldots,s_k,s,s'_1,\ldots,s'_{k'},s')|$

Rule `RemoveE` is defined as the union of `RemoveEl`, `RemoveEr` and `RemoveElr`.

$$(\texttt{DecompLet})\ \frac{(\Gamma \cup \{\texttt{letr}\ W_1.e_1,\ldots,W_n.e_n\ \texttt{in}\ r \doteq \texttt{letr}\ \ W'_1.e'_1,\ldots,W'_n.e'_n\ \texttt{in}\ r'\}, \nabla, \theta)}{(\Gamma \cup \{e_1 \doteq \xi{\cdot}e'_{\rho(1)},\ldots,e_n \doteq \xi{\cdot}e'_{\rho(n)}, r \doteq \xi{\cdot}r'\}, \nabla \cup \nabla', \theta)}$$
ρ is a (guessed) permutation on $\{1,\ldots,n\}$, $\xi := \xi(W'_{\rho(1)}, W_1,\ldots,W'_{\rho(k)}, W_k)$
$\nabla' := (\{W_i \mid i = 1,\ldots n\} \backslash \{W'_i \mid i = 1,\ldots n\}) \# (\texttt{letr}\ \ W'_1.e'_1,\ldots,W'_n.e'_n\ \texttt{in}\ r')$.

Figure 2: Extra Rules for NomEnv1D

Theorem 3.5. *The algorithm* NomEnv1 *is terminating and complete for admissible unification problems, i.e. for every solution the algorithm computes a unifier consisting only of a substitution θ and general constraints ∇. A single run takes polynomial time. The collection version of the algorithm will generate at most exponentially many unifiers.*

However, algorithm NomEnv1 does not decide solvability, since we do not know whether the constraints forbid all instances of the resulting substitution. To show decidability and to argue on the complexity, we will show that the algorithm NomEnv1D will find a (small) solution if there is any solution.

4 A Decision Algorithm

In this section we describe a decision algorithm for the nominal unification algorithm with letrec and environment-variables. We want to keep the description simple and also as close to potential applications as possible. Thus we describe the decision algorithm for the case that in letrec environments at most one environment-variable occurs. The advantage is that this is the variant that is required in most applications, and in addition the rule `RemoveE` is deterministic. Later we describe the necessary extensions and additional non-determinism for the general case.

Permutation-variables introduced by `DecompLet` can be so strongly restricted, that their instantiation can be completely determined from the other instantiations. Hence we introduce an extra notation to avoid permutation-variables. The benefit of this variant is that no new constraint concepts are required and thus is compatible with [11].

Definition 4.1. *Let Γ be an admissible set of equations. We say Γ is a* 1E*-problem, if in every letrec-environment in Γ, there is at most one environment-variable.*

Definition 4.2. *In order to denote permutations that come from mappings, we denote with $\xi(W_1, W_2, W_3, W_4 \ldots, W_{2k-1}, W_{2k})$ a permutation that obeys the following: $W_1 \mapsto W_2$, $W_3 \mapsto W_4$, $\ldots$, $W_{2k-1} \mapsto W_{2k}$ and the domain is contained in $\{W_1, \ldots, W_{2k}\}$. This is not a unique definition of the permutation, but the omitted parts will have no effect, when it is applied, due to freshness constraints. Furthermore, a representation as a list of swappings can be constructed from $W_1, \ldots, W_{2k}$ by using a recursion scheme.*

Definition 4.3. *The function $AtPos(e)$ returns all positions of atom suspensions (W-variables) in e, where also the arguments of $\xi(W_1, W_2, \ldots, W_{2k-1}, W_{2k})$ count as positions.*

Definition 4.4. *The algorithm* NomEnv1D *is defined for* 1*E-problem and uses the rules in Fig. 1 on the input Γ_0, ∇_0, with the following exception: The rule E7 is replaced by the rules* `RemoveE` *and* `DecompLet` *in Fig. 2.*
Let $(\Gamma', \nabla', \theta')$ be the output of the algorithm. It succeeds if $\Gamma' = \varnothing$ and $\nabla'\theta'$ is satisfiable using a constraint-test of freshness-constraints, e.g. the test defined in [10].

Note that the rules can be applied in any order. All rules are deterministic with the exception of `DecompLet`, which requires a guess on the permutation of the let-variables.

Note also that the substitution θ is intended to be an instantiation of the input problem. The necessary instantiations of the current Γ, ∇ are done by the rules. However, it is necessary to assume a directed graph implementation of expressions in order to exploit sharing, in particular in the representation of permutations.
Note that the algorithm NomEnv1D is probably not complete, since by intention, its rules do not cover all solutions. However, we will show that it is sufficient for a decision algorithm, since the algorithm will find a (small) solution if there is one at all.

Lemma 4.5. *Given an* 1*E-problem Γ, ∇ as input or during the run of the Algorithm* NomEnv1D*, then rule* `RemoveE` *of* NomEnv1D *is correct, complete and deterministic, i.e. Γ', ∇' is solvable if and only if the input Γ, ∇ is solvable.*

Lemma 4.6. *All rules of* NomEnv1D *are correct and complete, where only* `DecompLet` *is non-deterministic.*

Theorem 4.7. *Given a admissible nominal unification problem Γ, ∇ without permutation-variables such that Γ is* 1*E. Then the Algorithm* NomEnv1D *is a decision algorithm, which runs in NP time, under the assumption that sharing is used in the representation of expressions and permutations.*

Remark 4.8 (Decision Algorithm for the General Case)**.** *We concentrate on the case that multiple environment-variables $E_1, \ldots, E_n$ are in the same letrec environment.*
The generalization is that the rule(s) `RemoveE` *has to try all possibilities of distributing the single bindings into the environment-variables. This makes the generalized rule(s)* `RemoveE` *non-deterministic. It is obvious how to transfer the correctness proof to this case.*

This nondeterminism could be omitted if the environment-variables are not restricted by freshness constraints, or if the freshness constraints contain them such that the environment-variables can ne interchanged. This determinism can be achieved by selecting a single environment-variable per environment, say E_1, and put all instantiation into E_1, end instantiate other environment-variables by $\varnothing$.

5 Examples

As a simple example to illustrate the algorithms presented, consider the following unification problem:

Example 5.1. *The algorithm* NomEnv1 *runs on* $(\{A\#B\}, \{\texttt{letr}\ E_1\ \texttt{in}\ A \doteq \texttt{letr}\ E_2\ \texttt{in}\ B\})$ *as follows:*

At first, as the only possible result of rule E7, the substitution $\theta_{res} = \{E_1 \to P \cdot E_2\}$ *is computed . The resulting equation is* $A \doteq P \cdot B$ *and the resulting constraints are:*

$$\nabla_{res} = \left\{ \begin{array}{c} dom(P) \subseteq LV(E_1) \cup LV(E_2), \\ P{\cdot}LV(E_2) = LV(E_1), \\ LV(E_1)\backslash LV(E_2)\#\texttt{letr}\ E_1\ \texttt{in}\ B, \\ P{\cdot}(LV(E_1)\backslash LV(E_2)) = (LV(E_2)\backslash LV(E_1)) \end{array} \right\}$$

The final result of the algorithm and most-general unifier of the problem is then:
$(\nabla_{res} \cup \{A\#B, A =_{\#} P \cdot B\}, \{E_1 \to P \cdot E_2\})$.

Until the are decision algorithms for these more complex constraints, solvability can be checked using NomEnv1D*, which would run as follows:*

There are 2 atom position in subexpressions of $\texttt{letr}\ E_1\ \texttt{in}\ A \doteq \texttt{letr}\ E_2\ \texttt{in}\ B$*. Therefore, the rule* `RemoveElr` *applies the substitution* $\theta_E = \{E_1 \mapsto \{C_1.S_1; C_2.S_2\}, E_2 \mapsto \{D_1.T_1; D_2.T_2\}\}$ *to the unification problem, resulting in* $\texttt{letr}\ C_1.S_1; C_2.S_2\ \texttt{in}\ A \doteq \texttt{letr}\ D_1.T_1; D_2.T_2\ \texttt{in}\ B$ *and* $\{C_1\#C_2, D_1\#D_2\}$.

The algorithm then guesses the identity as a first possible relation between bindings, resulting in:

$$\{S_1 \doteq \xi \cdot T_1, S_2 \doteq \xi \cdot T_2, A \doteq \xi \cdot B\}$$

with $\xi = \xi(C_1, D_1, C_2, D_2)$ *where* $\xi = (C_1\ ((C_1\ D_2) \cdot D_1)) \cdot (C_2\ D_2)$ *is a possible representation. The unifier (of this first run) is:*

$$(\{A\#B, C_1\#C_2, D_1\#D_2, A =_{\#} \xi \cdot B\}, \theta_E \circ \{S_1 \mapsto \xi \cdot T_1, S_2 \mapsto \xi \cdot T_2\})$$

which has a solvable constraint set. Thus, the probem is solvable.

6 Conclusion

Future work is to extend the algorithms NomEnv1 and NomEnv1D such that also non-linear input, in particular several occurrences of the same expression-variable can be in the input. Also extending this to context variables as in [8] would extend the expressibility.

Potential applications of nominal unification are overlapping the rules of functional call-by-need calculi with transformations, since several of these rules and transformation have abstract variables for parts of the environment, and where usually the occurrences of the environment-variables are linear.

References

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

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

[3] H. Community. Haskell, an advanced, purely functional programming language, 2019.

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

[5] S. Marlow, editor. *Haskell 2010 – Language Report.* 2010.

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

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

[8] M. Schmidt-Schauß and D. S. and. Nominal unification with atom and context variables. In H. Kirchner, editor, *Proc. 3rd FSCD 2018)*, volume 108 of *LIPIcs*, pages 28:1–28:20. Schloss Dagstuhl, 2018.

[9] M. Schmidt-Schauß, T. Kutsia, J. Levy, and M. Villaret. Nominal unification of higher order expressions with recursive let. In M. V. Hermenegildo and P. 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.

[10] M. Schmidt-Schauß, T. Kutsia, J. Levy, M. Villaret, and Y. Kutz. Nominal unification of higher order expressions with recursive let. *Fundamenta Informaticae*, 2019. submitted.

[11] M. Schmidt-Schauß, D. Sabel, and Y. Kutz. Nominal unification with atom-variables. *J. Symb. Comput.*, pages 42–64, 2019.

[12] M. Schmidt-Schauß, M. Schütz, and D. Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.

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

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

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

Parameters for Associative and Commutative Matching*

Luis Henrique Bustamante, Ana Teresa Martins, and Francicleber Martins Ferreira

Department of Computing, Federal University of Ceará, Fortaleza, Ceará, Brazil
lhbusta@lia.ufc.br; ana@dc.ufc.br; francicleber@dc.ufc.br

Abstract

We apply parameterized complexity theory to classify the associative, commutative, and associative-commutative matching problems ({A, C, AC}-MATCHING) with respect to different parameters. We primarily consider the number of variables, the size of the substitution, and the size of the vocabulary as parameters. We establish, for a combination of the size of the substitution, and the size of the vocabulary, that these matching problems are fixed-parameter tractable. For the other cases, we show membership in W[P] for C-MATCHING when considering the number of variables, and for {A, AC}-MATCHING, with respect to the size of the substitution.

1 Introduction

The NP-completeness of the term matching problem for associative (A), commutative (C), and associative-commutative (AC) terms of first-order logic are well-known results [5]. Here we consider the decision version of the matching problem that asks, given a term s and a ground term t using a set of function symbols $\mathcal{F}$, if there exists a substitution θ such that $s\theta =_E t$ modulo an equational theory $E \in \{\text{A, C, AC}\}$. The aim of this paper is to analyze the parameterized complexity of the matching problem with respect to the number of variables $|\text{var}(s)|$, the size of substitution $|\theta|$, and the size of the vocabulary $|\mathcal{F}|$.

We evoke parameterized complexity theory [4] as a framework able to distinguish the fine-grained complexity of these matching problems with respect to different parameters and, in some sense, to detect the source of their hidden complexity. In this theory, the measure of complexity is not restricted to the input size $|x|$, but it is also expressed in terms of an additional parameter k. A central notion of parameterized complexity theory is the relaxed idea of tractability, *fixed-parameter tractability* (fpt), which allows an algorithm that runs in time $f(k) \cdot |x|^{O(1)}$ for some arbitrarily computable function f. The class FPT is the class of problems decidable in "fpt-time". The parameterized intractability is described by a diversified collection of classes, and it is best represented by the classes W[1] and W[2] (see [4] for a precise definition), the lower level of the W-Hierarchy. On top of this, we have the class W[P] which is the class of problems decidable by a non-deterministic algorithm in FPT but with at most $h(k) \cdot \log |x|$ non-deterministic steps for some arbitrary function h.

In [1], the parameterized complexity of {A, C, AC}-unification/matching was studied with respect to $|\text{var}(s)|$. They obtained that $|\text{var}(s)|$-{A, AC}-MATCHING are W[1]-hard, and they conjecture that $|\text{var}(s)|$-C-MATCHING is in FPT via a dynamic Programming algorithm assuming an additional hypothesis. In Section 3, we give an algorithm in W[P] for C-MATCHING when parameterized by $|\text{var}(s)|$. Although, for {A, AC}-MATCHING, we would like to answer

*This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, and by the Brazilian National Council for Scientific and Technological Development (CNPq) under the grant number 424188/2016-3.

if these problems are within W[1] concluding their W[1]-completeness, we could only show the W[P] membership with $|\theta|$ as the parameter. The relevance to locate a problem within W[1] is related the possibility of algorithms with running time faster than exhaustive search over all $\binom{n}{k}$ subsets. For example, k-CLIQUE, a W[1]-complete problem, has an algorithm that runs in time $O(n^{(\omega/3)k})$ [7], achieved with the use of an $n \times n$ matrix multiplication algorithm with running time in $O(n^{\omega})$ (the best known value for ω is 2.3728639 [6]). For k-DOMINATING-SET, a W[2]-complete problem, we cannot do anything better than an algorithm running in $O(n^{1+k})$ unless CNF satisfiability has a $2^{\delta n}$ time algorithm for some $\delta < 1$ [8].

It seems that the size of the substitution $|\theta| = k + \sum_{i=1}^{k} |t_i|$, for a finite substitution $\{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ (see Section 2), represents a more natural parameter since it represents the size of the solution. In this case, we may also provide membership in W[P] for $|\theta|$-{A, AC}-MATCHING. In Section 4, we observe that with $|\mathcal{F}| + |\theta|$ as the parameter we can construct a fixed-parameter tractable brute-force algorithm. This idea is applied in [3] to the string morphism problem, with respect to different parameters. In the last section, we summarize the results and point out the open problems.

2 Preliminaries

For a complete picture of parameterized complexity theory, we refer to the textbook [4]. We adopt the notation in [2, 5] for unification theory.

2.1 Parameterized complexity

A *parameterized problem* is a pair (Q, κ) over the alphabet Σ, such that $Q \subseteq \Sigma^*$ is a decision problem and κ is a polynomial time computable function from Σ^* to natural numbers $\mathbb{N}$, called the *parameterization*. For an *instance* $x \in \Sigma^*$ of Q or (Q, κ), $\kappa(x) = k$ is the *parameter* of x. A *slice* of a parameterized problem (Q, κ) is the decision problem $(Q, \kappa)_\ell := \{x \in Q \mid \kappa(x) = \ell\}$.

We say that a problem (Q, κ) is *fixed-parameter tractable* if there is an algorithm that decides $x \in Q$ in time $f(\kappa(x)) \cdot |x|^{O(1)}$ for some computable function f. The class of all fixed-parameter tractable problems is called FPT. We can extend the notion of polynomial time reductions to fpt-reductions and, in the same way, we can handle the notions of hardness and completeness.

A parameterized problem (Q, κ) is in W[P] if there exists a non-deterministic Turing machine $\mathbb{M}$ with input alphabet Σ that decides Q in at most $f(\kappa(x)) \cdot |x|^{O(1)}$ steps for input x and at most $h(\kappa(x)) \cdot \log |x|$ non-deterministic steps for some computable functions $f, h : \mathbb{N} \to \mathbb{N}$. Let (Q, κ) be a parameterized problem. Then, (Q, κ) is in *paraNP*, if there is a non-deterministic algorithm that decides if $x \in Q$ in at most $f(\kappa(x)) \cdot |x|^{O(1)}$ steps, such that f is a computable function. The class *XP* is the parameterized version of the exponential time class. A parameterized problem (Q, κ) is in XP, if there is an algorithm that decides if $x \in Q$ in at most $f(\kappa(x)) \cdot |x|^{g(\kappa(x))}$ steps, for some computable functions $f, g : \mathbb{N} \to \mathbb{N}$.

2.2 The {A, C, AC}-MATCHING problems

Let $\mathcal{F}$ be a countable set of function symbols with some arity, and $\mathcal{V}$ a countable set of variables. A *term* t is inductively defined from variables in $\mathcal{V}$ closed under functions $f \in \mathcal{F}$. A function symbol with arity 0 is called a constant. We denote by $T(\mathcal{F}, \mathcal{V})$ the set of terms build up from $\mathcal{F}$ and $\mathcal{V}$. A ground term is a term without variables, and the set of ground terms is denoted by $T(\mathcal{F})$. For a term s, $\mathcal{F}_s$ is the set of function symbols occurring in s, $\mathrm{var}(s)$ is the set of variables occurring in s, the size $|s|$ is the number of symbols in s, and $|s|_{\mathrm{var}}$ is the

maximum number of occurrences of a variable in s. A function f is associative if it satisfies $f(f(x,y),z) = f(x,f(y,z))$, and it is commutative if it satisfies $f(x,y) = f(y,x)$.

A *substitution* θ is a mapping from the set of variables $\mathcal{V}$ to the set of terms $T(\mathcal{F})$. We are interested in finite substitutions, and we explicitly represent by $\{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$. The size of a substitution $|\theta|$ is given by $k + \sum_{i=1}^{k} |t_i|$. The domain of a substitution θ is extended to the set of all terms by inductively defining $\theta(f(t_1, \ldots, t_n))$ to be $f(\theta(t_1), \ldots, \theta(t_n))$. A substitution θ is said to *match* a term s with a term t if and only if $s\theta = t$. We can extend the notion of matching considering a set of equations E, an equational theory, taking into account the congruence classes of the congruence relation generated by E .

Here we consider the parameterized version of the matching problem. In this case, given two terms $s \in T(\mathcal{F},\mathcal{V})$ and $t \in T(\mathcal{F})$, the problem asks if there exists a substitution θ such that $s\theta =_E t$ for some equational theory $E \in \{\text{A, C, AC}\}$. For a list of parameters P, we define $[P]$-E-MATCHING as parameterized E-matching for $E \in \{\text{A, C, AC}\}$, where the parameterization is the sum of the parameters in P. We formalize this parameterized problem by

$[P]$-E-MATCHING	
Instance:	A first-order term $s \in T(\mathcal{F},\mathcal{V})$, and first-order term $t \in T(\mathcal{F})$, and a natural number k.
Parameter:	k such that $k = \sum_{\kappa \in P} \kappa(x)$.
Problem:	Does there exist a θ such that $s\theta =_E t$?

For example, $[|\mathcal{F}|, |\theta|]$-A-MATCHING is associative matching with parameterization $\kappa(s,t) = |\mathcal{F}| + |\theta|$. With respect to the set $\mathcal{F}$, we consider general matching where some function symbols may be uninterpreted, meaning that they are function symbols not appearing in E.

We will consider the equivalence of ground terms with respect to the associative, commutative and associative-commutative functions in the subsequent results.

Lemma 2.1 ([1, 2])**.** *Associative, commutative, and associative-commutative equality can be done in polynomial time.*

3 W[P] membership

The class W[P] is defined using algorithms with bounded non-determinism. Essentially, it contains problems that can be decided in FPT by an algorithm using at most $h(\kappa(x)) \cdot \log |x|$ non-deterministic steps.

3.1 $|\mathbf{var}(s)|$-C-MATCHING

In [1], C-MATCHING is conjectured to be in FPT. Here, we show that C-MATCHING is in W[P] by a non-deterministic algorithm with a limited number of non-deterministic steps in terms of the number of variables.

Theorem 3.1. *$|var(s)|$-C-MATCHING is in W[P].*

Proof. Given two terms $s \in T(\mathcal{F},\mathcal{V})$ and $t \in T(\mathcal{F})$ with $|\text{var}(s)| = k$. We design a Turing machine with running time $f(k) \cdot |(s+t)|^{O(1)}$ and at most $h(k) \cdot \log |(s+t)|$ non-deterministic steps for the input (s,t). For each variable x_i, it guesses a position v_i in t. To encode these positions in a Turing machine, it needs $k \log |t|$ non-deterministic steps. The machine apply the substitution θ to s producing $s\theta$, and then check if $s\theta =_C t$. For every variable, the process

corresponds to the detection of its encoding in the term s and a shift on the tape of the machine on the size of the sub-term t_i from t in the position v_i. The substitution and the equality modulo commutativity are made in polynomial time. This algorithm leads to a definition in W[P]. □

If we consider a parameter that it is greater than the number of variables, membership in W[P] remains for commutativity. One step further, considering the size of the substitution $|\theta| = k + \sum_{i=1}^{k} |t_i|$, we can verify membership in W[P] for the $|\theta|$-{A, AC}-MATCHING problems.

3.2 $|\theta|$-{A, AC}-MATCHING

Now, considering $|\theta|$ as the parameter, we can build up a non-deterministic Turing machine that behaves like the previous one. It guesses a substitution θ and then checks if the equivalence holds.

Theorem 3.2. *The $|\theta|$-{A, AC}-MATCHING problems are in W[P].*

Proof. This proof is similar to the previous one. Given two terms $s \in T(\mathcal{F}, \mathcal{V})$ and $t \in T(\mathcal{F})$, some natural number k, the parameter, and with $|\text{var}\ (s)| = \ell$. It guesses ℓ terms $t_i \in T(\mathcal{F})$ with size bounded by k, instantiate them in s and check if $s\ \theta =_E t$. Let $m = \max\{|ti| : 1 \leq i \leq \ell\}$. Again, the substitution and the equality modulo E are made in polynomial time observing the same procedure in the proof of Theorem 3.1. In both cases, we obtain an algorithm in W[P] for $|\theta|$-{A, AC}-MATCHING. □

If we consider $|s|_{\text{var}}$, the number of occurrences of variables, as a parameter, it is unlikely that E-MATCHING is in FPT. Moreover, it is unlikely to be within XP assuming P $\neq$ NP, once the problem with $|s|_{\text{var}} = 2$ is already a NP-complete problem [9] and, from the definition of XP, all slices of a problem in XP are polynomial time decidable. The size of the vocabulary $\mathcal{F}$ is not a good parameter for the same reasons. The {A, C, AC}-MATCHING problems are NP-complete with fixed vocabulary [2]. But if we add the size of the vocabulary to the size of the substitution, we can obtain fixed-parameter tractability.

4 Fixed-parameter tractability

From the perspective of parameterized complexity theory, the parameter is expected to be smaller than the input size. If we consider, for example, the size of the ground term $|t|$, it will lead to the case where the parameterized complexity is uninteresting, or trivially fixed-parameter tractable. In such conditions where the parameter increases monotonically with the size of the input, the problem is in FPT [4, Chapter 1]. However, this is not the case for the parameters $|\mathcal{F}|$ and $|\theta|$, and we will describe an algorithm in FPT for the matching problems considered here.

4.1 $[|\mathcal{F}|, |\theta|]$-{A, C, AC}-MATCHING

We show a brute-force algorithm that solves these matching problems and takes time in FPT when parameterized by $|\mathcal{F}| + |\theta|$. The solution enumerates of all possible substitutions checking whether it corresponds to a match.

Algorithm 1 follows the same idea that was described in [3]. First, it constructs the set of ground terms $T(\mathcal{F}_t)$ with size at most k, for the natural number k. Then, for every $(|\text{var}(s)|)$-tuple of terms in $T(\mathcal{F}_t)$, we build a substitution θ, apply it into s, i.e., for every variable

occurrence, we remove its encoding from s inserting the encoding of a term, and evaluate whether $s\theta$ is equal to t modulo E $\in$ {A, C, AC}. The equivalence of terms with respect to associative, commutative, and associative-commutative terms implemented in Step 5 can be computed in polynomial time (via Lemma 2.1).

Algorithm 1 {A, C, AC}-MATCHING via brute force

INPUT: A term s in $T(\mathcal{F}, \mathcal{V})$, a term t in $T(\mathcal{F})$, and a natural number k.
OUTPUT: Yes iff there exists a substitution θ s.t. $s\theta = t$, and $|\theta| \leq k$.

1: $T(\mathcal{F}_t) \leftarrow$ GENERATE(t, k) ▷ It constructs all terms in $\mathcal{F}_t$ with size bounded by k.
2: **for every tuple of terms** $(t_1, \dots t_{|\mathtt{var}(s)|})$ **in** $T(\mathcal{F}_t)$ **do**
3: **for** $i = 1$ to $|\mathrm{var}(s)|$ **do**
4: $\theta \leftarrow \theta \cup \{x_i \mapsto t_i\}$
5: **if** $s\,\theta =_{\mathrm{E} \in \{\mathrm{A},\ \mathrm{C},\ \mathrm{AC}\}} t$ **then return** Yes;
return No;

Proposition 4.1. *The running time of Algorithm 1 is* $|\mathcal{F}|^{k^2} \cdot p(|s| \cdot |t| \cdot k)$ *for some polinomial* p.

Proof. Let $|\mathcal{F}_t|$ be the number of symbols in t. The number of terms in $T(\mathcal{F}_t)$ with size at most k is $O(|\mathcal{F}_t|^{k+1})$, k comes from the input, and it is an upper bound for $|\theta|$. Then, the main loop will take at most $O((|\mathcal{F}|^{k+1})^{|\mathrm{var}(s)|})$ iterations. The construction of θ in Step 4 is bounded by a polynomial in k. The application of θ into s is obtained in time polynomial in $|s| \cdot k$. The equality modulo E can be done in time polynomial in $|s\theta| \cdot |t|$ by Lemma 2.1. Then, the whole computational complexity of the algorithm is $(|\mathcal{F}|^{k+1})^{|\mathrm{var}(s)|} \cdot p(|s| \cdot |t| \cdot k)$ for some polinomial p. □

Theorem 4.2. *The* $[|\mathcal{F}|, |\theta|]$*-E-MATCHING problem is in FPT for* $E \in \{A,\ C,\ AC\}$.

Proof. Algorithm 1 is an algorithm that solves these matching problems in time $f(|\mathcal{F}|, |\theta|) \cdot (|s| + |t|)^{O(1)}$ for some computable function f. Then, we can conclude that they are in FPT. □

5 Conclusion

We provide some parameterized complexity results for the matching problem of first-order terms concerning associative, commutative, and associative-commutative functions. Fixed-parameter tractability for these problems is achieved when considering the number of function symbols and the size of the substitution as the parameters.

With respect to the number of variables, we show that the C-MATCHING problem is in W[P] by an algorithm with a limited number of non-deterministic steps. Then, the conjecture stated in [1] remains, and it is open if this problem is in W[1]. For the other two problems, we cannot say anything better than the membership in para-NP, and we wonder if it is the case that $|\mathrm{var}(s)|$-{A, AC}-MATCHING is in W[1].

Considering the size of the substitution, we show that the $|\theta|$-{A, AC}-MATCHING problems are in W[P], and the previous open question can be restated for this parameterization. Can we locate this problem in a finite level of the W-Hierarchy?

References

[1] Tatsuya Akutsu, Jesper Jansson, Atsuhiro Takasu, and Takeyuki Tamura. On the parameterized complexity of associative and commutative unification. *Theoretical Computer Science*, 660:57–74, 2017.

[2] Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of matching problems. *Journal of symbolic computation*, 3(1-2):203–216, 1987.

[3] Henning Fernau, Markus L Schmid, and Yngve Villanger. On the parameterised complexity of string morphism problems. *Theory of Computing Systems*, 59(1):24–51, 2016.

[4] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Springer Verlag, Berlin, 2006.

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

[6] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th international symposium on symbolic and algebraic computation*, pages 296–303. ACM, 2014.

[7] Jaroslav Nešetřil and Svatopluk Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985.

[8] Mihai Pătraşcu and Ryan Williams. On the possibility of faster sat algorithms. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1065–1075. SIAM, 2010.

[9] Rakesh M Verma and IV Ramakrishnan. Tight complexity bounds for term matching problems. *Information and Computation*, 101(1):33–69, 1992.

On Asymmetric Unification for the Theory of XOR with a Homomorphism

Christopher Lynch[1], Andrew M. Marshall[2], Catherine Meadows[3], Paliath Narendran[4], and Veena Ravishankar[2]

[1] Clarkson University, Potsdam, NY, U.S.A.
clynch@clarkson.edu
[2] University of Mary Washington, Fredericksburg, VA, U.S.A.
marshall@umw.edu, vravisha@umw.edu
[3] Naval Research Laboratory, Washington, D.C., U.S.A.
catherine.meadows@nrl.navy.mil
[4] University at Albany-SUNY, Albany, NY, U.S.A.
pnarendran@albany.edu

1 Introduction

We examine the newly developed paradigm of asymmetric unification in the theory of *xor* with a homomorphism. Asymmetric unification is motivated by requirements arising from symbolic cryptographic protocol analysis [4]. These symbolic analysis methods require unification-based exploration of a space in which the states obey equational theories expressible as a decomposition $R \uplus \Delta$, where R is a set of rewrite rules. But in order to apply state space reduction techniques, it is usually necessary for at least part of this state to remain in normal form after unification is performed. This can be expressed as an *asymmetric* unification problem $\{s_1 =_{\downarrow} t_1, \ldots, s_n =_{\downarrow} t_n\}$ where the $=_{\downarrow}$ denotes a unification problem with the restriction that any unifier leaves the right-hand side of each equation irreducible.

The most commonly used algorithm is one based on *variant unification* [6], which turns an $R \uplus \Delta$-problem into a set of Δ-problems. Variant unification requires satisfaction of a number of conditions on the decomposition. Unfortunately, there is a class of theories important for cryptographic protocol analysis with no suitable decompositions: theories with an operator h homomorphic over an Abelian group operator $+$, that is AGh. In addition, for the usual decomposition of AGh with $\Delta = AC$, asymmetric unification has recently been shown to be undecidable [9]. Thus alternative approaches are called for.

In this paper we concentrate on asymmetric unfication for a special case of AGh: the theory of *xor* with homomorphism, or $ACUNh$, using a decomposition of the form $R \uplus ACh$. We first develop an automata-based asymmetric decision procedure for $R \uplus ACh$ with free function symbols. Note that it is known that unification modulo ACh is undecidable [8], so our result also yields the first asymmetric decision procedure for which Δ does not have a decidable finitary unification algorithm. We also consider the problem of producing complete sets of asymmetric unifiers modulo $R \uplus ACh$. We define an automaton that generates a (possibly infinite) complete set of solutions, and prove via an example that asymmetric unification modulo $R \uplus ACh$ is infinitary.

Outline: Section 2 provides a brief description of preliminaries. Section 3 develops an automaton based decision procedure for the $ACUNh$-theory. In Section 4 an automaton approach that produces substitutions is outlined. Section 5 develops the modified combination method needed to obtain general asymmetric algorithms.

2 Preliminaries

We use the standard notation of equational unification [2] and term rewriting systems [1]. Due to space considerations we only give a few definitions here.

Definition 1. *Let R be a term rewriting system and E be a set of identities. We say (R,E) is R,E-*convergent *if and only if $\rightarrow_{R,E}$ is terminating, and for all terms s, t, if $s \approx_{R\cup E} t$, there exist terms s', t' such that $s \rightarrow^{!}_{R,E} s'$, $t \rightarrow^{!}_{R,E} t'$, and $s' \approx_E t'$*

Definition 2. *We call $(\Sigma,\ E,\ R)$ a* weak decomposition *of an equational theory Δ over a signature Σ if $\Delta = R \uplus E$ and R and E satisfy the following conditions: Matching modulo E is decidable, R is terminating modulo E, i.e., the relation $\rightarrow_{R/E}$ is terminating, The relation $\rightarrow_{R,E}$ is confluent and E-coherent, i.e., $\forall t_1,t_2,t_3$ if $t_1 \rightarrow_{R,E} t_2$ and $t_1 =_E t_3$ then $\exists\ t_4,t_5$ such that $t_2 \rightarrow^{*}_{R,E} t_4$, $t_3 \rightarrow^{+}_{R,E} t_5$, and $t_4 =_E t_5$.*

This is a modification of the definition in [4] where asymmetric unification and the corresponding theory decomposition are first defined. These conditions ensure that $s \rightarrow^{!}_{R/E} t$ iff $s \rightarrow^{!}_{R,E} t$ (see [6, 4]).

Definition 3 (Asymmetric Unification)**.** *Given a weak decomposition (Σ,E,R) of an equational theory, a substitution σ is an* asymmetric R,E-unifier *of a set $\mathscr{S}$ of asymmetric equations $\{s_1 =_{\downarrow} t_1,\ \ldots,\ s_n =_{\downarrow} t_n\}$ iff for each asymmetric equation $s_i =_{\downarrow} t_i$, σ is an $(E\cup R)$-unifier of the equation $s_i =^{?} t_i$ and $(t_i \downarrow_{R,E})\sigma$ is in R,E-normal form.*

Example 1: Let $R = \{x \oplus 0 \rightarrow x,\ x \oplus x \rightarrow 0,\ x \oplus x \oplus y \rightarrow y\}$ and E be the *AC* theory for $\oplus$. Consider the equation $y \oplus x =_{\downarrow} x \oplus a$. The substitution $\{y \mapsto a\}$ is an asymmetric solution, but $\{x \mapsto 0,\ y \mapsto a\}$ is not. The instances of asymmetric unifiers need not be asymmetric unifiers.

Definition 4 (Asymmetric Unification with Linear Constant Restriction)**.** *Let $\mathscr{S}$ be a set of asymmetric equations with linear constant restriction (LCR) [2]. A substitution σ is an* asymmetric R,E-unifier *of $\mathscr{S}$ with LCR iff σ is an asymmetric solution to $\mathscr{S}$ and σ satisfies the LCR.*

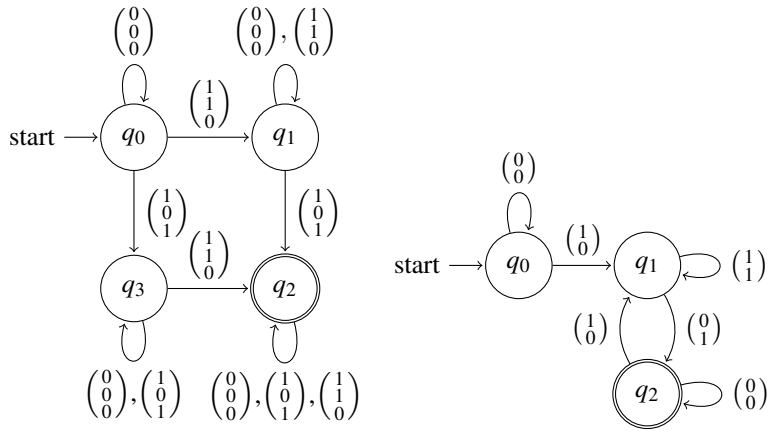
3 An Asymmetric ACUNh-unification Decision Procedure

In this section we develop a new asymmetric unification algorithm for the theory *ACUNh*. Following the definition of asymmetric unification, the theory *ACUNh* is decomposed into a set of rewrite rules, R, modulo a set of equations, Δ. The decomposition has *associativity*, *commutativity* and the distributive homomorphism identity as Δ, i.e., $\Delta = ACh$. Let R_2= $\{x+x \rightarrow 0,\ x+0 \rightarrow x,\ x+(y+x) \rightarrow y,\ h(0) \rightarrow 0\}$

Lemma 3.1. *$\longrightarrow_{R_2,ACh}$ is ACh-convergent.*

Another decomposition keeps *associativity* and *commutativity* as identities Δ and the rest as rewrite rules. This decomposition has the following *AC*-convergent term rewriting system R_1: $R_1 = R_2 \cup R_h$, where $R_h = h(x+y) \rightarrow h(x)+h(y)$

Decidability of asymmetric unification for the theory R_2,ACh can be shown by automata-theoretic methods analogous to the method used for deciding the Weak Second Order Theory of One successor (WS1S) [3] i.e., by reduction to the Weak Second Order Theory of One successor (WS1S). In WS1S we consider quantification over finite sets of natural numbers, along with one successor function. All equations or formulas are transformed into finite-state automata that accept the strings that correspond to a model of the formula [7]. This automata-based approach is key to showing decidability of WS1S,



(a) Automaton for $\mathsf{P} =_{\downarrow} \mathsf{Q}+\mathsf{R}$ (b) Automaton for $\mathsf{X} =_{\downarrow} \mathsf{h}(\mathsf{Y})$

Figure 1: Automata construction

since the satisfiability of WS1S formulas reduces to the automata intersection-emptiness problem. We follow the same approach here.

For ease of exposition, let us consider the case where there is only one constant a. Thus every ground term can be represented as a set of natural numbers. The homomorphism h is treated as a successor function. Just as in WS1S, the input to the automata are column vectors of bits. The length of each column vector is the number of variables in the problem. $\Sigma = \left\{ \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \ldots, \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \right\}$

The deterministic finite automata (DFA) for the equation $\mathsf{P} =_{\downarrow} \mathsf{Q}+\mathsf{R}$ is illustrated in Figure 1a. The $+$ operator behaves like the *symmetric set difference* operator. To preserve asymmetry on the right-hand side of this equation, $\mathsf{Q}+\mathsf{R}$ should be irreducible. If either Q or R is empty, or if they have any term in common, then a reduction will occur. For example, if $\mathsf{Q} = \mathsf{h}(\mathsf{a})$ and $\mathsf{R} = \mathsf{h}(\mathsf{a})+\mathsf{a}$, there is a reduction, whereas if $\mathsf{R} = \mathsf{h}(\mathsf{a})$ and $\mathsf{Q} = \mathsf{a}$, irreducibility is preserved, since there is no common term and neither one is empty. Since neither Q nor R can be empty, any accepted string should have one occurrence of $\binom{1}{0}$ and one occurrence of $\binom{1}{1}$. $\binom{P}{Q}$ is the ordering of variables.

Fig. **1b**: In this equation, $\mathsf{h}(\mathsf{Y})$ should be in normal form. So Y cannot be 0, but can contain terms of the form $\mathsf{u}+\mathsf{v}$. $\binom{Y}{X}$ is the ordering of variables. Therefore the bit vector $\binom{1}{0}$ should be succeeded by $\binom{0}{1}$, with possible occurrences of the bit vector $\binom{1}{1}$ in between. Thus the string either ends with $\binom{0}{1}$ or $\binom{0}{0}$. For example, if $\mathsf{Y} = \mathsf{h}(\mathsf{a})+\mathsf{a}$, then $\mathsf{X} = \mathsf{h}^2(\mathsf{a})+\mathsf{h}(\mathsf{a})$. The string $\binom{1}{0}\binom{1}{1}\binom{0}{1}$ gets accepted.

Once we have automata constructed for all the equations, we take the intersection and check if there exists a string accepted by all automata. If the intersection is not empty, then we have a solution or an asymmetric unifier for set of equations. We omit the details here due to space considerations.

4 Automaton to find a complete set of unifiers

We create automata to find all solutions of an ACUNh asymmetric unification problem with constants.

Definition 5. *Let t be a term whose R_h normal form is $t_1+\cdots+\cdot t_n$. Then we define $mset(t) = \{t_1,\cdots,t_n\}$. Inversely, if $T = \{t_1,\cdots,t_n\}$ then $\Sigma T = t_1+\cdots+t_n$. We define the maximum degree of a term t in R_h*

normal form to be i if i is the largest number such that $h^i(s)$ is in $mset(t)$ and s does not have an h at the root. Let ζ be a substitution and X be a set of variables. Then ζ is a zero substitution *on X if $Dom(\zeta) \subseteq X$ and $x\zeta = 0$ for all $x \in Dom(\zeta)$. Let t be an object. Define $loseh(t) = \Sigma\{h^i(t) \mid h^{i+1}(t) \in mset(t \downarrow_{R_h})\}$.*

Definition 6. *Let P be a set of ACUNh asymmetric equations. Let m be the maximum degree of terms in P. Let Θ be the set of all substitutions θ such that $Dom(\theta) \subseteq Var(P)$ and for all $x \in Dom(\theta)$, $x\theta = \Sigma T$ where T is a nonempty set containing constants from P and $h(x)$. Let $u =^?_\downarrow v$ be an ACUNh asymmetric equation. The automaton $M(u =^?_\downarrow v, P) = (Q, q_{u=^?_\downarrow v}, F, \Theta, \delta)$, where Q is the set of states, $q_{u=^?_\downarrow v}$ is the start state, F is the set of accepting states, Θ is the alphabet, and δ is the transition function:*

- *Q is a set of states of the form $q_{s=^?_\downarrow t}$, where $s = \Sigma S$ and $t = \Sigma T$, for some S and T which are sets containing terms of at most degree m.*
- *$F = \{q_{s=^?_\downarrow t} \in Q \mid mset(s) = mset(t)\}$*
- *$\delta : Q \times \Theta \longrightarrow Q$ such that $\delta(q_{s=^?_\downarrow t}, \theta) = q_{loseh(s\theta)\downarrow_{R_1} =^?_\downarrow loseh(t\theta)}$ if $Dom(\theta_i) = Var(s =^?_\downarrow t)$, the mulitset of constants and variables in $mset((s\theta)\downarrow_{R_1})$ is the same as the multiset of constants and variables in $mset(t\theta)$, and $mset(t\theta)$ contains no duplicates.*

Now we show that these automata can be used to find all asymmetric ACUNh unifiers.

Theorem 4.1. *Let P be a set of asymmetric ACUNh equations, such that all terms in P are reduced by R_1. Let θ be a substitution which is reduced by R_1. Then θ is a solution to P if and only if there exists a zero substitution ζ on P where all right hand sides in $P\zeta$ are irreducible, and a sequence of substitutions $\theta_0, \cdots, \theta_m$ such that θ is more general than $\zeta\theta_0\cdots\theta_m$ and the string $\theta_0\cdots\theta_m$ is accepted by $M((u =^?_\downarrow v)\zeta)\downarrow_{R_1}, P'\zeta)$ for all $u =^?_\downarrow v \in P$, where $P' = P \cup \{c =^?_\downarrow c\}$ for a fresh constant c.*

Thus the set of solutions can be represented by a regular language (we could add linear constant restrictions and disequalities). If we only want to decide asymmetric unification, we check if there is an accepting state reachable from an initial state. We can enumerate all the solutions by finding all accepting states reachable in 1 step, 2 steps, etc. If there is a cycle on a path to an accepting state, then there are an infinite number of solutions, otherwise there are only a finite number of solutions. This will find all the ground substitutions. To find all substitutions, we generalize the solutions that we find and check them, and there are only a finite number of them. We only have to generalize terms containing c.

Figure 2 shows the automaton created for $h(x) + b =^?_\downarrow x + y$, where $P = \{h(x) + b =^?_\downarrow x + y\}$. The only valid zero substitution here is the identity. Note that c never appears in the domain of a substitution, because no such substitution satisfies the conditions for the transition function. This example shows that asymmetric ACUNh unification with constants is not finitary.

5 Combining with the Free Theory

In order to obtain a general asymmetric ACUNh-unification decision procedure we need to add free function symbols. We can do this by using disjoint combination. The problem of asymmetric unification in the combination of disjoint theories was studied in [5]. However, this algorithm does not immediately apply. For the automata approach it's not always possible to check solutions for theory preserving and injectivity properties since the automata may not actually produce a substitution. However, it is possible to build constraints into the automata that enforce these conditions. Therefore, the algorithm of [5] can be modified with the following properties. For each *ACUNh*-pure problem, partition, and

theory index, an automata is constructed enforcing the injective and theory preserving restrictions. Since these restrictions are built into the automata, the only *ACUNh* solutions produced will be both theory preserving and injective. These can then be combined with solutions from the free theory which can be easily checked.

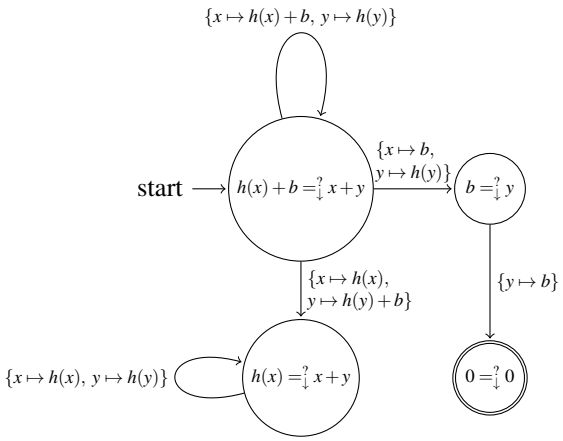


Figure 2: Substitution producing automaton

References

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

[2] Franz Baader and Wayne Snyder. Unification theory. *Handbook of Automated Reasoning*, 1:445–532, 2001.

[3] Calvin C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98(1):21–51, 1961.

[4] Serdar Erbatur, Santiago Escobar, Deepak Kapur, Zhiqiang Liu, Christopher A. Lynch, Catherine Meadows, José Meseguer, Paliath Narendran, Sonia Santiago, and Ralf Sasse. Asymmetric Unification: A New Unification Paradigm for Cryptographic Protocol Analysis. In *Automated Deduction, (CADE-24)*, volume 7898 of *LNCS*, pages 231–248. 2013.

[5] Serdar Erbatur, Deepak Kapur, Andrew M. Marshall, Catherine A. Meadows, Paliath Narendran, and Christophe Ringeissen. On asymmetric unification and the combination problem in disjoint theories. In *Foundations of Software Science and Computation (FOSSACS-17), volume 8412 LNCS*, pages 274–288, 2014.

[6] Santiago Escobar, José Meseguer, and Ralf Sasse. Variant narrowing and equational unification. *Electr. Notes in Theoretical Computer Science*, 238(3):103–119, 2009.

[7] Felix Klaedtke and Harald Ruess. Parikh automata and monadic second-order logics with linear cardinality constraints. Technical Report 177, Universität Freiburg, 2002.

[8] Paliath Narendran. Solving linear equations over polynomial semirings. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 466–472. IEEE Computer Society, 1996.

[9] Veena Ravishankar. *Asymmetric Unification and Disunification*. PhD thesis, University at Albany–SUNY, 2018.

On Forward-closed and Sequentially-closed String Rewriting Systems

Yu Zhang[1], Paliath Narendran[2], and Heli Patel[3]

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

Abstract

In this paper we introduce the new concept of *sequentially-closed string rewriting systems* which generalizes the concept of forward-closed systems. We also investigate subclasses and properties of finite sequentially-closed systems and regular forward-closed systems.

The motivation for defining this new class comes from investigating the reachability problem for what we define as $\widehat{O_3}$ systems. (A subclass of these were studied earlier in relation to the *accessibility problem*.) We prove that the reachability problem is undecidable by a reduction from the halting problem for Intercell Turing Machines. We show that checking whether a given string rewriting system is sequentially-closed is a decidable problem. We also show that every congruence class modulo a convergent, sequentially-closed string rewriting system is a context-free language (extending a similar result for forward-closed systems).

We also investigate the properties of infinite regular string rewriting systems. We show that checking whether a regular string rewriting system is forward-closed *and* confluent is decidable.

Keywords: string rewriting system, forward-closed, sequentially-closed

1 Introduction

The current work is inspired by the paper [6], which introduces a novel, intriguing class of string rewriting systems. The accessibility problem (in their formulation) is defined as follows:

> Let $G = (V,E)$ be a directed graph. Let τ be a type function mapping vertices to a set of types T. Two operations *Get* and *Insert* are defined in terms of two ternary relations $R_I, R_E \subseteq T \times T \times T$: *Get(a,b,c)*: Add edge $a \to c$, if edges $a \to b$ and $b \to c$ already exist *and* $(\tau(a), \tau(b), \tau(c)) \in R_E$. *Insert(a,x,c)*: If $a \to c$ is an edge in the graph, then add node x and edges $a \to x$ and $x \to c$ if $(\tau(a), \tau(x), \tau(c)) \in R_I$. A node p can access a node d iff there exists a graph G_1 and a finite sequence of operations $f_1, f_2, \ldots, f_k$ [6] such that $G_1 = G \circ f_1 \circ f_2 \circ \ldots \circ f_k$ and $p \to d \in E(G_1)$.

In this paper we investigate the properties of various classes of string rewriting systems. Our motivation partly comes from [6] where undecidability of the accessibility problem for the above system, which we call $\widehat{O_3}$, is mentioned. However, the proof of undecidability is not given in [6]; it has not been published anywhere to the best of our knowledge. We show that the problem is indeed undecidable. The undecidability proof is done by a reduction from the halting problem for Intercell Turing Machines.

We also propose a new class of string rewriting systems, *sequentially-closed* string rewriting systems, which generalizes the notion of a forward-closed string rewriting system [4, 5]. We show that checking whether a given string rewriting system is sequentially-closed is a decidable problem. We

also show that every congruence class modulo a convergent, sequentially-closed string rewriting system is a context-free language (extending a similar result for forward-closed systems). Two cases of sequentially-closed string rewriting systems are considered: length-reducing $\widehat{O_3}$ systems and monadic systems.

We also investigate the properties of infinite regular string rewriting systems. Our motivation comes from [7] where it was shown that confluence of regular monadic Thue systems is decidable. We show that checking whether a regular string rewriting system is forward-closed *and* confluent is decidable.

In the interest of brevity we have omitted many details and most of the proofs of the results that appear in this short paper. The interested reader can find all of the details in the papers (in preparation) uploaded at `https://www.albany.edu/~YZ719878/`.

In the future, we plan to further explore the relation between forward-closed and sequentially-closed systems, especially in the regular case.

2 Definitions

We present here some essential definitions of string rewriting systems. For more details, we refer the reader to [1] for term rewriting systems, and to [3] for string rewriting systems.

A string is irreducible with respect to string rewriting system T if and only if no rule of T can be applied to it. The set of strings that are irreducible (i.e., in normal form) modulo T is denoted by $IRR(T)$. Note that this set is a regular language, since $IRR(T) = \Sigma^* \setminus \{\Sigma^* l_1 \Sigma^* \cup \ldots \cup \Sigma^* l_m \Sigma^*\}$, where $l_1, \ldots, l_m$ are the left-hand side of the rules in T. A string w' is a *T-normal form* (or a *normal form* if the rewrite system is obvious from the context) of a string w for a string rewriting system T if and only if $w \to_T^* w'$ and w' is irreducible. We write this as $w \to_T^! w'$. Let w^{rev} denotes w's reversal.

A string of the form wl where $w \in \Sigma^*$ and l is a left-hand side is called a *redex*. A redex is *innermost* if no proper prefix of it is a redex. The set of innermost redexes is denoted by $INNER(T)$. A string rewriting system is said to be *forward-closed* if every innermost redex can be reduced to its normal form in one step. A rewrite step $xly \to xry$ is *leftmost-innermost-smallest* if and only if (a) xl is an innermost redex, (b) xr is in normal form, and (c) if $l \to r'$ is another rule in the rewrite system and xr' is irreducible, then $r' \succ_{LL} r$ ($\succ$ is a given total ordering on the alphabet Σ and $\succ_{LL}$ is its length-+-lexicographic extension). We denote this rewrite relation by $\to_{\ell is}$. If $w \to_{\ell is}^! w'$, then w' is said to be a *leftmost-innermost-smallest* normal form and is denoted as $\ell is_T(w)$. Clearly if T is terminating and forward-closed, then $\ell is_T(w)$ exists and is unique for every string w. For a language L_0, we define $T^!_{lis}(L_0) = \{x \mid w \to^!_{lis} x, \text{ for some } w \in L_0\}$.

A string rewriting system is said to be *regular* if and only if it can be represented as $\{R_1 \to \alpha_1, R_2 \to \alpha_2, \ldots, R_n \to \alpha_n\}$ where the R_i is a regular set and α_i is a string ($1 \le i \le n$). In this paper, we focus on infinite regular systems, i.e., at least one of the R_i contains an infinite number of strings.

We call a string rewriting system an $\widehat{O_3}$ system if and only if every rule is either of the form $\tau_1\tau_3 \to \tau_1\tau_2\tau_3$ or $\tau_1\tau_2\tau_3 \to \tau_1\tau_3$ where τ_1, τ_2 and τ_3 are symbols in the alphabet.

3 Undecidability of the Accessibility Problem for $\widehat{O_3}$ System

We will prove that the accessibility problem for $\widehat{O_3}$ system is undecidable by a reduction from the halting problem for Intercell Turing Machines.

Theorem 3.1. *Given an Intercell Turing Machine M, an $\widehat{O_3}$ string rewriting system T can be constructed such that the acceptance problem for M is reducible to a reachability problem for T.*

Proof. Let $q_1, ..., q_m$ be the states of M including an accepting state q_{accept} and a rejecting state q_{reject}. Let Σ be its input alphabet and Γ its tape alphabet including the blank symbol $\sqcup$. The alphabet of T is a superset of the tape alphabet Γ: it includes symbols for each state of M, additional symbols such as #, \$, $\¢$ etc., and a mirror alphabet $\overline{\Gamma}$ such that for every $a \in \Gamma$, we have $\bar{a} \in \overline{\Gamma}$. Symbols in $\overline{\Gamma}$ are used to represent symbols in the cells to the *left* of the tape head.

We represent a configuration of M by a string of the form $\¢\#\bar{a}_1\#\bar{a}_2\#\ldots\#q\#a_i\#\ldots\#a_n\#\sqcup\$$ where the state is q and the tape head is reading the symbol a_i. $\¢$ and \$ stand for the left and right endmarkers for the portion of the tape that has been "in action" (Note that the symbols of the tape alphabet will be the vertices of our accessibility system S.). We formulate rules for T based on the quintuples of the Intercell Turing machine. Each quintuple is given a distinct label l or r with subscripts, depending on whether the move is to the left or to the right. For label l and r, we also introduce new symbol $\#_l$ and $\#_r$. We now show how rules are created for right-moving quintuple, similar for the left-moving quintuple.

For instance, a right-moving quintuple $r_1 = (a_i, q_x, \bar{a}_j, R, q_y)$ represents changing part of the configuration from $\#q_x\#a_i\#$ to $\#\bar{a}_j\#q_y\#$. The corresponding productions are:

step 1 :	step 2 :	step 3 :	step 4 :
$q_x\#a_i \to q_x a_i$	$\#q_x\#_{r_1} \to \#\#_{r_1}$	$\#_{r_1}a_i\# \to \#_{r_1}\#$	$\bar{a}_j\#_{r_1}q_y \to \bar{a}_j q_y$
$q_x a_i \to q_x\#_{r_1}a_i$	$\#\#_{r_1} \to \#\bar{a}_j\#_{r_1}$	$\#_{r_1}\# \to \#_{r_1}q_y\#$	$\bar{a}_j q_y \to \bar{a}_j\#q_y$

Once the accepting or rejecting state is reached we erase all symbols in $\Gamma \cup \{\#\}$, i.e., all symbols except $\¢$, \$, the state symbols and the symbols associated with transition labels (e.g., $\#_r$) using the rules $q_{accept}xy \to q_{accept}y$, $xyq_{accept} \to xq_{accept}$, $\¢q_{accept}\$ \to \¢\$$. □

Claim 3.1.1. *If $C = \¢\#q_1\#a_1\#a_2\#\ldots\#a_n\#\sqcup\$$ is the string representing a given initial configuration of M, the string $\¢\$$ is derivable from the accessibility rules if and only if M halts when started from configuration C.*

Since the halting problem of M is undecidable, the accessibility problem is undecidable as well.

4 Sequentially-closed String Rewriting Systems

An innermost redex is said to be *sequential* if and only if it can be reduced in one step to either another innermost redex or to a normal form. If $\omega \to \omega'$ and both ω and ω' are innermost redexes, then we call ω' the *successor innermost redex* (or simply the *successor*) of ω. A string rewriting system

is *sequentially-closed* if every innermost redex is sequential. All the forward-closed string rewriting systems are sequentially-closed. Let L be the length of a system's longest left-hand side. The following lemma shows that checking whether a given string rewriting system is sequentially-closed is a decidable problem.

Lemma 4.1. *A string rewriting system T is sequentially-closed if and only if every innermost redex of length $2L$ or less is sequential.*

Lemma 4.2. *Let T be a sequentially-closed string rewriting system and L be the length of its longest left-hand side. Let x, y_1, y_2 be strings such that $xy_1 \in IRR(T)$ and $|y_1| = |y_2| = L$. Then xy_1y_2 is an innermost redex if and only if y_1y_2 is an innermost redex.*

Theorem 4.1. *Every congruence class modulo a convergent sequentially-closed string rewriting system T is a context-free language.*

The above theorem can be proved by creating a Pushdown automata (PDA) $\mathscr{M}$ that recognize the congruence class of $\$\chi$, where $\chi \in IRR(T)$. $\mathscr{M}$ will carry out the transitions by pushing symbols of the input string, and reducing each redex that appears on the stack. The string in the stack is either an innermost redex, or in normal form. Before pushing another symbol, the string on the stack should be in normal form. After pushing all the symbols of the input string, the contents of the stack must be $\$\chi$. By Lemma 4.2, the PDA only needs to consider the top $2L$ symbols in the stack in order to determine a successor innermost redex.

We also prove that length-reducing $\widehat{O_3}$ systems and monadic systems are sequentially-closed. The congruence class over them can be not regular. Here we use length-reducing $\widehat{O_3}$ system as example.

Lemma 4.3. *Every length-reducing $\widehat{O_3}$ system T is sequentially-closed.*

We now show that there is a length-reducing and convergent $\widehat{O_3}$ system which has a *non-regular* congruence class. Let $T_{\widehat{O_3}} = \{abc \to ac, dac \to dc, dcb \to db, adb \to ab\}$ which is confluent. Now consider the congruence class of ab, i.e., $[ab] = \{w \mid w \to^*_{T_{\widehat{O_3}}} ab\}$. $[ab]$ is not regular can be proved by letting $L_1 = [ab] \cap (\alpha^* ab \beta^*) = \{\alpha^m ab \beta^m | m \geq 0\}$ and L_1 is not regular.

5 Forward Closure and Confluence of Regular Forward-closed Systems are Decidable

The following lemma can prove the foward closure of regular SRS is decidable.

Lemma 5.1. *Let $T = \{R_1 \to \alpha_1, R_2 \to \alpha_2, \ldots, R_n \to \alpha_n\}$ be a regular string rewriting system. Then T is forward-closed if and only if $INNER(T)$ is an equivalent of*

$$(IRR(T)/\alpha_1) \cdot R_1 \cup (IRR(T)/\alpha_2) \cdot R_2 \cup \ldots \cup (IRR(T)/\alpha_n) \cdot R_n$$

where $(IRR(T)/\alpha_i) \cdot R_i = \{wx \mid w\alpha_i \in IRR(T)$ and $x \in R_i\}$, $1 \leq i \leq n$.

Our proof of the decidability of confluence of regular forward-closed string rewriting system is based on the following criteria from Section 3 of [7]. If β_1, β_2 are the left-hand side of the rules in T, then:

$$ACLASH(\beta_1, \beta_2) = \{(\beta_1 w, u\beta_2) \in \Sigma^* \times \Sigma^* \mid \exists v \neq \varepsilon : ((uv \to \beta_1) \in T \text{ and } (vw \to \beta_2) \in T)\}$$

$$BCLASH(\beta_1, \beta_2) = \{u\beta_2 w \in \Sigma^* \mid \exists v : ((v \to \beta_2) \in T \text{ and } (uvw \to \beta_1) \in T)\}$$

A regular string rewriting system T is confluent if and only if the following criteria hold:

(A) for every right-hand side β_1, β_2: $ACLASH(\beta_1,\beta_2) \cap \{(x,y) \mid x{\downarrow_T} \neq y{\downarrow_T}\} = \emptyset$;

(B) for every right-hand side β_1, β_2: $BCLASH(\beta_1,\beta_2) \cap \{w \mid w{\downarrow_T} \neq \beta_1{\downarrow_T}\} = \emptyset$;

From Lemma 3.2.1 in [7], we know $BCLASH(\beta_1,\beta_2)$ is always regular. If we can prove that $\{w \mid w{\downarrow_T} \neq \beta_1{\downarrow_T}\}$ is a context-free language, then that shows that criterion (B) is decidable, since it is decidable if the intersection of a regular set and a context-free language is empty.

Lemma 5.2. *If T is a regular forward-closed string rewriting system and $\chi \in IRR(T)$, then $L_2 = \{u\#\chi^{rev} \mid u \rightarrow^!_{\ell is} v,\ v \neq \chi\}$ is a context-free language, and a nondeterministic pushdown automaton (NPDA) recognizing L_2 can be constructed from T and χ.*

We can see that the language recognized by the NPDA in the above lemma is $\{w \mid w{\downarrow_T} \neq \beta_1{\downarrow_T}\}$.

To prove criterion (A) is decidable and make the decision procedure less complicated, we handle subsets of $\Sigma^*\#\Sigma^*$ rather than subsets of $\Sigma^* \times \Sigma^*$. We define:

$$\begin{aligned} ACLASH'(\beta_1,\beta_2) &= \{\beta_1 w\#u\beta_2 \mid (\beta_1 w, u\beta_2) \in ACLASH(\beta_1,\beta_2)\} \\ T^!_{\ell is}(ACLASH'(\beta_1,\beta_2)) &= \{y_1\#y_2 \mid (\beta_1 w, u\beta_2) \in ACLASH(\beta_1,\beta_2), \beta_1 w \rightarrow^!_{\ell is} y_1, u\beta_2 \rightarrow^!_{\ell is} y_2\} \end{aligned}$$

It is decidable if all the members of a finite set are in certain format. If we can decide whether $T^!_{\ell is}(ACLASH'(\beta_1,\beta_2))$ is finite and $T^!_{\ell is}(ACLASH'(\beta_1,\beta_2)) \subseteq \{y\#y \mid y \in \Sigma^*\}$, then criterion (A) is decidable. The proof of this section will depend on the following claim from [2]:

Claim 5.0.1. *Let $L \subseteq \Sigma^*\{\#\}\Sigma^*$ be a context-free language, where $\# \notin \Sigma$. Suppose that for each $x \in \Sigma^*$, $\{y \mid x\#y \in L\}$ is finite. Then $\{y \mid$ for some $x \in \Sigma^*$, $x\#y \in L\}$ is a regular set.*

Lemma 5.3. *Let T be a regular terminating and forward-closed string rewriting system, and R be a regular language. Then $T^!_{\ell is}(R)$ is regular.*

$ACLASH'(\beta_1,\beta_2)$ is a regular language can be proved by using the similar proof of the Lemma 3.3.2 in [7]. It follows that $T^!_{\ell is}(ACLASH'(\beta_1,\beta_2))$ is a regular language too.

Lemma 5.4. *Let $L = \{w\#w \mid w \in \Sigma^*, \# \notin \Sigma\}$. If $L_3 \subseteq L$ and L_3 is regular, then L_3 is finite.*

Thus if a regular forward-closed string rewriting system T is confluent, $T^!_{\ell is}(ACLASH'(\beta_1,\beta_2))$ will be a finite subset of $\{w\#w \mid w \in \Sigma^*, \# \notin \Sigma\}$ for all β_1 and β_2. Hence the confluence of T can be verified.

References

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

[2] Ronald V. Book, Matthias Jantzen and Celia Wrathall. Monadic Thue systems. *Theoretical Computer Science* **19**:231–251, 1982.

[3] Ronald V. Book and Friedrich Otto. *String-rewriting systems.* Texts and Monographs in Computer Science, Springer, 1993.

[4] Daniel S. Hono II and Paliath Narendran. On forward closed string rewriting systems. In preparation.

[5] Daniel S. Hono II, Paliath Narendran, and Rafael Veras. Lynch-Morawska systems on strings. *CoRR*, abs/1604.06509, 2016.

[6] Rajeev Motwani, Rina Panigrahy, Vijay A. Saraswat, and Suresh Venkatasubramanian. On the decidability of accessibility problems (extended abstract). In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA,* pages 306–315. ACM, 2000.

[7] Colm Ó'Dúnlaing. Infinite regular Thue systems. *Theoretical Computer Science* **25**:171–192, 1983.

Unification of Multisets with Multiple Labelled Multiset Variables *

Zan Naeem[1] and Giselle Reis[1]

Carnegie Mellon University, Doha, Qatar
znaeem@andrew.cmu.edu, giselle@cmu.edu

Abstract

We look into the problem of unifying multisets containing (first-order) terms and *multiple* multiset variables. The variables are labelled, meaning that a unifier that places a term in a multiset variable M_i is different from another that places a term in a multiset variable M_j, for $i \neq j$. We describe a sound, complete, and terminating algorithm for computing the set of all possible unifiers, and analyse its complexity. We also prove an input pre-processing step that avoids the computation of less general unifiers.

1 Introduction

Multiset is an important data-structure that is used to specify various object systems. Our motivation stems mostly from proof theory, where logical entailment is encoded as sequents $\Gamma \vdash \Delta$, where both Γ and Δ are typically considered as multisets. When reasoning about such objects, one might need to use an implementation of sets/multisets based on lists, since most reasoning tools (i.e., logical frameworks, proof assistants, and logic programming languages) do not have built-in support for these data structures [2, 3, 6, 7]. Adding this kind of support requires, among other things, a unification algorithm.

Multiset unification was studied in [1, 4], where the authors propose solutions for the problem of unifying multisets with at most one multiset variable. We extend those results for multisets with multiple multiset variables. In our setting, each multiset variable is *labelled*, meaning that assigning a term to either a multiset M_i or M_j, where $i \neq j$, should be considered different solutions. We describe a terminating algorithm and analyse its complexity. Moreover, we prove that a simple modification of our algorithm avoids the computation of less general unifiers.

The need for labelled multiset variables emerged when reasoning with multiplicative rules in sequent calculi, such as:

$$\frac{\Gamma_1 \vdash A \quad \Gamma_2, B \vdash C}{\Gamma_1, \Gamma_2, A \to B \vdash C} \to_l$$

To apply this rule to, e.g., the sequent $\Gamma, D, A \to B \vdash C$, where A, B, C, and D are formulas, we need to unify its antecedent $\Gamma, D, A \to B$ with $\Gamma_1, \Gamma_2, A \to B$. Assigning formula D to Γ_1 or Γ_2 should be considered two different solutions, since they result in two different applications of the rule.

1.1 Preliminaries

A *multiset $\mathcal{M}$ with multiple labelled multiset variables* is denoted by $\{\!\{t_1, ..., t_n | M_1, ..., M_k\}\!\}$. Each t_i is a *term* ranging over a first-order language $\mathcal{L} = \langle \Sigma, \mathcal{V} \rangle$, where Σ is a set of constants

*This publication was made possible by the support of Qatar Foundation through Carnegie Mellon University in Qatar's Seed Research program. The statements made herein are solely the responsibility of the authors.

and function symbols, and $\mathcal{V}$ is a denumerable set of *term variables*. Each M_i is a *multiset variable* ranging over a denumerable set $\mathcal{V}_M$ of multiset variables. When n and k are not relevant, we abbreviate $t_1, ..., t_n$ as $\overline{t}$ and $M_1, ..., M_k$ as $\overline{M}$. When $k = 0$, we write the multiset as $\{\!\{t_1, ..., t_n\}\!\}$. Henceforth, we refer to multisets with multiple labelled multiset variables as *mmsets* for brevity.

Definition 1 (Substitution)**.** *An* mmset substitution σ *is a finite mapping of term variables to terms, and of multiset variables to mmsets. The application of a substitution* σ *to an mmset* $\{\!\{t_1, ..., t_n | M_1, ..., M_k\}\!\}$ *is defined as:*

$$\{\!\{t_1, ..., t_n | M_1, ..., M_k\}\!\}\sigma = \{\!\{t_1\sigma, ..., t_n\sigma\}\!\} \uplus M_1\sigma \uplus ... \uplus M_k\sigma$$

where each $t_i\sigma$ *is the usual first-order substitution and* $\uplus$ *is left-associative and defined as:*

$$\{\!\{t_1, ..., t_n | M_1, ..., M_k\}\!\} \uplus \{\!\{s_1, ..., s_m | N_1, ..., N_l\}\!\} = \{\!\{t_1, ..., t_n, s_1, ..., s_m | M_1, ..., M_k, N_1, ..., N_l\}\!\}$$

Definition 2 (Equality)**.** *Mmsets* $\mathcal{M} = \{\!\{t_1, ..., t_n | M_1, ..., M_k\}\!\}$ *and* $\mathcal{N} = \{\!\{s_1, ..., s_m | N_1, ..., N_l\}\!\}$ *are considered* equal modulo a constraint theory $\mathcal{T}$*, written* $\mathcal{M} =_{\mathcal{T}} \mathcal{N}$ *iff:* $n = m$ *and* $t_1, ..., t_n$ *is a permutation of* $s_1, ..., s_m$*; and* $\mathcal{T} \vdash M_1 \cup ... \cup M_k \equiv N_1 \cup ... \cup N_l$.

2 Mmsets Unification

The *mmset unification problem* of mmsets $\mathcal{M}_1$ and $\mathcal{M}_2$ consists of finding a substitution σ and constraint theory $\mathcal{T}_\sigma$ such that $\mathcal{M}_1\sigma =_{\mathcal{T}_\sigma} \mathcal{M}_2\sigma$. The theory $\mathcal{T}_\sigma$ consists of an equality over unions of multiset variables, and it is computed *a posteriori* for each unifier σ.

2.1 Algorithm

In what follows, we use σ to denote a single substitution, Σs to denote sets of substitutions, $\times$ for the Cartesian product of two sets (or lists), and $\setminus$ for multiset difference. The pseudo code for all algorithms are listed in Appendix A, and an implementation in SML can be found at https://github.com/meta-logic/mmset-unif.

The main function for mmset unification is implemented by Algorithm 1. In the most general case (lines 11 to 17), the unifiers of $\{\!\{\overline{t}|\overline{M}\}\!\}$ and $\{\!\{\overline{s}|\overline{N}\}\!\}$ are computed by choosing a subset of terms (of the same size) from $\overline{t}$ and $\overline{s}$ to be unified, and distributing the rest among the multiset variables $\overline{M}$ and $\overline{N}$. The number of terms chosen to be unified can vary from 0 to the minimum length of $\overline{t}$ and $\overline{s}$. Two other cases are considered separately for efficiency purposes. The first one is when there are no multiset variables (line 1). Here a unification is only possible if $|\overline{t}| = |\overline{s}|$. The second case is when one of the mmsets does not have multiset variables (lines 4 and 7). If $\overline{M}$ is empty, then all terms in $\overline{s}$ must be unified with a term from $\overline{t}$. The remaining terms in $\overline{t}$ can be allocated in $\overline{N}$.

Function unify_c (Algorithm 2) chooses c terms from the multisets $\overline{t}$ and $\overline{s}$ to be unified, and distributes the rest of the terms among the multiset variables. The function choose(F, c) returns a set of tuples (F_c, F_r), where F_c is the multiset with the chosen c elements (thus $|F_c| = c$), and $F_r = F \setminus F_c$. Unifiers for the chosen terms are computed by unify_terms and stored in Σ_t. Substitutions for each context variable, containing the remaining terms, are computed by unify_distribute and stored in Σ_M. The final set of unifiers consists of the composition $\sigma_M\sigma_t$ for each $(\sigma_M, \sigma_t) \in \Sigma_M \times \Sigma_t$. Note that, since the image of σ_M may contain terms, the composition needs to be in this order.

Function unify_terms (Algorithm 3) finds all unifiers of two multisets of terms (without multiset variables) of equal length. This is done by testing all possible pairings of the two multisets, obtained by pairing some order of the first multiset with all possible permutations of the second one. For each pairing, the function unify_lists computes the most general unifier.

Function unify_distribute (Algorithm 4) computes unifiers for the mmsets $\{\!|\overline{t}|\overline{M}|\!\}$ and $\{\!|\overline{s}|\overline{N}|\!\}$ considering that all $\overline{t}$ occurs in $\overline{N}$ and all $\overline{s}$ occurs in $\overline{M}$. Let Σ_N denote the substitutions that distribute $\overline{t}$ into $\overline{N}$, and Σ_M the substitutions that distribute $\overline{s}$ into $\overline{M}$. The resulting substitutions are $\sigma_M \cup \sigma_N$ for each $(\sigma_M, \sigma_N) \in \Sigma_M \times \Sigma_N$. A simple union can be used in the case, as the image of σ_M is disjoint from the domain of σ_N (see below).

Function distribute (Algorithm 5) is used by unify_distribute and that is where the aforementioned Σ_N and Σ_M are computed. It computes substitutions for the multiset variables $N_1, ..., N_l$ such that all terms $t_1, ..., t_n$ occur in one of the multisets. This is done by calculating all ordered l-partitions of the multiset $\{\!|t_1, ..., t_n|\!\}$. For example, the ordered 2-partitions of the multiset $\{\!|a, b, c|\!\}$ are:

$$\begin{array}{llll} [\{\!|a,b,c|\!\}, \{\!|\,|\!\}] & [\{\!|a,b|\!\}, \{\!|c|\!\}] & [\{\!|a|\!\}, \{\!|b,c|\!\}] & [\{\!|a,c|\!\}, \{\!|b|\!\}] \\ [\{\!|\,|\!\}, \{\!|a,b,c|\!\}] & [\{\!|c|\!\}, \{\!|a,b|\!\}] & [\{\!|b,c|\!\}, \{\!|a|\!\}] & [\{\!|b|\!\}, \{\!|a,c|\!\}] \end{array}$$

Each computed substitution corresponds to an l-ordered partition. If there are no terms ($n = 0$), then there is only the trivial partition of l empty multisets. In this case, the algorithm returns a list with one substitution, which maps every multiset variable N_i to the mmset with no terms. If there are no multiset variables ($l = 0$), then there are no partitions, and thus no possible substitutions. The exceptional case is when there are no terms nor multiset variables ($l = n = 0$). In this case, the solution is the set containing only the empty substitution ($\{\}$).

The last parameter of distribute indicates whether the multiset variables should contain *exactly* the terms $t_1, ..., t_n$. If set to true, then there is no more space for other terms, and N_i is mapped to an mmset with the appropriate terms and no multiset variables. Otherwise it is mapped to an mmset with a set of terms and a *fresh* multiset variable. If there are no terms to place in the multiset variable, it is mapped to itself (to avoid unecessary renamings). This is needed to compute the constraint theory, after which such identity substitutions can be eliminated.

Constraint theory For a unifier σ, the constraint theory $\mathcal{T}_\sigma$ is defined as:

$$\bigcup\{M'_i \mid M_i \mapsto \{\!|ts|M'_i|\!\} \in \sigma\} \equiv \bigcup\{N'_i \mid N_i \mapsto \{\!|ts|N'_i|\!\} \in \sigma\}$$

Soundness and completeness of the algorithm are straightforward, since it exhaustively checks all possibilities for unifying multisets.

Theorem 1. *Soundness If* $\mathsf{unify}(\mathcal{M}, \mathcal{N}) \mapsto \{\sigma_1, ..., \sigma_n\}$ *then* $\forall \sigma_i . \mathcal{M}\sigma_i =_{\mathcal{T}_{\sigma_i}} \mathcal{N}\sigma_i$

Theorem 2. *Completeness If* $\exists \sigma . \mathcal{M}\sigma =_{\mathcal{T}_\sigma} \mathcal{N}\sigma$ *then* $\mathsf{unify}(\mathcal{M}, \mathcal{N}) \mapsto \{\sigma_1, ..., \sigma_n\}$ *and* $\exists \sigma_i$ *such that* $\sigma = \sigma_i \sigma'$.

Note that the use of a substitution σ' is needed even if the set of computed unifiers is not the minimal one.

2.2 Complexity

The most expensive part of the unification algorithm is the one between lines 11 and 17 in Algorithm 1, so we concentrate our complexity analysis to that case. For each i from 0 to the minimum number of terms, unify_c is called. This function has two nested loops over the sets T_c and S_c (Alg. 2, lines 4, 5), which contain all possible ways of choosing i elements from n and m, respectively. Thus $|T_c| = \binom{n}{i}$ and $|S_c| = \binom{m}{i}$. In the inner part of the loops, unify_terms is called, which finds all possible unifiers for two multisets of size i. Since all possible pairings of elements must be tried, and for each order unify_lists runs in i^2, the function (Alg. 3) has complexity $i^2 i!$. The function unify_distribute (Alg. 4) computes all possible ways of partitioning $n-i$ elements into l parts, and $m-i$ elements into k parts, and returns the Cartesian product of these sets. Therefore, its complexity is $l^{n-i}\ k^{m-i}$.

Putting those together, we get to the cost for the unification of mmsets $\{\!\{t_1, ..., t_n | M_1, ..., M_k\}\!\}$ and $\{\!\{s_1, ..., s_m | N_1, ..., N_l\}\!\}$:

$$\sum_{i=0}^{\min(n,m)} \binom{n}{i}\binom{m}{i}\ i^2\ i!\ l^{n-i}\ k^{m-i}$$

After some arithmetic manipulation, we can conclude that, on the worst case, unify runs in $O(n!\ m!\ l^n\ k^m)$. For the special case where the multisets have only one multiset variable: $l = k = 1$, and the unification algorithm runs in $O(n!\ m!)$.

2.3 Removing Less General Unifiers

The algorithm described in Section 2.1 does not compute the set of minimal unifiers. For example, given multiset $\{\!\{a, a | M\}\!\}$ and $\{\!\{a | N\}\!\}$, unify computes three unifiers with constraint: $\{N \mapsto \{\!\{a | N'\}\!\}\}$, where $M \equiv N'$, twice (once for each occurrence of a), and $\{M \mapsto \{\!\{a | M'\}\!\}\ ;\ N \mapsto \{\!\{a, a | N'\}\!\}\}$, where $M' \equiv N'$. These are the same unifiers obtained by the non-deterministic algorithm from [5].

In order to reduce the number of less general unifiers, we can remove every pair of equal terms t_i and s_j from the mmsets (i.e., t_i and s_j unify with the empty substitution). The rationale behind this is that, every other unifier that is obtained by unifying these terms with something else, or placing them in a multiset variable, can be recovered from the set of unifiers obtained when this pair is not in the mmset.

We start by showing that it is safe to eliminate pairs of equal terms from the problem of unifying multisets without multiset variables.

Theorem 3. *Let $\overline{t}$ and $\overline{s}$ be two multisets of terms such that $t_a \in \overline{t}$ and $s_b \in \overline{s}$ are equal, for some a and b. Let $\Sigma_{all} = \mathsf{unify\_terms}(\overline{t}, \overline{s})$, and $\Sigma = \mathsf{unify\_terms}(\overline{t} \backslash \{\!\{t_a\}\!\}, \overline{s} \backslash \{\!\{s_b\}\!\})$. Then for every $\sigma \in \Sigma_{all}$, there exists $\mu \in \Sigma$ s.t. $\sigma = \mu\sigma'$ for some substitution σ'.*

The proof for this theorem can be found in Appendix B. The overall idea is as follows. σ was obtained by some pairing of terms in $\overline{t}$ and $\overline{s}$. We choose μ as the unifier that used a pairing that is as close as possible as the one used for σ. Those pairings differ only for the terms involving t_a and s_b. Suppose t_a is paired with s_x and s_b is paired with t_y. Using the most general unifiers of t_y and s_x, we can conclude the existence of σ' such that $\sigma = \mu\sigma'$.

Theorem 4. *Let $\{\!\{\overline{t} | \overline{M}\}\!\}$ and $\{\!\{\overline{s} | \overline{N}\}\!\}$ be two mmsets such that $t_a \in \overline{t}$ is equal to $s_b \in \overline{s}$ for some a and b. Moreover, let $\Sigma_{all} = \mathsf{unify}(\{\!\{\overline{t} | \overline{M}\}\!\}, \{\!\{\overline{s} | \overline{N}\}\!\})$, and $\Sigma = \mathsf{unify}(\{\!\{\overline{t} \backslash \{\!\{t_a\}\!\} | \overline{M}\}\!\}, \{\!\{\overline{s} \backslash \{\!\{s_b\}\!\} | \overline{N}\}\!\})$. Then for every $\sigma \in \Sigma_{all}$ there exists $\mu \in \Sigma$ such that $\sigma = \mu\sigma'$ for some σ'.*

The proof for this theorem can be found in Appendix B. We proceed by a case analysis on whether t_a and s_b were chosen to be unified, or to be placed in a multiset variable. There are four cases. The case in which both are chosen to be unified is solved using Theorem 3. For the case in which both are placed in a multiset variable, we can construct σ'. The other two (dual) cases are the more involved ones. They use a combination of the two strategies of the first cases.

This modification is implemented in the algorithm available online, and extensive testing has shown that all less general unifiers are eliminated. In particular, only the unifier $\{N \mapsto \{\!|a|N'|\!\}\}$, where $M \equiv N'$ is computed for mmsets $\{\!|a, a|M|\!\}$ and $\{\!|a|N|\!\}$.

3 Conclusion

We have developed a sound and complete algorithm for finding unifiers of multisets with multiple multiset variables. The algorithm is deterministic and terminating. It is implemented in SML, and we also provide the pseudo code for reproducibility. The same algorithm can be used for the particular case where there is only one multiset variable.

The complexity of the unification procedure is analysed, and its cost is high. This is inherent to the problem, since it is of combinatorial nature. It may be possible to improve this result by using the right data-structures and heuristics, but for our purposes, since the numbers are quite small, it runs fast enough.

We have also tried to eliminate all sources of redundancy, so that the set of computed unifiers is as close as possible to the minimal one. In particular, we have shown that a simple pre-processing of the input problem will produce fewer unifiers, and all those that are no longer produced can be recovered. We conjecture that this optimization leads to the computation of the *minimal* set of unifiers, but we leave this investigation as future work.

References

[1] I. Cervesato. Solution Count for Multiset Unification with Trailing Multiset Variables. In C. Ringeissen, C. Tinelli, F. Trinen, and R. Verma, editors, *Sixteenth International Workshop on Unification — UNIF'02*, pages 64–68, 2002.

[2] K. Chaudhuri, L. Lima, and G. Reis. Formalized Meta-Theory of Sequent Calculi for Substructural Logics. *Electronic Notes in Theoretical Computer Science*, 332:57 – 73, 2017. LSFA 2016 - 11th Workshop on Logical and Semantic Frameworks with Applications (LSFA).

[3] J. E. Dawson and R. Goré. Generic Methods for Formalising Sequent Calculi Applied to Provability Logic. In *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, 2010. Proceedings*, pages 263–277, 2010.

[4] A. Dovier, A. Policriti, and G. Rossi. Integrating Lists, Multisets, and Sets in a Logic Programming Framework. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems: First International Workshop, Munich, March 1996*, pages 303–319. Springer Netherlands, 1996.

[5] A. Dovier, A. Policriti, and G. Rossi. A uniform axiomatic view of lists, multisets, and sets, and the relevant unification algorithms. *Fundam. Inf.*, 36(2-3):201–234, 1998.

[6] H. Tews. Formalizing Cut Elimination of Coalgebraic Logics in Coq. In *Automated Reasoning with Analytic Tableaux and Related Methods: 22nd International Conference, TABLEAUX 2013, Proceedings*, pages 257–272. Springer, 2013.

[7] B. Xavier, C. Olarte, G. Reis, and V. Nigam. Mechanizing Linear Logic in Coq. In *Proceedings of the 12^{th} Workshop on Logical and Semantics Frameworks with Applications (LSFA)*, pages 60–77, 2017.

A Algorithms

Algorithm 1 unify($\{\!\{t_1, ..., t_n | M_1, ..., M_k\}\!\}, \{\!\{s_1, ..., s_m | N_1, ..., N_l\}\!\}$)

if $k = 0 \wedge l = 0$ **then**
  if $n = m$ **then return** unify_terms($[t_1, ..., t_n], [s_1, ..., s_m]$)
  else return []
else if $k = 0$ **then**
  if $m \leq n$ **then return** unify_c($m, \{\!\{t_1, ..., t_n | M_1, ..., M_k\}\!\}, \{\!\{s_1, ..., s_m | N_1, ..., N_l\}\!\}$)
  else return []
else if $l = 0$ **then**
  if $n \leq m$ **then return** unify_c($n, \{\!\{t_1, ..., t_n | M_1, ..., M_k\}\!\}, \{\!\{s_1, ..., s_m | N_1, ..., N_l\}\!\}$)
  else return []
else
  $c \leftarrow \min(n, m)$
  $\Sigma \leftarrow []$
  for $i = 0$ **to** c **do**
    $\Sigma' \leftarrow$ unify_c($i, \{\!\{t_1, ..., t_n | M_1, ..., M_k\}\!\}, \{\!\{s_1, ..., s_m | N_1, ..., N_l\}\!\}$)
    $\Sigma \leftarrow \Sigma \cup \Sigma'$
  end for
  return Σ
end if

Algorithm 2 unify_c($c, \{\!\{t_1, ..., t_n | M_1, ..., M_k\}\!\}, \{\!\{s_1, ..., s_m | N_1, ..., N_l\}\!\}$)

$\Sigma \leftarrow []$
$T_c \leftarrow$ choose($\{\!\{t_1, ..., t_n\}\!\}, c$)
$S_c \leftarrow$ choose($\{\!\{s_1, ..., s_m\}\!\}, c$)
for $(\{\!\{t'_1, ..., t'_c\}\!\}, \{\!\{t'_{c+1}, ..., t'_n\}\!\}) \in T_c$ **do**
  for $(\{\!\{s'_1, ..., s'_c\}\!\}, \{\!\{s'_{c+1}, ..., s'_m\}\!\}) \in S_c$ **do**
    $\Sigma_t \leftarrow$ unify_terms($[t'_1, ..., t'_c], [s'_1, ..., s'_c]$)
    if $\Sigma_t \neq []$ **then**
      $\Sigma_M \leftarrow$ unify_distribute($(\{\!\{t'_{c+1}, ..., t'_n\}\!\}, \{M_1, ..., M_k\}), (\{\!\{s'_{c+1}, ..., s'_m\}\!\}, \{N_1, ..., N_l\})$)
      $\Sigma' \leftarrow$ map($\lambda(\sigma_t, \sigma_M).\sigma_M\sigma_t)(\Sigma_t \times \Sigma_M)$
      $\Sigma \leftarrow \Sigma \cup \Sigma'$
    end if
  end for
end for
return Σ

Algorithm 3 unify_terms($[t_1, ..., t_n], [s_1, ..., s_n]$)

$\Sigma \leftarrow []$
$P_s \leftarrow$ permutations($[s_1, ..., s_n]$)
$P \leftarrow [[t_1, ..., t_n]] \times P_s$
for $(T, S) \in P$ **do**
  $\sigma \leftarrow$ unify_lists(T, S)
  if $\sigma \neq$ None **then** $\Sigma \leftarrow \{\sigma\} \cup \Sigma$
end for

Algorithm 4 unify_distribute($(\{\!\{t_1, ..., t_n\}\!\}, \{M_1, ..., M_k\}), (\{\!\{s_1, ..., s_m\}\!\}, \{N_1, ..., N_l\})$)

$\Sigma_N \leftarrow$ distribute$(\{\!\{t_1, ..., t_n\}\!\}, \{N_1, ..., N_l\}, k = 0)$ {List of substitutions for N_i}
$\Sigma_M \leftarrow$ distribute$(\{\!\{s_1, ..., s_m\}\!\}, \{M_1, ..., M_k\}, l = 0)$ {List of substitutions for M_i}
$\Sigma \leftarrow$ map$(\lambda(\sigma_M, \sigma_N).\sigma_M \cup \sigma_N)(\Sigma_M \times \Sigma_N)$
return Σ

Algorithm 5 distribute($\{\!\{t_1, ..., t_n\}\!\}, \{M_1, ..., M_k\}$, exact)

if $n = 0 \wedge k = 0$ **then**
    return $[\{\}]$
end if
$\Sigma \leftarrow []$
$P_t \leftarrow$ ordered_partitions$(\{\!\{t_1, ..., t_n\}\!\}, k)$
for $p \in P_t$ **do**
    $\sigma \leftarrow \{\}$
    for $i = 1$ **to** k **do**
        $ts \leftarrow p[i]$
        if exact **then** $\sigma \leftarrow \sigma\{M_i \mapsto \{\!\{ts| \cdot \}\!\}\}$
        if $\neg$exact $\wedge\ ts = \emptyset$ **then** $\sigma \leftarrow \sigma\{M_i \mapsto \{\!\{ \cdot |M_i\}\!\}\}$
        if $\neg$exact $\wedge\ ts \neq \emptyset$ **then** $\sigma \leftarrow \sigma\{M_i \mapsto \{\!\{ts|M_i'\}\!\}\}$
    end for
    $\Sigma \leftarrow \{\sigma\} \cup \Sigma$
end for
return Σ

B Proofs

Proof for Theorem 3. We know that Σ_{all} contains at most $n!$ unifiers, one for each way of pairing elements of $\overline{t}$ with elements of $\overline{s}$. Analogously, Σ contains at most $(n-1)!$ unifiers. Let: $\Sigma_{\text{all}} = \{\sigma_1, ..., \sigma_{n!}\}$ and $\Sigma = \{\mu_1, ..., \mu_{(n-1)!}\}$. Then each σ_i can be obtained from some μ_j.

If σ_i is a unifier resulting from pairing t_a with s_b, then there exists $\mu_j = \sigma_i$ and we are done.

Let σ_i be a unifier resulting from pairing t_a with some s_x, s_b with some t_y, and some permutation P_t of $\overline{t} \backslash \{\!\{t_a, t_y\}\!\}$ with some permutation P_s of $\overline{s} \backslash \{\!\{s_b, s_x\}\!\}$. There exists a unifier $\mu_j \in \Sigma$ that is the result of unifying the same permutations P_t and P_s, and t_y with s_x. We show how σ_i can be reconstructed from μ_j. Since the order in which terms are unified does not matter, we assume that σ_i and μ_j are obtained as follows:

1. Computation of σ_i:
   (a) mgu σ_{ax} of t_a and s_x
   (b) mgu σ_{by} of $t_y\sigma_{ax}$ and $s_b\sigma_{ax}$
   (c) mgu σ_{P_i} of $P_t\sigma_{ax}\sigma_{by}$ and $P_t\sigma_{ax}\sigma_{by}$

2. Computation of μ_j:
   (a) mgu σ_{xy} of t_y and s_x
   (b) mgu σ_{P_j} of $P_t\sigma_{xy}$ and $P_s\sigma_{xy}$

Therefore $\sigma_i = \sigma_{ax}\sigma_{by}\sigma_{P_i}$ and $\mu_j = \sigma_{xy}\sigma_{P_j}$. We know that:

$$t_a\sigma_{ax} = s_x\sigma_{ax} \quad \text{from 1a} \tag{1}$$
$$t_y\sigma_{ax}\sigma_{by} = s_b\sigma_{ax}\sigma_{by} \quad \text{from 1b} \tag{2}$$
$$t_y\sigma_{ax}\sigma_{by} = t_a\sigma_{ax}\sigma_{by} \quad \text{because } t_a = s_b \tag{3}$$
$$t_y\sigma_{ax}\sigma_{by} = s_x\sigma_{ax}\sigma_{by} \quad \text{from 3 and 1} \tag{4}$$

Thus, $\sigma_{ax}\sigma_{by}$ is a unifier of t_y and s_x. But from 2a we have that σ_{xy} is the most general unifiers of these terms, which means that there exists a σ' such that $\sigma_{ax}\sigma_{by} = \sigma_{xy}\sigma'$ (5).

From 5 and 1c, we know that $P_t\sigma_{xy}\sigma'\sigma_{P_i} = P_s\sigma_{xy}\sigma'\sigma_{P_i}$, meaning that $\sigma'\sigma_{P_i}$ is a unifier of $P_t\sigma_{xy}$ and $P_s\sigma_{xy}$. But from 2b, σ_{P_j} is the most general unifier of these two lists, therefore, there exists σ'' such that $\sigma'\sigma_{P_i} = \sigma_{P_j}\sigma''$ (6).
Using 5, 6, and associativity of substitution composition: $\sigma_i = \sigma_{ax}\sigma_{by}\sigma_{P_i} = \sigma_{xy}\sigma_{P_j}\sigma'' = \mu_j\sigma''$
□

Proof for Theorem 4. We case on how t_a and s_b were used to compute σ. Let $\overrightarrow{t_c}$ and $\overrightarrow{s_c}$ denote the terms *and order* chosen from $\overline{t}$ and $\overline{s}$, respectively, to be unified. Let $\overline{t_r}$ and $\overline{s_r}$ denote the rest of the terms in $\overline{t}$ and $\overline{s}$ that will be distributed to $\overline{N}$ and $\overline{M}$, respectively.

We know that $\sigma = \sigma_M\sigma_t$ (Alg. 2, line 9), where σ_t is the unifier of $\overrightarrow{t_c}$ and $\overrightarrow{s_c}$, and σ_M is obtained from partitions denoted as $\pi_t(\overline{t_r})$ and $\pi_s(\overline{s_r})$.

1. $\boxed{t_a \in \overrightarrow{t_c} \text{ and } s_b \in \overrightarrow{s_c}}$
   Take $\mu = \mu_M\mu_t$ such that $\mu_M = \sigma_M$ is obtained from the same partitions $\pi_t(\overline{t_r})$ and $\pi_s(\overline{s_r})$, and μ_t is such that $\sigma_t = \mu_t\sigma'$ for some σ'. The existence of such μ_t is guaranteed by Theorem 3. Therefore, $\sigma = \sigma_M\sigma_t = \mu_M\mu_t\sigma' = \mu\sigma'$.
2. $\boxed{t_a \in \overline{t_r} \text{ and } s_b \in \overline{s_r}}$
   Let $\overline{t_{p_a}}, t_a$ be the part from $\pi_t(\overline{t_r})$ with t_a, and $\overline{s_{p_b}}, s_b$ the part from $\pi_s(\overline{s_r})$ with s_b.
   Take $\mu = \mu_M\mu_t$ such that $\mu_t = \sigma_t$ is the unifier of $\overrightarrow{t_c}$ and $\overrightarrow{s_c}$, and μ_M is obtained from partitions $\pi_t(\overline{t_r})$ where $\overline{t_{p_a}}, t_a$ is replaced by $\overline{t_{p_a}}$ and analogously for $\pi_s(\overline{s_r})$. Thus the mappings in σ_M and μ_M are the same, except for two multiset variables N_a and M_b:
   $\{M_b \mapsto \overline{s_{p_b}}, s_b, M_b'; N_a \mapsto \overline{t_{p_a}}, t_a, N_a'\} \subset \sigma_M$ and $\{M_b \mapsto \overline{s_{p_b}}, M_b'; N_a \mapsto \overline{t_{p_a}}, N_a'\} \subset \mu_M$.
   Since M_b' and N_a' are fresh multiset variable names: $\sigma_M = \mu_M\{M_b' \mapsto s_b, M_b' \ ; \ N_a' \mapsto t_a, N_a'\}$.
   And since $\sigma_t = \mu_t$: $\sigma_M\sigma_t = \mu_M\{M_b' \mapsto s_b, M_b' \ ; \ N_a' \mapsto t_a, N_a'\}\mu_t$. The image of μ_t does not contain M_b' nor N_a', therefore: $\sigma_M\sigma_t = \mu_M\mu_t\{M_b' \mapsto s_b, M_b' \ ; \ N_a' \mapsto t_a, N_a'\}\mu_t$. Thus: $\sigma = \mu\{M_b' \mapsto s_b, M_b' \ ; \ N_a' \mapsto t_a, N_a'\}\mu_t$.
3. $\boxed{t_a \in \overrightarrow{t_c} \text{ and } s_b \in \overline{s_r}}$
   Let s_k be the term from $\overrightarrow{s_c}$ that is paired with t_a and $\overline{s_{p_b}}, s_b$ be the part from $\pi_s(\overline{s_r})$ containing s_b. Assume that t_a is unified with s_k with mgu σ_{ak}, and that $(\overrightarrow{t_c}\backslash\{\!\{t_a\}\!\})\sigma_{ak}$ unifies with $(\overrightarrow{s_c}\backslash\{\!\{s_k\}\!\})\sigma_{ak}$ with mgu σ_c. Thus $\sigma_t = \sigma_{ak}\sigma_c$. Let M_b be the variable to which partition $\overline{s_{p_b}}, s_b$ is assigned. Thus:

$$\{M_b \mapsto \overline{s_{p_b}}, s_b, M_b'\} \subset \sigma_M \qquad \text{By definition} \qquad (1)$$
$$\{M_b \mapsto \overline{s_{p_b}}\sigma_{ak}, s_b\sigma_{ak}, M_b'\} \subset \sigma_M\sigma_{ak} \qquad \text{Composition with } \sigma_{ak} \qquad (2)$$
$$\{M_b \mapsto \overline{s_{p_b}}\sigma_{ak}, t_a\sigma_{ak}, M_b'\} \subset \sigma_M\sigma_{ak} \qquad t_a = s_b \qquad (3)$$
$$\{M_b \mapsto \overline{s_{p_b}}\sigma_{ak}, s_k\sigma_{ak}, M_b'\} \subset \sigma_M\sigma_{ak} \qquad t_a\sigma_{ak} = s_k\sigma_{ak} \qquad (4)$$

   Take $\mu = \mu_M\mu_t$ such that the terms $\overrightarrow{t_c}\backslash\{\!\{t_a\}\!\}$ and $\overrightarrow{s_c}\backslash\{\!\{s_k\}\!\}$ are unified with mgu μ_t, and μ_M is computed using partition $\pi_t(\overline{t_r})$ and $\pi_s(\overline{s_r})$ where part $\overline{s_{p_b}}, s_b$ is replaced by $\overline{s_{p_b}}, s_k$. Therefore, $\{M_b \mapsto \overline{s_{p_b}}, s_k, M_b'\} \subset \mu_M$ (5). Because μ_t is the mgu of the two lists, we have that: $\sigma_t = \sigma_{ak}\sigma_c = \mu_t\sigma_t'$ for some σ_t'. And from 4 and 5 we can also conclude: $\sigma_M\sigma_{ak} = \mu_M\sigma_{ak}$. Using these equalities: $\sigma = \sigma_M\sigma_t = \sigma_M\sigma_{ak}\sigma_c = \mu_M\sigma_{ak}\sigma_c = \mu_M\mu_t\sigma_t' = \mu\sigma_t'$.
4. $\boxed{t_a \in \overline{t_r} \text{ and } s_b \in \overrightarrow{s_c}}$ Analogous to the previous case.

□

Formalising Nominal AC-Unification

Mauricio Ayala-Rincón[1,3]*, Maribel Fernández[2], and Gabriel Ferreira Silva[3]†

[1] Department of Computer Science, Universidade de Brasília
ayala@unb.br
[2] Department of Informatics, Kings College London
maribel.fernandez@kcl.ac.uk
[3] Department of Mathematics, Universidade de Brasília
gabrielfsilva1995@gmail.com

Abstract

The nominal setting extends first-order syntax and represents smoothly systems with variable bindings, using instead of variables, nominal atoms and atom permutations for renaming them. Nominal unification adapts first-order unification modulo α-equivalence by taking into account this nominal approach. Nominal AC-unification is then simply nominal unification with associative-commutative function symbols. In this paper, we present a functional specification of a nominal AC-unification algorithm and discuss relevant aspects of current work on formalising its soundness and completeness. The algorithm explores the combinatory of the problem without taking into consideration efficiency, simplifying, in this manner, the formalisation.

1 Introduction

The nominal setting allows us to extend first-order syntax and represent smoothly systems with bindings, which are frequent in mathematics and computer science. Nevertheless, to represent these bindings correctly, α-equivalence must be taken into account. For instance, despite their syntactical difference, the formulas $\exists y : y > 0$ and $\exists z : z > 0$ should be considered equivalent. The nominal theory allows us to deal with these bindings in a natural way, instead of using indices as in explicit substitutions *à la de Bruijn* (e.g. [8], [7]).

1.1 Related Work

Nominal Unification was originally solved by Urban, Pitts and Gabbay [10], who proposed a set of inference rules to compute the most general unifier of a (solvable) nominal unification problem. The rules were formalised and proved to be correct and complete with the help of the proof assistant Isabelle/HOL [9]. A functional nominal unification algorithm was later formalised in PVS and proved correct and complete [4]. Nominal unification was extended to take into account commutative axioms [1]: a nominal C-unification algorithm given as a set of inference rules was proposed and formalised sound and complete in Coq. Based on the previous two papers, a functional algorithm for nominal C-unification was specified and verified in PVS [2]. In the standard first-order syntax, the first formalisation of AC-matching, introduced in [6], opens the way for formalisations of AC-unification, which is (to the best of our knowledge) yet to come.

*Author partially funded by CNPq research grant number 307672/2017-4.
†Author partially funded by CNPq scholarship scholarship number 139271/2017-1.

1.2 Contributions and Possible Applications

In this extended abstract, we discuss a functional specification of a nominal AC-unification algorithm pointing out interesting aspects of its formalisation in PVS. Although the formalisation is not yet complete, the specification of the algorithm is finished and fully available at: www.github.com/gabriel951/ac-unification. The work here presented, when completed, would not only give the first formalisation of nominal AC-unification but also, as far as we know, the first formalisation of first-order AC-unification, since the nominal theory encompasses first-order theory.

Since unification has applications in logic programming systems, type inference algorithms, theorem provers and so on, a nominal AC-unification algorithm has interesting potential uses. It could, for instance, be used in a logic programming language that employs the nominal setting, such as α-Prolog (see [5]). Another possibility is to translate the algorithm into an AC-matching algorithm. Since matching two terms t and s can be seen as unification where one of the terms (suppose t, without loss of generality) is not affected by a substitution, the translation to AC-matching would be performed by marking all variables that occur in t as "protected" variables at the beginning of the matching process and adapting the algorithm to prohibit the instantiation of "protected" variables. These AC algorithms could then be used to extend nominal rewriting [7] or nominal narrowing [3].

2 Preliminaries

Only the part of nominal theory relevant to unifying AC function symbols is explained. For a complete account, one can check, for instance, [4] or [3].

2.1 Nominal Terms, Permutations and Substitutions

In nominal theory, we have a countable set of atoms $\mathcal{A} = \{a, b, c, ...\}$ and a countable set of variables $\mathcal{X} = \{X, Y, Z, ...\}$, which are disjoint. A permutation π is a bijection of the form $\pi : \mathcal{A} \to \mathcal{A}$ such that the set of atoms that are modified by π is finite.

Definition 1 (Nominal Terms)**.** *Let Σ be a signature with function symbols and AC function symbols. The set $\mathcal{T}(\Sigma, \mathcal{A}, \mathcal{X})$ of nominal terms is generated according to the grammar:*

$$s, t \quad ::= \quad \langle\rangle \;\mid\; \overline{a} \;\mid\; \pi \cdot X \;\mid\; [a]t \;\mid\; \langle s, t\rangle \;\mid\; f\ t \;\mid\; f^{AC}\ t$$

where $\langle\rangle$ is the unit, $\overline{a}$ is an atom term, $\pi \cdot X$ is a suspended variable (the permutation π is suspended on the variable X), $[a]t$ is an abstraction (a term with the atom a abstracted), $\langle s, t\rangle$ is a pair, $f\ t$ is a function application and $f^{AC}\ t$ is an AC function application.

Remark. Although the function application has arity one, this is not a limitation, for we can use the pair to encode tuples with an arbitrary number of arguments. For instance, the tuple (t_1, t_2, t_3) could be constructed as $\langle t_1, \langle t_2, t_3\rangle\rangle$.

Finally, a substitution σ in the nominal setting is analogous to the concept in first-order theory: a mapping that sends a finite amount of variables in $\mathcal{X}$ to terms in $\mathcal{T}$.

2.2 Freshness and α-Equality

Two key notions in nominal theory are freshness and α-equality, represented, respectively, by the predicates $\#$ and $\approx_\alpha$:

- $a\#t$ means that if a occurs in t then it does so under an abstractor [a].
- $s \approx_\alpha t$ means that s and t are α-equivalent.

After explaining the ideas behind these two concepts, we define them formally for AC function symbols.

Definition 2 (Freshness)**.** *A freshness context ∇ is a set of constraints of the form $a\#X$. An atom a is said to be fresh on t under a context ∇ (which we denote by $\nabla \vdash a\#t$) if it is possible to build a proof using the rules for freshness, in accordance with the type of t (see [7]). The rule for freshness of an AC function application is the same for a function application.*

Definition 3 (α-equality with AC operators)**.** *Two terms t and s are said to be α-equivalent under the context Δ, written as $\Delta \vdash t \approx_\alpha s$, if it is possible to build a proof using the rules for α-equivalence, in accordance with the type of t (see [7]). The rule for an AC function symbol is:*

$$\frac{\Delta \vdash S_1(fs) \approx_\alpha S_i(ft) \qquad \Delta \vdash D_1(fs) \approx_\alpha D_i(ft)}{\Delta \vdash fs \approx_\alpha ft}\ (\approx_\alpha \texttt{ac-app})$$

for some i. Here f is an AC function symbol, $S_n(f*)$ is an operator that selects the nth argument of the flattened subterm $f*$ and $D_n(f*)$ is an operator that deletes the nth argument of the flattened subterm $f*$.

Remark. If the flattened subterm $f*$ contains only two arguments, then $D_n(f*)$ will contain only one argument. Also, if the flattened subterm $f*$ contains only one argument, then $D_n(f*)$ returns the unit. For instance, $D_1(f\langle a, b\rangle) = fb$ and $D_1(fb) = \langle\rangle$.

Example 1. *Let f be an AC-function symbol. In the above definition, $S_2(f\langle f\langle a, b\rangle, f\langle [a]X, \pi \cdot Y\rangle\rangle)$ is b, and $D_2(f\langle f\langle a, b\rangle, f\langle [a]X, \pi \cdot Y\rangle\rangle) = f\langle fa, f\langle [a]X, \pi \cdot Y\rangle\rangle\rangle$.*

2.3 Nominal AC-Unification

Definition 4 (Unification Problem)**.** *A unification problem is a pair $\langle \Delta, P\rangle$ where Δ is a freshness context and P is a finite set of equations and freshness constraints of the form $s \approx_? t$ and $a\#_? t$, respectively, with s and t terms and a an atom.*

Example 2. *An example of a unification problem with an empty context and one equation constraint: $\langle \Delta, P\rangle = \langle \emptyset,\ f^{AC}\langle f^{AC}\langle X, Y\rangle, c\rangle\rangle \approx_?\ f^{AC}\langle c, f^{AC}\langle a, b\rangle\rangle\rangle$*

Remark. Let ∇ and ∇' freshness contexts and σ and σ' substitutions. In order to define a solution to a unification problem, we need the following notation:

- $\nabla' \vdash \nabla\sigma$ denotes that $\nabla' \vdash a\#X\sigma$ holds for each $(a\#X) \in \nabla$.
- $\nabla \vdash \sigma \approx \sigma'$ denotes that $\nabla \vdash X\sigma \approx_\alpha X\sigma'$ for all X in $dom(\sigma) \cup dom(\sigma')$.

Definition 5 (Solution for a Triple or Problem)**.** *A solution for a triple $\mathcal{P} = \langle \Delta, \delta, P\rangle$ is a pair $\langle \nabla, \sigma\rangle$ that fulfills the following four conditions:*

- $\nabla \vdash \Delta\sigma$
- $\nabla \vdash s\sigma \approx_\alpha t\sigma$, *if* $s \approx_? t \in P$
- $\nabla \vdash a\#t\sigma$, *if* $a\#_? t \in P$
- *There exist λ such that* $\nabla \vdash \delta\lambda \approx \sigma$

Then, a solution for a unification problem $\langle \Delta, P\rangle$ is a solution for the associated triple $\langle \Delta, id, P\rangle$.

Remark. As in C-Unification, in AC-unification equations of the form $\pi \cdot X \approx_\alpha \pi' \cdot X$, called fixed point equations, are not solved because there is an infinite number of solutions to them. Instead, they are carried on, as part of the solution to the unification problem [1].

Example 3. *Consider the unification problem of Example 2:* $\langle \Delta, id, P\rangle = \langle \emptyset, id,\ f^{AC}\langle f^{AC}\langle X, Y\rangle, c\rangle \approx_?\ f^{AC}\langle c, f^{AC}\langle a, b\rangle\rangle\rangle$. *A possible solution is* $\langle \emptyset, \{X \to a, Y \to b\}\rangle$

3 Specification

We specified a functional nominal AC-unification algorithm for unifying two terms t and s. The algorithm is recursive, calling itself on progressively simpler versions of the problem until it has finished. It is an extension of the algorithm in [2], in order to deal with the case of t or s being rooted by an AC function symbol. See also Appendix A.

When one of t or s is rooted by an AC function symbol and the other term is a suspended variable, the algorithm instantiates the suspended variable term appropriately and solves the problem. Alternatively, when one of the terms is rooted by an AC function symbol and the other is not a suspended variable or rooted by the same AC function symbol, the algorithm recognises it is a situation where no solution is possible. The interesting case is, therefore, when both t and s are rooted by the same AC function symbol.

In this situation, the algorithm first extracts all arguments of t and then generates all *pairings* of those arguments, in any order. After that, the algorithm extracts all arguments of s and then generates all pairings of these arguments, again in any order. Finally, the algorithm tries to unify every pairing of arguments of t with every pairing of arguments of s.

Example 4. *Suppose we are trying to unify the terms in Example 2.*

- *The two pairings generated for the term* $f^{AC}\langle f^{AC}\langle X, Y\rangle, c\rangle$ *in the order* (X, Y, c) *are:* $\langle X, \langle Y, c\rangle\rangle$ *and* $\langle\langle X, Y\rangle, c\rangle$. *Two pairings would be generated for every order and the possible orders are:* (X, Y, c), (X, c, Y), (Y, X, c), (Y, c, X), (c, X, Y) *and* (c, Y, X).
- *The twelve pairings generated for the term* $f^{AC}\langle c, f^{AC}\langle a, b\rangle\rangle$ *include, for instance:* $\langle c, \langle a, b\rangle\rangle$, $\langle\langle c, a\rangle, b\rangle$, $\langle\langle b, c\rangle, a\rangle$ *and* $\langle a, \langle b, c\rangle\rangle$.

Remark. Our first idea to deal with the pairings of the arguments of t and s was to generate all pairings of the arguments of t, preserving the order and all pairings of the arguments of s, in any order. This approach, however, is not complete.

To see that, consider f an AC function symbol, $t = f\langle a, \langle b, c\rangle\rangle$ and $s = f\langle X, b\rangle$. The substitution $\sigma = \{X \to \langle a, c\rangle\}$ would not be found if we had generated only the pairings of the arguments in t preserving the order, but it can be found by our approach. That is because the substitution σ is found when trying to unify $\langle\langle a, c\rangle, b\rangle$ with $\langle X, b\rangle$ and the arguments of the pairing $\langle\langle a, c\rangle, b\rangle$ are not in the same order that they are in t.

4 Formalisation

Theorems 1 and 2 formalise soundness and completeness of unifying AC functions. The function `gen_unif_prb`(ft, fs), for f AC, generates all pairings of ft (in any order) and all pairings of fs (in any order) and then combines them to generate a list of unification problems.

Theorem 1 (Soundness of Unifying AC functions)**.** *Let* ft *and* fs *be AC function applications. Suppose that* $(t_1, s_1) \in$ `gen_unif_prb`(ft, fs) *and that* $\nabla \vdash t_1\sigma \approx_\alpha s_1\sigma$. *Then,* $\nabla \vdash (ft)\sigma \approx_\alpha (fs)\sigma$.

Theorem 2 (Completeness of Unifying AC functions)**.** *Let ft and fs be AC function applications. Suppose that $\nabla \vdash (ft)\sigma \approx_\alpha (fs)\sigma$. Then, there exists $(t_1, s_1) \in \texttt{gen\_unif\_prb}(ft, fs)$ such that $\nabla \vdash t_1\sigma \approx_\alpha s_1\sigma$.*

The natural way of proving the theorem of soundness would be by induction on the size of the term. If we had decided to prove the theorem directly, we would find the i that makes $\nabla \vdash S_1((ft)\sigma) \approx_\alpha S_i((fs)\sigma)$ and then try to use the induction hypothesis to prove that $\nabla \vdash D_1((ft)\sigma) \approx_\alpha D_i((fs)\sigma)$. We would not, however, be able to apply the induction hypothesis, since the term being deleted of $t\sigma$ could be the first term of t but it could have also being introduced by the substitution σ. A similar problem could happen for s. To get around this problem, we must first eliminate the substitutions from our problem, and then solve a version of the problem without substitutions by induction. As explained next, Lemmas 1 and 2, together with a convenient renaming of variables, are used to eliminate the substitution, while Lemma 3 solves a simplified version of the problem.

For taking substitutions out of the equation, we will need the operator F_{AO}, which generates all possible flattened versions of a term, in any order. Therefore, after applying this operator, we get a term that is not an AC function application.

Lemma 1. *Let ft and fs be AC applications. Suppose that $(t_1, s_1) \in \texttt{gen\_unif\_prb}(ft, fs)$. Then, $\forall t'_1 \in F_{AO}(t_1\sigma),\ s'_1 \in F_{AO}(s_1\sigma)$: $(t'_1, s'_1) \in \texttt{gen\_unif\_prb}(ft\sigma, fs\sigma)$.*

Remark. The operator F_{AO} is needed in the Lemma 1 because, although we have the guarantee that t_1 and s_1 are pairings of ft and fs, the substitution σ may reintroduce the AC function symbol f into the terms $t_1\sigma$ and $s_1\sigma$. Since an output of $\texttt{gen\_unif\_prb}((ft)\sigma, (fs)\sigma)$ cannot contain the AC function symbol, we must apply the flattener operator F_{AO} to $t_1\sigma$ and $s_1\sigma$. A case where this reintroduction of the AC function symbol occurs is illustrated in Example 5.

Example 5. *Let f be an AC function symbol. Consider $ft = fX$, $fs = fY$ and $\sigma = \{X \to f\langle a, b\rangle, Y \to f\langle b, a\rangle\}$. Then, $t_1 = X$ and $s_1 = Y$ do not indeed contain the AC function symbol f. However, the substitution σ given reintroduces this AC function symbol and we have $t_1\sigma = f\langle a, b\rangle$ and $s_1\sigma = f\langle b, a\rangle$.*

Lemma 2. *Let ft and fs be AC applications. Suppose that $(t_1, s_1) \in \texttt{gen\_unif\_prb}(ft, fs)$ and that $\nabla \vdash t_1\sigma \approx_\alpha s_1\sigma$. Then, $\exists t'_1 \in F_{AO}(t_1\sigma), s'_1 \in F_{AO}(s_1\sigma) : \nabla \vdash t'_1 \approx_\alpha s'_1$.*

Using Lemmas 1 and 2 we can obtain, from our original hypothesis of soundness, the existence of t'_1 and s'_1 such that $(t'_1, s'_1) \in \texttt{gen\_unif\_prb}((ft)\sigma, (fs)\sigma)$ and $\nabla \vdash t'_1 \approx_\alpha s'_1$ and we must prove that $\nabla \vdash (ft)\sigma \approx_\alpha (fs)\sigma$. Then, with a convenient renaming of variables, this is equivalent to proving Lemma 3, which can be done by induction on the size of the term t.

Lemma 3. *Let ft and fs be AC function applications, with the same function symbol. Suppose that $(t_1, s_1) \in \texttt{gen\_unif\_prb}(ft, fs)$ and that $\nabla \vdash t_1 \approx_\alpha s_1$. Then, $\nabla \vdash ft \approx_\alpha fs$.*

A similar analysis could be applied to prove the lemma of completeness.

Remark. We have not yet fully formalised any of the stated lemmas or theorems.

5 Conclusion

We commented on the specification of a functional algorithm that has been developed for doing nominal AC-unification and highlighted important points of the formalisation we are working on. We opted for a simple and inefficient algorithm in order to simplify the formalisation. Since

nominal AC-unification extends first-order AC-unification, completing this work is significant also to provide a formalisation of first-order AC-unification, which to the best of our knowledge does not yet exist.

References

[1] Mauricio Ayala-Rincón, Washington de Carvalho-Segundo, Maribel Fernández, and Daniele Nantes-Sobrinho. Nominal C-unification. In *27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2017), Revised Selected Papers*, volume 10855 of *Lecture Notes in Computer Science*, pages 235–251. Springer, 2018.

[2] Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes-Sobrinho. Soundness and completeness in PVS of a functional nominal C-unification algorithm. Available at http://ayala.mat.unb.br/publications.html, 2019.

[3] Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. Nominal narrowing. In *1st International Conference on Formal Structures for Computation and Deduction (FSCD)*, volume 52 of *LIPIcs*, pages 11:1–11:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[4] Mauricio Ayala-Rincón, Maribel Fernández, and Ana Rocha-Oliveira. Completeness in PVS of a nominal unification algorithm. *Electronic Notes in Theoretical Computer Science*, 323:57–74, 2016.

[5] James Cheney and Christian Urban. α-prolog: A logic programming language with names, binding and α-equivalence. In *20th International Conference on Logic Programming (ICLP)*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2004.

[6] Evelyne Contejean. A certified AC matching algorithm. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2004.

[7] Maribel Fernández and Murdoch Gabbay. Nominal rewriting. *Information and Computation*, 205(6):917–965, 2007.

[8] Andrew Pitts. *Nominal sets: Names and symmetry in computer science*. Cambridge University Press, 2013.

[9] Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

[10] Christian Urban, Andrew Pitts, and Murdoch Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004.

A Nominal AC-unification algorithm

The algorithm is specified in the functional PVS language and is available at www.github.com/gabriel951/ac-unification, as mentioned in the introduction. For brevity, we present here a partial description emphasising the elements related with AC functions. The treatment given to the other nominal elements can be checked in [2], where the case of nominal C-unification is treated.

Algorithm 1 is recursive and keeps track of the current context (Δ), the substitutions made so far (σ), the remaining terms to unify ($PrbLst$) and the current fixed point equations ($FPEqLst$). Therefore, the algorithm receives as input the quadruple $(\Delta, \sigma, PrbLst, FPEqLst)$.

Although extense, the algorithm is straightforward. It starts by analysing $PrbLst$, the list of terms it needs to unify. If it is an empty list, then it has finished and can return the answer computed so far, which is the list: $[(\Delta, \sigma, FPEqLst)]$. If $PrbLst$ is not empty, then there are terms to unify, and the algorithm proceeds by trying to unify the terms t and s that are in the head of the list. Only after that it goes to the tail of the list.

When the algorithm is unifying two AC function symbols t and s, the first step is to generate all pairings of t and s. As mentioned before, this is done by function `gen_unif_prb`. The result is a list of unification problems (each of these unification problems corresponds to an attempt of unifying a pairing for t and a pairing for s), stored in variable $NewPrbLst$. The algorithm will call the function UNIFY recursively, with the same Δ, σ and $FPEqLst$, but with a new list of unification problems, consisting of one unification problem from $NewPrbLst$ concatenated with the list of remaining problems $PrbList'$. The algorithm will do this for every unification problem in $NewPrbLst$. The algorithm does that in two steps. In the first step, the function `get_lst_quad` is called, constructing a list of quadruples, which are stored in variable $LstQuad$. Each quadruple in $LstQuad$ is of the form $(\Delta, \sigma, PrbLst, FPEqLst)$. Next, the `map` function applies recursively the function UNIFY to every quadruple in $LstQuad$ and since every application of UNIFY generates a list of solutions, the returned value of the `map` function is a list of lists of solutions, stored in variable $LstLstSol$. The algorithm then returns a flattened version of this list, via the use of the FLATTEN function.

Algorithm 1 Functional Nominal AC-Unification

procedure UNIFY$(\Delta, \sigma, PrbLst, FPEqLst)$
  if null$(PrbLst)$ **then**
    return list$((\Delta, \sigma, FPEqLst))$
  else
    $(t, s) + PrbLst' = PrbLst$
    if $(s == \pi \cdot X)$ and $(X$ not in $t)$ **then**
      check [2]
    else
      if $t == a$ **then**
        check [2]
      else if $t == \pi \cdot X$ **then**
        check [2]
      else if $t == \langle\rangle$ **then**
        check [2]
      else if $t == \langle t_1, t_2\rangle$ **then**
        check [2]
      else if $t == [a]t_1$ **then**
        check [2]
      else if $t == f\ t_1$ **then**
        check [2]
      else if $t == f^C\langle t_1, t_2\rangle$ **then**
        check [2]
      else ▷ t is of the form $f^{AC}t'$
        if s != $f^{AC}s'$ **then return** null
        else
          $NewPrbLst =$ `gen_unif_prb`(t, s)
          $LstQuad =$ `get_lst_quad`$(NewPrb, \Delta, \sigma, PrbList', FPEqLst)$
          $LstLstSol = map($UNIFY$, LstQuad)$
          return FLATTEN$(LstLstSol)$
        end if
      end if
    end if
  end if
end procedure

Solving Proximity Constraints*

Temur Kutsia and Cleo Pau

RISC, Johannes Kepler University Linz, Austria

Abstract. Proximity relations are binary fuzzy relations, which satisfy reflexivity and symmetry properties, but are not transitive. This relation induces the notion of distance between function symbols, which is further extended to terms. For two given terms we aim at bringing them "sufficiently close" to each other, by finding an appropriate substitution. A similar problem has been addressed by Julian-Iranzo and Rubio-Manzano. Unlike their work, we consider unrestricted proximal candidates, by allowing them to be close to two terms that themselves are not close to each other. We present an algorithm which works in two phases: first reducing the unification problem to constraints over sets of function symbols, and then solving the obtained constraints. The problem is decidable and finitary.

1 Introduction

Proximity relations are reflexive and symmetric fuzzy binary relations. They generalize similarity relations, which are a fuzzy version of equivalence. Proximity relations help to represent fuzzy information in situations, where similarity is not adequate. For unification, working modulo proximity or similarity means to treat different function symbols as if they were the same when the given relation asserts they are "close enough" to each other.

Unification for similarity relations was studied in [2] in the context of fuzzy logic programming. This work was generalized for proximity relations in [1], where the authors introduced the notion of proximity-based unification under a certain restriction imposed on the proximity relation. The restriction requires that the same symbol can not be close to two symbols at the same time, when those symbols are not close to each other. One should choose one of them as the proximal candidate to the given symbol. Looking at the proximity relation as an undirected graph, this restriction implies that cliques in the graph are disjoint. From the unification point of view, it means that $p(x, x)$ can not be unified with $p(a, c)$ when a and c are not close to each other, even if there exists a b which is close both to a and c.

In this paper we consider the general case: unification for a proximity relation without any restriction, and develop an algorithm which computes a compact representation of the set of solutions.

2 Preliminaries

Proximity relations. We define basic notions about proximity relations following [1]. A binary *fuzzy relation* on a set S is a mapping from $S \times S$ to the real interval $[0, 1]$. If $\mathcal{R}$ is a fuzzy relation on S and λ is a number $0 \leq \lambda \leq 1$, then the λ-*cut* of $\mathcal{R}$ on S, denoted $\mathcal{R}_\lambda$, is an ordinary (crisp) relation on S defined as $\mathcal{R}_\lambda := \{(s_1, s_2) \mid \mathcal{R}(s_1, s_1) \geq \lambda\}$. In the role of T-norm $\wedge$ we take the minimum.

A fuzzy relation $\mathcal{R}$ on a set S is called a *proximity relation* on S iff it is reflexive and symmetric. The *proximity class of level* λ *of* $s \in S$ (a λ-*neighborhood* of s) is a set $\mathbf{pc}(s, \mathcal{R}, \lambda) = \{s' \mid \mathcal{R}(s, s') \geq \lambda\}$. When $\mathcal{R}$ and λ are fixed, we simply write $\mathbf{pc}(s)$.

Terms. Given a set of variables $\mathcal{V}$ and a fixed arity signature Σ, *terms* over Σ and $\mathcal{V}$ are defined as usual, by the grammar $t := x \mid f(t_1, \ldots, t_n)$, where $x \in \mathcal{V}$ and $f \in \Sigma$ is n-ary. We assume to have Σ partitioned into a set of function symbols $\mathcal{F}$ and a set of names $\mathcal{N}$, having symbols of each arity in each of them.

The set of terms over $\mathcal{V}$ and Σ (resp. over $\mathcal{F}$, $\mathcal{N}$), is denoted by $\mathcal{T}(\Sigma, \mathcal{V})$ (resp. $\mathcal{T}(\mathcal{F}, \mathcal{V})$, $\mathcal{T}(\mathcal{N}, \mathcal{V})$). We denote variables by x, y, z, arbitrary function symbols by f, g, h, constants by a, b, c, names by $\mathrm{N}, \mathrm{M}, \mathrm{K}$, and terms by s, t, r. The elements of $\mathcal{T}(\mathcal{F}, \mathcal{V})$ are called $\mathcal{F}$-terms. The elements of $\mathcal{T}(\mathcal{N}, \mathcal{V})$ are $\mathcal{N}$-terms.

The *head* of an $\mathcal{F}$-term is defined as $\mathit{head}(x) := x$ and $\mathit{head}(f(t_1, \ldots, t_n)) := f$.

$\mathcal{F}$-Substitutions (resp. $\mathcal{N}$-substitutions) are mappings from variables to $\mathcal{F}$-terms (resp. to $\mathcal{N}$-terms), where all but finitely many variables are mapped to themselves. The letters $\sigma, \vartheta, \varphi$ are used for $\mathcal{F}$-substitutions, and

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

their upright versions $\sigma, \vartheta, \varphi$ for $\mathcal{N}$-substitutions. The identity substitution is denoted by Id. We use the usual set notation for substitutions, writing, e.g., σ as $\sigma = \{x \mapsto \sigma(x) \mid x \neq \sigma(x)\}$. Substitution application and composition are defined in the standard way.

For simplicity, below we refer to $\mathcal{F}$-terms and $\mathcal{F}$-substitutions just by terms and substitutions, and only mention $\mathcal{N}$-terms and $\mathcal{N}$-substitutions explicitly.

A *name-class mapping* Φ is a finite mapping from names to sets of function symbols such that if $\mathrm{N} \in dom(\Phi)$ (where *dom* is the domain of mapping), then N and each $f \in \Phi(\mathrm{N})$ have the same arity.

Proximity relations over terms and substitutions. Each proximity relation $\mathcal{R}$ we consider in this paper obeys the following restrictions: (a) It is defined on $\mathcal{F} \cup \mathcal{V}$; (b) $\mathcal{R}(f, g) = 0$ if $arity(f) \neq arity(g)$; (c) $\mathcal{R}(f, x) = 0$ if $f \in \mathcal{F}$ and $x \in \mathcal{V}$, (d) $\mathcal{R}(x, y) = 0$ if $x \neq y$, for all $x, y \in \mathcal{V}$. We extend such a relation $\mathcal{R}$ to terms: (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) $\mathcal{R}(s, t) := \mathcal{R}(f, g) \wedge \mathcal{R}(s_1, t_1) \wedge \cdots \wedge \mathcal{R}(s_n, t_n)$, if $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_n)$.

Two terms s and t are $(\mathcal{R}, \lambda)$-*close* to each other, written $s \simeq_{\mathcal{R},\lambda} t$, if $\mathcal{R}(s, t) \geq \lambda$. We say that s is $(\mathcal{R}, \lambda)$-*more general than* t and write $s \precsim_{\mathcal{R},\lambda} t$, if there exists a substitution σ such that $s\sigma \simeq_{\mathcal{R},\lambda} t$. This relation is defined for substitutions as well: $\sigma \precsim_{\mathcal{R},\lambda} \vartheta$ iff there exists φ such that $x\sigma\varphi \simeq_{\mathcal{R},\lambda} x\vartheta$ for all x.

The relation $\precsim_{\mathcal{R},\lambda}$ is not a quasi-order: it is reflexive, but not transitive.

Definition 1 (Approximate unification problem). *Given a proximity relation $\mathcal{R}$ and a cut value λ, an* $(\mathcal{R}, \lambda)$-unification problem *Γ is a finite set of approximate equations (i.e., pairs) of terms, written as $\Gamma := \{s_1 \simeq^?_{\mathcal{R},\lambda} t_1, \ldots, s_n \simeq^?_{\mathcal{R},\lambda} t_n\}$. The question mark indicates that it is supposed to be solved. An* unifier *of Γ is a substitution σ such that $s_i\sigma \simeq_{\mathcal{R},\lambda} t_i\sigma$ for all $1 \leq i \leq n$.*

Instead of writing "a unifier of an $(\mathcal{R}, \lambda)$-unification problem Γ", we often shortly say "an $(\mathcal{R}, \lambda)$-unifier of Γ". If σ is an $(\mathcal{R}, \lambda)$-unifier of Γ and $\sigma \preceq_{\mathcal{R},\lambda} \vartheta$, it might be that ϑ is not an $(\mathcal{R}, \lambda)$-unifier of Γ. For instance, if $\mathcal{R}_\lambda = \{(a, b), (b, c)\}$, then $\sigma = \{x \mapsto b\}$ is an $(\mathcal{R}, \lambda)$-unifier of $\{x \simeq^?_{\mathcal{R},\lambda} a\}$. Besides, $\sigma \preceq_{\mathcal{R},\lambda} \vartheta = \{x \mapsto c\}$, but ϑ is not an $(\mathcal{R}, \lambda)$-unifier of $\{x \simeq^?_{\mathcal{R},\lambda} a\}$.

However, if σ is an $(\mathcal{R}, \lambda)$-unifier of Γ, then $\sigma\varphi$ is also an $(\mathcal{R}, \lambda)$-unifier of Γ for any φ.

It might happen that a minimal complete set of $(\mathcal{R}, \lambda)$-unifiers (defined by using $\precsim_{\mathcal{R},\lambda}$ and $\simeq_{\mathcal{R},\lambda}$) of some Γ contains two substitutions σ and ϑ such that $\sigma \simeq_{\mathcal{R},\lambda} \vartheta$, but it is forbidden that $\vartheta = \sigma\varphi$ for some φ.

Definition 2. *Given $\mathcal{R}$ and λ, an $(\mathcal{R}, \lambda)$-neighborhood equation has one of the following forms: $f \approx^?_{\mathcal{R},\lambda} g$, $\mathrm{N} \approx^?_{\mathcal{R},\lambda} g$, $g \approx^?_{\mathcal{R},\lambda} \mathrm{N}$, or $\mathrm{N} \approx^?_{\mathcal{R},\lambda} \mathrm{M}$, where $f, g \in \mathcal{F}$ and $\mathrm{N}, \mathrm{M} \in \mathcal{N}$. The question mark indicates that they have to be solved.*

We say that a name-class mapping Φ is a solution of an $(\mathcal{R}, \lambda)$-neighborhood equation if

- *the equation is $f \approx^?_{\mathcal{R},\lambda} g$ and $f \approx_{\mathcal{R},\lambda} g$ holds (in this case any Φ would be a solution), or*
- *the equation is $\mathrm{N} \approx^?_{\mathcal{R},\lambda} g$ or $g \approx^?_{\mathcal{R},\lambda} \mathrm{N}$ and $f \approx_{\mathcal{R},\lambda} g$ holds for all $f \in \Phi(\mathrm{N})$, or*
- *the equation is $\mathrm{N} \approx^?_{\mathcal{R},\lambda} \mathrm{M}$ and $f \approx_{\mathcal{R},\lambda} g$ holds for all $f \in \Phi(\mathrm{N})$ and $g \in \Phi(\mathrm{M})$.*

An $(\mathcal{R}, \lambda)$-neighborhood constraint is a finite set of $(\mathcal{R}, \lambda)$-neighborhood equations. A name-class mapping Φ is a solution of an $(\mathcal{R}, \lambda)$-neighborhood constraint C if it is a solution of every $(\mathcal{R}, \lambda)$-neighborhood equation in C.

We shortly write "an $(\mathcal{R}, \lambda)$-solution to C" instead of "a solution to an $(\mathcal{R}, \lambda)$-neighborhood constraint C". Given a name-class mapping Φ and an $\mathcal{N}$-term t, we define a set of terms $\Phi(t)$ as

$$\Phi(x) := \{x\}, \qquad \Phi(\mathrm{N}(t_1, \ldots, t_n)) := \{f(s_1, \ldots, s_n) \mid f \in \Phi(\mathrm{N}), s_i \in \Phi(t_i), 1 \leq i \leq n\}.$$

The notation extends to $\mathcal{N}$-substitutions as well: $\Phi(\sigma) := \{\sigma \mid x\sigma \in \Phi(x\sigma) \text{ for all } x \in \mathcal{V}\}$.

Definition 3. *We say that a set of term equations $\{x \simeq^?_{\mathcal{R},\lambda} t\} \uplus P$ contains an* occurrence cycle *for the variable x if $t \notin \mathcal{V}$ and either*

- *$x \in var(t)$, or*
- *there exist term-pairs $(x_1, t_1[x_2]), \ldots, (x_n, t_n[x])$ such that $x_1 \in var(t)$ and for each $1 \leq i \leq n$, P contains an equation $x_i \simeq^?_{\mathcal{R},\lambda} t_i$ or $t_i \simeq^?_{\mathcal{R},\lambda} x_i$. (The notation $t[x]$ means that x occurs in t.)*

Lemma 1. *If a set of term equations contains an occurrence cycle for some variable, then it has no $(\mathcal{R}, \lambda)$-approximate unifier for any proximity relation $\mathcal{R}$ and cut value λ.*

Given an approximate unification problem Γ, our goal is to obtain a compact representation of its (minimal) complete set of unifiers as a pair of an $\mathcal{N}$-substitution σ and a name-class mapping Φ, so that $\Phi(\sigma)$ (restricted to the variables of Γ) gives the desired set. The algorithms below construct such a representation.

3 The Algorithms

In the rules below we will use the *renaming function* $\rho : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{T}(\mathcal{N}, \mathcal{V})$. Applied to a Σ-term, ρ gives its fresh copy, an $\mathcal{N}$-term, obtained by replacing each occurrence of a symbol from Σ by a new name and each variable occurrence by a fresh variable. For instance, if the term is $f(\mathrm{N}(a, x, x, f(a)))$, where $f, a \in \mathcal{F}$ and $\mathrm{N} \in \mathcal{N}$, then $\rho(f(\mathrm{N}(a, x, x, f(a))) = \mathrm{N}_1(\mathrm{N}_2(\mathrm{N}_3, x_1, x_2, \mathrm{N}_4(\mathrm{N}_5)))$, where $\mathrm{N}_1, \mathrm{N}_2, \mathrm{N}_3, \mathrm{N}_4, \mathrm{N}_5 \in \mathcal{N}$ are new names and x_1, x_2 are new variables.

The algorithm consists of two phases: pre-unification and constraint solving. In the pre-unification phase we either obtain a pre-unifier together with neighborhood constraints which have to be solved in the second phase, or get a failure which indicates that the problem does not have a solution.

3.1 Pre-Unification Rules

An *equational configuration* is a triple $P; C; \sigma$, where

- P is the unification problem to be solved. It is initialized with the equation between the original terms;
- C is a neighborhood constraint;
- σ is the pre-unifier computed so far, initialized by Id.

The pre-unification algorithm takes given terms s and t, creates the initial configuration $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \emptyset; Id$ and applies the rules given below exhaustively.

The rules are very similar to the syntactic unification algorithm with the difference that here the function symbol clash does not happen unless their arities differ, and variables are not replaced by other variables until the very end. (The notation $\overline{exp_n}$ in the rules below abbreviates the sequence $exp_1, \ldots, exp_n$.)

(Tri) Trivial:	$\{x \simeq^?_{\mathcal{R},\lambda} x\} \uplus P; C; \sigma \Longrightarrow P; C; \sigma.$
(Dec) Decomposition:	$\{F(\overline{s_n}) \simeq^?_{\mathcal{R},\lambda} G(\overline{t_n})\} \uplus P; C; \sigma \Longrightarrow \{\overline{s_n \simeq^?_{\mathcal{R},\lambda} t_n}\} \cup P; \{F \approx^?_{\mathcal{R},\lambda} G\} \cup C; \sigma,$ where $F, G \in \Sigma$.
(VE) Var. Elim.:	$\{x \simeq^?_{\mathcal{R},\lambda} t\} \uplus P; C; \sigma \Longrightarrow \{t' \simeq^?_{\mathcal{R},\lambda} t\} \cup P\{x \mapsto t'\}; C; \sigma\{x \mapsto t'\},$ where $t \notin \mathcal{V}$, there is no occurrence cycle for x in $\{x \simeq^?_{\mathcal{R},\lambda} t\} \uplus P$, and $t' = \rho(t)$.
(Ori) Orient:	$\{t \simeq^?_{\mathcal{R},\lambda} x\} \uplus P; C; \sigma \Longrightarrow \{x \simeq^?_{\mathcal{R},\lambda} t\} \cup P; C; \sigma,$ if $t \notin \mathcal{V}$.
(Cla) Clash:	$F(\overline{s_n}) \simeq^?_{\mathcal{R},\lambda} G(\overline{t_m})\} \uplus P; C; \sigma \Longrightarrow \bot,$ where $F, G \in \Sigma$ and $n \neq m$.
(Occ) Occur Check:	$\{x \simeq^?_{\mathcal{R},\lambda} t\} \uplus P; C; \sigma \Longrightarrow \bot,$ if there is an occurrence cycle for x in $\{x \simeq^?_{\mathcal{R},\lambda} t\} \uplus P$.
(VO) Vars Only:	$\{x \simeq^?_{\mathcal{R},\lambda} y, \overline{x_n \simeq^?_{\mathcal{R},\lambda} y_n}\}; C; \sigma \Longrightarrow \{\overline{x_n \simeq^?_{\mathcal{R},\lambda} y_n}\}\{x \mapsto y\}; C; \sigma\{x \mapsto y\}.$

It is easy to see that the pre-unification algorithm terminates either with $\bot$ or with $\emptyset; C; \sigma$, where C is a neighborhood constraint and σ is an $\mathcal{N}$-substitution. When the result is $\bot$, there is no unifier of s and t: terms of different number of arguments can not be unified (Clash), and a system with occurrence cycle (Occur Check) has no solutions (Lemma 1).

Theorem 1 (Soundness of pre-unification). *Let s and t be two terms, such that the pre-unification algorithm gives $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \emptyset; Id \Longrightarrow^* \emptyset; C; \sigma$. Let Φ be an $(\mathcal{R}, \lambda)$-solution of C. Then any substitution in the set $\Phi(\sigma)$ is an $(\mathcal{R}, \lambda)$-unifier of s and t.*

Theorem 2 (Completeness of pre-unification). *Let ψ be an $(\mathcal{R}, \lambda)$-unifier of s and t. Then there exists a derivation $\{s \simeq^?_{\mathcal{R},\lambda} t\}; \emptyset; Id \Longrightarrow^* \emptyset; C; \sigma$ such that $\varphi \preceq_{\mathcal{R},\lambda} \psi$ for some $\varphi \in \Phi(\sigma|_{var(s) \cup var(t)})$, where Φ is an $(\mathcal{R}, \lambda)$-solution of C.*

For solving neighborhood constraints, we will introduce an algorithm in the next section. But before that we illustrate the pre-unification rules with a couple of examples:

Example 1. Let $s = p(x, y, x)$ and $t = q(f(a), g(d), y)$. Then the pre-unification algorithm stops with the configuration $\emptyset; C, \sigma$, where $C = \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}$ and $\sigma = \{x \mapsto \mathrm{N}_1(\mathrm{N}_2), y \mapsto \mathrm{N}_3(\mathrm{N}_4)\}$.

Assume that for the given λ-cut, the proximity relation consists of pairs $\mathcal{R}_\lambda = \{(a, b), (b, c), (c, d), (a, b'), (b', c'), (c', d), (f, g), (p, q)\}$. The obtained constraint can be solved, e.g., by the name-class mappings $\Phi = [\mathrm{N}_1 \mapsto \{f, g\}, \mathrm{N}_2 \mapsto \{b\}, \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \{c\}]$ and $\Phi' = [\mathrm{N}_1 \mapsto \{f, g\}, \mathrm{N}_2 \mapsto \{b'\}, \mathrm{N}_3 \mapsto \{f, g\}, \mathrm{N}_4 \mapsto \{c'\}]$. From them and σ we can get the sets $\Phi(\sigma)$ and $\Phi'(\sigma)$ of $(\mathcal{R}, \lambda)$-unifiers of s and t. Each of them consists of 4 substitutions.

If we did not have the VO rule and allowed the use of VE rule instead, we might have ended up with the unification problem $\{y \simeq^?_{\mathcal{R},\lambda} f(a), y \simeq^?_{\mathcal{R},\lambda} g(d)\}$, which does not have a solution, because the neighborhoods of a and d do not have a common element. Hence, we would have lost a solution.

Example 2. Let $s = p(x, x)$ and $t = q(f(y, y), f(a, c))$. The pre-unification algorithm stops with the configuration $\emptyset; P; \sigma$, where $P = \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\}$ and $\sigma = \{x \mapsto \mathrm{N}_1(\mathrm{N}_2, \mathrm{N}_3), y_1 \mapsto \mathrm{N}_2, y_2 \mapsto \mathrm{N}_3, y \mapsto \mathrm{M}\}$. Let $\mathcal{R}_\lambda = \{(a, a_1), (a_1, b), (b, c_1), (c_1, c), (p, q)\}$. Then C is solved by $\Phi = [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a_1\}, \mathrm{M} \mapsto \{b\}, \mathrm{N}_3 \mapsto \{c_1\}]$ and $\Phi(\sigma|_{var(s)\cup var(t)})$ contains only one element, an $(\mathcal{R}, \lambda)$-unifier $\sigma = \{x \mapsto f(a_1, c_1), y \mapsto b\}$ of s and t. Indeed, $s\sigma = p(f(a_1, c_1), f(a_1, c_1)) \simeq_{\mathcal{R},\lambda} q(f(b, b), f(a, c)) = t\sigma$.

This example illustrates the necessity of introducing a fresh variable for *each occurrence* of a variable by the renaming function in the VE rule. If we used the same new variable, say y', for both occurrences of y in $f(y, y)$ (instead of using y_1 and y_2 as above), we would get the configuration $\emptyset; \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\}; \{x \mapsto \mathrm{N}_1(\mathrm{N}_2, \mathrm{N}_2), y' \mapsto \mathrm{N}_2, y \mapsto \mathrm{N}_2\}$. But for the given $\mathcal{R}_\lambda$, the constraint $\{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\}$ does not have a solution (because the neighborhoods of a and c are not close to each other). Hence, we would lose a unifier.

3.2 Rules for Neighborhood Constraints

Let Φ be a *name-class mapping*. Then the function of updating Φ by a mapping $\mathrm{N} \to S$ for $\mathrm{N} \in \mathcal{N}$ and $S \subset \mathcal{F}$, where N and the elements of S have the same arity, is defined as

$$\mathsf{update}(\Phi, \mathrm{N} \mapsto S) = \begin{cases} \Phi \cup [\mathrm{N} \mapsto S], \text{ if } \mathrm{N} \notin \mathit{dom}(\Phi), \\ (\Phi \setminus [\mathrm{N} \mapsto \Phi(\mathrm{N})]) \cup [\mathrm{N} \mapsto S \cap \Phi(\mathrm{N})], \text{ otherwise.} \end{cases}$$

We write $\mathsf{update}(\Phi, \mathrm{N}_1 \to S_1, \ldots, \mathrm{N}_n \to S_n)$ for $\mathsf{update}(\cdots \mathsf{update}(\Phi, \mathrm{N}_1 \to S_1), \ldots, \mathrm{N}_n \to S_n)$.

A *constraint configuration* is a pair $C; \Phi$, where C is a set of $(\mathcal{R}, \lambda)$-neighborhood constraints to be solved, and Φ is a name-class mapping (as a set of rules), representing the $(\mathcal{R}, \lambda)$-solution computed so far. It is initialized by the empty set. The constraint simplification algorithm $\mathcal{CS}$ transforms constraint configurations, exhaustively applying the following rules:

(FFS) Function Symbols:	$\{f \approx^?_{\mathcal{R},\lambda} g\} \uplus C; \Phi \Longrightarrow C; \Phi$, if $\mathcal{R}(f, g) \geq \lambda$.
(NFS) Name vs Function Symbol:	$\{\mathrm{N} \approx^?_{\mathcal{R},\lambda} g\} \uplus C; \Phi \Longrightarrow C; \mathsf{update}(\Phi, \mathrm{N} \mapsto \mathbf{pc}(g, \mathcal{R}, \lambda))$.
(FSN) Function Symbol vs Name:	$\{g \approx^?_{\mathcal{R},\lambda} \mathrm{N}\} \uplus C; \Phi \Longrightarrow \{\mathrm{N} \approx^?_{\mathcal{R},\lambda} g\} \cup C; \Phi$.
(NN1) Name vs Name 1:	$\{\mathrm{N} \approx^?_{\mathcal{R},\lambda} \mathrm{M}\} \uplus C; \Phi \Longrightarrow C; \mathsf{update}(\Phi, \mathrm{N} \mapsto \{f\}, \mathrm{M} \mapsto \mathbf{pc}(f, \mathcal{R}, \lambda))$, where $\mathrm{N} \in \mathit{dom}(\Phi)$, $f \in \Phi(\mathrm{N})$.
(NN2) Name vs Name 2:	$\{\mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}\} \uplus C; \Phi \Longrightarrow \{\mathrm{N} \approx^?_{\mathcal{R},\lambda} \mathrm{M}\} \cup C; \Phi$, where $\mathrm{M} \notin \mathit{dom}(\Phi)$, $\mathrm{N} \in \mathit{dom}(\Phi)$.
(Fail1) Failure 1:	$\{f \approx^?_{\mathcal{R},\lambda} g\} \uplus C; \Phi \Longrightarrow \bot$, if $\mathcal{R}(f, g) < \lambda$.
(Fail2) Failure 2:	$C; \Phi \Longrightarrow \bot$, if there exists $\mathrm{N} \in \mathit{dom}(\Phi)$ such that $\Phi(\mathrm{N}) = \emptyset$.

The NN1 rule causes branching, generating n branches where n is the number of elements in $\Phi(\mathrm{N})$ (assuming that the proximity class of each symbol is finite). When the derivation does not fail, the terminal configuration has the form $\{\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{M}_1, \ldots, \mathrm{N}_n \approx^?_{\mathcal{R},\lambda} \mathrm{M}_n\}; \Phi$, where for each $1 \leq i \leq n$, $\mathrm{N}_i, \mathrm{M}_i \notin \mathit{dom}(\Phi)$. Such a constraint is trivially solvable.

Theorem 3 (Soundness of $\mathcal{CS}$). *Let C be an $(\mathcal{R},\lambda)$-neighborhood constraint such that $\mathcal{CS}$ produces a maximal derivation $C;\emptyset \Longrightarrow^* C';\Phi$. Then Φ is an $(\mathcal{R},\lambda)$-solution of $C \setminus C'$, and C' is a set of constraints between names which is trivially $(\mathcal{R},\lambda)$-satisfiable.*

Theorem 4 (Completeness of $\mathcal{CS}$). *Let C be an $(\mathcal{R},\lambda)$-neighborhood constraint and Φ be its solution such that every name from $dom(\Phi)$ appears in C. Let $dom(\Phi) = \{\mathrm{N}_1,\ldots,\mathrm{N}_n\}$. Then for each n-tuple $c_1 \in \Phi(\mathrm{N}_1),\ldots,c_n \in \Phi(\mathrm{N}_n)$ there exists a maximal $\mathcal{CS}$-derivation $C;\emptyset \Longrightarrow^* C';\Phi'$ such that for each $1 \le i \le n$, either $c_i \in \Phi'(\mathrm{N}_i)$, or there exists $1 \le j \le n$ such that $c_i \in \Phi(\mathrm{N}_j)$ and $\mathrm{N}_i \approx^?_{\mathcal{R},\lambda} \mathrm{N}_j \in C'$.*

Remark 1. When a neighborhood constraint C is produced by the pre-unification algorithm, then every maximal $\mathcal{CS}$-derivation starting from $C;\emptyset$ ends either in $\bot$ or in the pair of the form $\emptyset;\Phi$. This is due to the fact that the VE rule, which introduces names in pre-unification problems, and the subsequent decomposition steps necessarily produce neighborhood equations of the form $\mathrm{N} \approx_{\mathcal{R},\lambda} f$ for each introduced N and for some f.

Example 3. The pre-unification derivation in Example 1 gives the neighborhood constraint $C = \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\}$. For $\mathcal{R}_\lambda = \{(a,b),(b,c),(c,d),(a,b'),(b',c'),(c',d),(f,g),(p,q)\}$, the constraint C can be solved by $\mathcal{CS}$ as follows:

$$\{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\};\emptyset \Longrightarrow_{\mathsf{FFS}}$$
$$\{\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\};\emptyset \Longrightarrow_{\mathsf{NFS}}$$
$$\{\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\};[\mathrm{N}_1 \mapsto \{f,g\}] \Longrightarrow_{\mathsf{NFS}}$$
$$\{\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g,\ \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\};[\mathrm{N}_1 \mapsto \{f,g\},\ \mathrm{N}_2 \mapsto \{a,b,b'\}] \Longrightarrow_{\mathsf{NFS}}$$
$$\{\mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\};[\mathrm{N}_1 \mapsto \{f,g\},\ \mathrm{N}_2 \mapsto \{a,b,b'\},\ \mathrm{N}_3 \mapsto \{f,g\}] \Longrightarrow_{\mathsf{NFS}}$$
$$\{\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\};[\mathrm{N}_1 \mapsto \{f,g\},\ \mathrm{N}_2 \mapsto \{a,b,b'\},\ \mathrm{N}_3 \mapsto \{f,g\},\ \mathrm{N}_4 \mapsto \{d,c,c'\}].$$

Here the algorithm branches, since the rule NN1 applies. Branch 1 continues with

$$\{\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\};[\mathrm{N}_1 \mapsto \{f,g\},\mathrm{N}_2 \mapsto \{a,b,b'\},\mathrm{N}_3 \mapsto \{f,g\},\mathrm{N}_4 \mapsto \{d,c,c'\}] \Longrightarrow_{\mathsf{NN1}}$$
$$\{\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\};[\mathrm{N}_1 \mapsto \{f\},\mathrm{N}_2 \mapsto \{a,b,b'\},\mathrm{N}_3 \mapsto \{f,g\},\mathrm{N}_4 \mapsto \{d,c,c'\}].$$

Branching again by NN1 produces three subbranches. Branch 1.1 fails: $\emptyset;[\mathrm{N}_1 \mapsto \{f\},\mathrm{N}_2 \mapsto \{a\},\mathrm{N}_3 \mapsto \{f,g\},\mathrm{N}_4 \mapsto \emptyset] \Longrightarrow_{\mathsf{Fail2}} \bot$. Branch 1.2 and branch 1.3 give two solutions, respectively:

$$\Phi_1 = [\mathrm{N}_1 \mapsto \{f\},\mathrm{N}_2 \mapsto \{b\},\ \mathrm{N}_3 \mapsto \{f,g\},\mathrm{N}_4 \mapsto \{c\}]$$
$$\Phi_2 = [\mathrm{N}_1 \mapsto \{f\},\mathrm{N}_2 \mapsto \{b'\},\mathrm{N}_3 \mapsto \{f,g\},\mathrm{N}_4 \mapsto \{c'\}].$$

Branch 2 also expands into three subbranches, the first of which fails. The other two return two more solutions:

$$\Phi_3 = [\mathrm{N}_1 \mapsto \{g\},\mathrm{N}_2 \mapsto \{b\},\ \mathrm{N}_3 \mapsto \{f,g\},\mathrm{N}_4 \mapsto \{c\}]$$
$$\Phi_4 = [\mathrm{N}_1 \mapsto \{g\},\mathrm{N}_2 \mapsto \{b'\},\mathrm{N}_3 \mapsto \{f,g\},\mathrm{N}_4 \mapsto \{c'\}].$$

Referring to the name-class mappings Φ and Φ' and the substitution σ in Example 1, it is easy to observe that $\Phi(\sigma) \cup \Phi'(\sigma) = \Phi_1(\sigma) \cup \Phi_2(\sigma) \cup \Phi_3(\sigma) \cup \Phi_4(\sigma)$.

4 Final Remarks

We described our work in progress towards solving equational constraints over proximity relations. The immediate next step is to incorporate the computation of unification degree into the procedure and use it to return only those unifiers which bring the original terms as close as possible to each other.

References

1. P. Julián-Iranzo and C. Rubio-Manzano. Proximity-based unification theory. *Fuzzy Sets and Systems*, 262:21–43, 2015.
2. M. I. Sessa. Approximate reasoning by similarity-based SLD resolution. *Theor. Comput. Sci.*, 275(1-2):389–426, 2002.

A Examples

The selected equations are underlined.

Example 4. Let $s = p(x, y, x)$ and $t = q(f(a), g(d), y)$. The following is a pre-unification derivation, which shows how the neighborhood constraint and the substitution shown in Example 1 are computed:

$\{\underline{p(x,y,x) \simeq^?_{\mathcal{R},\lambda} q(f(a),g(d),y)}\}; \emptyset; Id \Longrightarrow_{\mathsf{Dec}}$
$\{\underline{x \simeq^?_{\mathcal{R},\lambda} f(a)}, y \simeq^?_{\mathcal{R},\lambda} g(d), x \simeq^?_{\mathcal{R},\lambda} y\}; \{p \approx^?_{\mathcal{R},\lambda} q\}; Id \Longrightarrow_{\mathsf{VE}}$
$\{\underline{\mathrm{N}_1(\mathrm{N}_2) \simeq^?_{\mathcal{R},\lambda} f(a)}, y \simeq^?_{\mathcal{R},\lambda} g(d), \mathrm{N}_1(\mathrm{N}_2) \simeq^?_{\mathcal{R},\lambda} y\}; \{p \approx^?_{\mathcal{R},\lambda} q\}; \{x \mapsto \mathrm{N}_1(\mathrm{N}_2)\} \Longrightarrow_{\mathsf{Dec}^2}$
$\{\underline{y \simeq^?_{\mathcal{R},\lambda} g(d)}, \mathrm{N}_1(\mathrm{N}_2) \simeq^?_{\mathcal{R},\lambda} y\}; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a\}; \{x \mapsto \mathrm{N}_1(\mathrm{N}_2)\} \Longrightarrow_{\mathsf{VE}}$
$\{\underline{\mathrm{N}_3(\mathrm{N}_4) \simeq^?_{\mathcal{R},\lambda} g(d)}, \mathrm{N}_1(\mathrm{N}_2) \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_3(\mathrm{N}_4)\}; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a\};$
$\{x \mapsto \mathrm{N}_1(\mathrm{N}_2), y \mapsto \mathrm{N}_3(\mathrm{N}_4)\} \Longrightarrow_{\mathsf{Dec}^2}$
$\{\underline{\mathrm{N}_1(\mathrm{N}_2) \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_3(\mathrm{N}_4)}\}; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g, \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d\};$
$\{x \mapsto \mathrm{N}_1(\mathrm{N}_2), y \mapsto \mathrm{N}_3(\mathrm{N}_4)\} \Longrightarrow_{\mathsf{Dec}^2}$
$\emptyset; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} g, \mathrm{N}_4 \approx^?_{\mathcal{R},\lambda} d, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_3, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_4\};$
$\{x \mapsto \mathrm{N}_1(\mathrm{N}_2), y \mapsto \mathrm{N}_3(\mathrm{N}_4)\}.$

Note that we could have chosen the second equation in $\{y \simeq^?_{\mathcal{R},\lambda} g(d), \mathrm{N}_1(\mathrm{N}_2) \simeq^?_{\mathcal{R},\lambda} y\}$ to eliminate y, but the obtained result would differ from the above computed one by the choice of names only.

Together with the output from Example 3, we report the solutions for $\mathcal{R}_\lambda = \{(a,b),(b,c),(c,d),(a,b'),(b',c'),(c',d),(f,g),(p,q)\}$,

$\Phi_1 = [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{b\}, \mathrm{N}_3 \mapsto \{f,g\}, \mathrm{N}_4 \mapsto \{c\}],$
$\Phi_2 = [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{b'\}, \mathrm{N}_3 \mapsto \{f,g\}, \mathrm{N}_4 \mapsto \{c'\}],$
$\Phi_3 = [\mathrm{N}_1 \mapsto \{g\}, \mathrm{N}_2 \mapsto \{b\}, \mathrm{N}_3 \mapsto \{f,g\}, \mathrm{N}_4 \mapsto \{c\}],$
$\Phi_4 = [\mathrm{N}_1 \mapsto \{g\}, \mathrm{N}_2 \mapsto \{b'\}, \mathrm{N}_3 \mapsto \{f,g\}, \mathrm{N}_4 \mapsto \{c'\}],$
$\sigma = \{x \mapsto \mathrm{N}_1(\mathrm{N}_2), y \mapsto \mathrm{N}_3(\mathrm{N}_4)\}.$

Example 5. Now we illustrate the steps made for computations in Example 2. Let $s = p(x, x)$ and $t = q(f(y, y), f(a, c))$. Then the pre-unification derivation looks as follows:

$\{\underline{p(x,x) \simeq^?_{\mathcal{R},\lambda} q(f(y,y),f(a,c))}\}; \emptyset; Id \Longrightarrow_{\mathsf{Dec}}$
$\{\underline{x \simeq^?_{\mathcal{R},\lambda} f(y,y)}, x \simeq^?_{\mathcal{R},\lambda} f(a,c)\}; \{p \approx^?_{\mathcal{R},\lambda} q\}; Id \Longrightarrow_{\mathsf{VE}}$
$\{\underline{\mathrm{N}_1(y_1,y_2) \simeq^?_{\mathcal{R},\lambda} f(y,y)}, \mathrm{N}_1(y_1,y_2) \simeq^?_{\mathcal{R},\lambda} f(a,c)\}; \{p \approx^?_{\mathcal{R},\lambda} q\}; \{x \mapsto \mathrm{N}_1(y_1,y_2)\} \Longrightarrow_{\mathsf{Dec}}$
$\{y_1 \simeq^?_{\mathcal{R},\lambda} y, y_2 \simeq^?_{\mathcal{R},\lambda} y, \underline{\mathrm{N}_1(y_1,y_2) \simeq^?_{\mathcal{R},\lambda} f(a,c)}\}; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f\}; \{x \mapsto \mathrm{N}_1(y_1,y_2)\} \Longrightarrow_{\mathsf{Dec}}$
$\{y_1 \simeq^?_{\mathcal{R},\lambda} y, y_2 \simeq^?_{\mathcal{R},\lambda} y, \underline{y_1 \simeq^?_{\mathcal{R},\lambda} a}, y_2 \simeq^?_{\mathcal{R},\lambda} c\}; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f\};$
$\{x \mapsto \mathrm{N}_1(y_1,y_2)\} \Longrightarrow_{\mathsf{VE,Dec}}$
$\{\mathrm{N}_2 \simeq^?_{\mathcal{R},\lambda} y, y_2 \simeq^?_{\mathcal{R},\lambda} y, \underline{y_2 \simeq^?_{\mathcal{R},\lambda} c}\}; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a\};$
$\{x \mapsto \mathrm{N}_1(\mathrm{N}_2,y_2), y_1 \mapsto \mathrm{N}_2\} \Longrightarrow_{\mathsf{VE,Dec}}$
$\{\underline{\mathrm{N}_2 \simeq^?_{\mathcal{R},\lambda} y}, \mathrm{N}_3 \simeq^?_{\mathcal{R},\lambda} y\}; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c\};$
$\{x \mapsto \mathrm{N}_1(\mathrm{N}_2,\mathrm{N}_3), y_1 \mapsto \mathrm{N}_2, y_2 \mapsto \mathrm{N}_3\} \Longrightarrow_{\mathsf{Ori,VE}}$
$\{\underline{\mathrm{N}_3 \simeq^?_{\mathcal{R},\lambda} \mathrm{M}}\}; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c, \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};$
$\{x \mapsto \mathrm{N}_1(\mathrm{N}_2,\mathrm{N}_3), y_1 \mapsto \mathrm{N}_2, y_2 \mapsto \mathrm{N}_3, y \mapsto \mathrm{M}\} \Longrightarrow_{\mathsf{Dec}}$
$\emptyset; \{p \approx^?_{\mathcal{R},\lambda} q, \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f, \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c, \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2, \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\};$

$\{x \mapsto \mathrm{N}_1(\mathrm{N}_2, \mathrm{N}_3),\ y_1 \mapsto \mathrm{N}_2,\ y_2 \mapsto \mathrm{N}_3,\ y \mapsto \mathrm{M}\}$.

If $\mathcal{R}_\lambda = \{(a, a_1), (a_1, b), (b, c_1), (c_1, c), (p, q)\}$, then the obtained constraint is satisfied by the assignment $[\mathrm{N}_1 \mapsto \{f\},\ \mathrm{N}_2 \mapsto \{a_1\},\ \mathrm{M} \mapsto \{b\},\ \mathrm{N}_3 \mapsto \{c_1\}]$, computed by $\mathcal{CS}$ as follows (where NN1 causes branching, we display only the success branch):

$\{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\};\ \emptyset \Longrightarrow_{\mathsf{FFS}}$
$\{\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\};\ \emptyset \Longrightarrow_{\mathsf{NFS}}$
$\{\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\};\ [\mathrm{N}_1 \mapsto \{f\}] \Longrightarrow_{\mathsf{NFS}}$
$\{\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\};\ [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a, a_1\}] \Longrightarrow_{\mathsf{NFS}}$
$\{\mathrm{M} \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\};\ [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a, a_1\}, \mathrm{N}_3 \mapsto \{c, c_1\}] \Longrightarrow_{\mathsf{NN2}}$
$\{\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} \mathrm{M},\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\};\ [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a, a_1\}, \mathrm{N}_3 \mapsto \{c, c_1\}] \Longrightarrow_{\mathsf{NN1}}$
$\{\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{M}\};\ [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a_1\}, \mathrm{N}_3 \mapsto \{c, c_1\}, \mathrm{M} \mapsto \{b\}] \Longrightarrow_{\mathsf{NN1}}$
$\emptyset;\ [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a_1\}, \mathrm{N}_3 \mapsto \{c_1\}, \mathrm{M} \mapsto \{b\}]$.

Hence, the computed solution is

$\Phi = [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a_1\}, \mathrm{N}_3 \mapsto \{c_1\}, \mathrm{M} \mapsto \{b\}]$,
$\sigma = \{x \mapsto \mathrm{N}_1(\mathrm{N}_2, \mathrm{N}_3),\ y_1 \mapsto \mathrm{N}_2,\ y_2 \mapsto \mathrm{N}_3,\ y \mapsto \mathrm{M}\}$.
$\sigma|_{var(s) \cup var(t)} = \{x \mapsto \mathrm{N}_1(\mathrm{N}_2, \mathrm{N}_3),\ y \mapsto \mathrm{M}\}$.

If we used the same new variable, say y', for both occurrences of y in $f(y, y)$ (instead of using y_1 and y_2 as above), we would get the following pre-unification derivation:

$\{\underline{p(x, x) \simeq^?_{\mathcal{R},\lambda} q(f(y, y), f(a, c))}\};\ \emptyset;\ Id \Longrightarrow_{\mathsf{Dec}}$
$\{\underline{x \simeq^?_{\mathcal{R},\lambda} f(y, y)},\ x \simeq^?_{\mathcal{R},\lambda} f(a, c)\};\ \{p \approx^?_{\mathcal{R},\lambda} q\};\ Id \Longrightarrow_{\mathsf{VE}}$
$\{\underline{\mathrm{N}_1(y', y') \simeq^?_{\mathcal{R},\lambda} f(y, y)},\ \mathrm{N}_1(y', y') \simeq^?_{\mathcal{R},\lambda} f(a, c)\};\ \{p \approx^?_{\mathcal{R},\lambda} q\};\ \{x \mapsto \mathrm{N}_1(y', y')\} \Longrightarrow_{\mathsf{Dec}}$
$\{y' \simeq^?_{\mathcal{R},\lambda} y,\ \underline{\mathrm{N}(y', y') \simeq^?_{\mathcal{R},\lambda} f(a, c)}\};\ \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f\};\ \{x \mapsto \mathrm{N}_1(y', y')\} \Longrightarrow_{\mathsf{Dec}}$
$\{y' \simeq^?_{\mathcal{R},\lambda} y,\ \underline{y' \simeq^?_{\mathcal{R},\lambda} a},\ y' \simeq^?_{\mathcal{R},\lambda} c\};\ \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f\};\ \{x \mapsto \mathrm{N}_1(y', y')\} \Longrightarrow_{\mathsf{VE,Dec}}$
$\{\underline{\mathrm{N}_2 \simeq^?_{\mathcal{R},\lambda} y},\ \mathrm{N}_2 \simeq^?_{\mathcal{R},\lambda} c\};\ \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a\};\ \{x \mapsto \mathrm{N}_1(\mathrm{N}_2, \mathrm{N}_2),\ y' \mapsto \mathrm{N}_2\} \Longrightarrow_{\mathsf{VE,Dec}}$
$\{\underline{\mathrm{N}_2 \simeq^?_{\mathcal{R},\lambda} \mathrm{N}_3}\};\ \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c\};$
$\quad \{x \mapsto \mathrm{N}_1(\mathrm{N}_2, \mathrm{N}_3),\ y' \mapsto \mathrm{N}_2,\ y \mapsto \mathrm{N}_2\} \Longrightarrow_{\mathsf{Dec}}$
$\emptyset;\ \{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ \{x \mapsto \mathrm{N}_1(\mathrm{N}_2, \mathrm{N}_2),\ y' \mapsto \mathrm{N}_2,\ y \mapsto \mathrm{N}_2\}$.

If $\mathcal{R}_\lambda = \{(a, a_1), (a_1, b), (b, c_1), (c_1, c), (p, q)\}$, the obtained neighborhood constraint does not have a solution:

$\{p \approx^?_{\mathcal{R},\lambda} q,\ \mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ \emptyset \Longrightarrow_{\mathsf{FFS}}$
$\{\mathrm{N}_1 \approx^?_{\mathcal{R},\lambda} f,\ \mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ \emptyset \Longrightarrow_{\mathsf{NFS}}$
$\{\mathrm{N}_2 \approx^?_{\mathcal{R},\lambda} a,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ [\mathrm{N}_1 \mapsto \{f\}] \Longrightarrow_{\mathsf{NFS}}$
$\{\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} c,\ \mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a, a_1\}] \Longrightarrow_{\mathsf{NFS}}$
$\{\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a, a_1\}, \mathrm{N}_3 \mapsto \{c, c_1\}]$

From here, NN1 generates two branches. First:

$\{\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a, a_1\}, \mathrm{N}_3 \mapsto \{c, c_1\}] \Longrightarrow_{\mathsf{NN1}}$
$\emptyset;\ [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a\}, \mathrm{N}_3 \mapsto \emptyset] \Longrightarrow_{\mathsf{Fail2}} \bot$.

Second:

$\{\mathrm{N}_3 \approx^?_{\mathcal{R},\lambda} \mathrm{N}_2\};\ [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a, a_1\}, \mathrm{N}_3 \mapsto \{c, c_1\}] \Longrightarrow_{\mathsf{NN1}}$
$\emptyset;\ [\mathrm{N}_1 \mapsto \{f\}, \mathrm{N}_2 \mapsto \{a_1\}, \mathrm{N}_3 \mapsto \emptyset] \Longrightarrow_{\mathsf{Fail2}} \bot$.

Asymmetric Unification and Disunification for the theory of Abelian groups with a homomorphism (*AGh*)

Veena Ravishankar[1], Paliath Narendran[2], and Kimberly A. Cornell[3]

[1] University of Mary Washington
vravisha@umw.edu
[2] University at Albany-SUNY
pnarendran@albany.edu
[3] The College of Saint Rose
cornellk@strose.edu

1 Introduction and Motivation

We compare asymmetric unification [6] and disunification [3, 5] in terms of decidability with respect to the theory of Abelian groups with a homomorphism. Asymmetric unification is a new paradigm comparatively, which requires one side of the equation to be irreducible [6]. Asymmetric Unification is a type of equational unification where the right-hand sides of the equations have to be in normal form with respect to the given term rewriting system. Asymmetric unification is used heavily in symbolic cryptographic protocol analysis. For example, it is used in protocol analyzers, such as Maude-NPA, as a technique for state space reduction by eliminating infeasible states [6]. Methods for reducing this exponential search space are crucial in protocol analysis. In disunification [5] we solve equations and disequations with respect to an equational theory. Disequations allows additional contraints to be imposed such as a variable is not equivalent to a term by our equational theory E (e.g., $x \not\approx_E a$). Disunification has applications in Logic Programming and Artificial Intelligence [4].

Time complexity analysis has been performed on both asymmetric unification and disunification separately [3, 4] and more recently it was shown that they are incomparable [10] with respect to time complexity, but not much work has been done on contrasting the two paradigms in terms of decidability. In [6], it was shown that there are theories which are decidable for symmetric unification but are undecidable for asymmetric unification, so here we investigate this further. Asymmetric unification over the rewrite theory of Abelian groups with a homomorphism (*AGh*) is also of special interest in the field of cryptographic protocol analysis.

2 Preliminaries

Definition 1. *Asymmetric unification: Given a* decomposition *[6]* (Σ, E, R) *of an equational theory, a substitution* σ *is an asymmetric* R,E*-unifier of a set* Q *of asymmetric equations* $\{s_1 \approx^?_\downarrow t_1, \ldots, s_n \approx^?_\downarrow t_n\}$ *iff for each asymmetric equation* $s_i \approx^?_\downarrow t_i$*,* σ *is an* $(E \cup R)$*-unifier of the equation* $s_i \approx^? t_i$*, and* $\sigma(t_i)$ *is in* R,E*-normal form. In other words,* $\sigma(s_i) \to^!_{R,E} \sigma(t_i)$.

Example 1: Let $R = \{x + a \to x\}$ be a rewrite system. An asymmetric unifier θ for $\{u + v \approx^?_\downarrow v + w\}$ modulo this system is $\theta = \{u \mapsto v, w \mapsto v\}$. However, another unifier $\rho = \{u \mapsto a, v \mapsto a, w \mapsto a\}$ is not an asymmetric unifier, since $a + a$ reduces to a. But note that $\theta \preceq_E \rho$, i.e., ρ is an *instance* of θ,

or, alternatively, θ is more general than ρ. This shows that instances of asymmetric unifiers need not be asymmetric unifiers.

Definition 2. *Disunification: For an equational theory E, a disunification problem is a set of equations and disequations* $\mathscr{L} = \{s_1 \approx^?_E t_1, \ldots, s_n \approx^?_E t_n\} \bigcup \{s_{n+1} \not\approx^?_E t_{n+1}, \ldots, s_{n+m} \not\approx^?_E t_{n+m}\}$. *A solution to this problem is a substitution* σ, *known as a disunifier, such that:* $\sigma(s_i) \approx_E \sigma(t_i) \quad (i = 1, \ldots, n)$ *and* $\sigma(s_{n+j}) \not\approx_E \sigma(t_{n+j}) \quad (j = 1, \ldots, m)$.

Example 2: Given $E = \{x + a \approx x\}$, a disunifier θ for $\{u + v \not\approx_E v + u\}$ is $\theta = \{u \mapsto a, v \mapsto b\}$.

If $a + x \approx x$ is added to the identities E, then $\theta = \{u \mapsto a, v \mapsto b\}$ is clearly no longer a disunifier modulo this equational theory.

The *ground disunification problem* [3] for an equational theory Δ, denoted as (Γ, C) consists of a set of constants C and a set Γ of equations and disequations over terms from $T(Sig(\Gamma) \cup C, V)$. For any solution σ, $\mathscr{VR}an(\sigma) \subset T(\Sigma \cup C)$.

3 Decidability Results

Consider the following rewrite system R_1 for *AGh* modulo associativity and commutativity:

$$\begin{array}{llll} i(i(x)) \to x, & i(x+y) \to i(x) + i(y), & x + 0 \to x, & h(i(x)) \to i(h(x)), \quad h(0) \to 0, \\ i(0) \to 0, & x + (i(x) + y) \to y, & x + i(x) \to 0, & h(x+y) \to h(x) + h(y) \end{array}$$

Lemma 3.1. $\longrightarrow_{R_1, AC}$ *is AC-convergent.*

We show disunification is decidable for *AGh* and asymmetric unification for (R_1, AC) is undecidable.

3.1 Disunification is Decidable

Disunification modulo *AGh* can be reduced to solving linear equations over a ring of polynomials with integer coefficients [2, 9]. Solutions to these linear equations is known as first modules of Syzygy [1, 7] from which we can obtain a set of generators. Once we have a particular solution and a set of generators, we check whether there is a solution that also satisfies the disequations. Particularly, we solve the ground disunification problem where no new constants are allowed, except for the ones specified in input already. We show that the ground disunification problem for *AGh* is decidable.

Consider a set of equations and disequations over a set of variables $\{x_1, \ldots, x_m\}$. Let the disequations be of the form $\{x_{i_1} \not\approx^?_{R_1} 0, x_{i_2} \not\approx^?_{R_1} 0, \cdots, x_{i_k} \not\approx^?_{R_1} 0\}$. We express the general solution of the set of *equations* using a matrix where each column is a generator of the syzygy module. Let n be the number of generators. Thus the general solution can be expressed as follows: $\overset{r_1 \quad r_2 \quad \cdots \quad r_n}{\begin{bmatrix} & & Q & & \end{bmatrix}} \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}$

We have the solution represented as: $\begin{bmatrix} c_1 \\ \vdots \\ c_m \end{bmatrix} + z_1 \overset{r_1}{\begin{bmatrix} \\ \end{bmatrix}} + z_2 \overset{r_2}{\begin{bmatrix} \\ \end{bmatrix}} + \cdots + z_n \overset{r_n}{\begin{bmatrix} \\ \end{bmatrix}}$ where $\begin{bmatrix} c_1 \\ \vdots \\ c_m \end{bmatrix}$ is a

particular solution and the rest the general solution.

Thus, the set of solutions can be represented as: $\left[\begin{array}{c|ccc} c_1 & & & \\ \vdots & & & \\ c_m & & & \end{array}\right] \begin{matrix} r_1 & r_2 & \cdots & r_n \end{matrix} \left[\begin{array}{c} 1 \\ z_1 \\ z_2 \\ \vdots \\ z_n \end{array}\right]$

If $x_j \not\approx^?_{R_1} 0$ is the disequation to be satisfied, either c_j must be non-zero or the specific value or component r_{k_j} in some generator r_k must be non-zero, i.e., the j^{th} component in the generator r_k must be non-zero. This check is repeated for each of the disequations.

The following technique will produce a substitution that satisfies all disequations. The key idea is to pick a large enough number and use that to instantiate $z_1, \ldots, z_n$. Pick the largest coefficient (in terms of absolute value) l from both the particular solution and the generators and set $L = |l| + 1$. Now the substitution is $\begin{bmatrix} c_1 \\ \vdots \\ c_m \end{bmatrix} + L \overset{r_1}{\begin{bmatrix} \ \end{bmatrix}} + L^2 \overset{r_2}{\begin{bmatrix} \ \end{bmatrix}} + \cdots + L^n \overset{r_n}{\begin{bmatrix} \ \end{bmatrix}}$ i.e., we set $z_1 = L$, $z_2 = L^2$, $\ldots$, $z_n = L^n$.

Lemma 3.2. *Let $a_0, a_1, \cdots, a_k$ be any sequence of integers, where at least one of the a_i's is non-zero and let $L = 1 + max\{\ |a_i|\ \}$. Then $a_0 + a_1 M + a_2 M^2 + \cdots + a_n M^n \neq 0$ for any $M \geq L$.*

Corollary 1. *Let $a_0, a_1, \cdots, a_k$ be any sequence of integers where $a_0 \neq 0$. Let $L = 1 + max\{\ |a_i|\ \}$. Then $a_0 + a_1 M + a_2 M^2 + \cdots + a_n M^n \neq 0$ for any $M \geq L$.*

Proof. Suppose $x_j \not\approx^?_{R_1} 0$ is a disequation to be satisfied. Our substitution for x_j, say $\theta(x_j)$, is $c_j + L\, r_{1_j} + L^2\, r_{2_j} + \cdots + L^n\, r_{n_j}$ where each c_j, r_{i_j} is a univariate polynomial, with h as the indeterminate, of the form $c_j = \sum_{l=0}^{n} a_l h^l$. For example if $c_j = 2 - 3h^2 + h^5$, $r_{1_j} = 0$ and $r_{2_j} = -2 + 3h - h^5 + h^6$, then $L = 4$. After applying our substitution $\theta(x_j)$ we obtain the solution $-30 + 48h - 3h^2 - 15h^5 + 16h^6$, which satisfies all the disequations.

We now have to show that if at least one of $\{c_j, r_{1_j}, \ldots, r_{n_j}\}$ is non-zero, then $\theta(x_j) \neq 0$. Let h^k be a term which has a non-zero coefficient in one of the polynomials $\{c_j, r_{1_j}, \ldots, r_{n_j}\}$. Then the coefficient of h^k in $\theta(x_j)$ is $a_k + L\, a_{1_j} + L^2\, a_{2_j} + \cdots + L^n\, a_{n_j}$, where a_k is the coefficient of h^k in c_j and a_{i_j} is the coefficient of h^k in r_{i_j}. Since the sequence $a_k, a_{1_j}, \ldots, a_{n_j}$ has at least one non-zero element and $L \geq 1 + max\{a_k, a_{1_j}, \ldots, a_{n_j}\}$, $a_k + L\, a_{1_j} + L^2\, a_{2_j} + \cdots + L^n\, a_{n_j} \neq 0$ by Lemma 3.2. Hence $\theta(x_j) \neq 0$.

□

Theorem 3.1. *Disunification modulo AGh is decidable.*

Proof. Decidability of disunification modulo $\longrightarrow_{R_1,AC}$ follows from the construction provided above.

□

3.2 Asymmetric Unification is Undecidable

We show that asymmetric unification is undecidable for $\longrightarrow_{R_1,AC}$.

Lemma 3.3. *Let θ be any solution to the set of asymmetric equations $\left\{h(X)+b \approx^?_\downarrow X+Y,\quad Z \approx^?_\downarrow h(Y)\right\}$. Then $\theta = \left\{Y \mapsto h^k(b),\, Z \mapsto h^{k+1}(b),\, X \mapsto b+h(b)+\cdots+h^{k-1}(b)\right\}$ for some $k \geq 1$.*

Proof. Y cannot contain any $+$-terms or 0 since this will result in a reduction. Y cannot be equal to b either for the same reason since the only solution of $h(X) \approx^? X$ is $X = 0$. Y also cannot be a term of the form $h^k(c)$ since the equation $h(X)+b \approx_\downarrow X+h^k(c)$ has no solution. Hence $Y \mapsto h^k(b)$ for some $k \geq 0$ is the only possibility. Then $Z = h^{k+1}(b)$. For any $k \geq 1$, the only solution of the equation $h(X)+i(X) \approx^? h^k(b)+i(b)$ is $X = b+h(b)+\cdots+h^{k-1}(b)$ since $(h^k-1)/(h-1) = 1+h+h^2+\cdots+h^{k-1}$. □

Lemma 3.4. *Every solution to the set of asymmetric equations $\left\{h^d(X)+b \approx^?_\downarrow X+Y,\quad Z \approx^?_\downarrow h(Y)\right\}$ where d is a positive integer, is of the form $\theta = \left\{Y \mapsto h^{dk}(b),\, Z \mapsto h^{d(k+1)}(b),\, X \mapsto b+h^d(b)+\cdots+h^{d(k-1)}(b)\right\}$ for some $k \geq 1$.*

We call a term *simple* if it is of the form $h^j(b)$ or $i(h^j(b))$ where $j \geq 0$; j is called the degree of the term. A simple term is called *negative* if it is of the form $i(h^j(b))$ and *positive* otherwise.

Lemma 3.5. *Let $Z = h^k(b)$ for some $k > 0$. Let θ be any solution of the set of asymmetric equations $\left\{h(U) \approx^?_\downarrow U',\ U'+Y \approx^?_\downarrow U+U+Z,\, U+U+Z \approx^?_\downarrow U'+Y\right\}$. Then $\theta(Y)$ contains no negative simple terms, the highest degree in $\theta(Y)$ is less than k, and the highest degree in $\theta(U')$ is k.*

Proof. We first show that neither U' nor Y can contain a negative simple term.

If Y has a simple negative term, then it cannot get cancelled by a simple positive term in U' because of asymmetry. Similarly if U contains terms with negative coefficients they will not get cancelled out either. Let $i(h^m(b))$ be a negative term of the highest degree that occurs in $U'+Y$. Since cancellation is not possible, this term must also occur in U. But then U' contains a term of degree $m+1$, i.e., one more than U. This leads to a contradiction.

Next we show the highest degree of $\theta(Y)$ is less than k and the highest degree of $\theta(U')$ is k.

First we show that the highest degree of a term in $U'+Y$ is not greater than k. Let l be the highest degree in $U'+Y$. We prove that l cannot be greater than k. If $l > k$, i.e., $U'+Y$ contains $h^l(b)$, then $h^l(b)$ must also be in U to cancel out the term. If l is the highest degree of U, then the highest degree in U' is $l+1$, which is a contradiction. Hence k is the highest degree of a term in $U'+Y$.

Suppose k is the highest degree of a term in Y. Let $Y \approx^? Y'+h^k(b)$. Then we have $U'+Y' \approx^? U+U$, which has no solution, since $U' \approx^? h(U)$. Hence the highest degree of a term in Y has to be less than k. Since the highest degree of a term in $U'+Y$ is no more than k, the highest degree possible for a term in U' is k as well. □

We represent terms using polynomials over $\mathbb{N}$ as done in [2]: for example, $b+b+b$ is represented as $3(b)$ and $h^4(b)+b+b+b+b+b$ is represented as $(h^4+5)(b)$. We call terms of the form $n(b)$ where n is a natural number (i.e., a summation of n $b's$) as *b-sum-terms*.

One of the solutions for the asymmetric equations in the above lemma is $\left\{Y \mapsto 2^k(b),\, U \mapsto (h^{k-1}+2h^{k-2}+4h^{k-3}+8h^{k-4}\cdots+2^{k-1})(b),\, U' \mapsto (h^k+2h^{k-1}+4h^{k-2}+\cdots+2^{k-1})(b)\right\}$

Next we have a couple of lemmas about equations over polynomials.

Lemma 3.6. *Let x be an indeterminate and let W_1, W_2, X_1, X_2, X_3 be variables. Suppose $W_1 = x^m$ and $W_2 = x^{2n}$. Then the set of equations*

$$\{(x-1)X_1 =^? W_1-1,\ (x^2-1)X_2 =^? W_2-1,\ (x-1)X_3 =^? X_1-X_2\}$$

is solvable iff m = n.

Proof. In the first equation $X_1 = (1+x+\cdots+x^{m-1})$ and in the second equation $X_2 = (1+x^2+\cdots+x^{2(n-1)})$. $X_2 - X_1$ is divisible by $x-1$ if and only if $m = n$, since evaluating X_1 and X_2 at $x = 1$ results in $1+1+\cdots+1^{m-1} - (1+1^2+\cdots+1^{2(n-1)})$. □

Corollary 2. *Let $W_1 = h^m(b)$ and $W_2 = h^{2n}(b)$. Then the set of equations $\{h(X_1)+b \approx^?_\downarrow W_1+X_1$, $Y_1 \approx^?_\downarrow h(W_1)$, $h^2(X_2)+b \approx^?_\downarrow W_2+X_2$, $Y_2 \approx^?_\downarrow h(W_2)$, $h(X_3)+X_2 \approx^?_\downarrow X_3+X_1\}$ is solvable iff m = n.*

Lemma 3.7. *Let $W_1 = x^n$, $W_2 = x^{2n}$ and $P \in \mathbb{N}[x]$. Then the equations $(x-2)Y =^? W_1 - P$, $(x^2-2)Z =^? W_2 - P$ has a solution iff $P = 2^n$.*

Proof. Evaluating P at $x = 2$ and $x = \sqrt{2}$ gives us the same value 2^n and since P cannot have any negative coefficients, this is possible only if $P = 2^n$. □

Corollary 3. *Let $W_1 = h^n(b)$, $W_2 = h^{2n}(b)$ and P be a term that contains no negative simple terms. Then the equations $h(Y)+P \approx^?_\downarrow Y+Y+W_1$, $h^2(Z)+P \approx^?_\downarrow Z+Z+W_2$ has a solution iff $P = 2^n(b)$.*

Lemma 3.8. *Let $Z = x^n$ and v_1, $v_2 \in \mathbb{N}$, then the equations $(x-1)Y_1 =^? Z-1$, $(x-1)Y_2 =^? Y_1 - v_1$, $(x-1)Y_3 =^? Y_2 - v_2$ have a solution if and only if $v_1 = n$ and $v_2 = n(n-1)/2$.*

Proof. The proof follows from Lemma 3.5 in [8]. □

Corollary 4. *Let $V_1 = v_1(b)$ and $V_2 = v_2(b)$ where v_1, v_2 are natural numbers, then the equations $h(Y_1)+b \approx^?_\downarrow Y_1+Z$, $Z_3 \approx^?_\downarrow h(Z)$, $h(Y_2)+V_1 \approx^?_\downarrow Y_1+Y_2$, $h(Y_3)+V_2 \approx^?_\downarrow Y_2+Y_3$ have a solution if and only if $v_2 = v_1(v_1-1)/2$.*

Putting all equations together we can see that the following set of asymmetric equations forces V_1 and V_2 to be b-sum-terms: $h(X_1)+b \approx^?_\downarrow W_1+X_1$, $Z_1 \approx^?_\downarrow h(W_1)$, $h^2(X_2)+b \approx^?_\downarrow W_2+X_2$, $Z_2 \approx^?_\downarrow h(W_2)$, $h(X_3)+X_2 \approx^?_\downarrow X_1+X_3$, $h(Y_1)+P \approx^?_\downarrow Y_1+Y_1+W_1$, $h^2(Y_2)+P \approx^?_\downarrow Y_2+Y_2+W_2$, $P \approx^?_\downarrow V_1+V_2+V$

Theorem 3.2. *Asymmetric Unification modulo $\longrightarrow_{R_1,AC}$ is undecidable.*

Proof. We reduce Hilbert's Tenth Problem to our asymmetric unification problem in the following manner. We represent natural numbers by b-sum-terms. For example natural number 3 is represented as $b+b+b$. Addition can be simulated in a straightforward way using the $+$ symbol, since $v_1(b)+v_2(b)$ is the same as $(v_1+v_2)(b)$. Multiplication can be simulated using the identity

$$xy = \frac{(x+y)(x+y-1)}{2} - \frac{x(x-1)}{2} - \frac{y(y-1)}{2}$$ □

We now illustrate this with an example. Suppose we have a diophantine equation $z = x^2+y$. This can be 'decomposed' into a set of simple equations as follows: $z_1 = x*x$, $z = z_1+y$. Clearly these two equations have a solution if and only if the original equation has a solution.

The first equation can be transformed into $z_1 = \frac{(x+x)(x+x-1)}{2} - \frac{x(x-1)}{2} - \frac{x(x-1)}{2}$ and further into $z_2 = x+x$, $z_3 = \frac{z_2(z_2-1)}{2}$, $z_4 = \frac{x(x-1)}{2}$, $z_3 = z_1+z_4+z_4$

We represent the second equation using Corollary 4 in the following manner: $h(Y_1)+b \approx^?_\downarrow Y_1+Z$, $Z_1 \approx^?_\downarrow h(Z)$, $h(Y_2)+Z_2 \approx^?_\downarrow Y_1+Y_2$, $h(Y_3)+Z_3 \approx^?_\downarrow Y_2+Y_3$ where $Z_2 = z_2(b)$ and $Z_3 = z_3(b)$.

4 Conclusion and Future Work

We proved that asymmetric unification is undecidable for $\longrightarrow_{R_1,AC}$ and disunification is decidable for *AGh*. We are interested in finding theories for which asymmetric unification is decidable and disunification is undecidable. Another future direction is finding a different decomposition of the identities in *AGh* to obtain an asymmetric decision procedure.

References

[1] Matthias Aschenbrenner. Ideal membership in polynomial rings over the integers. *Journal of the American Mathematical Society*, 17(2):407–441, 2004.

[2] Franz Baader. Unification in Commutative Theories, Hilbert's Basis Theorem, and Gröbner Bases. *Journal of the ACM*, 40(3):477–503, July 1993.

[3] Franz Baader and Klaus U. Schulz. Combination techniques and decision problems for disunification. *Theoretical Computer Science*, 142(2):229–255, 1995.

[4] Wray L. Buntine and Hans-Jürgen Bürckert. On solving equations and disequations. *J. ACM*, 41(4):591–629, 1994.

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

[6] Serdar Erbatur, Santiago Escobar, Deepak Kapur, Zhiqiang Liu, Christopher A. Lynch, Catherine Meadows, José Meseguer, Paliath Narendran, Sonia Santiago, and Ralf Sasse. Asymmetric Unification: A New Unification Paradigm for Cryptographic Protocol Analysis. In *Automated Deduction, (CADE-24)*, volume 7898 of *LNCS*, pages 231–248. 2013.

[7] Zhiping Lin. On syzygy modules for polynomial matrices. *Linear Algebra and its Applications*, 298(1):73 – 86, 1999.

[8] Paliath Narendran. Solving linear equations over polynomial semirings. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 466–472. IEEE Computer Society, 1996.

[9] Werner Nutt. Unification in monoidal theories. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 618–632, Berlin, Heidelberg, 1990. Springer.

[10] Veena Ravishankar, Kimberly A. Cornell, and Paliath Narendran. Asymmetric Unification and Disunification. Springer, June 2019. To be published in LNCS Festshrift Series.