Laboratoire
L**O**rrain de
Recherche en
Informatique et ses
Applications

UMR 7503

Proceedings of the

19th International Workshop on Unification

Nara, Japan, April 22, 2005

Edited by Laurent Vigneron (LORIA – UN2-CNRS)

LORIA A05-R-022

LORIA, Campus Scientifique, B.P. 239, 54506 Vandœuvre-lès-Nancy cedex, FRANCE

Preface

UNIF is the main international meeting on unification. Unification is concerned with the problem of identifying given terms, either syntactically or modulo a given logical theory. Syntactic unification is the basic operation of most automated reasoning systems, and unification modulo theories can be used, for instance, to build in special equational theories into theorem provers.

The aim of UNIF'2005, as for the eighteen previous meetings, is to bring together people interested in unification, present recent (even ongoing) work, and discuss new ideas and trends in unification and related fields. This includes scientific presentations, but also descriptions of applications and softwares using unification as a strong component.

This workshop is the nineteenth in the series: UNIF'87 (Val d'Ajol, France), UNIF'88 (Val d'Ajol, France), UNIF'89 (Lambrecht, Germany), UNIF'90 (Leeds, England), UNIF'91 (Barbizon, France), UNIF'92 (Dagstuhl, Germany), UNIF'93 (Boston, USA), UNIF'94 (Val d'Ajol, France), UNIF'95 (Sitges, Spain), UNIF'96 (Herrsching, Germany), UNIF'97 (Orléans, France), UNIF'98 (Rome, Italy), UNIF'99 (Frankfurt, Germany), UNIF'00 (Pittsburgh, USA), UNIF'01 (Siena, Italy), UNIF'02 (Copenhagen, Denmark), UNIF'03 (Valencia, Spain), UNIF'04 (Cork, Ireland). For more information on the series:

`http://www.lsv.ens-cachan.fr/~treinen/unif/`

The UNIF'2005 meeting includes one invited talk, by Joachim Niehren, and a selection of 8 contributed talks.
It also includes a panel entitled 20 Years After OBJ2 organized by Kokichi Futatsugi (JAIST), with Joseph Goguen (University of California at San Diego), Jean-Pierre Jouannaud (Ecole Polytechnique) and Jose Meseguer (University of Illinois at Urbana-Champaign) as additional panelists.

Laurent Vigneron
UNIF'2005 Organization Chair
April 2005

Organization

UNIF'2005 is organized as part of the Federated Conference on Rewriting, Deduction, and Programming (RDP), collocated with RTA (International Conference on Rewriting Techniques and Applications) and TLCA (International Conference on Typed Lambda Calculi and Applications), and several other affiliated workshops.

Organization Committee

Philippe de Groote	LORIA – INRIA Lorraine
Joseph Goguen	University of California at San Diego
Yuichi Kaji	Nara Institute of Science and Technology
Pawel Urzyczyn	Warsaw University
Laurent Vigneron	LORIA – UN2-CNRS

Local Organization

Hitoshi Ohsaki	National Institute of Advanced Industrial Science and Technology (AIST)
Masahito Hasegawa	Research Institute for Mathematical Sciences, Kyoto University

Cover Design

Maki Ishida	National Institute of Advanced Industrial Science and Technology (AIST)

RDP Sponsors

Nara Convention Bureau

JSPS International meeting series

Information Processing Society of Japan Kansai Branch

The Telecommunications Advancement Foundation (TAF)

Kayamori Foundation of Infomational Science Advancement

Foundation for Nara Institute of Science and Technology

Table of Contents

Querying XML-Trees by Tree Automata*

Joachim Niehren, Laurent Planque, Jean-Marc Talbot, and Sophie Tison

INRIA Future, Mostrare project, LIFL, Lille, France
http://www.grappa.univ-lille3.fr/mostrare

Abstract. Information extraction from semi-structured documents requires to find n-ary queries in XML trees that define appropriate sets of n-tuples of nodes. In the first part of the talk, we discuss formalism by which to represent monadic queries in XML trees. In the second part, we report more recent workn on n-ary queries. We propose new representation formalisms by tree automata that capture MSO. We then investigate n-ary queries by unambiguous tree automata which are relevant for query induction in multi-slot information extraction. We show that this representation formalism captures the class of n-ary queries that are finite unions of Cartesian closed queries, a property we prove decidable.
Keyworks: Semi-structured documents, multi-slot information extraction, XML query, tree logic and automata.

1 Introduction

The problem of selecting nodes in trees is the most widespread database querying problem in the context of XML [13]. Many applications, however, are faced with the more general problem of selecting *tuples of nodes* in trees. We therefore study n-ary queries in trees which define sets of n-tuples of nodes.

We are particularly interested in multi-slot information extraction. A typical problem in this class of applications is to extract all pairs of products and prices from a set of XML documents. Such pairs are often expressed by pairs of data values in nodes of XML trees. In this case, we can solve the extraction problem by distinguishing an appropriate binary query in XML trees.

Monadic queries in trees received considerable attention in in previous work on node selection. The most popular representation formalism is the W3C standard XPATH which is used in many other standards in XML technology (XQuery [1], XPointer [2], etc). Other path based query languages were proposed in modal logical PDL style [16].

Monadic Datalog in trees is the logic programming approach for expressing monadic queries. Gottlob and Koch [13, 12] argue in favour of monadic Datalog for information extraction from semi-structured documents. Monadic Datalog is advantageous because of its high expressiveness (all monadic MSO queries can be specified), efficient linear time query answering, and usability in visual wrapper (query) specification. It underlies the Lixto system [3] for Web information extraction. Lixto indeed supports multi-slot information extraction by composing monadic queries for all slots.

Tree automata were proposed in several alternative approaches to represent monadic queries. They have a long tradition in querying which dates back to

* The talk presents results from the paper *N-ary Queries by Tree Automata*

Thatcher and Wright's seminal paper in 1968 [23]. More recent represention approaches for queries in XML trees proposed hedge automata [6] forest automata [17], query automata [19], selection automata [11], and stepwise tree automata [8].

In this paper, we investigate representation formalisms for n-ary queries in trees by tree automata. We are interested in the class of all regular n-ary queries – those that can be expressed by MSO formulas with n free variables – since we believe that this class provides the appropriate expressiveness for multi-slot information extraction from XML trees. We elaborate the tree automata approach towards expressing n-ary queries, inspired by previous work of Neven and Bussche [18] and Berlea and Seidl [4].

Representing n-ary queries in trees by tree automata is advantageous for query induction from annotated examples [7, 15]. Query induction is important for improving visual wrapper induction, as argued by Gottlob et. al. [14]. Recent induction methods [9, 7, 15], however, remain limited to monadic queries in trees. As a first step towards remedying this deficiency we present new representation formalisms for n-ary queries by tree automata.

Contributions of the paper.

1. We propose to represent n-ary queries in ranked trees by successful runs of tree automata.
   (a) The class of representable queries are precisely the class of regular queries, i.e., those expressible in MSO.
   (b) We show that universal and existential run-based queries have the same expressiveness.
2. We investigate the querying power of unambiguous tree automata.
   (a) We show that run-based queries with unambiguous tree automata capture the class of regular n-ary queries that are finite unions of some Cartesian closed queries.
   (b) We show that it is decidable whether a regular n-ary query can be expressed by an unambiguous automaton.
   (c) We show that the problem of answering n-ary queries by unambigous automata on trees has linear time combined complexity.
3. We transfer all results above to n-ary run-based queries in unranked trees – as in XML – with respect to hedge automata [6] and stepwise tree automata [8].

Contribution 1a is new for n-ary queries, but was known in the monadic case [19, 11, 22]. Neven and Bussche's RAG n-ary queries [18] are more expressive than MSO and thus run-based n-ary queries. Binary queries by forest automata [4] have not yet been related to MSO.

Result 1b might come as a surprise for n-ary queries, even though it is well known for monadic queries [18, 5, 11]. It follows by a new proof method that does not depend on the two phase query answering algorithm (which is well-known for monadic queries by attribute grammars, monadic Datalog, or selection automata (see e.g. [18]).

In the monadic case the characterization means that all regular monadic queries in tree can be expressed by unambiguous tree automata. This was proved before, by Neven and Bussche [18] in the framework of attribute grammars (see IBAGs), by Bloem and Engelfriet [5] in the context of MSO definable transformations, and a third time for selection automata [11]. Our proof for the general case is original, even for the monadic case. It relies on a correspondences between queries in trees and tree languages that we elaborate, rather than on the equivalence of tree automata and MSO.

Result 2b implies that it is decidable whether an n-ary query can be expressed a finite union of Cartesian closed queries. The proof 2b is non-trivial. It relies on the decidability of bounded ambiguity in tree automata [21].

Contribution 2c is obtained by generalizing the two phase query answering algorithm to the n-ary case.

Our results are highly relevant to a recent approach to query induction [7]. This approach induces monadic queries for XML trees that are expressed by runs of unambiguous tree automata. The algorithm extends without change to run-based n-ary queries (1a and 3). The important unambiguity assumption, however, limits its coverage to finite unions of Cartesian closed queries (2a). These can be answered efficiently (2c).

2 Regular queries

We recall the definition of n-ary queries in trees and propose to look at them as tree languages, so that we can define regular n-ary queries by tree automata and relate them to MSO.

2.1 N-ary queries in trees

To keep things as simple as possible, we develop our theory for binary trees rather than for more general ranked trees. Binary trees will prove sufficient to extend all our results to unranked trees (Section 6).

A signature Σ for binary trees fixes a finite set of binary function symbols f, g and constants a, b. A *binary tree* $t \in T_\Sigma$ is a ground term over Σ:

$$t ::= a \mid f(t_1, t_2)$$

We define a *node* of a tree t by its relative address from the root. More formally, we define a function $\mathsf{nodes} : T_\Sigma \to 2^{\{1,2\}^*}$ by structural induction:

$$\begin{aligned}&\mathsf{nodes}(a) = \{\epsilon\}\\&\mathsf{nodes}(f(t_1,t_2)) = \{\epsilon\} \cup \{i\pi \mid 1 \le i \le 2, \pi \in \mathsf{nodes}(t_i)\}\end{aligned}$$

The *root* of a tree t is the node ϵ. If $\pi 1 \in \mathsf{nodes}(t)$ then we call $\pi 1$ first child of π and $\pi 2$ its second child. A *leaf* is a node without children. An *inner node* is a node that is not a leaf. For convenience, we will freely identify trees t over Σ with *labeling functions* of type $\mathsf{nodes}(t) \to \Sigma$:

$$\begin{aligned}&a(\epsilon) = a,\\&f(t_1,t_2)(\epsilon) = f,\\&f(t_1,t_2)(i\pi) = t_i(\pi) \qquad 1 \le i \le 2, \pi \in \mathsf{nodes}(t_i)\end{aligned}$$

In this section, we will study queries for the class of binary trees T_Σ with some fixed signature Σ. Our definition of queries, however, will equally apply to other classes of trees or graphs that comes with a notion of nodes.

Definition 1. *Let $n \in \mathbb{N}$ and* tree *a class of trees. An n-ary query for this class is a function q that maps trees $t \in$* tree *to sets of n-tuples of nodes in t:*

$$\forall t \in \mathsf{tree} : q(t) \subseteq \mathsf{nodes}(t)^n$$

Simple examples for monadic queries in binary trees over Σ are the functions leaf and root that maps trees t to the sets of their leaves resp. to the singleton $\{\epsilon\}$. The monadic queries label_c for symbols $c \in \Sigma$ map trees t to the set of c-labeled nodes π of t:

$$\pi \in \mathsf{label}_c(t) \text{ iff } t(\pi) = c$$

The binary query first_child relates inner nodes to their first child, while last_child links inner nodes to their last and thus second child.

Our definition of n-ary queries is quite general in that it does not exclude non-regular queries. For instance, we can query for all pairs (π, π') in trees t such that the subtrees of t on below of π and π' are equal. This query can indeed be expressed by the RAG's of Neven and Bussche [18].

In order to formalize what we mean by regular or irregular queries, we will relate n-ary queries to tree languages (Section 2.2) so that the regularity notion for tree languages carries over (Section 2.3).

2.2 Canonical tree languages of queries

We will discuss canonical tree languages corresponding to n-ary queries in trees, which are inspired by early work on tree automata and MSO [23, 10]. In Section 3.2, we will discuss more compact language encodings of queries, that are particularly useful for query induction.

Let $\mathbb{B} = \{0, 1\}$ be the set of Boolean. We want to encode n-ary queries over Σ as tree languages over the extended signature $\Sigma \times \mathbb{B}^n$. In order to express an n-ary query q, the canonical approach is to encode all valid membership statements $(\pi_1, \ldots, \pi_n) \in q(t)$ into individual trees over $\Sigma \times \mathbb{B}^n$, so that we can collect all of them in a tree language. The coding idea is to annotate all nodes π of tree t by bit vectors $(b_1, \ldots, b_n)$ so that the i-th bit b_i is true if and only if $\pi_i = \pi$ for all $1 \leq i \leq n$:

$$b_i \leftrightarrow \pi_i = \pi$$

For instance, a subset of trees in the canonical tree language of the query child is displayed in Figure 1. We cannot display all of them since that canonical language is infinite.

Let us formally define the correspondence between queries and canonical tree languages. This requires two concepts: characteristic functions and function products. Let S be a set. For every $s \in S$ we define a characteristic function $\mathsf{c}_s : S \to \mathbb{B}$ so that $\mathsf{c}_s(s') \leftrightarrow s = s'$ for all $s' \in S$. For every tree t and $\pi \in \mathsf{nodes}(t)$ let

$$\mathsf{c}_\pi : \mathsf{nodes}(t) \to \mathbb{B}$$

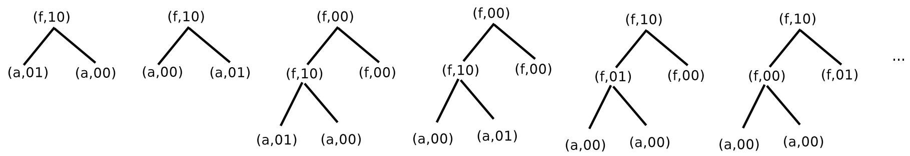

Fig. 1. Some of the trees in the canonical tree language of the binary query child

be the characteristic function of π with respect to $\mathsf{nodes}(t)$. Characteristic function c_π can be identified with a Boolean trees; these have the same nodes as t. Note that Booleans are overloaded in Boolean trees; they serve as binary function symbols and as constants.

We define the product of m functions $f_i : A \to B_i$ to be the function $f_1 * \ldots * f_m : A \to B_1 \times \ldots \times B_m$ that satisfies for all $a \in A$:

$$(f_1 * \ldots * f_m)(a) = (f_1(a), \ldots, f_m(a))$$

Thereby, we have defined the products of m trees $t_1, \ldots, t_m$ of the same shape but with possibly distinct signatures $\Sigma_1, \ldots, \Sigma_m$, since we identify trees with labeling functions:

$$t_1 * \ldots * t_m \text{ where } t_i \in T_{\Sigma_i} \text{ for } 1 \leq i \leq m$$

We call a tree t over $T_{\Sigma\times\mathbb{B}^n}$ *canonical* if for all $1 \leq i \leq n$ there exists precisely one node $\pi \in \mathsf{nodes}(t)$ such that the i-th Boolean $b_i = 1$ where $t(\pi) = (f, b_1, \ldots, b_n)$ for some $f \in \Sigma$. We call a tree language *canonical* if all its trees are canonical. We write Can^n_Σ for the set of all canonical trees over $\Sigma \times \mathbb{B}^n$.

We can now define canonical languages $\mathsf{can}(q)$ that correspond to n-ary queries q over Σ. The trees in $\mathsf{can}(q) \subseteq T_{\Sigma\times\mathbb{B}^n}$ correspond one-to-one and onto to the tuples that q selects in trees $t \in T_\Sigma$. Each tree in the canonical language of q is obtained by multiplying the characteristic functions of some selected node tuple:

$$\mathsf{can}(q) = \{t * \mathsf{c}_{\pi_1} * \ldots * \mathsf{c}_{\pi_n} \mid t \in T_\Sigma,\ (\pi_1, \ldots, \pi_n) \in q(t)\}$$

Lemma 1. *A tree language $L \subseteq T_{\Sigma\times\mathbb{B}^n}$ is canonical if and only if $L = \mathsf{can}(q)$ for some n-ary query q over Σ; this query q is always unique.*

We next define set operations on n-ary queries over the same signature Σ and show that they correspond precisely to the set operations on their canonical tree languages. This is why we consider canonical languages to be canonical. Given two n-ary queries q_1 and q_2 we define their union $q_1 \cup q_2$ by imposing for all $t \in T_\Sigma$:

$$(q_1 \cup q_2)(t) = q_1(t) \cup q_2(t)$$

The complement q^c of an n-ary query q is the n-ary query satisfying:

$$q^c(t) = \mathsf{nodes}(t)^n - q(t)$$

$$\mathsf{runs}_A(a) = \{p \mid a \to p \in \mathsf{rules}(A)\}$$

$$\frac{r_1 \in \mathsf{runs}_A(t_1) \quad r_2 \in \mathsf{runs}_A(t_2) \quad f(r_1(\epsilon), r_2(\epsilon)) \to p \in \mathsf{rules}(A)}{p(r_1, r_2) \in \mathsf{runs}_A(f(t_1, t_2))}$$

Fig. 2. Runs of a tree automaton A

The Cartesian product of an n-ary query q_1 and an m-ary query q_2 is the $n+m$ ary query $q_1 \times q_2$ such that for all trees $t \in T_\Sigma$:

$$(q_1 \times q_2)(t) = q_1(t) \times q_2(t)$$

Lemma 2. *For all n-ary queries q, q_1, q_2 and m-ary queries q' over Σ:*

$$\begin{aligned} \mathsf{can}(q_1 \cup q_2) &= \mathsf{can}(q_1) \cup \mathsf{can}(q_2) \\ \mathsf{can}(q^c) &= \mathsf{Can}^n_\Sigma - \mathsf{can}(q) \\ \mathsf{can}(q \times q') &= \{t * \beta * \beta' \mid t \in T_\Sigma, t * \beta \in \mathsf{can}(q),\ t * \beta' \in \mathsf{can}(q')\} \end{aligned}$$

2.3 Regularity

We recall the definitions of tree automata and regular tree languages and define the regularity of n-ary queries in trees.

A *tree automaton* A for binary trees over Σ consists of a finite set $\mathsf{states}(A)$, a finite set $\mathsf{rules}(A)$, and a set $\mathsf{final}(A) \subseteq \mathsf{states}(A)$. The rules of A may have two forms:

$$a \to p \quad \text{or} \quad f(p_1, p_2) \to p$$

where $f \in \Sigma$ is a binary function symbol, $a \in \Sigma$ a constant and $p, p_1, p_2 \in \mathsf{states}(A)$.

A *run* of a tree automata A on a tree t is a function $r : \mathsf{nodes}(t) \to \mathsf{states}(A)$ that associates states to nodes of t according to the rules of A, or equivalently, a tree labeled in $\mathsf{states}(A)$ with the same domain than t. Figure 2 defines the set $\mathsf{runs}_A(t)$ of runs of A on t by recursion on the structure of t.

A run r of a tree automaton A on a tree t is called *successful* if it labels the root of t by some state in $\mathsf{final}(A)$.

$$\mathsf{succ\_runs}_A(t) = \{r \in \mathsf{runs}_A(t) \mid r(\epsilon) \in \mathsf{final}(A)\}$$

Example 1. Consider automaton A_1 over signature $\Sigma = \{f, a\}$. Two runs of A_1 are presented in Figure 3. Successful runs of A_1 on arbitrary trees label the left most a-leaf by 1 and all others by $*$. The ancestors of the left most a-leaf will be assigned to y. All other inner nodes will be marked by $*$. The final states are y and 1. In summary, we have the following states:

$$\begin{aligned} \mathsf{states}(A_1) &= \{1, *, y\} \\ \mathsf{final}(A_1) &= \{1, y\} \end{aligned}$$

The rules in $\mathsf{rules}(A_1)$ need to verify the intuitive meaning that we associated with the states:

$$\begin{array}{lll} a \to 1 & f(1, *) \to y & f(y, *) \to y \\ a \to * & f(*, *) \to * & \end{array}$$

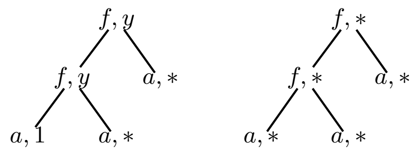

Fig. 3. Two runs of automaton A_1 on the same tree; only the left one is successful.

Note that every tree permits at most one successful run by automaton A_1 but possibly other unsuccessful runs.

A tree t is *accepted* by a tree automaton A if A has a successful run on t, i.e., $\mathsf{succ\_runs}_A(t) \neq \emptyset$. The *language* $L(A)$ recognized by a tree automaton A is the set of trees t that A accepts. A tree language $L \subseteq T_\Sigma$ is *regular* if and only if it is recognized by some tree automaton A over Σ, so that $L = L(A)$. Automaton A_1 from Example 1, for instance, accepts all trees in $T_{\{f,a\}}$.

Definition 2. *A n-ary query q in trees over Σ is regular if its canonical tree language* $\mathsf{can}(q) \subseteq T_{\Sigma\times\mathbb{B}^n}$ *is recognized by a tree automaton.*

Proposition 1. *Regular queries are closed under union, intersection, complementation, and Cartesian products.*

Proof. This follows from Lemma 2. To see this, let us exemplify the proof for the union operator. If q_1 and q_2 are regular queries then by definition their canonical languages are regular: $\mathsf{can}(q_1)$ and $\mathsf{can}(q_2)$. Unions of regular tree languages are regular, and thus $\mathsf{can}(q_1) \cup \mathsf{can}(q_2)$. This language is equal to $\mathsf{can}(q_1 \cup q_2)$ by Lemma 2. The regularity of $\mathsf{can}(q_1 \cup q_2)$ is equivalent to that $q_1 \cup q_2$ is regular by definition.

Let us recall three standard notions for tree automata: determinism, unambiguity, and relabelings [10]. A tree automaton A is (bottom-up) *deterministic* if no two of its rules have the same left hand side. It is *unambiguous*, if no tree $t \in T_\Sigma$ permits more than one successful run in $\mathsf{succ\_runs}_A(t)$.

A *relabeling morphism* for binary signatures Σ and Σ' is a mapping $h : \Sigma \to \Sigma'$ that maps constants to constants and binary function symbols to binary function symbols. Relabeling morphisms $h : \Sigma \to \Sigma'$ can be lifted homomorphically to trees $h : T_\Sigma \to T_{\Sigma'}$ by imposing for all $f \in \Sigma$ and $t, t' \in T_\Sigma$:

$$h(f(t_1, t_2)) = h(f)(h(t_1), h(t_2))$$

It is well-known that relabeling morphisms as well their inverse images preserve regularity, i.e. if $L \subseteq T_\Sigma$ is regular then $h(L)$ and if $L' \subseteq T_{\Sigma'}$ is regular then $h^{-1}(L')$.

2.4 MSO queries

We recall the monadic second-order logic (MSO) in binary trees and how it can be used to represent n-ary queries. The classical theorem of Thatcher and

Wright [23] then shows that regular n-ary queries capture precisely the class of MSO definable n-ary queries.

We identify binary trees $t \in T_\Sigma$ with *logical structures*. The domain of the structure of t is the set $\mathsf{nodes}(t)$. Its signature consists of the binary relation symbols $\mathsf{first\_child}$ and $\mathsf{last\_child}$, and the monadic relation symbols label_a for all $a \in \Sigma$. These symbols are interpreted by the corresponding node relations of t.

$$\begin{aligned}\mathsf{first\_child}^t &= \{(\pi, \pi 1) \mid \pi 1 \in \mathsf{nodes}(t)\}\\ \mathsf{last\_child}^t &= \{(\pi, \pi 2) \mid \pi 2 \in \mathsf{nodes}(t)\}\\ \mathsf{label}_a^t &= \{\pi \mid t(\pi) = a\}\end{aligned}$$

Let x, y, z range over an infinite set of node variables and p over an infinite set of monadic predicates. Formulas ϕ of MSO have the following abstract syntax:

$$\begin{aligned}\phi ::= &\ p(x) \mid \mathsf{first\_child}(x, y) \mid \mathsf{last\_child}(x, y) \mid \mathsf{label}_a(x)\\ \mid &\ \neg\phi \mid \phi_1 \wedge \phi_2 \mid \forall x.\phi \mid \forall p.\phi\end{aligned}$$

A variable assignment α into a tree t maps node variables to nodes of t and monadic predicates to sets of nodes of t. We define the validity of formulas ϕ in trees t under variable assignments α in the usual Tarskian manner:

$$t, \alpha \models \phi$$

Formulas ϕ with n free variables $x_1, ..., x_n$ represent n-ary queries $\mathsf{query}_{\phi(x_1,...,x_n)}$ which satisfies for all trees $t \in T_\Sigma$

$$\mathsf{query}_{\phi(x_1,...,x_n)}(t) = \{(\alpha(x_1), ..., \alpha(x_n)) \mid t, \alpha \models \phi\}$$

Theorem 1. *(Thatcher and Wright [23]): An n-ary query in binary trees is regular if and only if it can be expressed by some MSO formula with n free node variables.*

3 Run-based queries

We now introduce new representation formalisms for regular n-ary queries based on successful runs of tree automata, that conservatively extend previous approaches to monadic queries [18, 11, 22]. Run-based query formalisms are strictly less expressive that Neven and Bussche's [18] n-ary relational attribute grammars (RAGs) queries, and technically simpler. They provide a simpler and more general alternative to Seidl and Berlea's [4] binary queries in unranked trees by forest automata.

3.1 Existential run-based queries

The idea is to use successful runs of tree automata not only to accept trees but also to select nodes in them. This way, one can avoid the indirection through corresponding tree languages in representations formalism. As we will see, this correspondence will reappear when proving that run-based queries capture MSO.

A *run-based* n-ary query in binary trees over Σ is specified by a tree automaton A over Σ and a set $S \subseteq \mathsf{states}(A)^n$ of so called *selection tuples*. An *existential run-based query* $\mathsf{query}^{\exists}_{A,S}$ selects all those tuples of nodes $(\pi_1, \ldots, \pi_n)$ in a tree t that are assigned to a selection tuple by some successful run of A on t:

$$\mathsf{query}^{\exists}_{A,S}(t) = \{(\pi_1, \ldots, \pi_n) \mid \exists r \in \mathsf{succ\_runs}_A(t),\ (r(\pi_1), \ldots, r(\pi_n)) \in S\}$$

Example 2. Reconsider automaton A_1 from Example 1. The monadic query $\mathsf{query}_{A_1,\{1\}}$ selects the left most a-leaf.

This monadic query cannot be represented by any deterministic tree automaton. Non-determinism is needed to distinguish different occurrences of a-leaves. Automata have to guess states for nodes when processing trees bottom up. The guesses need be checked for correctness: they are correct only if they can be extended to a successful runs.

Example 2 illustrates that runs of deterministic tree automata are not sufficient to define all regular monadic queries (even though they can recognize all regular languages). Nevertheless, one can do with a limited form of non-determinism in monadic queries. Unambiguous automata are enough, as in the example. This means that there exists always exists at most one correct way of guessing states. This result is well know in the monadic case. It was proved in the context of attribute grammars by Neven and Bussche [18] and independently by Bloem and Engelfriet [5].

Example 3. Let us define the binary query that selects pairs of a-leaves coupled with right sister b-leaves. We assume that the signature is $\Sigma = \{f, a, b\}$. We define an automaton A_3 with $\mathsf{states}(A_3) = \{a, b, *, y\}$ that will produce successful runs of the form of Figure 4 that select the required pairs as follows:

$$\mathsf{query}_{A_3,\{(a,b)\}}$$

This means that the automaton will assign state a the selected a-leaf and state b the its selected younger b-sister. The final state y will be assigned to all common ancestors of the selected a and b siblings: $\mathsf{final}(A_3) = \{y\}$. State $*$ can be assigned to all other nodes. The following rules verify these properties:

$$\begin{array}{llll} a \rightarrow a & b \rightarrow b & f(*,*) \rightarrow * & f(a,b) \rightarrow y \\ a \rightarrow * & b \rightarrow * & f(y,*) \rightarrow y & f(*,y) \rightarrow y \end{array}$$

Every successful run of automaton A_3 will select a single pair (see e.g. Figure 4). Different pairs are separated by different runs; thus they cannot be mixed up.

This example illustrates the trick in our representation of n-ary queries. In order to not mix up the components of selected pairs, we separate them in different runs. This trick gives hope that run-based existential n-ary queries can represent all regular n-ary queries. However, it raises doubts on the querying power on unambiguous tree automata in existential run-based n-ary queries.

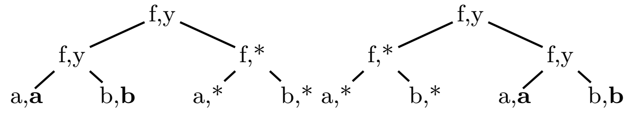

Fig. 4. The two successful runs select both pairs of a-leaves and right sister b-leaves

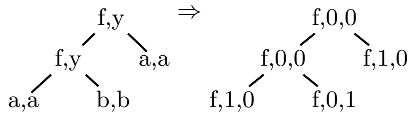

Fig. 5. From runs to Boolean query representations: select pairs of a-leaves and b-leaves

3.2 Compact tree languages for queries

Runs of tree automata can select a whole set of tuples. Sets of runs thus provide a much more compact query representations than canonical tree languages. This holds in particular for queries by unambiguous tree automata where all tuples for a tree are selected in a single run.

Understanding the expressive power of existential run-based queries is best based on a correspondence between arbitrary tree languages over $\Sigma \times \mathbb{B}^n$ and n-ary queries over Σ. This means that we give up canonicity in favour of compactness.

Example 4. Let us select pairs of a-leaves and b-leaves. This query is the Cartesian product of two monadic queries for a-leaves and b-leaves respectively. It can be defined by tree automaton A_4 with $\mathsf{states}(A_4) = \{a, b, y\}$ and rules:

$$a \to a, \qquad b \to b, \qquad f(p, p') \to y \text{ for all } p, p' \in \{a, b, y\}$$

All a-leaves will be mapped to state a, all b-leaves to state b, and all inner nodes to state y. All states are final: $\mathsf{final}(A_4) = \{a, b, y\}$. The query is defined by $\mathsf{query}_{A_4,\{(a,b)\}}$.

Automaton A_4 is deterministic and thus unambiguous. Let $t \in T_\Sigma$ and suppose that r is the unique successful run of A_4 on t. The tree $t * r$ can be easily transformed into a tree over $\Sigma \times \mathbb{B}^2$ that compactly encodes all tuples selected by the automaton on t. An example is illustrated in Figure 5.

The example illustrates that every tree language $L \subseteq T_{\Sigma\times\mathbb{B}^n}$ – canonical or not – corresponds to an n-ary query in trees over Σ. In order to define this query formally, we need a partial order on bit-vectors in $(b'_1, \ldots, b'_n), (b_1, \ldots, b_n) \in \mathbb{B}^n$:

$$(b'_1, \ldots, b'_n) \leq (b_1, \ldots, b_n) \text{ iff } \forall 1 \leq i \leq n.\ b'_i \leq b_i$$

We next lift this partial order to trees whose labels are extended by bit vectors $t * \beta', t * \beta \in T_{\Sigma\times\mathbb{B}^n}$:

$$t * \beta' \leq t * \beta \text{ iff } \forall \pi \in \mathsf{nodes}(t).\ \beta'(\pi) \leq \beta(\pi)$$

Let $L \subseteq T_{\Sigma\times\mathbb{B}^n}$ be a tree language. The corresponding n-ary query $\mathsf{corr\_query}(L)$ is the unique query whose canonical tree language satisfies:

$$\mathsf{can}(\mathsf{corr\_query}(L)) = \{t * \beta' \in \mathsf{Can}^n_\Sigma \mid \exists t * \beta \in L.\ t * \beta' \leq t * \beta\}$$

Lemma 3. *If $L \subseteq T_{\Sigma\times\mathbb{B}^n}$ is a regular tree language then the corresponding n-ary query* $\mathsf{corr\_query}(L)$ *is regular too.*

Proof. Let $\bot_0$ be a constant and $\bot_2$ be a binary symbol. We define two relabeling morphism such that for all $g \in \Sigma$ and $\boldsymbol{b}, \boldsymbol{b'} \in \mathbb{B}^n$: **JN:** Picture needed

$$\begin{array}{l}
\Sigma^{n,n} = \Sigma \times \mathbb{B}^n \times \mathbb{B}^n \\
h : \Sigma^{n,n} \to (\Sigma \times \mathbb{B}^n) \cup \{\bot_0, \bot_2\} \\
h((g, \boldsymbol{b}, \boldsymbol{b'})) = \mathsf{if}\ \boldsymbol{b} \leq \boldsymbol{b'}\ \mathsf{then}\ (g, \boldsymbol{b'})\ \mathsf{else}\ \bot_{\mathrm{arity}(g)} \\
h_1 : \Sigma^{n,n} \to \Sigma \times \mathbb{B}^n \\
h_1((g, \boldsymbol{b}, \boldsymbol{b'})) = (g, \boldsymbol{b})
\end{array}$$

We then have $\mathsf{can}(\mathsf{corr\_query}(L)) = h_1(h^{-1}(L)) \cap \mathsf{Can}^n_\Sigma$. This language is regular since L and Can^n_Σ are.

3.3 Existential queries and regularity

Our next goal is to show that existential run-based queries capture the class of regular n-ary queries.

Lemma 4. *Existential run-based n-ary query are regular.*

Proof. Existential run-based queries are finite unions of existential run-based queries with singleton selection sets:

$$\mathsf{query}^\exists_{A,S} = \cup_{\boldsymbol{p}\in S}\ \mathsf{query}^\exists_{A,\{\boldsymbol{p}\}}$$

By Proposition 1 it remains to show that run-based queries $\mathsf{query}^\exists_{A,\{\boldsymbol{p}\}}$ with singleton selection sets are regular. Let us fix an automaton A with signature Σ and selection tuple $\boldsymbol{p} = (p_1, \ldots, p_n) \in \mathsf{states}(A)^n$. For every $p \in \mathsf{states}(A)$, let $\mathsf{c}_p : \mathsf{states}(A) \to \mathbb{B}$ be the characteristic function of p. We construct a new automaton $A_{\boldsymbol{p}}$ over the signature $\Sigma \times \mathbb{B}^n$ so that for all trees $t * \beta \in T_{\Sigma\times\mathbb{B}^n}$ and functions $r : \mathsf{nodes}(t) \to \mathsf{states}(A)$:

$$\begin{array}{rl}
& r \in \mathsf{succ_runs}_A(t) \text{ and } \beta = \mathsf{c}_{p_1} \circ r * \ldots * \mathsf{c}_{p_n} \circ r \\
\text{iff} & r \in \mathsf{succ_runs}_{A_{\boldsymbol{p}}}(t * \beta)
\end{array}$$

Both automata have the same runs but on slightly different trees. The idea is that runs of $A_{\boldsymbol{p}}$ additionally test whether the bit vectors in tree β are licensed by runs of A on t with respect to the selection tuple $\boldsymbol{p}$. We define automaton $A_{\boldsymbol{p}}$ such that:

$$\frac{a \to p' \in \mathsf{rules}(A)}{(a, \mathsf{c}_{p_1}(p'), \ldots, \mathsf{c}_{p_n}(p')) \to p' \in \mathsf{rules}(A_{\boldsymbol{p}})}$$

$$\frac{f(p'_1, p'_2) \to p' \in \mathsf{rules}(A)}{(f, \mathsf{c}_{p_1}(p'), \ldots, \mathsf{c}_{p_n}(p'))(p'_1, p'_2) \to p' \in \mathsf{rules}(A_{\boldsymbol{p}})}$$

$$\mathsf{final}(A_{\boldsymbol{p}}) = \mathsf{final}(A)$$

The query corresponding to $L(A_{\boldsymbol{p}})$ is regular by Lemma 3. It remains to verify that this query is identical to $\mathsf{query}^{\exists}_{A,\{\boldsymbol{p}\}}$:

$$\mathsf{corr\_query}(L(A_{\boldsymbol{p}})) = \mathsf{query}^{\exists}_{A,\{\boldsymbol{p}\}}$$

This mainly follows from the definitions of corresponding tree languages and the above property of $A_{\boldsymbol{p}}$. We show for all trees $t \in T_\Sigma$ and nodes $\pi_1, \ldots, \pi_n \in \mathsf{nodes}(t)$:

$$\begin{aligned}
&(\pi_1, \ldots, \pi_n) \in \mathsf{corr_query}(L(A_{\boldsymbol{p}}))(t) \\
\Leftrightarrow\ & t * \mathsf{c}_{\pi_1} * \ldots * \mathsf{c}_{\pi_n} \in \mathsf{can}(\mathsf{corr_query}(L(A_{\boldsymbol{p}}))) \\
\Leftrightarrow\ & \exists t * \beta \in L(A_{\boldsymbol{p}}).\ t * \mathsf{c}_{\pi_1} * \ldots * \mathsf{c}_{\pi_n} \leq t * \beta \\
\Leftrightarrow\ & \exists \beta \exists r \in \mathsf{succ_runs}_{A_{\boldsymbol{p}}}(t * \beta).\ t * \mathsf{c}_{\pi_1} * \ldots * \mathsf{c}_{\pi_n} \leq t * \beta \\
\Leftrightarrow\ & \exists r \in \mathsf{succ_runs}_A(t).\ t * \mathsf{c}_{\pi_1} * \ldots * \mathsf{c}_{\pi_n} \\
& \qquad \leq t * \mathsf{c}_{p_1} \circ r * \ldots * \mathsf{c}_{p_n} \circ r \\
\Leftrightarrow\ & \exists r \in \mathsf{succ_runs}_A(t).\ r(\pi_1) = p_1, \ldots, r(\pi_n) = p_n \\
\Leftrightarrow\ & (\pi_1, \ldots, \pi_n) \in \mathsf{query}^{\exists}_{A,\{\boldsymbol{p}\}}(t)
\end{aligned}$$

Lemma 5. *Every regular n-ary query is equal to some existential run-based query.*

Proof. Let L be a regular language over $\Sigma \times \mathbb{B}^n$ and A be an automaton such that $L(A) = L$. We compute an automaton C over Σ and a selection set S such that $L(C) = \{t \mid t * \beta \in L(A)\}$ and $\mathsf{corr\_query}(L(A)) = \mathsf{query}^{\exists}_{C,S}$:

$$\frac{(a, b_1, ..., b_n) \to q \in \mathsf{rules}(A)}{a \to (q, b_1, ..., b_n) \in \mathsf{rules}(C)}$$

$$\frac{(f, b_1, ..., b_n)(q_1, q_2) \to q \in \mathsf{rules}(A)}{(f(q_1, b_1^1, ...b_1^n), (q_2, b_2^1, ..., b_2^n)) \to (q, b_1, ..., b_n) \in \mathsf{rules}(C)}$$

Finally, let $\mathsf{states}(C) = \mathsf{states}(A) * \mathbb{B}^n$, $\mathsf{final}(C) = \mathsf{final}(A) \times \mathbb{B}^n$ and $S = Q_1 \times ... \times Q_n$ where for all $1 \leq i \leq n$:

$$Q_i = \{(q, b_1, ..., b_n) \in \mathsf{states}(C) \mid b_i = 1\}$$

The correctness of the construction is proved by showing that for any term t, $\mathsf{runs}_C(t) = \{\beta \mid r \in \mathsf{runs}_A(t * \beta)\}$ and $\mathsf{succ\_runs}_C(t) = \{\beta \mid r \in \mathsf{succ\_runs}_A(t * \beta)\}$.

Theorem 2. *Existential run-based n-ary queries capture precisely the class of regular n-ary queries.*

3.4 Universal run-based queries

Universal run-based query quantify universally over successful runs rather than existentially.

$$\mathsf{query}^{\forall}_{A,S}(t) = \{(\pi_1, \ldots, \pi_n) \ \mid\ \forall r \in \mathsf{succ\_runs}_A(t),\ (r(\pi_1), \ldots, r(\pi_n)) \in S\}$$

$\mathsf{rules}(A_6)$:

$a \to 0$	$f(1,2) \to y$
$a \to 1$	$f(0,y) \to y$
$a \to 2$	$f(y,0) \to y$
	$f(0,0) \to 0$
	$f(0,0) \to 1$
	$f(0,0) \to 2$

$\mathsf{final}(A_6) = \{y, 1, 2\}$

selection tuples:
$S_6 = \{(1,2), (0,0), (y,0), (0,y)\}$

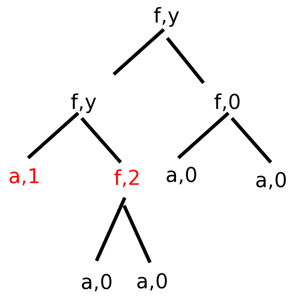

Fig. 6. An example for a universal run-based query: $\mathsf{next\_sibl} = \mathsf{query}^{\forall}_{A_6,S_6} = \mathsf{query}^{\exists}_{A_6,\{(1,2)\}}$

Universal queries were proposed before in selection automata [11] and universal BAGs [18].

An example is given in Figure 6. We represent the binary query $\mathsf{next\_sibl}$ universally, which relates first and second children with the same mother. Successful runs of automaton A_6 will assign the state pair $(1,2)$ to at most one node pair satisfying the query. Descendants and cousins of these nodes will be assigned to state 0, all others (ancestors in fact) to y. The required query can be expressed existentially by $\mathsf{query}^{\exists}_{A_6,\{(1,2)\}}$.

Runs in universal queries refute all those tuples that they don't select. Thus, one needs sufficiently many selection states so that correct tuples are never rejected. In the example, selected pairs will always be labeled by state pairs in $S_6 = \{(1,2), (0,0), (0,y), (y,0)\}$. All other node pairs can be refuted by successful runs that assign state pairs in the complement of S_6. Hence $\mathsf{query}^{\exists}_{A_6,\{(1,2)\}} = \mathsf{query}^{\forall}_{A_6,S_6}$.

Lemma 6. *The complement of an existential run-based query is an existential run-based query.*

Proof. Let q be an existential query. By Lemma 4, q is regular, so $\mathsf{can}(q)$ is regular. By Lemma 2, $\mathsf{can}(q^{\mathsf{c}}) = \mathsf{Can}^n_{\Sigma} - \mathsf{can}(q)$. As $\mathsf{can}(q) \subseteq \mathsf{Can}^n_{\Sigma} \subseteq T_{\Sigma\times\mathbb{B}^n}$, then $\mathsf{can}(q^{\mathsf{c}}) = \mathsf{Can}^n_{\Sigma} \cap \mathsf{can}^{\mathsf{c}}(q)$, with $\mathsf{can}^{\mathsf{c}}(q)$ is the complement of $\mathsf{can}(q)$ into $T_{\Sigma\times\mathbb{B}^n}$. So, $\mathsf{can}(q^{\mathsf{c}})$ is regular and by Lemma 5, q^{c} is existential.

Theorem 3. *Existential and universal queries have the same expressiveness.*

Proof. We prove first that any universal query has an equivalent existential query. An universal query can be written as $\mathsf{query}^{\forall}_{A,S}$ for an automaton A and a set $S \subseteq \mathsf{states}(A)^n$.

$$\mathsf{query}^{\forall}_{A,S}(t) = \{\boldsymbol{\pi} \mid \forall r \in \mathsf{succ\_runs}_A(t), r(\boldsymbol{\pi}) \in S\}$$

Obviously, the complement of $\mathsf{query}^{\forall}_{A,S}(t)$ is precisely the set

$$\{\boldsymbol{\pi} \mid \exists r \in \mathsf{succ\_runs}_A(t), r(\boldsymbol{\pi}) \in \mathsf{states}(t)^n \backslash S\}$$

This is by definition $\mathsf{query}^{\exists}_{A,\mathsf{states}(t)^n\backslash S}$. We can now conclude using that existential queries are closed under complement (Lemma 6). We prove now that any existential query has an equivalent universal query. By Lemma 6, for any existential query q, there exists A and S such that $q = (\mathsf{query}^{\exists}_{A,S})^{\mathsf{c}}$. Therefore, for any tree t, $q(t)$ is equal to

$$\{\boldsymbol{\pi} \mid \forall r \in \mathsf{succ\_runs}_A(t), r(\boldsymbol{\pi}) \in \mathsf{states}(t)^n \backslash S\}$$

This is precisely $\mathsf{query}^{\forall}_{A,\mathsf{states}(t)^n \backslash S}$

4 Unambiguous tree automata

Our next goal is to investigate the querying power of unambiguous tree automata in the n-ary case. Monadic queries by unambiguous tree automata are used in a recent approach to query induction from annotated examples [7]. We believe that this approach can be extended from monadic queries to n-ary queries. This is why we want to understand the precise expressiveness of n-ary queries by unambiguous tree automata.

The main idea for query induction in the cite approach to represent n-ary queries over Σ as tree languages over $\Sigma \times \mathbb{B}^n$ and to infer tree automata for such tree languages by methods of grammatical inference [20]. Compact tree languages for representing queries seem advantageous; they are much easier to infer than canonical tree language. N-ary queries by unambiguous tree automata allow for compact representation, where all tuples selected in a single tree over Σ can be represented by a single tree over $\Sigma \times \mathbb{B}^n$.

4.1 Finite unions of Cartesian closed queries

We call a n-ary query Cartesian closed if it is a Cartesian product of monadic queries.

Proposition 2. *Run-based n-ary queries $\mathsf{query}_{A,S}$ by unambiguous automata A are finite unions of Cartesian closed n-ary queries.*

Proof. We show that $\mathsf{query}_{A,S}$ is Cartesian closed for singletons S. Let $S = \{(p_1, \ldots, p_n)\}$. As for any tree t there exists at most one successful run r by A, we have:

$$\mathsf{query}_{A,S} = \mathsf{query}_{A,\{p_1\}} \times \ldots \times \mathsf{query}_{A,\{p_n\}}$$

Theorem 4. *Run-based queries by unambiguous tree automata capture the class of finite unions of Cartesian closed regular queries.*

The simple direction is proved by Proposition 2. The converse is more involved; it needs some auxiliary notions.

We define $\mathsf{sat}(q)$, the saturated language of q, as the least subset of $T_{\Sigma\times\mathbb{B}^n}$ satisfying (i) $\mathsf{can}(q) = \mathsf{can}(\mathsf{corr\_query}(\mathsf{sat}(q)))$ and (ii) for all $t \in T_\Sigma$, $t * \beta$ belongs to $\mathsf{sat}(q)$ if either there exists $t*\beta'$ in $\mathsf{can}(q)$ such that $t*\beta' \le t*\beta$ or β satisfies $\beta(\pi) = 0^n$ for all $\pi \in \mathsf{dom}(t)$. Moreover, we define $\mathsf{max}(q)$ the language of maximals for q defined as $\{t*\beta \in \mathsf{sat}(q) \mid \neg(\exists t*\beta' \in \mathsf{sat}(q) : t*\beta < t*\beta')\}$.

Lemma 7. *Let q be a regular query. Then both $\mathsf{sat}(q)$ and $\mathsf{max}(q)$ are regular and computable.*

Proof. Let $\bot_0$ be a constant and $\bot_2$ be a binary symbol. We define two relabeling morphisms such that for all $g \in \Sigma$ and $\boldsymbol{b}, \boldsymbol{b'} \in \mathbb{B}^n$:

$$
\begin{array}{l}
h' : \Sigma \times \mathbb{B}^n \times \mathbb{B}^n \to (\Sigma \times \mathbb{B}^n) \cup \{\bot_0, \bot_2\} \\
h'((g, \boldsymbol{b}, \boldsymbol{b'})) = \textsf{if } \boldsymbol{b} \leq \boldsymbol{b'} \textsf{ then } (g, \boldsymbol{b}) \textsf{ else } \bot_{\text{arity}(g)} \\
h_2 : \Sigma \times \mathbb{B}^n \times \mathbb{B}^n \to \Sigma \times \mathbb{B}^n \\
h_2((g, \boldsymbol{b}, \boldsymbol{b'})) = (g, \boldsymbol{b'})
\end{array}
$$

Let us define $\mathsf{up}(q)$ the set $h_2(h'^{-1}(\mathsf{can}(q)))$. Now, let us consider q^{c} be the query complement of q and let $\mathsf{up}(q^{\mathsf{c}})$ be the set $h_2(h'^{-1}(\mathsf{can}(q^{\mathsf{c}}))$. Obviously, one can built automata for $\mathsf{up}(q)$ and $\mathsf{up}(q^{\mathsf{c}})$. Therefore, one can do so for $\mathsf{sat}(q) = \mathsf{up}(q) - \mathsf{up}(q^{\mathsf{c}})$. Considering now the relabeling morphism h_1 defined in proofs of Lemma 3, $\mathsf{max}(q)$ can be defined as $\mathsf{sat}(q) \cap (h_1(h_2^{-1}(\mathsf{sat}(q)) \cap \mathsf{SG}))^{\mathsf{c}}$ where SG is the regular language of trees written over the signature $\Sigma \times \mathbb{B}^n \times \mathbb{B}^n$ such that for any $t * \beta * \beta'$ in SG, it holds that $t * \beta < t * \beta'$.

Proposition 3. *Let q be a Cartesian closed n-ary query. For all $t \in T_\Sigma$, there exists exactly one tree $t * \beta$ in $\mathsf{max}(q)$.*

Proof. The query q being Cartesian closed, for all $t \in T_\Sigma$, it can be written as $E_1^t \times \ldots \times E_n^t$. For each $t \in T_\Sigma$, we consider the tree $t * \beta^q$ defined as for all $\pi \in \mathsf{dom}(t)$ and $\beta^q(\pi) = (b_1, \ldots, b_n)$, $b_i \leftrightarrow (\pi \in E_i^t)$. Defining L_q as $\{t * \beta^q \mid t \in T_\Sigma\}$, it is easy to verify $q = \mathsf{corr\_query}(L_q)$.

Proposition 4. *Let q be a Cartesian closed regular n-ary query. Then q is a unambiguous query.*

Proof. By Lemma 7 we can compute an automaton M recognizing $\mathsf{max}(q)$. Wlog we assume M to be deterministic. By Lemma 5, we can compute some automaton C for trees from T_Σ such that $\mathsf{succ\_runs}_C(t) = \{r{*}\beta \mid r \in \mathsf{succ\_runs}_M(t{*}\beta)\}$. As M is deterministic, $\mathsf{succ\_runs}_M(t * \beta)$ contains at most one element. Moreover, there is a unique β such that $t * \beta$ is accepted by M for any tree $t \in T_\Sigma$ (by Proposition 3). Therefore, $\mathsf{succ\_runs}_C(t)$ is singleton set.

Now cnsider the query q such that there exists Cartesian closed regular queries $q_1, \ldots, q_k$ verifying for all $t \in T_\Sigma$, $q(t) = \bigcup_{j=1}^k q_j(t)$. We know by Proposition 4 that for each q_j there exists some pair (A_j, S_j) such that $q_j = \mathsf{query}_{A_j,S_j}$ and A_j is unambiguous. The rest of the proof is essentially the computation of the product of the A_j's. But, it must be done carefully to preserve the unambiguity of the result **JN:** unambiguity is not the problem, we have to ensure runs of A if there exists runs in some A_i: we consider $\bar{A}_i$ the deterministic automaton accepting the trees not accepted by A_i; assuming A_i and $\bar{A}_i$ have disjoint sets of states, we define A'_i as $A_i \cup \bar{A}_i$. This automaton A'_i is unambiguous and moreover, $\mathsf{query}^{\exists}_{A'_i,S_i} = \mathsf{query}^{\exists}_{A_i,S_i}$. For q, we define the automaton A as the product of the A'_i's and let $\mathsf{final}(A) = \mathsf{final}(A'_1) \times \ldots \times \mathsf{final}(A'_k)$. Note that A is unambiguous. We define S as the set of all $(p_1, \ldots, p_n) \in \mathsf{states}(A)^n$ for which there exists i $(1 \leq i \leq k)$ such that $(\mathsf{proj}_i(p_1), \ldots, \mathsf{proj}_i(p_n)) \in S_i$ (where $\mathsf{proj}_i(p)$ is the ith component of the state p). The two queries q and $\mathsf{query}^{\exists}_{A,S}$ are equivalent. This completes the proof of Theorem 4.

Proposition 5. *A query q is unambiguous iff it can be expressed as Boolean combinations of monadic MSO formulas (i.e., regular monadic queries)*

4.2 Deciding unambiguity of queries

We show in this section that one can decide whether a regular n-ary query is unambiguous, or equivalently by Theorem 4 whether the query is a finite union of Cartesian closed regular queries.

Note that deciding whether a regular query is Cartesian closed is straightforward using relationship between regular queries and MSO formulas and using that the Cartesian closed property is MSO-definable. Considering finite unions of Cartesian closed regular queries requires more sophisticated technique.

Theorem 5. *The problem whether an n-ary regular query is a finite union of Cartesian closed regular queries is decidable.*

We say that a tree automaton A is k-ambiguous if for any tree $t \in T_\Sigma$, there exists at most k accepting runs for t in A. An automaton A has a bounded ambiguity if there exists some natural k such that A is k-ambiguous.

Theorem 6. *[21] The boundedness of ambiguity is decidable for tree automata.*

Proposition 6. *For any query q, if there exists some pair (A, S) such that $q = \mathsf{query}^{\exists}_{A,S}$ and A has a bounded ambiguity then q is a finite union of Cartesian closed queries.*

Proof. By definition, we have $q(t) = \bigcup_{s \in S} q_s(t)$ for any $t \in T_\Sigma$ and $q_s \in \mathsf{query}^{\exists}_{A,\{s\}}$. Now, for any tree t and any accepting run for t in A, we define $q^s_r(t)$ as $\{(\pi_1, \ldots, \pi_n) \mid (r(\pi_1), \ldots, r(\pi_n)) = s\}$. Therefore, $q_s(t) = \bigcup_{r \in \mathsf{succ\_runs}_A(t)} q^s_r(t)$. As there exists some natural k such that the cardinality of $\mathsf{succ\_runs}_A(t)$ is smaller than k for any t, we have indeed that q is a finite union of Cartesian closed queries.

Proposition 7. *For any regular query q, if q is a finite union of Cartesian closed queries then there exists some natural l such that for all trees $t \in T_\Sigma$, the cardinality of the set $\{t * \beta \mid t * \beta \in \mathsf{max}(q)\}$ is upper-bounded by l.*

Proof. The query q being a finite union of Cartesian closed queries, there exists some natural k s.t. $q = \bigcup_{j=1}^{k} q^1_j \times \ldots \times q^n_j$, each q^i_j being a monadic regular query.

Let t be a tree from T_Σ. For each i, $1 \leq i \leq n$, we define $\equiv_i$, an equivalence relation on $\mathsf{dom}(t)$ by $\pi \equiv_i \pi'$ if for all $(\pi_1, \ldots, \pi_{i-1}, \pi_{i+1}, \ldots, \pi_n)$, $(\pi_1, \ldots, \pi_{i-1}, \pi, \pi_{i+1}, \ldots, \pi_n)$ belongs to $q(t)$ iff $(\pi_1, \ldots, \pi_{i-1}, \pi', \pi_{i+1}, \ldots, \pi_n)$ belongs to $q(t)$. This just means that π and π' are, in some sense, interchangeable in ith position. Then, let π and π' be two nodes. If for each j, $1 \leq j \leq k$, π belongs to $q^i_j(t)$ iff π' belongs to $q^i_j(t)$, then $\pi \equiv_i \pi'$. This implies that $\equiv_i$ is of finite index bounded by 2^k. Now let $t * \beta$ be a term in $\mathsf{max}(q)$. Let π selected in the ith position by β i.e. such that $\beta(\pi)_i = 1$. Then, by maximality of $t * \beta$, for each π' s.t. $\pi \equiv_i \pi'$, we have also $\beta(\pi')_i = 1$. This implies that $\{\pi \mid \beta(\pi)_i = 1\}$ is an union of equivalence classes for $\equiv_i$. So, the cardinality of the set $\{t * \beta \mid t * \beta \in \mathsf{max}(q)\}$ is upper-bounded by $2^{n.2^k}$.

Proposition 8. *For any regular query q, if there exists some natural l such that for all trees $t \in T_\Sigma$, the cardinality of the set $\{t * \beta \mid t * \beta \in \mathsf{max}(q)\}$ is upper-bounded by l then there exists some k-ambiguous automaton A and some set S of tuples of states such that $q = \mathsf{query}^{\exists}_{A,S}$.*

Proof. The proof is similar to that of Proposition 4 and relies on the construction defined in the proof of Lemma 5.

Theorem 5 easily follows from Theorem 6 and Propositions 6, 7 and 8. Moreover, we can show that

Corollary 1. *Let q be a regular query. Then the two following statements are equivalent:* (1) *q is a finite union of Cartesian closed queries.* (2) *q is a finite union of Cartesian closed* regular *queries.*

Proof. Obviously, (2) implies (1). We know by Propositions 6, 7 and 8 that q is a finite union of Cartesian closed queries iff there exists some natural l such that for all trees $t \in T_\Sigma$, the cardinality of the set $\{t \mid t * \beta \in \mathsf{max}(q)\}$ is upper-bounded by l. So, it suffices to prove that this latter implies that q is a finite union of Cartesian closed regular queries. We recall that q being regular, one can compute the automaton $\mathsf{max}(q)$. We define a total strict ordering $\prec$ on trees from $T_{\Sigma\times\mathbb{B}^n}$ as follows: $t * \beta \prec t * \beta'$ if (i) $\beta(\epsilon) < \beta'(\epsilon)$, or (ii) $\beta(\epsilon) = \beta'(\epsilon)$ and $t_1 * \beta_1 \prec t_1 * \beta'_1$ or (iii) $\beta(\epsilon) = \beta'(\epsilon)$, $t_1 * \beta_1 = t_1 * \beta'_1$ and $t_2 * \beta_2 \prec t_2 * \beta'_2$ (where $t_j * \beta_j$ is the subtree at position j of $t * \beta$). Obviously, $\prec$ is a recognizable relation of $T_{\Sigma\times\mathbb{B}^n} \times T_{\Sigma\times\mathbb{B}^n}$. Using ideas similar to those from proof of Lemma 7 we can compute automata for S_1, the set of greatest elements in the sense of $\prec$ from $\mathsf{max}(q)$ and for R_1 as $\mathsf{max}(q) - S_1$. Obviously, S_1 contains at most one tree $t * \beta$ for each $t \in T_\Sigma$. Therefore, the regular query q_1 defined as $q_1(t) = \mathsf{tuples}(t * \beta)$ for the unique $t * \beta$ in S_1, is a Cartesian closed query. We iterate the same process l times, extracting each time a Cartesian closed regular query q_j. It is straightforward that q is the union of all these q_j's.

4.3 Construction of unambiguous automata

We next give a direct construction of unambiguous automata for Cartesian closed run based queries $\mathsf{query}^{\exists}_{A,S}$. The idea of the construction is that of the two way querying answering algorithm. Note that the existence unambiguous automata follows already from the more general Theorem 4. The direct construction might be of interest nevertheless, if one wants to analyse the size of resulting unambiguous automaton.

We first compute a deterministic automaton $\mathsf{det}(A)$ that can perform all runs of A at once:

$$\mathsf{states}(\mathsf{det}(A)) = 2^{\mathsf{states}(A)}$$
$$\mathsf{final}(\mathsf{det}(A)) = \{P \mid P \cap \mathsf{final}(A) \neq \emptyset\}$$

$$P = \{p \mid a \to p \in \mathsf{rules}(A) a \to P \in \mathsf{rules}(\mathsf{det}(A))$$

$$\frac{P = \{p \mid f(p_1, p_2) \to p \in \mathsf{rules}(A), p_1 \in P_1, p_2 \in P_2\}}{f(P_1, P_2) \to P \in \mathsf{rules}(\mathsf{det}(A))}$$

Proposition 9. *For every run* $r \in \mathsf{runs}_{\mathsf{det}(A)}(t)$ *and node* $\pi \in \mathsf{nodes}(t)$*:* $r(\pi) = \{r'(\pi) \mid r' \in \mathsf{runs}_A(t)\}$.

We next compute an unambiguous automaton $\mathsf{u}(A)$ that for all trees t computes all successful runs of A on t at once:

$$\mathsf{states}(\mathsf{u}(A)) = \{(P', P) \mid P' \subseteq P, P \in \mathsf{states}(\mathsf{det}(A))\}$$

$$\mathsf{final}(\mathsf{u}(A)) = \{(P \cap \mathsf{final}(A), P) \mid P \in \mathsf{final}(\mathsf{det}(A))\}$$

$$\frac{a \to P \in \mathsf{rules}(\mathsf{det}(A)) \qquad P' \subseteq P}{a \to (P', P) \in \mathsf{rules}(\mathsf{u}(A))}$$

$$\frac{\begin{array}{c} f(P_1, P_2) \to P \in \mathsf{det}(A) \\ P_1' = \{p_1 \in P_1 \mid p_2 \in P_2, p' \in P', f(p_1, p_2) \to p' \in \mathsf{rules}(A)\} \\ P_2' = \{p_2 \in P_2 \mid p_1 \in P_1, p' \in P', f(p_1, p_2) \to p' \in \mathsf{rules}(A)\} \end{array}}{f((P_1', P_1), (P_2', P_2)) \to (P', P) \in \mathsf{rules}(\mathsf{u}(A))}$$

Proposition 10. *For all trees* t*, automata* A*, runs* $r \in \mathsf{succ\_runs}_{\mathsf{u}(A)}(t)$*, and nodes* $\pi \in \mathsf{nodes}(t)$*:*

$$r(\pi) = (\{r'(\pi) \mid r' \in \mathsf{succ\_runs}_A(t)\},\ \{r'(\pi) \mid r' \in \mathsf{runs}_A(t)\})$$

In particular, $\mathsf{u}(A)$ *is unambiguous.*

Theorem 7. *Cartesian closed* n*-ary queries* $\mathsf{query}^{\exists}_{A,S}$ *can be expressed by unambiguous tree automata* $\mathsf{u}(A)$*; if* $S_i = \{p_i \mid (p_1, \ldots, p_n) \in S\}$ *for all* $1 \leq i \leq n$ *then:*

$$\mathsf{query}^{\exists}_{A,S} = \mathsf{query}^{\exists}_{\mathsf{u}(A),S_1} \times \ldots \times \mathsf{query}^{\exists}_{\mathsf{u}(A),S_n}$$

5 Query Answering

We next show that we can answer n-ary queries by runs of unambiguous automata in linear time combined complexity. This can be done by extending the well-known two phase query answering algorithm.

The general problem seems to have a higher computational complexity. The data complexity of unrestricted run-based n-ary queries is remains polynomial for fixed $n \geq 0$ but the combined complexity seems to be higher. To see the polynomial bound on the data complexity, we fix an automaton A over Σ. The inputs are a set of selection tuples $S \subseteq \mathsf{nodes}(A)^n$ and a tree $t \in T_\Sigma$. We compute an automaton A' from A and S that recognizes the canonical language of $\mathsf{query}^{\exists}_{A,S}$. We then test for all tuples $(\pi_1, \ldots, \pi_n) \in \mathsf{nodes}(t)^n$ whether $t * \mathsf{c}^t_{\pi_1} * \ldots \mathsf{c}^t_{\pi_1} \in L(A')$. If we assume the size on A to be bounded by some constant then each of this tests will be in linear time $O(|t|)$. So the overall data complexity is bounded $O(|t|)^{n+1}$.

Theorem 8. *The combined complexity for answering* n*-ary queries in trees by unambiguous tree automata is linear.*

We only sketch the proof. Given a tree t and $\mathsf{query}^{\exists}_{A,S}$ for some unambiguous automaton A. We collect the answers of all queries $\mathsf{query}^{\exists}_{A,\{\vec{p}\}}$ where $\vec{p} \in S$. These queries are Cartesian closed so that Theorem 7 applies. We compute the unique run of $\mathsf{u}(A)$ on t in two phases, first first bottom up then top down. This can be done in time $O(|t|*|A|)$. We want to return: $\mathsf{query}^{\exists}_{\mathsf{u}(A),S_1} \times \ldots \times \mathsf{query}^{\exists}_{\mathsf{u}(A),S_n}$ where $S_i = \{p_i \mid (p_1, \ldots, p_n) \in S\}$ for all $1 \leq i \leq n$. In order to avoid a polynomial blow up, we simply return all individual queries $\mathsf{query}^{\exists}_{\mathsf{u}(A),S_i}$ rather then multiplying them out. Each of these queries can be computed in time $O(|t|)$ by inspecting the unique run of $\mathsf{u}(A)$ on t if it exists.

6 Querying unranked trees

Our results carry over to unranked trees by hedge automata [6]. An unranked tree is build from a set of constants $a, b \in \Sigma$ by the abstract syntax $t ::= a(t_1, \ldots, t_n)$ where $n \geq 0$. A *hedge automaton* H over Σ consists of a set $\mathsf{states}(H)$, a set $\mathsf{final}(H) \subseteq \mathsf{states}(H)$, and a set $\mathsf{rules}(H)$ of rules of the form $a(A) \rightarrow p$ where A is finite word automaton with alphabet $\mathsf{states}(H)$ and $p \in \mathsf{states}(H)$. Runs of hedge automata H on unranked trees t are functions $r : \mathsf{nodes}(t) \rightarrow \mathsf{states}(H)$ defined as

$$\frac{\begin{array}{ll} t = a(t_1, \ldots, t_n) & \forall 1 \leq i \leq n : r_i \in \mathsf{runs}_H(t_i) \\ a(A) \rightarrow p \in \mathsf{rules}(H) & r_1(\epsilon) \ldots r_n(\epsilon) \in L(A) \end{array}}{p(r_1, \ldots, r_n) \in \mathsf{runs}_H(t)}$$

Queries for the class of unranked trees over Σ are defined as before. The notion of unambiguity (that is the existence of at most one run for a tree) carries over literally to hedge automata (in contrast to determinism). The same holds for the notions of run-based queries by hedge automata.

Theorem 9. *Existential and universal n-ary queries in unranked trees by runs of hedge automata capture* MSO *over run-ranked trees (comprising the* next_sibl*-relation). Run-based based queries by unambiguous hedge automata capture the class of finite unions of Cartesian closed queries. This property is decidable. Queries by unambiguous hedge automata have linear combined complexity.*

We only give a sketch of the proof. The main idea is to convert queries by hedge automata into queries by stepwise tree automata [8] for which all results apply. Stepwise tree automata over an unranked signature Σ are tree automata for binary trees with constants in Σ and a single binary function symbol @. Stepwise tree automata can be understood as tree automata that operate on Currified binary encodings of unranked trees. The Currification of $a(b, c(d, e, f), g)$ for instance is the binary tree $a@b@(c@d@e@f)@g$.

Stepwise tree automata were proved to have two nice properties that yield a simple proof of the theorem. 1) N-ary queries by hedge automata can be translated to n-ary queries by stepwise automata in linear time, and conversely in polynomial time. The back and forth translations preserve unambiguity. 2) All presented results on run-based n-ary queries for binary trees apply to stepwise tree automata.

References

1. XML Path Language (XPath): W3C Recommendation, 1999.
2. XML Pointer Language Version 1.0 (Xpointer): W3C Candidate Recommendation, 2001.
3. R. Baumgartner, S. Flesca, and G. Gottlob. Visual Web Information Extraction with Lixto. In *The VLDB Journal*, pages 119–128, 2001.
4. A. Berlea and H. Seidl. Binary queries for document trees. *Nordic Journal of Computing*, 11(1):41–71, 2004.
5. R. Bloem and J. Engelfriet. A comparison of tree transductions defined by monadic second order logic and by attribute grammars. *JCSS*, 61(1):1–50, 2000.
6. A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets, 2001. Unpublished.
7. J. Carme, A. Lemay, and J. Niehren. Learning node selecting tree transducer from completely annotated examples. In *7th International Conference on Grammatical Inference*, LNAI. 2004.
8. J. Carme, J. Niehren, and M. Tommasi. Querying unranked trees with stepwise tree automata. In *RTA*, volume 3091 of *LNCS*, pages 105 – 118. 2004.
9. W. W. Cohen, M. Hurst, and L. S. Jensen. A flexible learning system for wrapping tables and lists in HTML documents. In *Web document analysis: challanges and opportunities*. 2003.
10. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997.
11. M. Frick, M. Grohe, and C. Koch. Query evaluation on compressed trees. In *Proc. LICS 2003*, 2003.
12. G. Gottlob and C. Koch. Monadic datalog and the expressive power of languages for web information extraction. In *PODS*, pages 17–28, 2002.
13. G. Gottlob and C. Koch. Monadic queries over tree-structured data. In *LICS* 2002, IEEE Press.
14. G. Gottlob, C. Koch, R. Baumgartner, M. Herzog, and S. Flesca. The Lixto data extraction project - back and forth between theory and practice. In *ACM PODS*. 2004.
15. R. Kosala, J. V. den Bussche, M. Bruynooghe, and H. Blockeel. Information extraction in structured documents using tree automata induction. In *PKDD-2002*, pages 299 – 310, 2002.
16. M. Marx. Conditional XPath, the first order complete XPath dialect. In *ACM PODS*, pages 13–22. 2004.
17. A. Neumann and H. Seidl. Locating matches of tree patterns in forests. In *FSTTCS*, pages 134–145, 1998.
18. F. Neven and J. V. D. Bussche. Expressiveness of structured document query languages based on attribute grammars. *Journal of the ACM*, 49(1):56–100, 2002.
19. F. Neven and T. Schwentick. Query automata over finite trees. *TCS*, 275(1-2):633–674, 2002.
20. J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis*, pages 49–61, 1992.
21. H. Seidl. On the finite degree of ambiguity of finite tree automata. *Acta Informatica*, 26(6):527–542, 1989.
22. H. Seidl, T. Schwentick, and A. Muscholl. Numerical document queries. In *PODS*, pages 155–166. 2003.
23. J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical System Theory*, 2:57–82, 1968.

A Omitted proofs

A.1 On regular queries

Lemma 5. *Every regular n-ary query is equal to some existential run-based query.*

Proof. Let L be a regular language over $\Sigma\times\mathbb{B}^n$ and A be an automaton such that $L(A) = L$. We compute C, an automaton over Σ, and S a selection set such that $L(C) = \{t \mid t*\beta \in L(A)\}$ and $\mathsf{corr\_query}(L(A)) = \mathsf{query}^{\exists}_{C,S}$ as follows:

$$\frac{(a, b_1, ..., b_n)\to p \in \mathsf{rules}(A)}{a\to(p, b_1, ..., b_n) \in \mathsf{rules}(C)}$$

$$\frac{(f, b_1, ..., b_n)(p_1, p_2)\to p \in \mathsf{rules}(A)}{(f(p_1, b_1^1, ...b_1^n), (p_2, b_2^1, ..., b_2^n))\to(p, b_1, ..., b_n) \in \mathsf{rules}(C)}$$

Finally, let $\mathsf{states}(C) = \mathsf{states}(A)\times\mathbb{B}^n$, $\mathsf{final}(C) = \mathsf{final}(A)\times\mathbb{B}^n$ and $S = P_1 \times ... \times P_n$ where for all $1 \leq i \leq n$:

$$P_i = \{(p, b_1, ..., b_n) \in \mathsf{states}(C) \mid b_i = 1\}$$

The correctness of the construction is proved by showing that for any term t, $\mathsf{runs}_C(t) = \{r*\beta \mid r \in \mathsf{runs}_A(t*\beta)\}$ and $\mathsf{succ\_runs}_C(t) = \{r*\beta \mid r \in \mathsf{succ\_runs}_A(t*\beta)\}$. First, we prove by induction that (a) if $r*\beta \in \mathsf{runs}_C(t)$, then $r \in \mathsf{runs}_A(t*\beta)$. If t is a constant, then it is obvious. Let $t = f(t_1, t_2)$. Suppose (a) is true for t_1 and t_2. Let $(p, \boldsymbol{b})(r_1*\beta_1, r_2*\beta_2) \in \mathsf{runs}_C(t)$. So, there is rule such that $f(r_1*\beta_1(\epsilon), r_2*\beta_2(\epsilon))\to(p, \boldsymbol{b}) \in \mathsf{rules}(C)$. By definition of our automaton, $(f, \boldsymbol{b})(r_1(\epsilon), r_2(\epsilon))\to p \in \mathsf{rules}(A)$. By hypothesis, $r_1 \in \mathsf{runs}_A(t_1*\beta_1)$ and $r_2 \in \mathsf{runs}_A(t_2*\beta_2)$. Then, we have $p(r_1, r_2) \in \mathsf{runs}_A(t)$.

We now prove by induction that (b) if $r \in \mathsf{runs}_A(t*\beta)$, then $r*\beta \in \mathsf{runs}_C(t)$. If t is a constant, then it is obvious. Let $t = f(t_1, t_2)$. Suppose (b) is true for t_1 and t_2. Let $p(r_1, r_2) =\in \mathsf{runs}_A((f, \boldsymbol{b})(t_1*\beta_1, t_2*\beta_2))$. So, there is a rule such that $(f, \boldsymbol{b})(r_1(\epsilon), r_2(\epsilon))\to p \in \mathsf{rules}(A)$. By definition of our automaton, $f(r_1*\beta_1(\epsilon), r_2*\beta_2(\epsilon))\to(p, \boldsymbol{b}) \in \mathsf{rules}(C)$. By hypothesis, $r_1*\beta_1 \in \mathsf{runs}_C(t_1)$ and $r_2*\beta_2 \in \mathsf{runs}_C(t_2)$. Then, we have $(p, \boldsymbol{b})(r_1*\beta_1, r_2*\beta_2)$.

As $\mathsf{runs}_C(t) = \{r*\beta \mid r \in \mathsf{runs}_A(t*\beta)\}$, then :

$$\begin{array}{ll}
\mathsf{succ_runs}_C(t) & = \\
\{r*\beta \mid r*\beta \in \mathsf{runs}_C(t), r*\beta(\epsilon) \in \mathsf{final}(C)\} & = \\
\{r*\beta \mid r \in \mathsf{runs}_A(t*\beta), r*\beta(\epsilon) \in \mathsf{final}(A)\times\mathbb{B}^n\} & = \\
\{r*\beta \mid r \in \mathsf{runs}_A(t), r(\epsilon) \in \mathsf{final}(A)\} & = \\
\{r*\beta \mid r \in \mathsf{succ_runs}_A(t)\} &
\end{array}$$

A.2 On construction of unambiguous automata

We prove Proposition 10.

Lemma 8. *For all trees t, automata A, and runs $r \in \mathsf{runs}_{\mathsf{u}(A)}(t)$, if $r(\epsilon) = (P', P)$ and $p' \in P'$ then there exists a run $r' \in \mathsf{runs}_A(t)$ with $r'(\epsilon) = p'$.*

Proof. By induction on the structure of trees t. If $t = a$, then $P' \subseteq P$ so that $a \to P \in \mathsf{det}(A)$. Hence, $a \to p \in \mathsf{rules}(A)$ for all $p \in P'$, and thus $a \to p' \in \mathsf{rules}(A)$, i.e., the function $r : \{\epsilon\} \to \mathsf{states}(A)$ with $r(\epsilon) = p'$ is in $\mathsf{runs}_A(t)$.

If $t = f(t_1, t_2)$, then there exist runs of $r_1 \in \mathsf{runs}_{\mathsf{u}(A)}(t_1)$ and $r_2 \in \mathsf{runs}_{\mathsf{u}(A)}(t_2)$, with

$$f(r_1(\epsilon), r_2(\epsilon)) \to r(\epsilon) \in \mathsf{rules}(\mathsf{u}(A))$$

Let $(P'_1, P_1) = r_1(\epsilon)$ and $(P'_2, P_2) = r_2(\epsilon)$. We have $f(P_1, P_2) \to P \in \mathsf{rules}(\mathsf{det}(A))$, so since $p' \in P$ there exist $f(p_1, p_2) \to p' \in \mathsf{rules}(A)$ for some states $p_1 \in P_1$ and $p_2 \in P_2$. By definition of P'_1 and P'_2 this yields $p_1 \in P'_1$ and $p_2 \in P'_2$. By induction hypothesis, there are runs $r'_1 \in \mathsf{runs}_A(t_1)$ and $r'_2 \in \mathsf{runs}_A(t_2)$ with $r'(\epsilon) = p_1$ and $r'_2(\epsilon) = p_2$. These can be composed into a run $r' \in \mathsf{runs}_A(t)$ with $r'(\epsilon) = p'$.

Lemma 9. *For all trees t, automata A, $\pi \in \mathsf{nodes}(t)$, and runs $r \in \mathsf{succ\_runs}_{\mathsf{u}(A)}(t)$: if $r(\pi) = (P', P)$ and $p' \in P'$ then there exists a successful run $r' \in \mathsf{succ\_runs}_A(t)$ with $r'(\pi) = p'$.*

Proof. By induction on the depth of trees π.

Case $\pi = \epsilon$. The success of r is equivalent to $r(\epsilon) \in \mathsf{final}(\mathsf{u}(A))$, and thus, $P' = P \cap \mathsf{final}(A) \neq \emptyset$. Thus, all states $p' \in P'$ satisfy $p' \in \mathsf{final}(A)$ and there exists such a state. By Lemma 8 there exists a run $r' \in \mathsf{runs}_A(t)$ with $r(\epsilon) = p'$. This run is successful.

Case $\pi = \tilde{\pi}1$. Let $\tilde{t}$ be subtree of t at node $\tilde{\pi}$. The rule of $\mathsf{u}(A)$ that justifies $r(\tilde{\pi})$ has the form:

$$f((P', P), r(\tilde{\pi}2) \to r(\tilde{\pi})$$

Let $(\tilde{P}', \tilde{P}) = r(\tilde{\pi})$ and $(P'_2, P_2) = r(\tilde{\pi}2)$. By definition of P' in $\mathsf{u}(A)$ there exist states $\tilde{p}' \in \tilde{P}'$ and $p_2 \in P_2$ such that

$$f(p', p_2) \to \tilde{p}' \in \mathsf{rules}(A)$$

By induction hypothesis, there exists $\tilde{r} \in \mathsf{succ\_runs}_A(t)$ with $\tilde{r}(\tilde{\pi}) = \tilde{p}'$ Let $f(\tilde{t}_1, \tilde{t}_2) = \tilde{(t)}$. Lemma 9 yields the existence of a run r_2 in $\mathsf{runs}_A(\tilde{t}_2)$ with $r_2(\epsilon) = p_2$. Since $p' \in P'$ we can apply Lemma 8; it proves the existence of a run r_1 in $\mathsf{runs}_A(\tilde{t}_1)$ with $r_1(\epsilon) = p'$. We can now compose the runs $\tilde{r}$, r_1, and r_2 into a successful run $r' \in \mathsf{succ\_runs}_A(t)$ which satisfies $r'(\pi) = r_1(\epsilon) = p'$.

The case $\pi = \tilde{\pi}2$ is symmetric.

Proposition 10. *For all trees t, automata A, runs $r \in \mathsf{succ\_runs}_{\mathsf{u}(A)}(t)$, and nodes $\pi \in \mathsf{nodes}(t)$:*

$$r(\pi) = (\{r'(\pi) \mid r' \in \mathsf{succ\_runs}_A(t)\},\ \{r'(\pi) \mid r' \in \mathsf{runs}_A(t)\})$$

Proof. Let $(P, P') = r(\pi)$. Proposition 9 yields $P = \{r'(\pi) \mid r' \in \mathsf{runs}_A(t)\}$. The preceding Lemma 9 proves $P' \subseteq \{r'(\pi) \mid r' \in \mathsf{succ\_runs}_A(t)\}$. It remains to show for all $\pi \in \mathsf{nodes}(t)$ that:

$$\{r'(\pi) \mid r' \in \mathsf{succ\_runs}_A(t)\} \subseteq P'$$

We fix $r' \in \mathsf{succ\_runs}_A(t)$ and prove $r'(\pi) \in P'$ by induction on paths π.

Case $\pi = \epsilon$. By Proposition 9, $r'(\epsilon) \in P$. Since r' is successful, $r'(\epsilon) \in \mathsf{final}(A)$. Since r is successful, $P' = P \cap \mathsf{final}(A)$ and thus $r'(\epsilon) \in P'$.

Case $\pi = \tilde{\pi}1$. Let $(\tilde{P}', P) = r(\tilde{\pi})$ and $(P'_2, P_2) = r(\tilde{\pi}2)$. By induction hypothesis, we know that $r'(\tilde{\pi}) \in \tilde{P}'$. Furthermore, runs satisfy:

$$f(r'(\pi), r'(\tilde{\pi}2)) \to r'(\tilde{\pi}) \in \mathsf{rules}(A)$$

Proposition 9 yields $r'(\tilde{\pi}2) \in P_2$ and $r'(\pi) \in P$. The definition of $\mathsf{u}(A)$ implies $r'(\pi) \in P'$.

The case $\pi = \tilde{\pi}2$ is analogous.

Lemma 10. *For all A and trees t, $\mathsf{succ\_runs}_A(t) = \emptyset$ iff $\mathsf{succ\_runs}_{\mathsf{u}(A)}(t) = \emptyset$.*

Proof. If $\mathsf{succ\_runs}_{\mathsf{u}(A)}(t) = \emptyset$ then there exists a run $r\mathsf{succ\_runs}_{\mathsf{u}(A)}(t)$ with $r(\epsilon) \in \mathsf{final}(A)$. By Proposition 10 this means that:

$$\begin{aligned}\{r'(\pi) \mid r' \in \mathsf{succ\_runs}_A(t)\} = \{&r'(\pi) \mid r' \in \mathsf{runs}_A(t)\})\cap \\ &\mathsf{final}(A) \neq \emptyset\end{aligned}$$

Conversely, if $\mathsf{succ\_runs}_A(t) = \emptyset$ the we can define a successful run of $r \in \mathsf{succ\_runs}_{una(A)}(t)$ by requiring for all nodes $\pi \in \mathsf{nodes}(t)$:

$$r(\pi) = (\{r'(\pi) \mid r' \in \mathsf{succ\_runs}_A(t)\},\ \{r'(\pi) \mid r' \in \mathsf{runs}_A(t)\})$$

It remains to verify, that r is indeed a successful run of $\mathsf{u}(A)$ on t.

Unification with Expansion Variables: Preliminary Results and Problems*

Adam Bakewell and Assaf J. Kfoury

Department of Computer Science, Boston University
{bake,kfoury}@cs.bu.edu

Abstract. Expansion generalises substitution. An *expansion* is a special term whose leaves can be substitutions. Substitutions map term variables to ordinary terms and *expansion variables* to expansions. Expansions (resp., ordinary terms) may contain expansion variables, each applied to an argument expansion (resp., ordinary term). Unification instances in this setting are constraint sets, where constraints are pairs of ordinary terms and unifiers are expansions. We present a research agenda, a methodology for tackling problems in this setting, and several preliminary results.

1 Background and Motivation

The study of unification with expansion variables has both practical and theoretical ramifications. The motivation comes from type systems for the lambda-calculus. The theoretical framework gives rise to a host of problems whose solutions require appropriately adapted algebraic and combinatorial techniques.

The key novelty in our framework is the concept of an *expansion variable*. We start right off with a brief tutorial on expansion concepts in Section 1.1. The connection with type systems is briefly discussed in Section 1.2. The scope and organization of the paper are presented in Sections 1.3 and 1.4.

1.1 Expansion concepts

An *expansion* generalizes the notion of a substitution. It is a term whose leaves can be substitutions. For example, an expansion called E_1 may be defined by[1]

$$\mathsf{E}_1 = (\{\mathsf{a}_1 \mapsto \mathsf{a}_1 \otimes \mathsf{a}_1\} \otimes \mathsf{a}_2) \otimes \{\mathsf{a}_1 \mapsto \mathsf{a}_3\}$$

where $\otimes$ is a binary term constructor and each a_i is a term variable (T-variable). Replacing each substitution leaf in E_1 by the identity substitution, we obtain the *term part* of E_1, namely $(\{\} \otimes \mathsf{a}_2) \otimes \{\}$. Applying the expansion E_1 to a T-variable, say a_1, written $[\mathsf{E}_1]\,\mathsf{a}_1$, results in the term part of E_1 with each substitution leaf S replaced by $S(\mathsf{a}_1)$. Thus,

$$[\mathsf{E}_1]\,\mathsf{a}_1 = ((\mathsf{a}_1 \otimes \mathsf{a}_1) \otimes \mathsf{a}_2) \otimes \mathsf{a}_3.$$

* Work partly funded by NSF grant CCR-0113193 *Implementing Modular Program Analysis via Intersection and Union Types*.

[1] The notation $\{\mathsf{a}_1 \mapsto \mathsf{a}_1 \otimes \mathsf{a}_1\}$ defines the support of a total function f; i.e. $f(x) = x$ for all x except a_1 where $f(\mathsf{a}_1) = \mathsf{a}_1 \otimes \mathsf{a}_1$. The identity substitution is therefore the function whose support is $\{\}$, the empty set.

Applying the expansion E_1 to a plain term τ_1 (without expansion variables) results in the term part of E_1 with each substitution leaf S replaced by $S(\tau_1)$. For example, if $\tau_1 = (\mathsf{a}_1 \otimes \mathsf{a}_1)$, then

$$[\mathsf{E}_1]\,\tau_1 = [\mathsf{E}_1]\,(\mathsf{a}_1 \otimes \mathsf{a}_1) = (((\mathsf{a}_1 \otimes \mathsf{a}_1) \otimes (\mathsf{a}_1 \otimes \mathsf{a}_1)) \otimes \mathsf{a}_2) \otimes (\mathsf{a}_3 \otimes \mathsf{a}_3).$$

An *expansion variable* (E-variable) e, then, is a kind of function variable. Occurrences in terms are always applied to one argument. We say that E-variable e *wraps* its argument; the *namespace* of a subterm is the sequence of E-variables encountered on the path to it from the root of the term; and e occurs *outermost* if its namespace is empty. For example, in

$$\tau_2 = \mathsf{e}_1\,(\mathsf{a}_1 \otimes \mathsf{e}_2\,\mathsf{a}_1),$$

E-variable e_1 occurs outermost; e_2 is in the namespace e_1 and T-variable a_1 has occurrences in the namespaces e_1 and $\mathsf{e}_1 \cdot \mathsf{e}_2$. Substitutions are extended to map E-variables to expansions as well as T-variables to terms. Substitution application is defined such that only the outermost variables of the argument are affected. For a more involved example, consider the following expansion E_2:

$$\begin{aligned} &\mathsf{E}_2 = (\{\} \otimes \mathsf{S}_1) \otimes \mathsf{S}_2 \qquad \text{where} \\ &\mathsf{S}_1 = \{\mathsf{a}_1 \mapsto \mathsf{a}_4, \mathsf{e}_2 \mapsto \mathsf{E}_1\}, \\ &\mathsf{S}_2 = \{\mathsf{a}_1 \mapsto \mathsf{a}_5, \mathsf{e}_2 \mapsto \mathsf{e}_3\,\{\mathsf{a}_1 \mapsto \mathsf{a}_6\}\} \end{aligned}$$

and E_1 was defined previously. The next expansion application shows how expansions can apply different substitutions to the same variable in different namespaces: Outermost variable e_1 becomes E_2; at the leaves of E_2, substitutions S_1 and S_2 are applied; then under e_2 in the substitution S_1, expansion E_1 is applied:

$$\begin{aligned} [\{\mathsf{e}_1 \mapsto \mathsf{E}_2\}]\ \tau_2 &= [\mathsf{E}_2]\,(\mathsf{a}_1 \otimes \mathsf{e}_2\,\mathsf{a}_1) \\ &= ((\mathsf{a}_1 \otimes \mathsf{e}_2\,\mathsf{a}_1) \otimes [\mathsf{S}_1]\,(\mathsf{a}_1 \otimes \mathsf{e}_2\,\mathsf{a}_1)) \otimes [\mathsf{S}_2]\,(\mathsf{a}_1 \otimes \mathsf{e}_2\,\mathsf{a}_1) \\ &= ((\mathsf{a}_1 \otimes \mathsf{e}_2\,\mathsf{a}_1) \otimes (\mathsf{a}_4 \otimes [\mathsf{E}_1]\,\mathsf{a}_1)) \otimes (\mathsf{a}_5 \otimes [\mathsf{e}_3\,\{\mathsf{a}_1 \mapsto \mathsf{a}_6\}]\,\mathsf{a}_1) \\ &= ((\mathsf{a}_1 \otimes \mathsf{e}_2\,\mathsf{a}_1) \otimes (\mathsf{a}_4 \otimes (((\mathsf{a}_1 \otimes \mathsf{a}_1) \otimes \mathsf{a}_2) \otimes \mathsf{a}_3))) \otimes (\mathsf{a}_5 \otimes \mathsf{e}_3\,\mathsf{a}_6) \end{aligned}$$

This layering created by expansion variables is very useful because it makes the control of variable name disjointness or equality easier.

A graphical summary of the expansions and terms used in this section is in Figure 1. Edges in expansions, as well as edges in terms inherited from applying an expansion, are shown in boldface in Figure 1. Although the examples in this section do not show it, E-variables can occur anywhere in terms, not only at the root or at the leaves (as in term τ_2 above).

In the setting just described, a *unifier* of two terms τ and τ' containing E-variables is an expansion E such that $[E]\,\tau = [E]\,\tau'$. Note that standard *first-order unification* is a special case, when the terms τ and τ' contain no E-variables; in this case, a unifier of τ and τ' is a first-order substitution, and therefore trivially an expansion. There is an obvious resemblance between E-variables here and functional variables in *second-order unification*, but the two are different and potential relationships between them are yet to be worked out.

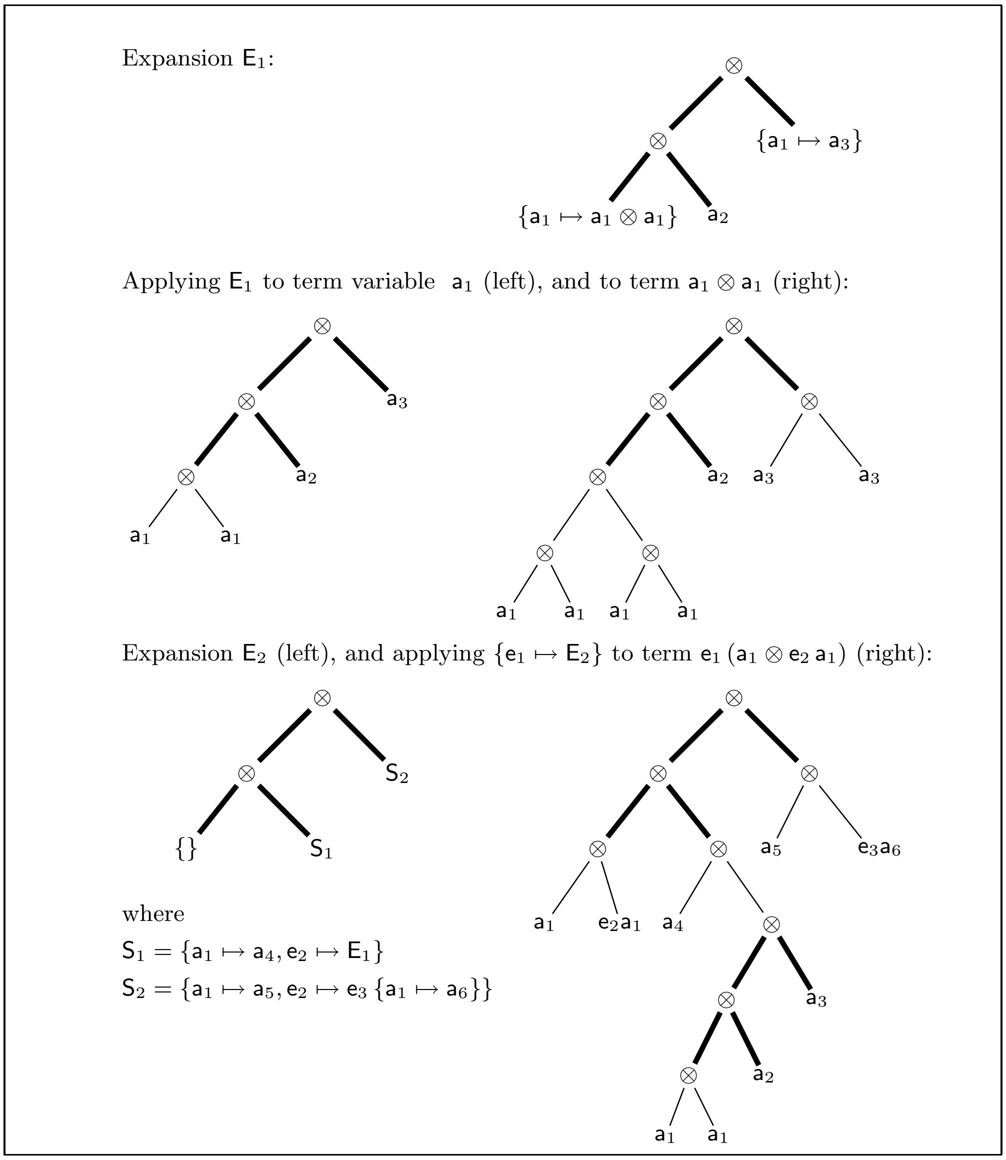

Fig. 1. Pictorial tutorial on expansion concepts (see Section 1.1).

1.2 Developments that led to unification with expansion variables

Unification of terms with expansion variables in the general sense introduced above is an interesting theoretical problem, but there is a lack of practical motivation for the general case in the absence of applications (at least so far). Moreover, desirable properties for unification such as the existence of unifiers, principal unifiers, complete or confluent unification systems, etc., may or may not be achieved depending on the form of the general framework we choose.

We study a particular form of unification with expansion variables which has applications in *intersection type* systems. Henceforth, ordinary terms are called *types*, as we build terms using two binary type constructors, $\rightarrow$ and $\cap$, instead of one $\otimes$, and a special constant ω. Expansions include $\cap$ and a special constant Ω, but not $\rightarrow$.[2]

Beta-unification. The ω-*free* restriction of the problem we called *β-unification* in [9, 13] and other more recent reports. The name β-unification refers to a precise connection. There is a constraint set, i.e., an instance of unification with expansion variables, for every term of the lambda-calculus. A term is β-strongly normalizing iff the corresponding constraint set has a unifier. This relationship was expounded in [9], where a somewhat different form of expansion variables was introduced. A theoretical presentation of another important connection was described: Constraint-set solving reveals an intersection *typing* for the corresponding lambda-calculus term. System I, a system of intersection types with expansion variables for the lambda-calculus [13], gave a procedure for solving an early formulation of unification with expansion variables and applied it to infer *principal typings* (not just *principal types*). Principal typings are important for *compositional* program analysis.

Expansions and expansion variables are now being applied and developed in the framework of System E, a more recent system of intersection types with expansion variables [5]. This has a more sophisticated and intelligible formulation of expansions and expansion variables than System I.

Differences in formulation. There are many notational variants between the present formulation of unification with E-variables and the various earlier formulations based on System I in [9, 11, 10, 13], which are not important. The core algebraic setup is similar: Types and constraint were built in the same way (but without ω). The main difference is in expansions and their semantics. Substitution was not a case of expansion: The leaves of expansions were always *holes*, denoted $\square$. Applying a substitution S to an applied E-variable e, i.e., $[S]\,(e\,\tau)$, gave the term-part of $S(e)$ (as in the present formulation), but with the ith leaf of $S(e)$ replaced by $S(\langle\tau\rangle^i)$ where $\langle\tau\rangle^i$ creates a copy of τ with all the variables renamed according to a certain scheme. For example,

$$[\{\mathsf{e}_1 \mapsto \square \cap \square\}]\,(\mathsf{e}_1\,\mathsf{a}_1) = \mathsf{a}_{1\cdot 1} \cap \mathsf{a}_{1\cdot 2}$$

where $\mathsf{a}_{1\cdot 1}$ and $\mathsf{a}_{1\cdot 2}$ are renamings of a_1. To apply substitutions that affect deep namespaces it was necessary to pre-empt the renamings. Thus the following substitution achieved the same result as $\{\mathsf{e}_1 \mapsto \{\} \cap \{\}\}$ in the present formulation:

$$[\{\mathsf{e}_1 \mapsto \square, \mathsf{a}_{1\cdot 1} \mapsto \mathsf{a}_1, \mathsf{a}_{1\cdot 2} \mapsto \mathsf{a}_1\}]\,(\mathsf{e}_1\,\mathsf{a}_1) = \mathsf{a}_1 \cap \mathsf{a}_1$$

To support this approach there were restrictions on variable names. This makes translating constraints and their solutions between the present and earlier formulations of unification with expansion variables slightly complicated.

[2] The purpose of the special constants ω and Ω is explained in Section 2. In papers on typed lambda-calculi, the binary constructors $\wedge$ and $\cap$ are often used instead of our $\cap$. We prefer $\cap$ to avoid symbol overloading. We use $\cap$ and $\wedge$ to denote set intersection and logical conjunction, respectively. Contrary to standard uses of $\cap$ and $\wedge$, our $\cap$ is not always associative, commutative or idempotent.

The main result established for the formulation used in System I centres around the connection between β-unification, β-strong normalisation and typability in the system of intersection types. The restrictions on constraints indicated above, and other restrictions on substitutions designed to make it fit well with certain type systems,[3] make it difficult to transfer this result to the present formulation via a translation because there are admissible solutions in the present formulation that have no counterpart in System I. However, we are able to establish the same connections and results independently, see Section 4 and [3]. In short, what is presented here completely supersedes the formulation used in System I.

1.3 What This Paper Is Not

Apart from the historical connection just reviewed, this paper is not about type systems and the lambda-calculus. Nor is it about the duplication operation known as "expansion" in the earlier work on intersection-type systems [17, 15, 16]. Nor is it about "expansion variables" as defined and used in our own [9] or even in [10, 13]. A cursory reading of the forementioned papers makes clear the differences with the framework of this paper: The mechanisms we define and bring into play here, in order to formulate problems and resolve them, are not found in the forementioned papers.

But they are found in the more recent papers on System E [5, 6, 2], where they are also mixed with a wide range of issues related to type systems and the lambda-calculus. Our first goal here is to disconnect the concepts of *expansion* and *expansion variable* underlying the process of constraint solving from all other issues in System E. This separate examination facilitates the explanation of the key concepts, which are quite natural, and offer new ways of tackling the still unresolved problems.

1.4 Our Contribution and Organization of This Paper

Once we extract the unification problem from the rest of the System E framework (in Section 2), involving a separate formalization in its own right, we present:

1. Several modifications of both theoretical and practical interest (in Section 3).
2. Research directions (in Section 4) and decidability and undecidability results. As with other unification problems, the aim is to design efficient systems that produce most-general unifiers.
3. An outline of preliminary results and a pared-down version of the methodology we use to obtain them (in Section 5).

[3] Specifically, substitutions were restricted to map T-variables to the *restricted types* (see Definition 1).

2 Problem Formulation

We first define the syntactic elements that comprise an instance of unification with expansion variables in Section 2.1, then the semantics of expansions in Section 2.2, and finally constraints and their unifiers in Section 2.3.

2.1 Syntax

Definition 1 (Types). *There are two kinds of variables,* expansion variables *(E-variables) and* type variables *(T-variables),*

$$\mathsf{E\text{-}Var} = \{\, \mathsf{e}_i \mid i \in \mathbb{I} \,\} \quad \textit{and} \quad \mathsf{T\text{-}Var} = \{\, \mathsf{a}_i \mid i \in \mathbb{J} \,\}$$

where $\mathbb{I}$ and $\mathbb{J}$ are countable subsets of the natural numbers. Metavariables e, a and v range over $\mathsf{E\text{-}Var}$, $\mathsf{T\text{-}Var}$ *and* $\mathsf{Var} = \mathsf{E\text{-}Var} \cup \mathsf{T\text{-}Var}$, *respectively.*[4] Restricted types *and* types *are defined as the least sets satisfying:*

$$\begin{aligned}
\mathsf{Typ}^{\rightarrow} &\supseteq \mathsf{T\text{-}Var} \,\cup\, \{\, \tau_1 \rightarrow \tau_2 \mid \tau_i \in \mathsf{Typ} \,\} \\
\mathsf{Typ} &\supseteq \mathsf{Typ}^{\rightarrow} \,\cup\, \{\upomega\} \,\cup\, \{\, \tau_1 \cap \tau_2 \mid \tau_i \in \mathsf{Typ} \,\} \,\cup\, \{\, e\,\tau \mid e \in \mathsf{E\text{-}Var} \textit{ and } \tau \in \mathsf{Typ} \,\}
\end{aligned}$$

Metavariable $\bar{\tau}$ and τ range over $\mathsf{Typ}^{\rightarrow}$ *and* Typ, *respectively. Expressions in* $\mathsf{Typ}^{\rightarrow}$ *have constructors of the simply-typed lambda-calculus outermost. Expressions in* Typ *add the binary constructor $\cap$, E-variables and the constant (or nullary constructor) $\upomega$.*[5]

The types are ambiguous, e.g., $a \cap a \rightarrow a$ can be understood as two different types. To disambiguate we assume $\rightarrow$ and $\cap$ associate to the right and $\cap$ binds more tightly than $\rightarrow$.

Definition 2 (Substitutions and Expansions). *If A and B are arbitrary sets, we write $f : A \twoheadrightarrow B$ to denote a total function f from A to B.*[6] Substitutions *and* expansions *are defined simultaneously as the least sets such that:*

$$\begin{aligned}
\mathsf{Substitution} &\supseteq \{\boxdot\} \,\cup\, \{\, S : \mathsf{Var} \twoheadrightarrow (\mathsf{Typ} \cup \mathsf{Exp}) \mid S(a) \in \mathsf{Typ} \textit{ and } S(e) \in \mathsf{Exp} \,\} \\
\mathsf{Exp} &\supseteq \{\Omega\} \,\cup\, \mathsf{Substitution} \,\cup\, \{\, e\,E \mid E \in \mathsf{Exp} \,\} \cup \{\, E_1 \cap E_2 \mid E_i \in \mathsf{Exp} \,\}
\end{aligned}$$

[4] We are careful to distinguish between literals and metavariables ranging over literals, and between names of particular objects and metavariables ranging over names. Literals and names are in sans-serif or upright Greek fonts, e.g., e, a, E and $\uptau$. Metavariables are in italic or regular Greek fonts, e.g., e, a, E and τ.

[5] In System E, the system of intersection types with expansion variables [5] which we discussed in Section 1.2, $\upomega$ is an important feature and represents the empty intersection. Among other things, it supports analysis of dead-code in terms (more generally, in functional programs) and allows for a uniform account of terms containing dummy $\uplambda$-bindings, e.g., a term of the form $((\uplambda x.M)\, N)$ where there are no free occurrences of variable x in M.

[6] We use the symbol "$\twoheadrightarrow$" to distinguish it from "$\rightarrow$" which is a type constructor.

The symbol "$\boxdot$" denotes a particular total function from Var *to* $\mathsf{Typ} \cup \mathsf{Exp}$*, recursively defined by:*[7]

$$\boxdot = \{a \mapsto a \mid a \in \mathsf{T\text{-}Var}\} \cup \{e \mapsto e\,\boxdot \mid e \in \mathsf{E\text{-}Var}\}$$

By Proposition 1 below, $\boxdot$ *is the* identity *substitution, denoted* $\{\}$ *in Section 1.1; we prefer the less ambiguous* $\boxdot$*. The symbol "Ω" denotes a particular expansion, whose interpretation is formally given by Definition 4 below. We call* Ω *an* annihilator *because it maps every type* τ *to* ω *and every expansion* E *to* Ω*.*

Substitutions *are sort-preserving total functions from* Var *to* $\mathsf{Typ} \cup \mathsf{Exp}$*, whereas* expansions *are formal expressions built from the binary constructor* $\cap$*, unary E-variables, and substitutions or* Ω *at the leaves.*[8]

Definition 3 (Substitution Support). *A substitution* S *is always a total function on* Var *whose* support *is:*

$$\mathsf{support}(S) = \{\, a \in \mathsf{T\text{-}Var} \,|\, S(a) \neq a \,\} \;\cup\; \{\, e \in \mathsf{E\text{-}Var} \,|\, S(e) \neq e\,\boxdot \,\}$$

which may or may not be finite. For convenience and whenever possible, we define a substitution by enumerating the restriction to its support, as in Section 1.1. So $\{\mathsf{a}_1 \mapsto \mathsf{a}_2\}$ *is not partial; it is equal to* $\boxdot$ *on all variables apart from* a_1*.*

2.2 Expansion Semantics

Expansions can be applied to any type or expansion. The definition of application is the same for the common constructors of both of these sorts.

Definition 4 (Expansion Application). *The application of an expansion* E *to a type* τ *(respectively, an expansion* E_1*), written* $[E]\,\tau$ *(resp.,* $[E]\,E_1$*), returns a type (resp., an expansion). The inductive definition follows, starting with substitutions which are a special case of expansions:*

Applying substitutions to types:	*Applying substitutions to expansions:*
$[S]\,\alpha = S(\alpha)$	$[S]\,S_1 = \{\, v \mapsto [S]\,(S_1(v)) \,\|\, v \in \mathsf{Var} \,\}$
$[S]\,(\tau_1 \to \tau_2) = [S]\,\tau_1 \to [S]\,\tau_2$	
$[S]\,\omega = \omega$	$[S]\,\Omega = \Omega$
$[S]\,(e\,\tau) = [S(e)]\,\tau$	$[S]\,(e\,E) = [S(e)]\,E$
$[S]\,(\tau_1 \cap \tau_2) = [S]\,\tau_1 \cap [S]\,\tau_2$	$[S]\,(E_1 \cap E_2) = [S]\,E_1 \cap [S]\,E_2$

[7] To understand $\boxdot$ informally, take the infinite unwinding of its definition. To simplify this, let $\mathsf{T\text{-}Var}$ and $\mathsf{E\text{-}Var}$ be the finite sets $\{\mathsf{a}_1\}$ and $\{\mathsf{e}_1\}$, for which we get:

$$\boxdot = \{\mathsf{a}_1 \mapsto \mathsf{a}_1, \mathsf{e}_1 \mapsto \mathsf{e}_1\{\mathsf{a}_1 \mapsto \mathsf{a}_1, \mathsf{e}_1 \mapsto \mathsf{e}_1\{\mathsf{a}_1 \mapsto \mathsf{a}_1, \mathsf{e}_1 \mapsto \mathsf{e}_1\{\mathsf{a}_1 \mapsto \mathsf{a}_1, \mathsf{e}_1 \mapsto \cdots\}\}\}\}$$

This also shows that substitutions are higher-order functions; their result for an E-variable argument can include substitutions.

[8] As substitutions are *not* formal expressions, strictly speaking, every $S \in \mathsf{Substitution}$ should have a formal symbol, say $S^{\#}$, whose interpretation is the function S. Expansions should be built from $\{S^{\#} \mid S \in \mathsf{Substitution}\}$ instead of $\mathsf{Substitution}$. But this level of precision is not needed. An alternative is to define substitutions as formal expressions. This approach is used in [5]; it corresponds more directly to the implementation of expansions, but our concern here is to reduce the definitional overhead.

Applying expansions to types:

$$[\Omega]\,\tau = \omega$$
$$[e\,E]\,\tau = e\,([E]\,\tau)$$
$$[E_1 \cap E_2]\,\tau = [E_1]\,\tau \cap [E_2]\,\tau$$

Applying expansions to expansions:

$$[\Omega]\,E = \Omega$$
$$[e\,E_1]\,E = e\,([E_1]\,E)$$
$$[E_1 \cap E_2]\,E = [E_1]\,E \cap [E_2]\,E$$

Note how the formal definition matches the description in Section 1.1: A substitution distributes down to the outermost variables of its argument then gets applied, in the usual fashion, whereas an expansion distributes its argument down into its leaves and then applies substitutions to it.

Proposition 1 ($\boxdot$ Lifts to the Identity). *For all types τ and expansions E, it holds that $[\boxdot]\,\tau = \tau$ and $[\boxdot]\,E = E = [E]\,\boxdot$.*

Example 1 (Expansion application). $[\{\mathsf{e}_1 \mapsto \Omega \cap \boxdot \cap \mathsf{e}_4\,\boxdot\}]\,\mathsf{e}_1\,(\omega \to \mathsf{e}_2\,(\mathsf{a}_1 \to \mathsf{a}_1))$ $= \omega \cap (\omega \to \mathsf{e}_2\,(\mathsf{a}_1 \to \mathsf{a}_1)) \cap \mathsf{e}_4\,(\omega \to \mathsf{e}_2\,(\mathsf{a}_1 \to \mathsf{a}_1))$

Proposition 2 (Expansion Composition Is Associative). *For every type or expansion X, it holds that $[[E_1]\,E_2]\,X = [E_1]\,([E_2]\,X)$.*

Notation 1 *For expansions E_1 and E_2, let $(E_1; E_2)$ be shorthand for the composition $[E_2]\,E_1$, i.e., E_1 is applied first and E_2 second (opposite to their order in $[E_2]\,E_1$). By Proposition 2, ";" is associative, so $E_1; E_2; E_3$ is unambiguous.*

2.3 Solutions and Unifiers

In unification theory, a problem instance is a set of constraints, and a unifier is a solution. We adapt these standard concepts to our framework. But first, some notation and definitions on *equality* and *subtyping*.

Definition 5 (Equality). *Equality of types is determined by an* equational theory*, denoted $=_{\mathrm{EQ}}$, induced by a set* EQ *of equations.*[9] *Two types τ_1 and τ_2 are* equal *iff $(\tau_1, \tau_2) \in =_{\mathrm{EQ}}$, more conveniently written $\tau_1 =_{\mathrm{EQ}} \tau_2$ or, if the context makes clear what* EQ *is, $\tau_1 = \tau_2$. Technically, $=_{\mathrm{EQ}}$ is the least congruence relation on the set* Typ *containing* EQ*. If $\mathrm{EQ} = \varnothing$, then $=_{\mathrm{EQ}}$ is syntactic equality. However, in the presence of ω, we require that* EQ *contains the following equations at a minimum:*

(1) $\omega \cap \tau = \tau = \tau \cap \omega$ *for all types τ.*
(2) $e\,\omega = \omega$ *for all E-variables e.*

Equations (1) make ω a neutral element *(or a* unit*) of the binary $\cap$, corresponding to the intuition that ω is the empty intersection. See footnote 5. Equations (2) are needed for a smooth theory with (1) and are part of the formal setup in [2, 5].*

Equations other than (1) and (2) are added to EQ *when we consider new type constructors (such as !) or when we want type constructors to obey new algebraic laws (such as associativity and commutativity of $\cap$). See Section 3.*

[9] We follow standard terminology of universal algebra. See [1], for example.

Definition 6 (Subtyping). *To determine whether a type is a subtype of another, we use a* subtyping theory $\leq_{\mathrm{SUB}}$*, induced by a set* SUB *of subtyping pairs. Instead of writing* $(\tau_1, \tau_2) \in \leq_{\mathrm{SUB}}$*, we write* $\tau_1 \leq_{\mathrm{SUB}} \tau_2$ *or, if the context makes clear what* SUB *is,* $\tau_1 \leq \tau_2$*. The relation* $\leq_{\mathrm{SUB}}$ *is the least closed under the rules:*

$$\frac{\tau_1 =_{\mathrm{EQ}} \tau_2}{\tau_1 \leq \tau_2}\ (\mathrm{EQ}\text{-}\leq) \qquad \frac{(\tau_1, \tau_2) \in \mathrm{SUB}}{\tau_1 \leq \tau_2}\ (\mathrm{SUB}\text{-}\leq) \qquad \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ (\mathrm{TRANS}\text{-}\leq)$$

$$\frac{\tau_3 \leq \tau_1 \quad \tau_2 \leq \tau_4}{\tau_1 \to \tau_2 \leq \tau_3 \to \tau_4}\ (\to\text{-}\leq) \qquad \frac{\tau_1 \leq \tau_3 \quad \tau_2 \leq \tau_4}{\tau_1 \cap \tau_2 \leq \tau_3 \cap \tau_4}\ (\cap\text{-}\leq) \qquad \frac{\tau_1 \leq \tau_2}{e\,\tau_1 \leq e\,\tau_2}\ (e\text{-}\leq)$$

Note that $=_{\mathrm{EQ}} \subseteq \leq_{\mathrm{SUB}}$*. If* $\mathrm{SUB} = \varnothing$*, it is easy to see that* $\tau_1 =_{\mathrm{EQ}} \tau_2$ *iff* $\tau_1 \leq_{\mathrm{SUB}} \tau_2$*.*

What subtyping relations we place in SUB *depends on intended applications and restrictions imposed on type constructors. Some of these are mentioned in Section 3 and again in Section 4 where we discuss research directions. Unless otherwise stated, we assume* $\mathrm{SUB} = \varnothing$*.*

Definition 7 (Constraints and Constraint Sets). *A* constraint *is an ordered pair of types* τ_1 *and* τ_2*, written* $\tau_1 \lessdot \tau_2$*. In the special case when* $\mathrm{SUB} = \varnothing$*, we may write the constraint* $\tau_1 \lessdot \tau_2$ *as* $\tau_1 \doteq \tau_2$ *instead. If we want to distinguish the general case from the special case, the first is a* subtyping constraint *and the second an* equality constraint*. An instance of* unification with expansion variables *is a finite set of constraints. Metavariables* δ *and* Δ*, possibly decorated, range over constraints and constraint sets, respectively.*

Example 2 (Constraint Set). Introducing a running example:
$\Delta_1 = \{\mathsf{e}_1\,(\omega \to \mathsf{e}_2\,(\mathsf{a}_1 \to \mathsf{a}_1)) \doteq \omega \cap (\mathsf{e}_3\,\mathsf{a}_1 \to \mathsf{e}_3\,\mathsf{a}_1) \cap \mathsf{e}_4\,\mathsf{a}_1\}$

Definition 8 (Solved Constraint Sets). *The predicate* solved *is defined on constraints and extended to sets as follows.*

$\mathsf{solved}(\tau_1 \lessdot \tau_2)$ *iff* $\tau_1 \leq \tau_2$*;*
$\mathsf{solved}(\Delta)$ *iff* $\mathsf{solved}(\delta)$ *for every* $\delta \in \Delta$*.*

A solved constraint remains solved after the application of any expansion.

Definition 9 (Solutions and Unifiers). *If* δ *is the constraint* $\tau_1 \lessdot \tau_2$ *and* E *an expansion, then* $[E]\,\delta$ *denotes the constraint* $[E]\,\tau_1 \lessdot [E]\,\tau_2$*. If* Δ *is a constraint set, then* $[E]\,\Delta$ *denotes the constraint set* $\{\,[E]\,\delta \mid \delta \in \Delta\,\}$*. Expansion* E *is a* solution *of constraint set* Δ *iff* $\mathsf{solved}([E]\,\Delta)$*.*

In the special case when $\mathrm{SUB} = \varnothing$ *and constraints are equality constraints, we may call a solution* E *a* unifier*, to emphasize that it equalizes the two sides of each constraint.*

Remark 1. The annihilator Ω trivially solves all constraints, which we thus call the *trivial solution*. In general, Ω is not the solution we want, if only because it cannot be a *principal solution* when other non-trivial solutions exist. We think of Ω as a solution of "last resort", when everything else has failed.[10]

[10] This is the way we deal with Ω in [3]. What plays the role of Ω here is the expansion ω in [3]. It is perhaps unfortunate that, in [3] following [5], we used the same symbol ω to denote a type (as we do here) and an expansion (as we do not do here).

3 Other Versions of Unification with Expansion Variables

There is much to investigate in relation to the the fundamental framework presented in Section 2. There are more details in the report [3], which is only a beginning, as it uncovers more problems than it solves. We consider *restrictions* of the fundamental framework to make more tractable some of the theoretical problems, and consider *generalizations* for possible applications.

Omega-Free Unification. This simple restriction consists in omitting ω from the syntax of types (Definition 1) and Ω from the syntax of expansions (Definition 2), as well as omitting every part dealing with ω and Ω in each of Definition 4 and Definition 5. In particular, this makes $\mathrm{E}\textsc{q} = \varnothing$, so that $=_{\mathrm{EQ}}$ is just syntactic equality. By translating the mechanism of substitution composition used in [13] to expansion composition as defined here, consistently throughout, we essentially obtain ω*-free unification with expansion variables*, for which we have the most complete set of results so far.

Ranked Unification. The *rank* of a type τ is the smallest integer k such that the path between the root of τ and any occurrence of "$\cap$" in τ goes to the left of an occurrence of "$\to$" less than k times. Formally, we stratify types as follows. The sets Typ_0 and Typ_k, for every $k \geqslant 1$, are the least such that:

$$
\begin{aligned}
\mathsf{Typ}_0 &\supseteq \mathsf{T\text{-}Var} \;\cup\; \{\, \tau_1 \to \tau_2 \,|\, \tau_i \in \mathsf{Typ}_0 \,\} \;\cup\; \{\omega\} \;\cup\; \{\, e\,\tau \,|\, e \in \mathsf{E\text{-}Var} \text{ and } \tau \in \mathsf{Typ}_0 \,\} \\
\mathsf{Typ}_k &\supseteq \mathsf{Typ}_{k-1} \;\cup\; \{\, \tau_1 \to \tau_2 \,|\, \tau_1 \in \mathsf{Typ}_{k-1} \text{ and } \tau_2 \in \mathsf{Typ}_k \,\} \\
&\qquad\qquad \cup\; \{\, \tau_1 \cap \tau_2 \,|\, \tau_i \in \mathsf{Typ}_k \,\} \;\cup\; \{\, e\,\tau \,|\, e \in \mathsf{E\text{-}Var} \text{ and } \tau \in \mathsf{Typ}_k \,\}
\end{aligned}
$$

Clearly $\mathsf{Typ} = \bigcup_{k \geqslant 0} \mathsf{Typ}_k$. Define $\mathsf{rank}(\tau)$ as the least $k \geqslant 0$ such that $\tau \in \mathsf{Typ}_k$. Let E be an expansion and δ the constraint $\tau_1 \leq \tau_2$. If E is a unifier of δ, we say E is a *rank-k unifier* of δ if $\max\{\mathsf{rank}([E]\,\tau_1), \mathsf{rank}([E]\,\tau_2)\} \leqslant k$. We say E is *rank-k unifier* of the constraint set Δ if E is a rank-k unifier of every $\delta \in \Delta$.

AC and ACI Unification. Generalizations required by potential applications will make the binary constructor $\cap$ satisfy one or more of:

(A) *associativity* $\tau_1 \cap (\tau_2 \cap \tau_3) = (\tau_1 \cap \tau_2) \cap \tau_3$ for all types τ_1, τ_2 and τ_3.
(C) *commutativity* $\tau_1 \cap \tau_2 = \tau_2 \cap \tau_1$ for all types τ_1 and τ_2.
(I) *idempotence* $\tau \cap \tau = \tau$ for all types τ.

With AC, and *a fortiori* ACI, we need to impose an additional requirement for the theory to work smoothly (see [5]). This is expressed by an equation relating E-variables to $\cap$, allowing the first to distribute over the second, as follows:

(D) $e\,(\tau_1 \cap \tau_2) = e\,\tau_1 \cap e\,\tau_2$ for all E-variables e and types τ_1 and τ_2.

Equations (A), (C), (I) and (D) are added to $\mathrm{E}\textsc{q}$ and the equational theory $=_{\mathrm{EQ}}$ modified accordingly.

Unification+Bang. In general, intersection types provide more precise typings when they are *linear*, i.e., when the binary constructor $\cap$ is not idempotent. However, linear types have a high cost, typically making unification computationally unfeasible. We can relax linearity in a controlled way by introducing the unary type constructor "!", and adding the following subtyping relation to Sub:[11]

$$!\,\tau \leq \tau \cap \tau \qquad \text{for all types } \tau.$$

In the presence of !, we also add the following equations to Eq, dictated by semantic considerations:

(1) $!\,\omega = \omega$ (! is absorbed by ω).
(2) $!\,!\,\tau = !\,\tau$ (! is idempotent) for all types τ.
(3) $e\,!\,\tau = !\,e\,\tau$ (E-variables and ! commute) for all E-variables e and types τ.
(4) $!\,(\tau_1 \cap \tau_2) = !\,\tau_1 \cap !\,\tau_2$ (! distributes over $\cap$) for all types τ_1 and τ_2.

Introducing !, subject to the subtyping and equations above, makes it possible to obtain the precision of linear types when it is useful, while preventing linearity from getting in the way when it is not needed. More on the use of ! can be found in [5, 6] in relation to building a flexible type system for the lambda calculus.

The integration of ! with E-variables is straightforward. In addition to a case for ! in the definitions of Typ and Exp (in Definition 1 and Definition 2), we add the following rules to expansion and substitution application (see Definition 4):

$$[!\,E]\,X = !\,[E]\,X \quad \text{and} \quad [S]\,!\,X = !\,[S]\,X$$

where X is a type or an expansion. But this also complicates the algebra of types considerably and their unification accordingly.

Other Versions of Unification. Restrictions and generalizations can be combined, guided by theoretical or practical considerations. For example, *ω-free ranked unification* imposes the restrictions of both the *ω-free* and *ranked* cases.

4 Research Directions

There are standard questions in unification theory that immediately transfer to our setting. Given an arbitrary unification instance Δ, we ask:

1. *Decidability*: Is it decidable whether Δ has a unifier?
   (a) If yes, what is the complexity of the decision procedure?
   (b) If yes and Δ has a unifier, what is the complexity of building a unifier?
   (c) If no, what meaningful restrictions on Δ make the problem decidable?
2. *Principality*: If Δ has a unifier, does it have a *principal* unifier from which all other unifiers can be obtained (by some formal mechanism, such as function composition, in our case *expansion composition*)?

These questions are typically further refined. For example, in relation to complexity we may ask for upper bounds (usually easier), and lower bounds (usually harder). Note that questions 1.(a) and 1.(b) are not equivalent.[12] Naturally,

[11] For an intuition, think of $!\,\tau$ as "an object of type $!\,\tau$ can be used at type τ any number of times", and $\tau \cap \tau$ as "an object of type $\tau \cap \tau$ can be used at type τ only twice".

[12] A case in point is first-order unification. It is decidable whether an instance of first-order unification has a unifier in low-degree polynomial time. However, when there is a unifier, building a most general unifier is low-degree polynomial time only with clever data-structures that represent terms by dag's instead of trees.

such questions arise for unification with expansion variables, or any of the restrictions/generalizations mentioned in Section 3 and Section 1.1. Currently, we have the most complete set of answers (and proof techniques) for ω-free unification, for which the most significant result is the following.

Theorem 2 (ω-Free Unification With E-Variables Is Undecidable).

1. *There is an algorithm $\mathcal{A}$ which, given an arbitrary term M of the pure lambda calculus, returns an ω-fee instance $\mathcal{A}(M)$ of unification with expansion variables such that: M is β-strongly normalising iff $\mathcal{A}(M)$ has a unifier.*
2. *It is undecidable whether an arbitrary ω-free instance has a unifier.*

Only the proof of part 1 in the theorem is complicated. Part 2 is an immediate consequence of part 1, using the undecidability of β-strong normalization of terms of the lambda calculus [4]. This undecidability result does not imply the undecidability of any of the other unification cases mentioned in Section 3.In particular, as noted in Remark 1, any modification of the framework that includes a type constant such as ω and a corresponding expansion such as Ω is trivially decidable. This corresponds to the fact that all terms can be assigned the type ω in intersection type systems that include ω such as System E. Decidable modifications also arise when there are no expansion constructors except E-variables, or only one constructor.

5 Proposed Methodology

Rewriting techniques provide powerful tools for analyzing and solving problems in other forms of unification. We use a methodology inspired by such techniques. We have already developed elements of this approach, although not to the point of enabling us to tackle all of the problems we raised in Section 4.

In this section we present a streamlined and highly condensed illustration of the proposed methodology: This is a very simple rewrite system that generates unifiers for constraint sets built in the most basic version of the framework of Section 2, with $\textsc{Sub} = \varnothing$ (Definition 6). As $\textsc{Sub} = \varnothing$, we write constraints as equality constraints, i.e., we write $\tau_1 \doteq \tau_2$ instead of $\tau_1 \stackrel{.}{\leq} \tau_2$.

The unification procedure interleaves two phases, *simplification* and *rewriting*. The same approach is used in the more powerful procedures considered in [3]. But first, some helpful notation.

Notation 3 *1. Let $\boldsymbol{e}$ be a metavariable ranging over finite sequences of E-variables, including the empty sequence ε.*

2. *A constraint of the form $\boldsymbol{e}\,\tau_1 \doteq \boldsymbol{e}\,\tau_2$, where τ_1 and τ_2 do* not *have applications of the same E-variable outermost, may be written $\boldsymbol{e}\,(\tau_1 \doteq \tau_2)$. Call $\boldsymbol{e}$ the* prefix *of such a constraint.*
3. *Let the notation e/S be shorthand for the substitution $\{e \mapsto e\,S\}$. Think of such a substitution as acting under e, or in namespace e.*
4. *We extend the "/" notation to sequences of E-variables: $\boldsymbol{e} \cdot e/S$ means $\boldsymbol{e}/(e/S)$ and ε/S means S.*

Definition 10 (Simplification). *The function* simplify *factors out common structure and eliminates solved constraints:*

$$\mathsf{simplify}(\varnothing) = (\varnothing),$$
$$\mathsf{simplify}(\{\delta\} \cup \Delta) = \mathsf{simplify}(\delta) \cup \mathsf{simplify}(\Delta),$$
$$\mathsf{simplify}(\delta) = \begin{cases} \mathsf{simplify}(\{\boldsymbol{e}\,\tau_1 \doteq \boldsymbol{e}\,\tau_2,\ \boldsymbol{e}\,\tau_1' \doteq \boldsymbol{e}\,\tau_2'\}) & \textit{if } \delta \textit{ is } \boldsymbol{e}\,(\tau_1 \sqcap \tau_1' \doteq \tau_2 \sqcap \tau_2'),\\ \mathsf{simplify}(\{\boldsymbol{e}\,\tau_1 \doteq \boldsymbol{e}\,\tau_2,\ \boldsymbol{e}\,\tau_1' \doteq \boldsymbol{e}\,\tau_2'\}) & \textit{if } \delta \textit{ is } \boldsymbol{e}\,(\tau_1 \to \tau_1' \doteq \tau_2 \to \tau_2'),\\ \varnothing & \textit{if } \delta \textit{ is } \tau \doteq \tau,\\ \{\delta\} & \textit{otherwise.} \end{cases}$$

For an arbitrary constraint set Δ, $\mathsf{solved}(\Delta)$ *is true iff* $\mathsf{simplify}(\Delta) = \varnothing$*. Simplification is* sound *in the sense that, for all constraint sets Δ and all expansion E, it holds* $\mathsf{solved}([E]\,\Delta)$ *iff* $\mathsf{solved}([E]\,(\mathsf{simplify}(\Delta)))$.

Example 3. $\mathsf{simplify}(\omega \sqcap (\omega \to \mathsf{e}_2\,\mathsf{a}_1) \sqcap \mathsf{e}_4\,(\omega \to \mathsf{e}_2\,\mathsf{a}_1) \doteq \omega \sqcap (\mathsf{e}_3\,\mathsf{a}_1 \to \mathsf{e}_3\,\mathsf{a}_1) \sqcap \mathsf{e}_4\,\mathsf{a}_1)$ $= \{\omega \doteq \mathsf{e}_3\,\mathsf{a}_1, \mathsf{e}_2\,\mathsf{a}_1 \doteq \mathsf{e}_3\,\mathsf{a}_1, \mathsf{e}_4\,((\omega \to \mathsf{e}_2\,\mathsf{a}_1) \doteq \mathsf{a}_1)\}$

Definition 13 introduces the relation $\overset{S}{\Longrightarrow}$, a one-step rewrite rule, which partly solves a constraint with substitution S. A few preliminary concepts are needed.

Definition 11 (Topmost Expansion Extraction). *By induction on types:*

$$\begin{aligned} \mathsf{extract}(\bar{\tau}) &= \boxdot\\ \mathsf{extract}(\omega) &= \Omega\\ \mathsf{extract}(e\,\tau) &= e\,(\mathsf{extract}(\tau))\\ \mathsf{extract}(\tau_1 \sqcap \tau_2) &= \mathsf{extract}(\tau_1) \sqcap \mathsf{extract}(\tau_2) \end{aligned}$$

In words, the topmost expansion of τ is the largest tree of expansion constructors from the root of τ down to where non-expansion constructors are encountered, with the identity substitution $\boxdot$ or annihilator substitution Ω at each leaf.

Example 4 (extraction). $\mathsf{extract}(\omega \sqcap (\mathsf{e}_3\,\mathsf{a}_1 \to \mathsf{e}_3\,\mathsf{a}_1) \sqcap \mathsf{e}_4\,\mathsf{a}_1) = \Omega \sqcap \boxdot \sqcap \mathsf{e}_4\,\boxdot$

The "outer-part unifier" in the next definition creates substitutions for variables that occur outermost, i.e., in the top-level namespace. We generate *substitution* unifiers: Forming *expansion* unifiers from a set of substitution unifiers is easily done separately.

Definition 12 (Outer-Part Unifier). *The binary relation* $\xrightarrow{\mathsf{unifier}}$ *relates constraints and substitutions according to the following rule:*

$$\boldsymbol{e}\,(\tau_1 \doteq \tau_2) \xrightarrow{\mathsf{unifier}} \begin{cases} \boldsymbol{e}/\{a \mapsto \bar{\tau}\} & \textit{if } \{\tau_1, \tau_2\} = \{a, \bar{\tau}\} & (\mathsf{T\text{-}unify})\\ \boldsymbol{e}/\{e \mapsto \mathsf{extract}(\tau)\} & \textit{if } \{\tau_1, \tau_2\} = \{e\,\bar{\tau}, \tau\} & (\mathsf{E\text{-}unify}) \end{cases}$$

Recall that in constraint $\boldsymbol{e}\,(\tau_1 \doteq \tau_2)$ there is no E-variable e such that $\tau_1 = e\,\tau_1'$ and $\tau_2 = e\,\tau_2'$, by Notation 3. And $\bar{\tau}$ refers to a restricted *type, by Definition 1.*

– (T-unify) *solves a constraint where one side is a T-variable and the other side is a restricted type, if a is not in the empty namespace of $\bar{\tau}$.*

– (E-unify) *solves the topmost part of a constraint by substituting the topmost expansion of one side for the E-variable e on the other side, provided the argument of e is a restricted type.*

Remark 2. The (T-unify) and (E-unify) rules do not include *occur checks* on the variables in the empty namespaces of τ and $\mathsf{extract}(\tau)$ respectively. Adding these conditions prevents perpetual rewriting of constraint sets such as $\{\mathsf{e}_1\,\mathsf{a}_1 \doteq \mathsf{e}_1\,\mathsf{a}_2 \cap \mathsf{e}_1\,\mathsf{a}_2\}$.

Example 5 (Outer-Part Unifier). $\mathsf{e}_2\,(\mathsf{a}_1 \to \mathsf{a}_1) \doteq \mathsf{e}_3\,\mathsf{a}_1 \xrightarrow{\mathsf{unifier}} \{\mathsf{e}_3 \mapsto \mathsf{e}_2\,\boxdot\}$

Definition 13 (One-Step and Multi-Step Rewriting). *The* one-step rewrite *rule* $\overset{S}{\Longrightarrow}$ *does two things: It first (non-deterministically) outer-part unifies one of the constraints, applying the resulting substitution S to the entire constraint set, and then it simplifies:*

$$\Delta \overset{S}{\Longrightarrow} \mathsf{simplify}([S]\,\Delta) \qquad \textit{if there is } \delta \in \mathsf{simplify}(\Delta) \textit{ such that } \delta \xrightarrow{\mathsf{unifier}} S.$$

We define $\Delta \Longrightarrow \Delta'$ if there is a substitution S such that $\Delta \overset{S}{\Longrightarrow} \Delta'$. The multi-step rewrite *rule* $\Longrightarrow\!\!\!\!\Rightarrow$ *is the reflexive transitive closure of* $\Longrightarrow$*. Formally, we first define for all Δ, Δ_1 and Δ_2:*

$$\begin{aligned} &\Delta \overset{\boxdot}{\Longrightarrow\!\!\!\!\Rightarrow} \mathsf{simplify}(\Delta) && \textit{and} \\ &\Delta \overset{S_1;S_2}{\Longrightarrow\!\!\!\!\Rightarrow} \Delta_2 && \textit{if } \Delta \overset{S_1}{\Longrightarrow} \Delta_1 \textit{ and } \Delta_1 \overset{S_2}{\Longrightarrow\!\!\!\!\Rightarrow} \Delta_2. \end{aligned}$$

We then define $\Delta \Longrightarrow\!\!\!\!\Rightarrow \Delta'$ if there is a substitution S such that $\Delta \overset{S}{\Longrightarrow\!\!\!\!\Rightarrow} \Delta'$.

Example 6 (Rewriting).

$$\begin{aligned} \Delta_1 &\xRightarrow{\{\mathsf{e}_1 \mapsto \Omega \cap \boxdot \cap (\mathsf{e}_4\,\boxdot)\}} \{\omega \doteq \mathsf{e}_3\,\mathsf{a}_1, \mathsf{e}_2\,(\mathsf{a}_1 \to \mathsf{a}_1) \doteq \mathsf{e}_3\,\mathsf{a}_1, \mathsf{e}_4\,(\omega \to \mathsf{e}_2\,(\mathsf{a}_1 \to \mathsf{a}_1) \doteq \mathsf{a}_1)\} \\ &\xRightarrow{\mathsf{e}_4/\{\mathsf{a}_1 \mapsto \omega \to \mathsf{e}_2\,(\mathsf{a}_1 \to \mathsf{a}_1)\}} \{\omega \doteq \mathsf{e}_3\,\mathsf{a}_1, \mathsf{e}_2\,(\mathsf{a}_1 \to \mathsf{a}_1) \doteq \mathsf{e}_3\,\mathsf{a}_1\} \\ &\xRightarrow{\{\mathsf{e}_3 \mapsto \mathsf{e}_2\,\boxdot\}} \{\omega \doteq \mathsf{e}_2\,\mathsf{a}_1, \mathsf{e}_2\,(\mathsf{a}_1 \to \mathsf{a}_1 \doteq \mathsf{a}_1)\} \\ &\xRightarrow{\{\mathsf{e}_2 \mapsto \Omega\}} \varnothing \end{aligned}$$

Theorem 4 (Soundness of Rewriting). *If $\Delta \overset{S}{\Longrightarrow\!\!\!\!\Rightarrow} \varnothing$ then* $\mathsf{solved}([S]\,\Delta)$*, i.e., S is a unifier of Δ.*

Thus a rewrite sequence resulting in $\varnothing$ produces a unifier. The length of a rewrite sequence is the number of applications of the one-step rewrite rule which it comprises. Infinite rewrite sequences have an unbounded length; terminating rewrite sequences have a finite length.

Remark 3 (Incompleteness). There are constraints, such as $\omega \doteq \mathsf{a}_1 \to \mathsf{a}_1$, with expansion-unifiers, that $\Longrightarrow$ cannot solve. There are also constraints with substitution-unifiers that $\Longrightarrow$ cannot solve; for an idea of the issues involved, witness:

$$\Delta_2 = \{\mathsf{e}_1\,(\omega \doteq \mathsf{a}_1 \to \mathsf{a}_1), \mathsf{e}_1\,\mathsf{a}_1 \doteq \mathsf{a}_1 \cap \mathsf{a}_1\} \xRightarrow{\{\mathsf{e}_1 \mapsto \boxdot \cap \boxdot\}} \Delta_2 = \{\omega \doteq \mathsf{a}_1 \to \mathsf{a}_1\}$$

Δ_2 is soluble with $\mathsf{S}_3 = \{\mathsf{e}_1 \mapsto \Omega \cap \Omega, \mathsf{a}_1 \mapsto \omega\}$ but the simple system has various inadequacies, preventing the discovery of such unifiers: It always assigns outermost E-variables the *topmost* extraction, rather than allowing lesser extractions; it does not rename variables under an expanded E-variable; and, most relevant for Δ_2, it does not introduce fresh E-variables so that such solutions can be built up gradually, e.g. $[\mathsf{S}_3]\,\Delta_2 = [\{\mathsf{e}_1 \mapsto \mathsf{e}_2\boxdot \cap \mathsf{e}_2\boxdot\}; \{\mathsf{e}_2 \mapsto \Omega\}; \{\mathsf{a}_1 \mapsto \omega\}]\,\Delta_2$.

Remark 4 (Termination and Non-Termination). Some constraints have infinite rewrite sequences. For example, the following rewrite produces a kind of repeatable cycle:

$$\Delta_3 = \{\mathsf{e}_1\,((\mathsf{e}_2\,((\mathsf{e}_3\,\mathsf{a}_1 \to \mathsf{a}_2) \cap \mathsf{e}_3\,\mathsf{a}_1)) \to \mathsf{e}_4\,\mathsf{a}_2) \doteq \mathsf{e}_2\,((\mathsf{e}_3\,\mathsf{a}_1 \to \mathsf{a}_2) \cap \mathsf{e}_3\,\mathsf{a}_1), \mathsf{e}_4\,\mathsf{a}_2 \doteq \mathsf{a}_2\}$$
$$\xrightarrow{\{\mathsf{e}_4 \mapsto \boxdot\};\{\mathsf{e}_1 \mapsto \mathsf{e}_2\,(\boxdot \cap \mathsf{e}_3\,\boxdot)\};\mathsf{e}_2\,\mathsf{e}_3/\{\mathsf{a}_1 \mapsto (\mathsf{e}_2\,((\mathsf{e}_3\,\mathsf{a}_1 \to \mathsf{a}_2) \cap \mathsf{e}_3\,\mathsf{a}_1)) \to \mathsf{e}_4\,\mathsf{a}_2\}} [\mathsf{e}_2]\,\Delta_3$$

So termination depends on the rewrite strategy because Δ_3 is solved by another strategy that generates the substitutions $\{\mathsf{e}_2 \mapsto \mathsf{e}_1\,\boxdot\}; \{\mathsf{e}_1 \mapsto \Omega\}; \{\mathsf{e}_4 \mapsto \boxdot\}$. Infinite rewriting is an essential ingredient for the correspondence to β-reduction (mentioned in Section 1.2).

Theorem 5 (Completeness of Rewriting for a Special Case). *The simple rewrite system presented in this section is complete for the special case when instances Δ satisfy the conditions spelled out in [3, 2].*[13] *That is, for such a Δ, if Δ has a unifier then $\Delta \overset{S}{\Longrightarrow} \varnothing$ for some substitution S.*

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
2. A. Bakewell, S. Carlier, A. J. Kfoury, and J. B. Wells. Exact intersection typing inference and call-by-name evaluation. Technical report, Department of Computer Science, Boston University, Dec. 2004.
3. A. Bakewell and A. J. Kfoury. Unification with expansion variables. Technical report, Department of Computer Science, Boston University, Dec. 2004.
4. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.
5. S. Carlier, J. Polakow, J. B. Wells, and A. J. Kfoury. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In *Programming Languages & Systems, 13th European Symp. Programming*, vol. 2986 of *LNCS*, pp. 294–309. Springer-Verlag, 2004.
6. S. Carlier and J. B. Wells. Type inference with expansion variables and intersection types in System E and an exact correspondence with β-reduction. In *Proc. 6th Int'l Conf. Principles & Practice Declarative Programming*, 2004.
7. D. D. Champeaux. About the paterson-wegman linear unification algorithm. *J. Comput. Syst. Sci.*, 32(1):79–90, 1986.
8. A. J. Kfoury. Beta-reduction as unification. A refereed extensively edited version is [9]. This preliminary version was presented at the Helena Rasiowa Memorial Conference, July 1996.

[13] These conditions cannot be presented within the confines of this extended abstract. But suffice it to say that if the constraint set Δ corresponds to the typability of a term of the lambda calculus in a certain system of intersection types, then Δ satisfies the conditions.

9. A. J. Kfoury. Beta-reduction as unification. In D. Niwinski, ed., *Logic, Algebra, and Computer Science (H. Rasiowa Memorial Conference, December 1996), Banach Center Publication, Volume 46*, pp. 137–158. Springer-Verlag, 1999. Supersedes [8] but omits a few proofs included in the latter.
10. A. J. Kfoury, G. Washburn, and J. B. Wells. Implementing compositional analysis using intersection types with expansion variables. In *Proceedings of the 2nd Workshop on Intersection Types and Related Systems*, 2002. The ITRS '02 proceedings appears as vol. 70, issue 1 of *Elec. Notes in Theoret. Comp. Sci.*
11. A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, pp. 161–174, 1999. Superseded by [13].
12. A. J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. Supersedes [11], Aug. 2003.
13. A. J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoret. Comput. Sci.*, 311(1–3):1–70, 2004. Supersedes [11]. For omitted proofs, see the longer report [12].
14. M. Paterson and M. Wegman. Linear unification. *J. Comput. Syst. Sci.*, 16(2):158–167, 1978.
15. S. Ronchi Della Rocca. Principal type schemes and unification for intersection type discipline. *Theoret. Comput. Sci.*, 59(1–2):181–209, Mar. 1988.
16. S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoret. Comput. Sci.*, 28(1–2):151–169, Jan. 1984.
17. S. J. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. Ph.D. thesis, Catholic University of Nijmegen, 1993.

$\boldsymbol{R}$-Unification thanks to Synchronized Context-Free Tree Languages

Pierre Réty, Jacques Chabin, and Jing Chen

LIFO - Université d'Orléans, B.P. 6759, 45067 Orléans cedex 2, France
E-mail: {rety, chabin, chen}@lifo.univ-orleans.fr
http://www.univ-orleans.fr/SCIENCES/LIFO/Members/rety

Abstract. Expressing descendants or the rewrite relation by tree-(tuple) languages allows to deal with R-matching and R-unification problems. We present in addition a new class of tree-tuple languages, more expressive than former ones, but having the same properties. Thanks to this new class, we hope to solve more interesting R-matching and R-unification problems.

1 The General Method

Given a confluent term rewrite system R, computing descendants (i.e. successors by rewriting) can be applied to R-matching and R-unification. Indeed the R-matching problem $t \lhd_R t'$ has at least one solution θ iff $\exists s,\, t\theta \rightarrow_R^* s \;{}_R^*\!\leftarrow t'$, which is equivalent to check that

$$R^*(\{t\sigma \mid \sigma \in \mathit{Subst}\}) \cap R^*(\{t'\}) \neq \emptyset$$

where Subst is the set of all substitutions and $R^*(E) \stackrel{def}{=} \{s \mid \exists e \in E,\, e \rightarrow_R^* s\}$ are the descendants of the set of terms E. In the same way, and if we assume that t and t' do not share variables, the R-unification problem $t \doteq_R t'$ has at least one solution iff

$$R^*(\{t\sigma \mid \sigma \in \mathit{Subst}\}) \cap R^*(\{t'\sigma' \mid \sigma' \in \mathit{Subst}\}) \neq \emptyset$$

The above remarks can be used to decide R-matchability or R-unifiability, but do not allow to express the possible R-matches or R-unifiers. However if we can compute the binary relation $\rightarrow_R^*$ by expressing a set of pairs of terms, we can in addition express the matched terms, i.e. $\{t\theta,\, t\theta =_R t'\}$ (or the unified terms in a similar way), by computing

$$M \;=\; \Pi_1((\{t\sigma,\, \sigma \in \mathit{Subst}\}\; {}_1\!\bowtie_1 \rightarrow_R^*)\; {}_2\!\bowtie_1 ({}_R^*\!\leftarrow \; {}_2\!\bowtie_1 \{t'\}))$$

where Π_1 is the projection onto the first component, and $T \;{}_i\!\bowtie_j T'$ denotes the natural join (like in a relational database) between the i-th component of T and the j-th component of T' [1]. Now, if we can solve the "set syntactic-matching problem", i.e. compute $\{\theta \in \mathit{Subst},\, t\theta \in M\}$, we get the R-matches. We can get R-unifiers in a similar way.

2 Using Existing Tree-(tuple) Languages

Some tree-(tuple) languages allow to achieve the general method presented in the previous section.

[1] Note that computing $\rightarrow_R^*$ allows to compute the descendants because $R^*(E) = \Pi_2(E \;{}_1\!\bowtie_1 \rightarrow_R^*)$.

2.1 By computing descendants

- Using regular tree languages.
  If t and t' are linear, then $\{t\sigma \mid \sigma \in \mathit{Subst}\}$ and $\{t'\sigma' \mid \sigma' \in \mathit{Subst}\}$ are regular languages. Moreover regular tree languages are closed under intersection, and emptiness is decidable. Therefore for every rewrite system R that preserves recognizability[2], R-matchability of problems $t \lhd_R t'$ s.t. t is linear is decidable, and R-unifiability of linear equations is decidable. Strong restrictions ensuring recognizability preservation have been studied $[1, 4, 3, 9, 12, 8, 11]$.
- Using context-free tree languages.
  Assume that the rewrite system does not preserve recognizability, but satisfies the following property: E is regular implies $R^*(E)$ is context-free [10]. However context-free languages are not closed under intersection, but are closed under intersection with a regular language. Therefore, we need to assume in addition that t' is irreducible in the matching problem $t \lhd_R t'$ so that $R^*(\{t'\}) = \{t'\}$ is regular. In the linear unification problem $t \doteq_R t'$, we need to assume that R is left-linear and constructor-based, and t' contains only constructors and variables, so that $E = \{t'\sigma' \mid \sigma' \in \mathit{Subst} \wedge \sigma' \text{ is normalized}\}$ is regular and contains only normalized terms, and consequently $R^*(E) = E$ is regular.

2.2 By expressing $\rightarrow_R^*$

Here, tree-tuple languages are needed. The class of synchronized tree-tuple languages, first presented thanks to Tree-Tuple Synchronized Grammars (TTSG) [5], then extended thanks to the so-called Constraint Systems (CS) [2], and finally extended using a logic-programming framework [7] called CS-programs, allows to express $\rightarrow_R^*$ [6] in a number of cases. Moreover, thanks to the logic-programming formalism, the "set syntactic-matching problem" is trivially solvable. Therefore we can express the R-matches (or R-unifiers) and (more interesting) handle them.

3 Merging Context-free and Synchronized Languages

Context-free tree languages and synchronized tree-tuple languages have the same properties : they are closed under union, closed under intersection (of one tuple-component) with a regular tree language, membership and emptiness are decidable.

In order to increase expressivity, we have defined a new class of tree-tuple languages: the synchronized context-free languages, obtained by mixing the previous ones, and we are proving that the properties recalled above still hold. This new class should allow to express both descendants and $\rightarrow_R^*$ in more cases than before, thanks to its bigger expressivity.

[2] I.e. E is regular implies $R^*(E)$ is regular.

To give an intuitive idea, let us give a few examples. The following language is context-free ($n \in \mathbb{N}$) :

$$\begin{array}{c} s^n \\ | \\ p^n \\ | \\ a \end{array}$$

The following language is synchronized :

$$\begin{array}{ccc} & f & \\ / & & \backslash \\ s^n & & g^n \\ | & & | \\ a & & a \end{array}$$

The following language is in the new class (synchronized context-free), but it is neither a context-free language, nor a synchronized language :

$$\begin{array}{ccc} & f & \\ / & & \backslash \\ s^n & & g^n \\ | & & | \\ p^n & & h^n \\ | & & | \\ a & & a \end{array}$$

Actually, the new class can express an unbounded counting both in several independent branches (like in a synchronized language), and in two nested positions (like in a context-free language). Moreover, this class allows to recursively copy subterms, which enables to express (among others) the set of binary balanced trees.

We use logic programs, instead of grammars or automata.

Definition 1. *A* logic program with modes *is a logic program such that a mode-tuple* $\boldsymbol{m} \in \{I, O\}^n$ *is associated to each predicate symbol* P *(n is the arity of P). In other words, each predicate argument has mode I (Input) or O (Output). To distinguish them,* **output arguments will be covered by a hat**.

Notation: Let P be a predicate symbol. $ArIn(P)$ is the number of input arguments of P, and $ArOut(P)$ is the number of output arguments.

Notation: Let B be a list of atoms (possibly containing only one atom). $Input(B)$ is the input part of B, i.e. the tuple composed of the input arguments of B. $ArIn(B)$ is the arity of $Input(B)$. $VarIn(B)$ is the set of variables that appear in $Input(B)$.
$Output(B)$, $ArOut(B)$, and $VarOut(B)$ are defined in a similar way.
We also define $Var(B) = VarIn(B) \cup VarOut(B)$.

Definition 2. *Let $B = A_1, \ldots, A_n$ be a list of atoms. We say that $A_j > A_k$ (possibly $j = k$) if $\exists y \in VarIn(A_j) \cap VarOut(A_k)$. In other words an input of A_j depends on an output of A_k.*
We say that B has a cycle *if $A_j >^+ A_j$ for some A_j ($>^+$ is the transitive closure of $>$).*
We say that the clause $H \leftarrow B$ has a cycle *if B has a cycle.*

Example 1. $Q(\widehat{x}, s(y))$, $R(\widehat{y}, s(x))$ (x, y are variables) has a cycle because $Q > R > Q$.

Definition 3. *A* Synchronized Context-Free program (S-CF program) *Prog is a logic program with modes, whose clauses $H \leftarrow B$ both satisfy:*

- *$Input(H).Output(B)$ (. is the tuple concatenation) is a linear tuple of variables, i.e. each tuple-component is a variable, and each variable occurs only once.*
- *B does not have a cycle.*

Given a predicate symbol P without input arguments, the tree-(tuple) language generated by P is $L(P) = \{\boldsymbol{t} \in (T_\Sigma)^{ArOut(P)} \mid P(\boldsymbol{t}) \in Mod(Prog)\}$, where $Mod(Prog)$ is the least Herbrand model of $Prog$. $L(P)$ is called Synchronized Context-Free language *(S-CF language).*

If in addition every clause $H \leftarrow B$ of $Prog$ satisfies:

- *$Output(H).Input(B)$ (. is the tuple concatenation) is a linear tuple, i.e. each variable of $Output(H).Input(B)$ occurs only once,*

then $Prog$ is said non-copying*.*

Remark 1. - If a predicate symbol P has no input argument, this means that the outputs of P do not depend on a given input value (like in a regular language). Therefore, a S-CF program whose predicate symbols do not have input arguments, is called *synchronized program*. It generates a synchronized tree-(tuple) language. This class of languages has already been studied in [7, 6]. In these papers, synchronized programs are called cs-programs.
- If a predicate symbol P has several output arguments, this means that these output arguments are linked (synchronized) together. Therefore, a S-CF program whose predicate symbols have only one output argument, is called *context-free program* (without synchronization). It generates a context-free tree language.
- A non-copying context-free program whose predicate symbols have no input arguments, is called *regular program*. It generates a regular tree language.

Example 2. (x, y are variables)

$$Prog = \{P(\widehat{s(x)}, y) \leftarrow P(\widehat{s(x)}, y)\}$$

is not a S-CF program because $Input(H).Output(B) = (y, s(x))$ is not a tuple of variables.

$$Prog' = \{P'(\widehat{s(x)}, y) \leftarrow P'(\widehat{x}, s(y))\}$$

is a S-CF program because $Input(H).Output(B) = (y, x)$ is a linear tuple of variables, and there is no cycle. $Prog'$ is not copying because $Output(H).Input(B) = (s(x), s(y))$ is a linear tuple.

Example 3. (for $n \in \mathbb{N}$, $f^n(a)$ is the term $f(f(...f(a)))$ where f occurs n times)

$$Prog = \{S(\widehat{c(x,y)}) \leftarrow P(\widehat{x}, \widehat{y}, a, b).\ \ P(\widehat{f(x)}, \widehat{g(y)}, x', y') \leftarrow P(\widehat{x}, \widehat{y}, h(x'), i(y')).\ \ P(\widehat{x}, \widehat{y}, x, y) \leftarrow\}$$

is a non-copying S-CF program. Note that *Prog* is neither a synchronized program, nor a context-free program. The language generated by S is :

$$L(S) = \{ \begin{array}{ccc} & c & \\ / & & \backslash \\ f^n & & g^n \\ | & & | \\ h^n & & i^n \\ | & & | \\ a & & b \end{array} \mid n \in \mathbb{N}\}$$

Actually, this language is neither a synchronized language, nor a context-free language.

Example 4.

$$Prog = \{S(\widehat{\begin{array}{ccc} & f & \\ / & & \backslash \\ x & & x \end{array}}) \leftarrow S(\widehat{x}). \quad S(\widehat{a}) \leftarrow\}$$

is a copying context-free program. The language $L(S)$ generated by S is the set of binary balanced terms (all leaves occur at the same depth) over the signature $\Sigma = \{f, a\}$. Actually, $L(S)$ cannot be generated by a non-copying S-CF program.

Example 5.

$$Prog = \{P(\widehat{x}) \leftarrow Q(\widehat{x}, y), R(\widehat{y}). \quad Q(\widehat{\begin{array}{ccc} & c & \\ / & & \backslash \\ y & & x \end{array}}, x) \leftarrow Q(\widehat{y}, \begin{array}{c} s \\ | \\ x \end{array}). \quad Q(\widehat{x}, x) \leftarrow . \quad R(\widehat{a}) \leftarrow\}$$

is a copying S-CF program.

$$L(P) = \{ \begin{array}{ccccccc} & & & & & c & \\ & & & & / & & \backslash \\ & & & & c & & a \\ & & & / & & \backslash & \\ & & c & & & & s \\ & / & & \backslash & & & | \\ & \vdots & & s^2 & & & a \\ & | & & | & & & \\ & c & & a & & & \\ / & & \backslash & & & & \\ s^n & & s^{n-1} & & & & \\ | & & | & & & & \\ a & & a & & & & \end{array} \mid n \in N\}$$

References

1. M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Fifth Annual IEEE Symposium on Logic Computer Science*, pages 242–248. IEEE Computer Society Press, Philadelphia, Pennsylvania, 1990.
2. V. Gouranton, P. Réty, and H. Seidl. Synchronized Tree Languages Revisited and New Applications. In *Proceedings of 6th Conference on Foundations of Software Science and Computation Structures, Genova (Italy)*, LNCS. Springer, 2001.
3. P. Gyenizse and S. Vagvolgyi. Linear Generalized Semi-monadic Rewrite Systems Effectively Preserve Recognizability. *Theoretical Computer Science*, 194, pp 87-122, 1998.
4. F. Jacquemard. Decidable approximations of term rewrite systems. In editor H. Ganzinger, editor, *Proceedings 7th Conference RTA, New Brunswick (USA)*, volume 1103 of *Lecture Notes in Computer Science*, pages 362–376. Springer-Verlag, 1996.
5. S. Limet and P. Réty. E-Unification by Means of Tree Tuple Synchronized Grammars. *Discrete Mathematics and Theoritical Computer Science (*`http://www.dmtcs.loria.fr`*)*, 1:69–98, 1997.

6. S. Limet and G. Salzer. Proving properties of term rewrite systems via logic programs. In *proceedings of RTA 2004*, volume 3091 of *LNCS*, pages 170–184. Springer Verlag, 2004.
7. Sébastien Limet and Gernot Salzer. Manipulating tree tuple languages by transforming logic programs. In Ingo Dahn and Laurent Vigneron, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.
8. C. Loding. Model-checking Infinite Systems Generated by Ground Tree Rewriting. In *Proceedings of the 7th Conference on Foundations of Software Science and Computation Structures*. LNCS, Springer, 2002.
9. P. Réty. Regular Sets of Descendants for Constructor-based Rewrite Systems. In *Proceedings of the 6th international conference LPAR*, number 1705 in Lecture Notes in Artificial Intelligence (LNAI), Tbilisi, Republic of Georgia, 1999. Springer Verlag.
10. P. Réty and J. Vuotto. Context-free tree languages for descendants. In *5th Workshop on Rule-Based Programming (RULE'2004). Proceedings in ENTCS vol 124(1)*, 2005.
11. H. Seki, T. Takai, F. Youhei, and Y. Kaji. Layered Transducing Term Rewriting System and its Recognizability Preserving Property. In *13th International Conference RTA*, volume 2378 of *Lectures Notes in computer Science*. Springer-Verlag, 2002.
12. T. Takai, Y. Kaji, and H. Seki. Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In L. Bachmair, editor, *Proceedings 11th Conference RTA, Norwich (UK)*, volume 1833 of *LNCS*, pages 246–260. Springer-Verlag, 2000.

Symbolic Debugging in Polynomial Time

Christopher Lynch[1] and Barbara Morawska[2]

[1] Department of Computer Science, Box 5815, Clarkson University, Potsdam, NY 13699-5815, USA, E-mail: clynch@clarkson.edu

[2] Chair for Automata Theory, Institute for Theoretical Computer Science, Dresden University of Technology, Germany, E-mail: morawska@tcs.inf.tu-dresden.de

Abstract. We show how to use Sorts and Normal Forms to speed up the Narrowing Procedure. Even when an equational theory is closed under Superposition, Narrowing may not terminate. When an equational theory is closed under Paramodulation Narrowing terminates, but it may have NP complexity. We show that it is possible in some equational theories to use Sorts and Normal Forms to model well-formed terms to force Narrowing to terminate with linear time complexity. We give some real examples which have linear behavior, such as the theory of lists. We also extend the theory of lists to the theory of lists plus length. We show how these results are useful in debugging.

1 Introduction

When we reason about programs, we must take into account the data structures that are used in the programs. Reasoning about programs then takes place modulo an equational theory of the data structures used in the programs. There are general inference procedures, such as Superposition, which allow such reasoning for any equational theory. However, these procedures are undecidable in general. Therefore, it is necessary to give a better analysis of the decidability and running time of these procedures for specific data structures.

For example, [1] shows how rewriting techniques can be used for verification of programs modulo the theory of lists and the theory of arrays. They show that the problem of program verification is the problem of showing if one set of ground equations implies another ground equation, modulo the theory of the data structures used in the program. They show that the Superposition inference system will halt when solving this problem for the theory of lists and the theory of arrays. In [3] it is shown that the problem can be solved modulo the theory of lists in $O(nlg(n))$.

The problem of program verification is a universal problem. We want to show that the precondition implies the postcondition for all values of the variables used in the program. In this paper, we consider a different problem, which we call the symbolic debugging problem. Suppose we want to know if is is possible to follow a certain path in the program. Furthermore, suppose we want to know for what values of the variables a certain path will be taken. A path in the program can be represented by a set of equations and disequations, which represent the conditions appearing at each choice point in the program. Therefore, the problem we need to solve is the problem of E-unification of that set of equations and disequations. If the E-unification problem has a solution, then that means it is possible to take that path. And the solution represents all the values of the variables that will cause the execution of the program to proceed along that path. The E-unification we need to solve will be unification of those equations and disequations modulo the data structures used in the program.

The problem of E-unification needed for debugging is much harder than the word problem necessary for verification. We now need to solve an existential problem instead of a universal problem. Whereas, universal equational problems can be solved by Rewriting, E-unification problems need to be solved by Narrowing. For some equational theories, Rewriting terminates in polynomial time, whereas Narrowing does not terminate. This is bad enough, but we are also interested in efficient procedures. Therefore, we want to show that Narrowing will halt in polynomial time. This is even more difficult. The problem is that Rewriting can be performed don't care nondeterministically, whereas Narrowing is don't know nondeterministic. Even simple ground equational theories, such as $\{f(a) \approx b\}$ have an exponential time Narrowing procedure [4].

We aim to make Narrowing procedures more deterministic, so that they halt in polynomial time for some interesting equational theories. We solve that problem using Sorts and Normal Forms. For example, in the theory of Lists, we have equations such as $car(cons(x, y)) = x$. Given a goal containing a term of the form $car(z)$, we narrow it don't know nondeterministically using the above equation. This causes exponential behavior for Narrowing in the theory of Lists.

Notice that in order for a term of the form $car(z)$ to make sense, z must be a nonempty list, and therefore z is of the form $cons(z_1, z_2)$. So we will say that z is equivalent to a term $cons(z_1, z_2)$. We extend our goal clause with a disequation $z \not\approx cons(z_1, z_2)$. We then select the extension literals. This will ensure that whenever there is a Narrowing step possible in a goal, then there is a Rewrite step at that position. This allows us to make the entire procedure don't care nondeterministic, and so it will halt in linear time for the theory of lists.

We give a syntactic condition on an equational theory and its associated Sort and Normal Form theory. And we show that for any equational theory meeting this condition, there is a linear time E-unification procedure for that theory. The E-unification procedure is just the Superposition inference system with eager Rewriting. This class of theories includes the theory of Lists.

The class of theories mentioned above does not include recursive theories. However, we show that we can extend the theory of Lists to include the theory of Length. E-unification for the combination of those theories can still be done in linear time. We will decide in linear time if the goal is E-unifiable. If it is E-unifiable, then we will generate a representation of the unifier. The representation will not be a recursive representation as in [5]. What we will do is to not allow any Narrowing steps into a term of the form $length(x)$. When the Narrowing procedure halts, the goal will be in a form for which E-unification is easily decidable using the the syntactic unification procedure. The representation of the unifier which we create may contain equations of the form $length(x) = length(y)$ for example. We argue that such a representation is easily understandable to a human who examines the result. There are infinitely many E-unifiers of this equation. The procedure of [5] can handle the theory of Lists and Length, but it will not run in polynomial time.

Finally we combine our existential results for E-unification with the universal results of the word problem. We consider problems of the form $\forall X.(A \rightarrow \exists Y.B)$, where X and Y are sequences of variables, A may contain variables from X and B may contain variables from X and Y. This is equivalent to the problem of showing E-unification for a ground theory modulo the theory of some data structure. We show that such problems are still solvable in polynomial time for the theory of Lists.

2 List Example

Consider the theory of lists as an example. In the theory of lists, we have the following equations:

Theory of Lists L

$$car(cons(x, y)) \approx x \qquad cdr(cons(x, y)) \approx y \qquad cons(car(x), cdr(x)) \approx x$$

The theory of lists is closed under Superposition, so it is a convergent rewrite system. Therefore, the word problem is decidable in the theory of lists, because to determine if two terms are equal, it is only necessary to reduce them both to their normal forms, and see if those two normal forms are the same. We note that in every application of a rewriting step from s to t in this theory, t will have fewer symbols than s. This implies that terms can be reduced to their normal form in a linear number of steps, and therefore the word problem can be decided in linear time.

But this may not be the case for E-unification. Even if a theory is closed under Superposition, Narrowing may not halt. And in theories where it does halt, it may halt in exponentially many steps. For the theory of lists, it is closed under Paramodulation, so E-unification can be decided in NP by Narrowing[7] or Basic Syntactic Mutation[4]. In practice, this may be exponential.

Consider the E-unification problem represented by $car(x) \not\approx car(y)$. It is possible to apply a Narrowing step to the left hand side of this goal. The result of that Narrowing step is $x_1 \not\approx car(y)$, with the substitution $x \mapsto cons(x_1, y_1)$. This is a don't-know nondeterministic step. Therefore, we also have to apply Narrowing to the right hand side of the original goal, which yields $car(x) \not\approx x_2$, with the substitution $y \mapsto cons(x_2, y_2)$. These substitutions are essentially the same substitution. In addition, we could have applied Equation Resolution at the beginning, which gives the substitution $x \mapsto y$. This substitution is less general than the other two.

It would be easy to define an exponential time E-unification problem based on this idea. For example, let s_0 be the term $cdr(x_0)$, and for $i \geq 0$, let $s_{i+1} = cons(car(x_{i+1}), s_i)$. Similarly, let t_0 be the term $cdr(y_0)$, and for $i \geq 0$, let $t_{i+1} = cons(car(y_{i+1}), t_i)$. Then the E-unification problem $s_n \not\approx t_n$ will require exponential time to solve using Narrowing.

However, all the solutions that are found are essentially the same. So how can we streamline the Narrowing Procedure so that it only finds one solution for this example? Our solution is to for the substitutions to only be in a form which makes sense.

The first part of our solution is to use Sorts, which forces the substitutions to only be in form which makes sense. So we define a Sort theory for the theory of lists, as

Sort Theory for Lists

$$I(car(x_L)) \qquad L(cdr(x_L)) \qquad L(cons(x_I, y_L))$$

This indicates that there are two sorts in the theory of lists. The Sort L is for list, and the sort I is for an item of a list. So this sort theory says that *car* takes a list and returns an item, *cdr* takes a list and returns a list, and *cons* takes an item and a list and returns a list.

More importantly, we define normal forms for the theory. For example, for lists, we have the following normal forms:

Normal Forms for List Theory

$$car(cons(x, y)) \qquad cdr(cons(x, y))$$

These normal forms say that a *car*-term must be applied to something equivalent to a *cons*-term. It does not need to be exactly a *cons*-term though. For example $car(cdr(x))$ is a valid term. A *cdr*-term must also be applied to something equivalent to a *cons*-term. In this example, the normal forms are from the left hand side of the equation, which will be the case in many theories. We do not require that there is only one normal form for each function symbol, although that will often be the case. The normal forms will end up providing more determinism to the procedure.

Now we need to modify everything to take into account the normal forms. We do it by extending the equations in L and also extending the goal. Each clause is extended so that whenever $car(t)$ or $cdr(t)$ appears in a clause, we add a new literal $t \not\approx cons(y, z)$ to that clause, where y and z are fresh variables.

Another thing which gives more determinism in the case of lists is to note that if the goal contains a literal $s \not\approx t$ where s and t only contain *cons* function symbols, then there are no inferences into s or t, so again Equation Resolution can be performed deterministically. This will also be generalized for some equational theories besides the list theory.

The equation $cons(car(x), cdr(x)) \approx x$ in the Theory of Lists L becomes a clause $cons(car(x), cdr(x)) \approx x \vee x \not\approx cons(y, z)$ and is no longer necessary, because it becomes redundant. We can see that this clause is implied by the other clauses in the expanded List Theory. If the second literal is false then $x \approx cons(y, z)$. Then $cons(car(x), cdr(x)) \approx cons(car(cons(y, z)), cdr(cons(y, z))) \approx cons(y, z) \approx x$. After removing a redundant clause and redundant literals from the clauses, the sort theory for lists is

Theory of Lists L'

$$car(cons(x, y)) \approx x$$
$$cdr(cons(x, y)) \approx y$$

Going back to our example, suppose the goal is $car(x) \not\approx car(y)$. This becomes $x \not\approx cons(x_1, x_2) \vee y \not\approx cons(y_1, y_2) \vee car(x) \not\approx car(y)$. We assume one of the equations added by Extension is selected first when possible. So assume the first literal is selected. A deterministic application of Equation Resolution gives $y \not\approx cons(y_1, y_2) \vee car(cons(x_1, x_2)) \not\approx car(y)$. Then Rewriting gives $y \not\approx cons(y_1, y_2) \vee x_1 \not\approx car(y)$. Now the first literal here must be selected. Another deterministic Equation Resolution gives $x_1 \not\approx car(cons(y_1, y_2))$, followed by Rewriting which gives $x_1 \not\approx y_1$. Finally, Equation Resolution gives the empty clause with the accumulated substitution $[x \mapsto cons(y_1, x_2), y \mapsto cons(y_1, y_2)]$ over the variables of the original goal. All the steps are deterministic or don't-care nondeterministic.

This is in fact the same solution we got earlier, if we take the first choice, since *car* must be applied to a nonempty list. But now this solution has no choice points. Therefore, the exponential example above can be solved in linear time. In fact all E-unifications in this theory can be solved in linear time, as we shall see.

3 Superposition

Now we formalize the informal ideas given in the previous section. We assume we are given a set of equations E, and a goal G. We will consider refutation theorem proving, so we think of G as a clause containing disequations. For now, we assume that G can only contain symbols appearing in E and variables.

3.1 Sorts

We introduce sorts in order to eliminate terms and possible solutions to E-unification which make no sense. Checking well-sortedness of terms in a goal can be done once at the beginning of our E-unification procedure and it is fast, because we use only a very simple concept of sorts. The more complicated demands on terms will be forced by normal form terms, which will be presented in the next subsection.

We assume a sort theory as a finite set of atoms of the form $S(t)$, composed of monadic predicates (denoting sorts) and terms. All variables in the terms are assumed to be universally quantified.

A sort theory is *simple* if for each function symbol f, there is at most one sorted term of the form $S(f(t_1, \cdots, t_n)$. The sort theory is *elementary*, if every term in a sort declaration is either a variable or a constant of of the form $f(x_1, \ldots, x_n)$, where all $x_1, \ldots, x_n$ are different. We assume that we have a sort theory which is simple and elementary.

Definition 1. *Let L be a simple sort theory. The set of well-sorted terms T_S of a sort S is defined recursively as follows:*

1. *$x_S \in T_S$, if x_S is a variable labeled with S,*
2. *$t \in T_S$, if $S(t)$ is a sort declaration in the sort theory L,*
3. *$t\sigma \in T_S$, if $t \in T_S$ and $x\sigma \in T_{S'}$, for $x_{S'} \in Var(t)$ and $x_{S'} \in dom(\sigma)$.*

An equation $s \approx t$ is called well-sorted if s and t belong to the set of well-sorted terms of the same sort.

Given a goal equation $s \approx t$ we can easily check if it is well-sorted. Use syntactic unification with sorts on two goals: $s' \approx s$ and $t \approx t'$, where s' is a term from a term declaration in the sort theory, such that s and s' have the same root symbol, and t' is a term from a term declaration of the same sort as s' such that t and t' have the same root symbol.

It is known that syntactic unification with sorts, in a simple and elementary sort theory is linear and unitary. [8] If our goal equation had variables not labeled with sorts, the above procedure can be used besides checking well-sortedness, also to assign the appropriate sort labels to the variables.

We assume that a given equational theory E is well-sorted with respect to some simple and elementary sort theory. Hence if $s \approx t \in E$, s and t belong to the same set of well-sorted terms of some sort.

Given a goal, we first apply this sorting procedure to guarantee that the goal is well-sorted. If the goal is not well-sorted, then Fail is returned.

In the next lemma we show that if a term is well-sorted, then the Superposition inferences, given in a later section, will preserve well-sortedness. The opposite is not the case, because e.g. the term $cons(car(cons(x, y)), cdr(cons(y, z)))$ is not well sorted (y appears once as a list and second as an item). But applying two rewrite steps to this term gives $cons(x, z)$ which is well-sorted.

Lemma 1. *Let L be a sort theory, and E – an equational theory such that if $s \approx t \in E$, then s and t are well-sorted. Then the Superposition inference system preserves well-sortedness of a goal equation.*

Proof. The conclusion of Equational Resolution is well-sorted, and Rewriting is a special case of Narrowing. Hence we only have to consider Narrowing. Assume that $e[u]$ is a goal equation and $\sigma = mgu(u, s)$, where $s \approx t \in E$. Since σ is a syntactic unifier, s, $s\sigma$ must be of the same sort as u. Since $s \approx t$ is a well-sorted equation, t and therefore also $t\sigma$ must be of the same sort as u. Therefore, by the definition of well-sorted equations, $e[t]\sigma$ is well-sorted.

3.2 Normal Forms

We define a set of normal form terms, N. All terms in N should be well-sorted.

As mentioned informally in the previous section, all equations in E and clauses in the goal must be initially extended according to the following rule.

Extension rule

$$\frac{e[f(u_1, \cdots, u_n)] \vee C}{e[f(u_1, \ldots, u_n)] \vee u_1 \not\approx v_1 \vee \cdots \vee u_n \not\approx v_n \vee C}$$

where $e[f(u_1, \ldots, u_n)]$ is a literal and $f(v_1, \ldots, v_n)$ is a fresh renaming of a term in N.

Think of the Extension rule as a rewrite rule, which rewrites the premise of the rule into its conclusion. We do not allow the same term to be extended more than once. Then $Ext(C)$ represents the set of all normal forms of C with respect to the Extension rule. We will give conditions so that normal forms always exist, and we will actually give conditions such that $Ext(C)$ contains just one extended clause. Define $Ext(E) = \bigcup_{C \in E} Ext(C)$. Let $Cl(E)$ be the saturation of $Ext(E)$ by Superposition.

We define a property called *erasing* to force the Extension process to halt.

Definition 2. *A set of normal forms N is* erasing *if the symbols in the signature can be divided into two sets F and G such that for every term t in N, $root(t)$ is in F, and every other symbol in t is in G.*

We can think of F as defined symbols, and think of G as constructors.

If N is erasing then a precedence can be defined such that every symbol in F is larger than every symbol in G. For an erasing set of normal forms, the Extension rule can only be applied a linear number of times, since an extension of a term cannot be further extended.

In the following, we will consider sets of equations that are extended and saturated by the Superposition inference system, given in the next section.

Notice that in the theory of lists L', an application of Equation Resolution to the extended clause gives back the original clause, which subsumes the extended clause. For example, the extension of $car(cons(x, y)) \approx x$ is $cons(x, y) \not\approx cons(z, w) \vee car(cons(x, y)) \approx x$. An application of Equation Resolution to this clause gives $car(cons(z, w)) \approx z$, which is a renaming of the original equation, and subsumes the extended equation.

3.3 Superposition Inference System

Now we define the Superposition inference system. This inference system is parameterized by a selection function such that one literal in each clause is selected, and only the selected literal is allowed to be involved in an inference.

Note that all the clauses used are Horn Clauses. Each goal is made up of only disequations, and Extension preserves that fact. Each equation in E is a single equation, and Extension adds negative literals, so it remains a Horn Clause.

Let L and M be literals. Define an ordering $\prec$ such that $L \prec M$ if and only if the multiset of symbols appearing in L is smaller than the multiset of symbols appearing in M, according to the multiset extension of the precedence on symbols. Note that this ordering is not stable under substitutions. If C is a clause, then a literal L is *minimal* in C if there is no literal in C which is smaller than L according to $\prec$.

We define the selection function as follows. In each clause containing a negative literal, some minimal negative literal is selected. If a clause contains only a single positive literal, then that literal is selected.

Recall the Superposition inference system.

Superposition

$$\frac{D \vee s \approx t \qquad e[u] \vee C}{D\sigma \vee e[t]\sigma \vee C\sigma}$$

where $s \approx t$ is selected, $e[u]$ is selected, u is not a variable, $s\sigma \not\leq t\sigma$, and $\sigma = mgu(s, u)$. If $e[u]$ is an equation $u'[s'] \approx v$, then we can require that $u'[s'] \not\leq v$. The ordering $<$ must be a reduction ordering. Notice that it is different from the ordering $\prec$.

If $e[u]$ is a disequation and D is empty, then the Superposition inference rule is also called Narrowing. We have noticed that everything here is a Horn Clause. Also, the selection function requires us to select a negative literal when one exists. Therefore, the Superposition rule is only applicable when D is empty.

If $s\sigma = u$, $s\sigma > t\sigma$ and $D$is empty, then a Superposition inference becomes a Rewriting step, and is a don't-care nondeterministic step because E is confluent. In other words, Rewriting is the following inference rule:

Rewriting

$$\frac{s \approx t \qquad e[u] \vee C}{e[t]\sigma \vee C}$$

where u is not a variable and there is a substitution σ such that $s\sigma > t\sigma$ and $s\sigma = u$. After rewriting, $e[u] \vee C$ is deleted.

Equation Resolution

$$\frac{u \not\approx v \vee C}{C\sigma}$$

where $\sigma = mgu(u, v)$. and $u \not\approx v$ is selected.

The inference system we have given is sound and complete, because the selection rule selects a negated literal when it can, [2]. In fact it is even sound and complete for answer substitutions if substitutions are saved and accumulated[6].

Theorem 1. *The Superposition inference system with the given selection function is sound and complete.*

4 Polynomial Time *E*-unification

Next we show how Superposition can be a terminating procedure for goal solving, and how it can have polynomial complexity. We need the following property to show the termination of Superposition for goal solving. The following definition is just an extension of a previous definition, to cover E as well as N.

Definition 3. *A set of equations E and normal forms N is* erasing *if the symbols in the signature can be divided into two sets F and G such that*

1. *for every equation $s \approx t \in E$, the root symbol of s in in F, every other symbol in the equation is in G, and every variable in t occurs in s, and*
2. *for every term t in N, $root(t)$ is in F, and every other symbol in t is in G.*

Notice that in a set of erasing equations, every equation can be oriented. Also, every proper subterm u of an equation in an erasing set of equations is reduced.

Consider a set of erasing equations E, where the symbols are divided into F and G. Then we can add an inference rule Eager Equation Resolution.

Eager Equation Resolution

$$\frac{u \not\approx v \vee C}{D}$$

where $u \not\approx v$ is selected, and there are no symbols from F in u and v, and D is *Fail* if u and v are not unifiable, and D is $C\sigma$ if $\sigma = mgu(u, v)$.

This can be applied deterministically to a selected disequation. This is obvious since no Narrowing or Rewrite rules apply, but this will become important later when we extend E with a ground theory. We can justify the correctness of Eager Equation Resolution, because we are going to assume that $g(x_1, \cdots, x_n) \approx g(y_1, \cdots, y_n) \models x_1 \approx y_1 \wedge \cdots \wedge x_n \approx y_n$ for all symbols in G. Later on when we add ground equations to the theory, we will still assume that this statement is true, so Eager Equation Resolution will still be correct.

Next is a property on N which will ensure that when the extension rule is applied to an equation e then $Ext(e)$ contains only one clause.

Definition 4. *The set N of normal forms is* simple *if whenever $s, t \in N$ and $s \neq t$ then $root(s) \neq root(t)$.*

Next is another property to ensure the determinism of the procedure. Due to the extensions, it will ensure that whenever there is a Narrowing step, there is also a Rewrite step.

Definition 5. *A set of normal forms N* defines *a set of equations E if*

1. *for every equation $s \approx t \in E$ with $s \not\leq t$, s is in N, and*
2. *for every term s in N there is an equation $s \approx t$ in E with $s \not\leq t$.*

Notice that if N is simple and defines E and E is erasing, then Extension does not add anything new.

Proposition 1. *Let N and E be erasing, and N a simple set of normal forms such that N defines E. Let $Cl(E)$ be the saturation of the Extension of E. Then $Cl(E) = E$.*

Proof. Suppose we perform an Extension on $e[f(s_1, \cdots, s_n)]$, where $f(t_1, \cdots, t_n)$ is a normal form. The result is $s_1 \not\approx t_1 \vee \cdots \vee s_n \not\approx t_n \vee e[f(s_1, \cdots, s_n)]$. Since N defines E, then e is of the form $f(s_1, \cdots, s_n) \approx t$, where $f(s_1, \cdots, s_n)$ is just a renaming of $f(t_1, \cdots, t_n)$. There are no Narrowing or Rewriting inferences into $s_i \not\approx t_i$, so only Equation Resolution applies to the new literals. The result is just a renaming of e which subsumes the extended equation.

Theorem 2. *Let N and E be erasing, such that E is saturated under Superposition. Let N be a simple set of normal forms such that N defines E. Let $Cl(E)$ be the saturation of the Extension of E. Then Superposition halts on $Cl(E)$ and any goal in linear time if Rewriting is performed eagerly, and either a single mgu is generated or the process fails.*

Proof. First note that $Cl(E) = E$. In order to show termination, we will show that whenever Narrowing applies, then Rewriting also applies.

Suppose that a Narrowing inference applies to the goal. Let's consider an innermost Narrowing step. Then there is some equation $s \approx t$ in E and a term u in the goal with $\sigma = mgu(s, u)$. If $s\sigma = u$, then there is a Rewrite step which can be performed.

Since N defines E, $s \in N$. Let u be of the form $f(u_1, \cdots, u_n)$. The term u must have descended from an initial term in the goal of the form $f(v_1, \cdots, v_n)$, since E is erasing. Let θ be the accumulated substitution applied to the goal. Then $u_i \approx v_i\theta$ for each i. There also must have been an Extension at the beginning where the disequations $v_1 \not\approx w_1 \cdots v_n \not\approx w_n$ were added to the goal. These disequations were selected first in the goal, so we must have already proved that $v_i\theta \approx w_i\theta$ for all i. This implies that $u_i \approx w_i\theta$ for all i.

Since we consider an innermost Narrowing step, u_i must be reduced. We also know that w_i is reduced, and there must be θ' such that $u_i = w_i\theta'$ for all i. Therefore $\sigma = \theta'$, and Rewriting can be applied.

Since we perform Eager Rewriting, this shows that no Narrowing steps will ever be performed. Therefore, all inferences are deterministic. Every Rewriting step applied to a term u in the goal will remove one occurrence of a symbol in F and replace it by symbols in G, since E is erasing.

Therefore, Rewriting will only be applied finitely many times, so the procedure will halt. Note that Equation Resolution does not increase the number of symbols of F appearing in the goal, as long as the goal is stored in a dag.

Since the result is don't care nondeterministic, a single *mgu* will be generated if the procedure halts.

Since each Rewriting removes a symbol of F from the goal, the total number of applications of Rewriting is at most n, the size of the goal. If we apply each unifier by representing the goal as a dag, then we can be sure that the set of inferences do not grow the goal to be more than a constant times its original size, since each inference can only increase the size of the goal by a constant amount. And the number of Equation Resolution steps is just the number of Extensions plus one, which is linear in the size of the goal. Since no Narrowing steps are performed, the whole process is in linear time.

The example of List Theory L' with the normal forms we have given is erasing, since there is only one normal form rooted with *car* and one normal form rooted with *cdr*. The set of normal forms N defines L' because the set N is just the set of left hand sides of L'. Furthermore, N and L' is simple if we let $F = \{car, cdr\}$ and let $G = \{cons\}$. Therefore, E-unification can be solved in linear time for the theory of lists using Superposition with eager Rewriting, and the result will always be a most general unifier if it is solvable.

5 Lists with Length

It is possible to add some recursive theories and still run in polynomial time. Here we discuss a theory of Lists with Length. We extend the List theory to include the theory of length of lists.

$$length(nil) \approx 0 \qquad length(cons(x, y)) = s(length(y))$$

The sort theory is extended to include:

$$L(nil) \qquad Int(length(x_L)) \qquad Int(0) \qquad Int(s(x_{Int}))$$

We could either make Int be the same type as the items of the list, or a different type.

This theory is difficult to handle, because many goals have an infinite complete set of unifiers. For example, the equation $length(x) \approx length(y)$ has infinitely many unifiers. The substitution $[x \mapsto y]$ is a unifier. But then so is $[x \mapsto cons(x_1, z), y \mapsto cons(y_1, z)]$, $[x \mapsto cons(x_1, cons(x_2, z)), [y \mapsto cons(y_1, cons(y_2, z))]$, etc. Normal forms will not help in this case. We have to find a different solution if we want to find a halting procedure, much less a polynomial time procedure.

Our solution to this problem is to not allow any inferences into terms of the form $length(x)$. Then we may not derive the empty clause of course. But, for each goal, we will derive something from which we can tell whether or not there is a solution. We will not give the entire complete set of unifiers. But we will give a representation that should be meaningful to a programmer. For example, if we halt with $length(x) \approx length(y)$, then we suggest that this is even more meaningful to a programmer than a schematization of the set of solutions.

The procedure for solving E-unification for lists with length is an extension of the procedure for lists. We start by applying the extension rule to the original equations and goal. Then we run the Superposition procedure with eager Rewriting, and the following restriction:

The Superposition rule is restricted so that u is not allowed to be a term of the form $length(x)$ with x a variable. Similarly, the Equation Resolution rule is modified so that u and v may not contain any subterms of the form $length(x)$ unless u or v is a variable. We add another rule to the inference system

$$\frac{cons(u_1, u_2) \not\approx cons(v_1, v_2) \vee C}{u_1 \not\approx v_1 \vee u_1 \not\approx v2 \vee C}$$

where $cons(u_1, u_2) \not\approx cons(v_1, v_2)$ is selected. We will call this inference system *Superposition with Length Restrictions*.

After Superposition with Length Restrictions is performed, we fail if a disequation of the form $cons(s, t) \not\approx nil$ appears in the leftover goal. Otherwise, we apply the inference system *Syntactic Length Unification*, which proceeds as follows. We transpose the leftover goal clause C into a Syntactic Unification problem in the following way. We replace each term $length(y)$ by a new variable $x_{length(y)}$. Then we treat the clause like a set of equations, and perform Syntactic Unification. If Syntactic Unification fails, then we fail. If syntactic unification succeeds, then we succeed. The answer substitution we get represents a solution to the problem, but it is not really a substitution, because the $length(x)$ terms have been replaced by variables.

Note that after Superposition with Length Restrictions finishes, there are no *car* terms or *cdr* terms in the goal, because any well-sorted instance of those terms must contain a variable or a cons-term as a subterm, and we have shown in the previous section that all *car* or *cdr* terms will be rewritten. Also, after Superposition with Length Restrictions, there can be no *cons* terms left in the goal. If a *cons* term appears directly underneath a *car* or *cdr* or *length* term, then it has been rewritten. This means that a *cons* symbol could only appear in a term where all nonvariable symbols are *cons* symbols. But then there must be an equation of the form $s \approx t$ where s contains only *cons* symbols and variables, and t is a *length* term or an s term or 0 or *nil*. If t is a *length* term or s term or *nil* then the equation is not well-sorted, and we would have failed. If t is *nil* then we would fail as described above. Therefore *length* must only be applied to a variable, because *length* applied to an s term or 0 is not well-sorted, and $length(nil)$ would have been rewritten.

We need to prove that this process is sound and complete, and also that it runs in polynomial time.

Theorem 3. *The inference system Superposition with Length restrictions followed by Syntactic Length Unification is sound and complete.*

Proof. Since Superposition is sound and complete, that means that Superposition with Length Restrictions preserves the set of solutions. So we need to prove that Syntactic Length Unification is sound and complete.

For completeness, we only need to show that Syntactic Length Unification does not fail when there is a solution. If Syntactic Length Unification fails because of the clash rule, then Syntactic Unification (without transformation of length terms into variables) would fail also. So we only need to show that failures by Occurs Check are truly failures. Suppose we have an equation of the form $x_{length(y)} \approx t[x_{length(y)}]$, where $x_{length(y)}$ is a proper subterm of t. In order to be well-sorted, $x_{length(y)}$ must be an immediate subterm of s and s must be the root of the term or else immediately under another s, etc. This implies that the occurs check must have occurred in an equation of the form $x_{length(y)} = s^n(x_{length(y)})$ and we note that $length(y) = s^n(length(y))$ has no solution.

For soundness, we need to show that if our procedure succeeds, then there really is a solution to the untransformed problem. Notice that if a variable $x_{length(y)}$ appears, then y was given no substitution by Superposition with Length Restrictions. Furthermore y could not have appeared anywhere here except in the form $length(y)$ because all *car*, *cdr* and *cons* have been removed, and since y is of type list $s(y)$ is not well-sorted. If y was one side of an equation, then an Equation Resolution would have been performed. Therefore, y will be given no value from Syntactic Length Unification. The only substitutions created by Syntactic Length Unification will be of the form $x_{length(y)} = s^n(x_{length(z)})$ with $n \geq 0$ or $x_{length(y)} = s^n(0)$ with $n \geq 0$. Note that a variable cannot appear more than once on a left hand side, or on both a left and right hand side. To get a solution to the original problem, we could just let a variable z appearing on the right hand side be any list. In the first case above, y could be any list with n more elements than z, and in the second case above y could be any list with n elements.

Theorem 4. *The inference system Superposition with Length restrictions followed by Syntactic Length Unification is don't-care nondeterministic and runs in linear time.*

Proof. The inference system is don't-care nondeterministic because the two individual parts of it are. We have shown that whenever there is a Narrowing involving a *car* or *cdr* term then there is also a Rewriting. That is also the case for length. If the argument is a *cons* term or *nil* then there is a Rewriting. If the argument is a variable, then we have disallowed Narrowing. Therefore, the first part is don't-care nondeterministic. The second part is too, because it is just Syntactic Unification.

The first part runs in linear time, because there are no Narrowing inferences. All the Rewrite inferences will result in a term with fewer symbols. And Equation Resolution won't increase the size of the goal if it is kept in dag form. So the first part of the procedure is linear. The second part runs in linear time too, because Syntactic Unification does.

6 Extending Theory with Ground Equations

Now we extend a theory E with a ground theory. A ground set of equations can be flattened so that every equation is of a flattened form, defined as follows.

Definition 6. *A ground equation is* flat *if it is of the form*

1. $f(c_1, \cdots, c_n) \approx c$, *where* $c_1, \cdots, c_n$ *and* c *are constants, or*
2. $c \approx d$, *where* c *and* d *are constants.*

A set of ground equations is flat *if every equation in the set is flat.*

So in this section we will consider an original theory with signature Σ, which will also be the signature of the data structure theory E. Let Σ_C be Σ extended with an infinite set of constants.

We begin with a set of equations E saturated under Superposition, with a signature Σ. We suppose that theory E is augmented with a flat set of ground equations S with signature Σ_C. We want to give conditions under which $E \cup S$ can be finitely saturated under Superposition. We also want to find the conditions under which Superposition can solve E-unification problems for $E \cup S$.

6.1 Automatic Decidability

Our first goal is to find conditions under which this extended theory can be saturated under Superposition. In order to show this for a particular theory, we can use the techniques of [1] automated in [3]. For example, for the theory of Lists with Length, all initial ground equations are of the form $car(a) \approx b$, $cdr(a) \approx b$, $cons(a, b) \approx c$, $length(a) \approx b$, $s(a) \approx b$ or $a \approx b$, where a, b and c are constants. The only additional equations created by Superposition are of the form $s(length(a)) \approx length(b)$ and $length(a) \approx s(b)$ where a and b are constants. This implies that Superposition can be performed in polynomial time.

6.2 *E*-unification with Augmented Theory

Now we consider the problem of E-unification in the list theory L' extended by a flat ground theory S. We allow S and the goal to contain symbols from L' plus additional constants, but no other nonvariable symbols.

Theorem 5. *Let L' be the list theory with set of normal forms N. Let S be a set of flat ground equations. Let S' be the saturation of $E \cup S$. Then Superposition with the addition of Eager Equation Resolution halts on S' in linear time for any goal if Rewriting is performed eagerly, and either a single mgu is generated or the process fails.*

Proof. The proof here is almost identical to the proof of Theorem 2. In this case, we can assume that $Cl(E) = E$, because if e cannot be inferred from an extension $Ext(e)$ of a flat ground equation e, then e will not be able to be used in the goal. So such equations e may be ignored.

Again, we just need to show that whenever Narrowing applies, Rewriting also applies. We consider a Narrowing inference. First suppose it is an inference with $car(cons(x, y)) \approx x$ or $cdr(cons(x, y)) \approx y$. Wlog, suppose it is the first one. Then it Narrows into a term of the form $car(z)$ or $car(cons(z_1, z_2))$. By the argument of Theorem 2, it cannot be a term of the form $car(z)$, so it is a term of the second form, and Rewriting is possible.

Next, suppose we Narrow with a term of the form $car(a) \approx b$ or $cdr(a) \approx b$, where a and b are constants. Wlog, assume it is the first one. Then it must be Narrowing into a term of the form $car(z)$ or $car(a)$. We have already said that it cannot be a term of the form $car(z)$, so it must be a term of the form $car(a)$, and then Rewriting applies.

All of the *car* and *cdr* symbols in a selected disequation will then be removed by eager Rewriting steps. At that point, the only nonvariable symbols left in that disequation will be *cons* terms and constants. At that point, we apply Eager Equation Resolution. So all inferences steps are still don't care nondeterministic or deterministic.

The procedure will still halt in a linear number of steps because all Rewriting steps reduce the number of symbols in a term.

We stress the fact that S is not allowed to contain additional nonvariable symbols that do not exist in E, except for constants, unless for such an extra symbol h, we could conclude that $h_1(x_1, \cdots, x_n) \approx h(y_1, \cdots, y_n) \models x_1 \approx y_1 \wedge \cdots \wedge x_n \approx y_n$. For constants, this statement is trivial.

7 Conclusion

We have given a procedure for solving E unification problems in linear time, if the equational theory E meets certain conditions. The conditions imply that E is closed under Paramodulation which implies that E-unification is in NP for these theories[7, 4]. However it is difficult to get a polynomial time procedure for such theories. To get this result, we needed to specify that some terms have a particular normal form.

Our result applies to the theory of lists, for example. Without our technique, Narrowing is exponential for the theory of lists. We also extended the technique to the theory of Lists plus Length. E-unification in that theory is not closed under Paramodulation, but it has been shown to be decidable by [5]. The [5] result runs in exponential time, while ours is linear. We also showed how to get a polynomial time procedure for E-unification modulo the theory of lists plus any ground theory.

Our results use the standard Superposition calculus, and do not need special inference rules.

Our result is applicable to symbolic debugging problems, where E-unification is required to tell which values for variables determine a particular path through the program. Equations determine the conditions and assignment statements in the program. For *else* statements, we need disequations too. Our results could be extended to cover that. It would also be interesting to consider all paths to a certain point in the program, not just particular paths.

References

1. A. Armando, S. Ranise and M. Rusinowitch. Uniform Derivation of Decision Procedures by Superposition. In *Proceedings 15th Workshop on Computer Science Logic.*, LNCS vol. 2142, 513-527, 2001.
2. L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. In *Journal of Logic and Computation* 4(3), 1-31, 1994.
3. C. Lynch and B. Morawska. Automatic Decidability. In *Proceedings 17th IEEE Symposium on Logic in Computer Science, LICS'02*, IEEE Computer Society Press, 7-16, 2002.
4. C. Lynch and B. Morawska. Basic Syntactic Mutation. In Proceedings of Conference on Automated Deduction (CADE), Vol. 2392 of LNAI, 471–485, 2002.
5. S. Mitra. Semantic Unification for Convergent Systems. Technical Report CS-R-94-1855, University of Illinois at Urbana-Champaign.
6. R. Nieuwenhuis. On Narrowing, Refutation Proofs and Constraints . In J. Hsiang, editor, 6th International Conference on Rewriting Techniques and Applications (RTA) Springer-Verlag LNCS 914, pages 56-70, Kaiserslautern, Germany, April 4-7, 1995. Springer-Verlag.
7. R. Nieuwenhuis. Decidability and Complexity Analysis by Basic Paramodulation . Information and Computation, 147:1-21, 1998.
8. C. Weidenbach. Sorted Unification and Tree Automata in Bibel W. and Schmitt P. H., editors, Automated Deduction - A Basis for Applications, Volume 1 of Applied Logic, Chapter 9, Kluwer, pp. 291-320, 1998

Combining Intruder Theories *

Yannick Chevalier[1] and Michaël Rusinowitch[2]

[1] IRIT Université Paul Sabatier, France
email: ychevali@irit.fr
[2] LORIA – INRIA Lorraine, France
email: rusi@loria.fr

Abstract. Most of the decision procedures for symbolic analysis of protocols are limited to a fixed set of algebraic operators associated with a fixed intruder theory. Examples of such sets of operators comprise XOR, multiplication/exponentiation, abstract encryption/decryption. In this paper we give an algorithm for combining decision procedures for arbitrary intruder theories with disjoint sets of operators, provided that solvability of ordered intruder constraints, a slight generalization of intruder constraints, can be decided in each theory. This is the case for most of the intruder theories for which a decision procedure has been given. In particular our result allows us to decide trace-based security properties of protocols that employ any combination of the above mentionned operators with a bounded number of sessions.

1 Introduction

Recently many procedures have been proposed to decide insecurity of cryptographic protocols in the Dolev-Yao model w.r.t. a finite number of protocol sessions [2, 5, 19, 17]. Among the different approaches the symbolic ones [17, 10, 4] are based on reducing the problem to constraint solving in a term algebra. This reduction has proved to be quite effective on standard benchmarks and also was able to discover new flaws on several protocols [4].

However while most formal analysis of security protocols abstracts from low-level properties, i.e., certain algebraic properties of encryption, such as the multiplicativity of RSA or the properties induced by chaining methods for block ciphers, many real attacks and protocol weaknesses rely on these properties. For attacks exploiting the *XOR* properties in the context of mobile communications see [7]. Also the specification of *Just Fast Keying* protocol (an alternative to IKE) in [1] employs a set constructor that is idempotent and commutative and a Diffie-Hellman exponentiation operator with the property $(g^y)^z = (g^z)^y$.

In this paper we present a general procedure for deciding security of protocols in presence of algebraic properties. This procedure relies on the combination of constraint solving algorithm for disjoint intruder theories, provided that solvability of ordered intruder constraints, a slight generalization of intruder constraints, can be decided in each theory. Such combination algorithm already exists for solving E-unification problems [20, 3]. We have extended it in order to solve intruder constraints on disjoint signatures. This extension is non trivial since intruder deduction rules allow one to build *contexts* above terms and therefore add some second-order features to the *standard* first-order E-unification problem.

* supported by IST AVISPA Project, ACI-SI SATIN, ACI-Jeune Chercheur

Our approach is more modular than the previous ones and it allows us to decide interesting intruder theories that could not be considered before by reducing them to simpler and independant theories. For instance it allows one to combine the exponentiation with abelian group theory of [18] with the Xor theory of [8].

Related works. Recently several protocol decision procedures have been designed for handling algebraic properties in the Dolev-Yao model [16, 6, 11, 8]. These works have been concerned by fixed equational theories corresponding to a fixed intruder power. A couple of works only have tried to derive generic decidability results for *class* of intruder theories. For instance, in [12] Delaune and Jacquemard consider the class of *public collapsing* theories. These theories have to be presented by rewrite systems where the right-hand side of every rule is a ground term or a variable, which is a strong restriction.

2 Motivation

Combination of algebraic operators. We consider in this section the Needham-Schroeder Public-Key protocol. This well-known protocol is described in the Alice and Bob notation by the following sequence of messages, where the comma denotes a pairing of messages and $\{M\}K_a$ denotes the encryption by the public key K_a of A.

$$\begin{array}{l} A \rightarrow B : \{A, N_a\}K_b \\ B \rightarrow A : \{N_a, N_b\}K_a \\ A \rightarrow B : \{N_b\}K_b \end{array}$$

Assume now that the encryption algorithm follows El-Gamal encryption scheme. The public key of A is defined by three publicly-available parameters: a modulus p_a, a base g_a and the proper public key $(g_a)^a \mod p_a$. The private key of A is a. Denoting $\exp_p$ the exponentiation modulo p and $\times_p$ the multiplication modulo $\varphi(p)$, and with new nonces k_1, k_2 and k_3 we can rewrite the protocol as:

$$\begin{array}{l} A \rightarrow B : \exp_{p_b}(g_b, k_1), (A, N_a) \oplus \exp_{p_b}(\exp_{p_b}(g_b, b), k_1) \\ B \rightarrow A : \exp_{p_a}(g_a, k_2), (N_a, N_b) \oplus \exp_{p_a}(\exp_{p_a}(g_a, a), k_2) \\ A \rightarrow B : \exp_{p_b}(g_b, k_3), (N_b) \oplus \exp_{p_b}(\exp_{p_b}(g_b, b), k_3) \end{array}$$

In this simple example we would like to model the group properties of the Exclusive-or ($\oplus$), the associativity of exponential ($(x^y)^z = x^{y \times z}$), the group property of the exponents. Several works have already been achieved toward taking into account these algebraic properties for detecting attacks on a bounded number of sessions. However none of these works can analyse protocols combining several algebraic operators like the example above. The algorithm given in this paper will permit to decide the trace-based security properties of such protocols.

Examples of intruder theories. A convenient way to specify intruder theories in the context of cryptographic protocols is by giving a set L of *deduction rules* that tell how the intruder can construct new messages from the ones she already knows and a set of *equational laws* $\mathcal{E}$ that are verified by the functions that are employed in messages. We give here two examples of intruder theories. Some other theories are given in 6.

Abelian group theory. This intruder may treat messages as elements of an abelian group. We assume here there is only one such group and that the composition law is $\cdot \times \cdot$, the inverse law is $\mathrm{i}(\cdot)$ and the neutral element is denoted 1.

$$L_\times \begin{cases} \to 1 \\ x \to \mathrm{i}(x) \\ x, y \to x \times y \end{cases} \qquad \mathcal{E}_\times \begin{cases} (x \times y) \times z = x \times (y \times z) \\ x \times y = y \times x \\ 1 \times x = x \\ x \times \mathrm{i}(x) = 1 \end{cases}$$

Dolev Yao with explicit destructors. The intruder is given with a pairing operator and projections to retrieve the components of a pair. There is a symmetric encryption operator $\mathrm{se}(\_,\_)$ and an operator $\mathrm{sd}(\_,\_)$ for the decryption algorithm too. For conciseness we omit the public-key encryption specification.

$$L_{DY} \begin{cases} x, y \to \langle x, y \rangle \\ x \to \pi_1(x) \\ x \to \pi_2(x) \\ x, y \to \mathrm{se}(x, y) \\ x, y \to \mathrm{sd}(x, y) \end{cases} \qquad \mathcal{E}_{DY} \begin{cases} \pi_1(\langle x, y \rangle) = x \\ \pi_2(\langle x, y \rangle) = y \\ \mathrm{sd}(\mathrm{se}(x, y), y) = x \end{cases}$$

3 Terms and subterms

We consider an infinite set of free constants C and an infinite set of variables $\mathcal{X}$. For all signatures $\mathcal{G}$ (i.e. a set of function symbols with arities), we denote by $\mathrm{T}(\mathcal{G})$ (resp. $\mathrm{T}(\mathcal{G}, \mathcal{X})$) the set of terms over $\mathcal{G} \cup \mathrm{C}$ (resp. $\mathcal{G} \cup \mathrm{C} \cup \mathcal{X}$). The former is called the set of ground terms over $\mathcal{G}$, while the later is simply called the set of terms over $\mathcal{G}$. Variables are denoted by x, y, terms are denoted by s, t, u, v, and finite sets of terms are written $E, F, ...$, and decorations thereof, respectively. We abbreviate $E \cup F$ by E, F, the union $E \cup \{t\}$ by E, t and $E \setminus \{t\}$ by $E \setminus t$.

A *constant* is either a free constant or a function symbol of arity 0. Given a term t we denote by $\mathrm{Var}(t)$ the set of variables occurring in t and by $\mathrm{Cons}(t)$ the set of constants occurring in t. We denote by $\mathrm{Atoms}(t)$ the set $\mathrm{Var}(t) \cup \mathrm{Cons}(t)$. We denote by $\mathcal{A}$ the set of all constants and variables. A substitution σ is an involutive mapping from $\mathcal{X}$ to $\mathrm{T}(\mathcal{G}, \mathcal{X})$ such that $\mathrm{Supp}(\sigma) = \{x | \sigma(x) \neq x\}$, the *support* of σ, is a finite set. The application of a substitution σ to a term t (resp. a set of terms E) is denoted $t\sigma$ (resp. $E\sigma$) and is equal to the term t (resp. E) where all variables x have been replaced by the term $x\sigma$. A substitution σ is *ground* w.r.t. $\mathcal{G}$ if the image of $\mathrm{Supp}(\sigma)$ is included in $\mathrm{T}(\mathcal{G})$.

In this paper, we consider 2 disjoint signatures $\mathcal{F}_1$ and $\mathcal{F}_2$, a consistent equational theory $\mathcal{E}_1$ (resp. $\mathcal{E}_2$) on $\mathcal{F}_1$ (resp. $\mathcal{F}_2$). We denote by $\mathcal{F}$ the union of the signatures $\mathcal{F}_1$ and $\mathcal{F}_2$, $\mathcal{E}$ the union of the theories $\mathcal{E}_1$ and $\mathcal{E}_2$. A term t in $\mathrm{T}(\mathcal{F}_1, \mathcal{X})$ (resp. in $\mathrm{T}(\mathcal{F}_2, \mathcal{X})$) is called a *pure* 1-term (resp. a pure 2-term).

The *syntactic subterms* of a term t are defined recursively as follows and denoted $\mathrm{Sub}_{\mathrm{syn}}(t)$. If t is a variable or a constant then $\mathrm{Sub}_{\mathrm{syn}}(t) = \{t\}$. If

$t = f(t_1, \ldots, t_n)$ then $\mathrm{Sub}_{\mathrm{syn}}(t) = \{t\} \cup \bigcup_{i=1}^{n} \mathrm{Sub}_{\mathrm{syn}}(t_i)$. The *positions* in a term t are defined recursively as usual (*i.e.* as sequences of integers), ϵ being the empty sequence. We denote by $t_{|p}$ the syntactic subterm of t at position p. We denote by $t[p \leftarrow s]$ the term obtained by replacing in t the syntactic subterm $t_{|p}$ by s. We denote by $\mathrm{Sign}(\cdot)$ the function that associates to each term $t \notin \mathrm{C} \cup \mathcal{X}$ the signature ($\mathcal{F}_1$, or $\mathcal{F}_2$) of its root symbol $t_{|\epsilon}$. For $t \in \mathrm{C} \cup \mathcal{X}$ we define $\mathrm{Sign}(t) = \perp$, with $\perp$ a new symbol. The term s is *alien* to u if $\mathrm{Sign}(s) \neq \mathrm{Sign}(u)$. *Factors.* We define the set of *factors* of a term t, and note it $\mathrm{Factors}(t)$, the set of maximal syntactic subterms of t that are alien to t and different of t. In particular $\mathrm{Factors}(t) = \emptyset$ for $t \in \mathcal{A}$.
Subterms. We now define the notion of *subterm values*. Given a term t, the set of its subterm values is denoted by $\mathrm{Sub}(t)$ and is defined recursively by: $\mathrm{Sub}(t) = \{t\} \cup \bigcup_{u \in \mathrm{Factors}(t)} \mathrm{Sub}(u)$. For a set of terms E, $\mathrm{Sub}(E)$ is defined as the union of the subterms values of the elements of E.

As an example consider $\mathcal{F}_1 = \{\oplus, a, b, c\}$ and $\mathcal{F}_2 = \{f\}$ where f has arity 1. Then $\mathrm{Sub}(a \oplus (b \oplus c)) = \{a \oplus (b \oplus c), a, b, c\}$. On the other hand $\mathrm{Sub}(f(b \oplus c)) = \{f(b \oplus c), b \oplus c, b, c\}$. This shows the difference with the notion of *syntactic subterms*. In the rest of this paper and unless otherwise indicated, *the notion of subterm will refer to subterm values.*

Congruences and ordered rewriting. We shall introduce the notion of *ordered rewriting* [13], which is a useful tool that has been utilized (e.g. [3]) for proving the correctness of combination of unification algorithms.

Let $<$ be a simplification ordering on $\mathrm{T}(\mathcal{G})$ [3] assumed to be total on $\mathrm{T}(\mathcal{G})$ and such that the minimum for $<$ is a constant $\mathrm{c}_{\min} \in \mathrm{C}$. Given a possibly infinite set of equations $\mathcal{O}$ on the signature $\mathrm{T}(\mathcal{G})$ we define the ordered rewriting relation $\rightarrow_{\mathcal{O}}$ by $s \rightarrow_{\mathcal{O}} s'$ iff there exists a position p in s, an equation $l = r$ in $\mathcal{O}$ and a substitution τ such that $s = s[p \leftarrow l\tau]$, $s' = s[p \leftarrow r\tau]$, and $l\tau > r\tau$.

It has been shown (see [13]) that by applying the *unfailing completion procedure* [15] to a set of equations $\mathcal{H}$ we can derive a (possibly infinite) set of equations $\mathcal{O}$ such that:

1. the congruence relations $=_{\mathcal{O}}$ and $=_{\mathcal{H}}$ are equal on $\mathrm{T}(\mathcal{F})$.
2. $\rightarrow_{\mathcal{O}}$ is convergent (i.e. terminating and confluent) on $\mathrm{T}(\mathcal{F})$.

We shall say that $\mathcal{O}$ is an *o-completion* of $\mathcal{H}$.

The relation $\rightarrow_{\mathcal{O}}$ being convergent on ground terms we can define $(t){\downarrow_O}$ as the unique normal form of the ground term t for $\rightarrow_{\mathcal{O}}$. Given a ground substitution σ we denote by $(\sigma){\downarrow_O}$ the substitution with the same support such that for all variables $x \in \mathrm{Supp}(\sigma)$ we have $x(\sigma){\downarrow_O} = (x\sigma){\downarrow_O}$. A substitution σ is *normal* if $\sigma = (\sigma){\downarrow_O}$.

We will denote by R an o-completion of $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2$. We denote by $\mathrm{C}_{\mathrm{spe}}$ the set containing the constants in $\mathcal{F}$ and $\mathrm{c}_{\min}$.

[3] by definition $<$ satisfies for all $s, t, u \in \mathrm{T}(\mathcal{G})$ $s < t[s]$ and $s < u$ implies $t[s] < t[u]$

4 Protocols, intruders and constraint systems

Security of a given protocol is assessed with respect to a class of environments in which the protocol is executed. Dolev and Yao have described the environment not in terms of possible attacks on the protocol but by the deduction an intruder attacking a protocol execution is able to perform.

In Subsection 4.1 we define an extension of Dolev-Yao model to arbitrary operators for modeling the possible deductions of the intruder. In Subsection 4.2 we define the protocol semantics for an execution within an hostile environment controlled by the intruder and in Subsection 4.3 we describe how we represent this execution by a constraint system.

4.1 Intruder deduction systems

We shall model messages as ground terms and intruders deduction rules as rewrite rules on sets of messages representing the knowledge of an intruder. The intruder derives new messages from a given (finite) set of messages by applying intruder rules. Since we assume some equational axioms $\mathcal{H}$ are satisfied by functions symbols in the signature, all these derivations have to be considered *modulo* the equational congruence $=_{\mathcal{H}}$ generated by these axioms.

An intruder deduction rule in our setting is specified by a term t in some signature $\mathcal{G}$. Given values for the variables of t the intruder is able to generate the corresponding instance of t.

Definition 1. *An* intruder system $\mathcal{I}$ *is given by a triple* $\langle \mathcal{G}, T, \mathcal{H} \rangle$ *where* $\mathcal{G}$ *is a signature,* $T \subseteq \mathrm{T}(\mathcal{G}, \mathcal{X})$ *and* $\mathcal{H}$ *is a set of axioms between terms in* $\mathrm{T}(\mathcal{G}, \mathcal{X})$. *To each* $t \in T$ *we associate a* deduction rule $\mathrm{L}^t : \mathrm{Var}(t) \to t$ *and* $\mathrm{L}^{t,\mathrm{g}}$ *denotes the set of ground instances of the rule* L^t*:*

$$\mathrm{L}^{t,\mathrm{g}} = \{l \to r \mid \exists \sigma, \textit{ground substitution on } \mathcal{G},\ l = \mathrm{Var}(t)\sigma \text{ and } r =_{\mathcal{H}} t\sigma\}$$

The set of rules $\mathrm{L}_{\mathcal{I}}$ *is defined as the union of the sets* $\mathrm{L}^{t,\mathrm{g}}$ *for all* $t \in T$.

Each rule $l \to r$ in $\mathrm{L}_{\mathcal{I}}$ defines an intruder deduction relation $\to_{l \to r}$ between finite sets of terms. Given two finite sets of terms E and F we define $E \to_{l \to r} F$ if and only if $l \subseteq E$ and $F = E \cup \{r\}$. We denote $\to_{\mathcal{I}}$ the union of the relations $\to_{l \to r}$ for all $l \to r$ in $L_{\mathcal{I}}$ and by $\to_{\mathcal{I}}^*$ the transitive closure of $\to_{\mathcal{I}}$. We simply denote by $\to$ the relation $\to_{\mathcal{I}}$ when there is no ambiguity about $\mathcal{I}$.

For instance we can define $\mathcal{I}_\times = \langle \{\times, \mathrm{i}, 1\}, \{x \times y, \mathrm{i}(x), 1\}, \mathcal{E}_\times \rangle$ and we have $a, b, c \to_{\mathcal{I}_\times} a, b, c, c \times a$.

A *derivation* D of length n, $n \geq 0$, is a sequence of steps of the form $E_0 \to_{\mathcal{I}} E_0, t_1 \to_{\mathcal{I}} \cdots \to_{\mathcal{I}} E_n$ with finite sets of ground terms $E_0, \ldots E_n$, and ground terms $t_1, \ldots, t_n$, such that $E_i = E_{i-1} \cup \{t_i\}$ for every $i \in \{1, \ldots, n\}$. The term t_n is called the *goal* of the derivation. We define $\overline{E}^{\mathcal{I}}$ to be equal to the set $\{t \mid \exists F \text{ s.t. } E \to_{\mathcal{I}}^* F \text{ and } t \in F\}$ *i.e.* the set of terms that can be derived from E. If there is no ambiguity on the deduction system $\mathcal{I}$ we write $\overline{E}$ instead of $\overline{E}^{\mathcal{I}}$.

Let $\mathcal{O}$ be an o-completion of $\mathcal{H}$. We will assume from now that all the deduction rules generate terms that are normalized by $\rightarrow_{\mathcal{O}}$ and the goal and the initial set are in normal form for $\rightarrow_{\mathcal{O}}$. It can be shown [9] that this is not restrictive for our main decidability result.

Given a set of terms $T \subseteq \mathrm{T}(\mathcal{G}, \mathcal{X})$ we define the set of terms $\langle T \rangle$ to be the minimal set such that $T \subseteq \langle T \rangle$ and for all $t \in \langle T \rangle$ and for all substitutions σ with image included in $\langle T \rangle$, we have $t\sigma \in \langle T \rangle$. Hence terms in $\langle T \rangle$ are built by composing terms in T iteratively. We can prove easily that the intruder systems $\mathcal{I} = \langle \mathcal{G}, T, \mathcal{H} \rangle$ and $\mathcal{J} = \langle \mathcal{G}, \langle T \rangle, \mathcal{H} \rangle$ define the same sets of derivable terms, i.e. for all E we have $\overline{E}^{\mathcal{I}} = \overline{E}^{\mathcal{J}}$.

We want to consider now the union of 2 intruder systems: $\mathcal{I}_1 = \langle \mathcal{F}_1, T_1, \mathcal{E}_1 \rangle$ and $\mathcal{I}_2 = \langle \mathcal{F}_2, T_2, \mathcal{E}_2 \rangle$. In particular we are interested in the derivations obtained by using $\rightarrow_{\mathcal{I}_1} \cup \rightarrow_{\mathcal{I}_2}$. It can be noticed that $\langle T_1 \cup T_2 \rangle = \langle \langle T_1 \rangle \cup \langle T_2 \rangle \rangle$. Hence by the remarks above the derivable terms using $\langle T_1 \cup T_2 \rangle$ or $\langle T_1 \rangle \cup \langle T_2 \rangle$ are the same. For technical reason it will be more convenient to use $\langle T_1 \rangle \cup \langle T_2 \rangle$ for defining the union of 2 intruder systems:

Definition 2. *The* union of the two intruder systems $\langle \mathcal{F}_1, T_1, \mathcal{E}_1 \rangle, \langle \mathcal{F}_2, T_2, \mathcal{E}_2 \rangle$ *is the intruder system* $\mathcal{U} = \langle \mathcal{F}_1 \cup \mathcal{F}_2, \langle T_1 \rangle \cup \langle T_2 \rangle, \mathcal{E}_1 \cup \mathcal{E}_2 \rangle$.

A derivation $E_0 \rightarrow_{\mathcal{U}} E_0, t_1 \rightarrow_{\mathcal{U}} \cdots \rightarrow_{\mathcal{U}} E_n$ of intruder system $\mathcal{U}$ is *well-formed* if for all $i \in \{1, \ldots, n\}$ we have $t_i \in \mathrm{Sub}(E_0, t_n)$; in other words every message generated by an intermediate step either occurs in the goal or in the initial set of messages. In the following lemma the derivations refer to the intruder system $\mathcal{U} = \langle \mathcal{F}_1 \cup \mathcal{F}_2, \langle T_1 \rangle \cup \langle T_2 \rangle, \mathcal{E}_1 \cup \mathcal{E}_2 \rangle$. For the proof see [9]:

Lemma 1. *A derivation of minimal length starting from E of goal t is well-formed.*

4.2 Protocol analysis

In this subsection we describe how protocols are modelled. In the following we only model a single session of the protocol since it is well-known how to reduce several sessions to this case. Our semantics follows the one by [12].

In Dolev-Yao model the intruder has complete control over the communication medium. We model this by considering the intruder *is* the network. Messages sent by honest agents are sent directly to the intruder and messages received by the honest agents are always sent by the intruder. From the intruder side a finite execution of a protocol is the interleaving of a finite sequence of messages it has to send and a finite sequence of messages it receives (and add to its knowledge).

We also assume that the interaction of the intruder with one agent is an atomic step. The intruder sends a message m to an honest agent, this agent tests the validity of this message and responds to it. Alternatively an agent may initiate an execution and in this case we assume it reacts to a dummy message sent by the intruder.

A *step* is a triplet $(\textsc{recv}(x); \textsc{send}(s); \textsc{cond}(e))$ where $x \in \mathcal{X}$, $s \in \mathrm{T}(\mathcal{G}, \mathcal{X})$ and e is a set of equations between terms of $\mathrm{T}(\mathcal{G}, \mathcal{X})$. The meaning of a step is that upon receiving message x, the honest agent checks the equations in e and sends the message s. An execution of a protocol is a finite sequence of steps.

Example 1. Consider the following simple protocol:

$A \to B : \{M \oplus B\}_K$
$B \to A : B$
$A \to B : K$
$B \to A : M$

Assuming the algebraic properties of $\oplus$, symmetric encryption se(,) and symmetric decryption sd(,) we model this protocol as:

$\text{RECV}(v_1); \text{SEND}(\text{se}(M \oplus B, K)); \text{COND}(v_1 = \text{c}_{\min})$
$\text{RECV}(v_2); \text{SEND}(B); \text{COND}(\emptyset)$
$\text{RECV}(v_3); \text{SEND}(K); \text{COND}(v_3 = B)$
$\text{RECV}(v_4); \text{SEND}(\text{sd}(v_2, v_4) \oplus B); \text{COND}(v_4 = K)$
$\text{RECV}(v_5); \text{SEND}(\text{c}_{\min}); \text{COND}(v_5 = M)$

Note that in our setting we can model that at some step i the message must match the pattern t_i by adding an equation $v_i \stackrel{?}{=} t_i$ to $\mathcal{S}$.

In order to define whether an execution of a protocol is feasible we must first define when a substitution σ satisfies a set of equations $\mathcal{S}$.

Definition 3. *(Unification systems) Let $\mathcal{H}$ be a set of axioms on* $\mathrm{T}(\mathcal{G}, \mathcal{X})$*. An* $\mathcal{H}$-Unification system $\mathcal{S}$ *is a finite set of equations in* $\mathrm{T}(\mathcal{G}, \mathcal{X})$ *denoted by* $(u_i \stackrel{?}{=} v_i)_{i \in \{1,\ldots,n\}}$*. It is satisfied by a ground substitution σ, and we note $\sigma \models \mathcal{S}$, if for all $i \in \{1, \ldots, n\}$ $u_i\sigma =_{\mathcal{H}} v_i\sigma$.*

Let $\mathcal{I}$=$\langle \mathcal{G}, T, \mathcal{H} \rangle$ be an intruder system. A *configuration* is a couple $\langle P, N \rangle$ where P is a finite sequence of steps and N is a set of ground terms (the knowledge of the intruder). From the configuration $\langle (\text{RECV}(x); \text{SEND}(s); \text{COND}(e)) \cdot P, N \rangle$ a transition to (P', N') is possible iff there exists a ground substitution σ such that $x\sigma \in \overline{N}^{\mathcal{I}}$, $\sigma \models e$, $N' = N \cup \{s\sigma\}$ and $P' = P\sigma$. Trace based-security properties like secrecy can be reduced to the following *Execution feasability* problem.

Execution feasability

Input:	an initial configuration $\langle P, N_0 \rangle$
Output:	SAT iff there exists a reachable configuration $\langle \emptyset, M \rangle$

4.3 Constraints systems

We express the execution feasability of a protocol by a constraint problem $\mathcal{C}$.

Definition 4. *(Constraint systems) Let $\mathcal{I} = \langle \mathcal{G}, T, \mathcal{H} \rangle$ be an intruder system. An* $\mathcal{I}$-Constraint system $\mathcal{C}$ *is denoted: $((E_i \rhd v_i)_{i \in \{1,\ldots,n\}}, \mathcal{S})$ and it is defined by a sequence of couples $(E_i, v_i)_{i \in \{1,\ldots,n\}}$ with $v_i \in \mathcal{X}$ and $E_i \subseteq \mathrm{T}(\mathcal{G})$ for $i \in \{1, \ldots, n\}$ and $E_{i-1} \subseteq E_i$ for $i \in \{2, \ldots, n\}$ and by an $\mathcal{H}$-unification system $\mathcal{S}$.*

An $\mathcal{I}$-Constraint system $\mathcal{C}$ *is satisfied by a ground substitution σ if for all $i \in \{1, \ldots, n\}$ we have $v_i\sigma \in \overline{E_i\sigma}$ and if $\sigma \models \mathcal{S}$. If a ground substitution σ satisfies a constraint system $\mathcal{C}$ we denote it by $\sigma \models_{\mathcal{I}} \mathcal{C}$.*

Constraint systems are denoted by $\mathcal{C}$ and decorations thereof. Note that if a substitution σ is a solution of a constraint system $\mathcal{C}$, by definition of constraints and of unification systems the substitution $(\sigma)\downarrow_O$ is also a solution of $\mathcal{C}$ (where $\mathcal{O}$ is an o-completion of H). In the context of cryptographic protocols the inclusion $E_{i-1} \subseteq E_i$ means that the knowledge of an intruder does not decrease as the protocol progresses: after receiving a message an honest agent will respond to it. This response can be added to the knowledge of an intruder who listens all communications.

Example 2. We model the protocol of Example 1 by the following constraint system. First we gather all conditions in an unification system $\mathcal{S}$

$$\mathcal{S} = \left\{ v_1 \stackrel{?}{=} \mathrm{c}_{\min} \ , \ v_3 \stackrel{?}{=} B \ , \ v_4 \stackrel{?}{=} K \ , \ v_5 \stackrel{?}{=} M \right\}$$

The protocol execution for intruder $\mathcal{I}$ with initial knowledge $\{\mathrm{c}_{\min}\}$ is then expressed by the constraint:

$$\begin{aligned} \mathcal{C} = ((\ & \mathrm{c}_{\min} \rhd v_1, \\ & \mathrm{c}_{\min}, \mathrm{se}(M \oplus B, K) \rhd v_2, \\ & \mathrm{c}_{\min}, \mathrm{se}(M \oplus B, K), B \rhd v_3, \\ & \mathrm{c}_{\min}, \mathrm{se}(M \oplus B, K), B, K \rhd v_4), \\ & \mathrm{c}_{\min}, \mathrm{se}(M \oplus B, K), B, K, \mathrm{sd}(v_2, v_4) \oplus B \rhd v_5, \mathcal{S}) \end{aligned}$$

We are not interested in general constraint systems but only in those related to protocols. In particular we need to express that a message to be sent at some step i should be built from previously received messages recorded in the variables $v_j, j < i$, and from the initial knowledge. To this end we define:

Definition 5. *(Deterministic Constraint Systems) An $\mathcal{I}$-constraint system $((E_i \rhd v_i)_{i\in\{1,\dots,n\}}, \mathcal{S})$ is* deterministic *if for all $i \in \{1, \dots, n\}$ we have* $\mathrm{Var}(E_i) \subseteq \{v_1, \dots, v_{i-1}\}$

The decision problems we are interested in are the *satisfiability* and the *ordered satisfiability* of intruder constraint systems.

Satisfiability

Input: an $\mathcal{I}$-constraint system $\mathcal{C}$
Output: SAT iff there exists a substitution σ such that: $\sigma \models_{\mathcal{I}} \mathcal{C}$.

In order to be able to combine solutions of constraints in component theories to get a solution for the full theory these solutions have to satisfy some ordering constraints too. Intuitively, this is to avoid introducing cycle when building a global solution. This motivates the following definition:

Ordered Satisfiability

Input: an $\mathcal{I}$-constraint system $\mathcal{C}$, X the set of all variables and C the set of all free constants occurring in $\mathcal{C}$ and a linear ordering $\prec$ on $X \cup C$.

Output: SAT iff there exists a substitution σ such that:

$$\begin{cases} \sigma \models_{\mathcal{I}} \mathcal{C} \\ \forall x \in X \text{ and } \forall c \in C, x \prec c \text{ implies } c \notin \mathrm{Sub}_{\mathrm{syn}}(x\sigma) \end{cases}$$

The main result of this paper is the following modularity result:

Theorem 1. *If the ordered satisfiability problem is decidable for two intruders $\langle \mathcal{F}_1, T_1, \mathcal{E}_1 \rangle$ and $\langle \mathcal{F}_2, T_2, \mathcal{E}_2 \rangle$ for disjoint signatures $\mathcal{F}_1$ and $\mathcal{F}_2$ then the satisfiability problem is decidable for the intruder $\mathcal{U} = \langle \mathcal{F}_1 \cup \mathcal{F}_2, \langle T_1 \rangle \cup \langle T_2 \rangle, \mathcal{E}_1 \cup \mathcal{E}_2 \rangle$.*

This result is obtained as a direct consequence of the next section where we give an algorithm for solving $\mathcal{U}$-constraints using algorithms for solving *ordered satisfiability* for intruders $\langle \mathcal{F}_1, T_1, \mathcal{E}_1 \rangle$ and $\langle \mathcal{F}_2, T_2, \mathcal{E}_2 \rangle$.

5 Combination of decision procedures

We introduce Algorithm 1 for solving satisfiability of constraint systems for the union $\mathcal{U}$ of two intruders systems $\mathcal{I}_1 = \langle \mathcal{F}_1, T_1, \mathcal{E}_1 \rangle$ and $\mathcal{I}_2 = \langle \mathcal{F}_2, T_2, \mathcal{E}_2 \rangle$ with disjoint signatures $\mathcal{F}_1$ and $\mathcal{F}_2$. The completeness of Algorithm 1 is sketched below, and the proofs (for completeness and soundness) are fully detailed in [9]. Let us explain this algorithm:

Step 2 The algorithm input is a $\mathcal{U}$-Constraint system $(\mathcal{D}, \mathcal{S})$. An equational system $\mathcal{S}$ is *homogeneous* if for all $u \stackrel{?}{=} v \in \mathcal{S}$, u and v are both pure 1-terms or both pure 2-terms. It is well-known that equational systems can be transformed into equivalent (w.r.t. satisfiability) homonogeneous systems. Thus we can assume that $\mathcal{S}$ is homogeneous without loss of generality.

Step 3 abstracts every subterm t of $\mathcal{C}$ by a new variable $\psi(t)$. A choice of ψ such that $\psi(t) = \psi(t')$ will lead to solutions that identify t and t'.

Steps 4-6 assign non-deterministically a signature to the root symbol of the subterms of $\mathcal{C}$ instanciated by a solution. The choice $th(\psi(t)) = 0$ corresponds to the situation where t gets equal to a free constant.

Steps 7-10 choose and order non-deterministically the intermediate subterms in derivations that witness that the solution satisfies the constraints in $\mathcal{D}$.

Step 11 defines a constraint problem $\mathcal{C}'$ collecting the previous choices on subterms identification, subterms signatures and derivation structures.

Step 12 splits the problem $\mathcal{S}'$ in two pure subproblems.

Step 13 splits non-deterministically the problem $\mathcal{D}'$, that is we select for each $E \rhd v$ in $\mathcal{D}'$ an intruder system to solve it.

Step 14 guesses an ordering on variables: this ordering will preclude the value of a variable from being a subterm of the value of a smaller variable. This is used to avoid cycles in the construction of the solution.

Step 15 solves independantly the 2 pure subproblems obtained at steps 12-13. In $\mathcal{C}_i$ the variables q with $th(q) \neq i$ will be considered as constants.

We assume $\mathrm{C}_{\mathrm{spe}} \subseteq \mathrm{Sub}(\mathcal{C})$. Recall that R is the rewrite system associated to $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2$. We say a normal substitution σ is *bound* if for all variables x with $x\sigma \neq x$ and for all $t \in \mathrm{Sub}(x\sigma)$ there exists $u \in \mathrm{Sub}(\mathcal{C}) \cup \mathrm{C}_{\mathrm{spe}}$ such that $(u\sigma){\downarrow_R} = t$. A key proposition is:

Algorithm 1 Combination Algorithm

1: **Solve**$_{\mathcal{U}}(\mathcal{C})$
2: **Let** $\mathcal{C} = ((E_i \rhd v_i)_{i\in\{1,\ldots,n\}}, \mathcal{S})$ with $\mathcal{S}$ homogeneous.
3: **Choose** ψ an application from $\mathrm{Sub}(C)$ to $\mathcal{X} \setminus \mathrm{Var}(\mathcal{C})$
and let $Q = \psi(\mathrm{Sub}(C))$
4: **for all** $q \in Q$ **do**
5: Choose a theory $th(q) \in \{0, 1, 2\}$
6: **end for**
7: **for** $i = 1$ to n **do**
8: Choose $Q_i \subseteq Q$
9: Choose a linear ordering over the elements of Q_i say $(q_{i,1}, \ldots, q_{i,k_i})$
10: **end for**
11: **Let** $\mathcal{C}' = (\mathcal{D}', \mathcal{S}')$ where

$$\begin{cases} \mathcal{S}' = \mathcal{S} \cup \left\{ z \stackrel{?}{=} \psi(z) \mid z \in \mathrm{Sub}(C) \right\} \\ \mathcal{D}' = \Delta_1, \ldots, \Delta_i, \ldots \Delta_n \end{cases}$$

and $\Delta_i = (K_i, Q_i^{<j} \rhd q_{i,j})_{j\in\{1,\ldots,k_i\}},\ (K_i, Q_i \rhd \psi(v_i))$ with

$$\begin{cases} K_i = \psi(E_i) \cup \bigcup_{j=1}^{i-1} Q_j \\ Q_i^{<j} = q_{i,1}, q_{i,2}, \ldots, q_{i,j-1} \end{cases}$$

12: **Split** $\mathcal{S}'$ into $\mathcal{S}_1, \mathcal{S}_2$ such that $\mathcal{S}' = \mathcal{S}_1 \cup \mathcal{S}_2$ and:

$$\begin{cases} \mathcal{S}_1 = \left\{ z \stackrel{?}{=} z' \in \mathcal{S}' \mid z, z' \text{are pure 1-terms} \right\} \\ \mathcal{S}_2 = \left\{ z \stackrel{?}{=} z' \in \mathcal{S}' \mid z, z' \text{are pure 2-terms} \right\} \end{cases}$$

13: **Split** non-deterministically $\mathcal{D}'$ into $\mathcal{D}_1, \mathcal{D}_2$
14: **Choose** a linear ordering $\prec$ over Q.
15: **Solve** $\mathcal{C}_i = (\mathcal{D}_i, \mathcal{S}_i)$ for intruder $\mathcal{I}_i$ with linear ordering $\prec$ for $i \in \{1, 2\}$
16: **if** both are satisfied **then**
17: **Output:** SATISFIED
18: **end if**

Proposition 1. *If $\mathcal{C}$ is a satisfiable constraint system there exists a bound substitution σ such that $\sigma \models \mathcal{C}$. Moreover* $\mathrm{Sub}((\mathrm{Sub}(\mathcal{C})\sigma)\downarrow_R) = (\mathrm{Sub}(\mathcal{C})\sigma)\downarrow_R \cup \mathrm{C}_{\mathrm{spe}}$.

5.1 Completeness of the algorithm

Proposition 2. *If $\mathcal{C}$ is satisfiable then there exists $\mathcal{C}_1$ and $\mathcal{C}_2$ satisfiable at Step 15 of the algorithm.*

Proof. First let us prove that the 11 first steps of the algorithm preserve satisfiability. Assume $\mathcal{C}$ is satisfiable. By Proposition 1 there exists a normal bound substitution σ which satisfies $\mathcal{C}$. Define ψ to be a function from $\mathrm{Sub}(\mathcal{C})$ to a set of variables Q such that $\psi(t) = \psi(t')$ if and only if $(t\sigma)\downarrow_R = (t'\sigma)\downarrow_R$. Thus by Proposition 1 there exists a bijection ϕ from Q to $\mathrm{Sub}((\mathrm{Sub}(C)\sigma)\downarrow_R)$. We let $th(q) = i$ if $\mathrm{Sign}(\phi(q)) = \mathcal{F}_i$ and $th(q) = 0$ if $\mathrm{Sign}(\phi(q)) = \perp$. By the construction of $\mathcal{S}'$ and the choice of ψ we can extend σ on Q by $q\sigma = (\psi^{-1}(q)\sigma)\downarrow_R$.

For each $i \in \{1, \ldots, n\}$ by Lemma 1 we can consider a well-formed derivation D_i starting from $F_i = (E_i\sigma)\downarrow_R$ and of goal $g_i = v_i\sigma$:

$$D_i : F_i \rightarrow_{\mathcal{U}} F_i, r_{i,1} \rightarrow_{\mathcal{U}} \cdots \rightarrow_{\mathcal{U}} F_i, r_{i,1}, \ldots, r_{i,k_i} \rightarrow_{\mathcal{U}} F_i, r_{i,1}, \ldots, r_{i,k_i}, g_i$$

We have $\text{Sub}(F_i, g_i) \subseteq \text{Sub}((\text{Sub}(\mathcal{C}\sigma))\downarrow_R)$. Since the derivation is well-formed we have $\{r_{i,1}, \ldots, r_{i,k_i}\} \subseteq \text{Sub}(F_i, g_i)$. By Proposition 1, $\text{Sub}((\text{Sub}(\mathcal{C}\sigma))\downarrow_R) = (\text{Sub}(\mathcal{C}\sigma))\downarrow_R$ Thus the function ϕ^{-1} is defined for each $r_{i,j}$. Let $q_{i,j} = \phi^{-1}(r_{i,j})$ and Q_i be the sequence of the $q_{i,j}$.

The algorithm will non-deterministically produce a $\mathcal{C}'$ corresponding to these choices and satisfied by σ (extended over Q by $\psi(t)\sigma = (t\sigma)\downarrow_R$) by construction. Since $\mathcal{S}$ is satisfiable, following the lines of F. Baader and K. Schulz [3] permits to prove that $\mathcal{S}_1$ and $\mathcal{S}_2$ are satisfiable with a linear constant restriction $\prec$ chosen such that $q \prec q'$ implies $q'\sigma$ is not a subterm of $q\sigma$.

We choose the sequence of constraints in $\mathcal{D}_1$ (resp. $\mathcal{D}_2$) to be the subsequence of constraints $F \rhd q$ from $\mathcal{D}'$ such that the corresponding transition in the solution was performed by a rule in $\mathrm{L}^{u,\mathrm{g}}$ with $\text{Sign}(u) = \mathcal{F}_1$ (resp. $\mathcal{F}_2$). By construction these two systems are satisfiable.

From the soundness and completeness of Algorithm 1 and the (non trivial) fact that the constraint $\mathcal{C}_i = (\mathcal{D}_i, \mathcal{S}_i)$ can be chosen to be deterministic, we can derive our main result on the combination of two intruders. It can be easily generalized to n intruders over disjoint signatures $\mathcal{F}_1, \ldots, \mathcal{F}_n$.

6 Application to Security Protocols

In order to combine constraint solving algorithms for *subtheories* we only need to show that ordered satisfiability is decidable in each component theory. To illustrate the benefit of our approach we show that this is the case for several theories encoding useful properties of cryptographic primitives (pair, xor, exponential, encryption). A consequence of our main result Theorem 1 is that we can decide the security of (finite sessions of) any protocol employing these primitives even assuming their algebraic properties and **even if they are employed all together**.

6.1 Abelian group operators

We consider in this subsection the case of an intruder $\mathcal{I}_\times = \langle \mathcal{F}_\times, S_\times, \mathcal{E}_\times \rangle$, where $\mathcal{F}_\times$ is the signature $\{\mathrm{i}(\cdot), \cdot \times \cdot, 1\}$, $T_\times$ is the set of terms $\{\mathrm{i}(x), x \times y, 1\}$ and with the equational theory:

$$\mathcal{E}_\times \begin{cases} (x \times y) \times z = x \times (y \times z) \\ x \times y = y \times x \\ 1 \times x = x \\ x \times \mathrm{i}(x) = 1 \end{cases}$$

We reduce decidability of $\mathcal{I}_\times$-constraints to satisfiability of affine systems of equations on $\mathbf{Z}$. This reduction is performed in two steps. First we prove that it suffices to consider ground sets E_i in the constraints, and thus that the v_i are linear combination of ground terms. Second the unification system is translated to a system of affine equations over $\mathbf{Z}$.

Lemma 2. *One can compute $\mathcal{C}' = ((E'_i \rhd v_i)_{i \in \{1,\ldots,n\}}, \mathcal{S})$ from $\mathcal{C}$ such that*

– *$\sigma \models \mathcal{C}$ iff $\sigma \models \mathcal{C}'$*
– *for all $i \in \{1, \dots, n\}$ the set E'_i is ground*

This leads to next proposition.

Proposition 3. *The ordered satisfiability problem for deterministic constraints and intruder $\mathcal{I}_\times$ is decidable in NPTIME.*

6.2 XOR operator

We consider the intruder $\mathcal{I}_\oplus = \langle \mathcal{F}_\oplus, \{x \oplus y, 0\}, \mathcal{E}_\oplus \rangle$ with the signature $\mathcal{F}_\oplus = \{0, \cdot \oplus \cdot\}$ and with equational theory:

$$\mathcal{E}_\oplus \begin{cases} (x \oplus y) \oplus z = x \oplus (y \oplus z) \\ \qquad x \oplus y = y \oplus x \\ \qquad 0 \oplus x = x \\ \qquad x \oplus x = 0 \end{cases}$$

Let $\mathcal{C} = ((E_i \rhd v_i)_{i \in \{1,\dots,n\}}, \mathcal{S})$ be a deterministic constraint problem for $\mathcal{I}_\oplus$. Lemma 2 can be adapted to this case. The main difference is that affine systems are over $(\mathbf{Z}/2\mathbf{Z})$. We refer to [14] for a more detailed description of the translation from unification problems to linear systems and of the resolution of such systems.

Proposition 4. *The ordered satisfiability problem for deterministic constraints and intruder $\mathcal{I}_\oplus$ is decidable in PTIME.*

6.3 Exponential operator

For simplicity of exposition we assume that all exponentiations are computed in the same modulus.

Let $\mathcal{F}_{\exp} = \{\exp(\cdot, \cdot), \mathrm{i}(\cdot), \cdot \times \cdot\}$ and $T_{\exp} = \{x \times y, \exp(x, y), \exp(x, \mathrm{i}(y))\}$. We now consider the intruder system $\mathcal{I}_{\exp} = \langle \mathcal{F}_{\exp}, T_{\exp}, \mathcal{E}_{\exp} \rangle$ where

$$\mathcal{E}_{\exp} \begin{cases} \exp(x, 1) = x \\ \exp(\exp(x, y), z) = \exp(x, y \times z) \\ (x \times y) \times z = x \times (y \times z) \\ x \times y = y \times x \\ 1 \times x = x \\ x \times \mathrm{i}(x) = 1 \end{cases}$$

Following [18] the satisfiability problem for deterministic constraint systems for this intruder can be reduced to the satisfiability problem for an abelian group operator. We can deduce:

Proposition 5. *The ordered satisfiability problem for deterministic constraints and intruder $\mathcal{I}_{\exp}$ is decidable in NPTIME.*

6.4 Equational Dolev-Yao theory with explicit decryption

We consider now the Dolev-Yao intruder $\mathcal{I}_{DY} = \langle \mathcal{F}_{DY}, T_{DY}, \mathcal{E}_{DY} \rangle$ over the signature $\mathcal{F}_{DY} = \{\langle \cdot, \cdot \rangle, \pi_1(\cdot), \pi_2(\cdot), \mathrm{se}(\cdot, \cdot), \mathrm{sd}(\cdot, \cdot)\}$ with deduction system defined by $S_{DY} = \{\langle x, y \rangle, \pi_1(x), \pi_2(x), \mathrm{se}(x, y), \mathrm{sd}(x, y)\}$ and the equational theory:

$$\mathcal{E}_{DY} \begin{cases} \pi_1(\langle x, y \rangle) = x \\ \pi_2(\langle x, y \rangle) = y \\ \mathrm{sd}(\mathrm{se}(x, y), y) = x \end{cases}$$

First we note that we can get from $\mathcal{E}_{DY}$ a convergent and finite rewrite system R_{DY} simply by orienting the axioms from left to right. Thanks to Theorem 8.5. of Schmidt-Schauss [20] satisfiability of equational systems modulo $\mathcal{E}_{DY}$ is decidable even in presence of linear constant restrictions. The idea is that the so-called *narrowing* procedure modulo R_{DY} terminates (since rules right-hand sides are variables) and is complete for solving equations modulo $\mathcal{E}_{DY}$ with linear constant restrictions.

The algorithm of [2] for deciding intruder $\mathcal{I}_{DY}$-constraints can be adapted to generate a finite and complete set of symbolic solutions. Then we can use the constant elimination technique of [20] to solve the ordered satisfiability problem: we apply *narrowing* (i.e. instanciating and rewriting) to the complete set of symbolic solutions provided by [2] and then we eliminate from the resulting substitutions the ones that do not satisfy the constant restrictions.

7 Conclusion

We have proposed an algorithm for combining decision procedures for intruder constraints on disjoint signatures. This algorithm allows for a modular treatment of algebraic operators in protocol analysis and a better understanding of complexity issues in the domain. Since only constraint satisfiability is required from the intruder subtheories the approach should permit one to handle more complex operators.

References

1. M. Abadi, B. Blanchet, and C. Fournet. Just Fast Keying in the Pi Calculus. In David Schmidt, editor, *Proceedings of ESOP'04*, volume 2986 of *Lecture Notes on Computer Science*, pages 340–354, Barcelona, Spain, 2004. Springer Verlag.
2. R. Amadio, D. Lugiez, and V. Vanackère. On the symbolic reduction of processes with cryptographic functions. *Theor. Comput. Sci.*, 290(1):695–740, 2003.
3. F. Baader and K. U. Schulz. Unification in the union of disjoint equational theories. combining decision procedures. *J. Symb. Comput.*, 21(2):211–243, 1996.
4. D. Basin, S. Mödersheim, and L. Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. In Einar Snekkenes and Dieter Gollmann, editors, *Proceedings of ESORICS'03*, LNCS 2808, pages 253–270. Springer-Verlag, 2003.
5. M. Boreale. Symbolic trace analysis of cryptographic protocols. In *Proceedings of the 28th ICALP'01*, LNCS 2076, pages 667–681. Springer-Verlag, Berlin, 2001.
6. M. Boreale and M. Buscemi. Symbolic analysis of crypto-protocols based on modular exponentiation. In *Proceedings of MFCS 2003*, volume 2747 of *Lecture Notes in Computer Science*. Springer, 2003.

7. N. Borisov, I. Goldberg, and D. Wagner. Intercepting mobile communications: the insecurity of 802.11. In *Proceedings of MOBICOM 2001*, pages 180–189, 2001.
8. Y. Chevalier, R. Kuesters, M. Rusinowitch, and M. Turuani. An NP Decision Procedure for Protocol Insecurity with XOR. In *Proceedings of the Logic In Computer Science Conference, LICS'03*, June 2003.
9. Y. Chevalier and M. Rusinowitch. Combining intruder theories. Technical report, INRIA, 2005. `http://www.inria.fr/rrrt/liste-2005.html`.
10. Y. Chevalier and L. Vigneron. A Tool for Lazy Verification of Security Protocols. In *Proceedings of the Automated Software Engineering Conference (ASE'01)*. IEEE Computer Society Press, 2001.
11. H. Comon-Lundh and V. Shmatikov. Intruder Deductions, Constraint Solving and Insecurity Decision in Presence of Exclusive or. In *Proceedings of the Logic In Computer Science Conference, LICS'03*, pages 271–280, 2003.
12. S. Delaune and F. Jacquemard. A decision procedure for the verification of security protocols with explicit destructors. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS'04)*, pages 278–287, Washington, D.C., USA, October 2004. ACM Press.
13. N. Dershowitz and J-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B*, pages 243–320. Elsevier, 1990.
14. G. Guo, P. Narendran, and D. A. Wolfram. Unification and matching modulo nilpotence. *Information and Computation*, 162((1-2)):3–23, 2000.
15. J. Hsiang and M. Rusinowitch. On word problems in equational theories. In *ICALP*, volume 267 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 1987.
16. C. Meadows and P. Narendran. A unification algorithm for the group Diffie-Hellman protocol. In *Workshop on Issues in the Theory of Security (in conjunction with POPL'02), Portland, Oregon, USA, January 14-15*, 2002.
17. J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *ACM Conference on Computer and Communications Security*, pages 166–175, 2001.
18. J. Millen and V. Shmatikov. Symbolic protocol analysis with an abelian group operator or Diffie-Hellman exponentiation. *Journal of Computer Security*, 2005.
19. M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *Proc.14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, June 2001.
20. M. Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. *J. Symb. Comput.*, 8(1/2):51–99, 1989.

Can Context Sequence Matching Be Used for XML Querying?*

Temur Kutsia[1] and Mircea Marin[2]

[1] Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria
tkutsia@risc.uni-linz.ac.at
[2] Graduate School of Systems and Information Engineering
University of Tsukuba
Tsukuba 305-8573, Japan
mmarin@cs.tsukuba.ac.jp

Abstract. We describe a matching algorithm for terms built over flexible arity function symbols and context, function, sequence, and individual variables. The algorithm is called a context sequence matching algorithm. Context variables allow matching to descend in term-trees to arbitrary depth. Sequence variables allow matching to move in term-trees in arbitrary breadth. The ability to explore terms in two orthogonal directions in a uniform way may be useful for querying data available as a large term, like XML documents. We extend the algorithm to process regular constraints and discuss its possible application in XML querying.

1 Introduction

We describe a context sequence matching algorithm and discuss its possible application in querying XML [27]. Context variables may be instantiated with a context—a term with a hole. They permit matching to descend to arbitrary depth in a term represented as a tree. Sequence variables may be instantiated with a finite (maybe empty) sequence of terms. They are normally used with flexible arity function symbols and permit matching to move to arbitrary breadth. Thus, context and sequence variables together allow exploring terms in two orthogonal directions in a uniform way which may be useful for querying data available as a large term, like XML documents.

Besides context and sequence variables we have function and individual variables. Function variables may be instantiated with a single function symbol or with another function variable. Individual variables may be bound with a single term. Like context and sequence variables, functional and individual variables can be used to traverse terms in depth and breadth, respectively, but only in one level.

In this paper first we describe a minimal and complete rule-based algorithm for context sequence matching. It operates on terms built using flexible arity function symbols and involving context, sequence, function, and individual variables. Then we show how to use context sequence matching in a declarative XML query language. At the end, we extend the algorithm to deal with regular

* Temur Kutsia has been supported by the Austrian Science Foundation (FWF) under Project SFB F1302 and F1322.

constraints, both in depth and in breadth. Context sequence matching, with or without regular restrictions, is finitary. Regular expressions provide a powerful mechanism for restricting data values in XML. Many languages have support for them.

Context matching and unification have been intensively investigated in the recent past years, see e.g, [9, 10, 20, 24–26]. Context matching is decidable. Decidability of context unification is still an open question. Schmidt-Schauß and Stuber in [26] gave a context matching algorithm and noted that it can be used similar to XPath [7] matching for XML documents. Sequence matching and unification was addressed, for instance, in [2, 12, 13, 16–18, 22]. Both matching and unification with sequence variables are decidable. Sequence unification procedure described in [17, 18] was implemented in the constraint logic programming language CLP(Flex) [8] and was used for XML processing.

Simulation unification [4] implemented in the Xcerpt language has a 'descendant' construct that is similar to context variables in the sense that it allows to descend in terms to arbitrary depth, but it does not allow regular expressions along it. Also, sequence variables are not present there. However, it can process unordered and incomplete queries, and it is a full scale unification, not a matching. Having sequence variables in a full scale unification would make it infinitary (see e.g., [18]).

In our opinion, context sequence matching can serve as a computational mechanism for a declarative, rule-based language to query and transform XML. Such a query language would have advantages of both path-based and pattern-based languages that form two important classes of XML query languages. Path-based languages usually allow to access a single set of nodes of the graph or tree representing an XML data. The access is based on relations with other nodes in the graph or tree specified in the path expression. Pattern-based languages allow access to several parts of the graph or tree at once specifying the relations among the accessed nodes by tree or graph patterns. (For a recent survey over query and transformation languages see [11].) Moreover, with context sequence matching we can achieve improved control on rewriting that can be useful for rewriting-based web site specification and verification techniques [1]. In our opinion, a system like ρLog [23] can be extended to a prototype of such a query language. ρLog is a rule-based language whose computational mechanism uses sequence matching with function variables. It supports single and multiple query answers, non-deterministic computations, and has a clean declarative semantics. However, implementation issues for such a query language is not a subject of this paper.

Another possible application area for context sequence matching is mathematical knowledge management. For instance, it can retrieve algorithms or problems from the schema library [5] of the Theorema system [6].

The results of this paper extend those of [19] by considering regular expressions on full contexts instead of functions only.

The paper is organized as follows: In Section 2 we introduce preliminary notions. In Section 3 we describe the context sequence matching algorithm. Its application in XML querying is discussed in Section 4. Section 5 is about regular expression matching for context and sequence variables. Section 6 concludes.

2 Preliminaries

We assume fixed pairwise disjoint sets of symbols: individual variables $\mathcal{V}_{\mathrm{Ind}}$, sequence variables $\mathcal{V}_{\mathrm{Seq}}$, function variables $\mathcal{V}_{\mathrm{Fun}}$, context variables $\mathcal{V}_{\mathrm{Con}}$, and function symbols $\mathcal{F}$. The sets $\mathcal{V}_{\mathrm{Ind}}$, $\mathcal{V}_{\mathrm{Seq}}$, $\mathcal{V}_{\mathrm{Fun}}$, and $\mathcal{V}_{\mathrm{Con}}$ are countable. The set $\mathcal{F}$ is finite or countable. All the symbols in $\mathcal{F}$ except a distinguished constant $\circ$ (called a *hole*) have flexible arity. We will use x, y, z for individual variables, $\overline{x}, \overline{y}, \overline{z}$ for sequence variables, F, G, H for function variables, $\overline{C}, \overline{D}, \overline{E}$ for context variables, and a, b, c, f, g, h for function symbols. We may use these meta-variables with indices as well.

Terms are constructed using the following grammar:

$$t ::= x \mid \overline{x} \mid \circ \mid f(t_1, \ldots, t_n) \mid F(t_1, \ldots, t_n) \mid \overline{C}(t)$$

In $\overline{C}(t)$ the term t can not be a sequence variable. We will write a for the term $a()$ where $a \in \mathcal{F}$. The meta-variables s, t, r, maybe with indices, will be used for terms. A *ground* term is a term without variables. A *context* is a term with a single occurrence of the hole constant $\circ$. To emphasize that a term t is a context we will write $t[\circ]$. A context $t[\circ]$ may be applied to a term s that is not a sequence variable, written $t[s]$, and the result is the term consisting of t with $\circ$ replaced by s. We will use C and D, with or without indices, for contexts.

A *substitution* is a mapping from individual variables to those terms which are not sequence variables and contain no holes, from sequence variables to finite, possibly empty sequences of terms without holes, from function variables to function variables and symbols, and from context variables to contexts, such that all but finitely many individual and function variables are mapped to themselves, all but finitely many sequence variables are mapped to themselves considered as singleton sequences, and all but finitely many context variables are mapped to themselves applied to the hole. For example, the mapping $\{x \mapsto f(a, \overline{y}), \overline{x} \mapsto \ulcorner\urcorner, \overline{y} \mapsto \ulcorner a, \overline{C}(f(b)), x\urcorner, F \mapsto g, \overline{C} \mapsto g(\circ)\}$ is a substitution. We will use lower case Greek letters $\sigma, \vartheta, \varphi$, and ε for substitutions, where ε will denote the empty substitution. As usual, indices may be used with the meta-variables.

Substitutions are extended to terms as follows:

$$\begin{aligned} x\sigma &= \sigma(x) \\ \overline{x}\sigma &= \sigma(\overline{x}) \\ f(t_1, \ldots, t_n)\sigma &= f(t_1\sigma, \ldots, t_n\sigma) \\ F(t_1, \ldots, t_n)\sigma &= \sigma(F)(t_1\sigma, \ldots, t_n\sigma) \\ \overline{C}(t)\sigma &= \sigma(\overline{C})[t\sigma] \end{aligned}$$

A substitution σ is *more general* than ϑ, denoted $\sigma \leq \vartheta$, if there exists a φ such that $\sigma\varphi = \vartheta$. A substitution σ is *more general* than ϑ *on a set of variables* $\mathcal{V}$, denoted $\sigma \leq^{\mathcal{V}} \vartheta$, if there exists a φ such that $v\sigma\varphi = v\vartheta$ for all $v \in \mathcal{V}$. A *context sequence matching problem* is a finite multiset of term pairs (*matching equations*), written $\{s_1 \ll t_1, \ldots, s_n \ll t_n\}$, where the s's and the t's contain no holes, the s's are not sequence variables, and the t's are ground. We will also call the s's the *query* and the t's the *data*. Substitutions are extended to matching equations and matching problems in the usual way. A substitution σ is called a *matcher* of the matching problem $\{s_1 \ll t_1, \ldots, s_n \ll t_n\}$ if $s_i\sigma = t_i$ for all $1 \leq i \leq n$. We will use $\varGamma$ and $\varDelta$ to denote matching problems. A *complete set of matchers* of a matching problem $\varGamma$ is a set of substitutions S such that (i) each element of S is a matcher of $\varGamma$, and (ii) for each matcher ϑ of $\varGamma$ there exist a substitution $\sigma \in S$ such that $\sigma \leq \vartheta$. The set S is a *minimal complete set of matchers* of $\varGamma$ if it is a complete set and two distinct elements of S are incomparable with respect to $\leq$. For solvable problems this set is finite, i.e. context sequence matching is finitary.

Example 1. The minimal complete set of matchers for the context sequence matching problem $\{\overline{C}(f(\overline{x})) \ll g(f(a,b), h(f(a), f))\}$ consists of three elements: $\{\overline{C} \mapsto g(\circ, h(f(a), f)),\ \overline{x} \mapsto \ulcorner a, b \urcorner\}$, $\{\overline{C} \mapsto g(f(a,b), h(\circ, f)),\ \overline{x} \mapsto a\}$, and $\{\overline{C} \mapsto g(f(a,b), h(f(a), \circ)),\ \overline{x} \mapsto \ulcorner\urcorner\}$.

3 Matching Algorithm

We now present inference rules for deriving solutions for matching problems. A *system* is either the symbol $\perp$ (representing failure) or a pair $\langle \varGamma; \sigma \rangle$, where $\varGamma$ is a matching problem and σ is a substitution. The inference system $\mathfrak{I}$ consists of the transformation rules on systems listed below. We assume that the indices n and m are non-negative unless otherwise stated.

T: **Trivial**

$\{t \ll t\} \cup \varGamma';\ \sigma \Longrightarrow \varGamma';\ \sigma.$

IVE: **Individual Variable Elimination**

$\{x \ll t\} \cup \varGamma';\ \sigma \Longrightarrow \varGamma'\vartheta;\ \sigma \cup \vartheta,$ where $\vartheta = \{x \mapsto t\}$.

FVE: **Function Variable Elimination**

$$\{F(s_1, \ldots, s_n) \ll f(t_1, \ldots, t_m)\} \cup \varGamma';\ \sigma$$
$$\Longrightarrow \{f(s_1\vartheta, \ldots, s_n\vartheta) \ll f(t_1, \ldots, t_m)\} \cup \varGamma'\vartheta;\ \sigma \cup \vartheta,$$

where $\vartheta = \{F \mapsto f\}$.

TD: **Total Decomposition**

$$\{f(s_1, \ldots, s_n) \ll f(t_1, \ldots, t_n)\} \cup \varGamma';\ \sigma$$
$$\Longrightarrow \{s_1 \ll t_1, \ldots, s_n \ll t_n\} \cup \varGamma';\ \sigma,$$

if $f(s_1, \ldots, s_n) \neq f(t_1, \ldots, t_n)$ and $s_i \notin \mathcal{V}_{\mathrm{Seq}}$ for all $1 \leq i \leq n$.

PD: **Partial Decomposition**

$$\{f(s_1, \ldots, s_n) \ll f(t_1, \ldots, t_m)\} \cup \varGamma';\ \sigma$$
$$\Longrightarrow \{s_1 \ll t_1, \ldots, s_{k-1} \ll t_{k-1}, f(s_k, \ldots, s_n) \ll f(t_k, \ldots, t_m)\} \cup \varGamma';\ \sigma,$$

if $f(s_1, \ldots, s_n) \neq f(t_1, \ldots, t_m)$, $s_k \in \mathcal{V}_{\mathrm{Seq}}$ for some $1 < k \leq \min(n, m) + 1$, and $s_i \notin \mathcal{V}_{\mathrm{Seq}}$ for all $1 \leq i < k$.

SVD: **Sequence Variable Deletion**

$\{f(\overline{x}, s_1, \ldots, s_n) \ll t\} \cup \Gamma';\ \sigma \Longrightarrow \{f(s_1\vartheta, \ldots, s_n\vartheta) \ll t\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$
where $\vartheta = \{\overline{x} \mapsto \ulcorner\urcorner\}$.

W: **Widening**

$\{f(\overline{x}, s_1, \ldots, s_n) \ll f(t, t_1, \ldots, t_m)\} \cup \Gamma';\ \sigma$
$\Longrightarrow \{f(\overline{x}, s_1\vartheta, \ldots, s_n\vartheta) \ll f(t_1, \ldots, t_m)\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$
where $\vartheta = \{\overline{x} \mapsto \ulcorner t, \overline{x}\urcorner\}$.

CVD: **Context Variable Deletion**

$\{\overline{C}(s) \ll t\} \cup \Gamma';\ \sigma \Longrightarrow \{s\vartheta \ll t\} \cup \Gamma'\vartheta;\ \sigma\vartheta, \qquad \text{where } \vartheta = \{\overline{C} \mapsto \circ\}.$

D: **Deepening**

$\{\overline{C}(s) \ll f(t_1, \ldots, t_m)\} \cup \Gamma';\ \sigma \Longrightarrow \{\overline{C}(s\vartheta) \ll t_j\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$
where $\vartheta = \{\overline{C} \mapsto f(t_1, \ldots, t_{j-1}, \overline{C}(\circ), t_{j+1}, \ldots, t_m)\}$ for some $1 \le j \le m$, and $m > 0$.

SC: **Symbol Clash**

$\{f(s_1, \ldots, s_n) \ll g(t_1, \ldots, t_m)\} \cup \Gamma';\ \sigma \Longrightarrow \bot,$
if $f \notin \mathcal{V}_{\mathrm{Con}} \cup \mathcal{V}_{\mathrm{Fun}}$ and $f \neq g$.

AD: **Arity Disagreement**

$\{f(s_1, \ldots, s_n) \ll f(t_1, \ldots, t_m)\} \cup \Gamma';\ \sigma \Longrightarrow \bot,$
if $m \neq n$ and $s_i \notin \mathcal{V}_{\mathrm{Seq}}$ for all $1 \le i \le n$.

$\mathsf{E1}$: **Empty 1**

$\{f() \ll f(t, t_1, \ldots, t_n)\} \cup \Gamma';\ \sigma \Longrightarrow \bot.$

$\mathsf{E2}$: **Empty 2**

$\{f(s, s_1, \ldots, s_n) \ll f()\} \cup \Gamma';\ \sigma \Longrightarrow \bot, \qquad \text{if } s \notin \mathcal{V}_{\mathrm{Seq}}.$

We may use the rule name abbreviations as subscripts, e.g. $\Gamma_1; \sigma_1 \Longrightarrow_{\mathsf{T}} \Gamma_2; \sigma_2$ for the Trivial rule. SVD, W, CVD, and D are non-deterministic rules. A *derivation* is a sequence $\Gamma_1; \sigma_1 \Longrightarrow \Gamma_2; \sigma_2 \Longrightarrow \cdots$ of system transformations.

Definition 1. *A* context sequence matching algorithm $\mathfrak{M}$ *is any program that takes a system* $\Gamma; \varepsilon$ *as an input and uses the rules in* $\mathfrak{I}$ *to generate a complete tree of derivations, called the* matching tree for Γ*, in the following way:*

1. *The root of the tree is labeled with* $\Gamma; \varepsilon$*.*
2. *Each branch of the tree is a derivation. The nodes in the tree are systems.*
3. *If several transformation rules, or different instances of the same transformation rule are applicable to a node in the tree, they are applied concurrently. No rules are applicable to the leaves.*

The leaves of a matching tree are labeled either with the systems of the form $\emptyset; \sigma$ or with $\bot$. The branches that end with $\emptyset; \sigma$ are *successful branches*, and those that end with $\bot$ are *failed branches*. We denote by $\mathcal{S}ol_{\mathfrak{M}}(\Gamma)$ the solution set of Γ generated by $\mathfrak{M}$, i.e., the set of all σ's such that $\emptyset; \sigma$ is a leaf of the matching tree for Γ.

Theorem 1 (Main Theorem). *The matching algorithm* $\mathfrak{M}$ *terminates for any input problem* Γ *and generates a minimal complete set of matchers of* Γ.

Proof. See Appendix A. □

Moreover, note that $\mathfrak{M}$ never computes the same matcher twice.

If we are not interested in bindings for certain variables, we can replace them with the anonymous variables: "$\_$ " for any individual or function variable, and "$\_\_\_$ " for any sequence or context variable. It is straightforward to adapt the rules in $\mathfrak{I}$ to anonymous variables: If an anonymous variable occurs in the rule IVE, FVE, SVD, W, CVD, or D then the substitution ϑ in the same rule is the empty substitution ε. It is interesting to note that a context sequence matching equation $s \ll t$ whose all variables are anonymous variables can be considered as a problem of computing simulations of s in t that can be efficiently solved by the algorithm described in [14].

4 Querying XML

We assume the existence of a declarative, rule-based query and transformation language for XML that uses the context sequence matching to answer queries. We refer to this language as $\mathcal{L}$. Queries in $\mathcal{L}$ are expressed as (conditional) rules *pattern* $\rightarrow$ *result* `if` *condition*. We do not go into the details, just mention that conditions can be effectively checked. In particular, arithmetic formulae, matchability tests, and queries can be used in conditions, but special care has to be taken about variable occurrences. Note that conditions can also be omitted (assumed to be true). The *pattern* matches the data in the root position. One can choose between getting all the results or only one of them.

To put more syntactic sugar on queries, we borrow some notation from [4]. We write $f\{s_1, \ldots, s_n\}$ if the order of arguments $s_1, \ldots, s_n$ does not matter. The following (rather inefficient) rule relates a matching problem in which the curly bracket construct occurs, to the standard matching problems:

Ord: **Orderless**

$$\{f\{s_1, \ldots, s_n\} \ll t\} \cup \Gamma';\ \sigma \Longrightarrow \{f(s_{\pi(1)}, \ldots, s_{\pi(n)}) \ll t\} \cup \Gamma';\ \sigma,$$

if $f(s_1, \ldots, s_n) \neq t$ and π is a permutation of $1, \ldots, n$.

Moreover, we can use the double curly bracket notation $f\{\!\{s_1, \ldots, s_n\}\!\}$ for the term $f\{\_\_\_, s_1, \_\_\_, \ldots, \_\_\_, s_n, \_\_\_\}$, and the double bracket notation $f(\!(s_1, \ldots, s_n)\!)$ for $f(\_\_\_, s_1, \_\_\_, \ldots, \_\_\_, s_n, \_\_\_)$. The matching algorithm can be easily modified to work directly (and more efficiently) on such representations.

Now we show how in this language the query operations given in [21] can be expressed. (This benchmark was used to compare five XML query languages in [3].) The case study is that of a car dealer office, with documents from different auto dealers and brokers. The `manufacturer` documents list the manufacturers name, year, and models with their names, front rating, side rating, and rank; the `vehicle` documents list the vendor, make, year, color and price. We consider XML data of the form:

```
<manufacturer>
       <mn-name>Mercury</mn-name>
       <year>1999</year>
       <model>
          <mo-name>Sable LT</mo-name>
          <front-rating>3.84</front-rating>
          <side-rating>2.14</side-rating>
          <rank>9</rank>
       </model>
       <model>...</model>
       ...
</manufacturer>
```

while the dealers and brokers publish information in the form

```
<vehicle>
       <vendor>Scott Thomason</vendor>
       <make>Mercury</make>
       <model>Sable LT</model>
       <year>1999</year>
       <color>metallic blue</color>
       <option opt="sunroof"/>
       <option opt="A/C"/>
       <option opt="lthr seats"/>
       <price>26800</price>
</vehicle>.
```

Translating the data into our syntax is pretty straightforward. For instance, the manufacturer element can be written as:

$$\begin{aligned}&\mathit{manufacturer}(\mathit{mn\text{-}name}(\mathit{Mercury}), \mathit{year}(\mathit{1999}),\\&\quad \mathit{model}(\mathit{mo\text{-}name}(\mathit{SableLT}), \mathit{front\text{-}rating}(\mathit{3.84}), \mathit{side\text{-}rating}(\mathit{2.14}), \mathit{rank}(\mathit{9}))).\end{aligned}$$

The query operations and their encoding in our syntax are given below.

Selection and Extraction: We want to select and extract <**manufacturer**> elements where some <**model**> has <**rank**> less or equal to 10:

$$\begin{aligned}&\_(\!(\mathit{manufacturer}(\overline{x}_1, \mathit{model}(\overline{y}_1, \mathit{rank}(x), \overline{y}_2), \overline{x}_2))\!)\\&\qquad \to \mathit{manufacturer}(\overline{x}_1, \mathit{model}(\overline{y}_1, \mathit{rank}(x), \overline{y}_2), \overline{x}_2) \;\texttt{if}\; x \leq \mathit{10}.\end{aligned}$$

Reduction: From the <**manufacturer**> elements, we want to drop those <**model**> sub-elements whose <**rank**> is greater than 10. We also want to elide the <**front rating**> and <**side rating**> elements from the remaining models.

$$\begin{aligned}&\_(\!(\mathit{manufacturer}(\overline{x}_1,\\&\qquad\qquad \mathit{model}(\overline{y}_1, \mathit{front\text{-}rating}(\_\,), \mathit{side\text{-}rating}(\_\,), \mathit{rank}(x), \overline{y}_2), \overline{x}_2))\!)\\&\quad \to \mathit{manufacturer}(\overline{x}_1, \mathit{model}(\overline{y}_1, \mathit{rank}(x), \overline{y}_2), \overline{x}_2) \;\texttt{if}\; x \leq \mathit{10}.\end{aligned}$$

Joins: We want our query to generate pairs of <manufacturer> and <vehicle> elements where <mn-name>=<make>, <mo-name>=<model>, and <year>= <year>.

$$\begin{aligned}
&_\{\!\!\{ \mathit{manufacturer}(\overline{x}_1, \mathit{mn\text{-}name}(x_1), \overline{x}_2, \mathit{year}(x_2), \overline{x}_3,\\
&\quad \overline{C}(\mathit{mo\text{-}name}(y_1)), \overline{x}_4),\\
&\quad \mathit{vehicle}(\overline{z}_1, \mathit{make}(x_1), \overline{z}_2, \mathit{model}(y_1), \overline{z}_3, \mathit{year}(x_2), \overline{z}_4)\}\!\!\}\\
&\to \mathit{pair}(\mathit{manufacturer}(\overline{x}_1, \mathit{mn\text{-}name}(x_1), \overline{x}_2, \mathit{year}(x_2), \overline{x}_3,\\
&\quad \overline{C}(\mathit{mo\text{-}name}(y_1)), \overline{x}_4),\\
&\quad \mathit{vehicle}(\overline{z}_1, \mathit{make}(x_1), \overline{z}_2, \mathit{model}(x_2), \overline{z}_3, \mathit{year}(y_1), \overline{z}_4)).
\end{aligned}$$

Restructuring: We want our query to collect <car> elements listing their make, model, vendor, rank, and price, in this order:

$$\begin{aligned}
&_\{\!\!\{ \mathit{vehicle}((\mathit{vendor}(y_1), \mathit{make}(y_2), \mathit{model}(y_3), \mathit{year}(y_4), \mathit{price}(y_5))),\\
&\quad \mathit{manufacturer}((\overline{C}(\mathit{rank}(x_1))))\}\!\!\}\\
&\to \mathit{car}(\mathit{make}(y_2), \mathit{model}(y_3), \mathit{vendor}(y_1), \mathit{rank}(x_1), \mathit{price}(y_5)).
\end{aligned}$$

Hence, all these operations can be easily expressed in our framework.

At the end of this section we give an example how to extract elements from an XML document that do not meet certain requirements (e.g., miss certain information). Such problems arise in web site verification tasks discussed in [1].

We want our query to select from the <manufacturer> elements those <model> sub-elements which miss the <rank> information:

$$\_(\!(\mathit{manufacturer}(\!(\mathit{model}(\overline{y}))\!))\!) \to \mathit{model}(\overline{y}) \;\mathtt{if}\; \mathit{model}(\!(\mathit{rank}(\!()\!))\!) \not\ll \mathit{model}(\overline{y}).$$

The condition in the query requires the term $\mathit{model}(\!(\mathit{rank}(\!()\!))\!)$ not to match $\mathit{model}(\overline{y})$. The variable $\overline{y}$ gets instantiated while matching the query pattern $\_(\!(\mathit{manufacturer}(\!(\mathit{model}(\overline{y}))\!))\!)$ against the data. Since context sequence matching is decidable, the condition can be effectively checked.

5 Regular Expressions

Regular expressions provide a powerful mechanism for restricting data values in XML. Many languages have support for them. In [15] regular expression pattern matching is proposed as a core feature of programming languages for manipulating XML. The classical approach uses finite automata for regular expression matching. In this section we show that regular expressions matching can be easily incorporated into the rule-based framework of context sequence matching. We assume that the set $\mathcal{F}$ is finite.

Regular expressions on terms are defined by the following grammar:

$$\mathtt{R} ::= t \mid \ulcorner\urcorner \mid \ulcorner \mathtt{R}_1, \mathtt{R}_2 \urcorner \mid \mathtt{R}_1 | \mathtt{R}_2 \mid \mathtt{R}^*,$$

where t is a term without holes, $\ulcorner\urcorner$ is the empty sequence, "," is concatenation, "|" is choice, and $*$ is repetition (Kleene star). The symbols "$\ulcorner$" and "$\urcorner$" are there just for the readability purposes. The operators are right-associative; "*" has the highest precedence, followed by "," and "|".

Substitutions are extended to regular expressions on terms in the usual way: $\ulcorner\urcorner\sigma = \ulcorner\urcorner$, $\ulcorner \mathtt{R}_1, \mathtt{R}_2 \urcorner\sigma = \ulcorner \mathtt{R}_1\sigma, \mathtt{R}_2\sigma\urcorner$, $(\mathtt{R}_1|\mathtt{R}_2)\sigma = \mathtt{R}_1\sigma|\mathtt{R}_2\sigma$, and $\mathtt{R}^*\sigma = (\mathtt{R}\sigma)^*$. Each regular expression on terms $\mathtt{R}$ define the corresponding regular language $L(\mathtt{R})$.

Regular expressions on contexts are defined as follows:

$$\mathtt{Q} ::= C \mid \ulcorner \mathtt{Q}_1, \mathtt{Q}_2 \urcorner \mid \mathtt{Q}_1|\mathtt{Q}_2 \mid \mathtt{Q}^*.$$

Like for regular expressions on terms, substitutions are extended to regular expressions on contexts in the usual way. Each regular expression on contexts $\mathtt{Q}$ defines the corresponding regular tree language $L(\mathtt{Q})$ as follows:

$$\begin{aligned} L(C) &= \{C\}. \\ L(\ulcorner \mathtt{Q}_1, \mathtt{Q}_2 \urcorner) &= \{C_1[C_2] \mid C_1 \in L(\mathtt{Q}_1) \text{ and } C_2 \in L(\mathtt{Q}_2)\}. \\ L(\mathtt{Q}_1|\mathtt{Q}_2) &= L(\mathtt{Q}_1) \cup L(\mathtt{Q}_2). \\ L(\mathtt{Q}^*) &= \{\circ\} \cup L(\ulcorner \mathtt{Q}, \mathtt{Q}^* \urcorner). \end{aligned}$$

Membership atoms are atoms of the form Ts `in` $\mathtt{R}$ or Cv `in` $\mathtt{Q}$, where Ts is a finite, possibly empty, sequence of terms, and Cv is either a context or a context variable. Membership-pairs are pairs $(p, \mathtt{f})$ where p is a membership atom and $\mathtt{f}$ is a flag that is an integer 0 or 1. The intuition behind the membership-pair $(\overline{x}$ `in` $\mathtt{R}, \mathtt{f})$ is that if $\mathtt{f} = 0$ then $\overline{x}$ is allowed to be replaced with $\ulcorner\urcorner$ if $\mathtt{R}$ permits. If $\mathtt{f} = 1$ then the replacement is impossible, even if the corresponding regular expression permits. Similarly, the intuition behind $(\overline{C}$ `in` $\mathtt{Q}, \mathtt{g})$ is that if $\mathtt{g} = 0$ then $\overline{C}$ is allowed to be replaced with $\circ$ if $\mathtt{Q}$ permits. If $\mathtt{g} = 1$ then the replacement is impossible, even if the corresponding regular expression permits. It will be needed later to guarantee that the regular matching algorithm terminates. Substitutions are extended to membership-pairs in the usual way.

Now, we can extend the query language $\mathcal{L}$ allowing (with some care) membership pairs in conditions. Then such conditions can be checked using finite (tree) automata. We can also tailor this check into the matching process itself, and this is what we discuss in details below.

A *context sequence regular matching problem* is a multiset of matching equations and membership-pairs of the form:

$$\begin{aligned} &\{s_1 \ll t_1, \ldots, s_n \ll t_n,\ (\overline{x}_1 \texttt{ in } \mathtt{R}_1, \mathtt{f}_1), \ldots, (\overline{x}_m \texttt{ in } \mathtt{R}_m, \mathtt{f}_m), \\ &(\overline{C}_1 \texttt{ in } \mathtt{Q}_1, \mathtt{g}_1), \ldots, (\overline{C}_k \texttt{ in } \mathtt{Q}_k, \mathtt{g}_k)\}, \end{aligned}$$

where all $\overline{x}$'s and all $\overline{C}$'s are distinct and do not occur in $\mathtt{R}$'s and $\mathtt{Q}$'s. We will assume that all $\overline{x}$'s and $\overline{C}$'s occur in the matching equations. A substitution σ is called a *regular matcher* for such a problem if $s_i\sigma = t_i$, $\overline{x}_j\sigma \in L(\mathtt{R}_j\sigma)_{\mathtt{f}_j}$, and $\overline{C}_l\sigma \in L(\mathtt{Q}_l\sigma)_{\mathtt{g}_l}$ for all $1 \le i \le n$, $1 \le j \le m$, and $1 \le l \le k$, where $L(\mathtt{R})_0 = L(\mathtt{R})$, $L(\mathtt{R})_1 = L(\mathtt{R}) \setminus \{\ulcorner\urcorner\}$, $L(\mathtt{Q})_0 = L(\mathtt{Q})$, and $L(\mathtt{Q})_1 = L(\mathtt{Q}) \setminus \{\circ\}$.

We define the inference system $\mathfrak{I}_R$ to solve context sequence regular matching problems. It operates on systems $\Gamma; \sigma$ where Γ is a regular matching problem and σ is a substitution. The system $\mathfrak{I}_R$ includes all the rules from the system $\mathfrak{I}$, but SVD, W, CVD, and D need an extra condition on applicability: For the variables $\overline{x}$ and $\overline{C}$ in those rules there should be no membership-pair $(\overline{x}\ \mathtt{in}\ \mathtt{R}, \mathtt{f})$ and $(\overline{C}\ \mathtt{in}\ \mathtt{Q}, \mathtt{g})$ in the matching problem. There are additional rules in $\mathfrak{I}_R$ for the variables constrained by membership-pairs listed below. The meta-functions $\mathtt{NonEmpty}$ and $\oplus$ used in these rules are defined as follows: $\mathtt{NonEmpty}() =\ 0$ and $\mathtt{NonEmpty}(r_1, \ldots, r_n) = 1$ if $r_i \notin \mathcal{V}_{\mathrm{Seq}} \cup \mathcal{V}_{\mathrm{Con}}$ for some $1 \leq i \leq n$; $0 \oplus 0 = 1 \oplus 1 = 0$ and $1 \oplus 0 = 0 \oplus 1 = 1$.

ESRET: **Empty Sequence in a Regular Expression for Terms**

$\{f(\overline{x}, s_1, \ldots, s_n) \ll t,\ (\overline{x}\ \mathtt{in}\ \ulcorner\urcorner, \mathtt{f})\} \cup \Gamma';\ \sigma$
$\quad \Longrightarrow \{f(\overline{x}, s_1, \ldots, s_n)\vartheta \ll t\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$

where $\vartheta = \{\overline{x} \mapsto \ulcorner\urcorner\}$ if $\mathtt{f} = 0$.

TRET: **Term in a Regular Expression for Terms**

$\{f(\overline{x}, s_1, \ldots, s_n) \ll t,\ (\overline{x}\ \mathtt{in}\ s, \mathtt{f})\} \cup \Gamma';\ \sigma$
$\quad \Longrightarrow \{f(\overline{x}, s_1, \ldots, s_n)\vartheta \ll t\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$

where $\vartheta = \{\overline{x} \mapsto s\}$ and $s \notin \mathcal{V}_{\mathrm{Seq}}$.

SVRET: **Sequence Variable in a Regular Expression for Terms**

$\{f(\overline{x}, s_1, \ldots, s_n) \ll t,\ (\overline{x}\ \mathtt{in}\ \overline{y}, \mathtt{f})\} \cup \Gamma';\ \sigma$
$\quad \Longrightarrow \{f(\overline{x}, s_1, \ldots, s_n)\vartheta \ll t\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$

where $\vartheta = \{\overline{x} \mapsto \overline{y}\}$ if $\mathtt{f} = 0$. If $\mathtt{f} = 1$ then $\vartheta = \{\overline{x} \mapsto \ulcorner y, \overline{y}\urcorner\}$ where y is a fresh variable.

ChRET: **Choice in a Regular Expression for Terms**

$\{f(\overline{x}, s_1, \ldots, s_n) \ll t,\ (\overline{x}\ \mathtt{in}\ \mathtt{R}_1|\mathtt{R}_2, \mathtt{f})\} \cup \Gamma';\ \sigma$
$\quad \Longrightarrow \{f(\overline{x}, s_1, \ldots, s_n)\vartheta \ll t,\ (\overline{y}_i\ \mathtt{in}\ \mathtt{R}_i, \mathtt{f})\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$

for $i = 1, 2$, where y_i is a fresh variable and $\vartheta = \{\overline{x} \mapsto \overline{y}_i\}$.

CRET: **Concatenation in a Regular Expression for Terms**

$\{f(\overline{x}, s_1, \ldots, s_n) \ll t,\ (\overline{x}\ \mathtt{in}\ \ulcorner\mathtt{R}_1, \mathtt{R}_2\urcorner, \mathtt{f})\} \cup \Gamma';\ \sigma$
$\quad \Longrightarrow \{f(\overline{x}, s_1, \ldots, s_n)\vartheta \ll t,\ (\overline{y}_1\ \mathtt{in}\ \mathtt{R}_1, \mathtt{f}_1), (\overline{y}_2\ \mathtt{in}\ \mathtt{R}_2, \mathtt{f}_2)\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$

where $\overline{y}_1$ and $\overline{y}_2$ are fresh variables, $\vartheta = \{\overline{x} \mapsto \ulcorner\overline{y}_1, \overline{y}_2\urcorner\}$, and $\mathtt{f}_1$ and $\mathtt{f}_2$ are computed as follows: If $\mathtt{f} = 0$ then $\mathtt{f}_1 = \mathtt{f}_2 = 0$ else $\mathtt{f}_1 = 0$ and $\mathtt{f}_2 = \mathtt{NonEmpty}(\overline{y}_1) \oplus 1$.

RRET1: **Repetition in a Regular Expression for Terms 1**

$\{f(\overline{x}, s_1, \ldots, s_n) \ll t,\ (\overline{x}\ \mathtt{in}\ \mathtt{R}^*, \mathtt{f})\} \cup \Gamma';\ \sigma$
$\quad \Longrightarrow \{f(\overline{x}, s_1, \ldots, s_n)\vartheta \ll t\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$

where $\vartheta = \{\overline{x} \mapsto \ulcorner\urcorner\}$ and $\mathtt{f} = 0$.

RRET2: **Repetition in a Regular Expression for Terms 2**

$\{f(\overline{x}, s_1, \ldots, s_n) \ll t,\ (\overline{x}\ \mathtt{in}\ \mathtt{R}^*, \mathtt{f})\} \cup \Gamma';\ \sigma$
$\quad \Longrightarrow \{f(\overline{x}, s_1, \ldots, s_n)\vartheta \ll t,\ (\overline{y}\ \mathtt{in}\ \mathtt{R}, 1),\ (\overline{x}\ \mathtt{in}\ \mathtt{R}^*, 0)\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$

where y is a fresh variable and $\vartheta = \{\overline{x} \mapsto \ulcorner\overline{y}, \overline{x}\urcorner\}$.

HREC: **Hole in a Regular Expression for Contexts**

$\{\overline{C}(s) \ll t,\ (\overline{C}\ \mathsf{in}\ \circ, \mathsf{g})\} \cup \Gamma';\ \sigma \Longrightarrow \{\overline{C}(s)\vartheta \ll t\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$

where $\vartheta = \{\overline{C} \mapsto \circ\}$ and $\mathsf{g} = 0$.

CxREC: **Context in a Regular Expression for Contexts**

$\{\overline{C}(s) \ll t,\ (\overline{C}\ \mathsf{in}\ C, \mathsf{g})\} \cup \Gamma';\ \sigma \Longrightarrow \{\overline{C}(s)\vartheta \ll t\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$

where $C \neq \circ$, $\mathcal{H}ead(C) \notin \mathcal{V}_{\mathrm{Con}}$, and $\vartheta = \{\overline{C} \mapsto C\}$.

CVREC: **Context Variable in a Regular Expression for Contexts**

$\{\overline{C}(s) \ll t,\ (\overline{C}\ \mathsf{in}\ \overline{D}(\circ), \mathsf{g})\} \cup \Gamma';\ \sigma \Longrightarrow \{\overline{C}(s)\vartheta \ll t\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$

where $\vartheta = \{\overline{C} \mapsto \overline{D}(\circ)\}$ if $\mathsf{g} = 0$. If $\mathsf{g} = 1$ then $\vartheta = \{\overline{C} \mapsto F(\overline{x}, \overline{D}(\circ), \overline{y})\}$, where $F, \overline{x}$, and $\overline{y}$ are fresh variables.

ChREC: **Choice in a Regular Expression for Contexts**

$\{\overline{C}(s) \ll t,\ (\overline{C}\ \mathsf{in}\ \mathtt{Q}_1|\mathtt{Q}_2, \mathsf{g})\} \cup \Gamma';\ \sigma$
$\Longrightarrow \{\overline{C}(s)\vartheta \ll t,\ (\overline{D}_i\ \mathsf{in}\ \mathtt{Q}_i, \mathsf{g})\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$

for $i = 1, 2$, where $\overline{D}_i$ is a fresh variable and $\vartheta = \{\overline{C} \mapsto \overline{D}_i(\circ)\}$.

CREC: **Concatenation in a Regular Expression for Contexts**

$\{\overline{C}(s) \ll t,\ (\overline{C}\ \mathsf{in}\ \ulcorner\mathtt{Q}_1, \mathtt{Q}_2\urcorner, \mathsf{g})\} \cup \Gamma';\ \sigma$
$\Longrightarrow \{\overline{C}(s)\vartheta \ll t,\ (\overline{D}_1\ \mathsf{in}\ \mathtt{Q}_1, \mathsf{g}_1), (\overline{D}_2\ \mathsf{in}\ \mathtt{Q}_2, \mathsf{g}_2)\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$

where $\overline{D}_1$ and $\overline{D}_2$ are fresh variables and $\vartheta = \{\overline{C} \mapsto \overline{D}_1(\overline{D}_2(\circ))\}$, and g_1 and g_2 are computed as follows: If $\mathsf{g} = 0$ then $\mathsf{g}_1 = \mathsf{g}_2 = 0$ else $\mathsf{g}_1 = 0$ and $\mathsf{g}_2 = \mathtt{NonEmpty}(\overline{D}_1) \oplus 1$.

RREC1: **Repetition in a Regular Expression for Contexts 1**

$\{\overline{C}(s) \ll t,\ (\overline{C}\ \mathsf{in}\ \mathtt{Q}^*, \mathsf{g})\} \cup \Gamma';\ \sigma \Longrightarrow \{\overline{C}(s)\vartheta \ll t\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$

where $\vartheta = \{\overline{C} \mapsto \circ\}$ and $\mathsf{g} = 0$. If $\mathsf{g} = 1$ the rule fails.

RREC2: **Repetition in a Regular Expression for Contexts 2**

$\{\overline{C}(s) \ll t,\ (\overline{C}\ \mathsf{in}\ \mathtt{Q}^*, \mathsf{g})\} \cup \Gamma';\ \sigma$
$\Longrightarrow \{\overline{C}(s)\vartheta \ll t,\ (\overline{D}\ \mathsf{in}\ \mathtt{Q}, 1),\ (\overline{C}\ \mathsf{in}\ \mathtt{Q}^*, 0)\} \cup \Gamma'\vartheta;\ \sigma\vartheta,$

where $\overline{D}$ is a fresh variable and $\vartheta = \{\overline{C} \mapsto \overline{D}(\overline{C}(\circ))\}$.

A context sequence regular matching algorithm $\mathfrak{M}_R$ is defined in the similar way as the algorithm $\mathfrak{M}$ (Definition 1) with the only difference that the rules of $\mathfrak{I}_R$ are used instead of the rules of $\mathfrak{I}$. ¿From the beginning, all the flags in the input problem are set to 0. Note that the rules in $\mathfrak{I}_R$ work either on a selected matching equation, or on a selected pair of a matching equation and a pattern-pair. No rule selects a patter-pair alone. We denote by $\mathcal{S}ol_{\mathfrak{M}_R}(\Gamma)$ the solution set of Γ generated by $\mathfrak{M}_R$.

Theorem 2. *The algorithm $\mathfrak{M}_R$ is sound, terminating and complete.*

Proof. See Appendix B. ◻

Note that we can extend the system $\mathfrak{I}_R$ with some more rules that facilitate an early detection of failure, e.g., $\{f(\overline{x}, s_1, \ldots, s_n) \ll f(), (\overline{x} \text{ in } \mathtt{R}, 1)\} \cup \Gamma'; \sigma \Longrightarrow \bot$ would be one of such rules.

Turning back to the query language $\mathcal{L}$, now the queries that contain membership pairs in conditions can be resolved by context sequence regular matching, forming the matching problem with the pattern from the query against the data, and the corresponding membership pairs from the condition.

As a syntactic sugar on regular expressions on contexts, we let function symbols, function variables, and context variables be used as the basic building blocks for regular expressions. Such regular expressions are understood as abbreviations for the corresponding regular expressions on contexts. We demonstrate the correspondence on the example: The regular expression $\ulcorner F, f | \ulcorner \overline{C}, g \urcorner^* \urcorner$ abbreviates the regular context expression

$$\ulcorner F(\overline{x}_1, \circ, \overline{y}_1), f(\overline{x}_2, \circ, \overline{y}_2) | \ulcorner \overline{C}(\overline{x}_3, \circ, \overline{y}_3), g(\overline{x}_4, \circ, \overline{y}_4) \urcorner^* \urcorner$$

where $\overline{x}$'s and $\overline{y}$'s are fresh variables. In this way, the query language $\mathcal{L}$ will understand also the regular path expression syntax.

6 Conclusions

We showed how to use context sequence matching to explore terms (represented as trees) in two orthogonal directions: in depth (by context variables) and in breadth (by sequence variables). We developed a minimal complete rule-based algorithm for context sequence matching. Moreover, we showed that regular restrictions can be easily incorporated in the rule-based matching framework, and sketched proofs of soundness, termination and completeness of such an extension.

Context sequence matching can serve as a computational mechanism for a declarative rule-based XML query and transformation language. In our opinion, an advantage of such a language would be its flexibility and expressiveness: It would combine in itself the features of both path-based and pattern-based languages, and would easily support, for instance, a wide range of queries (selection and extraction, reduction, negation, restructuring, combination), parent-child and sibling relations and their closures, access by position, unordered matching, order-preserving result, partial and total queries, multiple results, and other properties. Moreover, rule-based paradigm would provide a clean declarative semantics.

References

1. M. Alpuente, D. Ballis, and M. Falaschi. A rewriting-based framework for web sites verification. *Electronic Notes on Theoretical Computer Science*, 2004. To appear.
2. H. Boley. *A Tight, Practical Integration of Relations and Functions*, volume 1712 of *LNAI*. Springer, 1999.
3. A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *ACM SIGMOD Record*, 29(1):68–79, 2000.

4. F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Proc. of International Conference on Logic Programming (ICLP)*, number 2401 in LNCS, Copenhagen, Denmark, 2002. Springer.
5. B. Buchberger and A. Crăciun. Algorithm synthesis by lazy thinking: Examples and implementation in Theorema. In *Proc. of the Mathematical Knowledge Management Symposium*, volume 93 of *Electronic Notes on Theoretical Computer Science*, pages 24–59, 2003.
6. B. Buchberger, C. Dupré, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger. The Theorema project: A progress report. In M. Kerber and M. Kohlhase, editors, *Proc. of Calculemus'2000 Conference*, pages 98–113, 2000.
7. J. Clark and S. DeRose, editors. *XML Path Language (XPath) Version 1.0*. W3C, 1999. Available from: `http://www.w3.org/TR/xpath/`.
8. J. Coelho and M. Florido. CLP(Flex): Constraint logic programming applied to XML processing. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE. Proc. of Confederated Int. Conferences*, volume 3291 of *LNCS*, pages 1098–1112. Springer, 2004.
9. H. Comon. Completion of rewrite systems with membership constraints. Part I: Deduction rules. *J. Symbolic Computation*, 25(4):397–419, 1998.
10. H. Comon. Completion of rewrite systems with membership constraints. Part II: Constraint solving. *J. Symbolic Computation*, 25(4):421–453, 1998.
11. T. Furche, F. Bry, S. Schaffert, R. Orsini, I. Horroks, M. Kraus, and O. Bolzer. Survey over existing query and transformation languages. Available from: `http://rewerse.net/deliverables/i4-d1.pdf`, 2004.
12. M. L. Ginsberg. The MVL theorem proving system. *SIGART Bull.*, 2(3):57–60, 1991.
13. M. Hamana. Term rewriting with sequences. In: Proc. of the First Int. *Theorema* Workshop. Technical report 97–20, RISC, Johannes Kepler University, Linz, Austria, 1997.
14. M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 453–462. IEEE Computer Society Press, 1995.
15. H. Hosoya and B. Pierce. Regular expression pattern matching for XML. *J. Functional Programming*, 13(6):961–1004, 2003.
16. T. Kutsia. *Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols*. PhD thesis, Johannes Kepler University, Linz, Austria, 2002.
17. T. Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, editors, *Artificial Intelligence, Automated Reasoning and Symbolic Computation. Proc. of Joint AISC'2002 – Calculemus'2002 Conference*, volume 2385 of *LNAI*, pages 290–304. Springer, 2002.
18. T. Kutsia. Solving equations involving sequence variables and sequence functions. In B. Buchberger and J. A. Campbell, editors, *Artificial Intelligence and Symbolic Computation. Proc. of AISC'04 Conference*, volume 3249 of *LNAI*, pages 157–170. Springer, 2004.
19. T. Kutsia. Context sequence matching for XML. In M. Alpuente, S. Escobar, and M. Falaschi, editors, *Proc. of First International Workshop on Automated Specification and Verification of Web Sites (WWV'05)*, pages 103–119, Valencia, Spain, March 14–15 2005. (Full version to appear in Elsevier ENTCS).
20. J. Levy and M. Villaret. Linear second-order unification and context unification with tree-regular constraints. In L. Bachmair, editor, *Proc. of the 11th Int. Conference on Rewriting Techniques and Applications (RTA'2000)*, volume 1833 of *LNCS*, pages 156–171. Springer, 2000.
21. D. Maier. Database desiderata for an XML query language. Available from: `http://www.w3.org/TandS/QL/QL98/pp/maier.html`, 1998.
22. M. Marin and D. Ţepeneu. Programming with sequence variables: The *Sequentica* package. In P. Mitic, P. Ramsden, and J. Carne, editors, *Challenging the Boundaries of Symbolic Computation. Proc. of 5th Int.* Mathematica *Symposium*, pages 17–24, London, 2003. Imperial College Press.

23. M. Marin and T. Kutsia. Programming with transformation rules. *Analele Universitatii de Vest din Timisoara*, XVI:163–177, 2003.
24. M. Schmidt-Schauß. A decision algorithm for stratified context unification. *J. Logic and Computation*, 12(6):929–953, 2002.
25. M. Schmidt-Schauß and K. U. Schulz. Solvability of context equations with two context variables is decidable. *J. Symbolic Computation*, 33(1):77–122, 2002.
26. M. Schmidt-Schauß and J. Stuber. On the complexity of linear and stratified context matching problems. Research Report 4923, INRIA-Lorraine, France, 2003.
27. World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0. Second edition. Available from: `http://www.w3.org/`, 1999.

A Proof of Theorem 1

Theorem 1 is an immediate consequence of Soundness, Termination, Completeness, and Minimality theorems for $\mathfrak{M}$ given below.

Theorem 3 (Soundness of $\mathfrak{M}$). *Let Γ be a matching problem. Then every substitution $\sigma \in \mathcal{S}ol_{\mathfrak{M}}(\Gamma)$ is a matcher of Γ.*

Proof. (Sketch) Inspecting the rules in $\mathfrak{I}$ one can conclude that for a derivation $\Gamma; \varepsilon \Longrightarrow^{+} \emptyset; \sigma$ the problems $\Gamma\sigma$ and $\emptyset$ have the same set of matchers. It implies that σ is a matcher of Γ. □

Theorem 4 (Termination of $\mathfrak{M}$). *The algorithm $\mathfrak{M}$ terminates on any input.*

Proof. With each matching problem Δ we associate a complexity measure as a triple of non-negative integers $\langle n_1, n_2, n_3 \rangle$, where n_1 is the number of distinct variables in Δ, n_2 is the number of symbols in the ground sides of matching equations in Δ, and n_3 is the number of subterms in Δ of the form $f(s_1, \ldots, s_n)$, where s_1 is not a sequence variable. Measures are compared lexicographically. Every non-failing rule in $\mathfrak{I}$ strictly decreases the measure. Failing rules immediately lead to termination. Hence, $\mathfrak{M}$ terminates on any input. □

Theorem 5 (Completeness of $\mathfrak{M}$). *Let Γ be a matching problem and let ϑ be a matcher of Γ. Then there exists a derivation $\Gamma; \varepsilon \Longrightarrow^{+} \emptyset; \sigma$ such that $\sigma \leq \vartheta$.*

Proof. We construct the derivation recursively. For the base case $\Gamma_1; \sigma_1 = \Gamma; \varepsilon$ we have $\varepsilon \leq \vartheta$. Now assume that the system $\Gamma_n; \sigma_n$, where $n \geq 1$ and $\Gamma_n \neq \emptyset$, belongs to the derivation and find a system $\Gamma_{n+1}; \sigma_{n+1}$ such that $\Gamma_n; \sigma_n \Longrightarrow \Gamma_{n+1}; \sigma_{n+1}$ and $\sigma_{n+1} \leq \vartheta$. We have $\sigma_n \leq \vartheta$. Therefore, there exists φ such that $\sigma_n \varphi = \vartheta$ and φ is a matcher of Γ_n. Without loss of generality, we pick an arbitrary matching equation $s \ll t$ from Γ_n and represent Γ_n as $\{s \ll t\} \cup \Gamma'_n$. Depending on the form of $s \ll t$, we have three cases:

Case 1. The terms s and t are the same. We extend the derivation with the step $\Gamma_n; \sigma_n \Longrightarrow_{\mathsf{T}} \Gamma'_n; \sigma_n$. Therefore, $\sigma_{n+1} = \sigma_n \leq \vartheta$.

Case 2. The term s is an individual variable x. Then $x\varphi = t$. Therefore, for $\psi = \{x \mapsto t\}$ we have $\psi\varphi = \varphi$ and, hence, $\sigma_n \psi \varphi = \vartheta$. We extend the derivation with the step $\Gamma_n; \sigma_n \Longrightarrow_{\mathsf{IVE}} \Gamma'_n; \sigma_{n+1}$, where $\sigma_{n+1} = \sigma_n \psi \leq \vartheta$.

Case 3. The terms s and t are not the same and s is a compound term. The only non-trivial cases are those when the first argument of s is a sequence variable, or when the head of s is a context variable. If the first argument of s is a sequence variable $\overline{x}$ then φ must contain a binding $\overline{x} \mapsto \ulcorner t_1, \ldots, t_k \urcorner$ for $\overline{x}$, where $m \geq 0$ and t_i's are ground terms. If $k = 0$ then we take $\psi = \{\overline{x} \mapsto \ulcorner\urcorner\}$ and extend the derivation with the step $\Gamma_n; \sigma_n \Longrightarrow_{\mathsf{SVD}} \Gamma'_n; \sigma_{n+1}$, where $\sigma_{n+1} = \sigma_n \psi$. If $k > 0$ then we take $\psi = \{\overline{x} \mapsto \ulcorner t_1, \overline{x} \urcorner\}$ and extend the derivation with the step $\Gamma_n; \sigma_n \Longrightarrow_{\mathsf{W}} \Gamma'_n; \sigma_{n+1}$, where $\sigma_{n+1} = \sigma_n \psi$. In both cases we have $\sigma_{n+1} = \sigma_n \psi \leq \sigma_n \varphi = \vartheta$. If the head of s is a context variable $\overline{C}$ then φ must contain a binding $\overline{C} \mapsto C$ for $\overline{C}$, where C is a ground context. If $C = \circ$ then we take $\psi = \{\overline{C} \mapsto \circ\}$ and we extend the derivation with the step $\Gamma_n; \sigma_n \Longrightarrow_{\mathsf{CVD}} \Gamma'_n; \sigma_{n+1}$, where $\sigma_{n+1} = \sigma_n \psi$. If $C \neq \circ$ then C should have a form $f(t_1, \ldots, t_{j-1}, D, t_{j+1}, \ldots, t_m)$, where D is a context and $f(t_1, \ldots, t_m) = t$. Then we take $\psi = \{\overline{C} \mapsto f(t_1, \ldots, t_{j-1}, \overline{C}(\circ), t_{j+1}, \ldots, t_m)\}$ and extend the derivation with the step $\Gamma_n; \sigma_n \Longrightarrow_{\mathsf{W}} \Gamma'_n; \sigma_{n+1}$, where $\sigma_{n+1} = \sigma_n \psi$. In both cases $\sigma_{n+1} = \sigma_n \psi \leq \sigma_n \varphi = \vartheta$. □

Theorem 6 (Minimality). *Let Γ be a matching problem. Then $\mathcal{S}ol_{\mathfrak{M}}(\Gamma)$ is a minimal set of matchers of Γ.*

Proof. For any matching problem Δ the set

$$S(\Delta) = \{\varphi \mid \Delta; \varepsilon \Longrightarrow \Phi; \varphi \text{ for some } \Phi\}$$

is minimal. Moreover, every substitution ϑ in $S(\Delta)$ preserves minimality: If $\{\sigma_1, \ldots, \sigma_n\}$ is a minimal set of substitutions then so is the set $\{\vartheta\sigma_1, \ldots, \vartheta\sigma_n\}$. It implies that $\mathcal{S}ol_{\mathfrak{M}}(\Gamma)$ is minimal. □

B Proof of Theorem 2

Theorem 2 follows from Soundness, Termination, and Completeness theorems for $\mathfrak{M}_R$ given below.

Theorem 7 (Soundness of $\mathfrak{M}_R$). *Let Γ be a regular matching problem. Then every substitution $\sigma \in \mathcal{S}ol_{\mathfrak{M}_R}(\Gamma)$ is a regular matcher of Γ.*

Proof. (Sketch) Inspecting the rules in $\mathfrak{I}_R$ one can conclude that for a derivation $\Gamma; \varepsilon \Longrightarrow^+ \emptyset; \sigma$ every regular matcher of $\emptyset$ is also a regular matcher of $\Gamma\sigma$. It implies that σ is a regular matcher of Γ. □

Theorem 8 (Termination of $\mathfrak{M}_R$). *The algorithm $\mathfrak{M}_R$ terminates on any input.*

Proof. The tricky part of the proof is related with patterns containing the star "*". A derivation that contains an application of the RRET2 rule on a system with a selected matching equation and pattern-pair $s_0 \ll t_0, (\overline{x} \texttt{ in } \mathtt{R}_0^*, \mathtt{f})$ either fails or eventually produces a system that contains a matching equation $s_1 \ll t_1$ and a pattern-pair $(\overline{x} \texttt{ in } \mathtt{R}_1^*, 0)$ where $\mathtt{R}_1$ is an instance of $\mathtt{R}_0$ and $\overline{x}$ is the first argument of s_1:

$$\begin{aligned}
&\{s_0 \ll t_0, (\overline{x} \texttt{ in } \mathtt{R}_0^*, \mathtt{f})\} \cup \Gamma; \sigma \\
&\quad \Longrightarrow_{\mathsf{RRET2}} \{s_0\vartheta \ll t_0, (\overline{y} \texttt{ in } \mathtt{R}_0, 1), (\overline{x} \texttt{ in } \mathtt{R}_0^*, \mathtt{f})\} \cup \Gamma\vartheta; \sigma\vartheta \\
&\quad \Longrightarrow^+ \{s_1 \ll t_1, (\overline{x} \texttt{ in } \mathtt{R}_1^*, 0)\} \cup \Delta; \varphi.
\end{aligned}$$

Hence, the rule RRET2 can apply again on $\{s_1 \ll t_1, (\overline{x} \texttt{ in } \mathtt{R}_1^*, 0)\} \cup \Delta; \varphi$. The important point is that the total size of the ground sides of the matching equations strictly decreases between these two applications of RRET2: In $\{s_1 \ll t_1\} \cup \Delta$ it is strictly smaller than in $\{s_0 \ll t_0\} \cup \Gamma$. This is guaranteed by the fact that $(\overline{y} \texttt{ in } \mathtt{R}_0, 1)$ does not allow the variable $\overline{y}$ to be bound with the empty sequence. The same argument applies to derivations that contain an application of the RREF2 rule. Applications of the other rules also lead to a strict decrease of the size of the ground sides after finitely many steps. Since no rule increases the size of the ground sides, the algorithm $\mathfrak{M}_R$ terminates.

Theorem 9 (Completeness of $\mathfrak{M}_R$). *Let Γ be a regular matching problem, ϑ be a regular matcher of Γ, and $\mathcal{V}$ be a variable set of Γ. Then there exists a substitution $\sigma \in \mathcal{S}ol_{\mathfrak{M}_R}$ such that $\sigma \leq^{\mathcal{V}} \vartheta$.*

Proof. Similar to the proof of Theorem 5. □

Tree vs Dag Automata

Siva Anantharaman[1], Paliath Narendran[2], and Michaël Rusinowitch[3]

[1] LIFO - Université d'Orléans (France)
e-mail: siva@lifo.univ-orleans.fr
[2] University at Albany–SUNY (USA)
e-mail: dran@cs.albany.edu
[3] INRIA-Lorraine, Nancy (France)
e-mail: rusi@loria.fr

Abstract. The complexity of several classical algorithms has been lowered with success by using structures over dags instead of over trees. It is natural then to try to replace automata running over trees by those running over dags, at least for some of the problems solved by using tree automata techniques. The purpose of this paper is to show that algebraically dag automata behave very differently from tree automata, but they can be finer modeling tools in several complex situations.

1 Introduction

The expressive power of tree automata has proved to be very useful in several contexts, such as rewriting (e.g., [8]), the analysis of XML documents (e.g., [13]), and formal verification techniques based on set constraints. They have also been employed in solving unification problems over theories extending $ACUI$ (AC with Unit element plus Idempotence), see for instance [4] and [1, 2].

On the other hand, right from the early days of syntactic unification, dags have often been used for the same purposes as trees, generally with the advantage that algorithms using dag structures have a lower complexity than those using trees. It is therefore only natural to investigate the possibility of using automata over dags instead of over trees, for solving problems. Dag automata (DA) were first introduced and studied in detail in [6]: a dag automaton is a bottom-up tree automaton running on dags, not on trees. Their emptiness problem was shown in [6] to be NP-complete, and the membership problem was proved to be in NP; but the stability under complementation of the class of dag automata was raised as an open problem, closely linked with that of their determinization. These two issues are settled negatively by our results of [3]; the principal reason is that the set of all terms represented by the set of dags accepted by a deterministic DA is necessarily regular, but that is no longer true for a non-deterministic DA.

The notion of labeled tree automaton was introduced in [7] essentially as a means for some control over the runs: a labeled tree automaton is a tree automaton where the transitions are labeled; it runs on trees with labeled nodes; the runs have then to use transitions whose labels tally with those at the nodes reached. But their expressive power is the same as for tree automata, since any labeled tree automaton can be easily translated into a usual tree automaton without labels, in a way that preserves determinism and accepting runs.

Recall that for (deterministic or non-deterministic) tree automata, the membership problem is decidable in polynomial time, and universality is known to be EXPTIME-complete, cf. [7], Section 1.7, Theorems 10 and 14.

Now, one can define analogously the notion of a labeled dag automaton (LDA), as a dag automaton with labeled transitions, running on dags with labeled nodes, using transitions whose labels tally with those at the nodes reached. It was shown in [1] that unification modulo the theory *ACID* (obtained by adjoining a binary operator assumed 2-sided distributive over a basic *ACI* symbol) is decidable with a DEXPTIME lower bound and a NEXPTIME upper bound complexity; this was done by formulating an *ACID*-unification problem as the emptiness problem of a deterministic labeled dag automaton that can be constructed naturally, and in exponential time, from the given unification problem. Thus, if emptiness of *deterministic* LDAs could be shown to be decidable in polynomial time, one could have deduced that *ACID*-unification is DEXPTIME-complete. But our results of [3] show that deciding emptiness is NP-complete for deterministic LDAs. The reason for this is that an LDA can in general be translated into a DA without labels in a way that only preserves accepting runs, but not determinism.

We have also shown in [3] that (i) the class of dag automata is not stable under complementation, (ii) the membership problem is NP-hard for non-deterministic dag automata, and (iii) universality is undecidable for dag automata. The results on emptiness and membership are obtained via reduction from boolean satisfiability, while that on universality is obtained via reduction from the reachability problem for Minsky 2-counter machines [12]. Part of the constructions and proof details of [3] are reproduced in this paper, along with some additional comments, remarks, as well as examples borrowed from some of our other works. The intended message is that, however ill-behaved DAs and LDAs may be from an algebraic point of view, they can be the appropriate tools for modeling or analyzing several complex situations.

2 DAs: Definitions, Properties

A *term-dag* over a ranked alphabet Σ is a rooted dag where each node has a symbol from Σ such that:

(i) the out-degree of the node is the same as the rank of the symbol,
(ii) edges going out of a node are ordered, and
(iii) no two distinct subgraphs are isomorphic.

Every node represents a unique term in a term-dag, so we often treat "node" and "term" as synonymous on a term-dag.

Definition 1. *A term-dag automaton (or dag automaton, DA for short) over a ranked alphabet Σ is a tuple (Σ, Q, F, Δ), where Q is a finite non-empty set of states, $F \subseteq Q$ is the set of final (or accepting) states, and Δ is a set of transition rules of the form : $f(q_1, q_2, ..., q_n) \to q$, where $f \in \Sigma$ is of arity (rank) n, and the $q_i, \ldots, q_n, q$ are in Q.*

A *run* r of a DA $\mathbf{A} = (\Sigma, Q, F, \Delta)$ on a term-dag t is a mapping from the set of nodes of t to the set of states Q that respects the transition relation Δ; i.e., for every node u, if the symbol at u is f of arity k, then $f(r(u_1), \ldots, r(u_k)) \to r(u)$ must be a transition in Δ, where $u_1, \ldots, u_k$ are the successor-nodes of u given in order. A run r is *accepting* on t if and only if $r(t) \in F$, i.e, it maps the root node to an accepting state. A term-dag t is accepted by a DA iff there is an accepting run on t. The language of a DA is the set of all term-dags that it accepts.

In brief, a dag automaton is none other than a bottom-up automaton running on term-dags, not on trees. It was already pointed out in [6], that this makes an essential difference for the language. Thus, the following bottom-up automaton:

$$a \to q_1, \qquad a \to q_2, \qquad f(q_1, q_2) \to q_a$$

with q_0, q_1, q_a as states and q_a as the accepting state, has an empty language as a DA; however, as a tree automaton it accepts $f(a, a)$.

Remark 1. The above difference disappears though, if the DA is deterministic. (A dag automaton is said to be *deterministic* iff any two distinct transition rules have distinct left-hand-sides.) Indeed, if an automaton is bottom-up deterministic, then there is no difference whether it runs on a tree or on the dag representing this tree. It follows that if $\mathbf{A}$ is a deterministic DA and L its language, then the set of terms represented by the dags of L is a regular tree language. This last assertion is not true however, for general non-deterministic DAs, as is shown by the following construction.

Construction: Consider the infinite set M of term-dags defined recursively over the signature $\{a^{(0)}, g^{(2)}\}$, as follows (superscripts denote ranks):

- $a \in M$,
- if $t \in M$ then $g(t,t) \in M$,
- nothing else is in M

Note first that there is no DA accepting precisely the dags of M. The proof is by contradiction, based on a standard pumping argument. This is illustrated in the figure below. (Note: the same pumping argument also shows that the set of terms represented by the dags of M cannot be a regular tree language.)

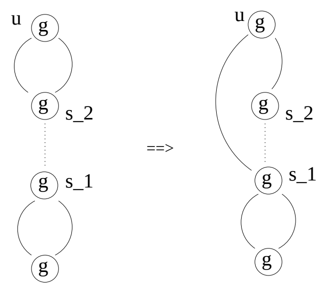

Fig. 1. Pumping out some nodes

However the complement M' of M (with respect to the set of *all* ground terms generated by $\{a^{(0)}, g^{(2)}\}$), *is* accepted by a DA. To show this observe first that, for any ground term t, we have $t \in M'$ iff t contains a subterm of the form $g(t_1, t_2)$ with $t_1 \neq t_2$. It is easily seen that the following DA, where q_a is the unique accepting state, accepts precisely the dags of the terms in M':

$$
\begin{array}{rlrl}
a & \longrightarrow q_0 & & \\
a & \longrightarrow q_1 & g(q_0,\, q_1) & \longrightarrow q_a \\
g(q_0,\, q_0) & \longrightarrow q_0 & g(q_1,\, q_0) & \longrightarrow q_a \\
g(q_0,\, q_0) & \longrightarrow q_1 & g(q_a,\, \_) & \longrightarrow q_a \\
g(q_1,\, q_1) & \longrightarrow q_0 & g(\_,\, q_a) & \longrightarrow q_a \\
g(q_1,\, q_1) & \longrightarrow q_1 & &
\end{array}
$$

From this construction and Remark 1, we get the following:

Proposition 1. *(i) The class of DAs is not closed under complementation.*

(ii) DAs are not determinizable in general.

(ii) There exists a set T of term-dags recognized by a DA such that the set of all terms represented by the dags in T is not a regular tree language.

This proposition answers all the three questions on DAs raised in [6], after the proof that their emptiness problem is in NP. (Note: It is easy to show that the class of DAs is stable under union and intersection.)

3 LDAs: Definitions, Properties

A *labeled term-dag*, or *lt*-dag for short, is a term-dag equipped additionally with a mapping from the nodes of the dag to a given set of labels E. Adding labels has several advantages: e.g., in the case where the labels are boolean, i.e., when $E = \{0, 1\}$, a labeled term-dag can be used to specify finite sets of terms. For instance, the *lt*-dag of Figure 2 naturally represents the set $\{a, g(g(a, a), b)\}$ of terms. More generally, if the nodes are labeled with boolean vectors of length m, then each *lt*-dag represents naturally an m-tuple of finite sets of terms.

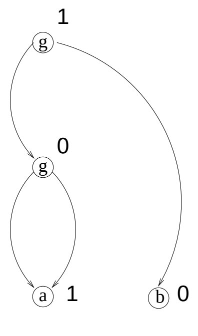

Fig. 2. A labeled term-dag

Definition 2. *A* labeled dag automaton *(or LDA for short) over a ranked alphabet Σ is a quintuple $(\Sigma, Q, F, E, \Delta)$, where Q is a finite non-empty set of states, $F \subseteq Q$ is the set of final (or accepting) states, E is a finite set of* labels*, and the transition relation Δ consists of* labeled *rewrite rules of the form $f(q_1, \ldots, q_k) \xrightarrow{l} q$, where k is the rank of f, $l \in E$, and $q_1, \ldots, q_k, q \in Q$.*

A *run* r of an LDA $(\Sigma, Q, F, E, \Delta)$ on an *lt*-dag t with label E is a mapping from the nodes of t to Q that respects the labels and the transition relation Δ in the following sense:

- for every node u on t, if the symbol at u is f of arity k, and the label of t at u is l, then transitions are possible via rules in Δ of the form $f(r(u_1), \ldots, r(u_k)) \xrightarrow{l} r(u)$, where u_1, …, u_k are the successor nodes of u on t given in order.

The above condition says in intuitive terms that the label on an LDA-transition must be the one at the node reached. A run r is said to be *accepting* on t iff $r(t) \in F$, i.e, it maps the root node to an accepting state. An *lt*-dag t is said to be accepted by an LDA iff there is an accepting run on t. The language of an LDA is the set of all *lt*-dags that it accepts.

An LDA is thus none other than a bottom-up automaton with labeled transition rules, running on labeled term-dags instead of on labeled trees. We say that an LDA is *deterministic* iff no two distinct transition rules have the same left-hand-side *and* the same label.

Unlabeling an LDA into a DA: To clearly bring out the difference with automata over trees, we first consider bottom-up labeled tree automata (that we shall refer to as LTAs): these are defined exactly as above, but operate on labeled *trees*. From the known results on unlabeled tree automata (such as emptiness ..), one deduces similar results for LTAs, via an 'unlabeling' process, as follows: For any function symbol $f \in \Sigma$ of rank n, and any label $l \in E$, define a new function symbol f_l, also of rank n; and for any transition rule $f(q_1, \ldots, q_n) \xrightarrow{l} q$ on the LTA, define an unlabeled transition rule $f_l(q_1, \ldots, q_n) \to q$. By 'pushing the labels onto the function symbols' in this manner, we derive in polynomial time a usual unlabeled tree automaton (TA) from the given LTA, such that the following holds: a labeled tree t is accepted by the LTA if and only if the TA accepts the unlabeled tree derived from t by pushing the label at each node onto the function symbol at that node; and the accepting runs are 'preserved up to labels'. Via such an unlabeling, not only do the results carry over from the TA to the LTA, but the complexity is unaffected; moreover, if the LTA is deterministic, the unlabeled TA derived will be deterministic too. This explains that the expressive power of LTAs is the same as that of TAs; and therefore, labeling a TA appears only as notational convenience.

But such an unlabeling process cannot be carried over to the case of LDAs, since it will destroy the structure of the term-dags. So, unlabeling an LDA has to be done by 'pushing the labels onto the state symbols', and not onto the function symbols. Even this has to be done with caution, in order that the complexity of the results concerned remains unaffected:

Proposition 2. *The emptiness problem for LDAs is in NP.*

Proof. This is a consequence of the result of [6] that the emptiness problem for DAs is in NP. Here is how. Given the LDA $\mathbf{A}$, construct an unlabeled DA denoted $\mathbf{A}'$, as follows: the states of $\mathbf{A}'$ are the pairs of form (q, L_q), denoted as $\hat{q}$, where q is a state of $\mathbf{A}$ and L_q is the set of all labels of the transitions of $\mathbf{A}$ which have q as target; the (unlabeled) transitions of $\mathbf{A}'$ are of the form $f(\hat{q_1}, \ldots, \hat{q_k}) \longrightarrow \hat{q}$, whenever $f(q_1, \ldots, q_k) \xrightarrow{l} q$ is a (labeled) transition on the given LDA $\mathbf{A}$; the accepting states of $\mathbf{A}'$ are the $\hat{q}$'s corresponding to the accepting states q on $\mathbf{A}$. Note that $\mathbf{A}'$ is constructed from $\mathbf{A}$ in linear time: its number of states is the same as for $\mathbf{A}$ (since L_q is completely determined by q on $\mathbf{A}$), and its number of transitions is at most that of $\mathbf{A}$.

It is not hard to check then that the LDA $\mathbf{A}$ has a non-empty language if and only if the unlabeled DA $\mathbf{A}'$ accepts some term-dag. □

Remark 2. In the above construction of the unlabeled $\mathbf{A}'$ from the LDA $\mathbf{A}$, the accepting runs are preserved up to labels; but even if the LDA $\mathbf{A}$ is deterministic, the DA $\mathbf{A}'$ will be non-deterministic in general. This shows that deterministic LDAs do not behave in general like deterministic LTAs; it turns out in particular, that their emptiness problem is NP-hard (so NP-complete, thanks to [6]):

Theorem 1. *The emptiness problem is* NP-hard for deterministic LDAs.

Proof. The proof is by reduction from boolean satisfiability. Let B be any arbitrarily chosen boolean formula over a given set of boolean variables $\{x_1, \ldots, x_n\}$, and the usual boolean connectives $\{\wedge, \vee, \neg\}$. Let t_B be a term-dag for B, and m its number of distinct nodes. We then construct an LDA $\mathbf{A}$ with $2m$ states, such that $\mathbf{A}$ accepts *exactly* the term-dag t_B labeled suitably with boolean values $0, 1$ *if and only if* B is satisfiable. The construction goes as follows.

Corresponding to each node we have two states which stand for that subformula getting the corresponding truth-value. For ease of exposition we represent the states in the form $q_{(s,0)}$ or $q_{(s,1)}$ where s is a subterm of t_B. The labels are 0 and 1. The labeled transition rules on $\mathbf{A}$ are of the form:

$$x_i \xrightarrow{0} q_{(x_i,0)}, \qquad x_i \xrightarrow{1} q_{(x_i,1)},$$

and, more generally, for any $h \in \{\wedge, \vee\}$, will have the form:

$$h(q_{(s_1,b_1)}, q_{(s_2,b_2)}) \xrightarrow{h(b_1,b_2)} q_{(h(s_1,s_2),h(b_1,b_2))}$$

where $b_1, b_2 \in \{0, 1\}$, and $h(s_1, s_2)$ is a subterm of t_B; for the connective $\neg$ we have the transitions of the form: $\neg(q_{(s,b)}) \xrightarrow{\neg b} q_{(\neg s, \neg b)}$. The unique accepting state is $q_{(t_B,1)}$. This LDA meets the requirements (cf. [3]). □

Remarks 3. i) By replacing the labels of the above constructed LDA by a "don't-care" boolean label (the LDA becomes non-deterministic, but) we may deduce that the membership problem for LDAs is NP-hard (so is NP-complete).

ii) The above proof is a further illustration that deterministic LDAs do *not* behave like deterministic LTAs: the reason is that on a tree the same subterm can be at two different nodes with different labels. Thus, a deterministic LTA can be easily constructed to accept the boolean formula $a \wedge \neg a$ as a suitably labeled tree.

iii) Exploiting the fact that on a dag two equal subterms must occupy the same node, one can construct a DA that accepts precisely all the non-accepting or incorrect computations of a deterministic 2-counter machine, cf. [3]. It follows that the universality problem for DAs is undecidable, consequently so is also the problem of equivalence of DAs.

4 LDA for Solving a Unification Problem

This section presents briefly the ideas on how the LDA formalism was used in [1], for solving the *ACID*-unification problem. (It was this problem that was the motivation for our initial interest in DAs and LDAs.) The full details – technical and complex – will be left out, but can be found in the research report RR-2004-12: *How useful are Dag Automata?*, `http://www.univ-orleans.fr/lifo/prodsci/rapports/RR2004.htm.fr`.

By *ACID* we mean the theory defined by the following equational axioms:

$$x + (y + z) \approx (x + y) + z, \quad x + y \approx y + x, \quad x + x \approx x.$$
$$x * (y + z) \approx (x * y) + (x * z), \quad (u + v) * w \approx (u * w) + (v * w)$$

ACID-unification with free constants is the problem of solving, modulo this theory, a finite family of equations of the form: $\{s_1 = t_1, \ldots, s_k = t_k\}$ as *finite, non-empty* sets of ground terms (over the free constants) for the variables of the problem. Such a problem can always be reduced to a *standard form,*: every equation in the problem has one of the following forms (respectively referred to as of type *'product', 'sum',* or *'constant'*):

$$x = y * z, \qquad u = v + w, \qquad u = c$$

where u, v, w, x, y, z are variables and c is any constant or 0. Since '+' is idempotent and '$*$' distributes left and right over '+', we may view this *ACID*-unification problem as a set constraint problem; e.g. if y and z are interpreted as sets of terms over $*$ and the constants, then $y * z = \{s * t \mid s \in y, t \in z\}$.

To every such problem **P**, we associate an LDA in such a way that solving **P** amounts – in the following sense – to showing that the language of the LDA is non-empty. Let $X_i, i = 1..n$, denote the set variables of the problem **P**:

(i) If $S_i, i = 1..n$, are (finite, non-empty) sets of ground terms such that $X_i = S_i, i = 1..n$, is a solution to **P**, then one can construct an *lt*-dag representing this family of sets (so labeled with boolean vectors of length n), such that the LDA associated accepts the *lt*-dag.

(ii) Conversely, if t is an *lt*-dag t (labeled with boolean vectors of length n) accepted by the LDA associated to **P**, the n sets of ground terms S_i recovered as elements of X_i from the labels of t may *not* have the *closure property* w.r.t. **P**, i.e. some of the 'product'-equations of **P** may remain unsatisfied; however, from the accepting run of the LDA on t, we can derive a grammar, with the help of which one can produce a solution to the problem **P**.

Suppose given an *ACID*-unification problem **P**, and let $\{X_i\}_{i=1..n}$ be its set variables, arranged in some fixed order. The *lt*-dags on which we shall consider runs of our LDA (to be defined), will be term dags over the symbol '$*$' and the given ground constants, labeled with n-bit vectors at their various nodes. These labels will in general be denoted as $(m_1, m_2, \ldots, m_n),\ m_i \in \{0, 1\},\ i = 1..n$. They have as semantics that $m_i = 1$ iff the subterm of the *lt*-dag at the current node is an element of the set X_i.

The LDA that we shall be associating with our *ACID*-problem is somewhat similar to the tree automata with free variables defined in [9]. Let **S** be the set of all $2n$-bit vectors of the form $\{..., l_i, ...; ..., h_i, ...\}$ such that $l_i \leq h_i$ for all $i \in 1..n$; the elements of **S** will be referred to as *pstates*; we denote them by barred capital letters such as $\overline{A}, \overline{B}, \ldots$. For any pstate $\overline{A} \in \mathbf{S}$, we shall be denoting its first-half (or lower) and second-half (or higher) n-bit vectors by $\overline{A}.l, \overline{A}.h$ respectively.

Any state A of our LDA will have as its first component a pstate, that we shall denote $\overline{A}$; this corresponds to two sets of boolean valuations on the set expressions $\{X_i\}_{i=1..n}$; the lower i-th bit $\overline{A}.l_i$ of the LDA state A will signify (when 1) that the term *at* the current node on an lt-dag mapped to A under a given run r is accepted as element of X_i; the upper i-th bit $\overline{A}.h_i$ is meant to signify (when 1) that some subterm *below* the current node has been accepted in X_i by the run; this explains why we only consider pstates $\overline{A}$ with $\overline{A}.l \leq \overline{A}.h$.

An accepting state on the LDA will then be in particular such that $\overline{A}.h_i = 1$ for all $i = 1..n$; but this condition alone will *not* be sufficient for acceptance: many of the 'product'-equations of $\mathbf{P}$ may still remain unsatisfied. To circumvent this, we add as the second component of any state A of the LDA, a set of equalities formed from some new symbols called *dsymbols* which are meant to keep some book-keeping on the nodes traversed by the run. A dsymbol is of the form $X^i_{\overline{B}}$, where $1 \leq i \leq n$, and $\overline{B}$ is a pstate. The second component at any state A will be referred as the *defect set* at A and denoted as $\mathbf{M}_A$; it will contain some dsymbols doing the book-keeping along the run, together with some defect equalities: a *defect equality* over the dsymbols is by definition an equality having one of the two following forms:

$$X^k_{\overline{C}} = X^i_{\overline{A}} X^j_{\overline{B}}, \qquad \mathbf{T} = X^i_{\overline{A}} X^j_{\overline{B}}$$

where $\overline{A}, \overline{B}, \overline{C} \in \mathbf{S}$, and (k, i, j) are triples corresponding to some 'product'-equation in $\mathbf{P}$ of the form $X_k = X_i * X_j$. Equalities of the first type are said to be *closed*, and those of the second type *open*. (Note: the number of all the dsymbols and defect equalities is polynomial in the size of $\mathbf{S}$.)

The presence of a dsymbol $X^i_{\overline{B}}$ in the defect set $\mathbf{M}_A$ of a state A has the semantics that some node below, along the run, got mapped to a state on the LDA whose pstate is $\overline{B}$, and the subterm at that node is in the set X_i. Closed (resp. open) defect equalities in the defect set $\mathbf{M}_A$ keep track of the 'product'-equations of $\mathbf{P}$ which have remained 'covered' along the run (resp. which still remain to be covered). The open defect equalities will also be witnesses that the term accepted at the current node by the run contributes (along with a subterm accepted at some earlier node) to one or more of the 'product'-equations of $\mathbf{P}$.

By definition, the *accepting states* of the LDA are the A satisfying the following two conditions:

(i) $\overline{A}.h_i = 1$ for all $i \in \{1, .., n\}$;

(ii) the defect set $\mathbf{M}_A$ at A contains no *open* defect equalities.

We skip the other technical details on how the transitions are defined on the LDA associated to $\mathbf{P}$, in particular on how the pstates and the defect sets evolve under them. Suffices to note that they are defined in such a way that the LDA will be deterministic. (This fact is crucial for proving the completeness of the LDA approach for solving $ACID$-unification problems.)

We give below a couple of examples illustrating the approach sketched above; the first one shows in particular why we need a grammar from an accepted lt-dag to generate a solution for the unification problem. Guessing how the runs get initialized must not be too difficult, from either of the examples.

4.1 Examples of $ACID$-Unification

Example 1. Consider the $ACID$-problem $Z =^? X * Y$, and order the list of variables as $\{X, Y, Z\}$. Let a, b, c, d be ground constants; we construct an accepted lt-dag rooted at $(c * (c * (a * b))) * (c * d)$, with leaves at a, c put into X; and at b, d put into Y; in addition the nodes at $(a * b), (c * (a * b))$ are put into Y; cf. figure below. From the labels of this accepted lt-dag we only get the following terms in the set Z: $(a*b)$, $c*(a*b)$, $c*(c*(a*b))$, $(c*d)$, so the sets of terms recovered from the labels of the lt-dag is not a solution to the problem.

The accepting run is as follows (in the figure, the states to which the nodes get mapped are indicated in italic capitals).

$a, c \xrightarrow{100} A = (100; 100; \{X_{\overline{A}}\});$ $\qquad b, d \xrightarrow{010} B = (010; 010; \{Y_{\overline{B}}\});$
$A*B \xrightarrow{011} C = (011; 111; \{X_{\overline{A}}, Y_{\overline{B}}, Y_{\overline{C}}, Z_{\overline{C}},\ Z_{\overline{C}} = X_{\overline{A}}Y_{\overline{B}},\ \mathbf{T} = X_{\overline{A}}Y_{\overline{C}}\});$
$A * B \xrightarrow{001} D = (001; 111; \{X_{\overline{A}}, Y_{\overline{B}}, Z_{\overline{D}},\ Z_{\overline{D}} = X_{\overline{A}}Y_{\overline{B}}\});$
$A*C \xrightarrow{011} C = (011; 111; \{X_{\overline{A}}, Y_{\overline{B}}, Y_{\overline{C}}, Z_{\overline{C}},\ Z_{\overline{C}} = X_{\overline{A}}Y_{\overline{B}},\ \mathbf{T} = X_{\overline{A}}Y_{\overline{C}}\});$

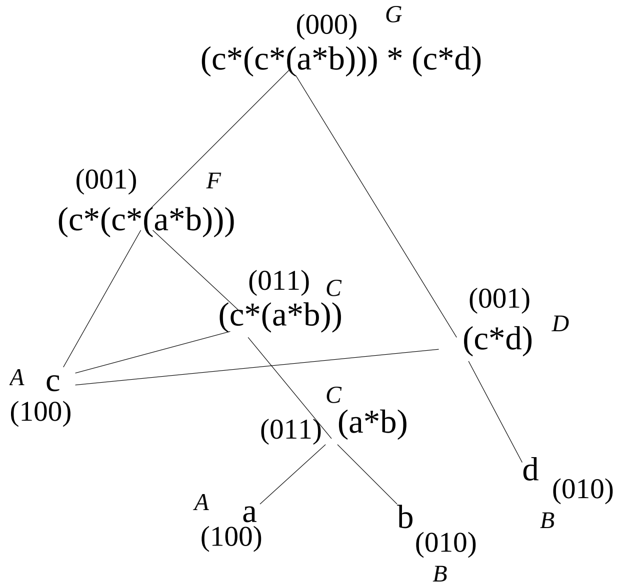

Fig. 3. An accepted lt-dag for $Z =^? X * Y$

Then: $\qquad A * C \xrightarrow{001} F = (001; 111; \mathbf{M}_F)$, where
$\mathbf{M}_F = \{X_{\overline{A}}, Y_{\overline{B}}, Y_{\overline{C}}, Z_{\overline{C}},\ Z_{\overline{C}} = X_{\overline{A}}Y_{\overline{B}}, Z_{\overline{F}}, Z_{\overline{F}} = X_{\overline{A}}Y_{\overline{C}}\}$.

And finally: $\qquad F * D \xrightarrow{001} G = (000; 111; \mathbf{M}_G)$, where
$\mathbf{M}_G = \{X_{\overline{A}}, Y_{\overline{B}}, Y_{\overline{C}}, Z_{\overline{C}},\ Z_{\overline{C}} = X_{\overline{A}}Y_{\overline{B}},\ Z_{\overline{D}},\ Z_{\overline{D}} = X_{\overline{A}}Y_{\overline{B}}, Z_{\overline{F}}, Z_{\overline{F}} = X_{\overline{A}}Y_{\overline{C}}\}$, where $\overline{D} = \overline{F}$. The states D, F, G are all accepting; so the sub-dags rooted at the nodes mapped to these states are accepted. The solution for the problem derived at the node mapped to D is the simplest; the assignment is: $X =$

$\{\overline{A}\},\ \ Y = \{\overline{B}\},\ \ Z = \{\overline{A} * \overline{B}\}$. where $\overline{A} \longrightarrow a \mid c,\ \overline{B} \longrightarrow b \mid d$; this is a correct solution.

Let us compute the contribution of the run to X, Y, Z on the sub-dag rooted at node F, by looking at $\mathbf{M}_F$: The productions of the grammar w.r.t. this accepting sub-run are as follows:

$$\overline{F} \longrightarrow \overline{A} * \overline{C}, \quad \overline{C} \longrightarrow \overline{A} * \overline{B}, \quad \overline{A} \longrightarrow a \mid c, \quad \overline{B} \longrightarrow b \mid d.$$

So the contribution of the run is the assignment: $X = \{\overline{A}\},\ \ Y = \{\overline{B},\ \overline{A} * \overline{B}\},\ \ Z = \{\overline{A} * \overline{B},\ \overline{A} * (\overline{A} * \overline{B})\}$, where $\overline{A} \longrightarrow a \mid c,\ \overline{B} \longrightarrow b \mid d$; this is again a correct solution.

The grammar for the accepting run at the root node G has an additional production: $\overline{D} \longrightarrow \overline{A} * \overline{B}$, with $\overline{D} = \overline{F}$. The solution derived here is the same as at F: the terms derivable from this production are also derivable from $\overline{C}$. □

Example 2. The $ACID$-unification problem: $X + Z = X * Y + U$ may be transformed into the following standard form:

$$V = X + Z, \quad V = W + U, \quad W = X * Y.$$

Let us show that the lt-dag in Figure 4 is accepted by the associated LDA. Arrange the set variables into an ordered list, say as $\{X, Y, Z, U, V, W\}$.

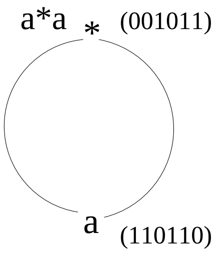

Fig. 4. lt-dag solving $V = X + Z,\ V = W + U,\ W = X * Y$

The states on the LDA are therefore 12-bit vectors. From the node 'a' on the lt-dag with label $m = (110110)$ we first have an initial transition to a state A:

$$a \xrightarrow{110110} A = (110110\,; 110110\,; \mathbf{M}_A)$$

where $\mathbf{M}_A = \{X_{\overline{A}}, Y_{\overline{A}}, U_{\overline{A}}, V_{\overline{A}},\ \mathbf{T} = X_{\overline{A}} Y_{\overline{A}}\}$ representing the fact that at the current node the values $X = Y = a = V = U$ have been accepted (and a product $X * Y$ remains to be covered). Next we have a transition, with label (001011), from $A * A$ to the state $B = (001011; 111111; \mathbf{M}_B)$, where $\mathbf{M}_B = \{X_{\overline{A}}, Y_{\overline{A}}, U_{\overline{A}}, V_{\overline{A}}, Z_{\overline{B}}, V_{\overline{B}}, W_{\overline{B}}, W_{\overline{B}} = X_{\overline{A}} Y_{\overline{A}}\}$; at this node, the assignments $W = a * a = Z = V$ have been accepted. The state reached at the root node is an accepting state. So the lt-dag is accepted.

Note that in this case the lt-dag t has the closure property w.r.t. the problem: the sets of terms deduced from the labels of t is a solution to the $ACID$-problem.

On the other hand the grammar derived from the accepting run has two productions: $\overline{B} \longrightarrow \overline{A} * \overline{A},\ \ \overline{A} \longrightarrow a$. The contribution of the run to the variables are therefore the sets of terms representable as: $X = \{\overline{A}\} = Y = U,\ \ V = \{\overline{A},\ \overline{A} * \overline{A}\},\ \ Z = W = \{\overline{A} * \overline{A}\}$; that is to say:

$$X = \{a\} = Y = U, \qquad V = \{a,\ a * a\}, \qquad Z = W = \{a * a\}$$

which is the same as the one deduced from the labels of the lt-dag. □

5 Conclusion

The results of [3] that we have partially presented here illustrate that the algebraic behavior of dag automata can be very different from that of tree automata. But at the same time the proof techniques employed in [3], as well as the application to unification that we have illustrated in Section 4, show that the expressive power of dag automata can be extremely useful in modeling and/or analyzing several complex situations. Furthermore, the dag representation has been shown to be clearly space efficient for compressed XML documents, cf. [5, 10, 11]. All these lead us to the belief that DAs and LDAs may also useful for handling certain classes of XML/XPath queries, especially where bottom-up analysis techniques can be brought into play.

References

1. S. Anantharaman, P. Narendran, M. Rusinowitch, *ACID-Unification is NEXPTIME-Decidable*, Proc. of MFCS'03, pp. 169–179, LNCS 2747, 2003.
2. S. Anantharaman, P. Narendran, M. Rusinowitch, *Unification modulo ACUI plus Distributivity Axioms*, Journal of Automated Reasoning, Vol. 33, $n^0 1$, pp.1–28, 2004.
3. S. Anantharaman, P. Narendran, M. Rusinowitch, *Closure Properties and Decision Problems of Dag Automata* Information Processing Letters, (To appear), 2005.
4. F. Baader, P. Narendran, *Unification of Concept Terms in Description Logics*, Journal of Symbolic Computation 31 (3):277–305, 2001.
5. P. Buneman, M. Grohe, C. Koch, *Path queries on compressed XML.* Proc. of the 29*th* Conf. on VLDB, 2003, pp. 141–152, Ed. Morgan Kaufmann.
6. W. Charatonik, *Automata on DAG Representations of Finite Trees*, Technical Report MPI-I-99-2-001, Max-Planck-Institut für Informatik, Saarbrücken, Germany.
7. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, *Tree Automata Techniques and Applications*, `http://www.grappa.univ-lille3.fr/tata/`
8. T. Genet and F. Klay, *Rewriting for Cryptographic Protocol Verification*, Proc. of the 17th CADE, pp. 271-290, LNAI 1831, 2000.
9. R. Gilleron, S. Tison, M. Tommasi, *Set Constraints and Tree Automata*, Information and Computation 149, 1–41, 1999. (cf. also Technical Report IT 292, Laboratoire-LIFL, Lille, 1996.)
10. M. Frick, M. Grohe, C. Koch, *Query Evaluation of Compressed Trees*, Proc. of LICS'03, IEEE, pp. 188–197.
11. M. Marx. *XPath and Modal Logics for Finite DAGs.* Proc. of TABLEAUX'03, pp. 150–164, LNAI 2796, 2003.
12. M. Minsky, *Computation: Finite and Infinite Machines*, *Prentice-Hall International*, London, 1972.
13. F. Neven, Automata Theory for XML Researchers, SIGMO Record 31(3), September 2002.

Relating Nominal and Higher-Order Pattern Unification

James Cheney

University of Edinburgh
jcheney@inf.ed.ac.uk

Abstract. Higher-order pattern unification and nominal unification are two approaches to unifying modulo some form of α-equivalence (consistent renaming of bound names). The higher-order and nominal approaches seem superficially dissimilar. However, we show that a natural *concretion* (or name-application) operation for nominal terms can be used to simulate the behavior of higher-order patterns. We describe a form of nominal terms called *nominal patterns* that includes concretion and for which unification is equivalent to a special case of higher-order pattern unification, and then show that full higher-order pattern unification can be reduced to nominal unification via nominal patterns.

1 Introduction

Higher-order unification is the unification of simply-typed λ-terms up to α-, β-, and (sometimes) η-equivalence. It has been studied for over thirty years.. Although it is undecidable and of infinitary unification type, Huet's algorithm [6] performs well in practice, and Miller identified a well-behaved special case called *higher-order pattern unification* [7, 13, 11, 1] that is decidable in linear time and possesses unique most general unifiers. In higher-order patterns, uses of metavariables (variables for which terms may be substituted) are limited so that the nondeterministic search needed in full higher-order unification can be avoided. A key aspect of higher-order unification is that substitution is capture-avoiding. For example, the unification problem $\lambda x.M \approx ? \ \lambda y.f\ y$ has no solution, since although both sides could be made equal by making a capturing substitution $f\ x$ for M, there is no way to make both sides equal using capture-avoiding substitution to instantiate M.

Nominal unification [15] is the unification of *nominal terms*, which include special *name* or *atom* symbols, a *name-swapping* operation, an *abstraction* operation for name-binding, and *freshness* relation. Equality and freshness for nominal terms coincide with classical definitions of α-equivalence and the "not-free-in" relation $- \notin FV(-)$, respectively. Nominal unification is decidable in at worst quadratic time (exact complexity bounds are not yet known). Nominal unification is based on *nominal logic*, a logic formalizing a novel approach to abstract syntax with bound names due to Gabbay and Pitts [3]. There are two aspects of nominal unification that contrast sharply with higher-order unification. First, abstraction is not considered to bind names, and metavariables may mention arbitrary names, so that the problem $\langle a \rangle M \approx ? \ \langle b \rangle f(b)$ does have solution $M = f(a)$. Second, abstractions are not considered to be functions, and there is no built-in notion of "abstraction application". Instead, nominal unifiers can be expressed in terms of the swapping operation $(a\ b) \cdot t$, which describes the result of exchanging all occurrences of a and b within t, and freshness constraints $a \# t$, which assert that a name a is fresh for a term t. For example, the unification problem $\langle a \rangle M \approx ? \langle b \rangle f(N, b)$ has most general solution $M = f((a\ b) \cdot N, a)$ subject to the constraint that $a \# N$. This unifier shows how to compute M as a function of N while excluding false solutions such as $M = f(b, a), N = a$, since $\langle a \rangle f(a, b) \not\approx \langle b \rangle f(b, a)$.

Despite these differences, nominal and higher-order pattern unification appear closely related. In fact, at first glance, one might wonder if they are not merely different presentations of the same algorithm. Both are techniques for equational reasoning about languages involving bound identifiers. Both algorithms rely on computing with permutations of bound names: the higher-order pattern restriction can be seen as a sufficient condition to ensure that such permutations always exist. In fact, as noted by Urban, Pitts, and Gabbay [15], there is a translation from nominal unification problems to higher-order pattern unification problems that preserves satisfiability. In this translation, metavariables are "lifted" so as to be functions of all the names in context. A freshness constraint such as $a \# M$ can be translated to an equation like $\lambda a, b, c.M\ a\ b\ c \approx \lambda a, b, c.N\ b\ c$, which asserts that M cannot be dependent on its first argument (namely, a). However, as argued by Urban et al., it is not straightforward to convert the resulting solutions back to solutions to the original nominal unification problem. As a result, it appears much easier to solve such problems directly using Urban et al.'s algorithm (which seems much simpler than that for higher-order pattern unification in any case).

Another reason to study the relationship between higher-order pattern unification and nominal unification is to provide a logical foundation for higher-order patterns. While both nominal and higher-order unification are grounded in clear logical foundations, higher-order patterns appear motivated solely by algorithmic concerns. If higher-order patterns can be explained using nominal terms, the semantic foundations of the latter could also be used for the former.

In this paper we argue that higher-order pattern unification can be reduced to nominal unification. This relationship helps justify the higher-order pattern restriction and explain why it works. The key idea is that the pattern restriction (that metavariable occurrences are of the form $X\ \overline{v}$ where $\overline{v}$ is a list of distinct names) is essentially the same as a natural freshness restriction on the *concretion* operation. This operation is an elimination form for abstraction that has been considered in some versions of FreshML [12, 14], but so far not incorporated into nominal logic or nominal unification.

The structure of this paper is as follows. First (Section 2), we review higher-order pattern unification and nominal unification. In Section 3, we introduce a type system that enforces the higher-order pattern restriction in a particularly convenient way. In Section 4, we identify a variant of nominal terms called *nominal patterns* that includes the concretion operation and for which unification is equivalent to a special case of higher-order pattern unification. We then (Section 5) show that full higher-order pattern unification can be reduced to nominal pattern unification and (Section 6) that nominal pattern unification can be reduced to nominal unification. Put together, these reductions show that higher-order pattern unification can be implemented via nominal unification. Section 7 and Section 8 discuss related work and conclude.

2 Background

2.1 Higher-order terms, patterns, and unification

Consider infinite sets of variable names $x, y, z, \ldots \in \mathit{Var}$ and metavariables $X, Y, Z \ldots \in \mathit{MVar}$. The terms of the λ-calculus are as follows:

$$t ::= c \mid x \mid \lambda x.t \mid t\ t' \mid X$$

We assume that common notions such as the set of free variables of a term $FV(-)$, α-equivalence, capture-avoiding substitution $-[-/-]$ etc. are defined as usual. In addition, we assume that there are some given base types δ, that types include function types $\tau \to \tau'$, and that well-formedness is defined as usual provided that types are assigned to constants via a signature Σ. A term with no free variables is called *closed*; a term with no metavariables is called *ground*. We often write $f(t_1, \ldots, t_n)$ as a shorthand for $f\ t_1\ \cdots\ t_n$.

Terms are considered equal up to α-equivalence plus two additional equations: β-reduction and η-expansion

$$\begin{array}{lll} (\beta)\ (\lambda x.t)\ u \approx t[u/x] & & \\ (\eta) \qquad\quad t \approx \lambda x.(t\ x) & & (t : \tau \to \tau', x \notin FV(t)) \end{array}$$

We write θ for a substitution mapping metavariables to λ-terms. Such a substitution may be applied to any λ-term by replacing each metavariable X with $\theta(X)$. If Γ and Γ' are contexts consisting only of metavariables, we write substitution is well-formed ($\Gamma' \vdash \theta : \Gamma$) provided that for each $X : \tau \in \Gamma$, $\Gamma' \vdash \theta(X) : \tau$. Thus, there is no danger of variable capture during substitution, and we have:

Lemma 1. *If $\Gamma \vdash \theta : \Gamma'$ and $\Gamma' \vdash t : \tau$, then $\Gamma \vdash \theta(t) : \tau$.*

We consider higher-order unification to be unification of λ-terms up to the above equational theory, that is, up to $\alpha\beta\eta$-equivalence. (Higher-order unification sometimes refers to unification up to only α and β-equivalence, but for this paper, we do not consider this problem.) Huet [6] gave an algorithm for generating complete sets of higher-order unifiers which performs well in practice. Technically, we consider only problems of the form $\exists \overline{X}.\forall \overline{y}.t \approx ?\ u$, since substitutions θ cannot mention free variables. This excludes problems such as $\forall y.\exists X.X \approx ?\ y$. However, such problems can always be transformed to equivalent $\exists\forall$-problems by *raising* [9] metavariables in order to make their dependence on other variables explicit: for example, transforming $\forall y.\exists X.X \approx ?\ y$ to $\exists F.\forall y.F\ y \approx ?\ y$, where $X \approx F\ y$.

Miller investigated a decidable special case of higher-order unification called *higher-order patterns*. To define higher-order patterns, we first recall that any λ-term (possibly involving metavariables) can be put into a normal form called *η-long, β-normal form* (or ηlβn form), such that (a) no β-redices exist, and

(b) no η-expansions can be performed without introducing a β-redex. Note that this normal form is dependent on the types of metavariables. For example, the normal form of $\lambda y,z.(\lambda x.xy)\ (F\ G)$ is $\lambda y,z,b.F\ (\lambda a.G\ a)\ y\ b$, provided $F : (\tau \to \tau') \to \sigma_1 \to \sigma_2 \to \sigma$ and $G : \tau \to \tau'$. Such normal forms conform to the following grammar:

$$t ::= \lambda\overline{v}.x\ \overline{t} \mid \lambda\overline{v}.X\ \overline{t}$$

The insight behind higher-order pattern unification is that all the nondeterminism in higher-order unification comes about because of uncertainty concerning how an unknown X can act on its arguments $\overline{t}$. In general, $\overline{t}$ may include repeated variables or more complex terms involving other metavariables. In higher-order patterns, this uncertainty is eliminated by requiring the argument list $\overline{t}$ in each subterm of the form $\lambda\overline{v}.X\ \overline{t}$ to be a list of distinct bound variables $\overline{w}$. Thus, the above example ηlβn-normal form $\lambda y,z,b.F\ (\lambda a.G\ a)\ y\ b$ is not a pattern, while $\lambda x,y.c\ (F\ y\ x)$ is a pattern.

Higher-order patterns are closed under substitution modulo β-normalization; in fact, the only redices introduced by substituting a higher-order pattern for a metavariable in another higher-order pattern are of the form called β_0 by Miller:

$$(\beta_0) \quad (\lambda x.t)\ y = t[y/x]$$

Unification for higher-order patterns is decidable (in linear time [13]) and most general unifiers exist.

2.2 Nominal terms and unification

We now consider a different language called *nominal terms*[1]. Let ν, ν' be basic *name types*. Let Nm be a set of *names* $a_\nu, b_{\nu'}, \ldots$ tagged with name types and let $MVar$ be a set of metavariables $X, Y, Z, \ldots$. The set of nominal terms is generated by the grammar

$$t ::= a_\nu \mid \langle a_\nu\rangle t \mid c \mid t_1\ t_2 \mid \pi \cdot X \quad \pi ::= \mathsf{id} \mid (a_\nu\ b_\nu) \circ \pi$$
$$\tau ::= \sigma \mid \sigma \to \tau \quad \sigma ::= \delta \mid \nu \mid \langle\nu\rangle\delta$$

Metavariables are annotated with *suspended permutations* of names, that are to be applied to any value substituted for the variable. A nominal term with no metavariables is called *ground*.

Terms of the form $\langle a\rangle t$ are called *abstractions*. An abstraction is an object with a single bound name. However, the name is not considered syntactically bound as in a λ-abstraction; instead, an abstraction describes a semantic value with a bound name. For example, $\langle a\rangle b$ and $\langle b\rangle b$ are not considered to be the same term; however, they have the same meaning. In particular, while term equality behaves like (and is intended to model) α-equivalence for *ground terms*, this is not the case for terms mentioning metavariables (e.g., the equation $\langle a\rangle X \approx \langle b\rangle X$ is not valid in general).

[1] Our version of nominal terms is superficially different from that used on Urban, Pitts, and Gabbay's paper, in order to minimize the number of unimportant differences from higher-order patterns.

We assume that there is a signature Σ assigning types τ to constants c, such that there are no constants or other closed terms inhabiting any name type. A permutation is considered well-formed if it is composed of swappings of names of the same type only. Contexts Γ associate metavariables to types. The following well-formedness rules are considered:

$$\frac{}{\Gamma \vdash a_\nu : \nu} \quad \frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau} \quad \frac{\pi \text{ well-formed}}{\Gamma, X : \tau \vdash \pi \cdot X : \tau}$$
$$\frac{\Gamma \vdash t : \tau \to \tau' \quad \Gamma \vdash u : \tau'}{\Gamma \vdash t\ u : \tau'} \quad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \langle a_\nu \rangle t : \langle \nu \rangle \tau}$$

We define a *swapping* function on nominal terms as follows:

$$(a\ b) \cdot a' = \begin{cases} b & (a = a') \\ a & (b = a') \\ a' & (a \neq a' \neq b) \end{cases} \qquad \begin{aligned} (a\ b) \cdot c &= c \\ (a\ b) \cdot (t_1\ t_2) &= ((a\ b) \cdot t_1)\ ((a\ b) \cdot t_2) \\ (a\ b) \cdot \langle a \rangle t &= \langle (a\ b) \cdot a \rangle (a\ b) \cdot t \\ (a\ b) \cdot (\pi \cdot X) &= (a\ b) \circ \pi \cdot X \end{aligned}$$

Also, we define $\pi \cdot t$ as follows:

$$\mathsf{id} \cdot t = t \qquad (a\ b) \circ (\pi \cdot t) = ((a\ b) \cdot \pi) \cdot t$$

We are now in a position to define the meaning of nominal terms. We do this by introducing axioms describing equality and an auxiliary *freshness* relation.

$$\frac{}{a \approx a} \quad \frac{}{c \approx c} \quad \frac{t \approx t' \quad u \approx u'}{t\ u \approx t'\ u'} \quad \frac{t \approx u}{\langle a \rangle t \approx \langle a \rangle u} \quad \frac{t \approx (a\ b) \cdot u \quad a \# u \quad (a \neq b)}{\langle a \rangle t \approx \langle b \rangle u}$$
$$\frac{a \neq b}{a \# b} \quad \frac{}{a \# c} \quad \frac{a \# t \quad a \# u}{a \# t\ u} \quad \frac{}{a \# \langle a \rangle t} \quad \frac{a \# t \quad (a \neq b)}{a \# \langle b \rangle t}$$

Given a substitution function θ mapping metavariables to terms, we write $\theta(t)$ for the result of applying substitution θ to term t. To be precise, the definition of substitution is as follows.

$$\begin{aligned} \theta(a) &= a \qquad & \theta(\langle a \rangle t) &= \langle a \rangle \theta(t) \\ \theta(c) &= c \qquad & \theta(\pi \cdot X) &= \pi \cdot \theta(X) \end{aligned} \quad \theta(t_1\ t_2) = \theta(t_1)\ \theta(t_2)$$

We require that substitutions are well-formed so that they preserve types, but (unlike for higher-order unification) substitutions are allowed to mention both metavariables and names, so "capturing" substitutions are allowed. For example, if $\theta(X) = a$ then $\theta(\langle a \rangle X) = \langle a \rangle a$.

If θ is a valuation (ground substitution), we write $\theta \vDash t \approx u$ to indicate that $\theta(t) \approx \theta(u)$ and write $\theta \vDash a \# t$ if $a \# \theta(t)$. As usual, a formula A is valid (satisfiable) if for all (resp. some) well-formed substitutions, $\theta \vDash A$ holds. This is extended to validity or satisfiability of sets of formulas P in the obvious way. Similarly, if P is a set of formulas, we write $P \vDash A$ to indicate that whenever $\theta \vDash P$, we also have $\theta \vDash A$.

Urban et al.'s nominal unification algorithm solves the satisfiability problem for sets of equations and freshness constraints. Given a problem P, it produces a unique (up to renaming) most general answer of the form θ, ∇, where θ is a substitution and ∇ is a set of freshness constraints of the form $a \# X$. This answer has the property that $\nabla \vDash \theta(P)$. Moreover, for any other answer ∇', θ' having this property, there exists a substitution ρ such that $\rho(\nabla) \vDash \nabla'$ and $\rho(\nabla) \vDash \rho \circ \theta \approx \theta'$ (where $\theta \approx \theta'$ means $dom(\theta) = dom(\theta')$ and $\forall X \in dom(\theta).\theta(X) \approx \theta'(X)$).

Urban et al. argue that their algorithm can be implemented in quadratic time; however, the exact complexity has not been established. We omit the precise details of the algorithm.

3 A Refined Type System for Higher-Order Patterns

We modify the notation of λ-terms to distinguish between *rigid* applications involving terms $t\ u$ where the head of t is rigid (i.e., a constant or bound variable), and *flexible* applications $t\,\hat{}\,a$, where the head of t is flexible (a metavariable). Also, we assume that variables are tagged with their types: for example, x_τ indicates that x is a variable of type τ. The grammar of such terms is as follows:

$$t ::= c \mid x_\tau \mid \lambda x_\tau.t \mid t\ t' \mid X \mid t\,\hat{}\,x_\tau$$

We use a type system for ηlβn-normalized terms that enforces the pattern restriction. Contexts Γ bind metavariables to types. There are three judgment forms: $\Gamma \vdash t \downarrow \tau$, indicating that t is a rigid atomic term of type τ; $\Gamma \vdash t \Downarrow \tau$, indicating that t is a flexible atomic term of type τ; and $\Gamma \vdash t \uparrow \tau$, indicating that t is a normal term of type τ. Examples of rigid atomic, flexible atomic, and normal terms include $x\ (\lambda y.y)z$, $X\,\hat{}\,y\,\hat{}\,z$, and $\lambda x, y, z.y\ (x\ (\lambda y.y)z)\ (X\,\hat{}\,y\,\hat{}\,z)$, respectively. The well-formedness rules for nominal patterns are as follows:

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c \downarrow \tau} \quad \frac{}{\Gamma \vdash x_\tau \downarrow \tau} \quad \frac{\Gamma \vdash t \uparrow \tau'}{\Gamma \vdash \lambda x_\tau.t \uparrow \tau \to \tau'} \quad \frac{\Gamma \vdash t \downarrow \tau \to \tau' \quad \Gamma \vdash u \uparrow \tau}{\Gamma \vdash t\ u \downarrow \tau'}$$

$$\frac{}{\Gamma, X : \tau \vdash X \Downarrow \tau} \quad \frac{\Gamma \vdash t \Downarrow \tau \to \tau' \quad (x \notin FV(t))}{\Gamma \vdash t\,\hat{}\,x_\tau \Downarrow \tau'} \quad \frac{\Gamma \vdash t \downarrow \delta}{\Gamma \vdash t \uparrow \delta} \quad \frac{\Gamma \vdash t \Downarrow \delta}{\Gamma \vdash t \uparrow \delta}$$

There is no way to bind a metavariable: λ binds ordinary variables only. We can only convert from an atomic typing to a normal typing at base types δ; this ensures that all the necessary η-expansions take place.

All well-formed terms in this system are ηlβn-normalized. Moreover, as for higher-order patterns generally, only β_0 reductions $(\lambda x_\tau.t)\,\hat{}\,y_\tau \to t[y_\tau/x_\tau]$ need to be performed after a substitution of higher-order patterns for metavariables.

Lemma 2 (Renaming). *Let R be one of $\downarrow, \Downarrow, \uparrow$. If $y_\tau \notin FV(t)$ and $\Gamma \vdash t\ R\ \tau$, then $\Gamma \vdash t[y_\tau/x_\tau]\ R\ \tau$, respectively.*

Lemma 3 (Substitution). *If $\Gamma, X : \tau' \vdash t \uparrow \tau$ and $\Gamma \vdash u \uparrow \tau'$ for $FV(u) = \varnothing$, then $\Gamma \vdash t[u/X] \uparrow \tau$.*

4 Nominal Patterns

We now introduce a slight variant of nominal terms that provides a closer match to higher-order patterns. This language, called *nominal patterns*, is defined by the following grammar:

$$t ::= c \mid t\ t' \mid X \mid a_\nu \mid \langle a_\nu \rangle t \mid t \,@\, a_\nu$$
$$\tau ::= \sigma \mid \sigma \to \tau \quad \sigma ::= \delta \mid \nu \mid \langle \nu \rangle \sigma$$

where as before, δ denotes a base (data) type and ν denotes a name type. As before, we assume that there is a signature assigning τ-types to constants c. Metavariables may not have arbitrary types, but only σ-types (i.e., types built using only data, name, and abstraction types). In addition, we assume that name symbols a_ν are tagged with their name types ν. As for nominal terms, we assume that the only ground terms inhabiting name-types are literal names.

The main difference between ordinary nominal terms and nominal patterns is the presence of the *concretion* operation $(-)@(-)$ that has also been considered in some versions of FreshML [12, 14]. Our type system requires well-formed nominal patterns to satisfy an analogue of the higher-order pattern restriction: in every subterm of the form $t@a$, we require that $a \# t$ holds. In order to simplify this check (and to make nominal patterns more similar to higher-order patterns), we only consider substitutions of patterns such that $FN(t) = \varnothing$, where

$$\begin{array}{ll} FN(c) = \varnothing & FN(X) = \varnothing \\ FN(a) = \{a\} & FN(t\ u) = FN(t) \cup FN(u) \\ FN(t@a) = FN(t) \cup \{a\} & FN(\langle a \rangle t) = FN(t) - \{a\} \end{array}$$

As a result, $\vDash a \# X$ and $\vDash (a\ b) \cdot X \approx X$ are valid for any names a, b and metavariable X; using these facts we can lift the freshness relation and swapping function to patterns involving metavariables.

As before, signatures Σ map constants to types and names to name types, whereas contexts map metavariables to metavariable types σ. We require patterns to be well-typed, subject to the following rules:

$$\frac{}{\Gamma \vdash a_\nu \downarrow \nu} \quad \frac{c : \tau \in \Sigma}{\Gamma \vdash c \downarrow \tau} \quad \frac{}{\Gamma, X : \sigma \vdash X \downarrow \sigma} \quad \frac{\Gamma \vdash t \downarrow \tau \to \tau' \quad \Gamma \vdash u \uparrow \tau}{\Gamma \vdash t\ u \downarrow \tau'}$$
$$\frac{\Gamma \vdash t \uparrow \sigma}{\Gamma \vdash \langle a_\nu \rangle t \uparrow \langle \nu \rangle \sigma} \quad \frac{\Gamma \vdash t \downarrow \langle \nu \rangle \sigma \quad a \notin FN(t)}{\Gamma \vdash t@a_\nu \downarrow \sigma} \quad \frac{\Gamma \vdash t \downarrow \epsilon \quad (\epsilon = \delta, \nu)}{\Gamma \vdash t \uparrow \epsilon}$$

Abstraction and concretion are construction and destruction operations for the abstraction sort. Thus, nominal patterns are subject to the following β_α- and η_α-laws:

$$\begin{array}{lll} (\beta_\alpha) & (\langle a \rangle t)@b \approx (a\ b) \cdot t & \\ (\eta_\alpha) & t \approx \langle a \rangle (t@a) & (t : \langle \nu \rangle \sigma) \end{array}$$

Note that the typing rules ensure that $b \# \langle a \rangle t$ must hold in the first case and $a \# t$ must hold in the second case. We first state some basic properties of nominal patterns.

Lemma 4 (Swapping). *Let R be one of $\downarrow, \uparrow$. If $\Gamma \vdash t\ R\ \tau$ then $\Gamma \vdash (a\ b) \cdot t\ R\ \tau$.*

Lemma 5 (Substitution). *If $\Gamma, X : \tau' \vdash t \uparrow \tau$ and $\Gamma \vdash u \uparrow \tau'$ where $FN(u) = \varnothing$, then $\Gamma \vdash t[u/X] \uparrow \tau$.*

We now show that this axiomatization satisfies the previously given laws of nominal abstraction.

Proposition 1. *For nominal patterns, we have $\langle a\rangle t \approx \langle b\rangle u$ if and only if $a \approx b, t \approx u$ or $a \# u, t \approx (a\ b) \cdot u$. Similarly, if $t : \langle\nu\rangle\tau$, then there exists a_ν and $u : \tau$ such that $t \approx \langle a\rangle u$.*

Proof. Suppose $\langle a\rangle t \approx \langle b\rangle u$. Then $a \# \langle a\rangle t \approx \langle b\rangle u$, so we have

$$t \approx (a\ a) \cdot t \approx (\langle a\rangle t)@a \approx (\langle b\rangle u)@a \approx (a\ b) \cdot u$$

There are two cases. If $a = b$ then $t \approx (a\ b) \cdot u = (a\ a) \cdot u = u$. Otherwise, $a \# u$ and $t \approx (a\ b) \cdot u$.

Now suppose $t : \langle\nu\rangle\tau$. Since $FN(t)$ is finite, we can always find a name $a \notin FN(t)$, so we can form the term $\langle a\rangle(t@a)$. By the η-rule, we have $t \approx \langle a\rangle(t@a)$, thus, a is the required name and $t@a$ the required term of type τ.

The similarity between the β_α and η_α rules for nominal patterns and the β_0 and η rules for higher-order patterns is not a coincidence. We now consider a typed translation from nominal to higher-order patterns. We assume (for convenience) that the constants, names and metavariables of nominal patterns are the same as the constants, variables, and metavariables of higher-order patterns respectively. Similarly, we assume that the name types and data types of the nominal language are base types of the higher-order language. Terms are translated as follows:

$$\begin{array}{rlrl} c^* &= c & a_\nu^* &= a_\nu \\ (t\ u)^* &= t^*\ u^* & (\langle a_\nu\rangle t)^* &= \lambda a_\nu.t^* \\ X^* &= X & (t@a_\nu)^* &= (t^*)\,\hat{}\,a_\nu \end{array}$$

The translation of types is as follows:

$$\begin{array}{rlrl} \delta^* &= \delta & (\sigma \to \tau)^* &= \sigma^* \to \tau^* \\ \nu^* &= \nu & (\langle\nu\rangle\tau)^* &= \nu \to \tau^* \end{array}$$

Contexts and signatures are translated by replacing each type with its starred form.

Example 1. The translation of $t = \langle a\rangle X @ a @ b$ is $\lambda a.X\ a\ b$, where $X : \langle\nu\rangle\langle\nu\rangle\delta$ in the former and $X : \nu \to \nu \to \delta$ in the latter.

Lemma 6. *If $a \notin FN(t)$ then $a^* \notin FV(t^*)$. Also, if $\Gamma \vdash t \uparrow \tau$ then $\Gamma^* \vdash t^* \uparrow \tau^*$.*

Theorem 1. *The translation $(-)^*$ has an inverse $(-)^\dagger$ on its range.*

Proof. Clearly $(-)^*$ is injective, and it is surjective on its range by definition.

Lemma 7. *If $\Gamma \vdash t : \tau$ is a nominal pattern and $b \notin FN(t)$, then $((a\ b) \cdot t)^* = t^*[b/a]$. Dually, if $\Gamma \vdash u : \tau$ is a higher-order pattern in the range of $(-)^*$, and $b \notin FV(t)$, then $(u[a/b])^\dagger = (a\ b) \cdot u^\dagger$.*

Proof. Proof is by induction on the structure of t. If $t = a$, then $((a\ b) \cdot t)^* = b^* = b$ and $t^*[b/a] = a[b/a] = b$. Otherwise, t is a name other than a or b, and swapping, substitution, and the $(-)^*$ translation all fix t. The case for t a constant or metavariable is similar. For $t = t_1\ t_2$, the induction step is straightforward. This leaves the case of abstraction. If $t = \langle a\rangle u$, then $b \# u$ so by induction we have $((a\ b) \cdot u)^* = u^*[b/a]$, hence

$$\begin{aligned} ((a\ b) \cdot \langle a\rangle u)^* &= (\langle b\rangle(a\ b) \cdot u)^* = \lambda b.((a\ b) \cdot u)^* = \lambda b.u^*[b/a] \\ &\approx_\alpha \lambda a.u^* = (\lambda a.u^*)[b/a] = (\langle a\rangle u)^*[b/a] \end{aligned}$$

If $t = \langle b\rangle u$, then the induction hypothesis does not apply directly, but we can choose a fresh name $b' \# a, b, t$ such that

$$\begin{aligned} &((a\ b) \cdot \langle b\rangle t)^* \ \approx\ ((a\ b) \cdot \langle b'\rangle(b\ b') \cdot t)^* \ =\ (\langle b'\rangle(a\ b) \cdot (b\ b') \cdot t)^* \\ &\quad =\ \lambda b'.((b\ b') \cdot t)^*[b/a] \ =\ \lambda b'.t^*[b'/b][b/a] \ \approx_\alpha\ \lambda b.t^*[b/a] \ =\ (\langle b\rangle t)^*[b/a] \end{aligned}$$

where the two middle steps rely on the facts that $b \notin FN((b\ b') \cdot t)$ and $b' \notin FN(t)$. The case for $t = \langle a'\rangle t$ for $a' \neq a, b$ is straightforward.

The second part follows immediately from the first by setting $t = u^\dagger$.

Theorem 2. *Let $t, u : \tau$ be nominal patterns. Then $t \approx u$ if and only if $t^* \approx u^*$.*

Proof. Proof is by induction on the derivation of $t \approx u$ in the forward direction. The interesting cases are for β_α and η_α rules. While η_α is straightforward, for β_α we have $(\langle a\rangle t)@b \approx (a\ b) \cdot t$ and want to show that $(\lambda a.t^*)\ b \approx ((a\ b) \cdot t)^*$. By β_0 and the previous lemma we have $(\lambda a.t^*)\ b \approx t^*[b/a] = ((a\ b) \cdot t)^*$.

The reverse direction is similar, except that we need to use the identity $(t[a/b])^\dagger = (a\ b) \cdot t^\dagger$ in the β_0 case.

Corollary 1. *$t \approx u$ is satisfiable if and only if $t^* \approx u^*$ is; moreover, the satisfying valuations θ, θ^* are in bijective correspondence via $(-)^*$.*

This shows that nominal pattern unification coincides with a special case of higher-order pattern unification: specifically, the case for terms in which the only form of binding is λ-abstraction over void base types ν. In fact, many applications of higher-order patterns are possible within this fragment: it is commonplace to use an abstract or empty type for the "type of variable names" in, for example, a higher-order abstract syntax encoding of the π-calculus [10]. However, applications involving λ-abstraction over non-void types are also common [7].

This translation is interesting, but we have only shown that there is a correspondence between two very limited special cases of the two problems. Next we show how to translate full higher-order pattern unification to nominal pattern unification.

5 Higher-order pattern unification as nominal pattern unification

In higher-order patterns, λ-term variables are not limited to a collection of void base types, but may be of any type, so variables may be applied to argument lists including repeated variables, metavariables, or more general terms (that is, the pattern restriction is not required of argument lists whose head is not a variable). This permits the formation of terms such as $\lambda x, y.y\ (\lambda z.Fz)\ x\ x$ which are not in the range of $(-)^*$; i.e., which do not correspond to a nominal pattern. Such terms are not in the domain of $(-)^\dagger$, so the approach investigated in the last section does not apply.

However, there is another translation that works. The reason the idea of the previous section doesn't work is that in higher-order patterns, variables play one of two roles: they can be passed as arguments to metavariables, but they can also act as functions on lists of arguments. The latter role is not supported directly by nominal patterns, because name types ν are populated only by names.

Given a higher-order language L, we construct a nominal language L^{**} possessing a name-type ν_τ for each simple type τ of L and a data type δ for each basic type δ of L. We define a translation on L-types as follows:

$$\delta^{**} = \delta \quad (\tau_1 \to \tau_2)^{**} = \langle \nu_{\tau_1} \rangle {\tau_2}^{**}$$

Note that each τ-type of L translates to a σ-type of L^{**}. Given a signature Σ, we write Σ^{**} for the result of replacing all the types in Σ with their $(-)^{**}$ translations; similarly for contexts Γ^{**}. Moreover, we add the following new constants to L^{**}:

$$\begin{aligned} var_\tau &: \nu_\tau \to \tau^{**} \\ app_{\tau_1\tau_2} &: (\tau_1 \to \tau_2)^{**} \to {\tau_1}^{**} \to {\tau_2}^{**} \end{aligned}$$

This signature is infinite, since the function symbols var_τ and $app_{\tau_1\tau_2}$ are indexed with types. However, in any particular situation, only finitely many ν_τ types and finitely many constants of the above signature need to be considered. After unwinding definitions, the type of $app_{\tau_1\tau_2}$ is $\langle \nu_{\tau_1} \rangle {\tau_2}^{**} \to {\tau_1}^{**} \to {\tau_2}^{**}$. The types of these constants are legal τ-types in L^{**}.

Intuitively, ν_τ is the type of *names of variables of type* τ, and var "casts" a ν_τ to its value, simulating the evaluation of a variable at the head of an application in a higher-order term. Similarly, app simulates application: given an abstraction $\langle \nu_{\tau_1} \rangle {\tau_2}^{**}$ and a translated term of type ${\tau_1}^{**}$, application produces a term of type ${\tau_2}^{**}$.

The idea of the translation is to use the var, app, and lam constructors to represent ground λ-term structure, and use names, abstraction and concretion to represent subterms involving metavariables. In this translation, we assume that λ-calculus variables are the same kinds of symbols as names in nominal patterns.

$$\begin{aligned} c^{**} &= c & (t\ u)^{**} &= app(t^{**}, u^{**}) \\ {x_\tau}^{**} &= var(x_{\nu_\tau}) & X^{**} &= X \\ (\lambda x.t)^{**} &= \langle x \rangle t^{**} & (t\,\hat{}\,a)^{**} &= t^{**} \mathbin{@} a \end{aligned}$$

Example 2. Note that variable occurrences are treated differently depending on context: variables on the left-hand side of a flexible application $(-)\,\hat{}\,(-)$ are left alone, while others are encapsulated in a $var(-)$-constructor which casts a variable name of type ν_τ to an expression of type τ^{**}. Thus, the translation of $\lambda x, y.c\ (F\,\hat{}\,x\,\hat{}\,y)$ is $\langle x\rangle\langle y\rangle app(var(c), F \mathbin{@} x \mathbin{@} y)$, where $F : \tau_1 \to \tau_2 \to \delta$ in the former is mapped to $F : \langle \nu_{\tau_1}\rangle\langle \nu_{\tau_2}\rangle\delta$ in the latter.

The translation preserves well-formedness and is invertible; these facts are easy to show by induction.

Proposition 2. *If $x \notin FV(t)$ then $x^{**} \notin FN(t^{**})$. If $\Gamma \vdash t \uparrow \tau$ where t is a higher-order pattern, then $\Gamma^{**} \vdash t^{**} \uparrow \tau^{**}$. Similarly, if $\Gamma \vdash t \downarrow \tau$ or $\Gamma \vdash t \Downarrow \tau$, then $\Gamma^{**} \vdash t^{**} \downarrow \tau^{**}$. Also, the translation has an inverse $(-)^{\dagger\dagger}$.*

As observed by Miller, in a ηlβn higher-order pattern unification problem, the only kinds of redices that occur are β_0 redices. Since the $\beta_0\eta$ theory is simulated by the $\beta_\alpha\eta_\alpha$ theory in nominal patterns, higher-order pattern unification is equivalent to nominal pattern unification.

Theorem 3. *A higher-order pattern unification problem $t \approx ?\ u$ in ηlβn-normal form has a solution if and only if its translation $t^{**} \approx ?\ u^{**}$ has a nominal pattern unifier.*

Proof. For the forward direction, suppose that t, u are normalized and have a higher-order pattern unifier θ, so that $\theta(t) \approx \theta(u)$ up to ηlβn-normalization. Moreover, this normalization process can only involve β_0-redices, because there are no metavariables of extensional function types in t, u (as argued above). Let $\theta^{**} = [X := \theta(X)^{**} \mid X \in Dom(\theta)]$ be the translation of θ. Following a similar argument to the one used in Theorem 2, β_0-normalization can be simulated in the nominal pattern calculus via β_α-normalization. Thus, θ^{**} is a nominal pattern unifier of $t^{**} \approx ?\ u^{**}$.

The reverse direction is similar. Since only β_α-redices can be introduced in a nominal pattern unifier θ for $t^{**} \approx ?\ u^{**}$, we can use the reverse translation $(-)^{\dagger\dagger}$ to translate θ to a higher-order pattern unifier $\theta^{\dagger\dagger}$.

6 Nominal Pattern Unification as Nominal Unification

In this section, we show how to reduce nominal pattern unification to nominal unification. This is not as trivial as it sounds, for nominal patterns include the concretion operation not found in ordinary nominal terms, and so nominal pattern unification is not an immediate special case of nominal unification. In addition, nominal unification permits metavariables to be instantiated with terms containing free names, whereas nominal pattern unifiers must be closed.

We deal with the second problem first. Given a problem P with metavariables $\overline{X}$ and names $\overline{a}$, let $\#(\overline{a}, \overline{X}) = \{a \# X \mid a \in \overline{a}, X \in \overline{X}\}$. This set of constraints ensures that no name mentioned in P can appear free in any substitution for P's metavariables. Moreover, the nominal unifiers produced by Urban et al.'s algorithm only involve the names mentioned in the original problem.

Concretion can be eliminated from nominal pattern unification problems as follows. If $t \mathbin{@} a$ is a subterm of a nominal pattern unification problem $P[t \mathbin{@} a]$, then that problem is equivalent to the problem $P[Y], \langle a\rangle Y \approx ?\ t$, where Y is a fresh metavariable. This is because we know that a must be fresh for t (because of the well-formedness constraint) and so by the η-rule, we know that t can always be expressed as $\langle a \rangle Y$ for some value Y; this is precisely the value denoted by $t @ a$.

Given a nominal pattern unification problem P over names $\overline{a}$ and variables $\overline{X}$, we write $P^{\#}$ for the result of eliminating concretions from P and adding the freshness constraints $\#(\overline{a}, \overline{X})$. We claim that $P^{\#}$ and P are equivalent problems, and in addition that the answer to P can be computed from that of $P^{\#}$. However, the final step in this process is complicated, so we will illustrate it via examples first.

Example 3. Consider the problem $\langle a \rangle X \approx ? \langle a \rangle Y @ a$. In this case the translation is

$$\#(\{a\}, \{X, Y\}), Y \approx \langle a \rangle Y', \langle a \rangle X \approx \langle a \rangle Y'$$

The nominal unifier is $a \# X, Y = \langle a \rangle X$. Since $a \# X$, and a is the only name in scope, $Y = \langle a \rangle X$ is a nominal pattern unifier for the original problem.

Example 4. The translation of the problem $\langle a \rangle \langle b \rangle X @ a @ b \approx ? \langle a \rangle \langle c \rangle Y @ c @ a$ is (after some trivial simplifications)

$$\#(\{a, b\}, \{X, Y\}), X \approx \langle a \rangle \langle b \rangle X', Y \approx \langle c \rangle \langle a \rangle Y', \langle a \rangle \langle b \rangle X' \approx \langle a \rangle \langle c \rangle Y'$$

The most general unifier is $b \# Y', X \approx \langle a \rangle \langle b \rangle (b\ c) \cdot Y', Y \approx \langle a \rangle \langle c \rangle Y'$. Since $b \# Y'$, we know that Y' can only depend on a and c, so there must be a Z such that $Y' = Z @ a @ c$, where Z is a nominal pattern metavariable (i.e., can be substituted only with closed patterns). Solving for X, Y in terms of Z, we obtain $X = \langle a \rangle \langle b \rangle (a\ b) \cdot (Z @ a @ c) = \langle a \rangle \langle b \rangle Z @ a @ b$, $Y = \langle a \rangle \langle c \rangle Z @ a @ c$; this is a nominal pattern unifier of the original problem.

Remark 1. There is a minor hitch in this argument, due to the fact that there may be types that have no closed terms. For example, if data type δ consists only of terms of the form $v(a)$ for names a_ν and $v : \nu \to \delta$, then the unification problem $X \approx ? X$, where $X : \delta$, has no solution among closed terms, but it does have a nominal unifier.

This is similar to the difficulty in ordinary (typed) unification in the presence of possibly-void types. It is customary to either ignore this problem or assume that all types have at least one (closed) term. In our case, it is decidable (for finite signatures) whether each σ-type possesses any closed terms. We call such types *nonvoid*. For example, ν and δ (where δ is as in the previous paragraph) obviously possesses no closed terms, while $\langle \nu \rangle \nu$ and $\langle \nu \rangle \delta$ are nonvoid. Moreover, a substitution θ is called nonvoid if all the metavariables mentioned in its range are nonvoid.

Theorem 4. *If P is satisfiable then its translation $P^{\#}$ is satisfiable. Furthermore, if $P^{\#}$ is satisfiable then its unifier can be translated to a substitution which unifies P if and only if it is nonvoid.*

Proof. For the forward direction, clearly if θ satisfies P then θ satisfies each constraint in $\#(\overline{a}, \overline{X})$. In addition, it is easy to show that $\#(\overline{a}, \overline{X}), P[t @ a]$

is satisfiable if and only if $\#(\overline{a}, \overline{X}), P[Y], \langle a \rangle Y \approx ?\ t$ is satisfiable; thus, by induction if P is satisfiable then so is $P^{\#}$.

For the reverse direction, suppose ∇, θ is the most general nominal unifier for $P^{\#}$. Let $\overline{a}$ be the names of P. Suppose that the free variables in ∇, θ are $\overline{Y}$. For each $Y \in \overline{Y}$, there is a list of names $\overline{a_{Y_i}}$ such that $a \in \overline{A}$ but $a \# Y_i \notin \nabla$. Thus, we have $Y_i = Z_i \mathbin{@} \overline{a_{Y_i}}$ for some fresh metavariables Z_i such that $\overline{a} \# Z_i$. If we make this substitution, then we obtain a nominal pattern possibly involving swappings, but these swappings can be eliminated since each Z_i satisfies $(a_i\ a_j) \cdot Z \approx Z$. This produces the desired substitution θ'. If θ' is nonvoid, i.e. each σ_i is nonvoid for $Z_i : \sigma_i$, then each Z_i can be replaced with a closed term 0_{σ_i} to obtain a satisfying valuation for P. Conversely, it is not difficult to show that if θ' is not nonvoid, then P is unsatisfiable, since (by the first part) any valuation satisfying P can be used to construct a valuation satisfying $P^{\#}$, which would have to be an instance of ∇, θ because it is most general. Closed instantiations of all the Z_i could be extracted from such a valuation.

7 Related work

As discussed by Urban et al. [15], nominal unification can apparently be reduced to higher-order pattern unification, but it is difficult to see how to translate the resulting higher-order pattern unifier to a nominal unifier. Nevertheless, such a translation is of interest because if nominal unifiers can be extracted from higher-order pattern unifiers in linear time, this would give a linear algorithm for nominal unification. We believe that it would be equivalent (and notationally simpler) to investigate the reduction of full nominal unification to nominal pattern unification.

Miller [8] showed that higher-order unification problems (and higher-order logic programs) can be translated to logic programs in L_λ [7], a logic programming language based on higher-order pattern unification. This reduction takes advantage of hereditary Harrop goals and clauses featured in L_λ. We believe that a similar reduction could be performed in a nominal logic programming language that provides hereditary Harrop goals and program clauses. While such features are present in the current implementation of the nominal logic programming language αProlog, the semantics of goals of the form $\forall x.G$ and $D \supset G$ have not been studied carefully yet for nominal logic programming. This question, and more generally, the question of whether L_λ programs can be translated to nominal logic programs (or vice versa) is of interest.

Hamana [4] has investigated the problem of unification modulo the β_0-rule for *binding algebra terms* [2]. Such terms are similar to nominal or higher-order patterns except that the lists of names supplied to metavariables may include repeated names. As a result, unification appears to require some searching for suitable renamings, and most general unifiers appear not to be unique. Obviously, this is a special case of higher-order unification, but it appears to be at worst of nondeterministic polynomial time complexity (since one can guess a sequence of appropriate renamings to find a unifier in polynomial time). We are interested in seeing whether this form of unification can also be implemented via nominal logic programming using Miller's approach.

Finally, we are interested in combining nominal and higher-order unification, or more generally, developing *nominal equational unification* techniques that include higher-order unification, Hamana's β_0-unification, structural equivalence in the π-calculus, and other equational theories involving name-binding as special cases. We believe that nominal equational unification techniques would be extremely useful for programming, prototyping, and formalizing programming languages, logics, and type systems.

8 Conclusion

We have shown that higher-order pattern unification can be reduced to nominal unification via an intermediate language of *nominal patterns*. This shows that any computation that can be performed using higher-order pattern unification can also be performed using nominal unification. It also shows that higher-order patterns are not just an ad-hoc invention of interest for efficiency reasons, but that they can be given formal status using nominal logic: in particular, semantic models of binding syntax for higher-order patterns can be constructed using the same techniques as for nominal logic.

Previous work has been focused on determining whether nominal unification is really "new" (that is, whether it is trivially reducible to higher-order pattern matching). We agree with Urban et al. that while it may not be new, there are good reasons for studying nominal unification directly rather than through the lens of higher-order unification. Moreover, our experience has been that nominal unification is much closer to first-order unification and considerably simpler to explain and implement than higher-order pattern unification (compare Urban et al. [15] to treatments such as Miller [7, 8], Nipkow [11], Dowek et al. [1], or Hamana [5]). This is not meant as a criticism of these works! Instead, our point is that even if one does *not* believe that nominal techniques are worth investigating as an *alternative* to higher-order abstract syntax, we believe that they are of value as an aid to *understanding* higher-order abstract syntax, particularly higher-order patterns.

References

1. G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. Technical Report Rapport de Recherche 3591, INRIA, December 1998.
2. M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In Giuseppe Longo, editor, *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, pages 193–202, Washington, DC, 1999. IEEE, IEEE Press.
3. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
4. Makoto Hamana. A logic programming language based on binding algebras. In *Proc. Theoretical Aspects of Computer Science (TACS 2001)*, number 2215 in Lecture Notes in Computer Science, pages 243–262. Springer-Verlag, 2001.
5. Makoto Hamana. Simple β_0-unification for terms with context holes. In *Proceedings of the 16th International Workshop on Unification (UNIF 2002)*, pages 9–13, 2002.

6. Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–67, 1975.
7. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, 1(4):497–536, 1991.
8. Dale Miller. Unification of simply typed lambda-terms as logic programming. In Koichi Furukawa, editor, *Logic Programming, Proceedings of the Eighth International Conference*, pages 255–269, Paris, France, June 24–28 1991. MIT Press.
9. Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.
10. Dale Miller and Alwen Tiu. A proof theory for generic judgments: extended abstract. In *Proc. 18th Symp. on Logic in Computer Science (LICS 2003)*, pages 118–127. IEEE Press, 2003.
11. Tobias Nipkow. Functional unification of higher-order patterns. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 64–74, 1993.
12. A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Proc. 5th Int. Conf. on Mathematics of Programme Construction (MPC2000)*, number 1837 in Lecture Notes in Computer Science, pages 230–255, Ponte de Lima, Portugal, July 2000. Springer-Verlag.
13. Zhenyu Qian. Linear unification of higher-order patterns. In *Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 391–405. Springer-Verlag, 1993.
14. M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programmming with binders made simple. In *Proc. 8th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2003)*, pages 263–274, Uppsala, Sweden, 2003. ACM Press.
15. C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1–3):473–497, 2004.

Efficiently Computable Classes of Second Order Predicate Schema Matching Problems

Masateru HARAO[1], Shuping YIN[2], Keizo YAMADA[1], and Kouichi HIRATA[1]

[1] Department of Artificial Intelligence Kyushu Institute of Technology
Kawazu 680-4, Iizuka 820-8502, Japan
[2] Graduate School of Computer Science and Systems Engineering
{harao,yin,yamada,hirata}@ai.kyutech.ac.jp *

Abstract. Second order predicate schema matching is concerned with finding the matchers of a given pair such that $\langle \Phi, \phi \rangle$ where Φ is a formula containing second-order predicate variables and ϕ is a first order logical formula. In general, this problem is intractable even if we impose some strict syntactic restrictions. In this paper, we propose a class of second order predicate schemas which is defined by introducing syntax free variables and show that the computational complexity of the schema matching is almost linear on input size. We also show that this class possesses enough expressive power to implement a schema-guided automatic theorem prover.

1 Introduction

A *schema* is a template of formulas and knowledge processing which uses schemas as guiding information is called *schema guided knowledge processing* [6]. Especially, processing based on second-order schemas has been studied in research areas such that *program transformations* [12], *automatic program syntheses* [6], *analogical reasoning* [2, 4, 7], and so on. Here, second order matching has an important role in implementing these systems.

A second order predicate schema is a formula including second-order predicate variables, and a second order predicate schema matching is a problem of obtaining matchers of any pair such that $\langle \Phi, \phi \rangle$ where Φ is a second order predicate schema and ϕ is a closed first order formula. This predicate schema matching can be regarded as a special case of the *second-order matching* and is known to be intractable in general [16].

Though the *second-order matching* is intractable in general [1], a sharp characterization between tractable and intractable second-order matching has been given from syntactical viewpoint [10]. Especially, a class of second order terms called *higher order pattern* in which matchers can be derived in linear time has been reported [14]. Furthermore, it has been shown that a matching algorithm which works more efficient than the standard one [11, 12] can be designed by using pre-checking method [3]. These results indicate that the efficiency of second order matchings strongly depends on the existence of bound and free variables and can be improved by devising the strategy of the procedure.

* This work is partially supported by Grand-in-Aid for Scientific Research 13558036 from the Ministry of Education, Culture, Sports, Science and Technology, Japan, and Foundation for promotion of researches on artificial intelligence

In this paper, we propose a class of second order predicate schemas in which an efficient schema matching exists from the motivation of constructing a schema-guided theorem prover [17]. Firstly, we define the syntax of second order schemas which we propose here by introducing syntax free variables. Next, we design a general schema matching algorithm based on the projection position indexing. Furthermore, we study the heuristics to improve its computational efficiency. In particular, we present an algorithm which derives matchers efficiently by introducing strategy. Finally, we introduce a class of predicate schema matching problems in which a unique matcher can be derived almost in linear time. We also show that this class possesses enough expressive power to implement a schema-guided automatic theorem prover.

2 Preliminaries

We use a term language L to represent second order language [10] instead of the general language defined by using λ-notation. The term language L is defined using the following symbol sets.

(1) IC :a set of *individual constants* $\{a, b, c, \ldots\}$
(2) IV :a set of *individual variables* $\{x, y, z, \ldots\}$,
(3) FC :a set of *function constants* $\{f, g, h, \ldots\}$,
(4) FV :a set of *function variables* $\{F, G, H, \ldots\}$,
(5) PC :a set of *predicate constants* $\{p, q, r, \ldots\}$,
(6) PV :a set of *predicate variables* $\{P, Q, R, \ldots\}$,
(7) ISV: a set of *individual syntax variables* $\{x_c, y_c, z_c, \ldots\}$,
(8) FSV: a set of *function syntax variables* $\{F_c, G_c, H_c, \ldots\}$,

Throughout of this paper, we assume a set of elementary types containing the Boolean type o. Furthermore, we assume that each $d \in IC \cup IV \cup FC \cup PC \cup PV \cup ISV \cup FSV$ has a type denoted by $\tau(d)$. Each $d \in IC \cup IV \cup ISV$ has an elementary type not equal to o, each $d \in FC \cup FV \cup FSV$ has a type $\mu_1 \times \cdots \times \mu_n \to \mu$ where neither μ_i nor μ is o, and each $d \in PC \cup PV$ has a type $\mu_1 \times \cdots \times \mu_n \to o$. Especially, we deal with the logical connectives $\wedge, \vee, \supset$ as predicate constants satisfying that $\tau(\wedge) = \tau(\vee) = \tau(\supset) = o \times o \to o$ and $\tau(\neg) = o \to o$. For a quantifier Q ($Q \in \{\forall, \exists\}$) we also treat $Qx.$ as a predicate constant satisfying that $\tau(Qx.) = o \to o$. If $\tau(\varphi) = o$ and $\tau(x) \neq o$, then $Qx.\varphi$ has the type o.

Typed terms are defined as usual [1]. Here, we assume that each term contains no λ *abstraction*. A variable $x \in IV$ is called *bound* if x appears in the scope of a quantifier $Qx(Q \in \{\forall, \exists\})$ and *free* otherwise. A formula is called *closed* if it contains no free individual variables. A *second order predicate schema* or simply *predicate schema* is a closed formula which contains predicate variables but contains no function variables. Note that a closed predicate schema may contain some syntax free variables $x_c, F_c, \ldots$. In the following, we denote schemas and formulas by $\Phi, \Psi, \ldots$ and $\phi, \varphi, \psi, \ldots$, respectively. Examples of a predicate schema Φ and a closed first order formulas ϕ are given next.

$$\Phi = P(x_c) \wedge \forall x.(P(x) \supset P(F_c(x))) \supset P(F_c(F_c(c))),$$
$$\phi = \forall z.p(z, 0) \wedge \forall x.(\forall z.p(z, x) \supset \forall z.p(z, f(x))) \supset \forall z.p(z, f(f(0)))$$

In this Φ, syntax variables x_c and F_c denote constants from the logical viewpoint, but they mean indefinite constants. From this reason, we treat them as free variables which range over IC and FC respectively. By treating them as syntax free variables, we can raise the expressive power of schemas. A *head* of a schema Φ is a left-most symbol of Φ in its prefix expression and is denoted by hd(Φ). In case of above Φ, $hd(\Phi) = " \supset "$.

Let $\mathcal{V} = \{IV \cup FV \cup ISV \cup FSV\}$. A *substitution* θ is a function from $\mathcal{V}$ to the set of all terms such that $\theta(v) \neq v$ holds only for finitely many $v \in \mathcal{V}$. For terms $t_i (1 \leq i \leq m)$ and variables $v_i (1 \leq i \leq m)$, a substitution such that $\theta(v_i) = t_i$ is denoted by $\theta = [v_1 := t_1, \ldots, v_m := t_m]$. Intuitively, $t\theta$ denotes the term obtained by replacing a variable v_i in t with t_i simultaneously under the renaming, and is the same operation to $(\lambda v_1 \cdots v_n.t)t_1 \cdots t_n$ in λ-calculus.

A set of substitutions $\{\theta_1, \ldots, \theta_m\}$ is *consistent* if $(\theta_1 \cup \ldots \cup \theta_m)$ is well defined. For example, if $\theta_1 = [x := a]$ and $\theta_2 = [x := b]$, then $\{\theta_1, \theta_2\}$ is not consistent. $\theta \in \{[H := t] \mid t \in S_{\sqcap E}\}$ is a matcher of E if it is consistent.

We denote an m-tuple of terms $t_1, \cdots, t_m$ by $\overline{t_m}$. For a term t and a substitution θ, $t\theta$ is defined inductively as follows:

(1) If $t = c (c \in IC)$, then $t\theta = c$.
(2) If $t = x (x \in IV)$, and $[x := t'] \in \theta$, then $t\theta = t'$; otherwise $t\theta = x$.
(3) If $t = x_c (x_c \in ISV)$, and $[x_c := c] \in \theta (c \in IC)$, then $t\theta = c$; otherwise $t\theta = x_c$.
(4) If $t = f(\overline{t_n}) (f \in FC \cup PC)$, then $t\theta = f(\overline{t_n\theta})$.
(5) If $t = P(\overline{t_n}) (P \in PV \cup FV)$ and $[P := \lambda\overline{v_n}.t'] \in \theta$, then $t\theta = t'[v_1 := t_1\theta, \ldots, v_n := t_n\theta]$; otherwise $t\theta = P(\overline{t_n\theta})$.
(6) If $t = F_c(\overline{t_n}) (F_c \in FSV)$ and $[F_c := \lambda\overline{v_n}.f(\overline{v_n})] \in \theta$, then $t\theta = f(\overline{t_n\theta})$.
(7) If $t = Qx.t'$ and $Q \in \{\forall, \exists\}$, then $t\theta = Qy.((t'[x := y])\theta)$, where y is a new variable.

Example 1. Let Φ be the schema stated above, and let θ be a substitution such that $\theta = [P := \lambda u.\forall z.p(z,u), x_c := 0, F_f := \lambda v.f(v)]$. Then we have:

$$\Phi\theta = P(x_c)\theta \wedge (\forall x.(P(x) \supset P(F_c(x)))\theta \supset P(F_c(F_c(x)))\theta = \phi. \qquad \Box$$

A finite set of pairs of schemas and formulas is called an *expression*. An expression of the form $\{\langle P_i(t_1^i, \ldots, t_{n_i}^i), \varphi_i \rangle \mid i \in N\}$ is called a *reduced expression*, where P_i is a predicate variable and φ_i is a formula. The *size* of a term t is the number of occurrences of all symbols in t and is denoted by $|t|$. For an expression $E = \{\langle \Phi_i, \varphi_i \rangle \mid i \in N\}$, the *size* of E is defined by $\sum_{i \in N}(|\Phi_i| + |\varphi_i|)$ and is denoted by $|E|$. A substitution θ such that $\Phi_i\theta = \varphi_i$ for all $i \in N$ is called a *matcher* of E. The *schema matching* for E is a procedure to find a matcher of E. If there is a matcher of E, then E is called *matchable*.

3 Projection Position Indexing for Schema Matching

We define rules for our schema matching by modifying the ones of [12]. Let E be an expression and let $\langle s, t \rangle \in E$. Assume that $hd(s) = @$, $hd(t) = \natural$. In the following, we denote the transformation using a rule "*rule*" by " $\Rightarrow_{rule}$ ".

Rules for second-order matching:
$E{=}E' \cup \{\langle s,t\rangle\}$, $s{=}@(\overline{s_r})$, $t{=}\natural(\overline{t_d})$

<u>*Simp*</u> (Simplification rule):
(1) If $s = t$ and $s \in IC$ or $s = w$: $E{=}E' \cup \{\langle s,t\rangle\} \Rightarrow_{Simp} E'$
(2) If $hd(s) = hd(t) = @$ and $@ \in (IC \cup FC \cup PC \cup \{\vee, \wedge, \supset, \neg\})$:
$E{=}E' \cup \{\langle @(\overline{s_m}), \natural(\overline{t_m})\rangle\} \Rightarrow_{Simp} \ E' \cup \{\langle s_1,t_1\rangle, \cdots, \langle s_m,t_m\rangle\}$
(3) If $s = Qx.\Phi$, $\ t = Qy.\varphi, (Q \in \{\forall, \exists\})$:
$E{=}E' \cup \{\langle s,t\rangle\} \Rightarrow_{Simp} E' \cup \{\langle \Phi[x := w], \varphi[y := w]\rangle\}$

<u>*Imit*</u> (Imitation rule):
(1) If $t{=}Qx.\varphi(\overline{t_d})$ $(Q \in \{\forall, \exists\})$ and $\tau(s){=}\tau(Qx.\varphi(\overline{t_d})){=}o$:
$E \ \Rightarrow_{Imit} E[@ := \lambda\overline{v_r}.Qx.\varphi(H_1(x,\overline{v_r}), \ldots, H_d(x,\overline{v_r})]$,
where $H_i \in FV (1 \le i \le d)$.
(2) If $hd(s){=}x (x \in IV)$ and t contains no bound variables:
$E \ \Rightarrow_{Imit} E[x := t]$
(3) If $hd(s){=}x_c (x_c \in ISV)$ and $t \in IC$:
$E \ \Rightarrow_{Imit} E[x := t]$
(4) If $hd(s){=}F_c (F_c \in FSV)$ and $hd(t) \in FC$:
$E \ \Rightarrow_{Imit} E[x := hd(t)]$
(5) If $hd(s) \in FV$ and $t{=}x (x \in IV)$: $E \ \Rightarrow_{Imit} E[@ := \lambda\overline{v_r}.x]$
(6) If $hd(s) = @(@ \in PV \cup FV)$ and $hd(t) = \natural(\natural \in IC \cup FC \cup PC)$:
$E \ \Rightarrow_{Imit} \ E[@ := \lambda\overline{v_r}.\natural(H_1(\overline{v_r}), \cdots, H_d(\overline{v_r}))]$.
(7) If $hd(s) = @(@ \in FSV)$ and $hd(t) = \natural(\natural \in FC)$, $r = d$:
$E \ \Rightarrow_{Imit} \ \ E[@ := \lambda\overline{v_r}.\natural(v_1, \cdots, v_r)]$.

<u>*Proj*</u> (Projection rule), where $\bar{s_r} = (s_1, \ldots, s_r)$.
If $\tau(s_i) = \tau(t)(1 \le i \le r)$: $E \ \Rightarrow_{Proj} \ E[@ := \lambda\overline{v_r}.v_i]$

The rules <u>*Simp*</u> decompose a given E into reduced form E'. Note that <u>Simp</u> (3) replaces quantified variables $x, y, \ldots$ with different symbols $w_1, w_2, \ldots$ to denote bound variables in schemas explicitly after the quantifiers are deleted. In the following, we also call them *bound variables*. For example, for an expression

$$\langle \forall x.(P(x) \supset P(F_c(x))), \forall x.(\forall z.p(z,x) \supset \forall z.p(z, f(x)))\rangle,$$

we have the following reduced expression by applying <u>Simp</u>(3):

$$\{\langle P(w), \forall z.p(z,w)\rangle, \langle P(F_c(w)), \forall z.p(z, f(w))\rangle\}.$$

A rule of <u>*Imit*</u> imitates $hd(t)$ of the target formula t. The rule <u>*Imit*</u> (1) is the newly introduced rule for the predicate matching. A predicate variable Φ imitates also formulas having $\forall$ and $\exists$ as heads such that $\underline{\forall x}.p(\ldots, \underline{x}, \ldots)$ and $\underline{\exists x}.p(\ldots, \underline{x}, \ldots)$ by treating quantifiers as constants. There, the bound variable x has to appear on the inside of p since it makes sense only by the pair. For example, let $E = \{\langle \Phi(a), \forall x.p(x)\rangle\}$, then the substitution $[\Phi := \lambda u.\forall x.\Psi(x,u)]$ is available, where $\forall x.\Psi(x,u)$ expresses any formulas quantified with $\forall x$. Then, we have $\forall x.p(H(x,a))$ by applying $[\Psi := \lambda v_1 v_2.p(H(v_1,v_2))]$. By combining these transformations into one operation, the rule <u>*Imit*</u> (1) is defined. Here, each $H_i \in FV$ is a second order function variable introduced newly, and is called *fresh schema variable.*

Note that *Imit* cannot be applied for any pair such that $\langle s, w\rangle$, where w is a bound variables (see *Imit* (2)). For syntax free variables, the rule *Imit*(3)(4) are used, where each syntax variable can be substituted with a symbol. For example, let $\langle H(x_c), f(g(a))\rangle$. Then we have $[H := \lambda v.f(g(v)), x_c := a]$, but $[H := \lambda v.f(v), x_c := g(a)]$ is not available. Similarly, $\langle F_c(x_c), f(g(a))\rangle$ is not matchable.

A rule of *Proj* simulates a subterm of the target formula t by projecting an argument s_i of $s (1 \leq i \leq r)$.

For an expression E, we have an expression in the form $E' = \{\langle P_i(s^i_1, \ldots, s^i_{r_i}), \varphi_i\rangle \mid i \in N\}$ by applying Simp rules to E repeatedly. We call this E' *reduced form* of E and this process *pre-processing*. Let $\Rightarrow^*$ denote finitely many applications of the rules *Simp*,*Imit*,*Proj*. Then the following theorem holds in a similar way to [12]:

Theorem 1. *Let E be an expression and E' be its reduced form. Then E is matchable if and only if $E' \Rightarrow^* \emptyset$, and matchers of E are given as the composition of all substitutions derived in the transformation from E' to $\emptyset$.* □

Let E be a reduced form $\{\langle P_i(s^i_1, \ldots, s^i_{r_i}), \varphi_i\rangle \mid i \in N\}$. Then, our schema matching applies the *projection position indexing* to E. Assume that s^j_i is a term which contains no fresh schema variables but may contain syntax free variables, and t is a term which contains no free variables. By $s \doteq t$, we denote that s and t are matchable. For example, $F_c(x_c) \doteq f(a)$, since a matcher $[F_c := \lambda u.f(u), x_c := a]$ exists.

Definition 1. *Let s be a schema $P(s_1, \ldots, s_r)$, t a formula, $\rho(t)$ a set $\{i \mid t \doteq s_i, 1 \leq i \leq r\}$. Then, an indexed term $J(s, t)$ is defined inductively as follows:*

(1) $J(s, w) = *^{\rho(w)}$, $J(s, c) = c^{\rho(c)}$ $(c \in IC)$, and $J(s, x) = x^{\emptyset}$ $(x \in IV)$.
(2) If $t = f(t_1, \ldots, t_m)$, $f \in FC$ and $J(s, t_i)$ is an indexed term of t_i for s $(1 \leq i \leq m)$, then $J(s, t) = f^{\rho(t)}(J(s, t_1), \ldots, J(s, t_m))$.
(3) If $t = p(t_1, \ldots, t_m)$, $p \in PC \cup \{\neg, \wedge, \vee, \supset\}$ and $J(s, t_i)$ is an indexed term of t_i for s $(1 \leq i \leq m)$, then $J(s, t) = p^{\emptyset}(J(s, t_1), \ldots, J(s, t_m))$.
(4) If $t = Qx.t_1$ $(Q \in \{\forall, \exists\})$ and $J(s, t_1)$ is an indexed term of t_1 for s, then $J(s, t) = Qx^{\emptyset}.J(s, t_1)$. □

Here, $*$ is a new symbol to denote that any imitation cannot be applied. From indexed terms, we form a *common indexed term* as a common part of them.

Definition 2. *For a reduced expression $E = \{\langle s_i, t_i\rangle \mid i \in N\}$ such that $\mathrm{hd}(s_i) = P$, let $J(s_i, t_i)$ be an indexed term of s_i for t_i. Then, a common indexed term $J(E) = \sqcap_{i \in N} J(s_i, t_i)$ of $\{J(s_i, t_i) \mid i \in N\}$ is defined inductively as follows:*

(1) If $J(s_i, t_i) = c^{\rho_i}$ and $c \in IC \cup IV \cup \{*\}$ for each $i \in N$, then $J(E) = c^{\bigcap_{i \in N} \rho_i}$.
(2) If $J(s_i, t_i) = f^{\rho_i}(t^i_1, \ldots, t^i_m)$ and $f \in FC$ for each $i \in N$, then:

$$J(E) = f^{\cap_{i\in N}\rho_i}(\sqcap_{i\in N} J(s_i, t_1^i), \ldots, \sqcap_{i\in N} J(s_i, t_m^i)).$$

(3) If $J(s_i, t_i) = p^{\emptyset}(t_1^i, \ldots, t_m^i)$ and $p \in PC \cup \{\neg, \wedge, \vee, \supset\}$ for each $i \in N$, then:

$$J(E) = p^{\emptyset}(\sqcap_{i\in N} J(s_i, t_1^i), \ldots, \sqcap_{i\in N} J(s_i, t_m^i)).$$

(4) If $J(s_i, t_i) = Qx_i^{\emptyset}.t_i'$ and $Q \in \{\forall, \exists\}$ for each $i \in N$, then:

$$J(E) = Qx^{\emptyset}. \sqcap_{i\in N} (J(s_i, t_i')[x_i := x]).$$

(5) If there exist $k, j \in N$ such that $\mathrm{hd}(J(s_k, t_k)) \neq \mathrm{hd}(J(s_j, t_j))$, then $J(E) = *^{\cap_{i\in N}\rho_i}$. □

Finally, we form a *reduced indexed term* from a common indexed term.

Definition 3. *Let $E = \{\langle s_i, t_i\rangle \mid i \in N\}$ be a reduced expression. Then a reduced indexed term $\sqcap E$ of E is an indexed term obtained by applying the following rule to $J(E)$ as often as possible:*

For a subterm t' of $J(E)$ if there exists a $j (1 \leq j \leq m)$ such that $t' = f^{\emptyset}(t_1', \ldots, t_j', \ldots, t_m')$ and $t_j' = *^{\emptyset}$, then replace t' with $*^{\emptyset}$. □

If $\sqcap E \neq *^{\emptyset}$, then at least a transformation from E to $\emptyset$ exists, and the composition of substitutions derived in the transformation becomes a matcher of E if they are consistent. The problem of indexing for a $E = \langle P(s_1, \ldots, s_r), t\rangle$ can be reduced to the matching of ordered subtrees with logical variables in [13], and the indexing for each pair $\langle s_i, t\rangle$ can be done in time $O(\mid t \mid)$. Hence, the total time required for the indexing concerning E is $O(r\cdot \mid t \mid)$. By estimating $O(r\cdot \mid t \mid) \leq O(|E|^2)$, we have the following result.

Theorem 2. *Let E be a reduced expression. Then, $\sqcap E$ can be constructed in $O(|E|^2)$ time, and E is matchable only if $\sqcap E \neq *^{\emptyset}$.* □

Especially, if E contains no syntax variables, then E is matchable if $\sqcap E \neq *^{\emptyset}$ since the set of obtained substitutions is always consistent. Hence we have the following property.

Corollary 1. *Let E be a reduced expression which contains no syntax free variables. Then, E is matchable if and only if $\sqcap E \neq *^{\emptyset}$.* □

Example 2. Matchability test based on reduced indexed terms.

(1) Consider the following expression E_1:

$$\langle \forall x_1(P(x_1, F_c(x_1)) \wedge P(x_1, G_c(x_1)),\ \forall x_2(\exists z_1.p(z_1, f(x_2)) \wedge \exists z_2.p(z_2, g(x_2)))\rangle$$

By applying the pre-processing to E_1, we obtain the following reduced expression E_1'.

$$\begin{aligned} E_1' &= \{\langle P(w, F_c(w)), \exists z_1.p(z_1, f(w))\rangle, \langle P(w, G_c(w)), \exists z_2.p(z_2, g(w))\rangle\} \\ &= \{\langle s_1, t_1\rangle, \langle s_2, t_2\rangle\}. \end{aligned}$$

It holds that $\sqcap_{i\in\{1,2\}} J(s_i, t_i) = \exists z^{\emptyset}.p^{\emptyset}(z^{\emptyset}, *^{\{2\}}) = \sqcap E_1' \neq *^{\emptyset}$. Accordingly, E_1 is matchable if the obtained substitutions are consistent by Theorem 2.

(2) Consider the following expression E_2 which contains no syntax free variables:

$$E_2 = \left\{ \begin{array}{l} \langle \forall x_1.((P(x_1, F_c(x_1)) \wedge P(x_1, G_c(x_1))) \supset P(f(x_1), x_1)), \\ \quad \forall x_2.((\exists z_1.p(z_1, f(x_2)) \wedge \exists z_2.p(z_2, g(x_2))) \supset \exists z_3.p(z_3, f(x_2)))\rangle \end{array} \right\}.$$

Firstly, we apply the pre-processing to E_2. Then we have the following reduced expression E_2'.

$$\begin{array}{rl} E_2 \Rightarrow & \left\{ \begin{array}{l} \langle (P(w, F_c(w)) \wedge P(w, G_c(w))) \supset P(F_f(w), w), \\ \quad (\exists z_1.p(z_1, f(w)) \wedge \exists z_2.p(z_2, g(w))) \supset \exists z_3.p(z_3, f(w)))\rangle \end{array} \right\} \\ \Rightarrow & \left\{ \begin{array}{l} \langle P(w, F_c(w)) \wedge P(w, G_c(w)),\ \exists z_1.p(z_1, f(w)) \wedge \exists z_2.p(z_2, g(w))\rangle, \\ \langle P(F_f(w), w),\ \exists z_3.p(z_3, f(w))\rangle \end{array} \right\} \\ \Rightarrow & \left\{ \begin{array}{l} \langle P(w, F_c(w)),\ \exists z_1.p(z_1, f(w))\rangle, \\ \langle P(w, G_c(w)),\ \exists z_2.p(z_2, g(w))\rangle, \\ \langle P(F_c(w), w),\ \exists z_3.p(z_3, f(w))\rangle \end{array} \right\} \quad \left(= \left\{ \begin{array}{l} \langle s_1, t_1\rangle, \\ \langle s_2, t_2\rangle, \\ \langle s_3, t_3\rangle \end{array} \right\} = E_2' \right). \end{array}$$

Next, we apply the projection point indexing to E_2'. The indexed terms@are given as follows:

$$J(s_1, t_1) = \exists z_1^{\emptyset}.p^{\emptyset}(z_1^{\emptyset}, f^{\{2\}}(w^{\{1\}})),\ J(s_2, t_2) = \exists z_2^{\emptyset}.p^{\emptyset}(z_2^{\emptyset}, g^{\{2\}}(w^{\{1\}})),$$
$$J(s_3, t_3) = \exists z_3^{\emptyset}.p^{\emptyset}(z_3^{\emptyset}, f^{\{1\}}(w^{\{2\}})).$$

By Definition 2, it holds that $\sqcap_{i \in \{1,2,3\}} J(s_i, t_i) = \exists z^{\emptyset}.p^{\emptyset}(z^{\emptyset}, *^{\emptyset})$. By Definition 3, it holds that $\sqcap E_2' = *^{\emptyset}$. Hence, E is not matchable by Corollary 1. □

4 Matcher Derivation Algorithm

4.1 A general matcher derivation algorithm

In this section we study algorithms of deriving any matchers of E from a given common reduced indexed term $\sqcap E$. Intuitively, projection rules can be applied to the positions of $\sqcap E$ whose index set ρ is not $\emptyset$, and imitation rules can be applied to any positions of $\sqcap E$ which are not $\star$.

Definition 4. *Let E be an expression such that $\{\langle s_i, t_i\rangle \mid 1 \le j \le m\}$,where $hd(s_i) = P$ for any $i (1 \le i \le m)$. The set of substitutions $S_{\sqcap E}$ for E is defined inductively as follows.*

(1) $\sqcap E = c^{\rho}$: $S_{\sqcap E} = \{\lambda \bar{v_n}.c\} \cup \{\lambda \bar{v_n}.v_j \mid j \in \rho\}$.
(2) $\sqcap E = \star^{\rho}$: $S_{\sqcap E} = \{\lambda \bar{v_n}.v_j \mid j \in \rho\}$.
(3) $\sqcap E = f^{\rho}(\sqcap E_1, \ldots, \sqcap E_m)$:
$S_{\sqcap E} = \{\lambda \bar{v_n}.f(t_1, \ldots, t_m) \mid \lambda \bar{v_n}.t_i \in S_{\sqcap E_i}, 1 \le i \le m\} \cup \{\lambda \bar{v_n}.v_j \mid j \in \rho\}$. □

A *matcher derivation term* of E is a subterm of a reduced indexed term $\sqcap E$ formed by deleting the descendant of every node to which a projection rule is chosen and the choiced index is labelled to each node of it. In case of $\sqcap E = p^{\emptyset}(f^{\{1,2\}}(c^{\{3,4\}}\star^{\{5,6\}}))$, the terms $p^{\emptyset}(f^{\{1\}})$, $p^{\emptyset}(f^{\emptyset}(c^{\emptyset}, \star^{\{5\}}))$ and $p^{\emptyset}(f^{\emptyset}(c^{\{4\}}, \star^{\{5\}}))$ are examples of matcher derivation terms of $\sqcap E$. A atcher derivation term specifies a substitution $\theta \in \{[H := t] \mid t \in S_{\sqcap E}\}$.

Note that the substitutions for syntax free variables arise only when projection rules are applied. Accordingly, consistency check is required only when projection rules are applied.

Theorem 3. *Let E be an expression and let $\sqcap E$ be its reduced indexed term. Then each consistent $\theta \in \{[H := t] \mid t \in S_{\sqcap E}\}$ is a matcher of E.* □

Corollary 2. *Let E be an expression which contain no syntax free variables and let $\sqcap E$ be its reduced indexed term. Then each $\theta \in \{[H := t] \mid t \in S_{\sqcap E}\}$ is a matcher of E.* □

Example 3. Matcher derivation from $S_{\sqcap E}$

(1) Let E be an expression such that

$$E = \left\{ \begin{array}{l} \langle H(w, F_c(x_a, w)), f(a, w)\rangle, \\ \langle H(x_a, F_c(x_a, x_b)), f(a, b)\rangle, \\ \langle H(x_a, F_c(x_a, x_c)), f(a, c)\rangle \end{array} \right\}.$$

Then $\sqcap E = f^{\{2\}}(a^{\emptyset}, w^{\{1\}})$. The derivation terms of $\sqcap E$ are $f^{\emptyset}(a^{\emptyset}, w^{\{1\}})$ and $f^{\{2\}}$. Hence, we have the following substitutions:

$$\left\{ \begin{array}{l} \theta_1 = [H := \lambda v_1 v_2.f(a, v_1), x_a := b, x_a := c], \\ \theta_2 = [H := \lambda v_1 v_2.v_2, F_c := \lambda v_1 v_2.f(v_1, v_2), x_a := a]. \end{array} \right\}$$

Here, θ_2 is a consistent matcher, but θ_1 is not.

(2) Let E_2 be the expression in Example2. Then $\sqcap_{E_2} = \exists z^{\emptyset}.p^{\emptyset}(z^{\emptyset}, *^{\{2\}})$. In this case, the derivation term of E_2 is uniquely defined as $\exists z^{\emptyset}.p^{\emptyset}(z^{\emptyset}, *^{\{2\}})$. Hence we have the following matcher. $\theta = [P := \lambda v_1 v_2.\exists z.p(z, v_2)]$. □

A matcher derivation term is given by choosing rules and indices from reduced indexed terms. The efficiency of the matching strongly depend on this choosing procedure. We introduce a preference order $\succ_{od}$ into S_E such that if $r_1 \succ_{od} r_2$, then r_1 is chosen in preference to r_2, and denote the ordered set by $(S_E, \succ_{od})$. A strategy which derives matchers according to a preference order $\succ_{od}$ is called *od-strategy*, and a procedure which is based on od-strategy is denoted by $Match^{od}$. A general procedure of $Match^{od}$ is given as follows:

Algorithm $Match^{od}$:
Input: E=$\{\langle P(\bar{s}^j), t_j\rangle \mid 1 \leq j \leq m\}$
Output: Matchers of E
begin
(1)**Pre-process** E to a reduced expression E';
(2)**Derive** the reduced indexed term $\sqcap E$;
  If $\sqcap E = \star^{\emptyset}$ **then** output fail;
  else
(3) **While** a possible matcher derivation term exists **do**;
  (3-1)**Choose** a possible matcher derivation term under od-strategy;
  (3-2)**Derive** substitutions according to the chosen matcher derivation term;
  (3-3)**Check** the consistency of the obtained substitutions;
  If consistent **then output** the substitutions;
end

As a basic preference order, we introduce the *imitation preference order*.

Definition 5. *The imitation preference order denoted by* $\succ_I$ *and the ordered set* $(S_{\sqcap E}, \succ_I)$ *are defined inductively as follows:*

(1) $\sqcap E = c^\rho$: $(S_{\sqcap E}, \succ_I) = \{\lambda \bar{v_n}.c \succ_I \lambda \bar{v_n}.v_1 \succ_I \ldots \succ_I \lambda \bar{v_n}.v_{|\rho|}\}$.
(2) $\sqcap E = \star^\rho$: $(S_{\sqcap E}, \succ_I) = \{\lambda \bar{v_n}.v_1 \succ_I \ldots \succ_I \lambda \bar{v_n}.v_{|\rho|}\}$.
(3) $\sqcap E = f^\rho(S_{\sqcap E_1}, \ldots, S_{\sqcap E_m})$: $(S_{\sqcap E}, \succ_I) =$
$\{\{\lambda \bar{v_n}.f(t_1, \ldots, t_m) \mid \lambda \bar{v_n}.t_i \in (S_{\sqcap E_m}, \succ_I)\} \succ_I \lambda \bar{v_n}.v_1 \succ_I \ldots \succ_I \lambda \bar{v_n}.v_{|\rho|}\}$.
□

Theorem 4. $Match^I$ *is sound and complete, that is, it derives any matchers of* E. □

Especially, if E contains no syntax free variables, then the consistency check (3-3) is not necessary. Hence, for each choice in (3-1), $Match^I$ derives a consistent matcher.

Example 4. Let E be an expression such that $\langle P(w, w, F_c(w), x_c, x_c),\ p(f(w), a)\rangle$. Then $\sqcap E = p^\emptyset(f^{\{3\}}(\star^{\{1,2\}}), a^{\{4,5\}})$, and $Match^I$ derives the following 9 matchers in this order.

$$\left\{\begin{array}{l}
(1)[P := \lambda v_1 v_2 v_3 v_4 v_5.p(f(v_1), a)] \\
(2)[P := \lambda v_1 v_2 v_3 v_4 v_5.p(f(v_2), a)], \\
(3)[P := \lambda v_1 v_2 v_3 v_4 v_5.p(f(v_1), v_4), x_c := a], \\
(4)[P := \lambda v_1 v_2 v_3 v_4 v_5.p(f(v_1), v_5), x_c := a] \\
(5)[P := \lambda v_1 v_2 v_3 v_4 v_5.p(f(v_2), v_4), x_c := a] \\
(6)[P := \lambda v_1 v_2 v_3 v_4 v_5.p(f(v_2), v_5), x_c := a] \\
(7)[P := \lambda v_1 v_2 v_3 v_4 v_5.p(v_3, a), F_c := \lambda v.f(v)] \\
(8)[P := \lambda v_1 v_2 v_3 v_4 v_5.p(v_3, v_4), F_c := \lambda v.f(v), x_c := a] \\
(9)[P := \lambda v_1 v_2 v_3 v_4 v_5.p(v_3, v_5), F_c := \lambda v.f(v), x_c := a]
\end{array}\right\}$$

4.2 Efficiently Computable Predicate Schema Matching Classes

There exist one imitation rule and $\mid \rho \mid$ projection rules which can be applied to positions indexed as c^ρ or $f^\rho(\cdots)$. Hence, the number of matcher derivation terms increases in exponential as for the number of such positions. It is essential for designing an efficient matching algorithm to suppress this combinational explosion.

One of the reasons of the increase of the size of ρ is the occurrence count of identical terms in arguments of schemas. For example, in Example 4, the same symbols w, x_c occur twice in the schema variable P. Hence, there exist 4 combinations as for w, x_c to form matcher derivation terms. However, it is meaningless to derive matchers for all such combinations. In order to suppress such meaningless combinations, we restrict the occurrence count of each term in an atom to be at most one. From this viewpoint, we introduce the condition of *simplex*.

For a position indexed as $\star^\rho$, we have to apply projection rules. The size of ρ depend on the occurrence count of each bound variable w in an atom. For example, let E be $\{\langle P(w, F_c(w)), p(f(w), a, b)\rangle\}$. Then, $\sqcap E$=$p^\emptyset(f^{\{2\}}(w^{\{1\}}), a^\emptyset, b^\emptyset)$, and 2 matcher derivation terms exist. If we restrict the occurrence count of w

up to one, then the matcher derivation term is decided uniquely. For example, in case of $\{\langle P(w, F_c(x_c)), p(f(w), a, b)\rangle\}$, its matcher derivation term is decided uniquely as $p(f^{\emptyset}(w^{\{1\}}), a^{\emptyset}, b^{\emptyset})$. From this viewpoint, we introduce the condition of *linear*.

The existence of bound variables in each argument of any atoms reduces the indeterminacy of choosing rules. For example, let E be an expression such that

$$\{\, \langle P(x_c, y_c),\ p(a, b)\rangle, \langle P(y_c, x_c),\ p(a, c)\rangle \,\}\,.$$

Then $\sqcap E = p^{\emptyset}(a^{\{1,2\}}, \star^{\{1,2\}})$. For $\star^{\{1,2\}}$, there exist 2 possible projections. On the other hand, in case of

$$\{\, \langle P(x_c, y_c),\ p(a, b)\rangle, \langle P(y_c, x_c),\ p(a, c)\rangle\ \langle P(w, x_c)),\ p(a, w)\rangle \,\}\,,$$

$\sqcap E = f^{\emptyset}(a^{\{2\}}, \star^{\{1\}})$, and the index set of the position $\star$ is decided uniquely. Thus, for a position $\star^{\rho}$, if there exists a $s_i^j (1 \leq j \leq m)$ for each $i \in \rho$ which contains bound variables, that is, the position $\star$ is dominated with bound variables, then the matcher derivation term is uniquely decided. From this viewpoint, we introduce the condition of *dominated*.

Definition 6. *Let $\{\langle P(s_1^j, \ldots, s_r^j), t^j\rangle \mid 1 \leq j \leq m\}$ be a reduced expression of $\langle \Phi, \phi\rangle$. Then*

(1) An atom $P(s_1^j, \ldots, s_r^j)$ is simplex if it contains bound variables, then $s_i^j \neq s_{i'}^j (1 \leq i, i' \leq r)$ holds for any $j (1 \leq j \leq m)$. A schema Φ is simplex if each atom of Φ is simplex.
(2) An atom $P(s_1^j, \ldots, s_r^j)$ is linear if the occurrence count of each bound variable of the atom is at most one. A schema Φ is linear if each atom of Φ is linear.
(3) A set of atoms $\{P(s_1^j, \ldots, s_r^j) \mid (1 \leq j \leq m)\}$ is dominated if $s_i^j \neq s_i^{j'} (1 \leq j, j' \leq m)$ holds for some $i (1 \leq i \leq r)$, then at least a $s \in \{s_i^j \mid 1 \leq j \leq m\}$ contains bound variables. A schema Φ is dominated if each atom set of Φ with a same head is dominated. □

Example 5. Let Φ_1, Φ_2, Φ_3 be schemas such that:

$$\left\{ \begin{array}{l} \Phi_1 = \forall xy.P(x, y, x_c, y_c) \supset P(x_c, y_c, x_c, y_c) \\ \Phi_2 = \forall x.P(x, F_c(x), x_c, y_c) \supset P(x_c, x_c, x_c, x_c) \\ \Phi_3 = \forall xy.P(x, y, x_c, x_c) \supset P(x_c, y_c, x_c, x_c) \end{array} \right\}$$

Then Φ_1 is simplex, dominated and linear. Φ_2 is simplex, but is not dominated since the 4th argument of its atoms are different ($y_c \neq x_c$) but no bound variables occurs in the 4th argument of both atoms. Further Φ_2 is not linear since x occurs twice in $\forall x.P(x, f(x), x_c, y_c)$. Φ_3 is not *simplex* since x_c occurs twice in $\forall x.P(x, f(x), x_c, x_c)$, but is dominated and linear. □

A schema Φ is called *s-l-d* if it satisfies all the conditions of simplex, dominated and linear. Let $E = \{\langle s_i, t_i\rangle \mid i \in N\}$ be a reduced expression obtained from a *s-l-d* schema Φ, and let $J(E)$ be the common indexed term of E. Assume that $\star^{\rho}$ be a position of $J(E)$. If $\star^{\rho}$ is derived by $hd(J(s_k, t_k)) \neq hd(J(s_j, t_j)$

and $s_l^k = s_l^j$ holds for any $k, j \in N$ where $s_k = P(s_1^k, \ldots, s_r^k)$,$s_j = P(s_1^j, \ldots, s_r^j)$ and $l \in \rho$, then the substitutions obtained by applying projection rules to the position ,i.e.,$\lambda\overline{v_n}.v_l, l \in \rho$, are inconsistent. Therefore, this matching fails. For this reason, for the case of s-l-d schema, we modify the definition of common indexed term (Definition 2(5)) as follows:

(5)' If there exist $k, j \in N$ such that $\mathrm{hd}(J(s_k, t_k)) \neq \mathrm{hd}(J(s_j, t_j))$, then $J(E) = *^{\cap_{i \in N}\rho_i - \alpha}$, where $\alpha = \{l \mid s_l^k = s_l^j, \forall k, j \in N\}$ and $s_k = P(s_1^k, \ldots, s_r^k)$, $s_j = P(s_1^j, \ldots, s_r^j)$.

From the condition of linear, the relation $\mid \rho \mid \leq 1$ holds for any $\star^\rho$. Accordingly, if Φ is s-l-d schema, then the positions to which projection rules must be applied are uniquely specified.

Example 6. Matching for s-l-d schemas.

(1) Let $E_1 = \{\langle P(x_c, y_c), p(a)\rangle, \langle P(x_c, z_c), p(b)\rangle, \langle P(x_c, w), p(c)\rangle\}$. Then we have $\sqcap E_1 {=} p^\emptyset(\star^\emptyset)$ instead of $p^\emptyset(\star^{\{1\}})$. Finally we have the reduced indexed term $\star^\emptyset$, and therefore this matching fails. We can ascertain that the substitutions $[x_c := a, x_c := b, x_c := c]$ obtained by applying the projection $\lambda uv.u$ to the $\star$ position are inconsistent.

(2) Let $E_2 = \{\langle P(F_c(x_c)), p(f(a))\rangle, \langle H(F_c(x_c)), p(f(b))\rangle\}$. Then we have $\sqcap E_2 {=} \star^\emptyset$ by reducing from $p^\emptyset(f^{\{1\}}(\star^\emptyset))$. Hence this matching fails.

(3) Let $E_3 = \{\langle P(F_c(x_c)), p(f(a))\rangle, \langle P(F_c(y_c)), p(f(b))\rangle, \langle P(F_c(w)), p(f(w))\rangle$. Then $\sqcap E_3 {=} p^\emptyset(f^{\{1\}})$, and E_3 is matchable. The matcher is $[P := \lambda u.p(u), F_c := \lambda v.f(v), x_c := a, y_c := b]$. □

Let $t = \sqcap E$ be the reduced indexed term of E obtained from a s-l-d schema. Then a strategy which uses only imitation rules for the positions indexed as c^ρ or $f^\rho(\cdots)$ is called *strong imitation preference*. The matching $Match^S$ based on this strong imitation preference strategy is defined as follows.

Definition 7. *s-l-d schema matching.*

(1) $\sqcap E = c^\rho$: $(S^s_{\sqcap E}, \succ) = \{\lambda\bar{v_n}.c\}$.
(2) $\sqcap E = \star^\rho$: $(S^s_{\sqcap E}, \succ) = \{\lambda\bar{v_n}.v_j\}$, *where* $j \in \rho$ $(\mid \rho \mid \leq 1)$.
(3) $\sqcap E = f^\rho(S_{\sqcap E_1}, \ldots, S_{\sqcap E_m})$: $(S^s_{\sqcap E}, \succ) {=} \{\lambda\bar{v_n}.f(t_1, \ldots, t_m) \mid \lambda\bar{v_n}.t_i \in S_{\sqcap E_m}\}$.
□

Next, we estimate the time complexity of $Match^S$. Let E be a reduced expression. The procedures in (3) are linear except for (3-1). However, in the case of the s-l-d schema matching, this procedure (3-1) becomes linear since the projection position set is uniquely decided and $\mid \rho \mid \leq 1$.

The projection position indexing procedure is computable in time $O(r\cdot \mid t \mid)$ as stated in Theorem2. Especially, in the case of schema matching, we can assume that $\mid t \mid \leq (k \mid s_i \mid)$ holds for some fixed $k \in N$. Accordingly, in this case, the complexity of the matching becomes $O(\mid E \mid)$. Therefore, in most of the cases, the s-l-d schema matching runs almost in linear time. Thus, we have the following results.

Theorem 5. *Let Φ be a s-l-d schema and E be a reduced expression for some $\langle \Phi, \phi \rangle$. Then $Match^S$ derives a unique matcher of E almost in linear time if E is matchable.* □

Example 7. Let E be a matching pair $\langle \Phi, \varphi \rangle$ such that

$$\left\{ \begin{array}{l} \Phi = P(x_c) \wedge \forall x.(P(x) \supset P(F_c(x))) \supset P(F_c(F_c(x_c))). \\ \phi = (p(0) \wedge q(0)) \wedge \forall x.(p(x) \wedge q(x) \supset p(suc(x)) \wedge suc(x)) \\ \qquad\qquad \supset (p(suc(suc(0))) \wedge q(suc(suc(0))) \end{array} \right\}$$

At first, we have the following reduced expression:

$$E = \left\{ \begin{array}{ll} \langle P_1(x_c), p(0) \rangle & \langle P_1(w), p(w) \ \rangle, \\ \langle P_1(F_c(w)), p(suc(w)) \rangle, & \langle P_1(F_c(F_c(x_c))), p(suc(suc(0))) \rangle \\ \langle P_2(x_c), q(0) \rangle & \langle P_2(w), q(w) \ \rangle \\ \langle P_2(F_c(w)), q(suc(w)) \rangle, & \langle P_2(F_c(F_c(x_c))), q(suc(suc(0))) \rangle \end{array} \right\}$$

Then we have the reduced indexed terms such that :

$$\left\{ \sqcap E_1 = p^{\emptyset}(\star^{\{1\}}), \quad \sqcap E_2 = q^{\emptyset}(\star^{\{1\}}). \right\}$$

Hence, we have the substitutions such that

$$\{P_1 := \lambda u_1.p(u_1), P_2 := \lambda u_2.q(u_2), F_f := \lambda v.suc(v), x_c := 0\}.$$

Finally, we have the matcher $[P := \lambda u_1 u_2.p(u_1) \wedge q(u_2), F_c := \lambda v.suc(v), x_c := 0\}$. □

5 Discussions and Conclusion

We have discussed in this paper the second order predicate schema matching motivating to apply it to the schema guided theorem proving. We proposed an algorithm $Match^I$ which pre-checks the projection positions and showed an efficiently computable class characterized by the conditions simplex,linear and dominated exists. Though the conditions seem to be too restrictive, these restrictions do not reduce the expressive power of schemas. For example,$\forall x.P(x, x_c, F_c(y_c))$ matches with formulas containing the bound variable x such that

$$\left\{ \begin{array}{l} \forall x.p(x, a, f(a), g(x)), \\ \forall x.(\exists y.q(x, y, a, f(a), b, f(x), g(y)), \\ \forall x.(\exists y.p(x, f(y), a, g(x)) \wedge \forall y \exists z.q(x, y, z, a, b, f(a), g(b)) \\ \qquad\qquad \cdots\cdots \end{array} \right\}$$

The conditions of dominated are motivated from the provability of schemas. Let Φ be a schema expressed in sequent such that

$$\forall x \forall y.P(x_a, x, F_c(y)) \Rightarrow P(x_a, x_b, x_c).$$

Then its sequent style proof (LK proof) [15] is given as follows:

$$\dfrac{\dfrac{P(x_c, w_b, F_c(w_c)) \Rightarrow \ P(x_c, y_c, F_c(z_c))}{\forall y.P(x_c, w_b, F_c(y)) \Rightarrow P(x_c, y_c, F_c(z_c))}\ (\forall\_L)}{\forall x \forall y.P(x_c, x, F_c(y)) \Rightarrow P(x_c, y_c, F_c(z_c))}\ (\forall\_L)$$

A sequent which contains a same formula in the both side of $\Rightarrow$ is called an *axiom*, and each leaf of LK-proofs must be an axiom. Therefore, every quantified variable must be substituted for some constant so that leaves become axioms. In the above example, w_b and $F_c(w_c)$ must be substituted for y_c, $F_c(z_c)$, respectively. Thus, all the syntax free variables occurring in Φ are either remain unchanged or identified with bound variables. In this example, "x_c" is unchanged and "y_c","$F_c(z_c)$" are identified with $w_b, F_c(w_c)$,respectively. This relation must be preserved for each formula ϕ to be matched. Thus, a schema which satisfies the s-l-d conditions is provable. However, provable schema is not always s-l-d. The characterization from the ligical viewpoint should be discussed further.

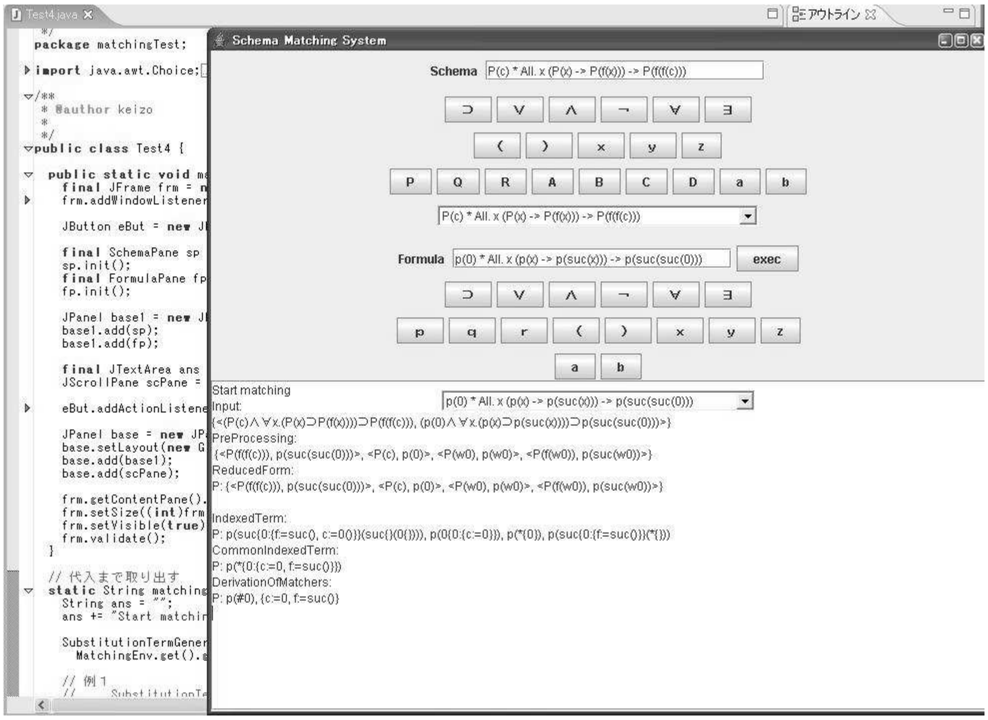

Fig. 1. Schema matching system

The proposed matching algorithm is implemented and is used in the schema guided theorem prover which we have developed (Fig.1). The algorithm works satisfactory and the class which we have introduced in this paper covers enough schemas for our system. Thus the second order matching is very complex in general, but is useful if we amalgamate some first order features with the second order ones.

References

1. L. D. Baxter. *The complexity of unification*. PhD thesis, Department of Computer Science ,University of Waterloo, 1977.
2. B. Brock, S. Cooper, and W. Pierce. Analogical reasoning and proof discovery. *LNCS*, No.310:454–468, 1988.
3. R. Curien, Z. Qian, and H. Shi. Efficient second order matching. In *RTA 96(Rewriting Techniques and Application)*, pages 317–331, 1996.
4. M. R. Donat and L. A. Wallen. Learning and applying generalised sotutions using higher order resolution. *LNCS*, No.310:41–60, 1988.
5. G. Dowek. Third order matching is decidable. In *7th Annual IEEE Symposium on Logic in Computer Science*, pages 2–10, 1992.
6. P. Flener. *Logic program synthesis from incomplete information*. Kluwer Academic Press, 1995.
7. M. Harao. Proof discovery in lk system by analogy. In *LNCS(Proc. of the 3rd Asian Computing Science Conference)*, volume 1345, pages 197–211, 1997.
8. M. Harao and K. Iwanuma. Complexity of higher-order unification algorithm. *Society for Software Science and Technology*, No.8:41–53, 1991. In Japanese.
9. M. Harao, K. Yamada, and K. Hirata. Efficient second order predicate matching algorithm. In *Proc. Korea-Japan Joint Workshop on Algorithm and Computations*, pages 31–39, 1999.
10. K. Hirata, K. Yamada, and M. Harao. Tractable and intractable second-order matching problems. *Journal of Symbolic Computation*, 37:611–628, 2004.
11. G. P. Huet. A unification algorithm for typed λ-calculus. *Journal of Theoretical Computer Science*, 1:27–57, 1975.
12. G. P. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
13. P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, University of Helsinki Department of Computer Science, 1992.
14. D. A. Miller. A logic programming language with lambda-abstraction function variables, and simple unification. *J. of Logic and Computations*, 1(4):494–536, 1991.
15. A. S. Troelstra and H. Schwichtenberg. *Basic proof Theory*. Cambridge University Press, 1996.
16. K. Yamada, K. Hirata, and M. Harao. Schema matching and its complexity. *Trans. IEICE*, J82-D-I:1307–1316, 1999. In Japanese.
17. K. Yamada, S. Yin, M. Harao, and K. Hirata. Development of an analogy-based generic sequent style automatic theorem prover amalgamated with interactive proving. In *Proc. of IWIL'05 (the International Workshop on the Implementation of Logics 2005)*, 2005.

Panel Discussion: 20 Years After OBJ2

Organized by Kokichi Futatsugi

A memorable paper on OBJ2 [1] was published in the year 1985 in the proceedings of the Twelfth POPL. OBJ2 was an algebraic specification language designed and implemented from 1983 to 1984 by the four authors of the paper at SRI International in Menlo Park, California. OBJ2 was a novel executable formal specification language, and it had many new features its predecessor languages were lacking.

Although the definitive implementation of OBJ2 is OBJ3, which has been widely used as the latest OBJ language, almost all important language features were defined in OBJ2. The novel features of OBJ2 such as powerful module system, order sorted signature, rewriting modulo equational theories (A/C/I), evaluation strategy (E-strategy), etc. have influenced the designs of many languages in the last 20 years. Those languages include, at least:

BOBJ (`http://www-cse.ucsd.edu/groups/tatami/bobj/`)
CafeOBJ (`http://www.ldl.jaist.ac.jp/cafeobj/`)
CASL (`http://www.cofi.info/CASL.html`)
ELAN (`http://elan.loria.fr/`)
Maude (`http://maude.cs.uiuc.edu/`)

The four authors of the OBJ2 paper share the luck of participating RDP2005 exactly 20 yeas after the paper was published. All of the four have been frequent international travelers and attended many international conferences / symposiums / workshops, but it is really rare to have this kind of coincidence. The panel discussion of 20 Years After OBJ2 is organized by making use of this precious chance for discussing the future possibilities of algebraic specification languages, rewriting techniques, rewriting logic, behavioral / observational specification, etc. based on 20 years of experiences after OBJ2.

Panelists include the following four authors of the OBJ2 paper and one or two more persons.

Kokichi Futatsugi
JAIST (Japan Advance Institute of Science and Technology)
`futatsugi@jaist.ac.jp`
Joseph Goguen
University of California at San Diego
`goguen@cs.ucsd.edu`
Jean-Pierre Jouannaud
Ecole Polytechnique
`jouannaud@lix.polytechnique.fr`
Jose Meseguer
University of Illinois at Urbana-Champaign
`meseguer@cs.uiuc.edu`

References

1. K. Futatsugi, J.A. Goguen, J.P. Jouannaud and J. Meseguer. Principles of OBJ2. In: Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, January 1985 (POPL85), ACM (1985) 52-66.

Author Index