CADE 23

the 23rd International Conference on Automated Deduction

University of Wrocław, Wrocław, Poland

31 July - 5 August 2011

UNIF 2011

The 25th International Workshop on Unification

Proceedings

Editors:

Franz Baader   Barbara Morawska   Jan Otop

31 July 2011

Preface

UNIF 2011 was the 25th event in a series of international meetings devoted to unification theory and its applications. Unification is concerned with the problem of identifying given terms, either syntactically or modulo an equational theory. The aim of UNIF 2011, as that of the previous meetings, was to bring together researchers interested in unification theory and related topics, to present recent (even unfinished) work, and discuss new ideas and trends in this and related fields.

Program Committee:

Franz Baader(Chair)	(TU Dresden)
Maribel Fernández	(King's College London)
Jordi Levy	(Spanish National Research Council (CSIC))
Markus Lohrey	(University of Leipzig)
Mircea Marin	
Barbara Morawska	(TU Dresden)
Paliath Narendran	(University at Albany–SUNY)
Jan Otop	(University of Wrocław)
Manfred Schmidt-Schauß	(Frankfurt University)
Ralf Treinen	(Université Paris Didero)

Organization Committee:

Franz Baader	(TU Dresden)
Barbara Morawska	(TU Dresden)
Jan Otop	(University of Wrocław)

UNIF 2011 Schedule:

9.00-10.00 morning session

- **9:00-10:00** INVITED TALK by Christopher Lynch *Unification in Cryptographic Protocol Analysis*

10.00-10.30 coffee break

10.30-12.00 morning session

- **10:30-11:00** Franz Baader, Nguyen Thanh Binh, Stefan Borgwardt and Barbara Morawska *Computing Local Unifiers in the Description Logic EL without the Top Concept*
- **11:00-11:30** Jan Otop *Unification of anti-terms*
- **11:30-12:00** Łukasz Stafiniak *Joint Constraint Abduction Problems*
- **12:00-12:30** Wojciech Dzik and Piotr Wojtylak *Projective Unifiers in Modal Logics*

12.30-14.00 lunch break

14.00-15.30 afternoon session

- **14.00-14.30** Ştefan Ciobâcă *Computing finite variants for subterm convergent rewrite systems*
- **14:30-15:00** Conrad Rau and Manfred Schmidt-Schauß *A Unification Algorithm to Compute Overlaps in a Call-by-Need Lambda-Calculus with Variable-Binding Chains*
- **15:00-15:30** Ben Kavanagh and James Cheney *Higher-Order Unification for the lambda-alpha-nu calculus*

15.30-16.00 coffee break

16.00-17.30 afternoon session

- **16:00-16:30** Rakesh Verma and Wei Guo *Does Unification Help in Normalization?*
- **16:30-17:00** Serdar Erbatur, Santiago Escobar, Deepak Kapur, Zhiqiang Liu, Christopher Lynch, Catherine Meadows, José Meseguer, Paliath Narendran and Ralf Sasse *Asymmetric Unification: A New Unification Paradigm for Cryptographic Protocol Analysis*
- **17:00-17:30** Business meeting

Table of Contents

Invited talk: Unification in Cryptographic Protocol Analysis 1
Christopher Lynch

Computing Local Unifiers in the Description Logic $\mathcal{EL}$ without the Top Concept 2
Franz Baader, Nguyen Thanh Binh, Stefan Borgwardt, and Barbara Morawska

Unification of anti-terms 9
Jan Otop

Joint Constraint Abduction Problems 15
Łukasz Stafiniak

Projective Unifiers in Modal Logics 21
Wojciech Dzik and Piotr Wojtylak

Computing finite variants for subterm convergent rewrite systems 28
Ştefan Ciobâcă

A Unification Algorithm to Compute Overlaps in a Call-by-Need Lambda-Calculus with Variable-Binding Chains 35
Conrad Rau and Manfred Schmidt-Schauß

Higher-Order Unification for the $\lambda_{\alpha\nu}$ calculus 42
Ben Kavanagh and James Cheney

Does Unification Help in Normalization? 52
Rakesh M. Verma and Wei Guo

Asymmetric Unification: A New Unification Paradigm for Cryptographic Protocol Analysis 59
Serdar Erbatur, Santiago Escobar, Deepak Kapur, Zhiqiang Liu, Christopher Lynch, Catherine Meadows, José Meseguer, Paliath Narendran, and Ralf Sasse

Invited talk: Unification in Cryptographic Protocol Analysis

Christopher Lynch

Clarkson University, Potsdam, NY, USA
clynch@clarkson.edu

Abstract. Symbolic cryptographic protocol analysis represents messages as ground terms. Rules are given from the point of view of each principal, indicating that if a principal receives a message of a particular type then that principal will send out a message of a particular type. Variables are used to indicated that parts of a message the principal receives are previously unknown to the principal, and therefore the principal will accept anything for those parts of the message. We assume that a malicious intruder can see everything on the network, read and modify messages, and masquerade as the principals. Every message that is passed on the network will be known to the intruder, so everything is represented as intruder knowledge. The intruder also has the ability to perform functions on the data. For example, if an intruder knows a message m and a key k then the intruder can encrypt m with k. Also, if the intruder knows a message m encrypted with k, and the intruder knows k, then the intruder will know m. Intruders have some initial knowledge. Cryptographic protocol analysis asks the question whether the intruder can mount an attack, which is often equivalent to asking whether the intruder can learn a particular goal message, such as a secret key.

The problem of cryptographic protocol analysis is to determine whether the intruder, given the initial knowledge, actions of the principals, and abilities of the intruders, can learn the goal message. This is a deduction problem, which is solved by working through the search space. Unification is a key component of this search, because we must unify messages that are sent with messages that are expected to be received.

Much of the work on cryptographic protocol analysis considers the encryption function as a black box. However, this level of abstraction misses many attacks. In reality, encryption algorithms have certain properties. For example, exclusive OR is often used in encryption, and XOR is associative, commutative, nilpotent and has an identity. Encryption algorithms often involve multiplication which has Abelian group properties. Homomorphisms are common in encryption, representing the fact that an operation can be performed on encrypted data which acts on the data itself. In order to reason about these algebraic theories, unification must be performed modulo an equational theory.

We will discuss the work that has been done on incorporating equational unification into cryptographic protocol analysis. New algorithms have been developed, because it is important to not only have efficient algorithms, but also algorithms that create a small complete set of unifiers, to avoid blowing up the search space. We will discuss new techniques used to develop these algorithms.

Computing Local Unifiers in the Description Logic $\mathcal{EL}$ without the Top Concept

Franz Baader[1⋆], Nguyen Thanh Binh[2], Stefan Borgwardt[1⋆], and Barbara Morawska[1⋆]

[1] TU Dresden, Germany, {baader,stefborg,morawska}@tcs.inf.tu-dresden.de
[2] ETH Zürich, Switzerland, thannguy@inf.ethz.ch

Introduction

Unification in Description Logics (DLs) has been proposed in [7] as a novel inference service that can, for example, be used to detect redundancies in ontologies. For instance, assume that one knowledge engineer defines the concept of *professors that are mothers* as $\mathsf{Person} \sqcap \mathsf{Female} \sqcap \exists \mathsf{child}.\top \sqcap \exists \mathsf{job}.\mathsf{Professor}$, whereas another knowledge engineer represents this notion in a somewhat different way, e.g., by using the concept term $\mathsf{Mother} \sqcap \exists \mathsf{job}.(\mathsf{Teacher} \sqcap \mathsf{Researcher})$. These two concept terms are not equivalent, i.e., they are not interpreted by the same set of individuals in every interpretation, but they are nevertheless meant to represent the same concept. They can obviously be made equivalent by substituting the concept name Professor in the first term by the concept term $\mathsf{Teacher} \sqcap \mathsf{Researcher}$ and the concept name Mother in the second term by the concept term $\mathsf{Person} \sqcap \mathsf{Female} \sqcap \exists \mathsf{child}.\top$. We call a substitution that makes two concept terms equivalent a *unifier* of the two terms. Such a unifier proposes definitions for the concept names that are used as variables. In our example, we know that, if we define Mother as $\mathsf{Person} \sqcap \mathsf{Female} \sqcap \exists \mathsf{child}.\top$ and Professor as $\mathsf{Teacher} \sqcap \mathsf{Researcher}$, then the two concept terms from above are equivalent w.r.t. these definitions.

The concept terms of the above example are formulated in the DL $\mathcal{EL}$, which has the concept constructors conjunction ($\sqcap$), existential restriction ($\exists r.C$), and the top concept ($\top$). This DL has recently drawn considerable attention since, on the one hand, important inference problems such as the subsumption problem are polynomial in $\mathcal{EL}$ [1, 4]. On the other hand, though quite inexpressive, $\mathcal{EL}$ can be used to define biomedical ontologies. For example, the large medical ontology SNOMED CT[3] can be expressed in $\mathcal{EL}$. Unification in $\mathcal{EL}$ was first investigated in [5], where it was shown that the decision problem is NP-complete. Basically, the proof that one can check for the existence of an $\mathcal{EL}$-unifier within nondeterministic polynomial time given in [5] proceeds as follows. First, it is shown that any solvable $\mathcal{EL}$-unification problem has a *local* unifier, i.e., a unifier that is "built from atoms" of the input problem. Second, since the definition of locality implies that a local substitution can be guessed in polynomial time,

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

one can test for the existence of a local unifier within NP by guessing a local substitution and then checking whether it is indeed a unifier. In particular, this means that the results of [5] also show how to compute all local unifiers of a given $\mathcal{EL}$-unification problem. In [6] it was shown that one can employ a SAT solver to search for local $\mathcal{EL}$-unifiers.

Actually, if one takes a closer look at the concept definitions in SNOMED CT, then one sees that they do not use the top concept, i.e., SNOMED CT is not formulated in $\mathcal{EL}$, but rather in its sub-logic $\mathcal{EL}^{-\top}$, which differs from $\mathcal{EL}$ in that the use of the top concept is disallowed. If we employ $\mathcal{EL}$-unification to detect redundancies in (extensions of) SNOMED CT, then a unifier may introduce concept terms that contain the top concept, and thus propose definitions for concept names that are of a form that is not used in SNOMED CT. Apart from this practical motivation for investigating unification in $\mathcal{EL}^{-\top}$, we also found it interesting to see how such a small change in the logic influences the unification problem. Surprisingly, it turned out that the complexity of the problem increases considerably: we were able to show in [2] that deciding unifiability in $\mathcal{EL}^{-\top}$ is PSpace-complete. In [2], we restricted the attention to the decision problem, and did not address the problem of how to compute unifiers of solvable $\mathcal{EL}^{-\top}$-unification problems.

In the present paper we introduce a notion of locality for $\mathcal{EL}^{-\top}$-unifiers, and show that we can always compute a local unifier for a solvable $\mathcal{EL}^{-\top}$-unification problem. However, whereas any $\mathcal{EL}$-unification problem has only exponentially many local $\mathcal{EL}$-unifiers, each of which can be represented in polynomial space using structure sharing, a given $\mathcal{EL}^{-\top}$-unification problem can have infinitely many local $\mathcal{EL}^{-\top}$-unifiers. We show that a solvable $\mathcal{EL}^{-\top}$-unification problem always has a local $\mathcal{EL}^{-\top}$-unifier of at most exponential size, which can effectively be computed.

The Description Logics $\mathcal{EL}$ and $\mathcal{EL}^{-\top}$

Starting with a set N_C of concept names and a set N_R of role names, *$\mathcal{EL}$-concept terms* are built using the concept constructors *top-concept* ($\top$), *conjunction* ($C \sqcap D$), and *existential restriction* ($\exists r.C$ for every $r \in N_R$). The $\mathcal{EL}$-concept term C is an *$\mathcal{EL}^{-\top}$-concept term* if $\top$ does not occur in C. Since $\mathcal{EL}^{-\top}$-concept terms are special $\mathcal{EL}$-concept terms, most definitions transfer from $\mathcal{EL}$ to $\mathcal{EL}^{-\top}$, and thus we only formulate them for $\mathcal{EL}$.

The semantics of $\mathcal{EL}$ and $\mathcal{EL}^{-\top}$ is defined in the usual way, using the notion of an interpretation $\mathcal{I} = (\mathcal{D}_\mathcal{I}, \cdot^\mathcal{I})$, which consists of a nonempty domain $\mathcal{D}_\mathcal{I}$ and an interpretation function $\cdot^\mathcal{I}$ that assigns binary relations on $\mathcal{D}_\mathcal{I}$ to role names and subsets of $\mathcal{D}_\mathcal{I}$ to concept terms, as shown in the semantics column of Table 1. The concept term C *is subsumed by* the concept term D (written $C \sqsubseteq D$) iff $C^\mathcal{I} \subseteq D^\mathcal{I}$ holds for all interpretations $\mathcal{I}$. We say that C *is equivalent to* D (written $C \equiv D$) iff $C \sqsubseteq D$ and $D \sqsubseteq C$, i.e., iff $C^\mathcal{I} = D^\mathcal{I}$ holds for all interpretations $\mathcal{I}$.

In order to define locality of unifiers in $\mathcal{EL}$, we need the notion of an atom. An $\mathcal{EL}$-concept term is called an *atom* iff it is a concept name $A \in N_C$ or an

Name	Syntax	Semantics	$\mathcal{EL}$	$\mathcal{EL}^{-\top}$
concept name	A	$A^{\mathcal{I}} \subseteq \mathcal{D}_{\mathcal{I}}$	×	×
role name	r	$r^{\mathcal{I}} \subseteq \mathcal{D}_{\mathcal{I}} \times \mathcal{D}_{\mathcal{I}}$	×	×
top-concept	$\top$	$\top^{\mathcal{I}} = \mathcal{D}_{\mathcal{I}}$	×	
conjunction	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$	×	×
existential restriction	$\exists r.C$	$(\exists r.C)^{\mathcal{I}} = \{x \mid \exists y : (x,y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$	×	×
subsumption	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$	×	×
equivalence	$C \equiv D$	$C^{\mathcal{I}} = D^{\mathcal{I}}$	×	×

Table 1. Syntax and semantics of $\mathcal{EL}$ and $\mathcal{EL}^{-\top}$.

existential restriction $\exists r.D$. Concept names and existential restrictions $\exists r.D$, where D is a concept name or $\top$, are called *flat atoms.* The set $\mathrm{At}(C)$ of *atoms of an $\mathcal{EL}$-concept term C* consists of all the subterms of C that are atoms. For example, $C = A \sqcap \exists r.(B \sqcap \exists r.\top)$ has the atom set $\mathrm{At}(C) = \{A, \exists r.(B \sqcap \exists r.\top), B, \exists r.\top\}$. Obviously, any $\mathcal{EL}$-concept term C is a conjunction $C = C_1 \sqcap \ldots \sqcap C_n$ of atoms and $\top$. We call the atoms among $C_1, \ldots, C_n$ the *top-level atoms* of C. The $\mathcal{EL}$-concept term C is called *flat* if all its top-level atoms are flat.

The notion of a top-level atom allows for a simple *recursive characterization of subsumption* in $\mathcal{EL}$. We have $C \sqsubseteq D$ iff every top-level atom of D subsumes some top-level atom of C. In addition, the only atom subsumed by $A \in N_C$ is A itself, and all atoms subsumed by $\exists r.E$ are of the form $\exists r.E'$ with $E' \sqsubseteq E$.

In order to define locality of unifiers in $\mathcal{EL}^{-\top}$, we additionally need the notion of a particle: $\mathcal{EL}^{-\top}$-concept terms of the form $\exists r_1. \cdots \exists r_n.A$ for $n \geq 0$ role names $r_1, \ldots, r_n$ and a concept name A are called *particles* . The set $\mathrm{Part}(C)$ of all particles of a given $\mathcal{EL}^{-\top}$-concept term C is defined as

- $\mathrm{Part}(C) := \{C\}$ if C is a concept name,
- $\mathrm{Part}(C) := \{\exists r.E \mid E \in \mathrm{Part}(D)\}$ if $C = \exists r.D$,
- $\mathrm{Part}(C) := \mathrm{Part}(C_1) \cup \mathrm{Part}(C_2)$ if $C = C_1 \sqcap C_2$.

For example, the particles of $C = A \sqcap \exists r.(A \sqcap \exists r.B)$ are $A, \exists r.A, \exists r.\exists r.B$.

Unification in $\mathcal{EL}$ and $\mathcal{EL}^{-\top}$

To define unification in $\mathcal{EL}$ and $\mathcal{EL}^{-\top}$ simultaneously, let $\mathcal{L} \in \{\mathcal{EL}, \mathcal{EL}^{-\top}\}$. When defining unification in $\mathcal{L}$, we assume that the set of concept names is partitioned into a set N_v of concept variables (which may be replaced by substitutions) and a set N_c of concept constants (which must not be replaced by substitutions). An *$\mathcal{L}$-substitution* σ is a mapping from N_v into the set of all $\mathcal{L}$-concept terms. This mapping is extended to concept terms in the usual way, i.e., by replacing all occurrences of variables in the term by their σ-images. An $\mathcal{L}$-concept term is called *ground* if it contains no variables, and an $\mathcal{L}$-substitution σ is called *ground* if the concept terms $\sigma(X)$ are ground for all $X \in N_v$.

Unification tries to make concept terms equivalent by applying a substitution.

Definition 0.1. *An* $\mathcal{L}$-unification problem *is of the form* $\Gamma = \{C_1 \equiv^? D_1, \ldots, C_n \equiv^? D_n\}$*, where* $C_1, D_1, \ldots C_n, D_n$ *are* $\mathcal{L}$*-concept terms. The* $\mathcal{L}$*-substitution* σ *is an* $\mathcal{L}$-unifier *of* Γ *iff it solves all the equations* $C_i \equiv^? D_i$ *in* Γ*, i.e., iff* $\sigma(C_i) \equiv \sigma(D_i)$ *for* $i = 1, \ldots, n$*. In this case,* Γ *is called* $\mathcal{L}$-unifiable.

In the following, we will use the subsumption $C \sqsubseteq^? D$ as an abbreviation for the equation $C \sqcap D \equiv^? C$. Obviously, σ solves this equation iff $\sigma(C) \sqsubseteq \sigma(D)$.

Clearly, every $\mathcal{EL}^{-\top}$-unification problem Γ is also an $\mathcal{EL}$-unification problem. Whether Γ is $\mathcal{L}$-unifiable or not may depend, however, on whether $\mathcal{L} = \mathcal{EL}$ or $\mathcal{L} = \mathcal{EL}^{-\top}$. As an example, consider the problem $\Gamma := \{A \sqsubseteq^? X, B \sqsubseteq^? X\}$, where A, B are distinct concept constants and X is a concept variable. Obviously, the substitution that replaces X by $\top$ is an $\mathcal{EL}$-unifier of Γ. However, Γ does not have an $\mathcal{EL}^{-\top}$-unifier. In fact, for such a unifier σ, we would need to have $A \sqsubseteq \sigma(X)$ and $B \sqsubseteq \sigma(X)$, and it is easy to see that this is only possible if $\sigma(X) \equiv \top$.

As shown in [5], we may without loss of generality restrict our attention to ground unifiers of *flat* $\mathcal{L}$*-unification problems*, i.e., unification problems in which the left- and right-hand sides of equations are flat $\mathcal{L}$-concept terms. Given a flat $\mathcal{L}$-unification problem Γ, we denote by $\mathrm{At}(\Gamma)$ the set of all atoms of Γ, i.e., the union of all sets of atoms of the concept terms occurring in Γ. By $\mathrm{Var}(\Gamma)$ we denote the variables that occur in Γ, and by $\mathrm{NV}(\Gamma) := \mathrm{At}(\Gamma) \setminus \mathrm{Var}(\Gamma)$ the set of all *non-variable atoms* of Γ.

Local unifiers

In $\mathcal{EL}$, every solvable unification problem has a *local* $\mathcal{EL}$*-unifier*, i.e., an $\mathcal{EL}$-unifier γ such that, for every variable X, the top-level atoms of $\gamma(X)$ are of the form $\gamma(D)$ for $D \in \mathrm{NV}(\Gamma)$.

Example 0.2. Consider the flat $\mathcal{EL}$-unification problem Γ that consists of the three equations

$$X \equiv^? Y \sqcap A, \quad Y \sqcap \exists r.X \equiv^? \exists r.X, \quad Z \sqcap \exists r.X \equiv^? \exists r.X.$$

Then the substitutions $\sigma_0 := \{X \mapsto A, Y \mapsto \top, Z \mapsto \top\}$ and $\sigma_1 := \{X \mapsto A, Y \mapsto \top, Z \mapsto \exists r.A\}$ are the only local $\mathcal{EL}$-unifiers of Γ. In fact, we have $\mathrm{NV}(\Gamma) = \{A, \exists r.X\}$, and thus the only possible image for X in a local unifier σ is A (since $\sigma(\exists r.X) = \exists r.\sigma(X)$ obviously cannot be a conjunct of $\sigma(X)$). Since the first equation implies that $A = \sigma(X) \sqsubseteq \sigma(Y)$, we know that $\sigma(Y)$ can only be $\top$ or A. However, the second equation prevents the second possibility. Finally, the third equation ensures that $\sigma(Z)$ is $\top$ or $\exists r.A$.

Note that σ_0 and σ_1 both contain $\top$, and thus are not $\mathcal{EL}^{-\top}$-unifiers. This shows that Γ does not have an $\mathcal{EL}^{-\top}$-unifier that is local in the sense defined above. Nevertheless, Γ has an $\mathcal{EL}^{-\top}$-unifier. For example, the substitution $\gamma_1 := \{X \mapsto A \sqcap \exists r.A,\ Y \mapsto \exists r.A, Z \mapsto \exists r.\exists r.A\}$ is such a unifier. Except for the atom A, the top-level atoms of $\gamma_1(X), \gamma_1(Y), \gamma_1(Z)$ are not of the form $\gamma(D)$ for some $D \in \mathrm{NV}(\Gamma)$, but the ones different from A are all particles of $\gamma(D)$ for some $D \in \mathrm{NV}(\Gamma)$. This motivates the following definition.

Definition 0.3. *The $\mathcal{EL}^{-\top}$-unifier γ of Γ is a* local $\mathcal{EL}^{-\top}$-unifier *of Γ if, for every variable X, each top-level atom of $\gamma(X)$ is of the form $\gamma(D)$ for some $D \in \mathrm{NV}(\Gamma)$ or a particle of $\gamma(D)$ for some $D \in \mathrm{NV}(\Gamma)$.*

The unification problem of Example 0.2 can be used to demonstrate that a given $\mathcal{EL}^{-\top}$-unification problem can have infinitely many local $\mathcal{EL}^{-\top}$-unifiers. It is easy to see that the substitutions

$$\gamma_n := \{X \mapsto A \sqcap \exists r.A \sqcap \cdots \sqcap (\exists r.)^n A, Y \mapsto \exists r.A \sqcap \cdots \sqcap (\exists r.)^n A, Z \mapsto (\exists r.)^{n+1} A\}$$

are all local $\mathcal{EL}^{-\top}$-unifiers of Γ in the sense of Definition 0.3. Indeed, every top-level atom of $\gamma_n(X)$, $\gamma_n(Y)$, and $\gamma_n(Z)$ is either A or a particle of $\gamma_n(\exists r.X)$.

We are now ready to formulate the main result of this paper.

Theorem 0.4. *Given a solvable $\mathcal{EL}^{-\top}$-unification problem Γ, we can construct a local $\mathcal{EL}^{-\top}$-unifier of Γ of at most exponential size in time exponential in the size of Γ.*

We now provide a high-level description of the procedure for $\mathcal{EL}^{-\top}$-unification from [2, 3] and show how it can be adapted such that it produces a local $\mathcal{EL}^{-\top}$-unifier of size at most exponential in the size of Γ whenever there is an $\mathcal{EL}^{-\top}$-unifier.

Constructing local $\mathcal{EL}^{-\top}$-unifiers

The first step of the $\mathcal{EL}^{-\top}$-unification procedure reduces $\mathcal{EL}^{-\top}$-unifiability of Γ to solvability of a certain kind of linear language inclusions over the alphabet N_R. These inclusions are of the form $X_i \subseteq L_0 \cup L_1 X_1 \cup \cdots \cup L_n X_n$, where $X_1, \ldots, X_n$ are indeterminates, $i \in \{1, \ldots, n\}$, and each L_i ($i \in \{0, \ldots, n\}$) is a subset of $N_R \cup \{\epsilon\}$. For each variable $X \in N_v$ and each constant $A \in N_c$, there is one indeterminate X_A in these inclusions.

A *solution* θ of such an inclusion assigns sets $\theta(X_i) \subseteq N_R^*$ to the indeterminates such that $\theta(X_i) \subseteq L_0 \cup L_1\theta(X_1) \cup \cdots \cup L_n\theta(X_n)$. A solution to a set $\mathcal{I}$ of such inclusions is called *admissible* if, for every variable $X \in N_v$, there is a constant $A \in N_c$ such that $\theta(X_A)$ is nonempty. This condition will ensure that the constructed unifier of Γ is indeed an $\mathcal{EL}^{-\top}$-substitution, i.e., it does not contain $\top$. We are also only interested in *finite* solutions, i.e., solutions θ such that all the sets $\theta(X_i)$ are finite.

The problem of finding an $\mathcal{EL}^{-\top}$-unifier for Γ can be reduced to the problem of finding a finite, admissible solution to a certain set of such language inclusions. More precisely, there is a set $\mathfrak{F}_\Gamma$ of exponentially many sets $\mathcal{I}$ of language inclusions (of polynomial size) such that Γ is $\mathcal{EL}^{-\top}$-unifiable iff there is a finite, admissible solution for one $\mathcal{I} \in \mathfrak{F}_\Gamma$. This reduction uses nondeterministic polynomial time in the size of Γ since we can guess an element of $\mathfrak{F}_\Gamma$ in polynomial time.

Lemma 0.5. *The $\mathcal{EL}^{-\top}$-unification problem Γ has an $\mathcal{EL}^{-\top}$-unifier iff there is a set $\mathcal{I} \in \mathfrak{F}_\Gamma$ that has a finite, admissible solution.*

In this paper, we are further concerned with *local* solutions and their connection to local $\mathcal{EL}^{-\top}$-unifiers of Γ.

Definition 0.6. *Let $\mathcal{I}$ be a finite set of inclusions of the above form. A solution θ of $\mathcal{I}$ is called* local *if all words $w \in \theta(X) \setminus \{\epsilon\}$ for some indeterminate X occur on the right-hand side of some inclusion $X_i \subseteq L_0 \cup L_1 X_1 \cup \cdots \cup L_n X_n$ under θ, i.e., either $w \in L_0$ or $w \in (L_i \setminus \{\epsilon\})\theta(X_i)$ for some $i \in \{1, \ldots, n\}$.*

The next lemma states the close connection between the two notions of locality.

Lemma 0.7. *If there is a finite, local, admissible solution θ for one $\mathcal{I} \in \mathfrak{F}_\Gamma$, then one can construct a local $\mathcal{EL}^{-\top}$-unifier σ of Γ that is of size at most exponential in the size of Γ and polynomial in the size of θ.*

Example 0.8. One element of $\mathfrak{F}_\Gamma$ for the $\mathcal{EL}^{-\top}$-unification problem Γ from Example 0.2 consists of the inclusions

$$Y_A \subseteq X_A, \quad X_A \subseteq \{\epsilon\} \cup Y_A, \quad Y_A \subseteq \{r\}, \quad Z_A \subseteq \{r\} X_A.$$

For any $n \in \mathbb{N}$, the mapping $\{X_A \mapsto \{\epsilon, r, \ldots, r^n\}, Y_A \mapsto \{r, \ldots, r^n\}, Z_A \mapsto \{r^{n+1}\}\}$ is a finite, local, admissible solution of these inclusions, which corresponds to the local $\mathcal{EL}^{-\top}$-unifier γ_n of Γ (see Example 0.2).

This illustrates that there may be infinitely many such solutions for a given $\mathcal{I} \in \mathfrak{F}_\Gamma$. However, there always is one of size at most exponential in the size of Γ if there is one at all. To show this, we consider the remaining part of the $\mathcal{EL}^{-\top}$-unification algorithm. There we use the computational model of *alternating finite automata with ϵ-transitions (ϵ-AFA)*, which are a special case of two-way alternating finite automata. In order to decide the existence of a finite, admissible solution of $\mathcal{I}$, for each variable X_A an ϵ-AFA $\mathcal{A}(X, A)$ is constructed that has the following property.

Lemma 0.9. *The language accepted by $\mathcal{A}(X, A)$ is non-empty iff there is a finite solution θ of $\mathcal{I}$ such that $\theta(X_A) \neq \emptyset$.*

The emptiness test for such automata is a PSpace-complete task [8]. Furthermore, if the language accepted by $\mathcal{A}(X, A)$ is non-empty, then one can construct a run of this automaton of size at most exponential in the size of Γ. This run can then be translated into a finite solution of $\mathcal{I}$ with the property that $\theta(X_A) \neq \emptyset$. Using a weak condition on the structure of runs of $\mathcal{A}(X, A)$, we can even construct a finite, local solution of $\mathcal{I}$ with this property.

Lemma 0.10. *If the language accepted by $\mathcal{A}(X, A)$ is non-empty, then one can construct a finite, local solution θ of $\mathcal{I}$ with $\theta(X_A) \neq \emptyset$.*

The set of all solutions of $\mathcal{I}$ is closed under point-wise union, i.e., if θ_1 and θ_2 are solutions of $\mathcal{I}$, then $\theta_1 \cup \theta_2$ is also one, where $(\theta_1 \cup \theta_2)(X) := \theta_1(X) \cup \theta_2(X)$ for each indeterminate X of $\mathcal{I}$. Thus, $\mathcal{I}$ has a finite, admissible solution iff for

each $X \in N_v$ there is a constant $A \in N_c$ such that $\mathcal{A}(X, A)$ accepts a non-empty language. Since the union of local solutions is again local, it is possible to construct a finite, local, admissible solution of $\mathcal{I}$ in exponential time in the size of Γ if there exists a finite, admissible solution of $\mathcal{I}$.

To summarize, assume that Γ is unifiable. Then we enumerate all elements $\mathcal{I}$ of $\mathfrak{F}_\Gamma$ and check whether they have a finite, admissible solution. By Lemma 0.5, at least one of them must have such a solution. Lemmata 0.9 and 0.10 show that one can construct a finite, local, admissible solution θ of $\mathcal{I}$ that is of size at most exponential in the size of Γ. Using Lemma 0.7, we can then construct a local $\mathcal{EL}^{-\top}$-unifier of Γ that is of size at most exponential in the size of Γ.

It is shown in [3] that this exponential bound is optimal, i.e., there is a sequence Γ_n of solvable $\mathcal{EL}^{-\top}$-unification problems of size polynomial in n such that any local $\mathcal{EL}^{-\top}$-unifier of Γ_n has size at least exponential in n.

Bibliography

1. Franz Baader. Terminological cycles in a description logic with existential restrictions. In *Proc. of the 18th Int. Joint Conf. on Artificial Intelligence (IJCAI 2003)*, pages 325–330, 2003. Morgan Kaufmann, Los Altos.
2. Franz Baader, Nguyen Thanh Binh, Stefan Borgwardt, and Barbara Morawska. Unification in the description logic $\mathcal{EL}$ without the top concept. In *Proc. of the 23rd Int. Conf. on Automated Deduction (CADE 2011)*, Springer LNCS, 2011. To appear.
3. Franz Baader, Nguyen Thanh Binh, Stefan Borgwardt, and Barbara Morawska. Unification in the description logic $\mathcal{EL}$ without the top concept. LTCS-Report 11-01, TU Dresden, Dresden, Germany, 2011. See http://lat.inf.tu-dresden.de/research/reports.html.
4. Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the $\mathcal{EL}$ envelope. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*, pages 364–369, 2005. Morgan Kaufmann, Los Altos.
5. Franz Baader and Barbara Morawska. Unification in the description logic $\mathcal{EL}$. In *Proc. of the 20th Int. Conf. on Rewriting Techniques and Applications (RTA 2009)*, Springer LNCS 5595, pages 350–364, 2009.
6. Franz Baader and Barbara Morawska. SAT encoding of unification in $\mathcal{EL}$. In *Proc. of the 17th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-17)*, Springer LNCS 6397, pages 97–111, 2010.
7. Franz Baader and Paliath Narendran. Unification of concept terms in description logics. *J. of Symbolic Computation*, 31(3):277–305, 2001.
8. Tao Jiang and Bala Ravikumar. A note on the space complexity of some decision problems for finite automata. *Information Processing Letters*, 40:25–31, 1991.

Unification of anti-terms

Jan Otop*

Institute of Computer Science, University of Wrocław
ul. Joliot-Curie 15, PL-50-383, Wrocław, Poland.
jotop@cs.uni.wroc.pl

1 Introduction

Unification can be defined in terms of validity in equational logic. An instance $\Gamma = \{t_1 =^? s_1, \ldots, t_k =^? s_k\}$ of the unification problem corresponds to the problem: does $\vdash \phi$ hold, where $\phi \equiv \exists \bar{X}[t_1(\bar{x}) = s_1(\bar{x}) \wedge \ldots \wedge t_k(\bar{x}) = s_k(\bar{x})]$, i.e. is ϕ valid? Similarly, an instance Γ of the $\mathcal{E}$-unification problem corresponds to the problem does $\mathcal{E} \vdash \phi$ hold, i.e. is ϕ valid in every model of $\mathcal{E}$.

The unification problems can be combined with negative constraints. The (classical) disunification problem results from allowing negated equations. An instance of the (classical) disunification problem is a set of equations, of the form $t =^? s$, and disequations, of the form $t \neq^? s$, $\{t_1 =^? s_1, \ldots, t_k =^? s_k, t'_1 \neq^? s'_1, \ldots, t'_n \neq^? s'_n\}$ and the question is to decide whether $\vdash \exists \bar{x}[t_1(\bar{x}) = s_1(\bar{x}) \wedge \ldots \wedge t_k(\bar{x}) = s_k(\bar{x}) \wedge t'_1(\bar{x}) \neq s'_1(\bar{x}) \wedge \ldots \wedge t'_m(\bar{x}) \neq s'_m(\bar{x})]$ holds. We refer to the aforementioned problem as the classical disunification problem, because in [6] the disunification problem is defined as deciding validity of any equational first order formula.

The (classical) disunification problem has been studied in presence of an equational theory. Basically, $\vdash$ has been replaced by $\mathcal{E} \vdash$. It has been studied in [2] (the AC-case), [3] (the ACI-case) and related papers (see [3] for references). Whereas the classical disunification problem modulo ACU is decidable, the problem of deciding $ACU \vdash \phi$ for any equational formula ϕ is undecidable [11]. The problem remains undecidable even for the $\exists^*\forall^*$ fragment [7].

A subfragment of the $\exists^*\forall^*$ fragment, called *the complement problem* in $\mathcal{E}$, has been intensively studied in the case $\mathcal{E} = AC$ ([8], [9], [10]). Let $\mathcal{E}$ be an equational theory. An instance of the complement problem in $\mathcal{E}$ is a term t and a sequence of terms $s_1, \ldots, s_k$, such that $\bar{x}$ are variables occurring in t, $\bar{y}$ are variables occurring in $s_1, \ldots, s_k$ and $\bar{x} \cap \bar{y} = \emptyset$. The question is to decide whether $\mathcal{E} \vdash \exists \bar{x} \forall \bar{y}[t \neq s_1 \wedge \ldots \wedge t \neq s_k]$ holds. The difference between the complement problem and $\mathcal{E}$-unification is essential, there are universally quantified variables in the complement problem. Therefore, despite having well understood AC-unification algorithms, the complement problem in AC is still an open problem.

Recently, Kirchner et. al. ([4], [5]) introduced another approach to combine negative constraints with unification. Instead of modifying the unification problem itself, the set of possible instances has been extended by allowing *anti-terms*.

* The author is supported by MNiSW under grant N206 379837 2009-2011

An anti-term is a term over $\Sigma \cup \{\urcorner\}$. However, the symbol $\urcorner$ is rather a logical symbol than a function symbol. The interpretation of an anti-term $\urcorner(t)$ is the set of all ground terms that are not instances of t. Moreover, variables that occur only below the $\urcorner$ operator are considered as universally quantified.

The papers [4], [5] present matching with anti-patterns, that is solving the equations where one side is an anti-term and the other is a ground term. It has been shown that matching with anti-patterns is unitary and it is decidable in polynomial time. Alos, they considered an A-matching of anti-patters, that is matching modulo associativity. It has been shown that the A-matching of anti-patters problem is decidable in NEXPTIME. However, the matching with anti-patterns is very restricted. Thus, the question emerges: what happens when both sides can be anti-terms? The unification of anti-terms problem is: given a set of equations where both sides are anti-terms, does this set have a solution?

Having some form of negation in unification problems, it is possible to express more properties. Moreover, even properties that can be described by pure unification equations can be described in a more concise way using negation. The approach to combine negative constraints and unification by adjoining "negation" to the signature is novel and interesting.

In this paper we define the unification of anti-patterns problem and we investigate its properties. We will show that unification of anti-terms generalizes disunification and the complement problem. We relate unification to aforementioned approaches and to fragments of the equational first order logic.

1.1 Preliminaries

The set $\mathcal{T}(\Sigma, \mathcal{X})$ denotes the set of all terms over the signature Σ with variables from $\mathcal{X}$ and $\mathcal{T}(\Sigma) = \mathcal{T}(\Sigma, \emptyset)$ denotes the set of all ground terms over Σ. The set of anti-terms $\mathcal{AT}(\Sigma, \mathcal{X})$ is the smallest set containing the set of terms $\mathcal{T}(\Sigma, \mathcal{X})$, $\urcorner(t)$ and $f(t_1, \ldots, t_k)$ where $f \in \Sigma$ and $t_1, \ldots, t_k \in \mathcal{AT}(\Sigma, \mathcal{X})$.

Let $\mathcal{E}$ be an equational theory. For a term t we define $[\![t]\!]_{\mathcal{E}} = \{s : s =_{\mathcal{E}} \Theta(t)$ and Θ is a grounding substitution $\}$. For an anti-term t we define $[\![t[\urcorner s]_\omega]\!]_{\mathcal{E}} = [\![t[x]_\omega]\!]_{\mathcal{E}} \setminus [\![t[s]_\omega]\!]_{\mathcal{E}}$ where x is a fresh variable and there is no $\urcorner$ in t on any position preceding ω. If $\mathcal{E} = \emptyset$ we will write $[\![t]\!]$ instead of $[\![t]\!]_{\mathcal{E}}$.

In order to avoid trivial cases, we assume that every signature Σ contains at least one constant and one function symbol of arity at least one.

A set of free variables $\mathcal{FV}ar(t)$ of the anti-term t is defined as:

- $\mathcal{FV}ar(X) = \{X\}$ where X is a variable
- $\mathcal{FV}ar(\urcorner q) = \emptyset$
- $\mathcal{FV}ar(f(t_1, \ldots, t_k)) = \bigcup_{i \in \{1,\ldots,k\}} \mathcal{FV}ar(t_i)$

The definition of the free variables is straightforwardly generalized to equations and sets of equations as follows: $\mathcal{FV}ar(t =^? s) = \mathcal{FV}ar(t) \cup \mathcal{FV}ar(s)$ and $\mathcal{FV}ar(\Gamma) = \bigcup_{t =^? s \in \Gamma} \mathcal{FV}ar(t =^? s)$. We assume that substitutions are applied only to the free variables of anti-terms.

1.2 The definition of the unification of anti-terms problem

This paper follows work on matching with anti-patterns (see [4], [5]), but according to our knowledge the notion of unification of anti-terms has not been previously defined.

Definition 1.1. *Let $\mathcal{E}$ be an equational theory and let $\Sigma_\Gamma, \Sigma_\Theta$ be signatures, where $\Sigma_\Gamma \subseteq \Sigma_\Theta$. Let $\Gamma = \{t_1 =^? s_1, \ldots, t_k =^? s_k\}$, where $t_1, \ldots, t_k, s_1, \ldots, s_k$ are anti-terms over Σ_Γ. A substitution Θ over Σ_Θ is a* solution *of an instance Γ of the unification of anti-terms problem modulo $\mathcal{E}$ if and only if $dom(\Theta) = \mathcal{FV}ar(\Gamma)$ and for every substitution Ξ with $dom(\Xi) = \mathcal{FV}ar(\Theta(\Gamma))$ it holds that $[\![\Xi(\Theta(t))]\!]_\mathcal{E} \cap [\![\Xi(\Theta(s))]\!]_\mathcal{E} \neq \emptyset$.*

Note that a solution Θ is defined only on the free variables of Γ and Ξ is defined on the free variables of $\Theta(\Gamma) = \{\Theta(t_i) =^? \Theta(s_i) : t_i =^? s_i \in \Gamma\}$.

Unification of anti-terms generalizes the matching with anti-patterns. Also, it can be shown that if t, s are terms, then for every substitution Ξ (with $dom(\Xi) \subseteq \mathcal{FV}ar(\Theta(\Gamma))$) it holds that $[\![\Xi(\Theta(t))]\!]_\mathcal{E} \cap [\![\sigma(\Theta(s))]\!]_\mathcal{E} \neq \emptyset$ if and only if $\Theta(t) =_\mathcal{E} \Theta(s)$.

In case of unification, we can restrict ourselves to consider only substitutions over the signature of unification equations without affecting solvability. In case of unification of anti-terms it is not true anymore. Let us consider the set $\{X =^? \urcorner f(Y), X =^? \urcorner c\}$, it has no solution over the signature $\{f, c\}$, but it has a solution $X = h(Z)$ where $h \neq f$. This shows us that the signature Σ_Θ is an essential parameter of unification of anti-terms.

2 Results

2.1 The syntactic case

We will show that the unification of anti-terms problem is NP-complete in the syntactic case, i.e. when $\mathcal{E} = \emptyset$.

Definition 2.1. *A set of equations Γ is in* negation-solved form *if and only if for every equation $t =^? s \in \Gamma$ either t, s are terms (do not contain $\urcorner$) or $t =^? s$ is of the form $\urcorner x = y$ where $x, y \in \mathcal{FV}ar(\Gamma)$.*

We say that Γ_2 is a *conservative extension* of Γ_1 if and only if $\mathcal{FV}ars(\Gamma_1) \subseteq \mathcal{FV}ars(\Gamma_2)$, every solution of Γ_2 is a solution of Γ_1 and every solution of Γ_1 can be extended to a solution of Γ_2. We define a set of inference rules $\mathcal{I}$, such that the following lemma holds:

Lemma 2.2. *Let Γ be a set of equations. Using the rules $\mathcal{I}$, Γ can be non-deterministically transformed in linearly many steps into saturated set Γ^s of polynomial size, such that Γ^s is a conservative extension of Γ and Γ^s is in negation-solved form.*

A set of equations in negation-solved form is an instance of the disunification problem. However, there is a simple condition under which Γ in negation-solved form is solvable.

Lemma 2.3. *Let Γ be a set of equations in negation-solved form. Then, $\Gamma = \Gamma_E \cup \Gamma_C$, where Γ_E contains only term equations and $\Gamma_C = \{\urcorner x_1 =^? y_1, \ldots, \urcorner x_k =^? y_k\}$, where $x_i, y_i \in \mathcal{F}var(\Gamma)$. The set Γ has a solution iff Γ_E is solvable and for the mgu Θ_E of Γ_E and for $i = 1, \ldots, k$ we have $\Theta_E(x_i) \neq \Theta_E(y_i)$.*

To prove Lemma 2.3, it is sufficient to have a single constant and a single function symbol of arity greater than 0. E.g. if $\Gamma = \{z_1 = f(f(x,y),x), z_2 = f(f(x,y),x), z_1 = \urcorner(z_2)$, then $f(f(x,y),x) \neq f(f(y,x),y)$ and $x = f(c,c)$, $y = f(f(f(c,c),c),c)$ solves Γ. The general intuition is that having a sequence of terms $u_1, \ldots, u_p$ such that pairwise, a ration of sizes $|u_i|/|u_j|$ (for $|u_i| > |u_j|$) is big enough, then for two terms t, s of bounded size, we have that $t[x_1 \leftarrow u_1, \ldots, x_k \leftarrow u_k] = s[x_1 \leftarrow u_1, \ldots, x_k \leftarrow u_k]$ if and only if $t = s$.

It can be verified in polynomial time, whether the mgu Θ of Γ satisfies $\Theta(X) = \Theta(Y)$. Hence, we have the following theorem:

Theorem 2.4. *Unification of anti-terms is in NP.*

On the other hand, by reduction of 3-SAT to the unification of anti-terms problem, we have:

Theorem 2.5. *Unification of anti-terms is NP-complete.*

2.2 The expression power of unification with anti-terms

We will show a translation of unification of anti-terms to equational logic and a fragment of equational logic that contains the unification with anti-terms. It is possible to deduce necessary conditions to the opposite translation, i.e. to give a translation of every formula from some fragment of equational logic to the unification of anti-terms problem.

For an anti-term t we define $reg(t)$ as the term resulting from removing all $\urcorner$ symbols from t. E.g. $reg(f(\urcorner(g(x)))) = f(g(x))$. The following fact is a direct consequence of the definition of $[\![t]\!]_{\mathcal{E}}$ for an anti-term t:

Fact 2.6 *Let t be an anti-term. The set $[\![t]\!]_{\mathcal{E}}$ is a Boolean combination of $[\![t_1]\!]_{\mathcal{E}}, \ldots, [\![t_s]\!]_{\mathcal{E}}$, where $t_1, \ldots, t_s$ are generalizations of $reg(t)$ (and thus, terms). Such a Boolean combination can be effectively computed.*

By Definition 1.1, an instance $\Gamma = \{t_1 =^? s_1, \ldots, t_k =^? s_k\}$ of the unification of anti-terms problem can be translated to the formula $\phi = \exists \bar{z} \exists \bar{x} [z_1 \in [\![t_1(\bar{x})]\!]_{\mathcal{E}} \wedge z_1 \in [\![s_1(\bar{x})]\!]_{\mathcal{E}} \wedge \ldots \wedge z_k \in [\![t_k(\bar{x})]\!]_{\mathcal{E}} \wedge z_k \in [\![s_k(\bar{x})]\!]_{\mathcal{E}}]$ where $\bar{X} \subseteq \mathcal{FV}ars(\Gamma)$ corresponds to a solving substitution. Due to Fact 2.6, the formula ϕ is equivalent to an existentially quantified Boolean combination of the following atoms: $z_i \in [\![t_i^j(\bar{x})]\!]_{\mathcal{E}}$ and $z_i \in [\![s_i^j(\bar{x})]\!]_{\mathcal{E}}$ where t_i^j, s_i^j are generalizations of $reg(t_i)$ and $reg(s_i)$

resp. As every Boolean combination it can be written in conjunctive normal form.

Note that terms $t_1, \ldots, t_s$ from Fact 2.6 may contain variables that do not belong to $\mathcal{FV}ars(\Gamma)$. Then, for $\bar{x} \subseteq \mathcal{FV}ars(\Gamma)$ and $\bar{y} \cap \mathcal{FV}ars(\Gamma) = \emptyset$ we have that $z_i \in [\![s(\bar{x})]\!]_{\mathcal{E}}$ and $z_i \notin [\![s(\bar{x})]\!]_{\mathcal{E}}$ are equivalent to $\exists \bar{y}[z = s(\bar{x}, \bar{y})]$ and $\forall \bar{y}[z \neq s(\bar{x}, \bar{y})]$ resp. The variables $\bar{y}$ can be renamed, such that in every $z_i \in [\![s(\bar{x})]\!]_{\mathcal{E}}$ variables $\bar{y}$ are different. Then, we have:

Proposition 2.7. *Let $\Gamma = \{t_1 =^? s_1, \ldots, t_k =^? s_k\}$ where t_i, s_i are anti-terms. A formula ψ_Γ of equational logic can be computed such that Γ has a solution if and only if $\vdash \psi_\Gamma$.*

The formula Γ is a disjunction of formulae of the following form:

$$\exists \bar{z} \exists \bar{x} \forall \bar{y} z_1 = u_1 \wedge \ldots \wedge z_s = u_s \wedge z_1 \neq v_1 \wedge \ldots \wedge z_s \neq v_s$$

where v_i, u_i are terms.

It has been shown in [7], that the problem of validity in the $\exists^*\forall^*$-fragment of equational logic modulo ACI is undecidable. Roughly, a set X of pairs of words has been described, which is closed under $(v, w) \wedge v \neq w \in X \Rightarrow (v_1 v, w_1 w) \in X \vee \ldots \vee (v_k v, w_k w)$. We see that the set X can be finite if and only if the instance of Post Correspondence Problem $\{(v_1, w_1), \ldots, (v_k, w_k)\}$ has a solution. The formulae defining the closure are $\exists^*\forall^*$ formulae, but they are not of the form given in Proposition 2.7. Intuitively, implication is not expressible by unification of anti-terms, so there are still chances to have an algorithm deciding the unification of anti-terms modulo ACI.

2.3 Unification of anti-terms modulo an equational theory

Clearly, disunification is a special case of unification of anti-terms. Also, the complement problem in $\mathcal{E}$, defined in the introduction, is a special case of unification of anti-terms modulo $\mathcal{E}$. Notice that the formula $\exists \bar{x} \forall \bar{y} t(\bar{x}) \neq_{\mathcal{E}} s_1(\bar{y}) \wedge \ldots \wedge t(\bar{x}) \neq_{\mathcal{E}} s_k(\bar{y})$ is valid if and only if an instance of the complement problem $\{t =^?_{\mathcal{E}} \urcorner s_1, \ldots, t =^?_{\mathcal{E}} \urcorner s_k\}$ is solvable. Clearly, the set $\mathcal{F}var(t)$ is equal to $\{x_1, \ldots, x_n\}$ and for each i we have $\mathcal{F}var(s_i) = \emptyset$, hence the variables $\bar{x}$ are existentially quantified and the variables $\bar{y}$ are universally quantified. However, for special class of equational theories, there is a reduction in the opposite direction.

An equational theory $\mathcal{E}$ is *finitary* if and only if for every set of equations there exists a finite complete set of most general unifiers. Exhaustive exposition of unification types can be found in [1].

Proposition 2.8. *Let $\mathcal{E}$ be a finitary theory having a recursive function which, given a set of unification equations Γ, enumerates a complete set of most general unifiers of Γ. If the signature Σ contains a free binary function symbol, then the unification of anti-terms problem modulo $\mathcal{E}$ reduces to the complement problem in $\mathcal{E}$.*

Due to Proposition 2.8 we have:

Corollary 2.9. *Unification of anti-terms modulo AC reduces to the complement problem in AC.*

The complement problem in AC had drawn much attention. Decidability of the complement problem in AC is still an open problem, but some special cases have been solved [8], [9], [10].

It has been shown in [4], [5] that matching with anti-patterns modulo AU is decidable in NEXPTIME. In case of unification of anti-terms, we have:

Proposition 2.10. *The complement problem in AU with a single unary function symbol and a single constant is undecidable.*

Proposition 2.10 and the above discussion implies that unification of anti-terms modulo AU is undecidable.

Bibliography

1. Baader, F. and Snyder, W. (2001): Unification theory, *A Chapter in Handbook of Automated Reasoning, Robinson A. and Voronkov A. eds, Elsevier/MIT Press, 2001*
2. Bürckert, H.J.(1988): Solving disequations in equational theories *9th CADE, LNCS, 1988, Vol. 310/1988, p. 517-526*
3. Dovier, A., Piazza, C., Pontelli, E.(2004): Disunification in ACI1 Theories *J. Constraints, Vol. 9 Issue 1, Jan. 2004*
4. Kirchner, C., Kopetz, R. and Moreau, P.-E.(2008): Anti-pattern Matching Modulo *LATA 2008, LNCS Vol. 5196/2008, p. 275-286*
5. Cirstea, H., Kirchner, C., Kopetz, R. and Moreau, P.-E.(2010): Reasoning with anti-terms in rule based languages *J. Symb. Comp. Vol. 45 Iss. 5*
6. Comon H., Disunification: a survey(1991): *Comp. Logic. Essays in honor of Alan Robinson J.-L. Lassez and G. Plotkin, Eds. The MIT press, Cambridge (MA, USA), Chap. 9, p. 322 - 359*
7. Marcinkowski, J.(2002): The $\exists^*\forall^*$ part of the theory of ground term algebra modulo an AC symbol is undecidable.d *Information and Computation 178*
8. Fernandez, M.(1996): AC Complement Problems: Satisfiability and Negation Elimination. *Journal of Symbolic Computation, Volume 22, pages 49-82. Academic PressLimited*
9. Lugiez, D., Moysset J. L.(1993): Complement Problems and Tree Automata in AC-like Theories. *STACS 1993: p. 515-524*
10. Kounalis, E., and Lugiez, D. and Pottier, L. (1991): The complement problem in associative-commutative theories. *MFCS 91, LNCS*
11. Treinen R.(1990): A New Method for Undecidability Proofs of First Order Theories *Proceedings of the Tenth Conference on Foundations of Software Technology and Theoretial Computer Science, LNCS 472, December 1990, edited by K. V. Nori and C. E. Veni Madhavan, p 48-62.*

Joint Constraint Abduction Problems

Łukasz Stafiniak

Institute of Computer Science
University of Wrocław

Abstract. Abduction is generally understood as an inference technique that from D and C derives an A such that $D, A \vdash C$. We formulate the joint (i.e. simultaneous) abduction problem to which type inference and invariant generation for programs using Generalized Algebraic Data Types can be reduced. We also combine abduction algorithms for independent sorts, in the simplest case where interaction occurs only via a sort of finite trees.

1 Introduction

Consider a function `eval` defined by cases over a datatype $\mathrm{Term}(\alpha)$ with constructors `Lit` $: \mathrm{Int} \rightarrow \mathrm{Term}(\mathrm{Int})$, `IsZero` $: \mathrm{Int} \rightarrow \mathrm{Term}(\mathrm{Bool})$, and

$$\texttt{If} : \forall \alpha.\, \mathrm{Term}(\mathrm{Bool}) \rightarrow \mathrm{Term}(\alpha) \rightarrow \mathrm{Term}(\alpha) \rightarrow \mathrm{Term}(\alpha).$$

In a type system with Generalized Algebraic Data Types, the result of reduction of the typing problem for `eval` to constraint solving resembles the following constraint problem (side by side with the corresponding parts of the program):

$$\begin{array}{r}
\exists \tau, \alpha, \beta \forall \gamma . \tau \doteq \alpha \rightarrow \beta \wedge \\
(\alpha \doteq \mathrm{Term}(\mathrm{Int}) \Rightarrow \beta \doteq \mathrm{Int}) \wedge \\
(\alpha \doteq \mathrm{Term}(\mathrm{Bool}) \Rightarrow \beta \doteq \mathrm{Bool}) \wedge \\
(\alpha \doteq \mathrm{Term}(\gamma) \Rightarrow \beta \doteq \gamma)
\end{array}$$

```
eval x = case x of
    Lit y -> y
    IsZero y -> y=0
    If y1 y2 y3 ->
      if eval y1 then eval y2
      else eval y3
```

from which we would like to find the solution $\alpha = \mathrm{Term}(\beta), \tau = \mathrm{Term}(\beta) \rightarrow \beta$, leading to the inferred type `eval` $: \forall \beta.\, \mathrm{Term}(\beta) \rightarrow \beta$. For more information about constraint-based type inference for GADTs, consult [8]. By iteratively solving problems of this kind for the free algebra of terms and other theories, not only could we infer types for programs with GADTs, but also generate invariants of recursive functions. Deciding satisfiability of the constraint is not enough as a solution, we would like to determine the satisfying substitution for existentially quantified variables that was intended by the programmer. For example, a substitution $\alpha = \mathrm{Bool}, \tau = \mathrm{Bool} \rightarrow \beta$ would satisfy the constraint by contradicting each of the premises, but does not explain the behavior of the program.

Abduction is a reasoning technique concerned with the search for explanations. An abduction problem is usually given by a background theory Θ and a formula C, and the solution, or answer, is a formula A such that $\Theta \cup \{A\} \vDash C$ (relevance), $\Theta \nvDash \neg A$ (consistency), and A has some restricted syntactical form, see [7]. We are however interested in *constraint abduction* problems, with constraints expressed over a fixed model $\mathcal{M}$. A constraint abduction problem is then given by formulas D, C and A is its answer when (at least) $\mathcal{M} \vDash (D \wedge A) \Rightarrow C$ (relevance) and $\mathcal{M} \vDash \exists \mathrm{FV}(D, A).D \wedge A$ (consistency), see [5]. We extend the formulation of joint constraint abduction (see [5]) to constraints with quantifiers; the quantifiers cannot be eliminated by Herbrandization (as in the general abduction algorithms for FOL) because the model is fixed. We develop a combination procedure for abduction algorithms when their domains of constraints are combined in a very simple way (yet sufficing for type-inference-driven invariant generation).

We use the language of first order logic with function symbols and equality, under standard interpretation. By the bar $\bar{e}$ we denote a sequence (or a set, depending on context) of elements e, by # we denote disjointness. With a free index i, $\overline{e_i}$ denotes $(e_1, \ldots, e_n)$ for some n associated with the index i; similarly, $\wedge_i \Phi_i$ denotes $\Phi_1 \wedge \ldots \wedge \Phi_n$. In some contexts, for a quantifier prefix $\mathcal{Q}$ we write $\mathcal{Q}$ to denote the set of variables quantified by $\mathcal{Q}$. Let FV be a generic function returning the free variables of any expression. For a set of variables V, let $\exists(V^c).\Phi$ denote $\exists \mathrm{FV}(\Phi) \setminus V.\Phi$, i.e. existential closure of Φ except for variables from V, which are kept free. Let $T(F)$ be the set of ground terms (i.e. finite trees) for signature F, and $T(F, X)$ the set of (possibly multisorted) terms for signature F and variables X. By $\Phi[\bar{\alpha} := \bar{t}]$ we denote a substitution of terms $\bar{t}$ for corresponding variables $\bar{\alpha}$ in the formula Φ (where $\bar{\alpha}$ and $\bar{t}$ are finite sequences of the same length).

2 Formulating the Joint Constraint Abduction Problem

In joint, also called simultaneous, problems, we expect a single answer to solve several problems, here: a conjunction of implications.

A *solved form* is a syntactically specified class of formulas, associated with a class of problems, for which satisfiability is trivial to check. We restrict solved forms to existentially quantified conjunctions of atoms, $\exists \bar{\alpha}.A$. The variables $\bar{\alpha}$ of a solved form $\exists \bar{\alpha}.A$ that is an abduction problem answer, are "free parameters" of the answer, they are required to be "unconstrained".

The constraints that we need to solve form a *joint constraint abduction under a quantifier prefix problem* (*JCAQP* problem for short) of the form $\mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$, where D_i and C_i are conjunctions of atomic formulas, and $\mathcal{Q}$ is an arbitrary quantifier prefix. We assume that $\mathrm{FV}(\wedge_i (D_i \Rightarrow C_i)) \subseteq \mathcal{Q}$.

Definition 2.1. *$\exists \bar{\alpha}.A$ is a $JCAQP_{\mathcal{M}}$ answer (answer to a JCAQP problem $\mathcal{Q}.\wedge_i(D_i \Rightarrow C_i)$ for model $\mathcal{M}$) when A is a conjunction of atoms, $\bar{\alpha} \# \mathrm{FV}\left(\overline{D_i, C_i}\right)$, meeting relevance condition: $\mathcal{M} \vDash \wedge_i(D_i \wedge A \Rightarrow C_i)$, validity condition: $\mathcal{M} \vDash \forall \bar{\alpha} \mathcal{Q}.A$, and consistency condition: $\mathcal{M} \vDash \wedge_i \forall \bar{\alpha} \exists (\bar{\alpha}^c).D_i \wedge A$.*

We can also consider JCAQP problems for a logic, checking relevance condition: $\vDash \wedge_i (D_i \wedge A \Rightarrow C_i)$ *and validity condition:* $\vDash \forall\bar{\alpha}\mathcal{Q}.A$*. The consistency conditions: for all* i*,* $D_i \nvDash \neg A$*, are always met.*

The natural setting for constraint abduction problems is with a fixed model. We extend the definition to the case where instead of a model just a logic is given, just to shed light on relations between constraint abduction, general abduction and decision (i.e. validity or satisfiability) problems.

We call a JCAQP$_{\mathcal{M}}$ problem $\mathcal{Q}.D \Rightarrow C$ (i.e. a non-simultaneous problem) a *simple constraint abduction under a quantifier prefix* problem SCAQP$_{\mathcal{M}}$.

Proposition 2.2. *If the JCAQP*$_{\mathcal{M}}$ *problem* $\mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$ *has an answer, then* $\mathcal{M} \vDash \mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$*.*

We say that JCAQP$_{\mathcal{M}}$ answer $\exists\bar{\alpha}.A$ is *more general* than $\exists\bar{\beta}.B$, when there exist terms $\bar{t}$ such that $\mathcal{M} \vDash B \Rightarrow A[\bar{\alpha} := \bar{t}]$.

An algorithm Abd $\left(\mathcal{Q}, \overline{D_i, C_i}\right)$ is a *complete abduction algorithm* for JCAQP$_{\mathcal{M}}$ if it generates a sequence of quantifed conjunctions of atoms $\overline{\exists\bar{\alpha}_j.A_j}$, possibly infinite, that are answers to the joint abduction under a quantifier prefix problem $\mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$, and if there is a JCAQP$_{\mathcal{M}}$ answer $\exists\bar{\alpha}.A$, there is a j and some $\bar{t}$ such that $\mathcal{M} \vDash A \Rightarrow A_j[\bar{\alpha}_j := \bar{t}]$ (with variables renamed so that $\bar{\alpha} \# \mathrm{FV}(A_j)$). If the sequence is empty, we write $\mathrm{Abd}_s \left(\mathcal{Q}, \overline{D_i, C_i}\right) = \bot$.

Example 2.3. For a free term algebra $T(F)$ over signature F containing binary functors f, g and constants a, b, and JCAQP$_{T(F)}$ problem $\exists x, y, z.(y \doteq f(a, x) \Rightarrow z \doteq a) \wedge (y \doteq f(b, x) \Rightarrow z \doteq b)$, the (most general) answer is $\exists\alpha.y \doteq f(z, \alpha)$. Indeed, $\forall\alpha\exists x, y, z.y \doteq f(z, \alpha) \wedge y \doteq f(a, x)$ holds with $x = \alpha, y = f(a, \alpha), z = a$ and $\forall\alpha\exists x, y, z.y \doteq f(z, \alpha) \wedge y \doteq f(b, x)$ holds with $x = \alpha, y = f(b, \alpha), z = b$. For the JCAQP$_{T(F)}$ problem $\exists x, y, z.(y \doteq f(a, x) \Rightarrow z \doteq a)$, the maximal set of answers is $\{\exists\alpha.y \doteq f(z, \alpha), z \doteq a\}$.

3 Combination of Domains

We combine non-interacting theories into a simple form of multisorted logic. In the context of type inference, these sorts are independent means of specifying properties of data structures; they are combined using data-type constructors that are free, i.e. injective.

All sorts share the equality relation, which is the default means of dependence between theories. We denote the set of sorts by sorts $= \{s_{\mathrm{ty}}\} \dot{\cup}$ usorts, where s_{ty} is the distinguished sort of "types proper", finite trees. Let $\mathcal{L}_s$ be the languages of the independent sorts $s \in$ usorts. We call the combined model $\mathcal{M}$. Due to space constraints, we omit explicit construction of $\mathcal{M}$ (which is provided in the accompanying technical report [9]). Below, when referring to JCAQP$_{\mathcal{M}}$, we omit the index $\mathcal{M}$ when it is clear from context.

Consider a conjunction of atoms C interpreted in any free term algebra T. If it is satisfiable, let $\boldsymbol{U}(C)$ be a conjunction of equations whose left-hand-sides are

variables not occuring in any of the right-hand-sides, such that $T \vDash C \Leftrightarrow \boldsymbol{U}(C)$, otherwise let $\boldsymbol{U}(C) = \bot$. Let $\boldsymbol{U}(\mathcal{Q}.C)$ in case $T \nvDash \mathcal{Q}.C$ be $\bot$, and otherwise be as before but with equations directed so that variables later in the prefix are on the left. $\boldsymbol{U}(\mathcal{Q}.C)$ can be computed by unification with linear constant restrictions, see [1].

Define an *alien subterm* (cf. [1]) of a term τ of sort s_{ty} to be a maximally large subterm t of τ of sort $s \neq s_{\text{ty}}$.

3.1 Abduction Algorithm for The Combination of Domains

We provide a plug-in architecture where to add a new sort to the logic it is enough to give an algorithm solving the JCAQP problem.

Let $\mathcal{L}_{\text{ty}} = T(F, \cup_{s \in \text{sorts}} X_s)$ be a language interpreted in a multisorted free term algebra $T = T(F \cup_{s \in \text{usorts}} D_s)$ which, besides the term variables $X_{s_{\text{ty}}}$, has *alien subterm variables* $\cup_{s \in \text{usorts}} X_s$ (but no other subterms of sorts $s \in$ usorts). For conjunctions of equations $A, \overline{A^i}$, $\exists \bar{\alpha}.A, \overline{A^i}$ is a *joint constraint abduction under a quantifier prefix problem with alien subterms* (JCAQPAS) answer to $\mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$ when $\bar{\alpha} \# \text{FV}(\wedge_i (D_i \Rightarrow C_i))$, $\bar{\alpha} \subset X_{\text{ty}}$, $T \vDash \forall \bar{\alpha} \mathcal{Q}.A$ and $T \vDash \wedge_i \forall \bar{\alpha} \exists (\bar{\alpha}^c).D_i \wedge A$, A is in solved form $A = \boldsymbol{U}(A)$ and only substitutes for term variables $X_{s_{\text{ty}}}$ (i.e. the left-hand-sides of A are $X_{s_{\text{ty}}}$ variables), A^i are equations over alien subterm variables only (i.e. equations with both sides in $\cup_{s \in \text{usorts}} X_s$), and $T \vDash \wedge_i (D_i \wedge A^i \wedge A \Rightarrow C_i)$.

An algorithm $\text{Abd}_T\left(\mathcal{Q}, \overline{D_i, C_i}\right)$ is a *complete abduction algorithm for JCAQPAS* if it generates a sequence, possibly infinite, of answers $\overline{\exists \bar{\alpha}_j.A_j, \overline{A^i_j}}$ (or $\bot$, identified with an empty sequence) that are answers to the JCAQPAS $\mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$, and if there is a JCAQPAS answer $\exists \bar{\alpha}.A, \overline{A^i}$, $\bar{\alpha} \# \text{FV}(A_j)$, there is a j and some $\bar{t}$ such that $T \vDash A \Rightarrow A_j[\bar{\alpha}_j := \bar{t}]$ and $T \vDash A^i \Rightarrow A^i_j$ for all i.

Let $\mathcal{Q}$ be a quantifier prefix and D_i, C_i be atomic conjunctions in $\mathcal{L}$ that form a joint abduction problem $\mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$. Let Abd_s be complete JCAQP algorithms for $s \in$ usorts and Abd_T be a complete JCAQPAS algorithm for s_{ty}. We start the multisorted abduction procedure $\text{Abd}\left(\mathcal{Q}, \overline{D_i, C_i}\right)$ by performing Abd_T on the s_{ty} part of constraints with alien subterms replaced by variables. For each JCAQPAS solution $\exists \bar{\alpha}_j.A_j, \overline{A^i_j}$, we replace the s_{ty} part of the ith premise by the "residual" formula A^i_j. We split the resulting JCAQP problem into single-sort problems for each sort $s \in$ usorts, and we solve them using Abd_s algorithms. Finally, we build answers as conjunctions of answers for each sort. We present the tedious but straightforward definition of $\text{Abd}\left(\mathcal{Q}, \overline{D_i, C_i}\right)$ and proof of theorem 3.1 in the accompanying technical report [9].

Theorem 3.1. Abd *definied above is a* complete abduction *algorithm for JCAQP*$_{\mathcal{M}}$*: Let $\mathcal{Q}$ be a quantifier prefix and D_i, C_i be conjunctions of atoms in $\mathcal{L}$ that form a joint abduction problem $\mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$.* $\text{Abd}\left(\mathcal{Q}, \overline{D_i, C_i}\right)$ *results in a possibly infinite sequence $\overline{\exists \overline{\alpha_j}.A_j}$ of answers to the JCAQP$_{\mathcal{M}}$ problem and for any JCAQP$_{\mathcal{M}}$ answer $\exists \bar{\alpha}.A$, there is an $\exists \overline{\alpha_{\text{ans}}}.A_{\text{ans}} \in \text{Abd}\left(\mathcal{Q}, \overline{D_i, C_i}\right)$ and some $\bar{t}$ such that $\mathcal{M} \vDash A \Rightarrow A_{\text{ans}}[\overline{\alpha_{\text{ans}}} := \bar{t}]$.*

4 Abduction for Terms

The JCAQP problem for first order logic with function symbols and equality is undecidable, because it is equivalent, by Herbrandization, to simultaneous rigid E-unification (see [3]): the substitution that is a solution to simultaneous rigid E-unification when expressed as a conjunction of equations has the same properties as a JCAQP answer (therefore the existence of JCAQP answers coincides with intuitionistic satisfiability).

The decision problem $T(F) \vDash \mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$ is decidable, see [2]. Actually, [2] provides a disjunction as a solution, each disjunct meeting the relevance and validity conditions of the $\mathrm{JCAQP}_{T(F)}$ problem. It is often the case though that each disjunct does not meet the consistency condition, despite the $\mathrm{JCAQP}_{T(F)}$ problem considered having answers.

A complementary approach is to find the "fully maximal answers" introduced in [5], using non-simultaneous abduction algorithm from [6]: abduction answer $\exists\bar{\alpha}.A$ to $D \Rightarrow C$ is *fully maximal* when $T(F) \vDash (\exists\bar{\alpha}.D \wedge A) \Leftrightarrow D \wedge C$. [5] refers to [4] as establishing that there are finitely many fully maximal answers. [6] gives an algorithm finding fully maximal answers for simple (i.e. non-simultaneous) constraint abduction problems. But joint abduction answers can be built from simple abduction answers (see [5]): Consider a $\mathrm{JCAQP}_{T(F)}$ problem $\mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$. Let Abd_S be a complete abduction algorithm for $\mathrm{SCAQP}_{T(F)}$, and

$$\mathcal{A}_0 = \left\{ \mathcal{J}(\overline{\exists\overline{\alpha_i}.A^i}) : \exists\overline{\alpha_i}.A^i \in \mathrm{Abd}_S\left(\mathcal{Q}, D_i, C_i\right) \wedge \boldsymbol{U}(\mathcal{Q}\exists(\cup_i\bar{\alpha}_i). \wedge_i A^i) \neq \bot \right\},$$

where $\mathcal{J}(\overline{\exists\overline{\alpha_i}.A^i}) = \exists\left((\cup_i\bar{\alpha}_i) \cap \mathrm{FV}(A)\right).A$ and A is built from $\boldsymbol{U}(\mathcal{Q}\exists(\cup_i\bar{\alpha}_i). \wedge_i A^i)$ by removing equations whose left-hand-sides belong to $\cup_i\bar{\alpha}_i$. Let $\mathcal{A}$ be $\mathcal{A}_0$ with elements that do not meet the consistency condition for $\mathrm{JCAQP}_{T(F)}$ problem $\mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$ removed.

Theorem 4.1. *Setting* $\mathrm{Abd}\left(\mathcal{Q}, \overline{D_i, C_i}\right) := \mathcal{A}$ *gives a complete abduction algorithm for* $\mathit{JCAQP}_{T(F)}$*: elements of* $\mathcal{A}$ *meet the relevance, validity and consistency conditions, and if* $\exists\bar{\alpha}.A$ *is a* $\mathit{JCAQP}_{T(F)}$ *answer to* $\mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$*, then there is* $\exists\bar{\alpha}_{\mathrm{ans}}.A^{\mathrm{ans}} \in \mathcal{A}$ *and some* $\bar{t}$ *such that* $T(F) \vDash A \Rightarrow A_{\mathrm{ans}}[\overline{\alpha_{\mathrm{ans}}} := \bar{t}]$.

Proof. Relevance and consistency conditions for answers in $\mathcal{A}$ follow directly from the construction, validity follows from the fact that variables not occurring on the left-hand-side of a solved form A can be arbitrarily substituted while preserving validity of $T(F) \vDash \mathcal{Q}.A$.

Since $\exists\bar{\alpha}.A$ is a $\mathrm{JCAQP}_{T(F)}$ answer to $\mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$, it is a $\mathrm{SCAQP}_{T(F)}$ answer to $\mathcal{Q}.D_i \Rightarrow C_i$ for each i. Therefore, for each i there exist $\exists\overline{\alpha_i}.A^i \in \mathrm{Abd}_S\left(\mathcal{Q}, D_i, C_i\right)$ and $\bar{t}_i$ such that $T(F) \vDash A \Rightarrow A^i[\overline{\alpha_i} := \bar{t}_i]$. From $T(F) \vDash \mathcal{Q}.A$ we have $T(F) \vDash \mathcal{Q}. \wedge_i A^i[\overline{\alpha_i} := \bar{t}_i]$, i.e. $T(F) \vDash \mathcal{Q}\exists(\cup_i\bar{\alpha}_i). \wedge_i A^i$. Therefore, $\boldsymbol{U}(\mathcal{Q}\exists(\cup_i\bar{\alpha}_i). \wedge_i A^i) \neq \bot$ and the corresponding $\exists\left((\cup_i\bar{\alpha}_i) \cap \mathrm{FV}(A_{\mathrm{ans}})\right).A_{\mathrm{ans}} \in \mathcal{A}_0$. Let $\bar{\alpha}_{\mathrm{ans}}$ be $(\cup_i\bar{\alpha}_i) \cap \mathrm{FV}(A_{\mathrm{ans}})$. Since $T(F) \vDash \wedge_i\forall\bar{\alpha}\exists(\bar{\alpha}^c).D_i \wedge A$, we have $T(F) \vDash \wedge_i\forall\bar{\alpha}\exists(\bar{\alpha}^c).D_i \wedge_j A^j[\overline{\alpha_j} := \bar{t}_j]$, therefore $T(F) \vDash \wedge_i\forall\bar{\alpha}\exists(\bar{\alpha}^c).D_i \wedge_j A^j$. From it follows that $T(F) \vDash \wedge_i\forall\bar{\alpha}\exists(\bar{\alpha}^c).D_i \wedge A_{\mathrm{ans}}$, and by renaming variables and

dropping quantification over unused variables, $T(F) \vDash \wedge_i \forall \bar{\alpha}_{\mathrm{ans}} \exists (\bar{\alpha}^c_{\mathrm{ans}}).D_i \wedge A_{\mathrm{ans}}$. We have shown that $\exists \bar{\alpha}_{\mathrm{ans}}.A^{\mathrm{ans}} \in \mathcal{A}$.

[5] remarks that the joint abduction problem for $T(F)$ is not known to be decidable. It remains to be seen whether there are practical cases where the intended answer cannot be built from fully maximal answers to the component simple abduction problems.

Bibliography

[1] Franz Baader and Klaus U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, pages 50–65, London, UK, 1992. Springer-Verlag.

[2] Hubert Comon. Disunification: a survey. In *Computational Logic: Essays in Honor of Alan Robinson*, pages 322–359. MIT Press, 1991.

[3] Anatoli Degtyarev and Andrei Voronkov. Simultaneous rigid e-unification is undecidable. Technical report, Computer Science Logic. 9th International Workshop, CSL'95, 1995.

[4] Michael Maher. On parameterized substitutions. Technical report, IBM Research Report RC 16042, 1990.

[5] Michael Maher. Herbrand constraint abduction. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 397–406, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Michael Maher and Ge Huang. On computing constraint abduction answers. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 5330 of *Lecture Notes in Computer Science*, pages 421–435. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-89439-1-30.

[7] Marta Cialdea Mayer and Fiora Pirri. First order abduction via tableau and sequent calculi. *Logic Journal of IGPL*, 1(1):99–117, 1993.

[8] Vincent Simonet and Francois Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems*, 29(1), January 2007.

[9] Łukasz Stafiniak. Joint constraint abduction problems. http://www.ii.uni.wroc.pl/~lukstafi/pubs/abduction.pdf. Manuscript, 2011.

Projective Unifiers in Modal Logics

Wojciech Dzik and Piotr Wojtylak

Institute of Mathematics, Silesian University, Katowice, Poland

Abstract. A *projective unifier* for a modal formula A, in a modal logic L, is a unifier σ for A (i.e. a substitution making A a theorem of L) such that $A \vdash_L x[\sigma] \leftrightarrow x$, for all variables x in A. Each projective unifier is a most general unifier for A. Let L be a normal modal logic containing $S4$. It is shown that every unifiable formula has a projective unifier in L iff L contains $S4.3 = S4$ plus $\Box(\Box A \to \Box B) \vee \Box(\Box B \to \Box A)$. The effective syntactic proof is given. As a corollary, we get that all normal modal logics L containing $S4.3$ are almost structurally complete, that is, all admissible rules having *unifiable* premises, are derivable in L.

Key words: unification, projective unifiers, modal logics $S4$, $S4.3$.

1 Introduction: Unification in Logic

Unification or E-unification is concerned with finding a substitution that makes given terms equal, modulo an equational theory E. Such a substitution is called a unifier for the terms in E. Unifiers can be seen as solutions of a system of equations on terms. A theory can have unitry, finitary, infinitary or nullary unification, depending on the number of general solutions that represent all solutions, see e.g. [1].

In logic, unification is concerned with finding a substitution that makes a given formula A a theorem of a logic L. A *unifier for* a formula A in a logic L is a substitution σ such that $\vdash_L \sigma(A)$. A *formula* A *is unifiable* in L, if such σ exists. If τ, σ are substitutions, then σ is *more general then* τ, $\tau \preccurlyeq \sigma$, if there is a substitution θ such that $\vdash_L \theta(\sigma(x)) \leftrightarrow \tau(x)$. Classical propositional logic CL has unitary unification. It means that every formula A, unifiable (= consistent) in CL, has a mgu, i.e. a substitution σ such that $\vdash_{CL} \sigma(A)$ and that every unifier τ for A is a special case of σ, i.e. $\vdash_{CL} \theta(\sigma(x)) \leftrightarrow \tau(x)$, for some θ. Unification type of a logic can be unitary, finitary, infinitary or nullary depending on the number of maximal unifiers, see e.g. [2], [4].

A *projective unifier* for a unifiable formula A in a logic L is a unifier σ for A such that $A \vdash_L \sigma(x) \leftrightarrow x$ for all $x \in Var(A)$, see [2]; a formula which has a projective unifier is called a *projective formula*; this notion was introduced and applied for modal logics by S. Ghilardi, see [9] [11].

Every projective unifier is an mgu but projective unifiers have many advantages over just mgu's: they are preserved by extensions and they provide a solution to the problem of admissibility of inference rules.

S. Ghilardi showed that unification in logics $K4, GL, S4, Grz$ is finitary, see [9], [10], and unification in $K4.2$ and $S4.2$ is unitary, [11]. It is known that unification in extensions of $S4.2$ is unitary or nullary, see [3]. In a joint paper [6] it is shown that in a modal logic L (containing $S4$) each unifiable formula has a projective unifier iff L contains modal logic $S4.3 = S4$ plus $\Box(\Box A \to \Box B) \vee \Box(\Box B \to \Box A)$ (an analogous theorem for intermediate logics was proved by A. Wroński, see [13], [14]). Unifiers have many applications in logic; they provide uniform method of recognizing admissible rules. In particular, if a formula A has a projective unifier, then the rule $\frac{A}{B}$ is admissible iff it is derivable.

Projective unifiers are also considered in equational theories (or varieties of algebras). In particular, the following modification of S.Burris theorem holds: in every discriminator variety, unifiable terms have a projective unifier, see [5].

2 Modal Logics

We consider the *modal language* $\{\to, \bot, \Box\}$. Let $Var = \{x, x_1, \ldots, y, \ldots, z, z_1, \ldots\}$ be the set of *propositional variables* and Fm be the set of *modal propositional formulas* built up, in the standard way, from propositional variables and constant $\bot$ by means of the operators $\to$ and $\Box$. For each formula A, let $Var(A)$ denote the (finite) set of variables occurring in A. The remaining classical connectives $\wedge, \vee, \sim, \leftrightarrow, \top$ are defined in a standard way, as well as the modal operator $\Diamond A := \sim \Box \sim A$. Hence we get the algebra of the language $\mathcal{F}$ with the universe Fm and the connectives as operations.

By a *substitution* we mean any finite mapping $\varepsilon \colon Var \to Fm$. We usually write: $\varepsilon := x_1/B_1 \cdots x_n/B_n$, if $\{x_1, \ldots, x_n\}$ is the domain of ε and the formulas $B_1, \ldots, B_n$ are its values, i.e. ε is a substitution *for* the variables $x_1, \ldots, x_n$. The result of the substitution on a formula A is denoted by

$$A[\varepsilon] \qquad \text{or} \qquad A[x_1/B_1 \cdots x_n/B_n].$$

We put $x[\varepsilon] = x$, if $x \notin \{x_1, \ldots, x_n\}$, hence each substitution can be extended to a total mapping : $Var \to Fm$. Each substitution determines uniquely an endomorphism of the algebra $\mathcal{F}$ of the language. In our approach we use traditional suffix-notation $A[\varepsilon]$ instead of $\varepsilon(A)$. The composition of substitutions ε and δ, applied to a formula A is denoted by $A[\varepsilon\delta]$; first the substitution ε is applied to a formula A to get $A[\varepsilon]$, and then, after applying δ, we receive $A[\varepsilon\delta] = (A[\varepsilon])[\delta]$.

By a *modal logic* we mean here any consistent and normal extension of $S4$ ($= K4T$), that is, a proper subset of Fm containing all classical tautologies, containing the following $S4$ axioms:

$$\begin{array}{l} K : \Box(A \to B) \to (\Box A \to \Box B) \\ 4 : \ \Box\Box A \to \Box A \\ T : \ \Box A \to A. \end{array}$$

and closed under substitutions and under the rules $MP : \frac{A \to B, A}{B}$ and $RG : \frac{A}{\Box A}$,

It is shown later that weaker systems like $K4$ fail to have projective unifiers and to get the results of this paper one must take $S4$ as a basic modal system.

Given a modal logic L, we consider here its *global entailment relation* $\vdash_L$. Hence, $X \vdash_L A$ means that A can be derived from $X \cup L$ using the rules MP and RG, i.e. Necessitation rule RG is a postulated inference rule for $\vdash_L$. The relation $\vdash_L$ is structural: if $X \vdash_L A$, then $X[\varepsilon] \vdash_L A[\varepsilon]$, for each substitution ε.

The following deduction theorem for a logic $L \supseteq S4$ is well known

$X, B \vdash_L A$ iff $X \vdash_L \Box B \to A$.

For each modal logic L, the relation $=_L$ of L-equivalence, defined as follows $A =_L B$ iff $\vdash_L A \leftrightarrow B$, is a congruence in the algebra of the language. $=_L$ is the largest congruence consistent with $\vdash_L$. The relation $=_L$ is a congruence on $\mathcal{F}$ as $S4$ enjoys the so-called *extensionality*, or *replacement* property:

$B \leftrightarrow C \ \vdash_L \ A[x/B] \leftrightarrow A[x/C]$.

We identify L-equivalent formulas. In particular, we identify L-equivalent substitutions, that is substitutions such that $x[\varepsilon_1] =_L x[\varepsilon_2]$ for each variable x, i.e. $\varepsilon_1 =_L \varepsilon_2$. Note that $\varepsilon_1 =_L \varepsilon_2$ iff $(A[\varepsilon_1] =_L A[\varepsilon_2]$, for each formula A).

The *Lindenbaum-Tarski algebra* of L is a quotient $\mathcal{F}/_{=_L}$ of the algebra of the language $\mathcal{F}$; in our case $\mathcal{F}/_{=_L}$ is a topological Boolean algebra in which $\Box$ gives rise to the interior operation. For each logic L, the constants $\{\bot, \top\}$ form a subalgebra of the Lindenbaum-Tarski algebra, isomorphic to the two-element Boolean algebra $\mathbf{2}$ in which $\Box a = a$, for each a. The *trivial* modal logic, Tr, is determined by $\mathbf{2}$; we have: $S4 \subseteq L \subseteq Tr$, for any consistent logic $L \supseteq S4$.

For a given modal logic L, a substitution ε is called a *unifier* for a formula A in L, or *L-unifier* for A if $\vdash_L A[\varepsilon]$. A formula A is said to be *unifiable* in L if there exists a L-unifier for A. Note: $\Diamond A \wedge \Diamond \sim A$ is not unifiable in any logic L.

Unifiers of the form $v\colon Var(A) \to \{\bot, \top\}$ are called *ground unifiers* for A. They can be identified with valuations in $\mathbf{2}$ which satisfy the formula A. Given a unifier ε for A and any substitution $\delta\colon Var \to \{\bot, \top\}$ we get the ground unifier $\varepsilon\delta$ for the formula A. Hence, the following conditions are equivalent: (i) A is L-unifiable ; (ii) there is a ground unifier for A in L; (iii) A is satisfiable in $\mathbf{2}$.

A substitution ε is *projective* for a formula A, on the ground of an L, if

$$A \vdash_L x[\varepsilon] \leftrightarrow x \qquad \text{for each variable } x.$$

A *projective unifier* for a formula A in L is an L-unifier which is projective for A on the ground of L. A formula which has a projective unifier is called projective.

Our main task is to find, for a given formula A, a substitution ε such that

(i) $\vdash_L A[\varepsilon]$;

(ii) $A \vdash_L x[\varepsilon] \leftrightarrow x$, for each variable x.

Corollary 2.1. *Projective unifiers are preserved by extensions of logics: if ε is a projective L_1-unifier and L_2 is an extension (by axiomax, rules and both) of L_1, then ε is a projective L_2-unifier*

The notions of a projective formula, a projective unifier and that of a projective substitution are due to S.Ghilardi, see [8] and [9] ('projective unifier' appeared first in [2]). "A is a projective formula" in L is the translation into logic

of the fact that the free algebra generated by $Var(A)$ divided by the congruence determined by "$A = \top$", is *projective* in the variety of algebras determined by L. It is known that projective algebras are retracts of free algebras, and "retracts" translated into logic give rise to the definition of projective unifiers.

S.Ghilardi, using projective formulas, proved that unification in logics $K4, GL$, $S4$, Grz is finitary and finite complete sets of unifiers can be effectively computed, see [9], [10]. Moreover, unification in $K4.2$ and $S4.2$ is unitary, [11].

A 'projective substitution' satisfy the above condition (ii) but may not be a unifier for A. One can extend (ii) to hold for any formula B: if ε is an L-projective substitution for A, then

(a) $A \vdash_L B[\varepsilon] \leftrightarrow B$, for each formula B;

(b) $A \vdash_L A[\varepsilon]$.

Projective substitutions are closed under compositions and are preserved under extensions of L.

If ε_0 is a unifier for A in L, then ε_0 is said to be a *most general unifier* (or, an *mgu*) for A, on the ground of L, if for each L-unifier ε there is a substitution δ such that $\varepsilon =_L \varepsilon_0\delta$, (i.e. each L-unifier is an instantiation of ε_0). Now, if δ is any unifier for A and ε a projective unifier for A then, by (ii), $\vdash_L x[\varepsilon\delta] \leftrightarrow x[\delta]$. Hence each projective unifier for A is an *mgu* for A.

Classical logic has projective unifiers for each unifiable formula A. Let τ_0 be a ground unifier for A. Then, ε given by a formula (for $x \in Var(A)$),

$$x[\varepsilon] = (A \to x) \wedge (A \vee x[\tau_0]),$$

is a projective unifier for A. ε is equivalent to the "Löwenheim unifier", cf. [2].

First we use (i),(ii) to establish that a modal logic L having "many" projective unifiers must contain $S4.3 = S4$ plus $\Box(\Box A \to \Box B) \vee \Box(\Box B \to \Box A)$.

Theorem 2.2. *If every unifiable formula in a modal logic L has a projective unifier, then* $\Box(\Box y \to \Box z) \vee \Box(\Box z \to \Box y) \in L$*, that is,* $S4.3 \subseteq L$.

Proof. Let ε be a projective L-unifier for a formula A. Then, by (ii),

$$\vdash_L (\Box A \wedge \Box x) \to \Box x[\varepsilon] \quad \text{and} \quad \vdash_L \Box x[\varepsilon] \to \Box(\Box A \to \Box x), \quad \text{for each variable } x;$$

the above holds true if one takes $A = \Box(\Box y \to \Box z) \vee \Box(\Box z \to \Box y)$. One can show that

$$(\star) \qquad \Box A \wedge \Box x =_L \Box x \ \text{ and } \ \Box(\Box A \to \Box x) =_L \Box x, \ \text{ for each } x \in Var(A).$$

Since each variable is 'boxed' in A, the above suffices to show that $A[\varepsilon] =_L A$ which would give us $\vdash_L A$ since ε is a unifier for A. □

3 Building Projective Unifiers in $S4.3$

Now we present some main steps of the construction of a projective unifier for a given unifiable formula A in $S4.3$, also using suitable examples.

Example 3.1. Assume: A is unifiable, $v\colon Var(A) \to \{\top, \bot\}$ is a ground unifier for A. The substitution ε, for $x \in Var(A)$: $x[\varepsilon] = (\Box A \to x) \wedge ((A \vee x[v]),)$ can be extended to arbitrary formula B (to be a unifier)

$$B[\varepsilon] = \begin{cases} \Box A \to B & \text{if} \quad B[v] = \top \\ \Box A \wedge B & \text{if} \quad B[v] = \bot \end{cases}$$

if $\Box(\Box A \to B_1) = \Box(\Box A \to B)$, but this holds only if $\Box A = \Diamond\Box A$, which is not $S4.3$ valid but $S5$ valid, we only get that ε is a projective unifier in **S5**.

Example 3.2. Let $A = \Box x \vee \Box\sim x$; two ground unfiers: two projective substitutions

$\varepsilon_0 : x/A \wedge x, \quad$ where $A \wedge x \ = \ (\Box x \vee \Box \sim x) \wedge x \ = \ \Box x$

$\varepsilon_1 : x/A \to x, \quad$ where $A \to x \ = \ (\Box x \vee \Box \sim x) \to x \ = \ \Diamond x$

Neither ε_0, nor ε_1 is a unifier for A, but if one takes $\varepsilon_0\varepsilon_1$ and $\varepsilon_1\varepsilon_0$ then: $A[\varepsilon_0\varepsilon_1] \ = \ \Diamond\Box\Diamond x \to \Box\Diamond x \ = \ \top$, $A[\varepsilon_1\varepsilon_0] \ = \ \Diamond\Box x \to \Box\Diamond\Box x \ = \ \top$. Hence, both $\varepsilon_0\varepsilon_1$ and $\varepsilon_0\varepsilon_1$ are projective unifiers for A but $x[\varepsilon_0\varepsilon_1] \ = \ \Diamond\Box x$ and $x[\varepsilon_1\varepsilon_0] \ = \ \Box\Diamond x$ i.e. $\varepsilon_0\varepsilon_1$ and $\varepsilon_1\varepsilon_0$ are not equivalent. Hence, a unifiable formula may have, if any, several non-equivalent projective unifiers.

Main idea. To get a projective unifier for a given unifiable formula A: • define a sequence $\varepsilon_1, \ldots, \varepsilon_n$ of projective substitutions for A in L, • take their composition $\varepsilon = \varepsilon_1 \cdots \varepsilon_n$. If sufficiently many projective substitutions is taken in the sequence $\varepsilon_1, \ldots, \varepsilon_n$, a unifier for A can be expected. But - what 'sufficiently many'means ? and - the order of substitutions in the sequence is important.

Ghilardi's *simplified Löwenheim substitutions* for A: $\Box A \to x$ or $\Box A \wedge x$, for each $x \in Var(A)$ are taken as $\varepsilon_1, \ldots, \varepsilon_n$. In logics containing $S4$, a formula A is projective iff a suitable composition of simplified Löwenheim substitutions is a unifier for A. We extend the class of simplified Löwenheim substitutions by allowing $x[\varepsilon_i]$ to be $A \to x$, or $A \wedge x$ (in addition to $\Box A \to x$ or $\Box A \wedge x$) and allowing $x[\varepsilon_i] = x$, as simplified Löwenheim substitutions, but not for the master formula A, but for its certain fragments.

The basic step is the reduction of a formula to its *conjunctive normal form* (CNF) in which only variables are 'boxed', at the cost of introducing fresh variables.

Theorem 3.3. *For every formula A one can find two disjoint sets of variables X and Y and a CNF formula $A^\star$ which is a conjunction of the following clauses:*

$$(\star) \qquad \Box x_1 \vee \cdots \vee \Box x_n \vee \sim \Box y_1 \vee \cdots \vee \sim \Box y_m \vee l_1 \vee \cdots \vee l_k$$

where $x_i \in X$ for $0 \leq i \leq n$, and $y_i \in Y$ for $0 \leq i \leq m$, and l_i is a variable – or its negation – from the set $X \cup Y$ for $0 \leq i \leq k$, and

(i) A is unifiable iff $A^\star$ is unifiable;

(ii) if $A^\star$ has a projective unifier, then A has a projective unifier, as well.

Example 3.4. Let $A = z \vee C$ where z is a variable. If $z \notin Var(C)$, take $z/A \to z$ as a projective unifier for A: $A[z/A \to z] = (A \to z) \vee C = (z \vee C) \to (z \vee C) = \top$. If $z \in C$, take $z/\Box A \to z$.

Similarly, $z/\Box A \wedge z$ is a projective unifier for $A = \sim z \vee C$.

Example 3.5. Let $A = \Box x_1 \vee \cdots \vee \Box x_n$. Then $x_1/A \to x_1 \quad \cdots \quad x_n/A \to x_n$. is a projective unifier for A. It is evident for $n = 2$.

Example 3.6. The worst case, U-clauses: $\Box x_1 \vee \cdots \vee \Box x_n \vee \sim \Box y_1 \vee \cdots \vee \sim \Box y_m$, where $n \geq 1$. Let $\{y_1, \ldots, y_m\} = U$ and write down the above clause as $\Box U \to \Box x_1 \vee \cdots \vee \Box x_n$. Let ε be the substitution $x_1/A \to x_1 \quad \cdots \quad x_n/A \to x_n$. It is clear that ε is projective for A. Note that ε is a substitution for the variables X and hence $A[\varepsilon] = \Box U \to \Box(A \to x_1) \vee \cdots \vee \Box(A \to x_n) =$
$\Box U \to \big(\Box U \wedge \Box(A \to x_1)\big) \vee \cdots \vee \big(\Box U \wedge \Box(A \to x_n)\big) \quad =$
$\Box U \to \Box\big((\Box x_1 \vee \cdots \vee \Box x_n) \to x_1\big) \vee \cdots \vee \Box\big((\Box x_1 \vee \cdots \vee \Box x_n) \to x_n\big)$

Theorem 3.7 ([6]). *Each unifiable formula has a projective unifier in* $S4.3$.

A projective unifier for a unifiable formula A is defined as the composition $\varepsilon_1 \cdots \varepsilon_n$ of the described 'partial' unifiers for A. The decisive step of the reduction procedure is the removal of all U-clauses, for any nonempty $U \subseteq Y$, in accordance with a *'linearization'* of the inclusion relation on subsets of Y (similar to Ghilardi [9]).

Corollary 3.8 ([6]). *Every unifiable formula in a modal logic L containing $S4$ has a projective unifier if and only if $S4.3 \subseteq L$.*

We give an alternative proof, based on Kripke semantics, of Theorem 3, by means of Ghilardi's [9] characterization of projective formulas in modal logics. We recall his result. A variant of a Kripke model $< \mathbf{F}, u >$, where $\mathbf{F} =< F, R, \rho >$ is a finite rooted frame and $u : F \to P(Var)$ is a Kripke model $< \mathbf{F}, u_0 >$ such that $u(p) = u_0(p)$ holds for all $p \notin cl(\rho)$. A class $\mathbf{K} \subseteq Mod_L$ of Kripke models over is said to have the extension property if for every Kripke model $< \mathbf{F}, u > \in Mod_L$, if $< \mathbf{F}_p, u_p > \in \mathbf{K}$ holds for every $p \notin cl(\rho)$, then there is a variant $< \mathbf{F}, u_0 >$ of $< \mathbf{F}, u >$, such that $< \mathbf{F}, u_0 > \in \mathbf{K}$.

Theorem 3.9 (Ghilardi [9]). . *The following conditions are equivalent:*
(i) A is projective;
(iii) $Mod_L(A)$ has the extension property.

$Mod_{S4.3}(A)$ has the extension property since any $S4.3$-frame $\mathbf{F}$ is a quasi-chain.

Corollary 3.10. *Let L be a logic containing $S4.3$. Then every formula A unifiable in L is projective, hence, A has a projective unifier.*

The drawback of this approach is its (highly) exponential complexity.

This result is, in a sense, optimal: removing any axiom from $S4.3$ results in lacking projective unifiers. The modal logic $K4.3$ (=$S4.3$ minus the axiom T) is axiomatized by the formulas: $4: \quad \Box\Box A \to \Box A; K: \quad \Box(A \to B) \to (\Box A \to \Box B); .3: \quad \Box(\Box A \to \Box B) \vee \Box(\Box B \to \Box A)$. In $K4.3$ the formula $\Box x$ does not have a projective unifier: the only unifier of $\Box x$ is $x/\top$, but $\Box x \vdash_{K4.3} x \leftrightarrow \top$ is not valid as it would lead to $K4.3 \vdash \Box x \to x$, a contradiction.

Applications: Almost Structural Completeness

A (structural) rule $r : A/B$ is *admissible* in L, if $\vdash_L A[\tau] \;\Rightarrow\; \vdash_L B[\tau]$, for every τ; a rule $r : A/B$ is *derivable* in L if $A \vdash_L B$; a rule $r : A/B$ is *passive* or *overflow* if A is not unifiable. e.g. $P_2 : \Diamond x \wedge \Diamond \neg\, x/ \perp$ is passive in $S5$.

Hence, a rule $r : A/B$ is *admissible* in L if $\vdash_L B[\sigma]$ for any unifier σ for A. If σ is a *projective* unifier for A then, immediately, r is derivable in L.

A logic L is *Structurally Complete*, *SC*, if every (structural) admissible rule is is derivable in L. A logic L is *Passively Structurally Complete*,*PSC*, if each passive rule is derivable in L. A logic L is *Almost Structurally Complete*, *ASC*, if every (structural) admissible rule which is not passive (or: rule with unifiable premises in L) is derivable in L. Hence, $SC = ASC + PSC$

Theorem 3.11. *Every modal logic L extending $S4.3$ is almost structurally complete. Moreover, L is hereditarily structurally complete iff McKinsey axiom M : $\Box\Diamond\alpha \to \Diamond\Box\alpha$ is in L.*

Bibliography

1. Baader,F., Snyder, W., *Unification Theory.* In: Robinson, A. and Voronkov, A. (eds.) *Handbook of Automated Reasoning*, Ch.8, Elsevier Science Publ., MIT, **1** (2001), 445–533.
2. Baader, F.,Ghilardi, S., *Unification in modal and description logics*, Logic Journal of the IGPL, doi:10.1093/jigpal/jzq008 in printing(2010),
3. Dzik, W., *Splittings of lattices of theories and unification types*, Contributions to General Algebra 17 (2006), 71– 81.
4. Dzik W., *Unification types in logic*, Silesian University Press, Katowice (2007).
5. Dzik, W., *Remarks on Projective Unifiers*, Bull. of the Sect. of Logic 40, (1-2), (2011), 71– 81.
6. Dzik, W., Wojtylak P., *Projective unification in modal logic*, Logic Journal of the IGPL, doi: 10.1093/jigpal/jzr028, published online: June 6, 2011.
7. Ghilardi S., *Unification through projectivity*, Journal of Symbolic Computation 7 (1997), 733–752.
8. Ghilardi S., *Unification in intuitionistic logic*, Journal of Symbolic Logic 64(2) (1999), 859–880.
9. Ghilardi S., *Best solving modal equations*, Annals of Pure and Applied Logic 102 (2000), 183–198.
10. Ghilardi S., *A resolution tabloux algorithm for projective approximation*, Logic Journal of the IGPL 10(3) (2002), 227–241.
11. Ghilardi, S., Sacchetti, L., *Filtering unification and most general unifiers in modal logic*, The Journal of Symbolic Logic 69 (2004),879–906.
12. Łoś J., Suszko R., *Remarks on sentential logics*; Indagationes Mathematicae 20 (1958), 177–183.
13. Wroński A., *Transparent Unification Problem*, Reports on mathematical logic 29 (1995), 105–107.
14. Wroński A., *Transparent verifiers in intermediate logics*, Abstracts of the 54-th Conference in History of Mathematics, Cracow (2008).

Computing finite variants for subterm convergent rewrite systems ⋆

Ştefan Ciobâcă

LSV, ENS Cachan & CNRS

Abstract. Driven by an application in the verification of security protocols, we introduce the strong finite variant property, an extention of the finite variant property defined in [1] and we show that subterm convergent rewrite systems enjoy the strong finite variant property modulo the empty equational theory.
We argue that the strong finite variant property is more natural and more useful in practice than the finite variant property. We also compare the two properties and we provide a prototype implementation of an algorithm that computes a finite strongly complete set of variants for any term t with respect to a subterm convergent rewrite system.

1 Introduction

Given a term (e.g. $t = \mathbf{dec}(x, y)$) and a convergent term rewriting system (e.g. $\mathcal{R} = \{\mathbf{dec}(\mathbf{enc}(x, y), y) \to x\}$), we are interested in having a convenient symbolic representation of all normal forms $t\sigma{\downarrow}$ of instantiations $t\sigma$ of the term t.

In the above case, the normal form of $t\sigma$ will fall into one of the following two cases:

1. either $\sigma(x) =_{\mathcal{R}} \mathbf{enc}(s, \sigma(y))$ for some term s, in which case $t\sigma{\downarrow} = s{\downarrow}$
2. or $\sigma(x) \neq_{\mathcal{R}} \mathbf{enc}(s, \sigma(y))$ for any term s, in which case $t\sigma{\downarrow} = t(\sigma{\downarrow})$ (where $\sigma{\downarrow}$ denotes the normal form of σ).

Informally, rewrite systems for which instantiations of any term t can be classified into a finite number of categories such as above are said to have the *finite variant property.*

The finite variant property is useful in symbolic analysis of security protocols [1] and in solving unification and disunification problems [1, 2].

Contributions. We work with subterm convergent rewrite systems, a class of rewrite systems relevant to security protocol analysis [3]. We show that these rewrite systems have the finite variant property and a slightly stronger property which we call the *strong finite variant property.* The proofs are constructive and we implement the algorithm for computing a strongly complete finite set of variants of a term in the tool SubVariant.

⋆ This work has been partly supported by the ANR SeSur AVOTÉ.

Related work. The finite variant property was first introduced in [1]. In [4], sufficient conditions and necessary conditions for the finite variant property are introduced. Variant narrowing [2] is a complete procedure for equational unification inspired by the finite variant property. A modular proof method for termination based on the notion of variant is proposed in [5]. Several techniques [6, 7, 8] for verifying security protocols make use of the finite variant property as a sub-step in the algorithm.

2 Preliminaries

We consider standard notations for a term algebra: a finite *signature* $\mathcal{F}$, each function symbol $f \in \mathcal{F}$ having an arity $\mathsf{ar}(f) \in \mathbb{N}$, a countably infinite set of $\mathcal{X}$ of *variables*, the set $\mathcal{T}(\mathcal{X}_0)$ denoting all *terms* bulid inductively from the variables $\mathcal{X}_0 \subseteq \mathcal{X}$ by applying function symbols from $\mathcal{F}$. Given a term t, $\mathsf{vars}(t)$ is the set of variables appearing in t.

Substitutions are defined as usual, with $t\sigma$ denoting the term t after application of the substitution σ. The identity substitution is denoted id. The restriction of a substitution σ to a set X of variables is denoted $\sigma[X]$.

We define *positions* as usual, with $\mathsf{pos}(t)$ denoting the positions of a term $t \in \mathcal{T}(\mathcal{X})$. We denote the subterm of t at position $p \in \mathsf{pos}(t)$ by $t|_p$. The term t with position $p \in \mathsf{pos}(t)$ instantiated to s is denoted $t[s]_p$. If $t \in \mathcal{T}(\mathcal{X})$, we will denote by $\mathsf{st}(t)$ the set of *subterms* of t. If $s \in \mathsf{st}(t)$, we write $s \sqsubseteq t$. By $\mathsf{mgu}(s,t)$ we denote the most general unifier of two unifiable terms s and t.

If $\mathcal{R}$ is a rewrite system, we use $\rightarrow_{\mathcal{R}}$ for the one-step rewrite relation defined by R and $\rightarrow^*_{\mathcal{R}}$ for the transitive and reflexive closure of $\rightarrow_{\mathcal{R}}$. If $\mathcal{R}$ is convergent, we will denote by $t\downarrow_{\mathcal{R}}$ the normal form of t. We say that two terms s and t are *equal modulo* $\mathcal{R}$, and we write $s =_{\mathcal{R}} t$, if $s\downarrow = t\downarrow$.

We are interested in a particular class of convergent term rewriting systems:

Definition 2.1 (subterm convergent rewrite system).
A rewrite system $\mathcal{R}$ is called subterm convergent *if it is convergent and if for all rewrite rules $l \rightarrow r$ of $\mathcal{R}$ we have:*

1. *either $r \sqsubseteq l$ (we then call $l \rightarrow r$ a* subterm rule*)*
2. *or $\mathsf{vars}(r) = \emptyset$ and $r = r\downarrow_{\mathcal{R}}$ (we then call $l \rightarrow r$ an* extended rule*)*

The first type of rewrite rule, which justifies the name of these rewrite systems, was introduced in [3]. Subsequently, in [9], the extended rules were introduced. We treat here both types of rules.

3 The finite variant property

For any convergent term rewriting system $\mathcal{R}$ and any term t, we define the notion of complete set of variants of t with respect to $\mathcal{R}$:

Definition 3.1. *A set of substitutions* $\{\sigma_1, \ldots, \sigma_n\}$ *is a* complete set of variants *of a term* t *(with respect to* $\mathcal{R}$*) if for any substitution* ω*, we have that* $(t\omega)\downarrow = (t\sigma_i)\downarrow\tau$ *for some* $1 \le i \le n$ *and some substitution* τ*.*

Note that the difficulty in the above definition is that the term $(t\sigma_i)\downarrow\tau$ is not normalized after the application of the substitution τ to the term $(t\sigma_i)\downarrow$. Therefore all the rewrite steps to reach a normal form from $t\omega$ must be captured by some substitution σ_i as it is demonstrated bellow in Example 3.2. This means in particular that the set $\{\mathsf{id}\}$ consisting of the identity substitution is in general not a finite complete set of variants.

A convergent term rewriting system $\mathcal{R}$ is said to have the finite variant property if any term t admits a finite complete set of unifiers with respect to the rewrite system $\mathcal{R}$.

Example 3.2. Let $t = \mathbf{dec}(x, y)$ and $\mathcal{R} = \{\mathbf{dec}(\mathbf{enc}(x, y), y) \to x\}$. Then we have that $\sigma_1 = \mathsf{id}$ (the identity substitution) and $\sigma_2 = \{x \mapsto \mathbf{enc}(z, y)\}$ form a complete set of variants of t.

Indeed, for any substitution ω in normal form, we have that $\mathbf{dec}(x, y)\omega\downarrow = \mathbf{dec}(x, y)\omega$ (if the decryption does not succeed at the head) or $\mathbf{dec}(x, y)\omega\downarrow = t'$ if the decryption succeeds and therefore $x\omega = \mathbf{enc}(t', y\omega)$.

The following example illustrates that a finite complete set of variants does not always exist.

Example 3.3. We consider the term rewriting system $\mathcal{R} = \{f(g(x)) \to g(f(x))\}$ and the term $t = f(x)$. By analyzing the substitutions $\omega_i = \{x \mapsto g^i(y)\}$ $(i \in \mathbb{N})$ and the normal forms $t\omega_i\downarrow = g^i(f(y))$ $(i \in \mathbb{N})$, it can be proven that any complete set of variants of t will contain all of the substitutions $\sigma_i = \{x \mapsto g^i(y)\}$ for $i \in \mathbb{N}$ and up to renaming of the variable y. Therefore this term rewriting system does not have the finite variant property.

Rewrite systems for which any term t has a finite complete set of variants are said to have the *finite variant property.*

4 The strong finite variant property

We now define what is a strongly complete set of variants of a term t with respect to a convergent term rewriting system $\mathcal{R}$:

Definition 4.1. *A set of substitutions* $\sigma_1, \ldots, \sigma_n$ *is a* strongly complete set of variants *of* t *(with respect to the rewrite system* $\mathcal{R}$*) if for any substitution* ω*, we have that* $\omega[X]\downarrow = (\sigma_i\downarrow\tau)[X]$[1] *for some substitution* τ *and some* σ_i *such that* $(t\omega)\downarrow = (t\sigma_i)\downarrow\tau$*, where* $X = \mathsf{vars}(t)$ *is the set of variables appearing in* t*.*

[1] Recall that the notation $\omega[X]$ denotes the restriction of the substitution ω to the variables in X.

Note that in the above definition the condition $\omega[X]\downarrow = (\sigma_i\downarrow\tau)[X]$ does not in general imply $(t\omega)\downarrow = (t\sigma_i)\downarrow\tau$: take $\mathcal{R} = \{\mathbf{dec}(\mathbf{enc}(x, y), y) \to x\}$, $t = \mathbf{dec}(x, y)$, $\omega = \mathbf{enc}(z, y)$, $\sigma_i = \mathsf{id}$ (the identity substitution). We have that $\tau = \omega$ is such that $\omega[X]\downarrow = (\sigma_i\downarrow\tau)[X] = \tau[X]$ but $(t\omega)\downarrow = z \neq (t\sigma_i)\downarrow\tau = \mathbf{dec}(\mathbf{enc}(z, y), y)$.

A convergent term rewriting system $\mathcal{R}$ is said to have the strong variant property if any term t admits a finite strongly complete set of variants.

As with complete sets of variants, a finite strongly complete set of variants does not exist in general.

The main difference is that in the strong version, we ask that the substitutions σ_i match the normal forms of all variables appearing in t. The following example illustrates this idea and shows that the notion of *complete set of variants* and the notion of *strongly complete set of variants* do not coincide.

Example 4.2. We consider the (subterm convergent) term rewriting system

$$\mathcal{R} = \{h(f(x), y) \to y, h(g(x), y) \to y\}$$

and the term $t = h(x, y)$.

The $S = \{\sigma_1 = \mathsf{id}, \sigma_2 = \{x \mapsto f(z)\}$ is a complete finite set of variants of t. Note that S does not contain the substitution $\sigma_3 = \{x \mapsto g(z)\}$.

However, S is not a strongly complete set of variants of t: if we consider the substitution $\omega = \{x \to g(a)\}$ for some constant a, we have that:

1. $\omega\downarrow = \sigma_1\tau_1$ (with $\tau_1 = \{x \mapsto g(a)\}$), but $t\omega\downarrow \neq t\sigma_1\downarrow\tau_1$.
2. $\omega\downarrow \neq \sigma_2\tau_2$ for any substitution τ_2.

However, the set $S \cup \{\sigma_3\}$ is a strongly complete set of variants of t.

One application of the finite variant property is in solving unification problems $s \stackrel{?}{=}_{\mathcal{R}} t$ modulo the rewrite system by treating the equality sign as a free function symbol and then finding all variants of the equation. In this context of equational unification, we argue that strongly complete sets of variants are more natural:

Example 4.3. Continuing Example 4.2, if only complete sets of variants (and not strongly complete sets) are used for equational unification, some unifiers are missed. The equation

$$h(z, y) \stackrel{?}{=}_{\mathcal{R}} y$$

has S (defined in Example 4.2) as a complete set of variants. Starting from S, only the unifier $\{z \mapsto f(x)\}$ is found. However, by considering the strongly complete set of variants $S \cup \{\sigma_3\}$, the unifier $\{z \mapsto g(x)\}$ is found as well.

Another application is the verification of security protocols where strongly complete set of variants can be used to get rid of the equational theory.

4.1 Strict containment

It is easy to see that a term rewriting system having the strong finite variant property also has the finite variant property. The reverse is not true: term rewriting systems having the finite variant property need not have the strong finite variant property. Let us consider the signature $\mathcal{F} = \{f/1, g/1, c_0/0, c_1/0, \ldots\}$ and the following convergent term rewriting system

$$\mathcal{R} = \{f(g(x)) \to f(x)\}.$$

It is easy to observe that any term t has a normal form which is either $t\downarrow = g^n(f^m(x))$ or $t\downarrow = g^n(f^m(c_k))$ for some variable x and some integers n, m, k.

The identity substitution id forms by itself a complete set of variants of any term t built over the signature $\mathcal{F}$.

However, $\mathcal{R}$ does not have the strong finite variant property. This is illustrated by the following example.

Example 4.4. Let $t = f(x)$. By analyzing the instantiations $t\omega_i$ where the substitutions ω_i are defined $\omega_i = \{x \mapsto g^i(y)\}$ $(i \in \mathbb{N})$, it can be shown that $\sigma_i[\{x\}] = \{x \mapsto g^i(z)\}$ $(i \in \mathbb{N})$ must be contained in any strongly complete set of variants (up to renaming of z). Therefore any strongly complete set of variants of t is infinite.

4.2 In the presence of free symbols

The above example depends on the signature $\mathcal{F}$. Indeed, in the presence of a free symbol of arity greater than or equal to 2, we have that the two notions coincide since a finite complete set of variants of the term $\mathit{tuple}(t, x_1, \ldots, x_n)$ (where $\mathsf{vars}(t) = \{x_1, \ldots, x_n\}$ and where *tuple* is a free function symbol) is a strongly complete set of variants of the term t.

Note that the free symbol *tuple* of arity $n+1$ can be *encoded* by a free symbol of arity ≥ 2 by replacing, for example, the term $\mathit{tuple}(t_1, \ldots, t_k)$ with $f(t_1, f(t_2, f(\ldots, f(t_{k-1}, t_k))))$ in case f is a free binary symbol.

We have shown that the notions of strong finite variant property and finite variant property coincide when the signature contains a free function symbol of arity ≥ 2. This follows because a complete set of variants of $\mathit{tuple}(t, x_1, \ldots, x_n)$ is a strongly complete set of variants of t. However note that even in the presence of such a free symbol, a *complete* set of variants of t is not always *strongly complete* for t (see Example 4.2).

5 Algorithm for a (strongly) complete set of finite variants

We show that subterm convergent term rewriting systems have the strong finite variant property by giving an algorithm that computes a finite strongly complete set of variants for a term t and for a subterm convergent rewrite system $\mathcal{R}$.

In the following we denote by $p\!\uparrow$ the set of all positions that are descendants of p (including p itself): $p\!\uparrow = \{q \mid q = p \cdot p' \textit{ for some } p'\}$.

The algorithm we present for computing a strongly complete finite set of variants is based on a refinement of *narrowing*. Each narrowing step (denoted hereafter $\hookrightarrow$) works on a configuration $(t, \mathcal{P}, \sigma)$ consisting of a term t, a set of positions $\mathcal{P}$ of t at which we will apply narrowing and a substitution σ in which a variant will be accumulated.

$$\frac{\begin{array}{l} p \in \mathcal{P} \\ l \to r \in \mathcal{R} \qquad \mathsf{vars}(\{l, r\}) \cap \mathsf{vars}(t) = \emptyset \\ \theta = \mathsf{mgu}(l, t|_p) \end{array}}{(t, \mathcal{P}, \sigma) \hookrightarrow (t\theta[r\theta]_p, \mathcal{P} \setminus p\!\uparrow, \sigma \circ \theta)}$$

Fig. 1. Narrowing step

To compute a complete finite set of variants of some term t, we will begin with the initial configuration $\mathcal{C}_0 = (t, \mathsf{pos}_{init}(t), \mathsf{id})$ where $\mathsf{pos}_{init}(t)$ denotes all non-variable positions of t and non-deterministically apply narrowing steps.

Each narrowing step non-deterministically chooses a rewrite rule $l \to r$ and a position p from $\mathcal{P}$ where narrowing is performed. The choice of $\mathcal{P} = \mathsf{pos}_{init}(t)$ in the initial configuration is a way to enforce the *basic restriction*, that is, narrowing is only performed strictly inside t (and not inside the variables of t). Furthermore, if we have performed narrowing at a position p and because of the specificity of subterm convergent rewrite systems, there is no need to consider this position or any of its descendants anymore and therefore they are removed from $\mathcal{P}$. At each narrowing step, the variant of the initial term is accumulated in σ. If by $\hookrightarrow^*$ we denote the reflexive-transitive closure of $\hookrightarrow$, we have that:

Theorem 5.1 (Correctness).
If $\mathcal{R}$ is a subterm convergent rewrite system, then the set

$$\Sigma = \{\sigma \mid (t, \mathsf{pos}_{init}(t), \mathsf{id}) \hookrightarrow^* (t', \mathcal{P}', \sigma)\}$$

is a finite complete set of variants of t with respect to $\mathcal{R}$.

A subterm convergent rewrite system remains subterm convergent by the addition of a free function symbol *tuple*. Therefore, to compute a finite strongly complete set of variants of a term t it is sufficient to compute a finite complete set of variants of the term $\textit{tuple}(t, x_1, \ldots x_n)$, where $\mathsf{vars}(t) = \{x_1, \ldots, x_n\}$.

6 Conclusion and further work

We have shown that subterm convergent rewrite systems have the strong finite variant property and we have implemented our algorithm in Section 5 in

the prototype tool SubVariant (available at `http://www.lsv.ens-cachan.fr/~ciobaca/subvariant`).

We are currently using this result to obtain a decision procedure for verifying equivalences between cryptographic processes. Another possible direction for future work is to find algorithms for computing strongly complete set of variants modulo associative-commutative function symbols. An extended version of this paper, including the proofs missing due to space constraints is available as a research report [10].

7 Acknowledgements

I would like to thank Steve Kremer, Stéphanie Delaune and the anonymous reviewers for interesting comments regarding this work.

Bibliography

1. H. Comon-Lundh and S. Delaune, "The finite variant property: How to get rid of some algebraic properties," in *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA'05)*, ser. Lecture Notes in Computer Science, J. Giesl, Ed., vol. 3467. Nara, Japan: Springer, Apr. 2005, pp. 294–307. [Online]. Available: http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/rta05-CD.pdf
2. S. Escobar, J. Meseguer, and R. Sasse, "Variant narrowing and equational unification," *Electron. Notes Theor. Comput. Sci.*, vol. 238, pp. 103–119, June 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1556507.1556661
3. M. Abadi and V. Cortier, "Deciding knowledge in security protocols under equational theories," in *ICALP*, 2004, pp. 46–58.
4. S. Escobar, J. Meseguer, and R. Sasse, "Effectively checking the finite variant property," in *RTA*, 2008, pp. 79–93.
5. F. Durán, S. Lucas, and J. Meseguer, "Termination modulo combinations of equational theories," in *FroCos*, 2009, pp. 246–262.
6. S. Bursuc and H. Comon-Lundh, "Protocol security and algebraic properties: Decision results for a bounded number of sessions," in *RTA*, 2009, pp. 133–147.
7. S. Bursuc, H. Comon-Lundh, and S. Delaune, "Deducibility constraints, equational theory and electronic money," in *Rewriting, Computation and Proof*, 2007, pp. 196–212.
8. Y. Chevalier and M. Kourjieh, "On the decidability of (ground) reachability problems for cryptographic protocols (extended version)," *CoRR*, vol. abs/0906.1199, 2009.
9. M. Baudet, V. Cortier, and S. Delaune, "YAPA: A generic tool for computing intruder knowledge," in *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA'09)*, ser. Lecture Notes in Computer Science, R. Treinen, Ed., vol. 5595. Brasília, Brazil: Springer, Jun.-Jul. 2009, pp. 148–163. [Online]. Available: http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/BCD-rta09.pdf
10. Ş. Ciobâcă, "Computing finite variants for subterm convergent rewrite systems," Laboratoire Spécification et Vérification, ENS Cachan, France, Research Report LSV-11-06, Apr. 2011, 16 pages. [Online]. Available: http://www.lsv.ens-cachan.fr/Publis/RAPPORTS_LSV/PDF/rr-lsv-2011-06.pdf

A Unification Algorithm to Compute Overlaps in a Call-by-Need Lambda-Calculus with Variable-Binding Chains

Conrad Rau and Manfred Schmidt-Schauß*

Institut für Informatik, Goethe-Universität, D-60054 Frankfurt, Germany
{rau,schauss}@ki.informatik.uni-frankfurt.de

Abstract. We extend the unification algorithm from previous work of the authors to cover the call-by-need λ-calculus LR. The main task of the unification algorithm is to compute all possible overlaps (also called forks) between the reduction rules of LR and a set of program transformations. The new contribution is that the variable-binding chains (a form of indirections) that occur in the rules and the transformations are in the scope of the unification method. This is achieved through the use of additional term syntax to treat variable-binding chains of any length. The result is a unification algorithm that terminates and computes a finite and complete set of overlaps (i.e. critical pairs) between all rules and given transformations.

1 Introduction and Motivation

Proving correctness of program transformations in call-by-need λ-calculi can be done using a method that heavily relies on sets of reduction diagrams, which can be interpreted as a description of local confluence between program transformations and the reduction rules of the λ-calculus. The material step to generate such diagram sets is the determination of overlaps (also called forks) between reductions and transformations. The work in [5] presented a terminating and complete unification algorithm that computes all forks and thus the critical pairs between the reduction rules and a set of transformation rules in the calculus L_{need}, a call-by-need lambda calculus with a recursive let. In this paper we extend this work to the more expressive call-by-need lambda calculus LR which was used in [7] to model the functional part of the programming language Haskell. The calculus LR extends L_{need} in several aspects: (i) There are data types in the form of data constructors and a syntax for case analysis, (ii) The reduction rules are defined using variable-binding chains, and (iii) there is a `seq`-construct for enforcing sequential reduction. The variable chains are required in this calculus to enable correctness proofs of the reduction rules seen as transformations.

The main motivation for this work is to make a step forward towards the automatic verification of the correctness of program transformations in the LR-calculus. Therefore, as an intermediate goal we are interested in the automatic

* The authors are supported by the DFG under grant SCHM 986/9-1.

verification of the reduction diagrams given in [7]. Another motivation is to develop tools that are also applicable to other, non deterministic and concurrent calculi aiming at automatically detecting program transformation and proving them to be correct.

The central idea for computing all overlaps is to use (a variant of) first-order unification. It works as follows: The expressions in the reduction rules (of the λ-calculus LR), which are in fact rule schemes, are translated into many sorted first-order terms. These terms are extended with several special syntactic constructs to finitely capture the infiniteness of reduction rule sets, the letrec-construct, the binding primitives and the different classes of context variables. The letrec-construct is translated using the equational theory of left-commutativity. The higher-order feature of bound variables is translated such that bound variables are terms (i.e. variables) of an empty sort, i.e. without ground terms. Moreover, syntactical restrictions like enforcing different variable names in letrecs together with the distinct variable convention provide a means that can be checked in the first-order translation to avoid illegal or unsound unifiers that otherwise would equate variables that must be kept different, like in the first-order translation of $\lambda x.x$ and $\lambda x.y$. Context names in rules are encoded as context variables at the term level. All these aspects of the first-order encoding of a higher order calculus are treated as in [5].

The novelty of our unification algorithm is that it can unify terms s_1, s_2, where both s_1 and s_2 may contain variable chains. Variable chains are the first-order encodings of environments like ($\texttt{letrec}\ x_1 = v, x_2 = x_1, x_3 = x_2, \ldots, x_n = x_{n-1}\ \texttt{in}\ A[x_n]$). There are also the binding chains with bindings $x = A[y]$, where A is a context variable. The additional complication is that it is unavoidable that these binding chains occur on both sides of equations in unification problems, in contrast to the (proper) binding chains in [5] that only appear on one side of equations. We devise a unification procedure that can treat binding chains of any length, first-order encoded as $\mathsf{VCh}(x, y, n)$ for variable-only chains and $\mathsf{NCh}(x, y, n')$ for other binding chains. The unification rules exploit the equational theory of left-commutativity as well as the distinct variable convention-restrictions and so can enforce termination without losing completeness.

The main result of this work is a translation of the higher-order overlapping problems into extended first-order unification problems (called initial problems) and a complete and terminating unification algorithm for these initial problems. This enables the automation of the computation of complete diagram sets for the calculus LR, in particular for the transformations that are derived from the reduction rules of the calculus.

2 Motivating Example for the Unification Algorithm

We demonstrate the main ideas and effects of the encoding and the unification rules by an extended example, since a formal development would exceed the space limit. Therefore we illustrate how the overlap of the reduction rule (no, cp-e)

with the transformation rule (cp-in) in the calculus LR can be computed. The respective rules are in Fig. 1, where C, A denote contexts.

(no, cp-e) $\texttt{letrec}\ x_1 = v, x_2 = x_1, \ldots, x_m = x_{m-1}, y_1 = A_1[x_m],$
$y_2 = A_2[y_1], \ldots, y_n = A_n[y_{n-1}], \mathit{Env}\ \texttt{in}\ A[y_n]$
$\rightarrow \texttt{letrec}\ x_1 = v, x_2 = x_1, \ldots, x_m = x_{m-1}, y_1 = A_1[v],$
$y_2 = A_2[y_1], \ldots, y_n = A_n[y_{n-1}], \mathit{Env}\ \texttt{in}\ A[y_n]$

(cp-in) $(\texttt{letrec}\ x_1 = v, x_2 = x_1, \ldots, x_k = x_{k-1}, \mathit{Env}\ \ \texttt{in}\ C[x_k])$
$\rightarrow (\texttt{letrec}\ x_1 = v, x_2 = x_1, x_k = x_{k-1}, \mathit{Env}\ \ \texttt{in}\ C[v])$

Where v is an abstraction and A_1 is a non-empty context.

Fig. 1. Reduction rule and Transformation rule of the calculus LR

The initial unification problem corresponding to the overlap of (no, cp-e) with the transformation (cp-in) is an equation between the first-order encoded left hand sides of the rules (with a context variable S indicating that the (cp-in) transformation may not occur at top position). It looks as follows:

$$\begin{array}{c} \mathit{let}(\mathit{env}^* \begin{pmatrix} \{\mathit{bind}(x_1, \mathit{lam}(v,s))\} & \cup\, \mathsf{VCh}(x_1, x, m) \cup \\ \{\mathit{bind}(y_1, A_1(\mathit{var}(x)))\} & \cup\, \mathsf{NCh}(y_1, y, n) \cup \mathit{Env} \end{pmatrix}, A(\mathit{var}(y))) \\ \doteq \\ S(\mathit{let}(\mathit{env}^*(\{\mathit{bind}(x_1', \mathit{lam}(v', s'))\} \cup \mathsf{VCh}(x_1', x', k) \cup \mathit{Env}'), C(\mathit{var}(x')))) \end{array} \quad (1)$$

We briefly elaborate on the translation used to arrive at the above unification problem: the letrec-environment (that is the set of LR-expression of form $x = s$) is encoded by a nesting of a binary function symbol env, similar to a list or set representation. The irrelevance of the order of elements in letrec-environments is achieved by the equality axiom $LC_{\mathit{env}} := \{\mathit{env}(x, \mathit{env}(y, z)) = \mathit{env}(y, \mathit{env}(x, z))\}$, i.e. env is a left-commutative function symbol (For the LC-theory and unification modulo LC see [2, 1]). We use the abbreviation $\mathit{env}^*(\{t_1, \ldots, t_m\} \cup r)$ to denote the term $\mathit{env}(t_1, \mathit{env}(t_2, \ldots, \mathit{env}(t_m, r) \ldots))$, and usually use union-notation within the first argument of env^*. A binding $x = y$ is encoded as $\mathit{bind}(x, \mathit{var}(y))$. Some terms in the first order encoding may not possess syntactical admissible counterparts in LR: for example $\mathit{env}^*(\{\mathit{bind}(x, \mathit{var}(y)), \mathit{bind}(x, \mathit{var}(z))\} \cup \mathit{Env})$, because x occurs twice at a binder position. Such terms are called syntactically incorrect w.r.t. LR.

The binding chains of variable length in the left hand sides of the LR-reductions (i.e. $x_2 = x_1, \ldots, x_m = x_{m-1}$ etc.) are encoded as $\mathsf{VCh}(x, y, n)$, and $\mathsf{NCh}(x, y, n)$, respectively, where x, y are the variables starting and ending the chain; they may also occur in the environment outside of the chain (which is the case for the encoding of the LR reductions and transformations). The integer variable n stands for the length, and it is assumed that the (implicit) intermediate variables and context variables are fresh. The meaning is that such a chain-construct expands to n bindings, for example $\mathsf{VCh}(x, y, 2)$ represents $z = x, y = z$, and in the first-order encoding:

$env(bind(z, var(x)), env(bind(y, var(z)), [.]))$, where z is a fresh variable. Similarly, $\mathsf{NCh}(x, y, 2)$ represents $z = A_1[x], y = A_2[z]$, and its first-order encoding is: $env(bind(z, A_1(var(x))), env(bind(y, (A_2(var(z)))), [.]))$, where A_1, A_2 are fresh context variables. The constructs for binding chains describe (possibly infinite) sets of terms. They bear some similarities to term schematizations used in [6, 3, 4].

We proceed to solve equation (1). Therefore we first use the unification rules as in [5]: One possibility is to guess the context variable S as empty. After that we decompose the *let*-terms. It remains to solve the equation $A(var(y)) \doteq C(var(x'))$ and

$$\begin{aligned} & env^* \begin{pmatrix} \{bind(x_1, lam(v, s))\} & \cup \mathsf{VCh}(x_1, x, m) \cup \\ \{bind(y_1, A_1(var(x)))\} & \cup \mathsf{NCh}(y_1, y, n) \cup Env \end{pmatrix} \\ \doteq\ & env^*(\{bind(x_1', lam(v', s'))\} \cup \mathsf{VCh}(x_1', x', k) \cup Env') \end{aligned} \tag{2}$$

Again the first equation can be treated using similar methods as in [5], taking care of the extended signature. The second equation (2), however, requires special treatment because of the variable chains occurring in both terms. We have to treat the integer variables m, n, k (for the lengths of chains) in a general way avoiding guessing natural numbers, since this would lead to an infinite number of solutions. Our unification method employs some crucial properties of the to-be-solved problems, such that infinite solution sets can be avoided, therefore leading to a terminating procedure without sacrificing completeness of the algorithm. The new unification rule that achieves this goal is **U-Chain** in Fig. 2, which states that there are two possibilities for unifying chains Ch_1 and Ch_2: They are

U-Chain:

$$\frac{\{env^*(\mathsf{Ch}(x_1, y_1, l_1) \cup M_1 \cup r_1) \doteq env^*(\mathsf{Ch}(x_2, y_2, l_2) \cup M_2 \cup r_2)\} \uplus P}{\text{choose one of the following possibilities}}$$

1) $\{l_1 \doteq l_1' + l + l_1'', l_2 \doteq l + l_2', env^*(\mathsf{Ch}(x_1, x_2, l_1'), \mathsf{Ch}(z, y_1, l_1'') \cup M_1 \cup r_1) \doteq env^*(\mathsf{Ch}(z, y_2, l_2') \cup M_2 \cup r_2)\} \cup P$

2) $\{l_1 \doteq l_2, x_1 \doteq x_2, y_1 \doteq y_2, env^*(M_1 \cup r_1) \doteq env^*(M_2 \cup r_2)\} \cup P$

Where z is a fresh variable of sort BV and l, l_1', l_1'', l_2' are fresh integer variables.

Dec-Chain

$$\frac{\{env^*(\{s_1\} \cup M_1 \cup r_1) \doteq env^*(\mathsf{Ch}(x, y, l) \cup M_2 \cup r_2)\} \uplus P}{\text{select one of the following possibilities}}$$

1) $\{l \doteq 1, s_1 \doteq bind(y, A(var(x))), env^*(M_1 \cup r_1) \doteq env^*(M_2 \cup r_2)\} \cup P$

2) $\{l \doteq 1 + l_1, s_1 \doteq bind(z, A(var(x))), env^*(M_1 \cup r_1) \doteq env^*(\mathsf{Ch}(z, y, l_1) \cup M_2 \cup r_2)\} \cup P$

3) $\{l \doteq l_1 + 1, s_1 \doteq bind(y, A(var(z))), env^*(M_1 \cup r_1) \doteq env^*(\mathsf{Ch}(x, z, l_1) \cup M_2 \cup r_2)\} \cup P$;

4) $\{l \doteq l_1 + 1 + l_2, s_1 \doteq bind(z_2, A(var(z_1))),$
$env^*(M_1 \cup r_1) \doteq env^*(\mathsf{Ch}(x, z_1, l_1) \cup \mathsf{Ch}(z_2, y, l_2) \cup M_2 \cup r_2)\} \cup P$

Where s_1 is a binding expression. z, z_1, z_2 are fresh variables of sort BV and A is either a fresh context variable of class $\mathcal{A}$ if Ch=NCh

Fig. 2. Two unification rules dealing with variable chains. (Here we use the symbol Ch to denote either chain construct.)

either identical (described by case 2, where the length of the chains and their start- and end-points are equated), or the initial part of Ch_2 is equal to some intermediate part of Ch_1 and both tails of Ch_2 and Ch_1 and the initial sequence of Ch_1 are disjoint (case 1). These (and the symmetrical case, where Ch_1 and Ch_2 are swapped) are the sole possibilities of equating bindings from chains: All other unification schemes of chain-bindings would result in terms that do not represent syntactically admissible LR-expressions. We will elaborate on this by applying the rule **U-Chain** to the equation (2) from our continued example, where one possible transformation (i.e. case 1) of **U-Chain**) yields:

$$\begin{array}{ll} m \doteq m_1 + l + m_2,\ k \doteq l + k_1, & \\ \mathit{env}^* \begin{pmatrix} \{\mathit{bind}(x_1, \mathit{lam}(v,s))\} & \cup\, \mathsf{VCh}(x_1, x_1', m_1) \cup \mathsf{VCh}(z, x, m_2) \cup \\ \{\mathit{bind}(y_1, A_1(\mathit{var}(x)))\} & \cup\, \mathsf{NCh}(y_1, y, n) \cup \mathit{Env} \end{pmatrix} & (a) \\ \doteq \mathit{env}^*(\{\mathit{bind}(x_1', \mathit{lam}(v', s'))\} \cup \mathsf{VCh}(z, x', k_1) \cup \mathit{Env}') & (b) \end{array} \quad (3)$$

Now we can solve the last equation from (3) by setting $\mathit{Env} \doteq (b)$ and $\mathit{Env}' \doteq (a)$ where in the equations (a) and (b) the environment variables Env and Env' are replaced by Env''. Now the system is in solved form. Applying the resulting unifier to one term of the original problem (1) yields:

$$\ldots \mathit{let}(\mathit{env}^* \left(\{\mathit{bind}(x_1', \mathit{lam}(v', s'))\} \cup \ldots \mathsf{VCh}(x_1, x_1', m_1) \cup \ldots\right), \ldots)$$

Instantiating m_1 with 1 and back-translating the term into LR results in (`letrec` $x_1' = \lambda v'.s', \ldots, x_1' = x_1$ `in` $\ldots$); a expression that is syntactically not admissible because the variable x_1' occurs twice at a binder position. Hence in the context of the specific unification problems (initial problems) we want to solve, case 1) of the rule **U-Chain** can be simplified to

$$\begin{array}{l} \{l_1 \doteq l{+}l_1',\ l_2 \doteq l{+}l_2',\ x_1 \doteq x_2, \\ \quad \mathit{env}^*(\mathsf{Ch}(z, y_1, l_1') \cup M_1 \cup r_1) \doteq \mathit{env}^*(\mathsf{Ch}(z, y_2, l_2') \cup M_2 \cup r_2)\} \cup P. \end{array}$$

I.e. two chains are equated beginning from their starting point up to some point from where they are disjoint. Here the justification is that initial problems have an additional property: bindings in variable chains are equipped with a (strict partial) order (like a linear list) with a least element called anchor-binding. In the case of VCh-constructs these bindings are always of the form $\mathit{bind}(x, v)$ where v is either an abstraction or a constructor application. The partial order in conjunction with the syntactical correctness criterion ensures the following: If some bindings of two chains are equated then all chain bindings that are smaller are also equated, until one anchor-binding is reached. The equation between such an anchor-binding and a non-anchor chain-binding (e.g. $\mathit{bind}(y, A(\mathit{var}(z))))$ can never hold, i.e. $\mathit{bind}(x, v) \doteq \mathit{bind}(y, A(\mathit{var}(z)))$ has no solution. Therefore our algorithm avoids the derivation of such unsolvable equations by equating initial parts of variable chains starting from their anchor-bindings. This strategy is crucial for the termination of the algorithm but it is only complete for initial problems (where each variable chain comes equipped with an anchor). Together with some additional unification rules and conditions that control the application of rules, termination of the algorithm can be assured.

One of those other rules, also concerned with solving equations with binding chains, is the rule **Dec-Chain** from Fig. 2. It covers the cases where a non-chain binding s_1 is equated with a chain binding. The possibilities are: 1) The chain consists only of one binding which is equated with s_1, or 2) the first binding of the chain is equated with binding s_1, or 3) the last chain binding is equated with s_1, or 4) a binding from the middle of the chain is equated with s_1 and the original chain is split around this externalized binding. All of these cases require that some of the internal chain variables (context and BV-sorted) are made explicit. These variables are always chosen as fresh (i.e. not occurring anywhere else in the unification problem).

Note that syntactical correctness and the distinct variable convention of the re-translated terms are enforced by the unification rules and by failure rules, and that without these precautions our rule-based unification algorithm would not terminate.

3 Overview of the Algorithm and Results

The unification algorithm for computing the overlaps in LR is applied to (initial) unification problems of the form $\{S[l_{T,i}] \doteq l_{no,j}\}$ where $l_{T,i}$ and $l_{no,j}$ are encoded left hand sides of LR reduction rules. Initial problems are restricted: They are linear in the variables and context variables, with the exception of variables of sort *Bind*, which is an empty sort. The occurrence of chains in initial problems is also restricted. These restrictions stem from the syntactical form of the reduction rules and the transformations of the LR calculus.

The unification rules of our algorithm consist of (i) rules from [5] that are adapted to the extended signature, and (ii) rules for dealing with equations $env^*(\ldots) \doteq env^*(\ldots)$, where both sides contain binding chains.
The following holds:

- The (nondeterministic) algorithm terminates on initial unification problems.
- The algorithm is sound and complete on initial unification problems, under the sensible restriction that only solutions are permitted that lead to syntactically correct expressions after translating them back to LR.
- The result of all nondeterministic executions is a finite set of final representations. These can be re-translated and lead to a finite set of overlaps of reduction rules and transformations.

4 Conclusion and Further Work

We devised an algorithm that computes complete sets of forks for the calculus *LR* from [7]. Therefore we first encode left hand sides of reduction rules into a term representation and use it to generate initial unification problems that describe all overlaps. Then we solve those unification problems using the sketched unification algorithm. After these steps we eventually instantiate the unification problems

that describe the forks with the computed solutions and translate them back to yield all forks in the LR calculus.

We plan to implement the computation and thus the verification of most of the diagrams of LR as presented in [7]. The core will be the unification algorithm as sketched above. This requires in addition closing the critical pairs using the normal-order reduction.

Bibliography

1. Dantsin, E., Voronkov, A.: A nondeterministic polynomial-time unification algorithm for bags, sets and trees. In: FoSSaCS. pp. 180–196 (1999)
2. Dovier, A., Pontelli, E., Rossi, G.: Set unification. TPLP 6(6), 645–701 (2006)
3. Hermann, M.: On the relation between primitive recursion, schematization and divergence. In: ALP. pp. 115–127 (1992)
4. Hermann, M., Galbavý, R.: Unification of infinite sets of terms schematized by primal grammars. Theor. Comput. Sci. 176(1–2), 111–158 (1997)
5. Rau, C., Schmidt-Schauß, M.: Towards correctness of program transformations through unification and critical pair computation. In: UNIF 2010. pp. 39–54. EPTCS (2010)
6. Salzer, G.: The unification of infinite sets of terms and its applications. In: LPAR 1992. LNCS, vol. 624, pp. 409–420 (1992)
7. Schmidt-Schauß, M., Schütz, M., Sabel, D.: Safety of Nöcker's strictness analysis. J. Funct. Programming 18(04), 503–551 (2008)

Higher-Order Unification for the $\lambda_{\alpha\nu}$ calculus

Ben Kavanagh and James Cheney

University of Edinburgh

Abstract. In this paper we propose an equational theory of a lambda calculus with names, name abstraction, and restriction, and a derived unification algorithm for this theory. Our calculus is very closely related to the calculus of Pitts 2011 [3]. We restrict the calculus presented there to obtain soundness of additional equalities, allowing us to define a unification algorithm very similar to the one given for the lambda calculus. We discuss the properties of this algorithm and give examples of its utility in meta-programming.

1 Introduction

Unification, the problem of finding a substitution under which a set of equations (i.e pairs or terms) are valid, is an essential tool used in automated reasoning, interactive theorem proving, and logic programming. Nominal terms, initially developed by Gabbay and Pitts in [1] offer an elegant way to represent alpha-equivalence classes. Unification for nominal terms was investigated by Urban et al. [5], and has been employed in nominal logic programming and rewriting.

More recently, Pitts [3] has developed the a calculus which extends the λ-calculus with both name-abstraction $\alpha a.e$ and a name-restriction operator $\nu a.e$. The name abstraction $\alpha a.e$, which is a binding form replaces $< a > e$ in nominal terms which is not. Although both represent equivalence classes, and both have elimination forms, called concretions, which instantiate the equivalence class at a particular representative, concretion over the new form $\alpha a.e$ is a total function and therefore there is an equational theory. We present a higher order unification algorithm over a variant of this calculus We aim to use this calculus to define a higher order logic, which may then be used to define a higher-order logic programming language with built in support for alpha equivalence classes.

Our calculus, $\lambda_{\alpha\nu}$, is based on that of Pitts [3] and has a similar denotation. The unification problem for terms in Pitts calculus are complicated by the name-restriction operation, which motivate changes. The two main differences are the removal of computational operators that inspect names (in particular '=') and the addition of two additional equations allowing us to move restriction operators up and down through terms. These allow us to decompose applications terms to simplify a unification problem. Our main results are an eta-long, permutation free, normal form over the additional equations, proofs that the equations are sound, and a sound constraint rewriting system to solve unification problems.

To illustrate our unification algorithm we will examine two example problems based on the lambda calculus. Imagine as part of semantic definition of the language we have the following rules.

$$\frac{}{(\lambda x.e)e' \longrightarrow e[e'/a]} \quad \frac{e \longrightarrow e'}{C[e] \longrightarrow C[e']}$$

which would correspond in our logic to rules constructed from terms in the $\lambda_{\alpha\nu}$ calculus as follows.

$$\frac{}{app(lam(\alpha a.E@a), E') \longrightarrow E[E'/a]} \quad \frac{E \longrightarrow E'}{CE \longrightarrow CE'}$$

The first rule is just beta reduction, and the second is a common rule to express congruence, usually seen in call-by-name calculi, where C represents a lambda term with a single hole. To reduce the term $t = app(lam(\alpha b.b), 0)$, i.e. $(\lambda b.b)0$ by calling a query `-? t --> ?V` we must unify t with $app(lam(\alpha a.E@a), E')$. A second example that uses the higher order properties is the reduction of $t' = lam(\alpha a.app(lam(\alpha b.e), e'))$, i.e. $\lambda a.(\lambda b.e)e'$ where we must unify t' with CE

2 The $\lambda_{\alpha\nu}$ calculus

Let $\mathbb{A}$ be a countably infinite set of names with decidable equality and inequality. Let $a \in \mathbb{A}$, then the terms of our calculus are as follows:

$$e ::= x \mid a \mid c \mid \lambda x.e \mid e_1\, e_2 \mid \alpha a.e \mid e@a \mid (a \wr b)e \mid \nu a.e$$

The forms for variables, constants, lambda-abstractions and applications, are standard. The purpose of name-abstraction, $\alpha a.e$, is to represent equivalence classes. Our equational theory for this form will not perform substitution (i.e. computation) but will only permute fresh names. Concretions, $e@a$, represent the extraction of a particular representative from the alpha equivalence class. We can understand concretion as a limited form of substitution where we can substitute only another name, implemented using permutation. The calculus includes terms to represent the pair-swapping permutations $(a \wr b)e$, since permuations are the basis of the equivalence classes and are thus required to give them an equational theory. Lastly we have the name-restriction operation $\nu a.e$ which is required to give a purely equational theory for concretion.

Motivation, background, and detailed semantics for this calculus can be found in Pitts [3]. The main two main differences between our calculus and the one presnted there are as follows: firstly we remove the pairs and booleans, secondly we use concretion, an equivalent form of name abstraction elimination, instead of the let form. We do this to have better properties for unification, i.e to be easier to adapt an existing higher order unification algorithm.

Types are defined as below where S is a metavariable that stands for type constants such as bool, int, etc.:

$$\tau \in T ::= S \mid \mathsf{Name} \mid \tau_1 \rightarrow \tau_2 \mid \mathsf{Name} \xrightarrow{\alpha} \tau$$

Here, type constants and function types are standard. We introduce a name type for names and a name arrow type for name-abstractions. The typing rules are given in Figure 1. In particular, note that both swapping and name-restriction preserve the type of their expression argument. Below when we discuss terms with arity of 2 or greater we use types of the form $e : \tau_1 \xrightarrow{\star_1} \tau_2 \xrightarrow{\star_2} \dots \tau_n \xrightarrow{\star_n} \gamma$ where γ is a metavariable indicating a base type. We also use the notation $e \star_1 e'_1 \dots \star_n e'_n$ where $\star_i$ is either @ or ' ' (i.e. function application),

In figure 1 we also summarize the denotational semantics of $\lambda_{\alpha\nu}$, based on [3]. The semantics employs constructions in the category $\mathcal{R}es$ of *nominal sets* with a swapping operator $(a\ b)x$, a finite support property, and a restriction operator $a \backslash x$. Space limits preclude a complete discussion of the semantics, however it is largely the same as in [3]. One difference is that we require the interpretation of terms of type $\tau \to \tau'$ to be an element of the exponential in $\mathcal{R}es$, $[\![\tau']\!]_\tau^{[\![\tau]\!]_\tau}$, which consists of functions with finite support that satisfy the equation

$$a \backslash (f\ x) = a \backslash f\ a \backslash x.$$

This restriction will give us a stronger equational theory than in [3] and will give us a simpler unification procedure. Note that the restriction rules out, for example, built-in functions such as name-equality $a = b$. For our semantics to be well defined we need to prove that the semantics for each typed term has this property. In particular we need the following lemma.

Lemma 2.1. *For all* $\Gamma, e, \tau,\ \rho \in [\![\Gamma]\!]_\tau,\ \Gamma \vdash e : \tau \implies [\![e]\!]_e \in [\![\tau]\!]_\tau$

We will also require the following property, which will allow us to push a ν operator below a concretion.

Lemma 2.2. *For all* $a, a', \Gamma, e, \tau, \rho \in [\![\Gamma]\!]_\tau, a \neq a' \implies a \backslash (([\![e]\!]_\tau \rho)@a') = (a \backslash [\![e]\!]_\tau \rho)@a'$

In the bottom of Figure 1 we give the equational laws of our variant of $\lambda_{\alpha\nu}$. These include standard $\beta\eta$ laws and an analogous β law for name-abstraction (1), alpha-equivalence laws (2), the four standard laws for name-restriction (3), the additional name-restriction laws for our calculus (4), and the rules for permutation (5). We assume the usual rules stating that the equality is a congruence and an equivalence relation. The rules given are shorthand since we are in a typed theory. Equality is a quaternary relation $\Gamma \vdash e = e' : \tau$ that can be read as saying in context Γ both e and e' are well-typed with type τ and are equal. We elide this notation in the rules given for simplicity.

Theorem 2.3 (Soundness). $\Gamma \vdash e = e' : \tau \implies \forall \rho \in [\![\Gamma]\!]_\tau, [\![e]\!]_e \rho = [\![e']\!]_e \rho$

Proof. straightforward proof by induction following the approach in Pitts [3] using Lemma 2.1 and Lemma 2.2 for the new rules for ν.

As in previous approaches to higher-order unification we will require canonical normal forms that allow us to compare terms up to our notion of equality.

Well-Typed Terms

$$\frac{(x:\tau)\in\Gamma}{\Gamma\vdash x:\tau}\quad \frac{f:\tau\in\Sigma}{\Gamma\vdash f:\tau}\quad \frac{a\in\mathbb{A}}{\Gamma\vdash a:\mathsf{Name}}\quad \frac{\Gamma,x:\tau_1\vdash e:\tau_2}{\Gamma\vdash \lambda x.e:\tau_1\to\tau_2}\quad \frac{\Gamma\vdash e:\tau_1\to\tau_2\ \ \Gamma\vdash e':\tau_1}{\Gamma\vdash e\,e':\tau_2}$$

$$\frac{\Gamma\vdash e:\tau\ \ a\in\mathbb{A}}{\Gamma\vdash \alpha a.e:\mathsf{Name}\xrightarrow{\alpha}\tau}\quad \frac{\Gamma\vdash e:\mathsf{Name}\xrightarrow{\alpha}\tau\ \ a\in\mathbb{A}}{\Gamma\vdash e@a:\tau}\quad \frac{\Gamma\vdash e:\tau\ \ a,b\in\mathbb{A}}{\Gamma\vdash (a\wr b)e:\tau}\quad \frac{\Gamma\vdash e:\tau\ \ a\in\mathbb{A}}{\Gamma\vdash \nu a.e:\tau}$$

Denotations

$$\begin{aligned}
[\![S]\!]_\tau &= [\![S]\!]_\Sigma\\
[\![\mathsf{Name}]\!]_\tau &= \mathbb{A}+1\\
[\![\tau_1\to\tau_2]\!]_\tau &= [\![\tau_2]\!]_\tau^{[\![\tau_1]\!]_\tau} \text{ in } \mathcal{R}es\\
[\![\mathsf{Name}\xrightarrow{\alpha}\tau]\!]_\tau &= [\mathbb{A}][\![\tau]\!]_\tau
\end{aligned}$$

$$\begin{aligned}
[\![x]\!]_e\rho &= \rho x\\
[\![a]\!]_e\rho &= Just(a)\\
[\![\lambda x.\tau_1 e]\!]_e\rho &= \lambda v:[\![\tau_1]\!]_\tau.[\![e]\!]_e((\rho[x\mapsto d]))\\
[\![e\,e']\!]_e\rho &= [\![e]\!]_e\rho([\![e']\!]_e\rho)\\
[\![\alpha a.e]\!]_e\rho &= a\backslash\langle a\rangle([\![e]\!]_e\rho) \quad \text{where } a\#\rho\\
[\![e@a]\!]_e\rho &= a'\backslash(a'\ a)[\![e]\!]_e\rho@a' \quad \text{where } a'\#\{\rho,[\![e]\!]_e\rho\}\\
[\![(a\wr b)e]\!]_e\rho &= (a\,b)\cdot[\![e]\!]_e\rho\\
[\![\nu a.e]\!]_e\rho &= a\backslash[\![e]\!]_e\rho \quad \text{where } a\#\rho
\end{aligned}$$

Equational Theory

$$\frac{}{(\lambda x.e)\,e' =_{\beta\eta\nu} e[e'/x]}\quad \frac{}{(\alpha a.e)@a' =_{\beta\eta\nu} \nu a.(a\wr a')e}\quad \frac{x\notin fv(e)}{\lambda x.e x =_{\beta\eta\nu} e} \quad (1)$$

$$\frac{x\notin fv(e_2),\ \ e_1 =_{\beta\eta\nu} e_2[x/y]}{\lambda x.e_1 =_{\beta\eta\nu} \lambda y.e_2}\quad \frac{a\notin fn(e),\ \ e =_{\beta\eta\nu} (a\wr a')e',\ \ \Delta\in\{\nu,\alpha\}}{\Delta a.e =_{\beta\eta\nu} \Delta a'.e'} \quad (2)$$

$$\frac{a\notin fn(e)}{\nu a.e =_{\beta\eta\nu} e}\quad \frac{}{\nu a.\nu a'.e =_{\beta\eta\nu} \nu a'.\nu a.e}\quad \frac{}{\nu a.\lambda x.e =_{\beta\eta\nu} \lambda x.\nu a.e}\quad \frac{a\neq a'}{\nu a.\alpha a'.e =_{\beta\eta\nu} \alpha a'.\nu a.e} \quad (3)$$

$$\frac{}{\nu a.(e\,e') =_{\beta\eta\nu} (\nu a.e)(\nu a.e')}\quad \frac{a\neq a'}{\nu a.(e@a') =_{\beta\eta\nu} (\nu a.e)@a'} \quad (4)$$

$$\frac{}{(a\wr b)(a') =_{\beta\eta\nu} (a\,b)a'}\quad \frac{c\in\Sigma}{(a\wr b)(c) =_{\beta\eta\nu} c} \quad (5)$$

$$\frac{}{(a\wr b)\lambda x.e =_{\beta\eta\nu} \lambda x.(a\wr b)(e[(a\wr b)x/x])}\quad \frac{\Delta\in\{\nu,\alpha\}}{(a\wr b)\Delta a'.e =_{\beta\eta\nu} \Delta(a\,b)a'.(a\wr b)e}$$

$$\frac{}{(a\wr b)(e_1\,e_2) =_{\beta\eta\nu} ((a\wr b)e_1)((a\wr b)e_2)}\quad \frac{}{(a\wr b)(e@a') =_{\beta\eta\nu} (a\wr b)e@(a\,b)a'}$$

Fig. 1. Well-typed terms, denotation, and equational theory for $\lambda_{\alpha\nu}$

As with β and η equations we take the right hand sides of the ν equations as reduced. We then consider η-long $\beta\nu$-reduced normal and neutral forms defined as :

$$n \in \mathrm{Nf} ::= \lambda x.n \mid \alpha a.n \mid u:\gamma \qquad \text{(where } \gamma \text{ a base type)}$$
$$u \in \mathrm{Ne} ::= a \mid Anon \mid \nu\overline{a}.\pi x \mid un \mid u@a \mid \nu a.u@a$$

where π stands for an arbitrary composition of swappings, equivalent to a finite name permutation, that is, a permutation of names where the set of names not mapped to themselves $\{a \mid \pi(a) \neq a\}$ is a finite set.

To normalise we first we first η-expand according to types, and then apply β-reductions and rules for pushing swappings and ν down into abstractions, applications, and concretions. All swappings can be pushed down to variables x, yielding a permutation π.

Name-restrictions can be pushed down until the argument is either the restricted name, a permuted variable, or a concretion of the same name. There are three neutral terms that describe where ν binders can exist in normal forms. One term is the *Anon* term described in Pitts [3]. The remaining two are a concretion term with a ν and a vector of ν quantifiers surrounding a suspension. We call this last term a $\nu-$suspension.

We define equality over the normal forms $=_{nf}$ as the congruence relation generated by alpha equivalence rules identical to those for $=_{\beta\eta\nu}$ and the additional rule below.

$$\frac{(\overline{a}, \pi) \approx_{\nu\pi} (\overline{b}, \pi')}{\nu\overline{a}.\pi =_{nf} \nu\overline{a}.\pi}$$

This rule says that two ν-suspensions are equal if and only if the number of ν quantifiers are equal and the permutations π, π', are equal up to re-labeling of the ν-quantified names. We use the relation $\approx_{\nu\pi}$ to represent these properties, that is,

$$(\overline{a}, \pi) \approx_{\nu\pi} (\overline{b}, \pi') ::= \left| \overline{a} \right| = \left| \overline{b} \right| \wedge \exists \pi''.\ dom(\pi'') \subseteq \overline{b} \wedge \pi = \pi' \cdot \pi''$$

This relation can be computed straightforwardly by computing the cycle graph of the two permutations and verifying that their cycles are equal on the vertices with names not bound by the ν quantifier. This comparison can be computed in linear time. For brevity we omit the algorithm. There is a trivial injection of the normal forms into $T_{\lambda_{\alpha\nu}}$. We denote this as $\ulcorner n \urcorner$.

3 Unification

Definition 3.1. *A problem P is a sequence of pairs of $\lambda_{\alpha\nu}$-terms in normal form*
$\{\langle n_1, n'_1\rangle, \ldots, \langle n_n, n'_n\rangle\}$. *A substitution is a total function $\sigma : X \to Nf$ with finite domain*

$\{x \mid \sigma(x) \neq x\}$. A unifier for problem P is a substitution σ, such that for each pair $\langle e_i, e_i' \rangle$ in P, $\sigma(e_i)$ and $\sigma(e_i')$ are convertible in the $\lambda_{\alpha\nu}$ calculus, i.e. $\sigma(e_i) =_{\beta\eta\nu} \sigma(e_i')$. The set of unifiers for a problem P is denoted $U(P)$. A problem P is solved for variable X if $P = \langle X, e \rangle \cup P'$ and X does not occur in P'. A problem P is solved if P is solved for all free variables occuring in P. A problem P is pre-solved if every variable in P that is not solved occurs only in flexible-flexible pairs.

The standard definitions for substitutions of 'more general than' ($\leq$), composition ($\circ$), union ($\cup$) apply.

We present a pre-unification algorithm as a rewrite system, an approach initially developed by Martelli and Montanari [2]. Our rules are largely based on the rules used by Gallier and Snyder [4]. The main difference is that due to the presence of restrictions we cannot use a multiple application shorthand (e.g. $ft_1 \dots t_n$) in decomposition but must instead decompose binary function and name application. Also since we give a pre-unification algorithm we can only use it for unification if all types are inhabited. The imitation and projection rule have the same basic form as that in [4] and require a term form that lifts all ν quantifiers above the head of the term. This term form is called unification-normal form. To produce this from the normal form we simply bubble up all ν-quantifiers in suspensions and name applications to the position above the head term. We describe the unification normal form below.

$$\begin{aligned} n_u \in \mathrm{Nf}_u &::= \lambda x.n_u \mid \alpha a.n_u \mid \nu\overline{a}.u_u \\ u_u \in \mathrm{Ne}_u &::= a \mid Anon \mid \pi x \mid u_u\, n_u \mid u_u@a \end{aligned}$$

Thus the terms are of the form $\Delta_1 p_1 \dots \Delta_k p_k.\nu\overline{a}.h \star_1 t_1 \dots \star_n t_n$ where Δ_i is either λ or α and and h is a meta-variable that denotes the head of normal terms, which is either a constant or a variable. As in the case for normal terms we represent the injection of this form into $T_{\lambda_{\alpha\nu}}$ as $\ulcorner n_u \urcorner$. Below we use bound elements p, q, and r to represent either a name or a variable. As in the classic literature on higher order unification we partition terms into *flexible* terms, i.e. those where the head h is a free variable, and *rigid* terms, i.e. those where the head h is a constant or a bound variable.

Lemma 3.2. $\forall n \in \mathit{Nf}.\exists n_u \in \mathit{Nf}_u.\ulcorner n \urcorner =_{\beta\eta\nu} \ulcorner n_u \urcorner$

Proof. simple induction on neutral form construction.

We now give a set of sound and complete transformations to rewrite problems to a solved form when a unifier exists. From the solved problem a unifier for the original problem can be constructed. Some of the transformations use the normal form, in particular the identity rule and the decomposition rule. The two rules for imitation and projection use the unification-normal form. Lemma 3.2 allows to switch between forms. To make the rules more readable we write the binders and arguments in vector form thus we write $\Delta_1 p_1 \dots \Delta_k p_k.\nu\overline{a}.h \star_1 t_1 \dots \star_n t_n$ as $\overline{\Delta p}_k.\nu\overline{a}.h\overline{\star t}_n$.

Definition 3.3. *(Set of transformations $\mathcal{HT}_{\alpha\nu}$) We define the set of transformations $\mathcal{HT}_{\alpha\nu}$ as the rewriting system on unification problems defined by the rules below.*

We may eliminate pairs of identical terms.

$$\{\langle e, e'\rangle\} \cup P \Longrightarrow P \qquad \text{if } \mathbf{nf}(e) =_{nf} \mathbf{nf}(e'). \tag{1}$$

If $\mathbf{nf}(e) = \overline{\Delta p}_k.F\overline{\star p}_k$, i.e. the eta-long form of a variable, we can substitute the variable to eliminate the variable from all other constraints. Here $P\downarrow$ means converting all terms in P to their normal form.

$$\{\langle e, e'\rangle\} \cup P \Longrightarrow \{\langle F, e'\rangle\} \cup \sigma(P)\downarrow \qquad \text{where } \sigma = [e'/F], F \notin fv(e) \tag{2}$$

If $\mathbf{head}(e) = \mathbf{head}(e') = h$, a rigid head, and $a' \notin fn(\{e, e'\})$, then we may decompose the constraint using the following three rules

$$\begin{aligned}&\{\langle \overline{\Delta p}_k.e_1 \star t, \overline{\Delta p}_k.e_1' \star t'\rangle\} \cup P \\ &\qquad \Longrightarrow \{\langle \overline{\Delta p}_k.e_1, \overline{\Delta p}_k.e_1'\rangle, \langle \overline{\Delta p}_k.t, \overline{\Delta p}_k.t'\rangle\} \cup P\end{aligned} \tag{3a}$$

$$\begin{aligned}&\{\langle \overline{\Delta p}_k.\nu a.e@a, \overline{\Delta p}_k.\nu b.e'@b\rangle\} \cup P \\ &\qquad \Longrightarrow \{\langle \overline{\Delta p}_k.\nu a'.(a \wr a')e, \overline{\Delta p}_k.\nu a'.(b \wr a')e'\rangle\} \cup P\end{aligned} \tag{3b}$$

Let $e_1 = \overline{\Delta p}_k.\nu\overline{a}.F\overline{\star t}_n$, $e_2 = \overline{\Delta p}_k.\nu\overline{b}.h\overline{\star t'}_m$, h a constant of type $\tau = \tau_1 \xrightarrow{\star_1} \ldots \tau_m \xrightarrow{\star_m} \gamma$, and $\overline{b'} = \overline{b} \cap \{t' \mid \exists b \in Name.\star_i\, t_i' = @b\}$ the top level name applications in e_2 with ν-quantified names, then

$$\begin{aligned}&\{\langle e_1, e_2\rangle\} \cup P \\ &\qquad \Longrightarrow \{\langle F, \overline{\Delta q}_n.\nu\overline{b''}.h\overline{\star\hat{H}}_m\rangle\} \cup \{\langle e_1, e_2\rangle\} \cup P\end{aligned} \tag{4}$$

where $\overline{b''}$ is a vector of names fresh for $q_i \ldots q_n$ with length $(\,|\overline{b'}| - |\overline{a}|\,) \leq |\overline{b''}| \leq |\overline{b}|$, and $\hat{H}_i$ as follows where $nArgs$ collects all names that are α-bound in $\overline{\Delta q}_n$.

$$\hat{H}_i = \begin{cases} a \in (nArgs(\overline{\Delta q}_n) \cup \overline{b''}) & \text{if } \exists b \in Name.\star_i\, t_i' = @b \\ \overline{\Delta r}_{|\tau_i|}.H_i\overline{\star q}_n\overline{@b''}\,\overline{\star r}_{|\tau_i|} & \text{otherwise}\end{cases}$$

with e, e' as in previous rule, if $t_i : \tau_1' \xrightarrow{\star_1} \ldots \tau_{m_i}' \xrightarrow{\star_m} \gamma_i$,

$$\begin{aligned}&\{\langle e_1, e_2\rangle\} \cup P \\ &\qquad \Longrightarrow \{\langle F, \overline{\Delta q}_n.\nu\overline{b'}.q_i\overline{\star\hat{H}}_{m_i}\rangle\} \cup \{\langle e_1, e_2\rangle\} \cup P\end{aligned} \tag{5}$$

where $\overline{b''}$ as before and

$$\hat{H}_i = \begin{cases} a \in (nArgs(\overline{q}_n) \cup \overline{b''}) & \text{if } \tau_i' = Name \wedge \xrightarrow{\star_i} = \xrightarrow{\alpha} \\ a \in (nArgs(\overline{\Delta q}_n) \cup \overline{b''}) & \text{if } \exists b \in Name.\star_i\, t_i' = @b \\ \overline{\Delta r}_{|\tau_i'|}.H_i\overline{\star q}_n\overline{@b'}\,\overline{\star r}_{|\tau_i'|} & \text{otherwise}\end{cases}$$

3.1 Correctness of $\mathcal{HT}_{\alpha\nu}$

First we show soundness of the transformations.

Lemma 3.4. *if $P \Longrightarrow P'$ by rules (1), (2), or (3) then $U(P') = U(P)$.*

Lemma 3.5. *if $P \Longrightarrow P'$ by rules (4) or (5) then $U(P') \subseteq U(P)$.*

Theorem 3.6 (Soundness). *if $P \Longrightarrow^* P'$, and P' is in pre-solved form then $\sigma \in U(P)$ where $\sigma = \{\langle X_i, t_{X_i}\rangle \mid X_i$ solved in $P'\} \cup \{\langle Y_i, \xi_{Y_i}\rangle \mid \underline{Y_i} \in res(P')\}$ with $res(P')$ the variables remaining in flex-flex pairs and $\xi_{X_i} = \overline{\Delta p_n}.k_\gamma$ where $X_i : \tau_1 \xrightarrow{\star_1} \ldots \tau_n \xrightarrow{\star_n} \gamma$ and k_γ a special constant for type γ.*

Proof. By induction on the length of the rewrite sequence using Lemma 3.4 and Lemma 3.5.

We give a completeness result similar to that of Huet in that we do not solve for complete sets of unifiers but only guarantee we will find a substitution if one exists.

We first show that any term can be be decomposed using a partial binding and a substitution, where a partial binding is a binding $\langle F, t\rangle$ generated by either the imitation or projection rule.

Lemma 3.7. *Given a $\lambda_{\alpha\nu}$ term s in eta-long normal form, i.e. $s = \overline{\Delta p_k}.\nu\overline{a}.h\overline{\star t_m}$ with h a constant or bound variable, there exists a partial binding t and a substitution η such that $\eta(t) =_{\beta\eta\nu} s$.*

We then show that we can form an equivalent substitution which decomposes one of its pairs.

Lemma 3.8. *let $\sigma = [s/F] \cup \sigma'$ then for a unique variant of a partial binding t valid for F and s such that*

$$\begin{aligned}\sigma &= ([s/F] \cup \eta \cup \sigma')|_{D(\sigma)} \\ &= ([t/F] \circ \eta \cup \sigma')|_{D(\sigma)}\end{aligned}$$

Definition 3.9. *We define a rewrite system for pairs of substitutions and problems $\overset{\sigma}{\Longrightarrow}$ as follows*

$$\frac{P \Longrightarrow P' \text{ via rules } (1)(2)(3) \wedge \sigma \in U(P)}{\sigma, P \overset{\sigma}{\Longrightarrow} \sigma, P'}$$

$$\frac{\{\langle e_1, e_2\rangle\} \cup P \Longrightarrow \{\langle F, t\rangle\} \cup [t/F](\{\langle e_1, e_2\rangle\} \cup P) \;\wedge\; [s/F] \cup \sigma' \in U(\{\langle e_1, e_2\rangle\} \cup P)}{[s/F] \cup \sigma, (\{\langle e_1, e_2\rangle\} \cup P) \overset{\sigma}{\Longrightarrow} [s/F] \cup \eta \cup \sigma, \{\langle F, t\rangle, \langle e_1, e_2\rangle\} \cup P}$$

We now show that if the problem is not solved we can always make progress.

Lemma 3.10 (Progress). *If $\sigma \in U(P)$ for P is not in solved form then there exists σ', P' such that $\sigma, P \overset{\sigma}{\Longrightarrow} \sigma', P' \wedge \sigma' \in U(P') \wedge P \Longrightarrow P'$.*

Theorem 3.11 (Completeness). *if* $\exists\sigma \in U(P)$ *then there is a* P' *such that* $P \Longrightarrow^* P'$*, and* P' *is in pre-solved form.*

Proof. Let the complexity measure of a substitution-problem $\mu(\sigma, P) = \langle M, N\rangle$, where M is the sum of term size (# atomic terms) for the range of σ over variables that are not solved and N is the sum of the termsizes of all terms in P. All substitution transitions decrease complexity. Using well-founded induction over complexity and our progress lemma we can show that we will always reach a solved form.

We can now return to the two examples we gave in the introduction. The problems described there translate into the following two unification problems.

$$P_1 \stackrel{def}{=} \{\langle app(lam(\alpha b.b), 0),\ app(lam(\alpha a.E@a), E')\rangle\}$$
$$P_2 \stackrel{def}{=} \{\langle lam(\alpha a.app(lam(\alpha b.e), e')),\ CE\rangle\}$$

By applying our rewrite rules both problems can be rewritten to pre-solved forms with the substitutions given below. The derivations are outlined in figure 3.1 with some repetitive steps elided.

- P_1 succeeds with $\sigma = [E \mapsto \alpha a.a, E' \mapsto 0]$
- P_2 succeeds with $\sigma = [C \mapsto \lambda x.lam(\alpha a.x@a), E \mapsto \alpha x.app(lam(\alpha y.e))e']$

$$\begin{array}{lll}
P_1 & \Longrightarrow \{\langle (Lam(\alpha a.E@a)),\ (Lam(\alpha b.b))\rangle, \langle E', 0\rangle\} & \text{(decompose)} \\
 & \Longrightarrow \{\langle \alpha a.E@a,\ \alpha b.b\rangle, \langle E', 0\rangle\} & \text{(decompose)} \\
 & = \{\langle \alpha a.E@a,\ \alpha a.a\rangle, \langle E', 0\rangle\} & \alpha - \text{equality} \\
 & \Longrightarrow \{\langle E, \alpha a.a\rangle, \langle \alpha a.E@a,\ \alpha a.a\rangle, \langle E', 0\rangle\} & \text{(projection)} \\
 & \Longrightarrow \{\langle \alpha a.a, \alpha a.a\rangle, \langle E', 0\rangle\} & \text{(var-elim)} \\
 & \Longrightarrow \{E'\}0 & \text{(ident-elim)} \\
 & \Longrightarrow \{\} & \text{(var-elim)}
\end{array}$$

$$\begin{array}{lll}
P_2 & \Longrightarrow \{\langle C, \lambda x.lam(\alpha a.Hx@a)\rangle, \langle CE, lam(\alpha x.app(lam(\alpha y.e))e')\rangle\} & \text{(imitation)} \\
 & \Longrightarrow \{\langle lam(\alpha a.HE@a),\ lam(\alpha x.app(lam(\alpha y.e))e')\rangle\} & \text{(var-elim)} \\
 & \Longrightarrow \{\langle \alpha a.HE@a,\ \alpha x.app(lam(\alpha y.e))e'\rangle\} & \text{(decompose)} \\
 & \Longrightarrow \{\langle H, \lambda x.\alpha a.x@a\rangle, \langle \alpha a.HE@a,\ \alpha x.app(lam(\alpha y.e))e'\rangle\} & \text{(projection)} \\
 & \Longrightarrow \{\langle \alpha a.E@a,\ \alpha x.app(lam(\alpha y.e))e'\rangle\} & \text{(var-elim)} \\
 & \ldots\ldots & \text{(seq of imitate/decomp)} \\
 & \Longrightarrow \{\} &
\end{array}$$

4 Conclusions

This paper reports work in progress on combining higher-order nominal unification. To adapt known techniques such as Huet's algorithm, we have adapted

Pitts' $\lambda_{\alpha\nu}$ system to provide $\beta\eta$-normal forms that are amenable to the standard imitation, projection and decomposition steps, and have shown that these changes are compatible with normalization and with the semantics of Pitts' calculus. Implementation and full proofs of soundness and completeness are the subject of ongoing work.

Since it extends the lambda calculus, unification over this calculus is undecidable. Nevertheless we may expect that as with higher order unification there will be useful decidable fragments. Our original interest in investigating this calculus was to give a unification algorithm for nominal terms with context variables, which may be represented as linear second order terms in our proposed calculus.

Bibliography

1. Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, 1999. IEEE Computer Society Press.
2. Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4:258–282, April 1982.
3. A. M. Pitts. Structural recursion with locally scoped names. *Journal of Functional Programming*, 2011. To appear.
4. Wayne Snyder and Jean Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(1-2):101 – 140, 1989.
5. Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1-3):473–497, 2004.

Does Unification Help in Normalization?

Rakesh M. Verma and Wei Guo

University of Houston Department of Computer Science
4800 Calhoun Rd., Houston, TX 77004, USA
rmverma@cs.uh.edu
http://www.cs.uh.edu/~rmverma

Abstract. In this paper, we present a preprocessor for rules based on unification, which has the potential to enable faster search for potential redexes in normalization. We implement this idea in Laboratory for Rapid Rewriting (LRR) [2] and compare our method with ELAN [9] and Maude [8] using both favorable and unfavorable examples to demonstrate the performance.

1 Introduction

Fast rewriting is needed for equational programming, rewrite based formal verification methods, and symbolic computing systems. In any implementation of rewriting techniques efficiency is a critical issue [6]. The goals of this paper are to enhance the efficiency of the normalization by integrating a preprocessor for rules and to determine how much can unification help in future matching attempts in practice, especially when built-in operators such as arithmetic present complications. The immediate motivation is to effectively cut the time spent in traversing both the term and the rules in order to find a match. The preprocessor for rules utilizes the unification results obtained from a set of rules to facilitate matching. Our idea is applicable to any interpreter. Since we have been working on LRR , an interpreter for a rule-based programming language with an efficient history option, we use it as a platform to demonstrate our idea.

The rest of this paper is organized as follows. We first present some preliminaries including a brief introduction to LRR in Section 2. Then we discuss how unification helps in normalization in Section 3. The experimental results are presented in Section 4. In Section 5, we conclude the paper with some promising directions for future research.

2 Preliminaries

A *Term Rewriting System* is a set of rewriting rules, R, and a given term t_0. The objective is to compute a normal form of t_0, t_n. We denote the ith rule as $rule_i : lhs_i \Rightarrow rhs_i;$. We define that *the* i^{th} *step of the normalization* is a process that builds a new term t_i by applying $rule_j$ at a subterm of term t_{i-1}, in which $i \in \mathbb{N}, 0 < i \leq n$. We use $t_{i-1} \rightarrow_{(i,j)} t_i$ to denote the ith step of the normalization. Thus, the whole process of normalization can be denoted as a sequence,

$t_0 \rightarrow_{(1,j)} t_1, ..., t_i \rightarrow_{(i+1,j')} t_{i+1}, ..., t_{n-1} \rightarrow_{(n,j'')} t_n$. Terms $t_1, ..., t_i, ..., t_{n-1}$ are called *intermediate results.*

A *position* of a term t is a sequence of natural numbers that is used to identify the locations of subterms of t. The subterm of $t = f(s_1, \ldots, s_n)$ at position p, denoted $t|_p$, is defined recursively: $t|_\lambda = t$, where λ is the empty sequence, $t|_k = s_k$, and $t|_{k.l} = (t|_k)|_l$ for $1 \leq k \leq n$ and undefined otherwise [7].

LRR is one of the interpreters for rule based programming. The input of LRR is a file representing the rules R and a file representing the given term t_0. It consists of a term graph interpreter TGR, and a term graph rewriter storing the history of its reductions, called Smaran, based on the Congruence Closure based Normalization Approach (CCNA) [4]. Smaran constructs signatures representing the terms and equivalence classes consisting of equivalent signatures. Please consult [4, 5] for more details. TGR uses Term Graph Rewriting which has no class or signature.

An extension of almost linear unification (ALU). The objective of unification is, given two terms l, r, to find a substitution σ such that $\sigma(l)$ and $\sigma(r)$ are syntactically identical. The ALU algorithm uses Directed Acyclic Graphs (DAGs) as the data structures of the terms and requires variables to be shared, which reduces the complexity from exponential to almost linear (please see [1] for more details). We extend the concept of unification as follows. Under strict unification, a constant never unifies with a function having at least one child. But if a function can be evaluated during the normalization and the constant is one of the possible results, we consider that they "unify". For example, consider the Fibonacci function in appendix, either term $true$ or $false$ is the result of term $> (x, 1)$ and here we consider that $true$ and $false$ unify with $> (x, 1)$.

3 How Does ALU Help in Normalization

We find that many matches found in normalization happen between an instance $\sigma(r|_p)$ of a subterm $r|_p$ of a RHS r and an LHS l. This implies that the LHS unifies with this subterm of the RHS, since their variables are "effectively" disjoint. And if a subterm from a RHS can unify with a LHS, there is a great chance to find a match between the instance of the subterm and the LHS when the instance is built by the RHS. Before normalization starts, we add a preprocessor for rules which collects the unification results between LHS's and RHS's using ALU. In normalization procedure, we introduce a list, the ALU-list, to let the unification results help to find a match. In one step of normalization, instead of looking for a match by scanning all subterms of the term to be normalized and all the rules, our normalization procedure first looks for a match from the ALU-list. In Figure 1 below, in $t_{i-1} \rightarrow_{(i,j)} t_i$, a match is found between a subterm u of t_{i-1} and lhs_j. Then the subterm u is replaced by v, the instance of rhs_j, and we get t_i. Term v shares the same overall structure as rhs_j. If we know that a subterm $x = rhs_j|_p$ can unify with lhs_k, there is a great chance to find a match between term $w = v|_p$, the instance of term x, and lhs_k in the next step. In the $i + 1^{th}$

step, normalization can try the term w and lhs_k first. If a match is found, term w is replaced by the instance of rhs_k.

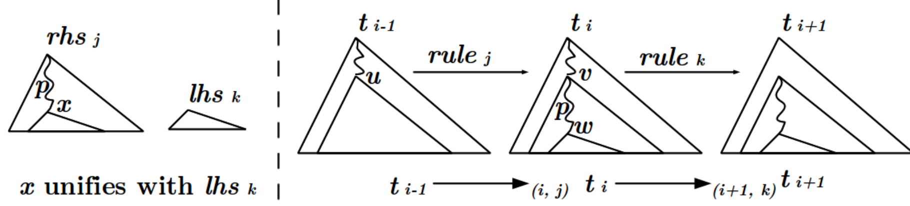

Fig. 1. Unification results can help in normalization

Actually, significant parts of all the intermediate results, $t_1, ..., t_i, ..., t_{n-1}$ and the normal form t_n are constructed from the RHS's and much of the overall structure of terms can be safely predicted from the RHS's (the exceptions are the variable substitutions and unexplored parts of the intermediate terms). If before normalization, we know the unification results between each subterm in every RHS and each LHS, we can use the results to find the matches in normalization efficiently. But not all the unification results lead to successful matches. In this case, and for normalizing t_0, LRR calls original Smaran or TGR to find matches.

A preprocessor for rules tries to unify every subterm in every RHS with every LHS and stores the successful results denoted by a list of pairs (C, P) before normalization. In Figure 1, subterm x from rhs_j unifies with lhs_k. We say that $rule_k$ is a *candidate* and for this example $C = k$. We define the position a *point*, which for this example would be stored in $P = p$. Storing term x in the node does not help normalization but storing P does because normalization needs to find w by following P from v. The preprocessor stores the indexes of the rule in C and uses a single linked list to store P which is essentially a sequence of natural numbers. There may be more than one pair for each RHS. For every RHS, the preprocessor uses a single linked list to store the nodes.

The ALU-list is a singly-linked list to store the information obtained from the unification results during the normalization. Each node in the list is a 3-tuple (i, c, s), where i indicates the ith step of normalization, c indicates a candidate, in which $c = C$, and s represents the term that is possible to match lhs_c, such as $w = v|_P$ in Figure 1. We use stack operations to implement a depth-first order in normalization. In one step, tuples obtained from the nodes of the RHS that is applied in this step are pushed into the ALU-list. In the next step, the ALU-list pops a tuple (i', c', s') and tries to match the term s' and $lhs_{c'}$. If they match, LRR continues normalization. If not, the ALU-list pops the next tuple. When the ALU-list is empty, normalization goes back to the original algorithm searching for new match. In LRR, normalization goes back to Smaran or TGR. In Figure 1, rhs_j has a node (k, p) in which $x = rhs_j|_p$. Normalization locates

term $w = v|_p$ and pushes the tuple (i, k, w) into the ALU-list. If the tuple is popped at the $i + 1^{th}$ step, $rule_k$ is applied at term w to form term t_{i+1}.

3.1 Optimizations

In order to improve the efficiency of integration of the preprocessor and the normalization procedure, we implemented the following optimizations.

Mutually exclusive detection is a method to cut unnecessary insertions into the ALU-list caused by the extension of ALU. For example, both terms *true* and *false* are considered as candidates for the term $> (x, 1)$. We add two nodes in unification. But only one tuple will succeed in matching. So, by evaluating the term $> (x, 1)$ before pushing a candidate into the ALU-list, normalization picks up only the "right" tuple.

Candidate elimination contains three ways to delete tuples from the ALU-list. *Same point elimination* is a method to cut unnecessary matching attempts. Tuples having the same value of i and same value of s apply at the same point of the same instance of the RHS. Once we find the first match from these tuples, which have the same value of s, the intermediate term probably will change in the next reduction step and the remaining tuples are deleted. *Descendants elimination* also cuts unnecessary matching attempts. If the parent succeeds in matching, the matching attempts for its children are unnecessary since the intermediate term probably will change. *Changed signature check* cuts unnecessary matching attempts only when the preprocessor works with Smaran. Smaran checks whether the unreduced signature of the class has changed since the tuple containing the class was added into the ALU-list. If yes, LRR deletes the tuple.

4 Experimental Results

The preprocessor for rules has still some room for improvement. A Linux version of LRR v3.0 and some examples can be downloaded from `http:/www.cs.uh.edu/~evangui`. LRR v3.0 provides: i) the original Smaran and TGR, ii) a preprocessor for rules with original Smaran and with original TGR. In the reference [2, 3, 4], several optimizations are discussed including structure sharing, and CCNA. We use Maude 2.6 32-bit version which can be found at `http://maude.cs.uiuc.edu/download`. We use ELAN interpreter 3.6g which can be found at `http://webloria.loria.fr/equipes/protheo/SOFTWARES/ELAN/manual/index-manual.html`.

Performance Results. We present the experimental results on nine benchmarks (rules can be found in [10] for lack of space) to illustrate the level of efficiency. LRR is implemented in C and runs on Linux. Normalization times are on a 2.67GHz Intel i5 560M Ubuntu 10.10 linux kernel 2.6.35-22 system with 8GB of memory using gcc compiler (v. 4.4.5) with optimization level 3. We are aware of the difficulties of comparing different software systems. Each benchmark for three systems uses exactly same algorithm. Rules in the benchmark are semantically identical. Syntactic differences are due to differences in the rule

specifications for the three interpreters. Table 1 shows the average results of 10 executions in seconds for nine benchmarks, which can be found at the URL given above. From Table 1, even though we find that Maude without memo is

Table 1. Experimental Results on normalization time

Benchmark	ELAN	Maude		LRR			
		w/o memo	w/ memo	Smaran	Smaran+ALU	TGR	TGR+ALU
binsort(1500)	164.2228	0.6936	463.6586	2.2301	2.6398	1.8197	1.7829
bintree(380)	0.1152	0.0044	0.0936	0.0160	0.0144	0.0116	0.0116
dfa(1363)	0.0016	0.0000	0.0008	0.0540	0.0540	0.0396	0.0424
fib(20)	1.4416	0.0272	0.0000	0.0000	0.0004	4.4851	4.5507
merge(20000)	17.8455	0.0404	70.2658	0.0460	0.0524	0.0300	0.0296
qsort(1800)	66.6180	1.1872	30.7426	10.0294	8.8518	3.4254	3.3874
rev(19900)	66.6304	0.0380	129.6359	0.0484	0.0548	0.0328	0.0332
rfrom(19996)	1.7005	0.0408	44.1588	0.0384	0.0408	0.0224	0.0220
sieve(10000)	169.6300	0.4900	29.6235	1.5889	1.7709	0.8277	0.8257

the fastest option in most benchmarks, Smaran and/or TGR are close. It is interesting to see that Smaran is not far behind even in examples that do not use history, despite saving the entire history of rule applications. ELAN interpreter runs slow in most cases. We are aware that the ELAN project focuses more on the compiler than the interpreter. Maude with memo runs faster for fib(20) and dfa but is much slower for the other benchmarks tested. The preprocessor does not completely beat TGR or Smaran. Apparently there is some inefficiency in the implementation of the preprocessor. We think we can improve it in the following ways. First, we plan to write a new function for matching since we have a great accuracy in prediction. The new function should explore the unification results deeper. The other, when the preprocessor cannot initiate a match, LRR should find the next match in a more efficient way. We plan to track the positions of variables in a RHS and direct LRR to try terms covered by the variables rather than traversing from the root. The preprocessor of rules runs slower than original methods in most examples, but it cuts the unnecessary matching attempts significantly. Although it does not yet control the normalization independently, the percentage of successful matches is relatively high.

Related work. We did an extensive search for rule-based programming interpreters using the papers [6, 11] and the Rewriting Page on the web, but we have been unable to find any interpreter that includes any such application of unification to speed up the matching process during normalization. Apart from Maude, in [11] a compiler for rules is described, but there is no comparable effort on speeding up normalization. The only other interpreter that we could find is CRSX [12], which does not include built-ins and could only handle a string of length 819 in the dfa example.

5 Discussion and Future Work

We presented a preprocessor for rules, a method to improve the efficiency of normalization. The preprocessor beats the earlier strategy in accurately finding the next match. We plan to implement the preprocessor in a more efficient way and try to use more information from unification to help speed up the normalization even more.

Acknowledgments. We want to thank S. Senanayake, J. Thigpen, and H. Shi for initial work on LRR, and Z. Liang for some examples.

Bibliography

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge Univ. Press, (1999)
2. Verma, R., Senanayake, S.:LR^2 : A Laboratory for Rapid Term Graph Rewriting. In: Proceedings of the 10th International Conference on Rewriting Techniques and Applications, pp. 252–255. (1999)
3. Verma, R.: Static Analysis Techniques for Equational Logic Programming. In: Proceedings of the 1st ACM SIGPLAN Workshop on Rule-based Programming. (2000)
4. Verma, R.: Smaran: A congruence-closure based system for equational computations. In: Proceedings of the 5th International Conference on Rewriting Techniques and Applications, pp. 457–461. (1999)
5. Shi, H.: Integrating associative and commutative matching in the LR^2 Laboratory for fast, efficient and practical rewriting techniques. University of Houston. (2000)
6. Hermann, M., Kirchner, C., Kirchner, H.: Implementations of term rewriting systems. The Computer Journal, 34(1), pp. 20–33. (1991)
7. Radcliffe, N., Verma, R.: Uniqueness of Normal Forms is Decidable for Shallow Term Rewrite Systems. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science. pp. 284–295. (2010)
8. Clavel, M., Eker, S., Lincoln, P., Meseguer, J.: Principles of Maude. Electronic Notes in Theoretical Computer Science. (1996)
9. Borovansky, P., Kirchner, C., Kirchner, H., Moreau, P.E., Vittek, M.:ELAN: A logical framework based on computational systems. In: Proceedings of the first international workshop on rewriting logic. Asilomar (1996).
10. Verma, R. and Guo, W.: Does Unification Help In Normalization? University of Houston Computer Science Department Technical Report, UH-CS-11-05, June 2011.
11. Vittek, M. :A Compiler for Nondeterministic Term Rewriting Systems. In: Proceedings 7th Conference on Rewriting Techniques and Applications. pp. 154-168 New Brunswick, New Jersey, USA (1996).
12. Klop, J.W., Oostrom, V.V., and Raamsdonk, F.V.: Combinatory Reduction Systems: Introduction and Survey, Theoretical Comp. Sci. 121, pp. 271-308 (1993).

Appendix

A Concrete Example

To illustrate the details, we use a concrete example in LRR which computes Fibonacci numbers (we use Fibo for short).

$$fib(x) \Rightarrow f(>(x,1),x) \tag{1}$$

$$f(true,x) \Rightarrow +(fib(-(x,1)),fib(-(x,2))) \tag{2}$$

$$f(false,x) \Rightarrow 1; \tag{3}$$

The normalization process using TGR is (under a depth-first left-most order) below:

$$\begin{aligned} fib(2) &\rightarrow_{(1,1)} f(true,2) \\ &\rightarrow_{(2,2)} +(fib(1),fib(0)) \\ &\rightarrow_{(3,1)} +(f(false,1),fib(0)) \\ &\rightarrow_{(4,3)} +(1,fib(0)) \\ &\rightarrow_{(5,1)} +(1,f(false,0)) \\ &\rightarrow_{(6,3)} 2 \end{aligned} \tag{4}$$

In the 1st step, LRR calls Smaran or TGR to initiate matching because the preprocessor has no information. $rule_1$ is picked up and $t_1 = f(true,2)$. Then, the preprocessor knows that the term $f(>(x,1),x)$ unifies with lhs_2:$f(true,x)$, and lhs_3:$f(false,x)$. It looks like LRR tries to match $f(true,2)$ with $f(true,x)$ and $f(false,x)$. But in Section 3.1, we show that LRR picks up only $f(true,x)$ while the original LRR attempts to match $f(true,2)$ with all 3 rules. LRR picks up $rule_2$ and gets $t_2 = +(fib(1),fib(0))$. The preprocessor still knows that the terms $fib(-(x,1))$ and $fib(-(x,2))$ unify with lhs_1:$fib(x)$. Under a depth-first left-most order, LRR starts from $fib(1)$ and succeeds in matching $fib(1)$ with $fib(x)$ while the original LRR traverses from the root of $+(fib(1),fib(0))$ searching for a match. LRR picks $rule_1$ and gets $t_3 = +(f(false,1),\ fib(0))$. After 6 steps, LRR stops at the normal form, 2.

In ALU, for each RHS, it is possible that some subterms unify with multiple rules. Thus, there may be more than one pair for each RHS. After LRR parses all the rules and before it starts the normalization, for each RHS, the preprocessor tries to unify every subterm with every LHS and stores the results for each RHS. In Fibo, $f(>(x,1),x)$ unifies with lhs_2:$f(true,x)$, lhs_3:$f(false,x)$; $fib(-(x,1))$ and $fib(-(x,2))$ unify with lhs_1:$fib(x)$. So, $rule_1$ has a list of two pairs $(2,\lambda)$, $(3,\lambda)$. $rule_2$ has a list of two pairs $(1,(1))$, $(1,(2))$.

In normalization, for the combination of Smaran and ALU, s in the 3-tuple (i,c,s) is the number of the class containing the term, indicating the unreduced signature of the class. For the combination of TGR and ALU, s is the term.

In Fibo, $rule_2$ is used in the 2nd step. So LRR (using TGR) gets two nodes from $rule_2$, $(1,(1))$, $(1,(2))$, creates two tuples $(2,1,fib(1))$ and $(2,1,fib(0))$, and pushes them into the ALU-list. In the 3rd step, $(2,1,fib(1))$ is at the top of the ALU-list. So LRR pops the tuple and tries to match $fib(1)$ with lhs_1. Since they match, LRR builds t_3 and evaluates the built-in operations. LRR creates 2 tuples $(3,2,f(false,1))$, $(3,3,f(false,1))$ but only pushes $(3,3,f(false,1))$ into the ALU-list because of mutually exclusive detection.

Asymmetric Unification: A New Unification Paradigm for Cryptographic Protocol Analysis

Serdar Erbatur[6], Santiago Escobar[1], Deepak Kapur[2], Zhiqiang Liu[3], Christopher Lynch[3], Catherine Meadows[4], José Meseguer[5], Paliath Narendran[6], and Ralf Sasse[5]

[1] DSIC-ELP, Universidad Politécnica de Valencia, Spain
sescobar@dsic.upv.es
[2] University of New Mexico, Albuquerque, NM, USA
kapur@cs.unm.edu
[3] Clarkson University, Potsdam, NY, USA
liuzh@clarkson.edu, clynch@clarkson.edu
[4] Naval Research Laboratory, Washington DC, USA
meadows@itd.nrl.navy.mil
[5] University of Illinois at Urbana-Champaign, USA
meseguer@illinois.edu, rsasse@illinois.edu
[6] University at Albany-SUNY, Albany, NY, USA
serdar.erbatur@gmail.com, dran@cs.albany.edu

Abstract. A new extension of equational unification, called *asymmetric unification*, is introduced. In asymmetric unification, the equational theory is divided into a set R of rewrite rules and a set E of equations. A substitution σ is an asymmetric unifier of a set of equations P iff for every $s = t \in P$, $s\sigma$ is equivalent to $t\sigma$ modulo $R \cup E$, and furthermore $t\sigma$ is in $E \backslash R$ normal form. This problem is at least as hard as the unification problem modulo $R \cup E$ and sometimes harder. The problem is motivated from cryptographic protocol analysis using unification techniques for handling equational properties of operators such as XOR.

1 Introduction

The problem we consider is inspired by our work on cryptographic protocol analysis in the Maude-NPA, where terms represent messages sent by a principal involved in a protocol. A query represents a possible secret to be learned by an intruder, and the tool explores symbolically whether there is a way back from the attack goal to an initial state. In the search, messages sent are unified with the message that a principal is expecting to receive. A key feature of the Maude-NPA is that it can handle equational theories [6], which represent properties of a cryptographic algorithm. For example, cryptographic algorithms involve exclusive OR (henceforth called XOR), so that the terms that represent messages are assumed to be associative and commutative, there is an identity element, and each term is its own inverse (nilpotent).

The fact that terms have equational properties means that the unification that is done to match sent messages with expected received messages must be

performed *modulo* the equational theory. Therefore, we have been building equational unification algorithms into the Maude-NPA [5]. But recently we have realized that more than equational unification is required.

Received messages use variables to indicate a part of the message that is unknown to the principal. For example, a principal may receive a message Y and check whether it is of the form $X + c$, where X is a variable, c is a constant, and $+$ is the XOR operator. This means that the principal expects to receive the XOR of something unknown with c. To check whether the principal accepts the message, we unify the sent message with $X + c$. If the principal receives the message $b + c$ then the principal will accept it. But what if an intruder sends the message a? Because of the self-cancellation properties of XOR, the principal will also accept that message, because $a = a + c + c$, so we use the substitution $[X \mapsto c + a]$.

We can account for both of these instances with the single unifier $[X \mapsto Y+c]$, thus giving a single solution. However, although this allows us to perform reachability analysis efficiently, it causes problems when we want to prune the search space; problems arise because search space pruning in Maude-NPA (as well as many other tools) depends upon syntactic, not equational, properties of terms. Suppose, for example, that c is a random nonce. Maude-NPA automatically discards any state in which an intruder learns a term containing a nonce that is not generated until a future state. Thus, if Maude-NPA encounters a state in which an honest principal generates the nonce c, and then searches backwards until it reaches a state in which the intruder learns $X + c$, it will discard that state. But suppose that if Maude-NPA had continued further in its backwards search it would have reached a state in which the substitution $[X \mapsto c + a]$ occurred. Then the subterm c would vanish and the state would be potentially reachable again, thus rendering the search algorithm incomplete.

This reliance upon syntactic checking is not unique to Maude-NPA, but appears in other unification-based tools, such as CPSA [1] and ProVerif [3]. CPSA constructs DAGs representing potential executions by using outgoing and incoming test pairs that are syntactically defined (e.g. they must contain the same nonce as a subterm). ProVerif includes the option of enforcing termination by substituting variables for certain terms whose depth is greater than a certain bound [1].

CPSA avoids these pitfalls by restricting itself to certain types of order-sorted equational theories, which guarantee that, under the appropriate circumstances, if a subterm appears at a certain position, then it continues to appear in that position even after substitution [7]. Instead, ProVerif avoids the problem by handling destructors (and the rewrite rules that describe their behavior) separately from constructors. Maude-NPA has taken a somewhat different approach, however. In that tool an equational theory is divided into $R \cup E$ such that R is a set of rewrite rules and E is a set of equations, and R is convergent modulo E and has the *finite variant property* with respect to E [2]. This has the consequence that for any term m, there is a finite set of substitutions Σ, such that, for any substitution θ, there is a $\sigma \in \Sigma$ and a substitution τ such that $m\theta{\downarrow} =_E ((m\sigma){\downarrow})\tau$.

In Maude-NPA each state S in which a message m is expected by a principal is replaced by a set of states $\{S\sigma\downarrow \mid \sigma \in \Sigma\}$. Unification of sent messages with expected messages is performed via a general-purpose unification algorithm known as *folding variant narrowing* [4], which can guarantee that any unifier θ of a sent message with $m\sigma\downarrow$ preserves the irreducibility of that term. Thus the syntactic checks necessary for search termination can proceed without affecting completeness, as long as they are invariant under E and any substitution that preserves irreducibility modulo R.

This solves the state space reduction for Maude-NPA, and has the potential for being applicable to other unification-based protocol analysis tools as well. However, it comes at a price, namely, that narrowing is inefficient. Ideally, we would like to be able to adapt special-purpose algorithms to this approach as well. This motivates the problem of *asymmetric unification*: given a theory $R \cup E$, and two terms u and v where v is in $E\backslash R$ normal form, find a complete set of ER unifiers of $u \stackrel{?}{=} v$ that preserves the irreducibility of v. We write $u \stackrel{?}{=}^{\bullet} v$ to denote an asymmetric unification problem.

In the next section, we define asymmetric unification precisely, by first defining equational unification, and then extending the definition. In the following two sections, we focus on asymmetric XOR unification. We choose this theory for two purposes. One purpose is to give illustrative examples of asymmetric unification. The other purpose is that we have devised a set of inference rules for asymmetric XOR unification that will be implemented in the Maude-NPA. The only other work we are aware of that can deal with asymmetric unification is the work on folding variant narrowing [4].

2 Preliminaries

Given a set of equations E and a substitution σ, we say that σ is an E-*unifier* of $u \stackrel{?}{=} v$ if $u\sigma =_E v\sigma$. If P is a set of equations, then σ is an E-*unifier* of P if σ is an E-unifier of every equation in P. If substitutions σ and θ are E-unifiers of P, then we write $\sigma \leq_E \theta|_P$ iff there is a substitution τ such that $x\sigma\tau =_E x\theta$ for all variables x in P. We sometimes just write $\sigma \leq_E \theta$ if P is obvious, and say that σ is *more general* than θ. If $\sigma \leq_E \theta|_P$ and $\theta \leq_E \sigma|_P$, we say that σ and θ are *equivalent modulo* E *over* P. Again, we often leave out P if it is obvious.

A *complete set of* E-*unifiers* for P is a set Σ of substitutions such that: (i) every member of Σ is an E-unifier of P and (ii) for every E-unifier θ of P there exists $\sigma \in \Sigma$ such that $\sigma \leq_E \theta$. If Σ contains only one element, we call that element a *most general unifier*.

Let R be a set of rewrite rules and E be a set of equations. We consider *class rewriting* modulo E, denoted by $\rightarrow_{R/E}$, which is defined as $=_E \circ \rightarrow_R \circ =_E$ and *extended rewriting* modulo E, denoted by $u \rightarrow_{E\backslash R} v$, where a term u rewrites to a term v if there is a rule $s \rightarrow t$ in R, a subterm s' of u at position p, a substitution σ such that $s\sigma =_E s'$, and $v = u[t\sigma]_p$. Clearly $\rightarrow_{E\backslash R} \subseteq \rightarrow_{R/E}$. We define $\stackrel{*}{\rightarrow}_{E\backslash R}$ as the reflexive transitive closure of $\rightarrow_{E\backslash R}$. We say R is E-*confluent*

iff whenever $s \xrightarrow{*}_{E\backslash R} t$ and $s \xrightarrow{*}_{E\backslash R} u$ then there exists a v such that $t \xrightarrow{*}_{E\backslash R} v$ and $u \xrightarrow{*}_{E\backslash R} v$. R is *E-convergent* iff R is E-confluent and the relation $\rightarrow_{R/E}$ is well founded. A term u is in $E\backslash R$ *normal form* iff there is no v such that $u \rightarrow_{E\backslash R} v$.

Now we extend the definition of E-unification to asymmetric E-unification. Given a set of rewrite rules R, let $Eq(R) = \{s = t \mid s \rightarrow t \in R\}$. Let ER and E be sets of equations and R be a set of E-convergent rewrite rules such that the theory of ER is equivalent to the theory of $E \cup Eq(R)$. We say that substitution σ is an *asymmetric* (R, E)*-unifier* of $s \stackrel{?}{=}^{\bullet} t$ iff σ is an ER-unifier of $s \stackrel{?}{=} t$ and $t\sigma$ is in $E\backslash R$ normal form. Substitution σ is an *asymmetric* (R, E)*-unifier* of P iff σ is an asymmetric (R, E)-unifier of every $s \stackrel{?}{=}^{\bullet} t$ in P. A set of substitutions Σ is a *complete set of asymmetric* (R, E) *unifiers* of P iff (i) every member of Σ is an asymmetric (R, E)-unifier of P, and (ii) for every (R, E)-unifier θ of P there exists $\sigma \in \Sigma$ such that $\sigma \leq_{ER} \theta$ (over $Var(P)$).

Given a complete set of ER-unifiers Σ of P, if θ is in Σ then $\theta\tau$ is an E-unifier of P for any substitution τ. However, this is not the case for asymmetric (R, E) unification. An example illustrating this is given in the next section.

It is easy to see that asymmetric (R, E) unification is at least as hard as ER unification. Given an ER asymmetric unification problem $\{s_1 \stackrel{?}{=} t_1, \cdots, s_n \stackrel{?}{=} t_n\}$, we replace every equation $s_i \stackrel{?}{=} t_i$ by a pair of equations $s_i \stackrel{?}{=} X_i$ and $t_i \stackrel{?}{=} X_i$, where each X_i is a fresh variable. The set of asymmetric (R, E) unifiers of the new set of equations over the original set of variables is the same as the set of ER unifiers of the original set of equations.

3 Examples of asymmetric XOR unification

In this section, we illustrate the above definitions for the case when ER is the XOR theory. We have developed an algorithm for generating a complete set of asymmetric unifiers for the XOR theory. In the next section, we review the key ideas employed in the algorithm. The details are omitted for lack of space.

Let E be the following set of equations:

$$X + Y = Y + X \qquad (X + Y) + Z = X + (Y + Z)$$

Let R be the following set of rewrite rules:

$$X + 0 \rightarrow X \qquad X + X \rightarrow 0 \qquad X + X + Y \rightarrow Y$$

Notice that the third equation is an extension of the second one, and it must be added to make R be E-confluent.

Consider the asymmetric unification problem:

$$c \stackrel{?}{=}^{\bullet} X + Y$$

The substitution $\sigma = [X \mapsto Y + c]$ is a most general XOR-unifier for $c \stackrel{?}{=} X + Y$. However, σ is not an asymmetric (R, E) unifier, because $Y + c + Y$ is not in $E\backslash R$ normal form. In fact, this problem has no asymmetric (R, E) unifier.

Consider another example:

$$a + b \stackrel{?}{=}^{\bullet} X + Y$$

In this case, $\sigma = [X \mapsto Y + a + b]$ is a most general XOR-unifier, but it is not an asymmetric (R, E) unifier. However, this problem does have an asymmetric (R, E) unifier. The complete set of asymmetric XOR unifiers here is $\{[X \mapsto a, Y \mapsto b], [X \mapsto b, Y \mapsto a]\}$, which are instances of the initial XOR-unifier.

Now consider the following example:

$$X \stackrel{?}{=}^{\bullet} Y + Z$$

The substitution $\sigma = [Y \mapsto X + Z]$ is an XOR-unifier, but not an asymmetric (R, E) unifier. But in this case, $\theta = [X \mapsto Y + Z]$ is equivalent to σ modulo XOR, and is also an asymmetric (R, E) unifier.

Finally, we consider one more XOR unification problem:

$$Z \stackrel{?}{=}^{\bullet} X_1 + X_2 \qquad Z \stackrel{?}{=}^{\bullet} Y_1 + Y_2$$

This problem has a most general XOR unifier $\sigma = [X_1 \mapsto Z + X_2, Y_1 \mapsto Z + Y_2]$. This is not an asymmetric (R, E) unifier. Unlike the previous example, we cannot swap some variables to get an equivalent unifier. However, we can solve this problem by adding some fresh variables. The unifier $\theta = [Z \mapsto X_1 + V + Y_2, X_2 \mapsto V + Y_2, Y_1 \mapsto X_1 + V]$ is an asymmetric (R, E) unifier that is equivalent to σ modulo XOR.

As stated above, even though θ is a most general asymmetric (R, E) unifier, its instances need not be asymmetric (R, E) unifiers; for example, $\theta\tau$ is not an asymmetric (R, E) unifier if $\tau = [X_1 \mapsto c, Y_2 \mapsto c]$.

4 An approach for solving asymmetric unification: The case of asymmetric XOR unification

We have developed a general strategy for solving the asymmetric (R, E) unification problem: (i) use an ER unification algorithm to generate a complete set of ER unifiers, and (ii) for each unifier in the complete set, determine whether it is also an asymmetric (R, E) unifier, in which case, it is retained; otherwise, determine if there is an equivalent ER unifier that is also an asymmetric (R, E) unifier; if so, generate it and retain it. In the case in which there does not exist an equivalent asymmetric (R, E) unifier corresponding to a given ER unifier, (a) if no instances of the ER unifier could serve as an asymmetric (R, E) unifier, discard that unifier; otherwise (b) find appropriate instances of the ER unifier and repeat the process.

We have instantiated this strategy for the case of asymmetric XOR unification with uninterpreted function symbols, to give an efficient algorithm, which we plan to implement in the Maude-NPA. Recall that E consists of AC equations for $+$; and R consists of three rules. To solve XOR unification, we use a

rule based XOR unification algorithm given in [8], which works well in practice and is already implemented in Maude-NPA. There are some deterministic rules and some nondeterministic ones. The deterministic rules are given precedence. Often, the deterministic rules completely solve the problem, and that will run in polynomial time, whereas the XOR unification problem is NP-complete in the presence of uninterpreted function symbols.

After solving XOR unification, we must solve the asymmetric unification problem. As in the XOR unification algorithm, the inference rules for asymmetric unification are also divided into deterministic and nondeterministic rules, with deterministic rules having preference over nondeterministic rules for efficiency. For lack of space, we do not give the inference rules here.

The asymmetric XOR unification problem, even without uninterpreted function symbols, is NP-complete. This can be shown by reducing 1-in-3 SAT to this problem. Note that XOR unification without uninterpreted function symbols is solvable in polynomial time. This shows that asymmetric (R, E) unification is harder than ER unification.

An XOR unification problem, without uninterpreted function symbols, always has a most general unifier when solvable. However, an asymmetric XOR unification problem may have a complete set of unifiers with cardinality greater than one, such as in the example $\{a + b \stackrel{?}{=}^{\bullet} X + Y\}$ above.

Bibliography

1. Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *CSFW,* 2001.
2. Hubert Comon-Lundh and Stéphanie Delaune. The finite variant property: How to get rid of some algebraic properties. In Jürgen Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science,* pages 294–307. Springer, 2005.
3. Shaddin F. Doghmi, Joshua D. Guttman, and F. Javier Thayer. Searching for shapes in cryptographic protocols. In *TACAS* 2007.
4. Santiago Escobar, José Meseguer, and Ralf Sasse. Folding Variant Narrowing and Optimal Variant Termination. *Journal of Logic and Algebraic Programming* to appear, 2011.
5. Santiago Escobar, Deepak Kapur, Christopher Lynch, Catherine Meadows, José Meseguer Meseguer, Paliath Narendran, and Ralf Sasse. Protocol analysis in Maude-NPA using unification modulo homomorphic encryption. *Accepted by The 13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming,* 2011.
6. Santiago Escobar, Catherine Meadows, and José Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *FOSAD*, volume 5705 of *Lecture Notes in Computer Science,* pages 1–50. Springer, 2007.
7. Moses Liskov and John D. Ramsdell. Implementing strand space algebras. Mitre Technical Report, March 2011. Available at `http://www.ccs.neu.edu/home/ramsdell/papers/index.html`.
8. Zhiqiang Liu and Christopher Lynch. Efficient general unification for exclusive or with homomorphism. To be presented at the *23rd International Conference on Automated Deduction,* 2011.

Author Index

Baader, Franz 2
Binh, Nguyen Thanh 2
Borgwardt, Stefan 2

Cheney, James 42
Ciobâcă, Ştefan 28

Dzik, Wojciech 21

Erbatur, Serdar 59
Escobar, Santiago 59

Guo, Wei 52

Kapur, Deepak 59
Kavanagh, Ben 42

Liu, Zhiqiang 59
Lynch, Christopher 1, 59

Meadows, Catherine 59
Meseguer, José 59
Morawska, Barbara 2

Narendran, Paliath 59

Otop, Jan 9

Rau, Conrad 35

Sasse, Ralf 59
Schmidt-Schauß, Manfred 35

Verma, Rakesh M. 52

Wojtylak, Piotr 21