Volume Editors

Jordi Levy
Institut d’Investigació en Intel.lí•ncia Artificial (IIIA), CSIC, Barcelona, Spain
Email: levy@iiia.csic.es

Michael Kohlhase
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
Email: kohlhase+@cs.cmu.edu

Joachim Niehren
Department of Computer Science, Universität des Saarlandes Saarbrücken, Germany
Email: niehren@ps.uni-sb.de

Mateu Villaret
Dept. Informàtica i Matemàtica Aplicada, Universitat de Girona (UdG), Girona, Spain
Email: villaret@ima.udg.es

Proceedings of the 17th International Workshop on Unification, UNIF’03
Valencia, Spain, June 8-9, 2003

ISBN:

Depósit Legal:

Impreso en España.

Technical Report DSIC-II/12/03,
http://www.dsic.upv.es
Departamento de Sistemas Informáticos y Computación,
Preface


Unification is concerned with the problem of identifying given terms, either syntactically or modulo a given logical theory. The topic is understood in a rather broad sense at this workshop. The aim of UNIF 2003, as that of the previous meetings, is to bring together people interested in unification, present recent (even unfinished) work, and discuss new ideas and trends in unification and related fields. A list of typical UNIF topics includes, but is not limited to:

- Unification
  - E-unification
  - Unification Algorithms
  - Higher-Order Unification
  - String Unification
  - Context Unification
  - Combination problems
  - Disunification
  - Typed Unification
- Implementations
- Related Topics
  - Constraint Solving
  - Tree Descriptions
  - Matching
  - Narrowing
  - Applications
    - Type Checking and Type Inference
    - Automated Deduction
    - Rewriting
    - Functional and Logic Programming
    - Grammars
    - Computational Linguistics

The 17th Int. Workshop on Unification (UNIF’03) was held June 8–9, in Valencia, Spain, as an affiliated workshop of the 14th Int. Conference on Rewriting Techniques and Applications (RTA’03), and as part of the Federated Conference on Rewriting, Deduction and Programming (RDP’03). This technical report collects extended abstracts of the talks, and system demonstrations, and the abstracts of the two invited talks given by Dale Miller, on “Definitions, Unification, and the Sequent Calculus”, and Wojciech Plandowski, on “Test Sets for Large Families of Languages”.

We are especially grateful to the invited speakers, Dale Miller and Wojciech Plandowski, for accepting our invitation. UNIF 2003 was made possible in part by the kind financial support of the Consejo Superior de Investigaciones Cientificas (CSIC), and the Network of Excellence in Computational Logic (CologNet). The organizing committee also wishes to thank the RTA steering committee, and the RDP’03 Organizing committee for their cooperation and support throughout the preparation of this meeting.

June 2003

Jordi Levy
UNIF’03 Organizing Committee

Jordi Levy (chair)
Institut d’Investigació en Intel·ligència Artificial (IIIA)
Consejo Superior de Investigaciones Científicas (CSIC)
Barcelona, Spain
Email: levy@iiia.csic.es
http://www.iiia.csic.es/~levy

Michael Kohlhase
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
Email: kohlhase+@cs.cmu.edu
http://www.ags.uni-sb.de/~kohlhase

Joachim Niehren
Programming Systems Lab
Department of Computer Science
Universität des Saarlandes
Saarbrücken, Germany
Email: niehren@ps.uni-sb.de
http://www.ps.uni-sb.de/~niehren

Mateu Villaret
Dept. Informàtica i Matemàtica Aplicada (IMA)
Universitat de Girona (UdG)
Girona, Spain
Email: villaret@ima.udg.es
http://ima.udg.es/~villaret
Workshop Program

Sunday, June 8

11:00 - 11:30 Coffee Starter

11:30 - 13:00 Session 1

Presentation

Invited Talk: Definitions, Unification, and the Sequent Calculus
Dale Miller

13:00 - 15:00 Lunch

15:00 - 16:30 Session 2

Functional Programming with Sequence Variables: The Sequentica Package
Mircea Marin

Matching in Flat Theories
Temur Kutsia

Alpha-Prolog, a Fresh Approach to Logic Programming Modulo alpha-Equivalence
James Cheney, Christian Urban

16:30 - 17:00 Coffee Break

17:00 - 18:30 Session 3

Undecidability of Unification over Two Theories of Modular Exponentiation
Deepak Kapur, Paliath Narendran, Lida Wang

Information Flow Analysis via Equational Reasoning
Siva Anantharaman, Gaëtan Hains

Easy Intruder Deductions
Hubert Comon-Lundh, Ralf Treinen

End of the First Day
Monday, June 9

11:00 - 11:30 Coffee Starter

11:30 - 13:00 Session 4
   \textit{Invited Talk:} Test Sets for Large Families of Languages
   Wojciech Plandowski

   A Prefix Notation and Unification Algorithm for Encoding Modal Logics
   Adrian Williams, Jim Cunningham

13:00 - 15:00 Lunch

15:00 - 16:30 Session 5
   On the Computation of Joins for non Associative Lambek Categorial Grammars
   Annie Foret

   Satisfiability of Structural Subtype Constraints
   Joachim Niehren, Tim Priesnitz

   Applying Unification Techniques to XML Document Management?
   Michael Kohlhase

16:30 - 17:00 Coffee Break

17:00 - 18:30 Session 6
   Best Student Paper Award
   Panel: Open Problems in Unification
   Business Meeting

End of the UNIF Workshop
Contents

Invited Talk: Definitions, Unification, and the Sequent Calculus . 1
Dale Miller

Invited Talk: Test Sets for Large Families of Languages ........... 3
Wojciech Plandowski

Information Flow Analysis via Equational Reasoning .............. 5
Siva Anantharaman, Gaëtan Hains

αProlog, a Fresh Approach to Logic Programming Modulo α-Equivalence ................................................................. 15
James Cheney, Christian Urban

Easy Intruder Deductions ................................................... 21
Hubert Comon-Lundh, Ralf Treinen

On the Computation of Joins for non Associative Lambek
Categorial Grammars ...................................................... 25
Annie Forest

Undecidability of Unification over Two Theories of Modular
Exponentiation ................................................................ 39
Deepak Kapur, Paliath Narendran, Lida Wang

Applying Unification Techniques to XML Document Management? 51
Michael Kohlhase

Matching in Flat Theories .................................................. 57
Temur Kutsia

Functional Programming with Sequence Variables:
The Sequentica Package ..................................................... 65
Mirea Marin

Satisfiability of Structural Subtype Constraints .................... 79
Joachim Niehren, Tim Priesnitz

A Prefix Notation and Unification Algorithm for Encoding
Modal Logics .................................................................. 81
Adrian Williams, Jim Cunningham

Author’s Index ................................................................. 93
Invited Talk:
Definitions, Unification, and the Sequent Calculus

Dale Miller
INRIA/Futurs/Saclay & École polytechnique, France
Email: dale.miller@inria.fr
http://www.cse.psu.edu/~dale

A proof theoretic approach to definitions provides left and right introduction rules for defined atomic expressions [1,2]. These introduction rules enjoy cut-elimination [10,4] and are much more compatible with the proof search approach to logic programming than viewing definitions as logical equivalences. Such definitions can be seen as holding for all fixed points of the definition. If one wants to declare that a definition hold only for the least or greatest fixed point, then induction or coinduction inference rules can be used [7]. Definitions have been used to explain finite failure [2] as well as simulation in transition systems [5].

The right introduction rule for defined atoms uses matching and the left introduction rule uses unification. While these rules were originally developed for first-order logics, they can be naturally lifted to higher-order types since both unification and matching can be employed on simply typed λ-terms [3]. Thus, definitions provide a natural framework for reasoning about specifications involving higher-order abstract syntax [9,6].

Adding definitions, induction, and coinduction, however, is not sufficient to naturally encode specifications involving higher-order abstract syntax. Generic judgments arise when reasoning about logic programs involving higher-order quantification. To properly encode positive and (particularly) negative instances of such judgments, the new quantifier \( \forall \) (nabla) has been introduced [8]. The interaction between \( \forall \) and the two other quantifiers, \( \exists \) and \( \forall \), is easy to describe proof theoretically, and standard (higher-order) unification and matching is sufficient to deal with \( \forall \) within sequent calculus with definitions.

We shall provide several examples of definitions, unification, matching, and \( \forall \) in the specification of and reasoning about computation.

References


Invited Talk:
Test Sets for Large Families of Languages

Wojciech Plandowski
Institute of Informatics
University of Warsaw
Banacha 2, 02-097 Warszawa, Poland
E-mail: wojtek@impan.pl

Abstract. We study the lower and upper bounds for sizes of test sets for the families of all languages, of commutative languages, of regular languages and of context-free languages. We show the connection of these problems to problems on systems of word equations.
Information Flow Analysis via Equational Reasoning

Siva Anantharaman and Gaétan Hains

LIFO, Université d’Orléans (France)
FRE 2490 (CNRS - France)
Email: {siva,ghains}@lifo.univ-orleans.fr

Abstract. In this work, a process algebra is designed around an ACUID equational theory extended with prefixes symbolizing actions, and by making parallel synchronous composition distributive over non-deterministic choice; such a synchronous composition is commutative and non-associative. Bisimulation between processes is then interpretable as congruence over such an equational theory. It is shown that information flow analysis is strictly finer when based on bisimulation on the synchronous algebra, than when it is based on trace or weak bisimulation equivalence.

1 Introduction

In the context of information flow analysis, an important issue is that of ensuring the security of the data transmitted over distributed systems. The notions of non-interference and of admissible interference have been proposed with such an objective. On approaches based on process algebras (e.g. [7],[8],[10]), the equivalence with respect to weak trace or weak bisimulation is the key notion above which these notions are defined. In these approaches the internal CCS action $\tau$ has kept its ‘magic’ property of being unobservable even for the clock. This is undoubtedly unsatisfactory since the ‘divergence’ of a process – where an internal action $\tau$ loops for too long – will then go unobservable by definition, defeating one of the objectives of process algebras namely representation of liveness, deadlock or livelock.

In this work we propose an approach which is different, by designing the process algebra around an ACUID equational theory extended with prefixes which symbolize actions, and by making parallel synchronous composition (left- and right-) distributive over non-deterministic choice; such a synchronous composition will be commutative and non-associative. There are two essential differences in our approach as compared to a CCS-like approach: i) parallel composition of actions is defined with an explicit focus on the synchronous side; and asynchronous parallel composition although not formally part of the setup, can be defined in our formalism in complete conformity with CCS; ii) the internal action $\tau$ of CCS – meant as unobservable (including even as clock) in all standard works – is replaced by an action denoted $\theta$, which will be ‘internal except for the clock’. One of our objectives in doing so is to eliminate the observational problems caused by the divergence of a process, and to introduce a notion of discrete time.
We also show that information flow analysis is finer, when it is based on bisimulation in our synchronous setup than when it is based on trace or weak bisimulation equivalence. This is done as follows: action symbols in processes are labeled with levels of secrecy, and a notion of non-interference between processes called \( RNI \) (‘robust non interference’) is developed with respect to observation criteria based on these levels; we then show that \( RNI \) is a strict generalization of the notion of non-deterministic non interference which is based on weak trace and weak bisimulation.

Bisimulation between processes admits in our formalism a characterization as a congruence defined by a \( PACUID \)-equational theory (meaning: an \( ACUID \)-equational theory extended with prefixes). Viewed under such an angle, robust non interference can be interpreted in a natural manner as unifiability between any two prefix strings for any given process (sub-)term, modulo the equational theory extending \( PACUID \) with the given observation criterion.

2 A Synchronous Process Algebra

The process algebra is constructed here, as usual, over a given set \( \text{Act} \) of action symbols; attributing secrecy levels to the action symbols will be our concern in a later section. Let

\[
\text{Act} = \{ \theta \} \bigcup \{ a, \bar{a}, b, \bar{b}, \ldots \}
\]

(1)

where \( \theta \) is a special ‘internalizing’ action symbol similar to the \( \tau \) in CCS, but its role in information flow will be different: it will ‘count’ a unit of time, meant to be observable; ‘bars’ serve in pairing out synchronizing (or communicating) actions, in particular \( \bar{x} = x \); we refer to \( \bar{x} \) as the conjugate of \( x \); the special action \( \theta \) is assumed conjugate to itself. We will use variables \( x, y, \ldots \in \text{Act} \), \( \alpha, \beta, \ldots \in \text{Act}^* \) and \( P, Q, R, \ldots \in \text{Proc} \).

The grammar of ground process terms is

\[
\text{Proc} ::= 0 \mid \text{Act.Proc} \mid \text{Proc} + \text{Proc} \mid \text{Proc} \cdot \text{Proc} \mid P \setminus x;
\]

(2)

its operations are respectively called null process, prefix, (non-deterministic) choice, (parallel) synchronous composition, and restriction. Process \( x:0 \) will be abbreviated to \( x \) and \( x:(y,0) \) to \( xy \) where no confusion is likely between processes and their traces. The synchronization of actions is defined as a partial binary operation denoted ‘\(*\)’, as follows:

\[
x \ast y = \begin{cases} 
  \theta & \text{if } x = \bar{y} \\
  x & \text{if } y = \theta \\
  y & \text{if } x = \theta \\
  \text{undef} & \text{otherwise}
\end{cases}
\]

The difference between CCS’s \( \tau \) and our \( \theta \) must be already perceptible: CCS’s \( \tau \) would not have synchronized with any non-internal action, nor with itself.
The operational semantics of our processes is a labeled transition system, corresponding to a relation on \( \text{Proc} \times \text{Act} \times \text{Proc} \), that we shall denote as \( P \xrightarrow{x} Q \) and define by the following rules:

\[
\begin{align*}
\text{(Prefix)} & \quad \frac{}{x.P \xrightarrow{x} P} \\
\text{(Sum)} & \quad \frac{P \xrightarrow{x} P'}{P + Q \xrightarrow{x} P'} \quad \frac{Q \xrightarrow{x} Q'}{P + Q \xrightarrow{x} Q'} \\
\text{(Sync)} & \quad \frac{P \xrightarrow{\text{Prod}} P' \quad Q \xrightarrow{\text{Prod}} Q'}{P \times Q \xrightarrow{\text{Cont}} P' \times Q'} \\
\text{(Restr)} & \quad \frac{P \xrightarrow{x} P', \quad x \neq y, \quad y \neq \theta}{P \setminus y \xrightarrow{x} P' \setminus y}
\end{align*}
\]

In the current work we shall be concerned only with finite processes.

**Definition 1.** For any process term \( P \) the set of possible actions for \( P \) is defined as the set: \( \text{Act}(P) = \{ x \in \text{Act} \mid \exists P'. P \xrightarrow{x} P' \} \) and the set of its direct successor processes as: \( \text{Next}(P) = \{ P' \in \text{Proc} \mid \exists x. P \xrightarrow{x} P' \} \).

**Remark 1.**

i) Rule (Sync) is symmetric in \( x, \bar{x} \) because \( \bar{x} = x \).

ii) The semantics behind rule (Sync) is that \( \theta \) is meant as a passive clock step to be absorbed by non-empty events in a synchronous composition. A process emitting \( \theta \) is “counting” non-\( \theta \) events in its parallel sibling; in particular \( \theta \times \theta = \theta \), i.e. two ‘counting’ processes synchronize.

iii) The semantics behind the condition \( y \neq \theta \) in the rule (Restr) is that it is impossible to prevent a process from counting its internal actions.

### 2.1 Bisimulation

Simulation and bisimulation are defined as usual.

**Definition 2.** Process \( P \) is said to be simulated by \( Q \) or \( P \leq Q \) when \( P \xrightarrow{x} P' \) implies the existence of a process \( Q' \) such that \( Q \xrightarrow{x} Q' \) and \( P' \leq Q' \). Processes \( P \) and \( Q \) are bisimilar, if \( P \leq Q \) and \( Q \leq P \); notation: \( P \approx Q \).

Simulation is a preorder relation on the set of all processes, and bisimulation is an equivalence relation. It is in fact a congruence: if \( X \approx X' \) and \( Y \approx Y' \), and \( x \) is any action symbol, then:

\[
0 \approx 0, \quad x.X \approx x.X', \quad X + Y \approx X' + Y', \quad X*y \approx X'*Y', \quad \text{and} \quad X \setminus x \approx X' \setminus x.
\]

Bisimulation satisfies the following properties (where \( P, Q, R, \ldots \) denote arbitrary processes, and \( a, b, \ldots, x, y, \ldots \) are arbitrary actions): they are all easy consequences of our definitions. (The proofs of these, as well as of some other properties concerning the restriction operation \( P \setminus x \), are given in the appendix of [2].)
(1) The choice operator ‘+’ is associative, commutative, idempotent, and admits a unit element, up to bisimulation: That is to say, for any process terms \( P, Q, R \) we have:

\[
P + (Q + R) \approx (P + Q) + R, \quad P + Q \approx Q + P, \quad P + P \approx P, \quad P + 0 \approx P,
\]

(2) Prefix and (parallel) synchronous composition ‘commute’: \( a.P \cdot b.Q \approx (a \cdot b).P \cdot Q \)

(3) Synchronous composition ‘*’ is commutative, and distributes over the choice operator ‘+’, up to bisimulation. We have:

\[
P \cdot Q \approx Q \cdot P, \quad P \cdot (Q + R) \approx P \cdot Q + P \cdot R, \quad P \cdot 0 \approx 0
\]

Although we have not formally defined asynchronous parallel composition, we can deduce that notion uniquely up to bisimulation in our formalism. For doing that we shall need the **additive (normal) form** for any process term, which is uniquely determined up to bisimulation:

**Proposition 1.** Any finite process term \( P \) is bisimilar to the process term:

\[
a_0.P_0 + \ldots + a_{n-1}.P_{n-1} \text{ where } \{a_0, \ldots, a_{n-1}\} = \text{Act}(P).
\]

The proof is by induction on the size of \( P \). (Details in [2].) Parallel asynchronous composition – that we shall denote ‘|’ – can now be defined with the help of the additive normal forms: Let \( P, Q \) be any process terms and \( P = \sum a_i.P_i \) and \( Q = \sum b_j.Q_j \) be their additive normal forms. Then their asynchronous (parallel) composition is the process term (recursively) defined uniquely up to bisimulation, as: \( P \mid Q = \sum_{a_i} a_i.(P_i \mid Q) + \sum_{b_j} b_j.(P \mid Q_j) + \sum_{a_i=b_j} \theta.(P_i \mid Q_j) \); we set as the base case: \( 0 \mid P = P = P \mid 0 \), for this definition of ‘|’ to be effective.

The parallel asynchronous composition of any finite family of processes \( P_1, \ldots, P_r \) can be defined along similar lines; it is (uniquely determined up to bisimulation, and is) associative-commutative on \( P_1, \ldots, P_r \); we shall employ the standard CCS notation \( P_1 \mid \ldots \mid P_r \) to denote this parallel asynchronous composition.

### 2.2 Bisimulation as an Equational Congruence

Let \( \equiv \) be the binary relation on the set \( \text{Proc} \) of all finite process terms obtained by making \( + \) an associative, commutative and idempotent operator, and adding an operator ‘*’ 2-sided distributive over ‘+’, and an element 0 neutral for ‘+’ and absorbent for ‘*’; we thus get an ACUIE theory in the rewriting terminology. We also assume ‘*’ to be commutative, and that it annihilates synchronous compositions for which the actions do not synchronize. In other words, \( P \equiv Q \)
iff \( P = Q \) or \( P \) can be obtained from \( Q \) by applying one or more of the following equational axioms:

\[
\begin{align*}
(\text{A}) \quad & X + (Y + Z) = (X + Y) + Z & (\text{C}) \quad & X + Y = Y + X \\
(\text{I}) \quad & X + X = X & (\text{D}) \quad & X \cdot (Y + Z) = X \cdot Y + X \cdot Z \\
(\text{U}) \quad & X + 0 = X & (\text{C*}) \quad & X \cdot Y = Y \cdot X \\
(\text{Z}) \quad & X \cdot 0 = 0. & (\text{P}) \quad & a \cdot X \cdot b \cdot Y = (a \cdot b) \cdot (X \cdot Y); \\
& & & \text{is } 0 \text{ if } (a \cdot b) \text{ is undefined}
\end{align*}
\]

Our claim is that the congruence ‘\( \equiv \)’ defined by the above equational theory on the set of process terms is the same as that defined by bisimulation. We show first that the equivalence \( \equiv \) is correct w.r.t. bisimulation. Let us denote by \( P[R] \) a process term \( P \) having \( R \) as sub-term; in this context \( P[R'] \) represents term \( P \) where occurrences of \( R \) are replaced by the process \( R' \).

**Proposition 2.** \( P \equiv Q \) implies \( P \approx Q \).

**Proof.** Suppose \( R \equiv R' \) be an instance of any one of the axioms. Then, one of the properties listed above shows that \( R \approx R' \) and since bisimulation is a congruence for process terms, this equivalence can be propagated to \( P[R] \approx P[R'] \). Any equational proof of \( P \equiv Q \) is a finite sequence of such transformations. All of them preserve bisimulation, so by transitivity of bisimulation \( P \approx Q \). \( \square \)

To prove the reverse implication, we need a preliminary lemma, whose proof is by rather straightforward structural induction (cf. [2] for the details).

**Lemma 1.** \( P \approx 0 \) implies \( P \equiv 0 \).

**Proposition 3.** \( P \approx Q \) implies \( P \equiv Q \).

**Proof.** If \( P \approx 0 \) then \( P \equiv 0 \) by the previous lemma and similarly for \( Q \equiv 0 \equiv P \). So let us assume that \( \text{Act}(P) \) and \( \text{Act}(Q) \) are non-empty.

It is easy to deduce first that \( \text{Act}(P) = \text{Act}(Q) \); let \( A \) denote this common set of actions. Then we know that \( P \approx \sum_{a \in A} P_a \) and \( Q \approx \sum_{a \in A} Q_a \). Bisimulation implies that for every \( a \in A \) we have \( P_a \approx Q_a \) and by induction hypothesis (the terms \( P_a, Q_a \) are smaller than \( P, Q \) respectively) \( P_a \equiv Q_a \). One deduces then, by commutativity and associativity of ‘\( + \)’, that \( P \equiv Q \). \( \square \)

**Remark 2.** i) (Parallel) Synchronous composition is non-associative: in general we do not have \( P \cdot (Q \cdot R) \not\approx (P \cdot Q) \cdot R \). Here is an example: Let \( P = a, Q = \bar{a} \) and \( R = b \), with \( b \not\in \{a, \bar{a}\} \). Then \( P \cdot (Q \cdot R) = a \cdot (\bar{a} \cdot b) \) has no possible transition because \( Q \cdot R \) has none. But on the other hand \( (P \cdot Q) \cdot R = (a \cdot \bar{a}) \cdot b \not\rightarrow 0 \cdot 0, \) because \( (P \cdot Q) = (a \cdot \bar{a}) \not\rightarrow 0 \) and \( R = b \not\rightarrow 0 \).

ii) The PAUCID-equational theory given in section 2.2 above can be turned into a convergent system of rewrite rules, modulo the ACUI-axioms for the choice operation ‘\( + \)’, commutativity axiom for the synchronization operation ‘\( \cdot \)’, and visualizing each prefix (i.e. action symbol) as a monadic function. In particular, bisimulation between finite process terms is decidable by normalization under this convergent rewrite system.
3 Secrecy Levels for Actions and Non-Interference

Information flow analysis approaches based on process algebra attribute levels of secrecy to the action symbols; we shall be doing likewise. Although it is possible to present the ideas on a semi-complete lattice of levels of secrecy, for readability we shall limit the discussion to two levels, denoted \( Hi \) and \( Lo \) (and referred to as *high* and *low*) respectively.

We assume that the set \( \text{Act} \) of action symbols is partitioned into two subsets denoted by \( Hi \) and \( Lo \), representing secret and public events respectively; \( \theta \) is assumed to be in \( Lo \) (‘clock is observable by anyone’). Both \( Hi \) and \( Lo \) are assumed stable under conjugacy, i.e. \( Hi = \overline{Hi}, Lo = \overline{Lo} \). An arbitrary action symbol in \( Hi \) (resp. \( Lo \)) will be denoted \( hi \) (resp. \( lo \)). When no confusion is likely, these sets will be referred to simply as \( H \) and \( L \). Following [4], we define then the *observation criterion* for low level users \( O_L \) as follows: For any process \( Q \), its observable image \( Q/O_L \) is the process constructed inductively from \( Q \) under the following inference rules:

\[
\begin{align*}
\text{(No Mask)} & \quad \frac{P \xrightarrow{\omega} P'}{P/O_L \xrightarrow{\omega} P'/O_L} \quad x \in Lo \\
\text{(Mask)} & \quad \frac{P \xrightarrow{\omega} P'}{P/O_L \xrightarrow{\omega} P'/O_L} \quad x \in Hi
\end{align*}
\]

The semantics behind these inference rules is that for any high level action \( x \), the part observable by a low level observer reduces to the time unit that \( x \) consumes.

**Remark 3.** i) It is not true that \( P/O_L \) is bisimilar to the process obtained from \( P \) by replacing any occurrence of any high level action symbol \( hi \) in \( P \) by an occurrence of \( \theta \). For instance for any \( P \), any \( lo \neq \theta \) and any \( hi \), we have by our definitions that: \( lo \ast hi. P \equiv 0 \), but \( (lo \ast \theta. P) \equiv 0 \). (This is understandable since a \(*\)-composition of actions is a single action.)

ii) One can show easily however that if the synchronization operator \(*\) does not appear explicitly in \( P \), then \( P/O_L \) is bisimilar to the process obtained from \( P \) by replacing any occurrence of any high level action symbol \( hi \) in \( P \) by an occurrence of \( \theta \). In particular, \( P/O_L \) is (bisimilar to) the process term obtained by such a replacement in the *additive normal form* of \( P \).

iii) \( O_L \) is compatible with bisimulation (cf. the appendix in [2]).

**Definition 3.** A process \( P \) is said to satisfy *non-interference* (w.r.t. \( O_L \)) iff the following holds:

\[
\text{(RNI)} \quad P/O_L \preceq P
\]

We shall refer to this notion as \( \text{RNI} \) (*Robust Non-Interference*). The adjective *Robust* is meant to reflect the fact that it is based on exact – and not weak – simulation. We first give a characterization of \( \text{RNI} \) in terms of bisimulation.
Following usual convention, for any process $Q$ the process $Q \setminus H$ is that obtained from $Q$ by eliminating the branches of $Q$ where the action prefixes are in $H$: if $H = \{h_1, \ldots, h_n\}$, then $Q \setminus H = Q \setminus h_1 \setminus \ldots \setminus h_n$.

**Theorem 1.** A process $P$ satisfies RNI if and only if: $P \setminus H \approx P/\mathcal{O}_L$.

*Proof.* We first prove the ‘only if’ assertion, inductively. Assume that $P$ satisfies RNI, i.e. $P/\mathcal{O}_L \leq P$, and let $P' \setminus H \xrightarrow{a} P'' \setminus H$ be a transition on the process $P' \setminus H$ to some process $P'' \setminus H$. Then $x$ must be a $lo$; so we also have $P/\mathcal{O}_L \xrightarrow{a} P''/\mathcal{O}_L$, and by inductive hypothesis we have that $P'' \setminus H \approx P'/\mathcal{O}_L$. we deduce therefore that $P' \setminus H \leq P/\mathcal{O}_L$. On the other hand, if $z$ is any action such that we have a transition $P/\mathcal{O}_L \xrightarrow{a} Q$ to some $Q$, then by definition of $\mathcal{O}_L$, the action $z$ must be $lo$, so cannot be a $hi$; so we actually get a transition of the form $P' \setminus H \xrightarrow{a} Q$; we thus deduce that $P' \setminus H \geq P/\mathcal{O}_L$. In other words: $P' \setminus H \approx P/\mathcal{O}_L$.

The proof of the ‘if’ assertion is even more straightforward: consider any transition on $P/\mathcal{O}_L$ labeled by an action symbol $a$; then $a$ must be a $lo$, so there is a transition on $P\setminus H$ labeled by $a$; which actually must be a transition on $P$ with the same label. So $P/\mathcal{O}_L$ is simulated by $P$, that is to say $P$ satisfies RNI.

$\square$

**Remark 4.** The notion RNI of robust non-interference defined above is a strict generalization of the notion of ‘observation based non-deterministic non-interference’ developed in the literature, e.g. the BNNI of [7]. However, a process satisfying BNNI does not in general satisfy RNI, cf. the example below where $h$ is $hi$ and $a$ is $lo$, and $R$ is the null process.

In the process $P$ to the left a low level user can only get information $a$, but the time he spends to get this information from its two different branches is not the same: whereas in the process $P$ to the right, the low level user spends the same amount of time to retrieve the same information, whatever be the branch observed.
3.1 Application to Information Flow Analysis

A) In these applications, processes are more conveniently expressed in a so-called value-passing setup. Following Milner’s treatment of CCS such value-passing terms are reduced to the pure process algebra by a non-deterministic enumeration of possible values for the value-variables. We recall briefly this extended syntax.

An action in the value-passing syntax consists of either \( \theta \) or combined names like \( c(x) \) or \( \bar{c}(v) \) where \( c \) is any channel, \( v \) a value from an externally defined domain (e.g. integers) and \( x \) is a variable on such a domain, not to be confused with the upper-case process variable \( X \). The product of actions is defined so that any \( \bar{c}(v) \) is the conjugate of \( c(x) \): \( c(x) \bullet \bar{c}(v) = \theta \) (the channel names must be equal), \( c(x) \bullet \theta = \theta \bullet c(x) = c(x), \bar{c}(v) \bullet \theta = \theta \bullet \bar{c}(v) = \bar{c}(v), \theta \bullet \theta = \theta \); no other product is defined. The \((\text{Sync})\) transition rule defined above will get divided, in the value-passing setup, into two rules:

\[
\text{(Sync-comm)} \quad P \xrightarrow{\bar{c}(v)} P' Q \xrightarrow{c(x)} Q' \\
\text{and a symmetrical rule where } P \text{ is exchanged with } Q.
\]

\[
\text{(Sync-absorb)} \quad P \xrightarrow{\alpha} P' Q \xrightarrow{\beta} Q' \quad \alpha \circ \beta = \gamma
\]

where it is assumed that \( \alpha, \beta \notin \{(c(x), c(v)), (\bar{c}(v), c(x))\} \)

It will also be useful to have terms of the form \(" \text{let } x = e \text{ in } P^p \) which are defined as synonyms of \( P^p[x/e] \); the issue of defining expression \( e \)'s syntax and evaluating it to a value is assumed settled. Value-passing processes are easily compiled into standard processes, while preserving their semantics, (cf. e.g. \([8]\), \([2]\) for details).

B) We also need to slightly generalize the notion of \( RNI \) as follows, in the setup of value-passing CCS. Let \( X \) be any given set of channels. We then define an observation criterion w.r.t. \( X \), that we shall denote by \( O_X \), on processes along the same lines as we did earlier: all communication units along channels \( \text{not in } X \) are seen as \( \theta \). We may then define a process \( P \) to be \( RNI_X \) w.r.t. the set of channels \( X \) if and only if:

\[
(\text{RNI}_X) \quad P/O_X \approx P \setminus X^c
\]

where \( X^c \) is the set of all channels \( \text{not in } X \). The observation criterion \( O_L \) defined earlier is the special case where \( X \) is the set of all low level channels.

Definition 4. Let \( P, I \) be processes, \( C \) the set of all channels in \( P \), and assume that \( C \) is a strict subset of the channels in \( I \). Let \( X \) be the set of action channels of \( I \) not shared with \( P \), and \( O_X \) the corresponding observation criterion. Denote by \( P_I \) the process \((P \mid I) \setminus C \), where the intruder \( I \) is in asynchronous composition with \( P \), in forced synchronization on the shared channels \( C \).
Then $P$ is said to be **confidentiality preserving w.r.t.** $I$ iff $P_I$ satisfies the property $RNI_X$. Otherwise, $P$ is said to present an **information leak** for $I$.

C) See [2] for explicit examples of application of the notion of robust non-interference to cryptographic protocol analysis.

4 Equational Considerations

I) As we saw earlier, bisimulation in our setup can be decided by normalization under a convergent rewrite system. On the other hand, if we are given two levels of secrecy $Hi$ and $Lo$ and the observation criterion $O_L$ defined above, then for any process $P$ in which `s' does not appear explicitly (e.g. after normalization under the rewrite system of section 2.2), its low-level observable image $P/O_L$ can be obtained by applying the rules of the canonical rewrite system: $h.X \rightarrow \theta.X$, where $h$ is any high-level action. This rewrite system is also referred to as $O_L$.

II) A remark similar to 1) holds in the value passing setup, for the observation criterion $O_X$ w.r.t. any given set of channels $X$.

III) For any $\alpha \in \text{Act}^+$, define $P \xrightarrow{\alpha} Q$ to mean that there is a sequence $P \xrightarrow{a_1} \ldots \xrightarrow{a_n} Q$ such that $a_1 \ldots a_n = \alpha$; we shall write then $P = \alpha.Q$. Let $P$ be any process, and $\alpha, \beta \in \text{Act}^+$. And consider then the following condition ($\text{condn}$):

$$(P = \alpha.Q = \beta.Q) \implies \alpha.Q, \beta.Q$$

have the same normal form under the rewrite system $O_L$.

That is to say: any two confluent traces from $P$ must be word-equivalent w.r.t. $O_L$.

The following fact is not hard to check: If $P$ satisfies $RNI$, then $P$ satisfies condition (\text{condn}). Question: Is the converse true too?

IV) For given processes $P, I$, the task of determining if $P$ is confidentiality preserving w.r.t. $I$ (cf. definition in the previous section) can be performed by normalization w.r.t. the rewrite system formed by the $PACUID$ rules, plus the others representing the observation criterion $O_X$ where $X$ is the set of action channels of $I$ not shared with $P$.

If only the process $P$ is given, then the existence (or non-existence) of 'intruder' processes $I$ for which $P$ has information leaks, necessitates unification over process variables w.r.t. these equational theories. We hope to build such techniques by extending known $ACUID$-unification algorithms (cf. e.g. [3]).

V) Hennessy and Lin pointed out (cf. [9]) that the process algebra approach, in the value-passing setup, is not entirely suited for the analysis of infinite state systems. They proposed therefore the (extended) notions of **symbolic action** and **symbolic bisimulation** with such a purpose. (A symbolic action is in essence an action with variables which are to satisfy the conditions of a 'guard' for the action to be executable.) It seems possible to extend our synchronous formalism to such a symbolic setup. However, in the symbolic case the "$PACUID$-rewrite
rules” will no longer form a convergent rewrite system; security levels can still be attributed to channels, but deciding equivalence between processes w.r.t. any given observation criterion will be based on unification rather than on rewriting.

References

\textbf{\textalpha Prolog, a Fresh Approach to Logic Programming Modulo \textalpha-Equivalence}

James Cheney\(^1\) and Christian Urban\(^2\)

\(^1\) Cornell University  
Email: jcheney@cs.cornell.edu  
\(^2\) University of Cambridge  
Email: Christian.Urban@cl.cam.ac.uk

\textbf{Abstract.} \textalpha Prolog is a prototype logic programming language with a built-in notion of binders and unification modulo \textalpha-equivalence. It is based on a mild extension of first-order Horn formulae; instead of the usual first-order terms and first-order unification, \textalpha Prolog uses nominal terms and nominal unification introduced in [3]. In this paper, we give three examples that demonstrate the advantages of \textalpha Prolog and describe our current implementation.

\section{Introduction}

Logic programming is particularly suited for implementing inference rules defining relations over terms. Many interesting examples, however, involve terms with bound variables and \textalpha-equivalence. For example the typing relation for \textlambda-terms is often formulated with the inference rules

$$
\begin{align*}
\Gamma \vdash x : A &\quad \Gamma \vdash M : A \rightarrow B &\quad \Gamma \vdash N : A \\
\{x : A\} \cup \Gamma \vdash M : B
\end{align*}
$$

with the implicit side-condition that in \(\Gamma\) (a set of type assignments for variables) no variable has more than one assignment. This side-condition is violated if the third rule is applied bottom-up and \(\Gamma\) in the conclusion already contains an assignment for \(x\). In this case the bound variable \(x\) in \(\textlambda x. M\) needs to be renamed to a new \(x'\) so that a type assignment for \(x'\) can be safely added to \(\Gamma\). It is understood that this renaming does not affect the ‘meaning’ of the term, because terms are identified modulo \textalpha-equivalence.

However, this intuitive account of \textalpha-equivalence does not carry over to a correct declarative implementation in Prolog. For example, if we naively implement the \textlambda-abstraction typing rule with the clause

$$
type \text{Gamma} (1 \text{am X M}) (\text{arrow A B})::\text{type} (\text{pair X A}):\text{Gamma} M B,
$$

where \(X\) is a (logical) variable, then we get incorrect answers for \textlambda-terms such as \(\textlambda x. x\). For this untypeable term, the naive implementation erroneously returns two possible types: \(\tau \rightarrow (\tau \rightarrow \sigma) \rightarrow \sigma\) and \((\tau \rightarrow \sigma) \rightarrow \tau \rightarrow \sigma\). Although this problem can be fixed by judicious use of cut, first-order terms are unwieldy for implementing relations about syntax with binders correctly. The traditional solution for this problem is to use a higher-order logic programming language,
such as λProlog (see for example [1]), with higher-order terms. Such languages provide elegant mechanisms for dealing with binders, but they also introduce new problems: unification for higher-order terms is in general undecidable and if unifiers exist for a problem they may not be subsumed by a single most general unifier.

In this paper we shall present a logic programming language called αProlog in which one can implement the rules given above correctly as follows:

\[
\text{type } \text{Gamma (var } X \text{) } A \text{:: } \text{member (pair } X \text{ } A \text{) } \text{Gamma.}
\]

\[
\text{type } \text{Gamma (app } M \text{ } N \text{) } B \text{:: } \text{type } \text{Gamma } M \text{ } (\text{arrow } A \text{ } B \text{)}, \text{type } \text{Gamma } N \text{ } A \text{.}
\]

\[
\text{type } \text{Gamma (lam } x.M \text{ ) } (\text{arrow } A \text{ } B \text{) } / x\#\text{Gamma } :: \text{type } \text{(pair } x \text{ } A \text{) }::\text{Gamma } M \text{ } B \text{.}
\]

\[
\text{member } A \text{ } A :: \text{Tail.}
\]

\[
\text{member } A \text{ } B :: \text{Tail } :: \text{member } A \text{ } \text{Tail.}
\]

Two novel features of αProlog are illustrated by the third clause. First, λ-abstractions are translated to nominal terms which include an abstraction construct a t. Thus, (lam x.M) is intended to unify with any λ-abstraction, for example (lam x.var x), (lam y.app (var x) (var y)) and so on. Note that binders in nominal terms have concrete values that are not restricted to the scope of the abstraction. These values can be therefore used in the body of the clause, for example to add (pair x A) to the context Gamma. In contrast, in higher-order abstract syntax binders are anonymous and cannot escape their scope. So, in λProlog, a binder from the head cannot be used in the body of a clause.

Second, freshness constraints, here x#Gamma, can be attached to the heads of clauses; the freshness constraint x#Gamma ensures that the variable Gamma is not instantiated with a term containing x freely. Since the third clause is intended to implement the type inference rule for λ-abstractions, its operational behaviour is given by: choose fresh names for Gamma, x, M, A and B (this is standard in Prolog-like languages), unify the head of the clause with the goal formula, apply the resulting unifier to the body of the clause and make sure that Gamma is not substituted with a term that contains free occurrences of the fresh name we have chosen for x.

To calculate unifiers, αProlog employs the nominal unification algorithm of Urban et al. (see [3] for the details). Nominal unification retains the main advantages of first-order unification: all unification problems are decidable and solvable problems possess most general unifiers. Moreover, the nominal unification algorithm produces unifiers that make two terms equal modulo α-equivalence. For example if we unify (lam x.M), where M is a logical variable, with any term of the form (lam y.t)—t being the representation of an arbitrary λ-term—then M will be instantiated such that all occurrences bound by y in t are bound in M by x. This is achieved in the nominal unification algorithm by two means: First, bound names are renamed via a swapping operation. For instance applying the swapping (x y) to the term

\[
\text{lam } x. \text{app (lam } x. \text{var } x \text{)(app (var } y \text{)(var } z \text{))}
\]
\[ \text{lam } y. \text{app } (\text{lam } y. \text{var } y)(\text{app } (\text{var } x)(\text{var } z)) \]

Unlike renaming via substitution, swappings preserve \( \alpha \)-equivalence. For example, consider \( \text{lam } y. \text{var } x \) and \( \text{lam } z. \text{var } x \), which are \( \alpha \)-equivalent. Swapping \( x \) and \( y \) results in \( \text{lam } x. \text{var } y \) and \( \text{lam } z. \text{var } y \), which are still \( \alpha \)-equivalent, but na"ively substituting \( y \) for \( x \) results in variable capture in the first term.

Second, unification may generate freshness constraints that ensure that possible solutions respect \( \alpha \)-equivalence. For example if one wishes to solve the unification problem \((\text{lam } x.t) \equiv (\text{lam } x'.t')\), then one needs to solve

\[ t \equiv (x x') \cdot t' \text{ and } x \neq? t' \]

where the first problem asks whether \( t \) and \((x x') \cdot t'\) unify, where \((x x') \cdot t'\) is \( t' \) with all occurrences of \( x \) and \( x' \) swapped. The second problem asks whether \( x \) is fresh for \( t' \), i.e. \( t' \) cannot contain any free occurrence of \( x \). Posing the queries

(a) type nil (\( \text{lam } x. (\text{lam } x. (\text{var } x)) \)) \( T \) and
(b) type nil (\( \text{lam } x. (\text{lam } x. (\text{app } (\text{var } x) (\text{var } x))) \)) \( T' \)

to \( \alpha \text{Prolog} \) produces the unique answer \( T \rightarrow \text{arrow } A \ (\text{arrow } B \ B) \) for (a), and no answer for (b).

One important point in using nominal unification is that binders have explicit names. This has the pleasant consequence that manipulations involving terms with binders can be expressed very elegantly in \( \alpha \text{Prolog} \). Consider for example two of the transformations required for calculating a prenex-normal form of a formula

\[ \forall x. P \land \forall x. Q \implies \forall x. (P \land Q) \text{ and } (\forall x. P) \land Q \implies \forall x. (P \land Q) \]

where in the second transformation \( Q \) must not contain any free occurrence of \( x \). These transformations can be implemented as the following clauses.

\[ \text{prenex } (\text{and } (\text{forall x.P}) (\text{forall x.Q})) (\text{forall x.}(\text{and } P \ Q)). \]
\[ \text{prenex } (\text{and } (\text{forall x.P}) Q) (\text{forall x.}(\text{and } P \ Q)) / x \# Q. \]

In the first clause, the use of nominal unification allows, roughly speaking, to synchronize the two binders. In effect this clause is applicable for any term of the form \((\text{and } (\text{forall y} \ldots) (\text{forall z} \ldots))\). What is pleasant about \( \alpha \text{Prolog} \) is that in this clause no explicit renaming of the binders is required: it will be done implicitly by the swapping operation of the unification algorithm. In the second clause, the freshness constraint \( x \# Q \) makes sure that the side-condition placed upon the second transformation holds.
Another example is given below. It implements the capture-avoiding substitution of a variable in a λ-term.\(^1\)

\[
\text{id } X X.
\]
\[
\text{subst } (\text{var } X) X T T.
\]
\[
\text{subst } (\text{var } Y) X T (\text{var } Y) \leftarrow \text{not}(\text{id } X Y).
\]
\[
\text{subst } (\text{app } M N) X T (\text{app } M' N') \leftarrow \text{subst } M X T N', \quad \text{subst } N X T N'.
\]
\[
\text{subst } (\text{lam } y.M) X T (\text{lam } y.M') / y#T, y#X \leftarrow \text{subst } M X T M'.
\]

Given this program, posing the query subst (lam (x. var y)) y (var x) R yields the result R \(\rightarrow\) lam x#34.var x (the index 43 arises during proof search where clauses are `freshened`; see below).

## 2 Syntax

Figure 1 summarizes the abstract syntax of αProlog. In αProlog, the syntactic entities that can be bound and swapped are called \textit{atoms}, as opposed to (logical) \textit{variables}, which can be instantiated with other terms via substitution.

The terms include constants, function applications, tuples, atoms \textit{a} and atom-abstractions \textit{a.t}. Goal formulae include \textit{T} (true), conjunctions, disjunctions, and atomic propositions (i.e. predicates together with a term). The program clauses are of the form \textit{p t / \bigtriangleup :- G}, where \textit{G} is a goal, \textit{p t} is an atomic proposition, and \textit{\bigtriangleup} is a set of \textit{freshness constraints} \textit{a \# X} which must be satisfied in order for the rule to apply. If \textit{\bigtriangleup} is empty, we omit it and the \textit{\leftarrow} symbol; similarly, if \textit{G} is \textit{T}, we omit it and the \textit{\leftarrow} symbol.

As usual, clauses are implicitly universally quantified over all free logical variables \textit{X}. Variables can be used in place of atom occurrences but not in abstractions; i.e., \textit{var X} is a legal term whereas \textit{lam X.(var X)} is not. Instead, one needs to write in αProlog \textit{lam x.(var x)}, which will unify with any α-equivalent λ-term.

## 3 Proof search

The main differences between αProlog and standard logic programming languages are (1) it uses nominal unification to decide whether a goal and a clause head are unifiable, (2) it interprets freshness constraints in clauses as obligations to generate fresh atoms, and (3) it answers queries with a substitution plus a set of freshness constraints on some variables in the substitution. We shall present a nondeterministic transition semantics of proof search in αProlog.

A program state is a tuple \((\Psi, G, \bigtriangleup, \theta)\), where \(\Psi\) is a set of program clauses, \(G\) a goal, \(\bigtriangleup\) a set of freshness constraints, and \(\theta\) a substitution. The result of proof search is a pair \((\theta, \bigtriangleup)\), where \(\theta\) is a substitution instantiating some variables of \(G\) to terms, and \(\bigtriangleup\) is a set of freshness constraints of the form \(a \# X\).

\(^1\) \text{not is the usual negation-by-failure.}
Nominal terms: $t ::= c | f \ t_1 \cdots t_n | (t_1, ..., t_n) | X | a | a. t$

Goal formulas: $G ::= \top | G \land G | G \lor G | p \ t$

Program clauses: $C ::= p \ t / \nabla :\neg G$

Fig. 1. αProlog syntax summary

If $G$ is a non-atomic formula, then proof search is relatively standard, as shown below:

1. $(\Psi, \top, \theta, \nabla)$ succeeds with answer $(\theta, \nabla)$.
2. $(\Psi, G_1 \land G_2, \theta, \nabla)$ steps to $(\Psi, \theta^\ast G_2, \theta^\ast \circ \theta, \nabla^\ast)$, provided $(\Psi, G_1, \theta, \nabla)$ succeeds with answer $(\theta^\ast, \nabla^\ast)$.
3. $(\Psi, G_1 \lor G_2, \theta, \nabla)$ steps to $(\Psi, G_i, \theta, \nabla)$ for $i \in \{1, 2\}$.

Otherwise, if the goal formula is atomic (that is of the form $p \ t$), then:

1. Let $p \ t' / \nabla^\ast :\neg G^\ast$ be an element of $\Psi$.
2. Let $p \ t^\ast / \nabla^\ast :\neg G^\ast$ be the ‘freshened’ version of $p \ t' / \nabla^\ast :\neg G^\ast$, i.e. the clause where all atoms mentioned in $\nabla$ and all variables are renamed with new names (the latter is standard in logic programming).
3. If $t^\ast$ unifies with $t$ subject to the constraints $\nabla^\ast$, resulting in $(\theta^\ast, \nabla^\ast)$, then $(\Psi, p \ t, \theta, \nabla)$ steps to $(\Psi, \theta^\ast G^\ast, \theta^\ast \circ \theta, \nabla^\ast)$.

That is, to solve an atomic goal by resolving it with a clause, we first replace atoms mentioned in $\nabla$ with fresh atoms, then unify the goal and the head of the clause, while also checking that all the constraints are satisfied. Unification may produce new constraints that cannot be resolved yet; if so, these constraints are propagated through proof search until they can be resolved.

4 Conclusions

We have developed a proof-of-concept implementation of αProlog and are in the process of scaling it up to realistic examples. There are many directions for future work. For example, the complexity of nominal unification remains to be investigated. On the theoretical side, we intend to relate proof search in αProlog to a suitable notion of uniform proofs in Nominal Logic [2].

Acknowledgments: Dominik Wee implemented a preliminary version of αProlog as a thesis project.

References

Easy Intruder Deductions

Hubert Comon-Lundh and Ralf Treinen

Laboratoire Spécification et Vérification
CNRS UMR 8643, INRIA Futurs projet SECSI, and ENS de Cachan
61, Avenue du Président Wilson
F-94235 Cachan cedex, France
http://www.isv.ens-cachan.fr/~comon/~treinen

Abstract. We investigate extensions of the Dolev-Yao model by some algebraic properties of cryptographic primitives. We provide sufficient conditions under which the intruder deduction problem is decidable (resp. decidable in polynomial time). We apply this result to the equational theory of homomorphism, and show that in this case the intruder deduction problem is linear, provided that the messages are in normal form.

The automatic verification of cryptographic protocols has obtained a lot of attention these last years since several attacks were found on established protocols [CJ97,SPO]. There are, however, strong algorithmic limitations to protocol verification. The verification problem is undecidable, even with several restrictions [DLMS99,AC02,CC03] and it is actually co-recursively enumerable. Hence we are left either to search for attacks, in which case there are complete methods (e.g. bounding the number of sessions as in [RT01]), or to use abstractions. In the latter case, when the protocol is proved to be secure at the abstract level, then it is secure. However, a failure of proof search does not necessarily indicate that there is an attack.

A cornerstone of both approaches is the intruder deduction problem: Given a finite (or regular) set of messages $T$, and a given message $m$, is it possible to retrieve $m$ from $T$? This corresponds to the security decision problem in presence of a passive eavesdropper.

Most of the approaches rely on the so-called Dolev-Yao model of an intruder [DY83]. This model describes in particular the capabilities of an intruder to execute a passive attack against the security of a cryptographic protocol, that is her capabilities to obtain a secret knowledge by analyzing the messages that she has observed. In the classical model as proposed by Dolev and Yao, these capabilities are just to deduce new messages from observed ones by building or decomposing pairs, encryption, and decryption of a cipher under the condition that she knows the key. This model is justified by the perfect cryptography assumption, that is the assumption that there is no way to obtain knowledge about the plaintext pertaining to a ciphertext without knowing the key.

The perfect cryptographic assumption conveniently separates the problem of analyzing a concrete cryptographic protocol into a cryptography part, which belongs in general to the realm of mathematics, and a protocol part, which can be seen as a distributed algorithm and hence falls into the realm of computer science. This is a useful abstraction since it helps us concentrate on the algorithmic aspects of a cryptographic protocol. In fact, many attacks to cryptographic protocols have been found using the perfect cryptography assumption, in some cases even attacks against protocols in industrial use for years, see for instance [SPO].
It is, however, not always possible to separate completely the algorithmic aspects of a cryptographic protocol from the properties of the encryption method used. The reason is that components of commonly used encryption methods have algebraic properties that may interfere with algorithmic properties of a cryptographic protocol. Examples are the properties of the exclusive or and homomorphism properties of the RSA encryption algorithm. The attack described by Simmons [Sim94] against the original TMN key distribution algorithm [TMN89], for instance, exploits in a clever way both some of the laws of exclusive or and the homomorphism properties of RSA.

That is why there are some recent attempts to include some of these properties into the deduction engine [CLS03, CKRT03]. However, these papers only consider two particular algebraic properties: the properties of exclusive or, and the properties of Abelian groups.

In this paper we investigate for which classes of equational axioms the intruder model of Dolev and Yao can be extended such that the problem, whether an intruder can deduce a given secret from a given set of messages, is decidable. As an application, we show that the intruder deduction problem in presence of the homomorphism property

\[
\{\langle u, v \rangle \}_k = \{\langle u \rangle_k, \langle v \rangle_k\}
\]

is decidable in polynomial time.

The full paper is available as [CT03].

References


On the Computation of Joins for non
Associative Lambek Categorial Grammars

Annie Foret
IRISA and University of Rennes1
Campus de Beaulieu
35042 Rennes Cedex, France
Email: foret@irisa.fr
http://www.irisa.fr/prime/foret

Abstract. This paper deals with an application of unification and rewriting to Lambek categorial grammars used in the field of computational linguistics. Unification plays a crucial role in the acquisition of categorial grammar acquisition, as in [Kan98]; a modified unification has been proposed [For01a] in this context for Lambek categorial grammars, to give an account of their logical part. This modified unification (≡-unification) relies both on deduction (Lambek derivation) and on substitution; it is strongly related to the conjoinability relation [Lam58,Pen93] that is characterized by a free group equivalence and by a quasi-group one in the non-associative version. In view of grammatical inference, we also need to compute joins of the conjoinability relation, when they exist. This paper deals with this issue for the non-associative version of Lambek grammars, and provides an algorithm based on quasi-group rewriting.

1 Introduction

Categorial grammars, introduced in [BH33] and extended to Lambek grammars in [Lam58], have been studied in the field of natural language processing. Since they are completely lexicalized, they are well adapted to learning perspectives and an actual way of research is to determine the sub-classes of such grammars that remain learnable in the sense of Gold [Go67].

Recent works from [Kan98] and [Nie99] following [BP90] have answered the problem for different sub-classes of classical categorial grammars (we recall that the whole class of classical categorial grammars is equivalent to context free grammars; the same holds for the class of Lambek grammars [Pen93] that is thus not learnable in Gold’s model).

The extension of such results for Lambek grammars is an interesting challenge that is addressed by works on logic types or unlearnability results from [FL02].

In general grammatical inference uses unification and substitution. In the context of Lambek categorial grammars it seems appropriate to incorporate an operation on types based both on deduction (Lambek derivation) and on substitution instead of standard substitution and standard unification. Such an operation has been proposed and characterized in [For01a,For01b].

This modified unification is strongly related to the conjoinability relation [Lam58,Pen93] that is characterized by a free group equivalence and by a quasi-group one in the non-associative version.

In order to apply this operation, we also need to compute joins when they exist. This paper deals with this issue for the non-associative version of Lambek grammars, and provides an algorithm based on quasi-group rewriting.
The paper is organized as follows.

Section 2 gives background definition on categorial grammars. Section 3 addresses unification in the context of categorial grammar acquisition. Section 4 explains the conjoinability relation and its properties in L. Section 5 gives details on the conjoinability relation in NL, (the non-associative version) including its connection with quasi-groups. Section 6 gives some recalls from [For01a,For01b] about $\vdash$ - unification in L and NL and their connections with conjoinability. Section 7 describes our new contribution related to conjoinability in NL: an algorithm based on quasi-group rewriting for computing joins. Section 8 concludes.

2 Categorial Grammars

The reader not familiar with Lambek Calculus and its non-associative version will find nice presentation in the first one written by Lambek [Lam58,Lam61] or more recently in [Kan88,AT93,dG99].

The types $Tp$, or formulas, are generated from a set of primitive types $Pr$, or atomic formulas, by three binary connectives $\cdot / \otimes$ (over), $\otimes \setminus$ (under) and $\bullet^*$ (product): $Tp ::= Pr \mid Tp \setminus Tp \mid Tp / Tp \mid Tp \cdot Tp$. As a logical system, we use a Gentzen-style sequent presentation. A sequent $\Gamma \vdash A$ is composed of a sequence of formulas $\Gamma$ which is the antecedent configuration and a succedent formula $A$.

Let $\Sigma$ be a fixed alphabet. A categorial grammar over $\Sigma$ is a finite relation $G$ between $\Sigma$ and $Tp$. If $c, A \in G$, we say that $G$ assigns $A$ to $c$, and we write $G : c \mapsto A$.

**Lambek Derivation $\vdash L$.**

The relation $\vdash L$ is the smallest relation $\vdash$ between $Tp^+$ and $Tp$, such that for all $\Gamma;\Gamma' \in Tp^+, \Delta, \Delta' \in Tp^+$ and for all $A, B \in Tp$: see figure 1.

![Diagram of Lambek Derivation](image)

Fig. 1. L System
Non Associative Lambek Derivation \( \vdash_{NL} \)

In the Gentzen presentation, the derivability relation of \( NL \) holds between a term in \( S \) and a formula in \( Tp \), where the term language is \( S ::= Tp \cup (S, S) \). Terms in \( S \) are also called \( G \)-terms. A sequent is a pair \((\Gamma, A) \in S \times Tp\). The notation \( \Gamma[\Delta] \) represents a \( G \)-term with a distinguished occurrence of \( \Delta \) (with the same position in premise and conclusion of a rule). The relation \( \vdash_{NL} \) is the smallest relation \( \vdash \) between \( S \) and \( Tp \), such that for all \( \Gamma, \Delta \in S \) and for all \( A, B, C \in Tp \): see figure 2. We write \( NL_\emptyset \) for

\[
\begin{align*}
\Gamma \vdash A & \quad \Delta \vdash A \quad \text{Cut} \quad A \vdash A \\
\Gamma \vdash A & \quad \Delta[B] \vdash C \quad \text{/L} \quad \Gamma \vdash B / A \quad \text{/R} \\
\Delta[(B / A, \Gamma)] \vdash C & \quad \text{/L} \quad (A, \Gamma) \vdash B \quad \text{/R} \\
\Delta[(\Gamma, A \setminus B)] \vdash C & \quad \Delta[(A, B)] \vdash C \quad \text{\bullet L} \quad \Gamma \vdash A \quad \Delta \vdash B \quad \text{\bullet R} \\
\Delta[A \bullet B] \vdash C & \quad (\Gamma, \Delta) \vdash (A \bullet B) \quad \text{\bullet R}
\end{align*}
\]

Fig. 2. NL System

the Non associative Lambek calculus with empty antecedents (left part of the sequent).
We also refer to [Bus97, Mod97] for more details on \( NL \).

Note

[Cut Elimination]. We recall that the cut rule is admissible in \( \vdash_L \) and \( \vdash_{NL} \) : every derivable sequent has a cut -free derivation.

Language.

Let \( G \) be a categorical grammar over \( \Sigma \). \( G \) generates a string \( c_1 \ldots c_n \in \Sigma^+ \) iff there are types \( A_1, \ldots, A_n \in Tp \) such that \( : G : c_i \mapsto A_i \ (1 \leq i \leq n) \) and \( A_1, \ldots, A_n \vdash_L S \).

The language of \( G \), written \( L(G) \), is the set of strings generated by \( G \). We define similarly \( NL_G \) and \( NL_\emptyset(G) \) replacing \( \vdash_L \) by \( \vdash_{NL} \) and by \( \vdash_{NL_\emptyset} \) in the sequent where the types are parenthesized in some way.

Notation.

In some sections, we may write simply \( \vdash \) instead of \( \vdash_L \) or \( \vdash_{NL} \) or \( \vdash_{NL_\emptyset} \). We may simply write \( L(G) \) accordingly.
Rigid and $k$-valued Grammars.

Categorial grammars that assign at most $k$ types to each symbol in the alphabet are called $k$-valued grammars; 1-valued grammars are also called rigid grammars.

Example 1. Let $\Sigma_1 = \{\text{John}, \text{Mary}, \text{likes}\}$ and let $\text{Pr} = \{S,N\}$ for sentences and nouns respectively. Let $G_1 = \{\text{John} \rightarrow N, \text{Mary} \rightarrow N, \text{likes} \rightarrow N \setminus (S / N)\}$. We get $(\text{John likes Mary}) \in \mathcal{L}_{NL}(G_1)$ since $((N, N \setminus (S / N)), N) \vdash_{NL} S$. $G_1$ is a rigid (or 1-valued) grammar.

Note. Classical (or AB) categorial grammars, correspond to derivation rules ($\vdash_{AB}$) that are also valid in Lambek grammars; they are not detailed here.

2.1 Unification and grammatical inference

Substitutions. A substitution $\sigma$ is a function from variables in $\text{Var}$ to types in $\text{Tp}$ which is extended from types to types by:

\[
\begin{align*}
\sigma(p) &= p \text{ for } p \in \text{Pr} \quad \text{(with } \sigma(S) = S) \\
\sigma(A \setminus B) &= \sigma(A) \setminus \sigma(B) \quad \sigma(B / A) = \sigma(B) / \sigma(A) \\
\sigma(B \odot A) &= \sigma(B) \odot \sigma(A) \quad \text{1}
\end{align*}
\]

We extend this definition to G-terms and sequences by: $\sigma(\Gamma, \Delta) = (\sigma(\Gamma), \sigma(\Delta))$ and respectively by: $\sigma(A_1, \ldots, A_n) = \sigma(A_1), \ldots, \sigma(A_n)$

Note. The following principle (hereafter called the replacement principle) holds for $\vdash_{AB}$, $\vdash_L$ and $\vdash_{NL}$: if $\Gamma \vdash B$ then $\sigma(\Gamma) \vdash \sigma(B)$.

Substitutions extended to grammars. Given a substitution $\sigma$, and a grammar $G$, $\sigma(G)$ denotes the grammar obtained by applying $\sigma$ in the type assignments, that is:

$\sigma(G) = \{< c, \sigma(A) >; < c, A > \in \sigma G\}$

Preorders based on substitutions. Substitution allows to define several preorders as follows:

$\preceq$ on types defined by $A \preceq B$ iff $\exists \sigma : \sigma(A) = B$ (B is said an instance of A, or also A is said more general than B); $\preceq$ on substitutions defined by $\sigma \preceq \theta$ iff $\forall A \in \text{Tp} : \sigma(A) \preceq \theta(A)$; this yields $\sigma \preceq \theta$ iff $\exists \rho : \rho \sigma = \theta$ (where we write $\rho \sigma$ the composition of substitutions $\sigma$ with $\rho$ as: $\forall x : \rho \sigma(x) = \rho(\sigma(x))$); $\sigma$ is said to be more general than $\theta$;

$\sqsubseteq$ on grammars as follows: $G_1 \sqsubseteq G_2$ iff $\exists \sigma : \sigma(G_1) \subseteq G_2$ $\text{2}$ where we write $G_1 \subseteq G_2$ whenever $G_2$ contains all type assignments of $G_1$.

Note. It is easy to see (using the replacement principle) that if $G_1 \sqsubseteq G_2$ then $L(G_1) \subseteq L(G_2)$ (also for $L$ and $NL$). This fact is useful in the learning approach.

1 for a calculus including $\odot$

2 this is a simplified version: Kanazawa adds a faithfulness condition such that if two types assigned to the same symbol in $G_1$ are distinct they are kept distinct by $\sigma$ which has no impact in the rigid case.
Unification. A set of types $\mathcal{A}$ is said unifiable whenever there exists a substitution $\sigma$ such that $\sigma(A) = \sigma(B)$ for all $A, B \in \mathcal{A}$; $\sigma$ is then a unifier of the types $\mathcal{A}$. Let $U(\mathcal{A})$ denote the set of unifiers of $\mathcal{A}$. A principal unifier of $\mathcal{A}$ is a unifier $\sigma$, such that $\forall \theta, \in U(\mathcal{A}) : \sigma \preceq \theta$.

The usual unification problem is as follows: given two types to indicate whether these types are unifiable and if so to give a principal unifier. One important property is that when two types are unifiable, they admit a principal unifier that is unique modulo variable renaming.

Unification extended to rigid categorial AB-grammars and least upper bounds. A substitution $\sigma$ is said to unify a family $\mathcal{F}$ of sets of types, if $\sigma$ unifies each set in the family. A principal unifier $\sigma$ of $\mathcal{F}$ is a unifier of $\mathcal{F}$ such that for all unifier $\theta$ of $\mathcal{F}$: $\sigma \preceq \theta$.

Let us fix mgu a function that computes a principal unifier (undefined if there is none) for each set of types (or family of sets of types). Let $G_1$ and $G_2$ be rigid grammars with no common variables. We consider the family $\mathcal{F}$ of the sets $\mathcal{A}_c$ for each $c$ in the alphabet of $G_1$ or $G_2$. $\mathcal{A}_c = \{ A; c, A > \in G_1 \text{ or } c, A > \in G_2 \}$. We let $G_1 \cup G_2 = \text{mgu}(\mathcal{F})(G_1 \cup G_2)$. This operation computes the least upper bound of rigid grammars (with respect to $\subseteq$); it has properties of particular interest for the convergence of the learning algorithm.

3 Conjoinability

In this section we first recall useful definitions and properties on joins.

3.1 Conjoinability

We now define the conjoinability relation introduced in [Lam58]. This equivalence can be viewed as the special case of $\models$-unifiability (defined later) without variables.

Equivalence on types.
1. The join-equivalence, written $\sim$, is defined by:
   \[ t \sim t' \text{ iff } \exists t_1, \ldots, t_n : \forall i < n (t_i \vdash t_i+1 \text{ or } t_i+1 \vdash t_i) \text{ with } t = t_1, t' = t_n \]
2. We also define $\sim^{-1}$ by: $t \sim^{-1} t'$ iff $t \vdash t'$ or $t' \vdash t$
   The equivalence $\sim$ is thus the transitive closure of $\sim^{-1}$.
3. Types $t_1, t_2, \ldots, t_n$ are said conjoinable whenever there exists $t$ such that $t_i \vdash t$ (for all $i \leq n$). In this case $t$ is called a join for $t_1, t_2, \ldots, t_n$.

   We now recall some properties of conjoinability in the $\vdash_L$ case. The following result is due to Lambek.

Proposition 1 (Diamond property in $L$). Let $t_1$ and $t_2$ be two types. The following assertions are equivalent in $L$:
   (i) $t_1$ and $t_2$ are conjoinable ($\exists t : t_1 \vdash t$ and $t_2 \vdash t$)
   (ii) ($\exists t' : t' \vdash t_1$ and $t' \vdash t_2$)

Proof. We follow the version of [Pen93]
- if $t_i \vdash t$, for $i=1, 2$, we verify that $t' \vdash t_i$, for $i=1, 2$, where:
  \[ t' = (t_1 \mid t) \cdot (t \setminus t_2) \]
- if $t' \vdash t_i$, for $i=1, 2$, we verify that $t_i \vdash t$, for $i=1, 2$, where:
  \[ t = (t' \setminus t_1) \setminus t' / (t_2 \setminus t') \]
Proposition 2. Let $t_1$ and $t_2$ be two types. The assertion $t_1 \sim t_2$ in $L$ is also equivalent to (i) and (ii) of the diamond property above.

Free group interpretation.

Let $FG$ denote the free group with generators $Pr$, operation $\cdot$, and with neutral element $I$.

We associate with each formula $A$ an element in $FG$ written $[[A]]$ as follows:

$[[p]] = p$ for $p$ atomic

$[[A_1 \setminus A_2]] = ([[A_1]]^{-1} \cdot [A_2])$

$[[A_1 / A_2]] = ([[A_1]] \cdot ([[A_2]])^{-1}$

$[[A_1 \otimes A_2]] = [[A_1]] \cdot [[A_2]]$

We extend the notation to non-empty sequents by:

$[[A_1, A_2, \ldots, A_n]] = [[A_1]] \cdot [[A_2]] \cdot \ldots \cdot [[A_n]]$

The following known property states that such groups are models:

If $\Gamma \vdash L A$ then $[[\Gamma]] =_{FG} [[A]]$

Proposition 2 together with the following completeness result of Pentus is of particular interest for the investigations of $[[\vdash]]$-unification.

Theorem 1 (characterization of $\sim$ in $L$ by groups).

For any types $t$ and $t'$:

$t \sim t'$ in $L$ iff $[[t]] =_{FG} [[t']]$

4 Conjoinability and quasi-groups in NL

The join-equivalence in the non associative version of Lambek grammars enjoys similar properties; a similar characterization holds using quasi-groups instead of groups. We give the main properties (details may be found in [For01a]).

4.1 Conjoinability in NL

Next property is a central one. It has analogues in the associative case, in [Lam58] or in [Pen93] where the constructed types are different. In the following property we take the construction in [Lam58] that extends to the non-associative case (the simplified one in [Pen93] is not suitable, this is also observed in [Ver96]).

Proposition 3 (Diamond property in NL). Let $t_1$ and $t_2$ be two types. The following assertions are equivalent in $NL$:

(i) $t_1$ and $t_2$ are conjoinable $(\exists t : t_1 \vdash t$ and $t_2 \vdash t )$

(ii) $(\exists t' : t' \vdash t_1$ and $t' \vdash t_2 )$

Proof. We extend the version of Lambek to NL

- if $t_i \vdash t$, for $i=1,2$, we verify that $t' \vdash t_i$, for $i=1,2$, where:

$$t' = (t_1 / ((t / t) \setminus t)) \otimes (t / t) \setminus t_2$$

- if $t' \vdash t_i$, for $i=1,2$, we verify that $t_i \vdash t$, for $i=1,2$, where:

$$t = (t_1 \otimes (t' \setminus t')) / (t_2 \setminus (t \otimes (t' \setminus t')))$$

\(^3\) it involves different parenthesizing
Corollaries.
1. As a corollary in NL, we get that two types $t_1$ and $t_2$ are join-equivalent ($t_1 \sim t_2$) iff they have a join, that is (i) $\exists t : t_1 \vdash t$ and $t_2 \vdash t$ or equivalently iff (ii) $\exists t' : t' \vdash t_1$ and $t' \vdash t_2$.
2. Note also that a finite set of types has a join iff the types are pairwise conjoinable.

4.2 Quasi-groups

Quasi-groups are sets equipped with three binary operations $\cdot$, $/$, $\setminus$ that satisfy the following equations:

$x \cdot (x \setminus y) = y$
$(x \setminus y) \cdot y = x$
$x \setminus (x \cdot y) = y$
$(x \cdot y) / y = x$

The equational theory, here written QG, of quasi-groups admits a canonical rewriting system as found by Knuth and Bendix (70) and further studied in [Hu80].

The above four equations oriented from left to right are completed with two new rules to produce such a canonical rewrite system:

$(x / y) \setminus x \rightarrow y$
$(x \setminus y \setminus x) \rightarrow y$

QG-Unification. Solving equations in quasi-groups, or QG-unification admits a complete finite algorithm as shown by [Hu80], whose procedure is based on rewriting. A similar result holds for quasi-groups with identity [Hu80].

4.3 Quasi-groups as models

Quasi-groups are models for $\vdash_{NL}$. Our interest is to use them as (complete) models for conjoinability.

Free quasi-group. Let $FQG$ denote the free quasi-group with generators $Pr$, operations $\cdot$, $/$, $\setminus$.

From types to $FQG$. We associate with each type formula $A$ an element in $FQG$ written $[\![A]\!]$ as follows:

$[\![p]\!] = p$ for $p$ atomic
$[\![A_1 \setminus A_2]\!] = [\![A_1]\!] \setminus [\![A_2]\!]$
$[\![A_1 / A_2]\!] = [\![A_1]\!] / [\![A_2]\!]$
$[\![A_1 \odot A_2]\!] = [\![A_1]\!] \cdot [\![A_2]\!]$

We extend the notation to G-terms by:
$[\![\langle A_1, A_2 \rangle]\!] = [\![A_1]\!] \cdot [\![A_2]\!]$

Proposition 4 (Models). If $\Gamma \vdash_{NL} A$ then $[\![\Gamma]\!] =_{FQG} [\![A]\!]$

4.4 Characterizing NL-conjoinability by quasi-groups

NL-conjoinability and quasi-groups are related, in a way similar to [Pen93] for L-conjoinability and groups.

We recall that $QG$ also denotes the set of four equations defining its equational theory as in section 4.2.

Theorem 2 (characterization of $\sim$ in NL by quasi-groups).

For any types $t$ and $t'$ : $t \sim t'$ in NL iff $[\![t]\!] =_{FQG} [\![t']\!]$
5 \( \equiv \text{-unification in L and NL} \)

\( \equiv \text{-unification} \) has been investigated in [For01b] for Lambeke calculus. It strongly relies on the characterization of \( \sim \) by free groups in [Pen03] in the L case; in the NL case it involves similarly [For01a] the characterization of \( \sim \) by free quasi-groups.

We now give an overview on this operation.

We are first interested in the following relation on types:

**Definition 1.** The relation \( \equiv \) on types is defined by

\[ t_1 \equiv t_2 \text{ iff } \exists \sigma : t_1 \vdash \sigma(t_2) \]

where \( t_1, t_2 \) are types and \( \sigma \) is a substitution.

We now adapt the standard definitions of unification to deal with deduction.

**Definition 2.** Two types \( \mathcal{A}, \mathcal{B} \) are said \( \equiv \text{-unifiable} \) whenever (i) there exists a type \( t \) and a substitution \( \sigma \) such that \( t \vdash \sigma(\mathcal{A}) \) and \( t \vdash \sigma(\mathcal{B}) \); \( \sigma \) is then said a \( \equiv \text{-unifier} \) of \( \mathcal{A} \) and \( \mathcal{B} \) and \( t \) is a \( \equiv \text{-unificand} \) of \( \mathcal{A} \) and \( \mathcal{B} \).

These definitions are extended to sets of types in the usual way.

Two remarks:

1. If \( \sigma \) is a (standard) unifier of types \( \mathcal{A} \) and \( \mathcal{B} \) (ie \( \sigma(\mathcal{A}) = \sigma(\mathcal{B}) \)) then \( \sigma \) also is a \( \equiv \text{-unifier} \) of \( \mathcal{A} \) and \( \mathcal{B} \) (taking \( \sigma(\mathcal{A}) \) as left-hand side).
2. When \( \mathcal{A} \) and \( \mathcal{B} \) have no common variables, this definition (i) is equivalent to (ii) the existence of a type \( t \) such that both \( t \equiv \mathcal{A} \) and \( t \equiv \mathcal{B} \).

The case without shared variable is precisely the interesting one for the unification step in a learning process such as the RG-algorithm in [Kan98].

The following property relates \( \equiv \text{-unification in L to G-unification} \) (For01a):

**Proposition 5 (Characterizing \( \equiv \text{-unifiability and \( \equiv \text{-unifiers in L} \)).** Let \( \mathcal{A} \) and \( \mathcal{B} \) be two types in \( \text{T}_p \), L is such that:

1. \( \mathcal{A} \) and \( \mathcal{B} \) are \( \equiv \text{-unifiable} \) iff their images \( \llbracket \mathcal{A} \rrbracket \) and \( \llbracket \mathcal{B} \rrbracket \) are G-unifiable.
2. \( \sigma \) is a \( \equiv \text{-unifier} \) of types \( \mathcal{A} \) and \( \mathcal{B} \) iff its translation \( \sigma_G \) is a G-unifier of \( \llbracket \mathcal{A} \rrbracket \) and \( \llbracket \mathcal{B} \rrbracket \), (where \( \sigma_G \) is defined by : \( \sigma_G(p) = \llbracket \sigma(p) \rrbracket \) for p atomic); conversely any G-unifier of \( \llbracket \mathcal{A} \rrbracket \) and \( \llbracket \mathcal{B} \rrbracket \) is the translation of a \( \equiv \text{-unifier} \) of \( \mathcal{A} \) and \( \mathcal{B} \).

**Note.** A similar result holds for \( \text{T}_q \). We now formulate a similar result for NL, relying on quasi-groups instead of groups.

**Proposition 6 (Characterizing \( \equiv \text{-unifiability and \( \equiv \text{-unifiers in NL} \)).** Let \( \mathcal{A} \) and \( \mathcal{B} \) be two types in \( \text{T}_p \), NL is such that:

1. \( \mathcal{A} \) and \( \mathcal{B} \) are \( \equiv \text{-unifiable} \) iff their images \( \llbracket \mathcal{A} \rrbracket \) and \( \llbracket \mathcal{B} \rrbracket \) are QG-unifiable.
2. \( \sigma \) is a \( \equiv \text{-unifier} \) of types \( \mathcal{A} \) and \( \mathcal{B} \) iff its translation \( \sigma_{QG} \) is a QG-unifier of \( \llbracket \mathcal{A} \rrbracket \) and \( \llbracket \mathcal{B} \rrbracket \), (where \( \sigma_{QG} \) is defined by : \( \sigma_{QG}(p) = \llbracket \sigma(p) \rrbracket \) for p atomic); conversely any QG-unifier of \( \llbracket \mathcal{A} \rrbracket \) and \( \llbracket \mathcal{B} \rrbracket \) is the translation of a \( \equiv \text{-unifier} \) of \( \mathcal{A} \) and \( \mathcal{B} \).
Example 2. Consider the lifting rule that holds in the non-associative NL. Suppose that after some type computation, we get the following grammars (where $N$ denotes a constant, and $x_1, \ldots$ are variables):

\[
G_1 = \{ c_1 \mapsto x_1, c_2 \mapsto x_2, c_3 \mapsto (x_1 / N) \setminus x_2 \}\n\]
\[
G_2 = \{ c_1 \mapsto x_3, c_2 \mapsto x_4, c_3 \mapsto N \}\n\]
\[
G_3 = \{ c_1 \mapsto x_5, c_2 \mapsto x_6, c_3 \mapsto x_7 / (N \setminus x_6) \}\n\]

Standard unification fails to produce a rigid grammar since the types for $c_3$ are not unifiable. However $\models$-unification succeeds in $L$ and in NL since the types for $c_3$ are $\models$-unifiable:

\[
\models [x_1 / N \setminus x_1] =_{\text{FQC}} [N] =_{\text{FQC}} [x_1 / (N \setminus x_1)]
\]

Example 3. Consider the Geach rule that holds in the associative calculus but not in the non-associative NL:

\[
[[A / B]] =_{\text{FQC}} [[A]] / [[B]]
\]
\[
\models ((A / C) / (B / C)) =_{\text{FQC}} ([A] / [[C]]) / ([B] / [[C]])
\]
do not simplify in quasi-groups for atomic $A, B, C$. Whereas in a group interpretation they get reduced to the same formula $[[A]] /[[B]]^{-1}$.

6 Constructing Joins via polarized rewriting

In this section we propose a method for constructing joins in NL when such joins exist. Our method is based on the rewriting system and an observation on the rewriting steps from the derivability point of view.

6.1 Rewriting rules and $\vdash$ derivability

**Definition 3.** Let us call a rewrite rule $g \rightarrow d$ increasing whenever $g \vdash d$ and decreasing whenever $d \vdash g$.

We observe that all rules in QG (free quasi-groups) are either increasing ($r1, r2$) or decreasing (from $r3$ to $r6$):

\[
x, (x \setminus y) \vdash y
\]
\[
(x / y), y \vdash x
\]
\[
y \vdash x \setminus (x, y)
\]
\[
x \vdash (x, y) / y
\]
\[
y \vdash (x / y) \setminus x
\]
\[
y \vdash x / (y \setminus x)
\]

6.2 Polaries

We now define the positive and negative occurrences in a formula, as usual.

**Definition 4.** (Polaries in formulas and sequents). Each occurrence in a formula has a polarity either positive or negative; we define the positive ones inductively as follows, (the others being negative) : an atomic formula has itself as positive occurrence, no negative occurrence. In a formula $A = A_1 \setminus A_2$, or $A = A_2 / A_1$, the positive occurrences are the formula $A$ itself, the positive occurrences in $A_2$ and the negative occurrences in $A_1$. In a formula $A = A_1 \otimes A_2$, the positive occurrences are the formula $A$ itself, the positive occurrences in $A_1$ and $A_2$. 

In a sequent $A_1, \ldots, A_n \vdash C$ the polarity of an occurrence of $B$ is:
- if this occurrence is in $C$ then its polarity in $C$,
- else when this occurrence is in $A_i$ then the opposite of its polarity in $A_i$.

We extend the notion of polarities to rewriting steps.

Definition 5. A step of rewriting on a term $A$ is said positive when it applies at a positive occurrence of $A$, it is said negative otherwise.

6.3 Computing joins on a rewriting chain

Along the steps of rewriting we repeat the use of Lambek diamond construct shown in previous sections about conjoinability. By this we mean the following:

Definition 6 (diamond construct). We write $right(x, y, z)$ the right join resulting from Lambek diamond construct where $z$ is a left join of $x, y$; we write similarly $left(x, y, t)$ for the left join where $t$ is the right join. That is:

$$right(x, y, z) = (x \otimes (z \setminus z)) / (y \setminus (z \otimes (z \setminus z)))$$

$$left(x, y, t) = (x / (t')) \otimes ((t / t) \setminus y)$$ where $(w^w) = (w / u) \setminus w$

Let $A = A_1 \rightarrow A_2 \ldots \rightarrow A_n = A^*$ denote the successive terms in the normalization of $A$ by the rewrite system.

Proposition 7 (Construction on a rewriting chain). For each step $i$ of $A = A_1 \rightarrow A_2 \ldots \rightarrow A_n = A^*$, we define a left-join written $lj(A, A_i)$ and a right-join $rj(A, A_i)$ for $A$ and $A_i$, as follows:

1. $rj(A_1, A_1) := A_1$
2. $lj(A_1, A_1) := A_1$
3. if $A_i \rightarrow A_{i+1}$ is positive with an increasing rule, or negative with a decreasing rule, then $A_i \vdash A_{i+1}$ therefore we put $lj(A, A_{i+1}) = lj(A, A_i)$ and we compute $rj(A, A_{i+1})$ applying Lambek construct to $A, A_{i+1}, lj(A, A_{i+1})$, that is:
   3.1 $lj(A, A_{i+1}) := lj(A, A_i)$
   3.2 $rj(A, A_{i+1}) := right(A, A_{i+1}, lj(A, A_{i+1}))$
4. if $A_i \rightarrow A_{i+1}$ is positive with a decreasing rule, or negative with an increasing rule, then $A_{i+1} \vdash A_i$ therefore we put $rj(A, A_{i+1}) = rj(A, A_i)$ and we compute $lj(A, A_{i+1})$ applying Lambek diamond construct to $A, A_{i+1}, rj(A, A_{i+1})$, that is:
   4.1 $rj(A, A_{i+1}) := rj(A, A_i)$
   4.2 $lj(A, A_{i+1}) := left(A, A_{i+1}, rj(A, A_{i+1}))$

We get indeed from the above construction and details that:

$$lj(A, A_i) \vdash A \rightarrow rj(A, A_i)$$

$$lj(A, A_i) \vdash A_i \rightarrow rj(A, A_i)$$

in other words $lj(A, A_i)$ is a left join of $A$ and $A_i$ and $rj(A, A_i)$ is a right join of $A$ and $A_i$.

6.4 Computing joins of two terms along their rewriting chains

Suppose $A$ and $B$ are conjoinable, by the characterization property this is equivalent to $A^* = B^*$ by the rewriting system for quasi-groups.

Let $A = A_1 \rightarrow A_2 \ldots \rightarrow A_n = A^*$ and $B = B_1 \rightarrow B_2 \ldots \rightarrow B_m = B^*$ denote respective chains of normalization by the rewrite system.

We then get:
(1.1) \(lj(A, A_n) \vdash A \text{ and } A \vdash rj(A, A_n)\)
(1.2) \(lj(A, A_n) \vdash A_n = B_m \text{ and } A_n = B_m \vdash rj(A, A_n)\)
(2.1) \(lj(B, B_m) \vdash B \text{ and } B \vdash rj(B, B_m)\)
(2.2) \(lj(B, B_m) \vdash A_n = B_m \text{ and } A_n = B_m \vdash rj(B, B_m)\)

We observe from (1.2) and (2.2) that \(lj(A, A_n)\) and \(lj(B, B_m)\) have \(A_n = B_m\) as right join, we thus compute a left join written \(D\) applying Lambek diamond construct to these terms. Similarly \(rj(A, A_n)\) and \(rj(B, B_m)\) have \(A_n = B_m\) as left join, we thus compute a right join \(C\) applying Lambek diamond construct to these terms, that is

\[ D := \text{left}(lj(A, A_n), lj(B, B_m), A_n) \]
\[ C := \text{right}(rj(A, A_n), rj(B, B_m), B_m) \]

These terms \(C\) and \(D\) are such that:

(5.1) \(D \vdash lj(A, A_n) \vdash A\)
(5.2) \(D \vdash lj(B, B_m) \vdash B\)
(6.1) \(A \vdash rj(A, A_n) \vdash C\)
(6.2) \(B \vdash rj(B, B_m) \vdash C\)

that is \(C\) and \(D\) are respectively a right-join and a left-join of \(A\) and \(B\) as desired. We may summarize this as follows:

**Proposition 8.** Let \(A = A_1 \rightarrow A_2 \ldots \rightarrow A_n = A^i\) and \(B = B_1 \rightarrow B_2 \ldots \rightarrow B_m = B^j\) denote respective chains of normalization by the rewrite system. The following construct \(D := \text{left}(lj(A, A_n), lj(B, B_m), A_n)\) and \(C := \text{right}(rj(A, A_n), rj(B, B_m), B_m)\) define respectively a left-join and a right-join of \(A\) and \(B\)

### 6.5 Example (sketch)

We consider \(A\) and \(B\) as follows:

(a) \(A = A_1 = z.(z \setminus (x / y) \setminus x)\)
and \(B = B_1 = y\)

\(A\) yields rewriting steps, according to two deductions

- an increasing one:
  - (b1) \(A_1 = z.(z \setminus (x / y) \setminus x) \vdash \text{inc} (x / y) \setminus x = A_2\)

- followed by a decreasing one:
  - (b2) \(A_3 = y \vdash (x / y) \setminus x = A_2\)

We now compute the joins for \(A\):

(c1) \(rj(A_1, A_1) = lj(A_1, A_1) = A_1\)
(c2) \(lj(A_1, A_2) = lj(A_1, A_1) = A_1\)
(c2') \(rj(A_1, A_2) = \text{right}(A_1, A_2, A_1) = \ldots\)
(c3) \(rj(A_1, A_3) = rj(A_1, A_2)\)
(c3') \(lj(A_1, A_3) = \text{left}(A_1, A_3, rj(A_1, A_3))\)
(c3'') \(lj(A_1, A_3) = \text{left}(A_1, A_3, \text{right}(A_1, A_2, A_1))\)

as concerns \(B\), there is no rewriting steps

(d3) \(rj(B_1, B_1) = lj(B_1, B_1) = B_1 = y\)
we conclude with

(1) \( l_j(A_1, A_2) \) and \( l_j(B_1, B_1) \) have \( A_3 = B_1 = y \) as right-join
we may compute \( D = \text{left}(l_j(A_1, A_2), l_j(B_1, B_1), y) \)
\[ D = \text{left}(l_j(A_1, A_2), y, y) = (l_j(A_1, A_3) / (y^p)) \otimes (y^p) \]

(2) \( r_j(A_1, A_2) \) and \( r_j(B_1, B_1) \) have \( A_3 = B_1 = y \) as left-join
we may compute \( C = \text{right}(r_j(A_1, A_3), r_j(B_1, B_1), y) \)
\[ C = \text{right}(r_j(A_1, A_3), y, y) = (r_j(A_1, y) \otimes (y \setminus y)) / (y \setminus y) \]

(3) \( C \) is a right-join of \( A,B \) and \( D \) is a left-join of \( A,B \).

7 Conclusion

Unification plays a crucial role in the acquisition of categorial grammar, used in the
field of computational linguistics. After an overview of a modified unification
proposed in [For01a] in this context for Lambek categorial grammars and of an overview
the conjoinability relation [Lam58,Pen93] we have focused on the quasi-groups that
characterize the non-associative version. We have shown how to compute joins of the
conjoinability relation, when they exists, this method is based on quasi-group rewriting
while using “Lambek diamond construct”.

For future algorithmic developments in the context of Lambek categorial grammars,
in automatic acquisition applications or compacting grammar issues, we may thus hope
to benefit from results (such as on groups or quasi-groups), in the field of term rewriting
systems and unification.

References

[AT95] E. Aarts and K. Trautwein. Non-associative Lambek categorial grammar in

[BH53] Y. Bar-Hillel. A quasi arithmetical notation for syntactic description. Lan

[BP90] Wojciech Buszkowski and Gerald Penn. Categorial grammars determined

and ter Meulen [vBtM97], chapter 12, pages 683–736.

[dG99] Philippe de Groote. Non-associative Lambek calculus in polynomial time. In
8th Workshop on theorem proving with analytic tableaux and related meth
ods, number 1617 in Lecture Notes in Artificial Intelligence. Springer-Verlag,
March 1999.

[FL02] Annie Foret and Yannick Le Nir. Lambek rigid grammars are not learnable
from strings. In COLING 2002, 19th International Conference on Computa
tional Linguistics, Taipei, Taiwan, 2002.

[For01a] Annie Foret. Conjoinability and unification in Lambek categorial grammars.

Undecidability of Unification over Two Theories of Modular Exponentiation

Deepak Kapur*, Paliath Narendran**, and Lida Wang***

1 Department of Computer Science,
University of New Mexico,
Albuquerque, NM 87131, USA
Email: kapur@cs.unm.edu
2 Department of Computer Science,
SUNY at Albany,
Albany, NY 12222, USA
Email: dran@cs.albany.edu, lidawang@cs.albany.edu

Abstract. Modular multiplication and exponentiation are common operations in modern cryptography. Unification problems with respect to some equational theories that these operations satisfy are investigated. Two different but related equational theories in which the (one-sided) distributivity property of exponentiation over multiplication is assumed are analyzed. The unifiability problem is shown to be undecidable for these two theories.

1 Introduction

Modular arithmetic is the mainstay of many modern cryptographic algorithms. Arithmetical operations such as modular multiplication and exponentiation are part of several algorithms [10,11].

Consider the following set of axioms relating modular multiplication with modular exponentiation:

\[
\begin{align*}
x \cdot (y \cdot z) &= (x \cdot y) \cdot z & A \\
x \cdot y &= y \cdot x & C \\
x \cdot 1 &= x & U \\
x \cdot x^{-1} &= 1 & Inv \\
x^1 &= x & Exp1 \\
1^x &= 1 & Exp2 \\
(x \cdot y)^2 &= (x^2) \cdot (y^2) & Exp3 \\
(x^y)^2 &= (x^y)^2 & Exp4 \\
x^y \cdot z &= x^{2 \cdot y} & Exp5 \\
x \circ (y \circ z) &= (x \circ y) \circ z & A' \\
x \circ y &= y \circ x & C' \\
x \circ 1 &= x & U' \\
x \circ (x) &= 1 & Inv'
\end{align*}
\]

* Research supported in part by the NSF grant nos. CCR-0098114, the ONR grant no. N00014-01-1-0429, and a grant from the Computer Science Research Institute at Sandia National Labs.

** Research supported in part by NSF grant no. CCR-0098095 and ONR grant no. N00014-01-1-0430.

*** Research supported in part by NSF grant no. CCR-0098095.
The first four equations \((A, C, U, Inv)\) characterize an Abelian group with \(\cdot\) representing multiplication modulo a prime number \(p\); for brevity, this axiomatization is referred to by \(AG\). The last four equations characterize an Abelian group with \(\circ\) standing for multiplication modulo \(q\), where \(q\) is a prime factor of \(p-1\). Operators \(\cdot\) and \(\circ\) share the unit 1. Modular exponentiation is denoted by \(^{\cdot}\); \(x^{\circ} y\) is also written as \(x^y\).

Note also that \(Exp\circ\) implies a ‘kind of’ associativity for \(\circ\), since
\[
x^{(u^{(v^{(w))})})} = \text{Exp}\circ \left( (x^u)^v \right)^w = \text{Exp}\circ \left( (x^u)^{(v^{(w))})} \right).
\]

Our results in this paper are:

1. The unification problem for the equational theory consisting of \(AG\) and axioms
\((\text{Exp}1, \text{Exp}2, \text{Exp}3, \text{Exp}5, A', U')\) is undecidable. This theory is denoted as \(E_1\).

   Here \(\cdot\) forms an Abelian group, whereas \(\circ\) forms a monoid.
   This result was announced in [5] (where the theory was labelled as \(E_{3,5}\)) but the proof was omitted.

2. If one adds the axiom \(Inv'\) to the equational theory \(E_1\) the unification problem is still undecidable. This theory is denoted by \(E_2\).

Both proofs are by reduction from Hilbert’s 10th problem, which is on the solvability of polynomial equations over the integers (alternatively, over the natural numbers); Hilbert’s 10th is well-known to be undecidable. In both cases, it is shown that squaring of numbers can be simulated. Since multiplication can be obtained through squaring and addition/subtraction, this gives the undecidability of the unification problems. The proof of the second undecidability result is a bit more involved because of the existence of the inverse \(i\) for the \(\circ\) operator; there are more cases to be considered. Both the proofs are motivated by the undecidability proof in [9].

We denote by \(=_{AC}\) equivalence modulo the associativity and commutativity properties of \(\cdot\):
\[
\begin{align*}
x \cdot (y \cdot z) &= (x \cdot y) \cdot z & A \\
x \cdot y &= y \cdot x & C
\end{align*}
\]

**Notation:** The expression \(u^i\) denotes \(u \cdot u \ldots \cdot u\) if \(i\) is positive, and \(u^{-1} \cdot u^{-1} \ldots \cdot u^{-1}\) if \(i\) is negative; of course, if \(i = 0\), then \(u^0 = 1\). We also use + and the integers merely for abbreviation; e.g., \(u^{i+j}\) is the abbreviation of \(u^i \cdot u^j\), \(u^{2i}\) is the abbreviation of \(u^i \cdot u^i\), and \(u^{-i}\) stands for \((u^{-1})^i\). Thus, \(u^{2i-c}\) is \(u^i \cdot u^i \cdot (u^{-1})^c\).

Similarly, we use \(\bigcirc_i(u)\) to denote \(u \circ u \ldots \circ u\). \(\bigcirc_i(u) = 1\) if \(i = 0\). In Section 2 where \(E_1\) is considered, \(i\) cannot be negative, whereas in Section 3, where \(E_2\) is discussed, \(i\) can be negative.

We use the symbols \(\simeq\) and \(\simeq\) to abbreviate \(=_{E_1}\) and \(=_{E_2}\) respectively.

Symbols \(a, b\) and \(c\) are used as (distinct) free constants (different from 0 and 1) throughout the paper.

It is easy to verify that the equation \(x \circ b \simeq^{i} b \circ x\) forces \(x \simeq \bigcirc_k(b)\) for any \(k \geq 0\). Similarly, the equation \(x \circ b \simeq^{i} b \circ x\) forces \(x \simeq \bigcirc_k(b)\) for any \(k\) (including negative values). Thus proofs in Section 3 often have three subcases: (i) \(k\) is positive, (ii) \(k\) is 0, and (iii) \(k\) is negative.
2 Undecidability of Unification over $\mathcal{E}_1$

Theory $\mathcal{E}_1$ has the following AC-convergent system $\mathcal{R}_1$ (the precedence used is $\cdot$ $\triangleright$ $\cdot$ $\triangleright^{-1}$ $\cdot$ $\triangleright$):

\[
\begin{align*}
x \cdot 1 & \rightarrow x \\
x \cdot x^{-1} & \rightarrow 1 \\
x \cdot (x^{-1} \cdot y) & \rightarrow y \\
(x^{-1})^{-1} & \rightarrow x \\
1^{-1} & \rightarrow 1 \\
(x \cdot y)^{-1} & \rightarrow x^{-1} \cdot y^{-1} \\
x & \rightarrow x \\
1^x & \rightarrow 1 \\
(x \cdot y)^x & \rightarrow (x^x) \cdot (y^x) \\
(x^y)^x & \rightarrow x^{y^x} \\
(x^{-1})^y & \rightarrow (x^y)^{-1} \\
(x \circ y) \circ z & \rightarrow x \circ (y \circ z) \\
x \circ 1 & \rightarrow x \\
1 \circ x & \rightarrow x
\end{align*}
\]

We denote the normal form a term $t$ w.r.t. $\mathcal{R}_1$ by $nf(t)$.

**Definition 1.** A term $s$ is said to be a factor of a term $t$ if and only if both $s$ and $t$ are in normal form and $t \approx_{AC} s \cdot s'$ for some $s'$.

The following claims and lemmas are needed in a proof of the undecidability of the unification problem over this theory. Proofs of many of these are obvious, in which case, they are omitted.

Recall that in equations below $a, b, c$ are distinct free constants (about which nothing can be assumed).

**Claim 1.** Equation $y^b \approx y^c$ has only one solution $y \approx 1$.

**Claim 2.** Equation $y^b \cdot a^i \approx y^c$ forces $y \approx 1$ and $i = 0$.

**Claim 3.** Equation $x^b \approx y^c$ forces $x \approx y \approx 1$.

**Claim 4.** Equations $x^b \cdot w \approx y^c$ and $y^c \cdot w \approx y^c$ force $x \approx y \approx w \approx 1$.

*Proof.* If $x^b \cdot w \approx x$ and $y^c \cdot w \approx y$, then $x^b \cdot (y^{-1})^c \approx x \cdot y^{-1}$. This is impossible since no cancellation can occur between $x^b$ and $(y^{-1})^c \approx (y^c)^{-1}$.

**Claim 5.** For a nonnegative integer $i$, the equations $x^b \cdot w \cdot a^i \approx x \cdot y^c \cdot y$ force $x \approx y \approx w \approx 1$ and $i = 0$.

*Proof.* Terms $(x^b \cdot (y^{-1})^c) \cdot a^i \approx x \cdot y^c$. This is impossible since no cancellation can occur between $x^b$, $(y^{-1})^c \approx (y^c)^{-1}$, and $a^i$.

**Lemma 1.** $x^k \cdot a \approx y^k \cdot a^i$ iff there is $k \geq 0$, $x \approx a^k \cdot (b_1 + \cdots + b_m)$ and $z \approx 0$. And $z \approx 0$. 

\[x^k \cdot a \approx y^k \cdot a^i \iff x \approx a^k \cdot (b_1 + \cdots + b_m) \text{ and } z \approx 0.\]
Proof. The “if” part is simple, since
\[
\left( a^{O_{k-1}(b)+O_{k-2}(b)+\ldots+1} \right)^h \cdot a \cong a^{O_{k-1}(b)+O_{k-2}(b)+\ldots+1} \cdot a
\]
\[
\cong a^{O_{k}(b)+O_{k-1}(b)+\ldots+k+1} \cong a^{O_{k}(b)} \cdot x.
\]
To prove the “only if” part, assume that \( x^h \cdot a \cong x \cdot a^e \) has a solution \( \{ x \cong X, \ z \cong Z \} \). We can assume without loss of generality that the terms \( X \) and \( Z \) are in normal form. Unless \( Z \cong 1 \), \( a^{-1} \) must be a factor of \( X^h \cdot X^{-1} \). This implies that \( a \) is a factor of \( X \). Let \( k \geq 0 \) be the largest integer such that \( a^{O_{k}(b)} \) is a factor of \( X \). Then \( a^{O_{k}(b)} \) is a factor of \( X^h \cdot X^{-1} \), forcing \( Z \) to be \( O_{k+1}(b) \).

\[\square\]

Lemma 2. For \( v \cong a^i \) for some \( i \), \( y^h \cdot v \cong a^{O_{k}(b)+\ldots+k+1} \cdot y \) has only one solution, namely
\[\{ v \cong a^{h+1}, \ y \cong a^{O_{k}(b)+\ldots+k+1} \}.\]

Proof. It can be checked that the above is a solution. Assume there are two different solutions \( \{ y \cong Y_1, v \cong a^i \} \) and \( \{ y \cong Y_2, v \cong a^i \} \) for this equation. That is,
\[
Y_1^h \cdot a^i \cong a^{O_{k}(b)+\ldots+k+1} \cdot Y_1
\]
\[
Y_2^h \cdot a^i \cong a^{O_{k}(b)+\ldots+k+1} \cdot Y_2
\]
So \( (Y_1 \cdot Y_2^{-1})^h \cdot a^{(i-j)} \cong Y_1 \cdot Y_2^{-1} \). Using Claim 2, \( Y_1 \cong Y_2 \) and \( i - j = 0 \), causing a contradiction.
\[\square\]

Lemma 3. The following two equations have a solution iff \( k_1 = k_2 \) and \( v_1 \cong a^{k_1+1} \):
\[
y_1^h \cdot v_1 \cong a^{O_{k_1}(b)+\ldots+k_1+1} \cdot y_1,
\]
\[
y_2^h \cdot v_1 \cong a^{O_{k_2}(b)+\ldots+k_2+1} \cdot y_2.
\]

Proof. The proof is by contradiction. Suppose the claim is not true. Now
\[\{ v_1 \cong a^{k_1+1}, \ y_1 \cong a^{O_{k_1}(b)+\ldots+k_1+1} \}\]

is a solution for the first equation. Similarly,
\[\{ v_1 \cong a^{k_2+1}, \ y_2 \cong a^{O_{k_2}(b)+\ldots+k_2+1} \}\]

is a solution for the second equation. Without any loss of generality assume \( k_1 > k_2 \).
Let \( i = k_1 - k_2 \) and
\[\{ v_1 \cong \hat{v}_1, \ y_1 \cong \alpha_1, \ y_2 \cong \alpha_2 \}\]
be a solution for the equations where \( \hat{v}_1 \) is not of the required form. Let \( \beta_1 = a^{O_{k_1-1}(b)+\ldots+k_1+1} \) and \( \beta_2 = a^{O_{k_2-1}(b)+\ldots+k_2+1} \). Let \( v_1' = \hat{v}_1 \cdot (a^{k_1+1})^{-1} \). But then the equations
\[
w_1^h \cdot v_1' \cong w_1,
w_2^h \cdot v_1' \cdot a \cong w_2
\]
are solvable too, with
\[\{ w_1 \cong \alpha_1 \cdot \beta_1^{-1}, \ w_2 \cong \alpha_2 \cdot \beta_2^{-1} \},\]
but this is impossible because of Claim 5.
\[\square\]
Lemma 4. The following two equations force \( v_2 \approx a^{k(k-1)/2} \).

\[
\begin{align*}
z_1^b \cdot v_2 & \approx 2^b a^{(\sum_{i=1}^{k-1} i + \sum_{i=1}^{k-1} (i-1) + k)} z_1, \\
z_2^c \cdot v_2 & \approx 2^c a^{(\sum_{i=1}^{k-1} i + \sum_{i=1}^{k-1} (i-1) + k)} z_2.
\end{align*}
\]

The proof of this lemma is very similar to that of Lemma 3.

Lemma 5. The following two equations force \( w_2 \approx a^b \).

\[
\begin{align*}
x_3^b \cdot a & \approx 2^b x_3 \cdot a^{w_2}, \\
y_3^b \cdot a & \approx 2^b x_3 \cdot y_3.
\end{align*}
\]

Proof. By Lemma 1, there must be an \( i \) such that \( w_2 \approx a^{i(b)} \) and \( x_3 \approx a^{(\sum_{i=1}^{k-1} i + \sum_{i=1}^{k-1} (i-1) + k)} \). According to Lemma 2, \( k \) should be equal to \( i \), thus forcing \( w_2 \) to be \( a^{i(b)} \). \( \square \)

Theorem 1. Unification over \( \mathcal{E}_1 \) is undecidable.

Proof. The reduction is from Hilbert’s 10th problem, which is to solve a multivariate polynomial over the integers (or natural numbers). It is necessary to simulate numbers as well as multiplication over numbers. Using the algebraic relation \( i \times j = ((i + j)^2 - (i - j)^2)/4 \), if we can simulate \( i^2 \), multiplication is obtained. Given \( \mathcal{O}_2(b) \), the following set of equations can be used to force \( w_2 \) to be \( \mathcal{O}_2(b) \), thus simulating the squaring of numbers.

1. \( x_1^b \cdot a \approx x_1 \cdot a^{z_1} \)
2. \( x_2^c \cdot a \approx x_2 \cdot a^{y_1} \)
3. \( y_1^b \cdot v_1 \approx x_1 \cdot y_1 \)
4. \( y_2^c \cdot v_1 \approx x_2 \cdot y_2 \)
5. \( z_1^b \cdot v_2 \approx y_1 \cdot z_1 \)
6. \( z_2^c \cdot v_2 \approx y_2 \cdot z_2 \)
7. \( s_1 \approx v_2 \cdot v_2 \)
8. \( s_2 \approx s_1 \cdot v_1 \)
9. \( x_3^b \cdot a \approx x_3 \cdot a^{w_2} \)
10. \( y_3^b \cdot s_2 \approx x_3 \cdot y_3 \)

According to Lemma 1, the first and second equations force \( x_1 \approx a^{\mathcal{O}_{k-1}(b)} \cdot a^{\mathcal{O}_{k-2}(b)} \cdot \ldots \cdot a \cdot z_1 = a^{\mathcal{O}_k(b)} \); \( x_2 \approx a^{\mathcal{O}_{k-1}(c)} \cdot a^{\mathcal{O}_{k-2}(c)} \cdot \ldots \cdot a \cdot z_2 = a^{\mathcal{O}_k(c)} \). According to Lemma 3, the third and fourth equations together force \( v_1 \) to be \( a^k \) as well as \( k_2 = k \). In a similar fashion, the fifth and sixth equations force \( v_2 \) to be \( a^{k(k-1)/2} \). It is easy to see that the seventh and eighth equations make \( s_1 \) and \( s_2 \) to be \( a^{k(k-1)/2} \) and \( a^{k^2} \), respectively. Finally, by Lemma 5, the ninth and tenth equations force \( w_2 \) to be \( \mathcal{O}_2(b) \). \( \square \)
3 Undecidability of Unifiability Check over $E_2$

Theory $E_2$ has the following AC-convergent system $R_2$:

- $x \cdot x^{-1} \rightarrow 1$
- $x \cdot 1 \rightarrow 1$
- $(x \cdot y)^{-1} \rightarrow x^{-1} \cdot y^{-1}$
- $(z^{-1})^{-1} \rightarrow z$
- $x^1 \rightarrow x$
- $1^{-1} \rightarrow 1$
- $1^z \rightarrow 1$
- $(x^{-1})^y \rightarrow (x^y)^{-1}$
- $(x^y)^z \rightarrow x^{y \cdot z}$
- $(x \circ y) \circ z \rightarrow x \circ (y \circ z)$
- $x \circ 1 \rightarrow x$
- $1 \circ x \rightarrow x$
- $x \circ i(x) \rightarrow 1$
- $i(x) \circ x \rightarrow 1$
- $x \circ (i(x) \circ z) \rightarrow z$
- $i(x) \circ (x \circ z) \rightarrow z$
- $i(i(x)) \rightarrow x$
- $i(1) \rightarrow 1$
- $i(x \circ y) \rightarrow i(y) \circ i(x)$

Let $\Sigma = \{ \cdot, \circ, ^{-1}, 1, \circ, i \}$, the signature of $E_2$. Let $n_f(t)$ denote the normal form of $t$ w.r.t. $R_2$.

Recall again that by our abbreviation, the following hold:

- $a^{-\bigcirc_a(b)} \simeq (a^b)^{x(b) \circ x(b)}^{-1}$
- $a^{-\bigcirc_2(b) + \bigcirc_2(b)} \simeq (a^{b \cdot b})^{-1} \cdot (a^{x(b) \cdot x(b)})^{-1}$
- $a^{-\bigcirc_2(b) \cdot \bigcirc_2(b)} \simeq (a^{b \cdot b})^{-1} \cdot (a^{x(b) \cdot x(b)})^{-1}$

The undecidability proof of the unification problem over $E_2$ is patterned after the proof given in the previous section for the unification problem over $E_1$. Hilbert’s $10^{th}$ problem is reduced to this unification problem as well. To simulate squaring, we have to go through the same series of steps as in the previous section. Due to the presence of $i$ with respect to $\circ$, additional cases have to be considered and cancellation can occur. Below, we reproduce all the steps from the proof given in the previous section, focusing on the changes which are needed.

**Definition 2.** We call a ground term $T$ on $\Sigma$ and free constants atomic iff $T$ is in normal form, $T \neq 1$, and $T$ is of the form $X^T; s_0; s_T$ or $\langle X^T; s_0; s_T \rangle^{-1}$ ($n$ can be 0) where the root symbol for $X$ is not $\cdot$ or $\circ$ and the root symbol of $T_i$ ($1 \leq i \leq n$) is not $\circ$. Also we define $op_{n,m}(T)$ as the number of occurrences of the operator $\circ$ in $T$. The power of $T$ is defined as $T^i$ if $n > 0$, and 1 if $n = 0$. $S$ is a factor of $T$ if $S$ is atomic and either $T = S$ or $T = S \cdot S_1 \cdots \cdot S_m$ for some atomic terms $S_1, \ldots, S_m$, where $S_i \neq S^{-1}$ for $1 \leq i \leq m$. (Alternatively, $n_f(T) =_{AC} S \cdot S_1 \cdots \cdot S_m$ for some atomic terms $S_1, \ldots, S_m$.)
For example, if $T = a^{(b,c)}$, then $T$ is atomic with power $c$ and $op_{num}(T) = 1$. Note also that the normal form of any term $S$ w.r.t. $R_2$ is either 1 or of the form $S_1 \cdot \ldots \cdot S_m$ where each $S_i (1 \leq i \leq m)$ is atomic.

**Definition 3.** For any two atomic terms $S, T$, we define $S \succ T$ if $op_{num}(S) > op_{num}(T)$.

According to the above definition, we have $e^{(a+b)} \succ e^a$ whereas $e^a$ and $(e^a)^{-1}$ are incomparable w.r.t. $\succ$.

**Claim 6.** For any two atomic terms $S, T$, $S \succ T$ iff $nf(S^{-1}) \succ nf(T^{-1})$.

**Claim 7.** Equation $y^b \simeq y$ has only one solution $y \simeq 1$.

**Claim 8.** Let $y$ be any ground term and $T$ be a factor of $nf(y^b \cdot y^{-1})$. Then there is an $n > 0$ such that either $nf(T^{\emptyset})(b)^{-1} = 0$ or $nf(T^{\emptyset})(b)^{-1}$ is a factor of $nf(y^b \cdot y^{-1})$.

*Proof.* Let $\alpha = nf(y^b \cdot y^{-1})$. It must be that either (a) $nf(T^{-1})$ is a factor of $y$, or (b) $nf(T^{\emptyset})(b)$ is a factor of $y$. Let us consider case (a) first. It must be that $nf(T^{\emptyset})(b)$ is either a factor of $\alpha$ or of $y$. If the former is the case, then we are done. If not, let $k$ be the largest integer such that $nf(T^{\emptyset}(b)^{-1})$ is a factor of $y$. But then $nf(T^{\emptyset}(b)^{-1})$ has to be a factor of $\alpha$.

For case (b), if $nf(T^{\emptyset}(b)^{-1})$ is a factor of $\alpha$ then we are done. Otherwise let $k$ be the largest integer such that $nf(T^{\emptyset}(b)^{-1})$ is a factor of $y$. Then $nf(T^{\emptyset}(b)^{-1})$ has to be a factor of $\alpha$ since it cannot be cancelled by any term in $nf(y^b)$.

**Corollary 1.** Equation $y^b \cdot a^{x} \simeq y$ forces $y \simeq 1$.

**Corollary 2.** Let $y$ be any ground term and $T$ be a maximal (w.r.t. $\succ$) factor of $nf(y^b \cdot y^{-1})$. Then the power of $T$ is either $b$ or $i(b)$.

**Lemma 6.** $x^b \cdot a \simeq x^a$ if one of the following holds:

(a) $k > 0$, $z \simeq 0_{1}(b)$ and $x \simeq a^{0_{k-1}(b) + 0_{k-2}(b) + \ldots + 1}$. 

(b) $k = 0$, $z \simeq 0_{0}(b) \simeq 1$ and $x \simeq 1$.

(c) $k < 0$, $z \simeq 0_{1}(b)$ and $x \simeq (a^{0_{k}(b) + 0_{k+1}(b) + \ldots + 0_{k-1}(b))^{-1}}$.

*Proof.* We first prove the "if" part. For case (b), it is quite obvious that when $k = 0$, \{ $z \simeq 1, z \simeq 1$\} is a solution. For case (a), we have

$x^b \cdot a \simeq (a^{0_{k-1}(b) + \ldots + 1})^b \cdot a$

\begin{align*}
&\simeq a^{0_{k}(b) + \ldots + b} \cdot a \\
&\simeq a^{0_{k}(b) + \ldots + 1}
\end{align*}

and

$x \cdot a^z \simeq a^{0_{k-1}(b) + \ldots + 1} \cdot a^{0_{k}(b)}$

\begin{align*}
&\simeq a^{0_{k}(b) + \ldots + 1}
\end{align*}
For case (c), we have
\[ x^b \cdot a \simeq (a^{O_1(b)} + O_{k_1}(b) + \ldots + O_{-i}(b))^{-1} \cdot a \]
\[ \simeq (a^{O_{k_1}(b)} + O_{k_2}(b) + \ldots + O_{-j}(b))^{-1} \]
\[ \Rightarrow (a^{O_{k_1}(b)} + O_{k_2}(b) + \ldots + O_{-j}(b))^{-1} \]
and
\[ x \cdot a^{-i} \simeq (a^{O_i(b)} + O_{k_1}(b) + \ldots + O_{-j}(b))^{-1} \cdot a^{O_i(b)} \]
\[ \Rightarrow (a^{O_i(b)} + O_{k_2}(b) + \ldots + O_{-j}(b))^{-1} \]

To prove the “only if” part, assume equation \( x^b \cdot a \simeq x \cdot a^z \) has a solution \( \{ x \simeq z, z \simeq Z \} \) where \( Z \simeq O_k(b) \) for any \( k \). Thus \( X^b \cdot X^{-1} \simeq a^x \cdot a^{-1} \). We can assume without loss of generality that the terms \( X \) and \( Z \) are in normal form. \( a^{-1} \) must clearly be a factor of \( X^b \cdot X^{-1} \), since \( Z \neq 1 \). By Claim 8, there must be \( n > 0 \) such that either \( n f(a^{O_n(b)}) \) or \( n f(a^{O_{-n}(b)}) \) is also a factor of \( X^b \cdot X^{-1} \). Since \( a^z \) is the only other factor of \( X^b \cdot X^{-1} \), this leads us to a contradiction.

It is now not hard to show that given \( Z \simeq O_k(b) \) for some \( k \), \( X \) has to be either \( a^{O_{k-1}(b)} + O_{k-2}(b) + \ldots + O_{-j}(b) \) where \( k > 0 \), or \( (a^{O_{k}(b)} + O_{j}(b) + \ldots + O_{-j}(b))^{-1} \) when \( k < 0 \). (Claim 7 is used here, since if there is another solution \( \{ x \simeq U \} \) for the same \( Z \), then \( (U \cdot X^{-1})^b \approx U \cdot X^{-1} \).)

**Example 1.** Consider the equation
\[ x^b \cdot a \simeq x \cdot z. \]
\[ \{ x \simeq a^{-O_{-n}(b)} + O_{-n}(b) \} \]
is a solution, since
\[ x^b \cdot a \simeq a^{-O_{-n}(b) - O_{-n}(b)} \cdot a \]
\[ \simeq a^{-O_{-n}(b) - O_{-n}(b)}. \]
The right-hand side is \( a^{-O_{-n}(b) - O_{-n}(b) - O_{-n}(b)} \cdot a^{O_{-n}(b)} \). Now \( a^{-O_{-n}(b)} = (a^{O_{-n}(b)})^{-1} \) cancels with \( a^{O_{-n}(b)}. \)

**Lemma 7.** Let \( v \simeq a^i \) for some \( i \) and \( w \simeq O_k(b) \) for some \( k \). Then the equations
\[ x^b \cdot a \simeq x \cdot a^w \]
\[ y^b \cdot v \simeq y \cdot y \]
have a solution if and only if \( k = i. \)

**Lemma 8.** The following equations have a solution if and only if \( z_1 = O_{k_1}(b), z_2 = O_{k_2}(c), \)
\[ k_1 = k_2 \] and \( v_1 \simeq a^{k_1}: \]
\[ x^b \cdot a \simeq x_1 \cdot a^{x_2}, \]
\[ y^b \cdot v_1 \simeq y_1 \cdot y_2. \]

**Proof.** Consider the equations:
\[ x^b \cdot a \simeq x_1 \cdot a^{x_2}, \]
\[ y^b \cdot v_1 \simeq y_1 \cdot y_2. \]
The "if" part is easy. We prove the "only if" part. Assume \( \{x_1 = X_1, x_2 = X_3, y_1 = Y_1, y_2 = Y_3, v_1 = V_1\} \) is a solution. We first prove that \( V_1 \) is of the form \( a^i \), \( X_1 \simeq a^{O_{k_i-1}(b) + O_{k_{i+1}}(b) + \cdots + O_{k_i}(b)} \) or \( X_1 \simeq (a^{O_{k_i}(b) + O_{k_{i+1}}(b) + \cdots + O_{k_i}(b)})^{-1} \). From the third equation, we have \( V_1^{-1} \), \( Y_1^{-1} \simeq X_1 \cdot V_1^{-1} \).

Let \( S \) be a maximal factor of \( V_1^{-1} \) such that \( S \not\simeq a^{-1} \). If \( S \) is canceled by a factor of \( X_1 \), then \( b \) (or \( i(b) \) if \( k_1 < 0 \)) is a power of \( S \). If \( S \) is not canceled, by Claim 8, there is an \( n > 0 \) such that either \( n f((S^{O_a(b)})^{-1}) \) or \( n f((S^{O_a(h)})^{-1}) \) is a factor of \( n f(Y_1^b \cdot Y_1^{-1}) \). Now this term is either a factor of \( V_1^{-1} \) or a factor of \( X_1 \). In either case, the power of \( S \) must be \( b \) or \( i(b) \).

By a similar argument using the second and fourth equations, we can deduce that the power of \( S \) is either \( c \) or \( i(c) \). We thus have \( S \) being a factor of \( V_1^{-1} \) but the power of \( S \) being both \( b \) (or \( i(b) \)) and \( c \) (or \( i(c) \)), a contradiction.

Since \( V_1 = a^i \), according to Lemma 7, we have \( i = k_1, k_1 = k_2 \).

The next two lemmas are easy to prove.

**Lemma 9.** For \( k > 0 \), the following two equations force \( v_2 \simeq a^{k(k+1)/2} \).

\[
\begin{align*}
\text{z}_1^b \cdot v_2 & \simeq a^{O_{k_i-1}(b) + O_{k_i}(b) + \cdots + O_{k_i}(b) + k}, z_1, \\
\text{z}_2^c \cdot v_2 & \simeq a^{O_{k_i}(c) + O_{k_i+1}(c) + \cdots + O_{k_i}(c) + k}, z_2.
\end{align*}
\]

A similar lemma can also be proved for the case when \( k \) is negative.

**Lemma 10.** For \( k < 0 \), the following two equations force \( v_2 \simeq a^{k(k+1)/2} \).

\[
\begin{align*}
\text{z}_1^b \cdot v_2 & \simeq a^{O_{k_i}(b) + O_{k_i+1}(b) + \cdots + O_{k_i}(b) + k}, z_1, \\
\text{z}_2^c \cdot v_2 & \simeq a^{O_{k_i}(c) + O_{k_i+1}(c) + \cdots + O_{k_i}(c) + k}, z_2.
\end{align*}
\]

Putting all these together,

**Lemma 11.** Consider the equations

\[
\begin{align*}
\text{x}_1^b \cdot a & \simeq a \cdot O_{k_i}(b), \\
\text{x}_2^c \cdot a & \simeq a \cdot O_{k_i}(c), \\
\text{y}_1^b \cdot a & \simeq x_1 \cdot y_1, \\
\text{y}_2^c \cdot a & \simeq x_2 \cdot y_2, \\
\text{z}_1^b \cdot v & \simeq y_1 \cdot z_1, \\
\text{z}_2^c \cdot v & \simeq y_2 \cdot z_2.
\end{align*}
\]

If \( k > 0 \), then \( v \simeq a^{k(k+1)/2} \) and if \( k < 0 \), then \( v \simeq a^{k(k+1)/2} \).

We can take advantage of this asymmetry to achieve squaring over the integers in the following way. Let \( w \simeq O_{k_i}(b) \) for some \( k \). Since we don’t know whether \( k \) is negative or positive, we can create another variable \( w' \), and form the equation \( w \circ w' \simeq 1 \). Thus
$w' \simeq \bigcirc_{\omega_k}(b)$. Now form the equations

\begin{align*}
  w_1 \circ c & \simeq' c \circ w_1, \\
  x_1^b \cdot a & \simeq' x_1 \cdot a^w, \\
  x_2^c \cdot a & \simeq' x_2 \cdot a^{w_1}, \\
  y_1^d \cdot v_1 & \simeq' x_1 \cdot y_1, \\
  y_2^e \cdot v_1 & \simeq' x_2 \cdot y_2, \\
  z_1^f \cdot v_2 & \simeq' y_1 \cdot z_1, \\
  z_2^g \cdot v_2 & \simeq' y_2 \cdot z_2,
\end{align*}

for $w$, and similar equations for $w'$ using variables $w'_1, x'_1, x'_2, y'_1, y'_2, v'_1$ and $v'_2$ respectively. Now $v \simeq v_2 \cdot v'_2$ forces $v \simeq a^{w'}$.

By Lemma 7, the following equations will force $z \simeq \bigcirc_{\omega_k}(b)$:

\begin{align*}
  x^b \cdot a & \simeq' x \cdot a^z, \\
  y^b \cdot v & \simeq' x \cdot y.
\end{align*}

**Theorem 2.** Unification over $E_2$ is undecidable.

**Proof.** Hilbert's 10th problem can be reduced to our unification problem since we are able to simulate squaring using the above lemmas. \qed

### 4 Conclusion

Unification problems for theories that specify multiplication and exponentiation modulo a prime are discussed. As shown above, for two theories that admit the property $(x^y)^z = x^{y \cdot z}$, unifiability check turns out to be undecidable. However, if this property is excluded, then unification algorithms for various restricted theories, depending upon properties of exponentiation used, can be obtained [8,4,5].

## References


Applying Unification Techniques to XML Document Management?

Michael Kohlhase

School of Computer Science, Carnegie Mellon University, Email: kohlhase@cs.cmu.edu

1 Introduction

the “eXtensible Markup Language” XML has changed the way we think about and work with data on the Internet. XML is a framework [BPSM97,Har01] for Internet markup languages, which are called “XML applications” in this framework. In the last 5 years we have seen the emergence of hundreds of XML applications; document formats for any kind of data and text, where the markup makes explicit the structure and often even the meaning of the content. XML parsing is now supported by almost all programming languages, and there are standardized tools for almost any aspect of XML transformation.

Conceptually speaking, XML views a document as a tree of so-called elements. For communication this tree is represented as a well-formed bracketing structure, where the brackets of an element e1 are represented as \texttt{<e1/>} (opening) and \texttt{</e1>} (closing); the leaves of this tree are represented as empty element \texttt{<e1/>}, which can be abbreviated as \texttt{<e1/>}. The element nodes of this tree can be annotated by further information in so-called attribute in opening brackets: \texttt{<e1 visible="no"/>} might add the information for a formatting engine to hide this element. The use of “uniform resource identifiers” (URIs; special strings of standardized form that can be used to identify resources, e.g. XML subtrees on the Internet) allows cross-referencing and extends the underlying tree data structure to arbitrary graph structures (see e.g. [DMOT01]).

Speaking from a Computer Science perspective, XML lifts the level of communication between applications from exchanging strings to exchanging data structures (tree or graph structures). As a consequence XML has become the main interoperability framework, and gave rise to the “web services” paradigm that is taking over the world wide web and is rapidly becoming a major organizing force in mainstream IT. Large bodies of documents are transformed to XML form; major publishers are converting their work-flow to XML, e.g. document formats like DocBook [WM99] have virtually taken over software documentation.

Even with the emphasis on documents that XML has brought back into IT, there is surprisingly little support for document management functions like version control and collaborative authoring; [Lin01,LF02,KA03] are among the exceptions that prove the thesis. In particular there are no systems like CVS (Concurrent Versions System, [CVS]) system, which is widely used to support collaborative software and document development. The main reason for this lack of applications is the this lack of efficient algorithms for computing differences between documents in a given XML application. Systems like CVS use the text utilities \texttt{diff}, \texttt{patch}, and \texttt{merge} to compute differences between documents in the repository, update local copies, and synchronize edits by collaborating authors. The repository stores a current version of the document together with edit scripts (a representation of the document differences) that can be
used to compute old revisions. See [KA03] for a proposal for a collaborative knowledge management system based on suitable “semantic” difference computation algorithms.

In this paper, I propose to use unification-based techniques to approach the problem of XML difference computing. This note is less a report on solutions, than an attempt to publicize a problem from applications and enlist the interest of the unification community. In the next section we will look concretely at the problem of computing differences between XML documents and formulate it in a way that is amenable to treatment with unification methods. We will conclude with some speculations about variants of the difference computation problem and their applicability in XML document management practice.

2 Computing Differences in XML applications

Since XML views documents as serializations of trees (the so-called Document Object Models), it implicitly considers two documents as “equal”, if they serialize the same tree. For instance, XML specifies that the order of attribute declarations in XML elements is immaterial, double and single quotes can be used interchangeably for strings, XML comments (<!-- ... -->) are ignored, and whitespace characters in the Unicode serialization is only meaningful in text nodes. Moreover, certain XML applications impose even more equalities, for instance, the order of the bibliographic children of a bibliography element in the DocBook format does not matter (they will be sorted by the presentation style sheet anyway). Another example is the OpenMath [CC98] representation of mathematical objects that allows alphabetic renaming of bound variables and Currying (but not \( \beta \)-equality).

Clearly, a stronger notion of equality leads to more compact, less intrusive edit scripts\(^1\). For instance, if we know that whitespace carries no meaning in a document format, two documents are considered equal even if they differ (with respect to the distribution of whitespace characters) in every single line; as a consequence, the computed difference would be empty. This motivates the following general statement of the problem at hand.

\textbf{The General Difference Computation Problem (DCP):} Let \( \mathcal{K} \) be a class of trees and an equality theory \( \mathcal{E} \) on \( \mathcal{K} \). Given two trees \( S \) and \( T \), find an optimal edit-script that transforms \( S \) to \( T \).

Of course, we need to make this more specific. Since we are dealing with XML, with “trees” we mean finite, labeled, ordered\(^2\) trees. In XML, the class \( \mathcal{K} \) of trees will be those that are valid in a given XML document model, they are often given by a simple grammar, e.g., a Document Type Definition (DTD) or an XML Schema. The equality theory \( \mathcal{E} \) is sometimes detailed in the informal part of the specification of the document format, but often left informal, since the XML suite of standards does not offer support for this.

\(^1\) Compactness of edit scripts is important for storage and query efficiency in document management systems, while minimal intrusiveness (patching does not disrupt document structure) is important for humans to track and understand changes.

\(^2\) In fact XML trees have a peculiar notion of order; internal nodes have two kinds of children: node children and attribute children. The first are ordered, the second kind are unordered. We will neglect this detail for the exposition of this note.
The notion of what constitutes an edit script and how to compare them are real parameters to the problems, which certainly influence its complexity and the practical impact of the corresponding algorithms. Even though there are some initial proposals, for edit script formats (e.g. Update [LM00]), much more (theoretical) research is needed here.

There is a large body of work on using the XML tree structure to compute differences of XML documents modulo XML-equality (see e.g. [YW]). The work has been mainly concerned with finding algorithms for optimal (least-cost) edit scripts and complexity issues. A special case of the intended XML domain is that document formats are often strongly keyed (e.g. all elements have unique ID attributes, which cannot be changed by the user) or the knowledge management system employs some node numbering system like the one proposed in [CTZ01]), then the key structure gives a very natural notion of node correspondence, and differing becomes relatively simple. For the un-keyed case, only the notion of structural isomorphism and of ordered and un-ordered trees has been considered e.g. in [CRGMW96]. The general DCP of modulo an equality theory (we will speak of the $\mathcal{E}$-DCP) has remained unaddressed.

3 A Term Instance of the DCP

In this section, we will formulate an almost trivial variant of the DCP in terms that resemble unification theory. Again, the point here is not to present a solution (the application is too trivial for that), but to refine the problem specification and point out the similarity to unification methods.

We take the class $\mathcal{K}$ of trees to be first-order terms over a finite signature, and choose the set of edit scripts to be of the form $[A_1/p_1], \ldots, [A_n/p_n]$, where the $A_i$ are terms and the $p_i$ are positions, i.e. sequences of natural numbers, with the intention that if we apply an atomic edit script $[A/p]$ to a term $B$, then the result (we write this as $[A/p]B$) is obtained from $B$ by replacing the subtrees at position $p$ in $B$ with $A$.

In analogy to substitutions, we introduce a pre-ordering on edit scripts by $\sigma \leq \theta[\mathcal{P}]$ ($\sigma$ is more general than $\theta$ on a set $\mathcal{P}$ of positions), iff there is an edit script $\rho$, such that $\theta = \rho \circ \sigma[\mathcal{P}]$, where $\circ$ and equality on $\mathcal{P}$ are defined in the obvious way.

With these definitions, the following inference system computes most general edit scripts for the empty theory $\mathcal{E}$:

$$
\frac{A \vdash_{\mathcal{P}} A \land D}{D} \quad \frac{f(S_1, \ldots, S_n) \vdash_{\mathcal{P}} f(T_1, \ldots, T_n) \land D}{S_1 \vdash_{\mathcal{P} \land} T_1 \land \cdots \land S_n \vdash_{\mathcal{P} \lor} T_n \land D}
$$

Here we represent DCP as conjunctions of transformation judgments, the initial DCP being $S \vdash T$, where $e$ is the empty position. Note that under reasonable assumptions on the disjointness of the $p_i$, DCP can be interpreted as edit scripts by interpreting $D = S_1 \vdash_{\mathcal{P} \land} T_1 \land \cdots \land S_n \vdash_{\mathcal{P} \lor} T_n$ as the edit script $D = [S_1/p_1], \ldots, [S_n/p_n]$.

Clearly, if $D$ is terminal under this calculus (no rule applies; $D$ is a minimal solution of $D$). Moreover, the calculus is terminating, confluent, and conserves solutions; thus this instance of the DCP is unitary.

\footnote{The action of changing keys in the data, can lead to un-intuitive and computationally sub-optimal edit scripts, but does not compromise the method per se.}
The advantage of approaching the difference in this way is that we can make adapt all the machinery developed for $\mathcal{E}$-unification and constraint solving the the DCP; for instance, we can adapt the mutation rule from $C$-unification to do DC for unordered trees, if we generalize the decomposition rule above to

$$f(S_1, \ldots, S_n) \xrightarrow{\phi} f(T_1, \ldots, T_n) \land D \quad \phi \text{-- } n\text{-permutation}$$

$$S_1 \xrightarrow{\phi(1)} T_1 \land \cdots \land S_n \xrightarrow{\phi(n)} T_n \land D$$

It seems plausible to assume that other intuitions from unification theory and constraint solving can be adapted in a similar fashion. In particular, it seems possible to extend narrowing approaches to a wide range of $\mathcal{E}$-DCP in the absence of specialized methods.

Of course, the $\mathcal{E}$-DCP need is no longer be unitary as we have seen in the unordered tree case. In contrast to deductive applications of unification, all solutions to a DCP yield acceptable edit scripts. Therefore, we must have a way to measure the utility and refine the algorithm (calculus) only to produce optimal solutions (the generality condition used above is only a necessary condition for optimality). Plausible measures for comparing edit scripts would be their “weight”, i.e. the sum of weights (number of symbols) in the terms involved, or their “intrusiveness” e.g. the inverse of the sum of the lengths of the positions. The first would favor edit scripts whose overall change in the document is minimal, whereas the second favors such that respect as much of the structure of the original document (which might help the user stay oriented between document versions).

4 Conclusion and Speculations

We have described a computational problem that forms a bottleneck for XML document management systems and have presented a formulation that might make techniques from unification and constraint solving applicable to it.

In the previous section, we have simplified the DCP for XML considerably. In particular, the fact that functions have fixed arities in first-order terms has simplified the algorithm and the notion of edit script. XML notions of edit scripts like Update [LM00] also allow operations such as deleting or adding a subtree (at a position) to minimize intrusiveness and weight. We anticipate that this change is easily integrated in algorithms.

Enhancing the expressivity of the edit script language by common edit operations like moving a subtree in a document, or allowing insertion or deletion of contexts (e.g. when restructuring the document; moving around paragraphs or adding sectioning groups) will have a much greater impact on the structure of $\cdot$, but also on the usefulness of the $\mathcal{E}$-DC algorithms.

Complexity issues of the $\mathcal{E}$-DC algorithms will play an overwhelming role in the practical applicability in the XML context, since documents of sizes exceeding several megabytes are quite common.

One of the attractions of this problem is the availability of large natural corpora, on which the DCPs can be tested and evaluated in practice. To make this possible, we will need to survey XML document formats for the equality theories implicit in XML applications; the author's intuition is that the range of theories is rather limited, and develop an XML-compatible format for specifying these (e.g. as an extension to
XML schema; see [KA03] for a first proposal. The availability of these corpora may even pave the way to utilizing statistical and machine-learning methods to fine-tune algorithms.

References


Matching in Flat Theories

Teunr Kutsia*

Research Institute for Symbolic Computation
Johannes Kepler University
A-4040, Linz, Austria
Email: kutsia@ris.uni-linz.ac.at

Abstract. Flat theory with sequence variables and flexible arity symbols has a decidable
infinitary matching and unification. We briefly describe a minimal complete flat
matching procedure and discuss its relations with the flat matching implemented in
the Mathematica system.

1 Preliminaries

Sequence variables can be instantiated with a finite, possible empty, sequence of terms.
We use $\mathcal{F}, \mathcal{G}, \ldots$ to denote them. We build terms over individual and sequence variables,
and fixed and flexible arity function symbols in the usual way, with the only restriction
that sequence variables are not allowed to be direct arguments of fixed arity symbols.
Similarly, in equalities over terms, sequence variables can not be the direct arguments of
the equality symbol. For instance, for a binary $g$ and flexible arity $f$, $f(\mathcal{F}, g(x, y)) \simeq f(\mathcal{G})$
is an equation, while $f(\mathcal{F}, g(x, y)) \simeq f(\mathcal{Y})$ and $\mathcal{F} \simeq g(x, y)$ are not.

Let $\mathcal{F}_{\text{fix}}$ be a set of fixed arity function symbols and $\mathcal{F}_{\text{flex}}$ be a set of flexible arity
function symbols. We write $\mathcal{T}(\mathcal{F}_{\text{fix}} \cup \mathcal{F}_{\text{flex}}, \mathcal{V}_{\text{ind}} \cup \mathcal{V}_{\text{seq}})$ for the set of terms over the
signature $\mathcal{F}_{\text{fix}} \cup \mathcal{F}_{\text{flex}}$, a set $\mathcal{V}_{\text{ind}}$ of individual variables, and a set $\mathcal{V}_{\text{seq}}$ of sequence
variables.

We assume that the reader is familiar with the standard notions of unification theory [2]. Here we mention only non-straightforward generalizations for sequence variables and flexible arity symbols.

Definition 1. A binding is either a pair $x \leftarrow s$ where $x \in \mathcal{V}_{\text{ind}}$ and $s \in \mathcal{T}(\mathcal{F}_{\text{fix}} \cup \mathcal{F}_{\text{flex}}, \mathcal{V}_{\text{ind}} \cup \mathcal{V}_{\text{seq}}) \setminus \{x\} \cup \mathcal{V}_{\text{seq}}$, or an expression $x \leftarrow s_{1}, \ldots, s_{n}$ where $x \in \mathcal{V}_{\text{seq}}$ and
$s_{1}, \ldots, s_{n}$ is a (possibly empty) sequence of terms such that $s_{1} \neq x$ if $n = 1$.

A substitution is a finite set of bindings $\{x_{1} \leftarrow s_{1}, \ldots, x_{n} \leftarrow s_{n}, \mathcal{F}_{1} \leftarrow t_{1}^{1}, \ldots, t_{m}^{m}, \mathcal{F}_{m} \leftarrow t_{1}^{m}, \ldots, t_{m}^{m}\}$ with $x_{1}, \ldots, x_{n}, \mathcal{F}_{1}, \ldots, \mathcal{F}_{m}$ distinct variables.

Greek letters denote substitutions, with $\varepsilon$ for the empty substitution. The instance
of a term $\mathcal{P}$ w.r.t. a substitution $\theta$, $\mathcal{P} \theta$, is the sequence $s_{1}, \ldots, s_{n}$, if $\mathcal{P} \leftarrow s_{1}, \ldots, s_{n} \in \theta$, and $\mathcal{P}$ otherwise. The notion of instance for terms other than sequence variables is
defined in the usual way.

Definition 2. A substitution $\theta$ is mon: general than $\sigma$ on a finite set of variables $V$
module a theory $E (\theta \supset^V \sigma)$ iff there exists a substitution $\lambda$ such that

- for all $\mathcal{P} \in V$, the binding $\mathcal{P} \leftarrow$ does not belong to $\lambda$; there exist terms
  $t_{1}, \ldots, t_{n}, s_{1}, \ldots, s_{n}, n \geq 0$, such that $\mathcal{P} \sigma = t_{1}, \ldots, t_{n}, \mathcal{P} \lambda = s_{1}, \ldots, s_{n}$, and for each $1 \leq i \leq n$, either $t_{i}$ and $s_{i}$ are the same sequence variables, or $t_{i} \simeq_{E} s_{i}$;

* Supported by the Austrian Science Foundation (FWF) under Project SFB F1302.
• for all $x \in V$, $x \sigma \simeq_E x \theta \lambda$.

Example 1. $\{ \varphi \leftarrow \psi \} \simeq \{ \varphi \leftarrow a, b \}$, $\{ \psi \leftarrow a, b \}$, but not $\{ \varphi \leftarrow \psi \} \simeq \{ \varphi \leftarrow \psi \} \{ \varphi \leftarrow \psi \} \{ \psi \leftarrow \varphi \} \{ \psi \leftarrow \varphi \}$.

Flat theory, or shortly $F$-theory, is defined as $E = \{ f(\varphi, f(\psi)), \varphi \simeq f(\varphi, \varphi, \varphi) \}; f \in \mathcal{F}_{\text{flex}}$ is called a flat flexible arity symbol. It should be noted that although (free or flat) unification with sequence variables and flexible arity symbols looks similar to A-unification, there are essential differences illustrated by the following example (even without sequence variables). Let $f(x, f(y, z)) \simeq f(f(a, b), c)$ be a unification problem, where $x, y, z$ are individual variables, and $a, b, c$ are constants. For associative $f$ the minimal complete set of solutions is the singleton $\{ \{ x \leftarrow a, y \leftarrow b, z \leftarrow c \} \}$; for free flexible arity symbol $f$ the problem has no solution; and for flat flexible arity $f$ there are 23 substitutions in the minimal complete set of solutions: $\{ \{ x \leftarrow f(), y \leftarrow f(), z \leftarrow f(a, b, c) \} \}, \{ x \leftarrow f(), y \leftarrow a, z \leftarrow f(b, c) \}, \{ x \leftarrow f(), y \leftarrow f(a), z \leftarrow f(b, c) \}, \{ x \leftarrow f(), y \leftarrow f(a, b), z \leftarrow f(c) \}, \{ x \leftarrow f(), y \leftarrow f(a, b, c), z \leftarrow f() \}, \{ x \leftarrow a, y \leftarrow f(), z \leftarrow f(b, c) \}, \{ x \leftarrow a, y \leftarrow f(a), z \leftarrow f(b, c) \}, \{ x \leftarrow a, y \leftarrow f(a, b), z \leftarrow f(c) \}, \{ x \leftarrow a, y \leftarrow f(a, b, c), z \leftarrow f() \}, \{ x \leftarrow a, y \leftarrow f(b), z \leftarrow f(c) \}, \{ x \leftarrow a, y \leftarrow f(b), z \leftarrow f() \}$.

Below we consider general $F$-unification and $F$-matching, i.e., besides flat flexible arity symbols we have also free fixed and free flexible arity symbols.

2 Unification and Matching

General flat unification is decidable. It can be proved using the Baader-Schulz method [1], reducing a flat unification solvability problem to a combination of solvability problems of word equations (with certain additional restrictions) and syntactic unification. For the details we refer to [4].

Flat unification is infinitary. Unification procedure is designed as a tree generation process, in the breadth-first manner. Each node of the tree is labeled either with a unification problem (kept in flattened form), with $\top$, or with $\bot$. The root node is labeled with the unification problem to be solved. $\top$ and $\bot$ are terminal nodes. Before expanding a non-terminal node, we first check whether the problem attached to the node is solvable. Children of the root are obtained by projection (elimination of all possible subsets of the set of sequence variables on the root node). Children of other non-terminal nodes are obtained by transformation rules. Due to a lack of space we do not describe those rules here, neither more details about the procedure. They can be found in [4]. Unifiers are constructed by composing substitutions on the edges of branches with the terminal node $\top$. The procedure enumerates minimal complete set of unifiers.

We give a bit more details about general $F$-matching, since it has some interesting properties and applications. Decidability and the existence of a minimal set of matchers for a general $F$-matching problem follows from the corresponding properties of general $F$-unification. Interestingly, it turns out that $F$-matching is infinitary, as the following simple example illustrates:
Example 2. The minimal complete set of solutions for the $F$-matching problem $f(\overline{x}) \approx f(a)$ with flat $f$ is $\{\{\overline{x} \leftarrow a\}, \{\overline{x} \leftarrow f(a)\}, \{\overline{x} \leftarrow f(\), f(a)\}, \{\overline{x} \leftarrow f(, a), f(a)\}, \{\overline{x} \leftarrow f(, f(a))\}, \{\overline{x} \leftarrow f(, a, f(a))\}, \ldots\}$.

$F$-matching procedure is designed like the one for $F$-unification — as a tree generation process. It enumerates the minimal complete set of matchers\(^1\). Transformation rules for $F$-matching are given in Appendix. Each of the them has one of the following forms: $M \rightarrow \bot$ or $M \rightarrow \langle\langle M_1, \sigma_1\rangle, \ldots, \langle M_n, \sigma_n\rangle\rangle$ where $M$ is a matching problem, each of the successors $M_i$ is either $\top$ or a new matching problem, and each $\sigma_i$ is a substitution used to generate $M_i$ from $M$.

The rules involving $f(\)$ in the substitutions are the reason why $F$-matching is infinitary. However, they are indispensable for completeness, as the following examples illustrate:

Example 3. The unique solution $\{x \leftarrow f(\)\}$ of $f(x, a) \approx f(a)$ with flat $f$ can not be computed unless the transformation substitution $\{x \leftarrow f(\)\}$ is allowed.

Example 4. The transformation rule $\{\overline{x} \leftarrow f(, \overline{x})\}$ is crucial for computing a unifier $\{\overline{x} \leftarrow f(, a)\}$ of $f(\overline{x}, g(\overline{x})) \approx f(a, g(f(, a)))$, where $f$ is flat and $g$ is free. The derivation is shown on Fig. 1. Composing substitutions top-down on the edges gives the solution.

\[
\begin{array}{c}
\text{f(\overline{x}, g(\overline{x}))} \approx f(a, g(f(, a))) \\
\downarrow \text{f(\overline{x}, g(\overline{x}))} \approx f(a, g(f(, a))) \\
\text{f(\overline{x})} \leftarrow f(, \overline{x}) \\
\downarrow \text{f(\overline{x})} \leftarrow f(, a) \\
\text{f(a, g(f(, a)))} \approx f(a, g(f(, a))) \\
\downarrow \varepsilon \\
\top
\end{array}
\]

Fig. 1. Computing a solution for $f(\overline{x}, g(\overline{x})) \approx f(a, g(f(, a)))$.

\(^1\) To ensure minimality, during the tree generation process an additional effort is needed each time when a new solution appears: to delete from the already computed (finite) set of solutions those substitutions which violate minimality condition.
3 Flat Matching in Mathematica

Mathematica system [6] implements matching modulo flatness. The algorithm itself, to our knowledge, is nowhere described, but is briefly explained on examples in [6]. However, it is not hard to observe that the algorithm is not complete. It does not match, for instance, \( f(x, a) \) to \( f(a) \), \( f(x, g(x)) \) to \( f(a, g(a)) \), or \( f(\mathcal{F}, g(\mathcal{F})) \) to \( f(a, g(f(a))) \), where \( f \) is flat and \( g \) is free.

The main difference between the F-matching procedure and the Mathematica flat matching is that the latter does not consider transformation rules involving \( f() \). It makes Mathematica flat matching finitary.

Example 5. Let \( f(\mathcal{F}) \prec \mathcal{F} f() \) be a matching problem, where \( f \) is flat. The F-matching procedure enumerates the infinite minimal complete set of matchers \( \{\{\mathcal{F} \leftarrow \}, \{\mathcal{F} \leftarrow f()\}, \{\mathcal{F} \leftarrow f(), f()\}, \ldots\} \). Mathematica flat matching algorithm returns a single solution \( \{\{\mathcal{F} \leftarrow \}\} \).

Another difference is in the case where an individual variable \( x \) matches a single argument \( s_1 \) in a term with a flat head \( f \). The F-matching procedure generates four successors as it is shown in the third case of Eliminate on Fig. 2, while the Mathematica matching algorithm chooses only the last two of those four. If in the same situation we have a sequence variable instead of an individual variable, the F-matching procedure tries 9 different ways to resolve the case (the fourth rule of Eliminate on Fig. 2), while Mathematica would choose only the second and sixth.

On the other hand, Mathematica can verify that each solution computed by the F-matching procedure is correct, e.g., it sees \( f(x, g(x))\{x \leftarrow a\} \) and \( f(a, g(a)) \) as identical expressions, although, as it was already mentioned, the Mathematica matching algorithm can not compute the substitution \( \{x \leftarrow a\} \) which matches \( f(x, g(x)) \) to \( f(a, g(a)) \).

To summarize the similarities and differences between the F-matching and flat matching in Mathematica, on Fig. 4 we give rules that can simulate the behavior of Mathematica flat matching. Note that in the tree generation process we do not need to check solvability at the nodes anymore. For a given matching problem, the output of the procedure involving the rules on Fig. 4 would be identical to the set of all possible matchers Mathematica matching algorithm computes (one can see all the Mathematica matchers using the function Replacelist, for instance).

However, when Mathematica tries to match patterns in the left hand side of its assignments or rules to some expression, from all possible matchers it selects the first one it finds. We can simulate also this behavior, imposing an order of choosing successors in the Eliminate \( \mathcal{F} \) step and in the projection phase, and stopping the development of the tree whenever the first solution appears.

4 Conclusion

We considered flat theories with sequence variables and flexible arity symbols. The interesting feature of such a theory is that it has decidable infinitary general matching

\(^2\) In fact, this is the case not only with this particular algorithm, but, in general, with the evaluation semantics of the programming language of Mathematica. See [3] for more discussion on this topic.

\(^3\) The Sequentica package [5] allows the user to gain more control on the selection of matchers.
and unification. We gave a sketch of a procedure which enumerates minimal complete set of solutions, and discussed a relation of the matching procedure with the pattern matching algorithm of Mathematica.

5 Acknowledgements

I am grateful to Prof. Bruno Buchberger who supervised this work, and to Mircea Marin with whom I had useful discussions.

References

A Transformation Rules

Success: \[ t \ll p t \rightarrow \langle \top, \varepsilon \rangle. \]
\[ x \ll p t \rightarrow \langle \top, \{ x \leftarrow t \} \rangle. \]

Failure: \[ f() \ll p f(t_1, \tilde{t}) \rightarrow \bot. \]
\[ f(t_1, \tilde{t}) \ll p f() \rightarrow \bot, \quad \text{if } t_1 \notin \mathcal{V}_\text{und} \cup \mathcal{V}_\text{seq}. \]
\[ f(t_1, \tilde{t}) \ll p f(s_1, \tilde{s}) \rightarrow \bot, \quad \text{if } t_1 \ll p s_1 \rightarrow \bot. \]
\[ c_1 \ll p c_2 \rightarrow \bot, \quad \text{if } c_1 \neq c_2. \]
\[ h_1(\tilde{t}) \ll p h_2(\tilde{s}) \rightarrow \bot, \quad \text{if } h_1 \neq h_2. \]
\[ h(\tilde{t}) \ll p h(\tilde{s}) \rightarrow \bot, \quad \text{if } t_1 \ll p s_1 \rightarrow \bot. \]

Split: \[ h_1(t_1, \tilde{t}) \ll p h_1(s_1, \tilde{s}) \rightarrow \langle \{ h_1, \tilde{t}, h_1(q_1, \tilde{q}), \sigma_1 \}, \ldots, \{ \{ h_1(q_k, \tilde{q}_k), \sigma_k \} \rangle, \]
\[ t_1 \ll p s_1 \rightarrow \langle \{ \{ h_1(q_1, \tilde{q}), \sigma_1 \}, \ldots, \{ \{ h_1(q_k, \tilde{q}_k), \sigma_k \} \rangle. \]

Eliminate: \[ f(x, \tilde{t}) \ll p f() \rightarrow \langle f(\tilde{t}) \ll p f(), \sigma \rangle, \quad \text{where } \sigma = \{ x \leftarrow f() \}. \]
\[ f(\tilde{x}, \tilde{t}) \ll p f() \rightarrow \langle f(\tilde{t}) \ll p f(), \sigma_1 \rangle, \quad \text{where } \sigma_1 = \{ \tilde{x} \leftarrow f() \}, \]
\[ \{ f(\tilde{x}, \tilde{t}) \ll p f() \ll p f(), \sigma_2 \}, \quad \text{where } \sigma_2 = \{ \tilde{x} \leftarrow f() \}. \]
\[ f(x, \tilde{t}) \ll p f(s_1, \tilde{s}) \rightarrow \langle f(\tilde{t}) \ll p f(), \sigma_1 \rangle, \quad \text{where } \sigma_1 = \{ x \leftarrow f() \} \]
\[ \{ f(\tilde{t}) \ll p f(s_1, \tilde{s}), \sigma_2 \}, \quad \text{where } \sigma_2 = \{ x \leftarrow s_1 \}, \]
\[ \{ f(\tilde{t}) \ll p f(s_1, \tilde{s}), \sigma_3 \}, \quad \text{where } \sigma_3 = \{ x \leftarrow f(s_1) \}, \]
\[ \{ f(x, \tilde{t}) \ll p f(s_1, \tilde{s}), \sigma_4 \}, \quad \text{where } \sigma_4 = \{ x \leftarrow f(s_1, x) \}. \]

Fig. 2. F-matching. Transformation rules. \( \tilde{t} \) and \( \tilde{s} \) are sequences of terms. \( h \in \mathcal{F}_\text{fix} \cup \mathcal{F}_\text{fix} \) is free. \( h_1, h_2 \in \mathcal{F}_\text{fix} \cup \mathcal{F}_\text{fix} \) can be free or flat. \( f \in \mathcal{F}_\text{fix} \) is flat.
Eliminate (cont.): \(f(t, \tilde{t}) \ll_{f} f(t, \tilde{s}) \leadsto \langle \langle f(\tilde{t}) \ll_{f} f(\tilde{s}), e \rangle \rangle\).
\(h(t_1, \tilde{t}) \ll_{h} h(s_1, \tilde{s}) \leadsto \langle \langle g(\tilde{t}) \ll_{g} g(\tilde{s}), \sigma \rangle \rangle\), if \(t_1 \ll_{f} s_1 \leadsto \langle \langle T, \sigma \rangle \rangle\).
\(h(\tilde{t}, \tilde{s}) \ll_{h} h(s_1, \tilde{s}) \leadsto \langle \langle g(\tilde{t}) \ll_{g} g(\tilde{s}), \sigma_1 \rangle \rangle\), where \(\sigma_1 = \{\tilde{t} \leftarrow s_1\}, \langle g(\tilde{t}) \ll_{g} g(\tilde{s}), \sigma_2 \rangle\), \(\sigma_2 = \{\tilde{t} \leftarrow s_1, \tilde{t}\} \in \mathcal{F}_{\text{Flat}}\).

**Fig. 3.** \(F\)-matching. Transformation rules (cont.). \(\tilde{t}\) and \(\tilde{s}\) are sequences of terms. \(h \in \mathcal{F}_{\text{Flat}} \cup \mathcal{F}_{\text{Free}}\) is free, \(g \in \mathcal{F}_{\text{Free}}\) is a new free symbol if \(h \in \mathcal{F}_{\text{Flat}}\), otherwise \(g = h\). \(f \in \mathcal{F}_{\text{Free}}\) is flat.

**SuccessM:**
\(t \ll_{f} t \leadsto \langle \langle T, e \rangle \rangle\).
\(x \ll_{f} t \leadsto \langle \langle T, \{x \leftarrow t\} \rangle \rangle\).

**FailureM:**
\(f(t_1, \tilde{t}) \ll_{f} f(t) \leadsto \bot\).
\(f(t_1, \tilde{t}) \ll_{f} f(s_1, \tilde{s}) \leadsto \bot\), if \(t_1 \ll_{f} s_1 \leadsto \bot\).
\(c_1 \ll_{f} c_2 \leadsto \bot\), if \(c_1 \neq c_2\).
\(h(t) \ll_{h} h(\tilde{t}) \leadsto \bot\).
\(h() \ll_{h} h() \leadsto \bot\).
\(h(t_1, \tilde{t}) \ll_{h} h(s_1, \tilde{s}) \leadsto \bot\), if \(t_1 \ll_{h} s_1 \leadsto \bot\).

**SplitM:**
\(h_1(t_1, \tilde{t}) \ll_{h_1} h_1(s_1, \tilde{s}) \leadsto \langle \langle h_1(q_1, \tilde{t}) \ll_{h_1} h_1(q_1, \tilde{s}), \sigma_1 \rangle \rangle\), if \(t_1 \notin \mathcal{V}_{\text{Inst}} \cup \mathcal{V}_{\text{Free}}\).
\(\ldots\langle h_1(q_k, \tilde{t}) \ll_{h_1} h_1(q_k, \tilde{s}), \sigma_k \rangle\). \(\ldots \langle h_1(q_k, \tilde{t}) \ll_{h_1} h_1(q_k, \tilde{s}), \sigma_k \rangle\).

**EliminateM:**
\(f(t, \tilde{t}) \ll_{f} f(t, \tilde{s}) \leadsto \langle \langle f(\tilde{t}) \ll_{f} f(\tilde{s}), e \rangle \rangle\).
\(f(x, \tilde{t}) \ll_{f} f(s_1, \tilde{s}) \leadsto \langle \langle f(\tilde{t}) \ll_{f} f(\tilde{s}), e \rangle \rangle\).
\(\ldots \langle f(\tilde{t}) \ll_{f} f(\tilde{s}), e \rangle \rangle\).
\(\ldots \langle f(\tilde{t}) \ll_{f} f(\tilde{s}), e \rangle \rangle\).

**Fig. 4.** Transformation rules to simulate Mathematica flat matching. \(\tilde{t}\) and \(\tilde{s}\) are sequences of terms. \(h_1, h_2 \in \mathcal{F}_{\text{Flat}} \cup \mathcal{F}_{\text{Free}}\) can be free or flat. \(h \in \mathcal{F}_{\text{Flat}} \cup \mathcal{F}_{\text{Free}}\) is free.
\(g \in \mathcal{F}_{\text{Free}}\) is a new free symbol if \(h \in \mathcal{F}_{\text{Flat}}\), otherwise \(g = h\). \(f \in \mathcal{F}_{\text{Free}}\) is flat.
Functional Programming with Sequence Variables: The \textit{Sequentica} Package

Mircea Marin

Johann Radon Institute for Computational and Applied Mathematics
Austrian Academy of Sciences,
A-4040, Linz, Austria
Email: marin@indmath.uni-linz.ac.at

\textbf{Abstract.} Sequence variables are an advanced feature of modern languages which allows to program in a declarative and easy to understand way. Functional programming with sequence variables relies on the choice of a matcher, which in general is not unique. We propose a number of idioms for programming with sequence variables which enable the user to control the choice of the matcher. To this end we have developed the package \textit{Sequentica}, which extends the language of \textit{Mathematica} with our programming constructs. Our extensions enable (1) to control the selection of matcher by annotating sequence variables with priorities and ranges for their lengths, and (2) to compute optimum values characterized by a score function which must be optimized. We illustrate the usefulness of our extensions and describe how they have been implemented.

1 Introduction

The concept of sequence variable has proved useful in many areas, such as symbolic computation [6], term rewriting [1,4], unification [5] and computational logic [3]. We feel that the expressive power of this concept is not yet completely understood and that a better formalization is still desirable.

A sequence variable is a function parameter which can be instantiated with a (possibly empty) sequence of terms. Currently, \textit{Mathematica} is one of the languages with best support to program with sequence variables. For instance, we can define a partial function

\begin{verbatim}
IntElem[{x__,y_Integer,z__}]:=y
\end{verbatim}

which yields an integer element of a list 1 upon the call \texttt{IntElem[1]}. For instance, the call \texttt{IntElem[\{a,1,x,s,2\}]} can yield either 1 by computing the matcher \{\texttt{x \rightarrow \texttt{\"a\"}, y \rightarrow 1, z \rightarrow \texttt{\"x\", s, 2\"}\}} or 2 by computing \{\texttt{x \rightarrow \texttt{\"a\", 1, x, \"s\", y \rightarrow 2, z \rightarrow \texttt{\"\"}}}\}. (Note: to aid the reading the matchers, we have written the bindings of sequence variables between " and "). The \textit{Mathematica} book [6] provides several convincing examples in favor of using such a programming style. But, as seen from the previous example, this approach relies on choosing a particular matcher during the evaluation of a function call. Apparently, the interpreter of \textit{Mathematica} chooses the matcher which assigns the shortest possible lengths to the bindings of the first sequence variables that show up when traversing the pattern in a leftmost-innermost manner. In our example, the \textit{Mathematica} interpreter

\footnote{The author was supported by the Austrian Academy of Sciences, Johann Radon Institute for Computational and Applied Mathematics.}
chooses the matcher \( \{ x \mapsto "a", \ y \mapsto 1, \ z \mapsto "x, x, 2" \} \) because the binding "a" for \( x \) in the first matcher is shorter than the binding "a, 1, x, s" for \( x \) in the second matcher.

It is often desirable to provide the user with means to control the choice of a matcher. In the sequel we propose a couple of language extensions for doing so. We precede the presentation of our contributions with a brief account of the programming capabilities of Mathematica with patterns. We refer the reader to [6, Sect. 2.3] for a complete description of all Mathematica patterns.

<table>
<thead>
<tr>
<th>Pattern Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>- one term</td>
</tr>
<tr>
<td>- sequence of 1 or more terms</td>
</tr>
<tr>
<td>- sequence of 0 or more terms</td>
</tr>
<tr>
<td>( _h ) sequence of 1 or more terms, all of whose heads are ( h )</td>
</tr>
<tr>
<td>( _h ) sequence of 0 or more terms, all of whose heads are ( h )</td>
</tr>
<tr>
<td>( _2 ) sequence of 1 or more term which satisfy ( \text{test} )</td>
</tr>
<tr>
<td>( _2 ) sequence of 0 or more term which satisfy ( \text{test} )</td>
</tr>
</tbody>
</table>

**Table 1. Patterns in Mathematica**

A pattern \( \text{patt} \) may be named for later reference, e.g., we can write \( x : \text{patt} \) for a pattern \( \text{patt} \) named \( x \). The separator : is usually omitted when \( \text{patt} \) starts with an underline. For example, we prefer to write \( x : \text{Real} \) and \( y : \_2 \text{test} \) instead of \( x : \text{Real} \) and \( y : \_2 \text{test} \). Another useful specification is \( \text{patt}/; \text{cond} \) which defines a pattern \( \text{patt} \) whose meaning is enhanced with the requirement that \( \text{cond} \) holds for the corresponding matcher.

A sequence variable is the name of a pattern for a sequence of terms.

## 2 Extension 1

This extension addresses the possibilities to (1) control the choice of a particular matcher instead of relying on some built-in pattern matching strategy; (2) confine the lengths of sequence variable bindings to certain intervals; and (3) impose equality constraints between the lengths of bindings of sequence variables. This can be achieved by annotating sequence variables with (1) binding priorities which specify the order in which the sequences are assigned bindings upon pattern matching, and (2) bounds for the lengths of their bindings. We propose the extension with sequence annotations shown in Table 2, where \( \text{patt} \) is a sequence pattern. The sequence variables are assigned

<table>
<thead>
<tr>
<th>Pattern Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{patt} \langle { p, m, M } ) sequence with priority ( p ) and length varying from ( m ) to ( M )</td>
</tr>
</tbody>
</table>

**Table 2. Sequence pattern annotation with priority and length bounds**

bindings in the increasing order of their priorities, by looking for the first matcher that is obtained by varying the sequence length from \( m \) to \( M \).
The following examples illustrate the expressive power of this extension.

**Example 1.** The function defined by

\[
\text{GetSubsequence}[x \leftarrow \{2, 0, \infty\}, y \leftarrow \{1, \infty, 1\}, z \leftarrow \{3, 0, \infty\}] := \{y\}
\]

yields the list of the longest nonempty subsequence in the input, which occurs repeatedly at least twice. The assigned priorities are: 1 for \(y\), 2 for \(x\) and 3 for \(z\), and thus we first try to bind the sequence variable \(y\), next \(x\), and finally \(z\). The bindings for \(y\) are looked up starting from the largest possible length (denoted here by \(\infty\)) down to length 1. □

**Example 2.** It is convenient to be able to constrain different sequences to have the same length. We achieve this by imposing that sequence variables annotated with the same priority denote sequences of same length. The function

\[
\text{GetPalindrome}[
  x \leftarrow \{2, 0, \infty\},
  y \leftarrow \{1, \infty, 1\},
  u \leftarrow \{1, \infty, 1\},
  z \leftarrow \{3, 0, \infty\}; \{y\} == \text{Reverse}[[z]] := \{y, u, z\}
\]

yields the longest palindrome of odd length contained in the input sequence. □

**Assumptions.** To avoid excessive annotations of sequence variables, we assume that (1) it is sufficient to annotate one occurrence of a sequence variable; all other occurrences of that sequence variable are assumed to have the same annotation, (2) anonymous sequence variables have assigned distinct names, and (3) a sequence variable \(y\) (resp. \(y\)) which is not explicitly annotated has an implicit sequence annotation of the form \(\leftarrow \{p, 1, \infty\}\) (resp. \(\leftarrow \{p, 0, \infty\}\)). The default priorities \(p\) are assigned in a leftmost-innermost manner, and are assumed to be larger than the priorities given explicitly. This means that the non-annotated sequence variables are the last ones which get assigned bindings.

### 2.1 Extension 2

Our second extension addresses the possibility to select matchers which render an expression optimal. To illustrate, let us consider the problem of defining a function \(\text{MaxVariation}\) which yields the sublist of reals \(l_i = \{v_1, \ldots, v_{n_i}\}\) of a list \(l = \{l_1, \ldots, l_n\}\) for which the variation \(\text{max}(l_i) - \text{min}(l_i)\) is maximum. We propose the following syntax call:

\[
\text{MaxVariation}[[\text{__} \{\text{x,Real}, \text{__}\}]] := \text{BestFit}[\text{x,Min}[\text{x}], \text{Greater}]
\]

to find the list \(\{x\}\) computed by a matching against \(\text{__} \{\text{x,Real}, \text{__}\}\) which produces the greatest value of the expression \(\text{Min}[x] - \text{Min}[x]\). More generally, we suggest the following new definitional mechanisms:

1. A definition

\[
f[\text{patt}] := \text{BestFit}[\text{expr, opthm, test}]
\]
defines a partial function \( f \) which associates an input call \( t = f[a_1, \ldots, a_n] \) to the instance \( \text{expr}/\theta \) produced by the matcher \( \theta \) between \( t \) and \( f[patt] \) which satisfies \( \text{test}[\text{optim}/\theta, \text{optim}/\theta'] = \text{True} \) for all other matchers \( \theta' \) between \( t \) and \( patt \). \( f \) is undefined for input calls which don’t match \( f[patt] \).

2. A definition

\[
 f[patt] := \text{BestFits}[n, \text{expr}, \text{optim}, \text{test}]
\]

with \( n > 0 \) defines a function \( f \) which associates to an input call \( t = f[a_1, \ldots, a_n] \) the list of instances \( \{\text{expr}/\theta_i | 1 \leq i \leq k\} \) produced by the longest list of distinct matchers \( \{\theta_1, \ldots, \theta_k\} \) between \( t \) and \( f[patt] \), such that:

1. \( k \leq n \)
2. \( \text{test}[\text{optim}/\theta_j, \text{optim}/\theta_{j+1}] = \text{True} \) if \( 1 \leq j < k \), and
3. \( \text{test}[\text{optim}/\theta_k, \text{optim}/\theta] = \text{True} \) for all matchers \( \theta \not\in \{\theta_1, \ldots, \theta_k\} \).

This definition is partial: \( f[a_1, \ldots, a_n] \) remains unevaluated if \( f[a_1, \ldots, a_n] \) does not match with \( f[patt] \).

In both cases, \( \text{test}[\cdot] \) is assumed to define a total quasi-order on the set of values \( \text{optim}/\theta \) generated by the matchers \( \theta \) of arbitrary expressions against \( f[patt] \).

3 Algorithmic Aspects and Implementation Issues

Before explaining the algorithmic challenges of implementing our extensions, we introduce a couple of useful notations and definitions.

3.1 Terminology

We denote by \( \text{Matchers}(\text{expr}, \text{patt}) \) the set of matchers between \( \text{expr} \) and a pattern \( \text{patt} \). We assume that \( \text{cut} \) is a distinguished variadic function symbol which is reserved for internal use. We employ \( \text{cut} \) to mark positions of sequence variables in a pattern and to control lengths of sub-sequences by wrapping them in \( \text{cut}[\ldots] \) subterms. Let \( u(\text{patt}) \) be the result of removing all annotations from a sequence-annotated pattern \( \text{patt} \), \( \text{Pos}(\text{patt}) \) be the set of positions of a pattern \( \text{patt} \), and \( \text{patt}^+ \) be the pattern obtained from \( u(\text{patt}) \) by wrapping with \( \text{cut} \) all sequence pattern occurrences. A position \( p \in \text{Pos}(\text{patt}) \) is a sequence position if there exists an occurrence of a (possibly annotated) sequence pattern immediately below \( p \). The set of sequence positions of \( \text{patt} \) is denoted by \( \text{Pos}_{seq}(\text{patt}) \). The stratification of \( \text{Pos}_{seq}(\text{patt}) \) is the partition \( \text{Pos}_{seq}(\text{patt})^1 \cup \ldots \cup \text{Pos}^n_{seq}(\text{patt}) \) of the set \( \text{Pos}_{seq}(\text{patt}) \) into \( n \) nonempty subsets, such that:

1. \( \bigcup_{i=1}^{n} \text{Pos}_{seq}^i(\text{patt}) = \text{Pos}_{seq}(\text{patt}), \)
2. \( \forall i \in \{1, \ldots, n\}, \forall p_1, p_2 \in \text{Pos}_{seq}^i(\text{patt}), p_1 \neq p_2 \Rightarrow p_1 \perp p_2, \) and
3. \( \forall i \in \{1, \ldots, n-1\}, \forall p_2 \in \text{Pos}_{seq}^i(\text{patt}), \exists p_1 \in \text{Pos}_{seq}^{i+1}(\text{patt}), p_1 < p_2. \)

The number \( n \) which shows up in the definition of stratification is called the sequence depth of \( \text{patt} \), and is denoted by \#(\text{patt}).

Example 3. If \( P \) is the sequence-annotated pattern \( f[x_\leftarrow \{1, \infty, 2\}, g[x_\leftarrow h[y_\leftarrow \{2, \infty, 1\}, y_\leftarrow t_\leftarrow \{1, \infty, 2\}], h[z_\leftarrow t_\leftarrow \{1, \infty, 2\}]] \text{then} \)

\[
u(P) = f[x_\leftarrow g[x_\leftarrow h[y_\leftarrow \{2, \infty, 1\}, y_\leftarrow t_\leftarrow \{1, \infty, 2\}], h[z_\leftarrow t_\leftarrow \{1, \infty, 2\}], \text{cut}[z_\leftarrow t_\leftarrow \{1, \infty, 2\}], h[z_\leftarrow t_\leftarrow \{1, \infty, 2\}], h[z_\leftarrow t_\leftarrow \{1, \infty, 2\}], h[z_\leftarrow t_\leftarrow \{1, \infty, 2\}], h[z_\leftarrow t_\leftarrow \{1, \infty, 2\}], h[z_\leftarrow t_\leftarrow \{1, \infty, 2\}]]
\]

\(^1\) In \textbf{Mathematica}, \( \text{expr}/\theta \) is the result of applying substitution \( \theta \) to \( \text{expr} \).
$\text{Pos}_{\text{seq}}(P) = \{A, 2, 3, 2.2\}$ where $A$ stands for the root position of a term, and $\#(P) = 3$. The stratification of $\text{Pos}_{\text{seq}}(P)$ is $\{A\} \cup \{2, 3\} \cup \{2.2\}$. □

Suppose $\text{Pos}_{\text{seq}}^i(\text{patt}) = \{p_{i,1}, \ldots, p_{i,n_i}\}$. The $i$-th approximation of $\text{patt}$ is the pair $\langle S_i, \text{patt}_i \rangle$ where:

- $S_i = \{s_{i,1}, \ldots, s_{i,n_i}\}$ is a set of fresh sequence variable identifiers, and
- $\text{patt}_i$ is obtained from $\text{patt}$ by: (1) dropping all conditions attached to (sub)patterns, (2) replacing the patterns immediately below position $p_{i,j}$ with the sequence variable pattern $s_{i,j}$ and (3) wrapping with cut all sequence pattern occurrences different from $s_{i,j}$.

Example 4. The approximations of the pattern $P$ of Example 3 are:

- $\langle S_1, P_1 \rangle = \{s_{1,1}, f[s_{1,1}]\}$,
- $\langle S_2, P_2 \rangle = \{s_{2,1}, s_{2,2}, f[\text{cut}[x_-, g[s_{2,1}], h[s_{2,2}]]]\}$,
- $\langle S_3, P_3 \rangle = f[\text{cut}[x_-], g[\text{cut}[y_-], h[\text{cut}[y_-], \text{cut}[t_-]], h[\text{cut}[z_-], \text{cut}[t_-]]]]$.

Remark 1. If $\langle S_i, \text{patt}_i \rangle$ is an approximation of a pattern with sequence variables, then matching against $\text{patt}_i$ is unitary. This is so because $\text{patt}_i$ has no nodes with more than one sequence variable as immediate descendants. □

The $i$-th system of linear diophantine constraints determined by $\text{patt}$ is

\[
\text{Dioph}(\text{patt}_i) : \left\{ \begin{array}{l}
\text{line xpr}_{i,1} = l_{s_{i,1}}, \\
\vdots \\
\text{line xpr}_{i,n_i} = l_{s_{i,n_i}}, \\
\text{length} \_ \text{equalities}, \\
l_{s_{i,j}} \in [m_1..M_1], \ldots, l_{s_{i,n_i}} \in [m_k..M_k]
\end{array} \right.
\] (1)

in variables $(l_{s_{i,1}}, \ldots, l_{s_{i,k}}, l_{s_{i,1}}, \ldots, l_{s_{i,n_i}})$ where:

- (a) $x_1, \ldots, x_k$ is the sequence of all variable identifiers which appear in $\text{patt}$ immediately below the positions of $\text{Pos}_{\text{seq}}^i(\text{patt})$. We assume that the enumeration $x_1, \ldots, x_k$ is in the increasing order of sequence variable priorities,
- (b) $m_j, M_j$ are the bounds for the lengths $l_{s_{i,j}}$ of the sequence variables $x_j$ which are given in the sequence pattern annotations of $\text{patt}$, $(1 \leq j \leq k)$,
- (c) $\text{length} \_ \text{equalities}$ are the equalities between the lengths of sequence variables $x_1, \ldots, x_k$ which can be inferred from the sequence pattern annotations, and
- (d) $\text{line xpr}_{i,j}$ is the linear expression in variables $l_{s_{i,1}}, \ldots, l_{s_{i,n_i}}$ which defines the number of arguments of the sub-pattern of $\text{patt}$ at position $p_{i,j}$. $(1 \leq j \leq n_i)$

Example 5. The second system of linear diophantine constraints determined by the sequence-annotated pattern $P$ given above is:

\[
\text{Dioph}(P_2) : \left\{ \begin{array}{l}
l_{x} + 1 = l_{x_{a,1}}, \\
l_{x} + l_t = l_{x_{a,2}}, \\
l_{x} = l_t, \\
l_{x}, l_t \in [\infty, 2], l_t \in [1..\infty].
\end{array} \right.
\] □

Given a substitution $\sigma$ and a sequence variable $x$, we define $l_x/\sigma = m + \Sigma_{k=1}^n p_d x_i$ where $x_1, \ldots, x_n$ are all sequence variables which occur in the sequence $x/\sigma$, $p_i$ is the number of occurrences of $x_i$ in the sequence $x/\sigma$, and

\[
m = \text{length of sequence } x/\sigma - \Sigma_{k=1}^n p_i.
\]
Example 6. Consider the pattern \( P_2 = f[\text{cut}[x_1, y, z], g[s_{2,1}, h[s_{2,2}, t]]] \) and the substitution \( \sigma = \{ x \mapsto r, y, z \mapsto t \} \) where \( x, y, z, t \) are variables. Then \( l_{s_{2,1}} / \sigma = 2 \), \( l_{s_{2,2}} / \sigma = 3 \) and \( l_{s_{2,2}} / \sigma = l_t + t_1 \).

Note that \( l_{s_{2,1}} / \sigma = l_t \) if \( x \not\in \text{Dom}(\sigma) \).

If \( \text{Dioph}(\text{patt}_i) \) is a system of the form (1) then \( \text{Dioph}(\text{patt}_i) / \sigma \) is the system obtained by applying \( \sigma \) to all components \( l_{e_j}, l_{e_k} \) of (1).

Let \( \text{expr} \) be an input expression and \( \{ S, \text{patt}_i \} \) be the \( i \)-th approximation of a sequence-annotated pattern \( \text{patt} \). Suppose \( \text{expr} \) matches \( \text{patt}_i \) with the matcher \( \sigma \), and \( \text{Sol} = (l_{e_1}, \ldots, l_{e_q}, l_{e_1}, \ldots, l_{e_q}) \) is a solution of \( \text{Dioph}(\text{patt}_i) / \sigma \). The \( i \)-th cut of \( \text{expr} \) determined by \( \text{patt}_i \) and \( \text{Sol} \) is the expression \( \text{cut}_{\text{patt}_i, \sigma, \text{Sol}}(\text{expr}) \) obtained from \( \text{expr} \) by replacing every subterm \( h_{[a_1, \ldots, a_p]} \) of \( \text{expr} \) at some position \( p_{i,j} \in \text{Pos}_{\text{Sol}}(\text{patt}) \) with the subterm \( h_{[v_1, \ldots, v_q]} \) computed as follows: if the sub-pattern of \( \text{patt} \) at position \( p_{i,j} \) is \( h_{[v_1, \ldots, v_q]} \) then

(a) \( b_h = a_{v_h} \) if \( v_h \) is not a sequence variable pattern,
(b) \( b_h = \text{cut}[(u_{v_h}, \ldots, u_{v_h}, t_{v_h})] \) if \( v_h = x_{v_h} \)

for all \( h \in \{ 1, \ldots, q \} \), where \( u_1 = 1 \) and \( u_{v_h} + \ell_{v_h} = u_{v_h+1} \) if \( 1 \leq h \leq q \).

Example 7. Let \( P_i \) \( (1 \leq i \leq 3) \) be the approximations of the pattern \( P \) shown in Example 4, and

\[
\text{expr} = f[\text{cut}[1, 2], g[1, 2, h[a, b, a, b, c], h[d, c]]].
\]

Then \( \sigma = \{ x \mapsto r, y, z \mapsto t, s_{2,1} \mapsto 1, 1, 2, 1, h[a, a, b, a, b, c], s_{2,2} \mapsto r, d, c \} \) is a matcher of \( \text{expr} \) against \( P_2 \). The diophantine system of constraints

\[
\text{Dioph}(P_2) / \sigma : \begin{cases} 2 + 1 = 3, \\ l_x + l_t = 2, \\ 2 = t, \\ l_e \in [2, \infty], l_x \in [1, \infty] \end{cases}
\]

in variables \( (l_x, l_y, l_z, l_t, l_{s_{2,1}}, l_{s_{2,2}}) \) has a solution \( \text{Sol} = (2, 2, 1, 1, 3, 2) \) and

\[
\text{cut}_{P_2, \sigma}(\text{expr}) = f[\text{cut}[1, 2], g[\text{cut}[1, 2], h[a, b, a, b, c], h[\text{cut}[d, \text{cut}[c]]]].
\]

The values 2 for \( l_x \), 3 for \( l_{s_{2,1}} \) and 2 for \( l_{s_{2,2}} \) are inferred from the bindings of \( \sigma \), whereas the other values are inferred by solving the system \( \text{Dioph}(P_2) / \sigma \).

3.2 Extension 1

First, we must give a definition to \text{matcher} between \text{inp} and a sequence-annotated pattern \text{patt} which is sensitive to the intended meaning of our sequence annotations. Our definition takes into account the stratification of \text{patt} and relies on the following observation: \( \theta \) is a matcher between \( \text{inp}_0 \) and \( u(\text{patt}) \) iff \( \exists \sigma_i, \text{Sol}_0, \ldots, \sigma_n, \text{Sol}_n \) such that

(1) \( n = \#(\text{patt}), \ (2) \ \sigma_i \in \text{Matchers}(\text{inp}_{i-1}, \text{patt}_i), \ \text{Sol}_i \) is solution of the system \( \text{Dioph}(\text{patt}_i) / \sigma_i, \ \text{inp}_i = \text{cut}_{\text{patt}_i, \text{Sol}_i}(\text{inp}_{i-1}) \) whenever \( 1 \leq i \leq n \), and

(2) \( \theta \in \text{Matchers}(\text{inp}_n, \text{patt}^+) \).
According to this definition, all matchers can be enumerated by enumerating successive cuts $\text{inp}_i$ of the input until we reach a cut $\text{inp}_{g[p, d_j]}$ that matches against $\text{patt}^+$. This enumeration is driven by the enumeration of the partial matchers $\sigma_i$, which is in the order imposed by the annotations of sequence variables. We define the matcher between $\text{inp}_0$ and the sequence-annotated pattern $\text{patt}$ as the first matcher between $\text{inp}_0$ and $\text{patt}$ which is encountered by following this lookup strategy.

**Algorithm ExtPattMatch.**

The computation of a matcher between $\text{inp}$ and a sequence-annotated pattern $\text{patt}$ is triggered by the call $\text{ExtPattMatch}(\text{inp}, \text{patt}, 1)$. The auxiliary method $\text{Matcher}(\text{inp}, \text{patt})$ returns a matcher between $\text{inp}$ and $\text{patt}$ if there exists one, and fail if there is no matcher. Note that, by Remark 1, the calls of $\text{Matcher}()$ from inside $\text{ExtPattMatch}()$ are deterministic. The method $\text{uncut}(\text{inp})$ undoes all the cuts performed on an expression. For example, $\text{uncut}(f[\text{cut}[1, 2], g[\text{cut}[3, 4, 5, 6, 7]])$ yields $f[1, 2, g[3, 4, 5, 6, 7]]$.

**Algorithm ExtPattMatch**

**Description** Matching with sequence-annotated patterns

**Input**
- $\text{inp}$: input data
- $\text{patt}$: sequence-annotated pattern
- $i$: recursion index

**Output**
- the matcher $\sigma$ between $\text{uncut}(\text{inp})$ and $\text{patt}$, if $\sigma$ exists; fail, otherwise.

begin
if $i = \#(\text{patt}) + 1$
return $\text{Matcher}(\text{inp}, \text{patt}^+)$
else
$\sigma \leftarrow \text{Matcher}(\text{inp}, \text{patt}_i)$
if $\sigma = \text{fail}$
return fail
else
(* Enumerate the sequence $S \leftarrow \{\text{Sol}_1, \text{Sol}_2, \ldots, \text{Sol}_{p_i}\}$ of solutions of
d$\text{Dioph}(\text{patt}_i)/\sigma$ in the order of priority annotations given in $\text{patt}$ *)
j $\leftarrow 1$
next:
if $j > p_i$
return fail
else
$\sigma \leftarrow \text{ExtPattMatch}(\text{cut}_{\text{patt}_i,sol_j}(\text{inp}), \text{patt}, i + 1)$
if $\sigma = \text{fail}$
j $\leftarrow j + 1$
goto next
else
return $\sigma$
fi
fi
fi
end
Implementation Issues.

**Sequentica** provides an interpreter for sequence annotations. More precisely, in order to interpret function definitions with sequence annotations, we wrap them into a **Sequentica** call

\[
\text{Sequentica}[\text{defn}_1, \ldots, \text{defn}_n]
\]

where each \(\text{defn}_i\) is a sequence-annotated function definition of the form \(\text{lhs}_i := \text{rhs}_i\). This call translates the definitions \(\text{defn}_1, \ldots, \text{defn}_n\) into **Mathematica** code and adds them to the global environment of the **Mathematica** session.

It is often desirable *not* to make the definitions global, but to handle them as a list of transformation rules. We can program such lists as follows:

\[
R = \text{Sequentica}[	ext{lhs}_1 \mapsto \text{rhs}_1, \ldots, \text{lhs}_n \mapsto \text{rhs}_n]
\]

where \(\text{lhs}_1, \ldots, \text{lhs}_n\) are patterns with annotated sequence variables. In this case, the identifier \(R\) is bound to the list \(\{r_1, \ldots, r_n\}\) of transformation rules produced by translating the sequence-annotated transformation rules \(\text{lhs}_1 \mapsto \text{rhs}_1, \ldots, \text{lhs}_n \mapsto \text{rhs}_n\) into pure **Mathematica** code.

In the remainder of this section we explain how **Sequentica** performs the translation of sequence-annotated code into pure **Mathematica** code. Let \([r]\) denote the translation of a sequence-annotated transformation rule \(r = \text{lhs} \mapsto \text{rhs}\) into pure **Mathematica** code. Suppose \(n = \#(\text{lhs})\). We define \([r]\) := \(r_1\) where

\[
\begin{align*}
    r_{n+1} &= \text{lhs}^+ \mapsto \text{rhs}, \\
    r_n &= \text{lhs}_n : \text{lhs}_n \mapsto \text{Module}[\{v, \text{ok}\}, v; \{\text{SeqFit}[\text{lhs}_n, \text{info}_i, r'_{n+1}; \text{ok}]\}], \\
    \ldots \\
    r_1 &= \text{lhs}_1 : \text{lhs}_1 \mapsto \text{Module}[\{v, \text{ok}\}, v; \{\text{SeqFit}[\text{lhs}_1, \text{info}_1, r'_1; \text{ok}]\}].
\end{align*}
\]

each \(\text{info}_i\) is a suitable encoding of the system \(\text{Dioph(}\text{lhs}_i)\) and each \(r'_{i+1}\) is a fresh variant\(^2\) of the transformation rule \(r_{i+1} (1 \leq i \leq n)\). We use fresh variants to avoid the *variable capture* problem that may happen upon term rewriting. According to [6], the meaning of a transformation rule \(r_i (1 \leq i \leq n)\) is:

replace by \(v\) an instance \(\text{lhs}_i / \theta\) of \(\text{lhs}_i\) only if \(\text{SeqFit}[\text{lhs}_i / \theta, \text{info}_i, r'_{i+1}]\) yields the pair \(\{v, \text{ok}\}\) with \(\text{ok} = \text{True}\).

\(\text{SeqFit}[\text{imp}, \text{info}_i, r'_{i+1}]\) with \(r'_{i+1} = \text{lhs}_i \mapsto \text{rhs}_i \mapsto \text{rhs}_{i+1}\) is designed to compute the pair \(\{v, \text{ok}\}\) where

- \(\text{ok}\) is \text{True} if \(\exists \text{Sol solution of the system } \text{Dioph(}\text{lhs}_i) / \sigma\) where \(\sigma\) is the matcher between \(\text{imp}\) and \(\text{lhs}_i\), such that \(\text{cut}_{\text{lhs}_i, \sigma}^i(\text{imp})\) can be rewritten with \(r'_{i+1}\) at position \(A\). Otherwise, \(\text{ok}\) is \text{False},

- if \(\text{ok}\) is \text{True} then \(v\) is the result of rewriting \(\text{cut}_{\text{lhs}_i, \sigma}^i(\text{imp})\) with \(r'_{i+1}\) at position \(A\), where \(\text{Sol}\) is the first solution of \(\text{Dioph(}\text{lhs}_i) / \sigma\) (w.r.t. the ordering imposed by the sequence annotations) which enables such a rewrite step. If \(\text{ok}\) is \text{False} then \(v\)

---

\(^2\) A fresh variant of a transformation rule \(r\) is an expression obtained from \(r\) by renaming apart all pattern variables of \(r\). E.g., a fresh variant of \(f[x \mapsto y, z] \mapsto \{y, x\}\) is \(f[z \mapsto v, y] \mapsto \{v, y\}\).
is irrelevant\(^3\) because, by definition, rule \(r_i\) can not rewrite \(inp\) at position \(A\). (In this subcase we define \(v\) to be \(inp\).)

for all \(i \in [1..n]\). The arguments \(info_i\) of \(\text{SeqFit}[]\) are computed at compile time, whereas the substitution \(\sigma\) is computed at runtime. Thus, the enumeration of the solutions of the system \(\text{Dioph}(lhs_i) / \sigma\) is performed at runtime.

\(\text{Sequentica}\) relies heavily on an iterative solver for systems of linear diophantine constraints (cf. [2]). This means that the solver does not compute all solutions at once, but enumerates them in the order imposed by the sequence variable annotations. Actually, it would be unreasonable to rely on a solver for linear diophantine constraints which computes all solutions at once because

1. the space of solutions can be very large, and
2. we are not interested in all solutions but only in the \(\text{first}\) one which realizes a matching (where \(\text{first}\) is defined w.r.t. the order which is inferred from the priorities attached to sequence variables).

Our current implementation of the linear diophantine solver resembles the constraint solving algorithm described in [2, Sect. 3.3]. For efficiency reasons, this solver should be fast. We are still looking for better approaches to enumerate the space of solutions.

So far, we have explained how the transformation rules with sequence annotations are encoded in \(\text{Mathematica}\), but didn’t say how sequence-annotated definitions are translated into pure \(\text{Mathematica}\) code. A sequence-annotated definition \(lhs := \text{rhs}\) is translated by \(\text{Sequentica}\) in two steps:

1. Compute the translation \([lhs \mapsto \text{rhs}]\) of \(\text{lhs} \mapsto \text{rhs}\) into pure \(\text{Mathematica}\) code. As we already explained, \([lhs \mapsto \text{rhs}]\) is a transformation rule of the form \(\text{lhs}_{1} := \text{lhs}_{1} \mapsto \text{t}\).
2. Generate the corresponding \(\text{Mathematica}\) definition \(\text{lhs}_{1} : \text{lhs}_{1} := \text{t}\).

### 3.3 Extension 2

There is a notable algorithmic distinction between the first extension and the second one. The first extension requires to enumerate the solutions of systems of linear diophantine constraints in a certain order (which is controlled by the programmer) until we reach one which induces the desired matcher, and then stop. The second extension also requires to enumerate the solutions of systems of linear diophantine equations, because these solutions induce possible candidates for pattern matching with sequence variables. However, in this case we can not stop when we reach a matcher because we have no knowledge that we have reached the optimum we are looking for. Therefore, we must explore the \(\text{whole}\) space of solutions of our systems of linear diophantine constraints and maintain a buffer which stores the binding(s) found so far for which optimum is achieved.

The problem posed by this extension can be formalized as follows:

Given

- An input expression \(inp\), a (possibly sequence-annotated) pattern \(patt\), a positive integer \(n\), an expression \(\text{optim}\) to be optimized, a predicate \(\text{test}\) which is total on the set of instances \([\text{optim} / \theta \mid \theta \in \text{Matcher}(\text{inp}, u(patt))]\), and a target expression \(\text{expr}\)

\(^3\) This is so because we are interested only in calls of \(\text{SeqFit}[]\) which are triggered by the application of some transformation rule \(r_i\).
Find The list \( L = \{ \text{expr}/\theta_1, \ldots, \text{expr}/\theta_{\min(n,p)} \} \) where \( \{\theta_1, \ldots, \theta_p\} \) is the list of matchers between \( \text{inp} \) and \( \text{patt} \), ordered such that \( 1 \leq i < p \) implies \( \text{test}(\text{expr}/\theta_i, \text{expr}/\theta_{i+1}) = \text{True} \).

Algorithm \text{BestFits}

The algorithm which solves this problem is straightforward to specify in terms of an auxiliary algorithm which enumerates all the matchers between \( \text{expr} \) and \( \text{patt} \). The algorithm \text{BestFits} shown below yields

- fail if \( p = 0 \) (i.e., there are no matchers between \( \text{inp} \) and \( \text{patt} \)),
- the list \( \{ \text{expr}/\theta_1, \ldots, \text{expr}/\theta_{\min(n,p)} \} \) if \( p > 0 \)

and makes use of three auxiliary methods: \text{getPos()}\text{, insert()}\text{ and replace()}. The call \text{getPos(}\{v_1, \ldots, v_n\}, \text{v, test}\text{)} yields the largest index \( j \) for which \( \text{test}(v_j, v) = \text{True} \), and 0 if such a \( j \) does not exist. \text{insert(}l, \text{v, i}\text{)} inserts \( v \) at position \( i \) in list \( l \). If \( l = \{ \} \) and \( i = 1 \) then \text{insert(}l, \text{v, i}\text{)} yields \( \{v\} \). \text{replace(}l, \text{v, i}\text{)} replaces the \( i \)-th element of list \( l \) with \( v \).

Algorithm \text{BestFits}

\textbf{Description}
Compute optimum values defined by a total relation \( \text{test} \) on instances of expressions ranging over a given syntax domain

\textbf{Input}
- \text{inp}: input expression
- \text{patt}: sequence-annotated pattern
- \( n \): desired number of best results
- \text{expr}: expression to be instantiated
- \text{optim}: expression to be optimized
- \text{test}: test predicate which characterizes the optimum

\textbf{Output}
- fail, if \( p = 0 \)
- \( \{ \text{expr}/\theta_1, \ldots, \text{expr}/\theta_{\min(n,p)} \} \) otherwise
- where \( \{\theta_1, \ldots, \theta_p\} \) is an enumeration of matchers between \( \text{inp} \) and \( \text{patt} \)
- such that \( 1 \leq i < \min(n, p) \) implies \( \text{test}(\text{optim}/\theta_i, \text{optim}/\theta_{i+1}) \)

begin
(* Initialize the lists \( L \) and \textit{optList} *)
\( L \leftarrow \{\} \), \textit{optList} \leftarrow \{\}
(* Generate an enum. \( \mathcal{E} = \{\theta'_1, \ldots, \theta'_p\} \) of matchers between \( \text{patt} \) and \( \text{inp} \) *)
if \( \mathcal{E} = \{\} \) then
return fail
else
for \( i \leftarrow 1 \) to \( p \) do
\( \text{id}x \leftarrow \text{getPos(optim}/\theta'_i, \text{optList, test)} + 1 \)
if \( \text{id}x < n \)
\( L \leftarrow \text{insert(}L, \text{expr}/\theta'_i, \text{id}x\text{)} \)
\( \text{optList} \leftarrow \text{insert(}\text{optList, optim}/\theta'_i, \text{id}x\text{)} \)
elseif \( \text{id}x = n \)
\( L \leftarrow \text{replace(}L, \text{expr}/\theta'_i, n\text{)} \)
\( \text{optList} \leftarrow \text{replace(}\text{optList, optim}/\theta'_i, n\text{)} \)
fi
od
return L
fi
end

The auxiliary data structure optList is maintained and used to control the positions where to insert new values in L such that, if \( E_q = \{ \theta'_1, \ldots, \theta'_n \} \) is the list of markers enumerated so far, then \( L = \{ expr/\theta_1, \ldots, expr/\theta_{\min(n,q)} \} \) for a rearrangement \( \{ \theta_1, \ldots, \theta_n \} \) of \( E_q \) which satisfies the requirement \( 1 \leq i < q \Rightarrow test(\text{optim} / \theta_i, \text{optin} / \theta_{i+1}) \).

An algorithm which enumerates the list \( E \) of all markers between \( \text{inp} \) and \( \text{patt} \) can be obtained by adjusting the algorithm \text{ExtPattMatch} as follows: instead of resuming the algorithm with the first matcher \( \sigma \) found, we continue to look up for matchers until we reach the condition \( j > p \).

Implementation Issues.

The \textit{Sequentica} package supports the second extension too: invocations of \textit{Sequentica}] recognize the definitions and the transformation rules which have definitions of the form \text{BestFits}[n, expr, optim, test] or \text{BestFit}[expr, optim, test] and translates them accordingly.

First, we describe briefly how \textit{Sequentica}] translates a transformation rule \( r \) of the form \( \text{lhs} : \rightarrow \text{BestFits}[n, expr, optim, test] \) into pure \textit{Mathematica} code.

Let \( n = \#(\text{lhs}) \). The pure \textit{Mathematica} code produced by the translation can be described in terms of two auxiliary functions SeqBF and SeqBF0: it is the expression \( rr \) defined recursively as follows:

\[
rr_{n+1} = \text{lhs}^+ : \rightarrow \text{BestFits}[n, expr, optim, test],
rr_n = \text{lhs}_n : \text{lhs}_n : \rightarrow \text{Module}[\{v,ok\},\{v,ok\} = \text{SeqBF0}[\text{lhs}_n, \text{info}_n, \text{rr}_{n+1}]; ok],
... 
rr_2 = \text{lhs}_2 : \text{lhs}_2 : \rightarrow \text{Module}[\{v,ok\},\{v,ok\} = \text{SeqBF}[\text{lhs}_2, \text{info}_2, \text{rr}_3]; ok],
rr_1 = \text{lhs}_1 : \text{lhs}_1 : \rightarrow \text{Block}[\{L = \}, \text{optList} = \{\}],
\text{Module}[\{v,ok\},\{v,ok\} = \text{SeqBF}[\text{lhs}_1, \text{info}_1, r_2]; ok]]
\]

where each \( rr_i \) is a fresh variant of \( r_i (1 < i \leq n + 1) \). Note the similarities and differences between this translation and the translation defined at page 72 for the first extension: \( rr \) is obtained from \( r \) by replacing \text{SeqFit} with SeqBF, \( r' \) with \( rr' \), and by wrapping the right hand side of the transformation rule in a Block construct; \( rr \) is obtained from \( r_n \) by replacing \text{SeqFit} with SeqBF0 and \( r' \) with \( rr' \); \( rr \) is defined in the same way as \( r_n \), by replacing with \text{lhs}^+ the left hand side of \( r \); and if \( 2 < i < n \) then \( rr_i \) is obtained from \( r_i \) by replacing \text{SeqFit} with SeqBF and \( r' \) with \( rr' \) with \( rr' \).

The purpose of the Block construct is to provide dynamic scoping for the buffers \( L \) and \text{optList} of algorithm \textit{BestFits}. These buffers are updated upon applying the transformation rule \( rr'_{n+1} \) from inside a SeqBF0[] call. (See below.)

The arguments \( \text{info}_i \) (1 \( i \leq n \)) have the same meaning as for the calls of SeqFit[]:

\[
\text{SeqBF}[\text{inp}, \text{info}_i, rr'_{i+1}] \text{ with } rr'_{i+1} = \text{lhs}_{i+1} : \rightarrow \text{rhs}_{i+1} \text{ is designed to compute the pair } \{v, ok\}
\]

- \( ok \) is True if \( \exists \text{Sol solution of the system Dioph(lhs)}/.\sigma \) where \( \sigma \) is the matcher between \( \text{inp} \) and \( \text{lhs} \), such that cut_{inp,\sigma}(inp) can be rewritten with \( rr'_{i+1} \) at position \( A \). Otherwise, \( ok \) is False.
• if \( ok \) is \textbf{True} then \( v \) is the list of instances of \( expr \) accumulated so far in the buffer \( L \) as a side effect of (1) enumerating all solutions \( \text{Dioph}(\text{lhs}_i)/\sigma \) where \( \sigma \) is the matcher between \( \text{inp} \) and \( \text{lhs}_i \) and (2) trying to rewrite \( \text{cut}_{\text{lhs}_i, \sigma}(\text{inp}) \) with \( \text{rr}'_{i+1} \) at position \( A \). If \( ok \) is \textbf{False} then \( v \) is irrelevant because in this case rule \( \text{rr}_i \) can not rewrite \( \text{inp} \) at position \( A \).

\( \text{SeqBF}[\text{inp}, \text{info}_n, \text{rr}'_{n+1}] \) is designed to (1) perform the same computation as \( \text{SeqBF}[\text{inp}, \text{info}_n, \text{rr}'_{n+1}] \), and (2) to update the buffers \( L \) and \( \text{optList} \) whenever \( \text{rr}'_{n+1} \) rewrites \( \text{cut}_{\text{lhs}_n, \sigma}(\text{inp}) \) at position \( A \). The update of these buffers is as shown in the algorithm \text{BestFits} of Sect. 3.3.

The translation of a function definition of the form \( \text{lhs} := \text{BestFit}[\ldots] \) (resp. \( \text{lhs} := \text{BestFits}[\ldots] \)) into pure \texttt{Mathematica} code is carried out in two steps: first, we compute the translation \( \text{lhs1;lhs1} := t \) of \( \text{lhs} := \text{BestFit}[\ldots] \) (resp. \( \text{lhs} := \text{BestFits}[\ldots] \)), and next we add the definition \( \text{lhs1;lhs1} := t \) to the global environment.

## 4 Conclusion and Future Work

Starting from the observation that the current support for programming with sequence variables is unsatisfactory, we have proposed two extensions and explained how they can be translated into pure \texttt{Mathematica} code.

We are aware of the existence of other useful patterns for programming with sequence variables. Therefore, we have decided to maintain a site at

\texttt{http://www.score.is.tsukuba.ac.jp/~mmarin/Sequentica}

and to freely distribute a \texttt{Mathematica} package with implementations of convenient extensions for programming with sequence variables. More convincing examples of programming with the extensions of \textit{Sequentica} are provided in the notebook \texttt{SequenticaDemo.nb} which can be downloaded from our site.

We envision that programming with sequence variables will become a powerful programming style with wide applications in research fields which rely on identifying and processing pattern similarities in large data structures. As with the other major programming styles, programming with sequence variables could be greatly improved if we succeed in identifying recurring programming patterns and to extend the programming language with facilities to represent and interpret them accordingly.

## References


Satisfiability of Structural Subtype Constraints

Joachim Nehren and Tim Priesnitz

Programming Systems Lab,
Universität des Saarlandes, Saarbrücken, Germany
www.ps.uni-sb.de/~{niehren,tim}

Abstract. We investigate the satisfiability problem of subtype constraints for the structural case. Our idea is to characterize subtype constraints by formulae in propositional dynamic logic. This approach yields a new result: restricted to crown signatures and models of possibly infinite trees satisfiability of structural subtype constraints is DEXPTIME-complete. This result subsumes and extends the known upper DEXPTIME bound given by Wand and Tiuryn [TW93].

1 Introduction

Mitchell [Mit96] has introduced a subtype relation on types to enrich type languages. A subtype relation $\tau_1 \leq \tau_2$ means that all functions in a program that expect an argument of type $\tau_1$ are sufficiently polymorphic so that they can also be applied to values of the subtype $\tau_1$. Subtyping algorithms invariably involve conjunctions of subtype constraints $\tau_1 \leq \tau_2$, where the types $\tau_i$ are built from type variables and type constructors of a given signature. Type variables are interpreted in the model of finite, rational, or possibly infinite trees. Two logical operations on subtype constraints were investigated: satisfiability [FM90,Mit91,AC93,EST95] and entailment [Pot98,Reh98,HR97,HR98,NP99,NP03]. Their complexities strongly depend on the given signature. For example subtype satisfiability can be checked in cubic time for some type languages [KP94,PWO97].

In this talk we focus on the satisfiability problem of subtype constraints in the structural case, i.e. $\tau_1 \leq \tau_2$ implies that the trees of $\tau_1$ and $\tau_2$ have the same shape. Further, we assume that the signature supports constants equipped with an ordering which does not form a lattice. For this case Wand and Tiuryn [TW93] have given an upper DEXPTIME bound by reducing it to the emptiness of Büchi tree automata which is known to be solvable in polynomial time. Thereby their reduction determines a deterministic blow up.

Our new idea is to characterize subtype constraints by formulae in propositional dynamic logic. Propositional dynamic logic, in short PDL, talks about intensional aspects of graphs. It goes back to Fischer and Ladner, who restricted Pratt’s dynamic logic to the propositional fragment. We prove that restricting PDL to tree models preserves its DEXPTIME-complete satisfiability result. Next we translate a deterministic version of PDL [BAHP82] to the mentioned subtype problem and vice versa.

It is work in progress whether this approach can also be applied to other cases of satisfiability problems of subtype constraints.
References


A Prefix Notation and Unification Algorithm for Encoding Modal Logics

Adrian Williams and Jim Cunningham

Department of Computing
Imperial College
Email: \{aw3, rjc\}@doc.ic.ac.uk

1 Introduction

When reasoning about modal logics, it is necessary to deal with the various properties of modal operators in different logics. When dealing with logics with well understood frame properties, it is common practice to try to encode (part of) the semantics at a syntactic level. Existing methods include expressing the accessibility relation classically (as relations and/or functions e.g. \cite{8}) or by using prefixes (\cite{1,3,6,10}). The exact nature of a prefix varies between authors but they all have the basic premise that a prefix is a string that represents paths of worlds. Each formula is prefixed by one of these strings to indicate at which world(s) it must hold. Beyond this, each prefix system encodes a different amount of semantic information into a prefix as well as differing in terms of presentation.

In this paper we present a prefix notation that encapsulates the nature of a number of different prefix systems. It is our intention to provide a clear consistent notation that not only can be used in different proof systems but which can be reasoned about purely on its own terms without being concerned with the modal logics that are being encoded. Obviously, a complete proof system needs to be concerned about such matters but when it comes to sub-problems such as unification (the task of determining whether two prefixes represent common worlds) we wish to be able to solve this task by a straightforward analysis of the structure of the prefixes. One of the advantages of such a modular approach is that optimisations can be made to unification algorithm without affecting or even needing to understand the rest of the proof system.

2 Prefixes

We begin by presenting a family of prefixes, $\Sigma_{XY}$. Each set of prefixes is distinguished by the length of the contained prefixes and the symbols that are used to construct them.

**Definition 1.** The set of general constants, $C$, is the set $\{c, C_\ast, C_+, C_+\}$

**Definition 2.** The sets $V, V_\ast, V_+$ and $V_*$ are disjoint countable sets of variable names.

**Definition 3.** $V$ is the set of all variable names ($V = V \cup V_\ast \cup V_+ \cup V_*$).

We will use variable names of the form $v, v_\ast, v_+, v_*$ for variables that belong to the sets $V, V_\ast, V_+$ and $V_*$ respectively.

A prefix will be some sequence of the symbols $\mathbb{N}, V$ and $C$. 
In order to describe the restrictions on the length of allowed prefixes in a particular prefix set we will use the following symbols:

- prefixes must have a length of 0 or 1
- prefixes must have a length of 1
- prefixes can be any finite non-empty sequence
- prefixes can be any finite sequence
- prefixes can be any sequence (finite or infinite)

Allowed symbols within a particular prefix set will be determined by subscripting the set name with the names of the allowed symbol sets.

For example, the prefix set $\Sigma_{N\cup\mathcal{V}}$ is the set of all sequences of the symbols $N \cup \mathcal{V}$ that have the length 0 or 1.

The most important prefix set is the set $\Sigma_\ast$, as it is this set which is used to define the meaning of more interesting prefixes. We call $\Sigma_\ast$, the set of basic prefixes and will usually use the shorthand $\Sigma$.

The other prefix set of particular interest is the set $\Sigma_{N\cup\mathcal{C}}$ as all other sets are subsets. We call this set the set of general prefixes and will use the shorthand $\Sigma_\ast$.

**Definition 4.** A basic prefix is a potentially infinite sequence of natural numbers ($\mathbb{N}$).

**Definition 5.** $\Sigma$ is the set of all basic prefixes ($= \Sigma_{\mathbb{N}}$).

**Definition 6.** A general prefix is a finite sequence of the symbols $N \cup C \cup \mathcal{V}$.

**Definition 7.** $\Sigma_\ast$ is the set of all general prefixes ($= \Sigma_{N\cup\mathcal{C}}$).

For reasoning about these prefixes we use a number of common operators.

We define ‘@’ as our ‘head and tail’ operator, and ‘;’ as our concatenation operator. i.e.

$$@h[T] = @h;T$$

$|\sigma|$ is used to indicate the length of the prefix $\sigma$, e.g.

$$|(1, 2, 1)| = 3$$

$\sigma(i)$ refers to the ith value in $\sigma$, e.g.

$$\sigma(3, 2, 1)(3) = 1$$

### 3 Unification

As we have already stated these prefixes are used to represent worlds at which formulas are known to hold. It is a requirement of any refutation system that we are able to determine when we have two contradictory statements. In this case, contradictory means that we have two contradictory formulas holding at the same world. Therefore, it is necessary to be able to decide when two prefixes represent common worlds. This process is referred to as prefix unification.
3.1 Semantics

Basic Prefixes

Basic prefixes are our core prefix structure and each is unique. As such, unification between basic prefixes is trivial. Two basic prefixes unify if they are equal.

\( \Sigma_{NV} \) prefixes

In our prefix system variables are typed. Variables from the sets \( V, V^*, V_{\delta}, \) and \( V_{\varepsilon} \) can be assigned values from the sets \( \Sigma_{NV}, \Sigma_{NV}, \Sigma_{NV_{\delta}}, \) and \( \Sigma_{NV_{\varepsilon}} \) respectively. Taking these restrictions into account unification between \( \Sigma_{NV} \) prefixes is essentially a variation on string unification.

**Definition 8.** A prefix substitution \( \theta : \mathcal{V} \rightarrow \Sigma_{NV} \) is a function mapping variables to prefixes that satisfies the following constraints:

\[
\begin{align*}
\theta(v) & \in \Sigma_{NV1} \\
\theta(v^*) & \in \Sigma_{NV^*} \\
\theta(v^+) & \in \Sigma_{NVV^*} \\
\theta(v_+) & \in \Sigma_{NV_+} \\
\theta(v_+) & \in \Sigma_{NV_+} 
\end{align*}
\]

A substitution, \( \theta \), can be represented by a set of \((\text{variable}, \text{prefix})\) pairs, \( \{v\sigma | \theta(v) = \sigma \text{ and } v \neq v\} \).

The application, \( \theta(\sigma) \), (usually written \( \sigma\theta \)) of a substitution, \( \theta \), to a prefix, \( \sigma \), leads to the prefix, \( \sigma' \), where the variables in \( \sigma \) are replaced by their values under \( \theta \).

More formally

\[
\begin{align*}
\langle \rangle \theta & = \langle \rangle \\
\langle i | T \rangle \theta & = \langle i \rangle ; \langle T \rangle \theta \\
\langle C : C | T \rangle \theta & = \langle C \rangle ; \langle T \rangle \theta \\
\langle v : \forall | T \rangle \theta & = \theta(v) ; \langle T \rangle \theta 
\end{align*}
\]

**Definition 9.** The composition, \( \gamma(\theta) \), of two substitutions, \( \theta \) and \( \gamma \), yields the substitution

\[
\{v'\gamma(\sigma) \mid v' \sigma \in \theta \text{ and } \langle v \rangle \neq \gamma(\sigma) \} \cup \{v'\sigma \mid v' \sigma \in \gamma \text{ and } v' \sigma \notin \theta \text{ for some } \tau \}
\]

**Definition 10.** A substitution, \( \gamma \), is called an instance of a substitution, \( \theta \), (or \( \theta \) is more general than \( \gamma \)) iff there is a substitution, \( \rho \), s.t. \( \gamma = \rho(\theta) \)

Two \( \Sigma_{NV} \) prefixes, \( \sigma \) and \( \tau \), unify iff \( \exists \theta . \sigma\theta = \tau\theta \).

\( \Sigma_{NC} \) prefixes

In addition to our typed variables we have the special constant symbols in \( C \). Unlike the natural numbers, each symbol in \( C \) represents multiple values and unlike variables we never actually consider them being assigned a particular value. For example, the prefix \( \langle 1, c \rangle \) represents all prefixes of the form \( \langle 1, n \rangle \) where \( n : N \).

In an alternative presentation we may have chosen to replace each occurrence of our \( C \) symbols with variables of an appropriate type, where each variable is used only once within its scope. However, as we know that these variables are used only once, we
also know that for any substitutions we make in order to unify two prefixes we could promptly forget any part that involved these variables. This is because we know that we will never use them again. In order to take advantage of this fact in our unification algorithm, we would need some mechanism that reflects this property. Instead, we decided on using our \( \mathcal{C} \) symbols which are never actually assigned a value but can match any value of the correct type.

When defining unification of prefixes containing \( \mathcal{C} \) symbols, we don’t think in terms of a unifying substitution. Instead, we look at the sets of basic prefixes they represent and determine if they overlap. For this purpose, we use a grounding function, \( \mathcal{G} : \Sigma_{NC} \to \Sigma \), that takes a \( \Sigma_{NC} \) prefix and returns the set of basic prefixes it represents.

**Definition 11.** The operator \( \mathcal{G} : \Sigma_{NC} \to \Sigma \) is defined as follows:

\[
\mathcal{G}(\langle \rangle) = \{ \langle \rangle \} \\
\mathcal{G}(\langle i \rangle \mathcal{T}) = \{ \langle i \rangle \}; \mathcal{G}(\mathcal{T}) \\
\mathcal{G}(\langle \mathcal{T} \rangle) = \Sigma_{N1}; \mathcal{G}(\mathcal{T}) \\
\mathcal{G}(\langle C_i \mathcal{T} \rangle) = \Sigma_{N1}; \mathcal{G}(\mathcal{T}) \\
\mathcal{G}(\langle C_i \mathcal{A} \mathcal{T} \rangle) = \Sigma_{N1}; \mathcal{G}(\mathcal{T}) \\
\mathcal{G}(\langle C_i \mathcal{A} \mathcal{B} \mathcal{T} \rangle) = \Sigma_{N1}; \mathcal{G}(\mathcal{T})
\]

where the operator \( ' ; ' \) is extended over sets of prefixes as follows

\[ S; T = \{ s; t | s \in S \land t \in T \} \]

**Definition 12.** A prefix, \( \sigma \), is an instance of a prefix, \( \tau \), if \( \sigma \in \mathcal{G}(\tau) \)

For two \( \Sigma_{NC} \) prefixes, \( \sigma \) and \( \tau \), \( \sigma \) and \( \tau \) unify iff \( \mathcal{G}(\sigma) \cap \mathcal{G}(\tau) \neq \emptyset \).

A unification of \( \sigma \) and \( \tau \) is a \( \Sigma_{NC} \) prefix, \( \pi \) s.t. \( \mathcal{G}(\pi) \subseteq \mathcal{G}(\sigma) \cap \mathcal{G}(\tau) \).

**General Prefixes**

As general prefixes contain both variables and \( \mathcal{C} \) symbols, our definition of unification needs to combine both \( \theta \)-substitution and the grounding function, \( \mathcal{G} \). As \( \mathcal{G} \) is only defined on prefixes that do not contain variables, we need to be able to reduce a general prefix into a \( \Sigma_{NC} \) prefix. For this we define the idea of a prefix mapping.

**Definition 13.** A prefix mapping, \( m : \mathcal{V} \to \Sigma \) is a mapping between prefix variables and basic prefixes such that

\[
m(v) \in \Sigma_{N1} \\
m(v_i) \in \Sigma_{N*} \\
m(v_+) \in \Sigma_{N+} \\
m(v_\ast) \in \Sigma_{N*}
\]

We apply a prefix mapping, \( m \), to a prefix, \( \sigma \), by replacing each occurrence of a variable, \( \nu \) in \( \sigma \) with \( m(\nu) \). We write the result as \( \sigma_m \). This is analogous to the application of a substitution \( \theta \) to \( \sigma \), \( \sigma\theta \).

If we are considering two general prefixes, \( \sigma \) and \( \tau \), for any prefix mapping \( m \), we are interested in the set \( \mathcal{G}(\sigma_m) \cap \mathcal{G}(\tau_m) \)

Two general prefixes, \( \sigma \) and \( \tau \), unify iff \( \exists m \mathcal{G}(\sigma_m) \cap \mathcal{G}(\tau_m) \neq \emptyset \).

\( \pi \) is a unification of \( \sigma \) and \( \tau \) iff \( \exists m \mathcal{G}(\sigma_m) \subseteq \mathcal{G}(\sigma) \cap \mathcal{G}(\tau) \).
3.2 Unification Algorithm

Below, we give the details of our unification algorithm. $v : \Sigma_* \times \Sigma_* \rightarrow \mathcal{P}\Sigma_*$ takes two general prefixes and returns a set of general prefixes that represent their unification.

If we extend $\mathcal{G}$ over sets of prefixes such that

$$\mathcal{G}(\Pi) = \bigcup_{\pi \in \Pi} \mathcal{G}(\pi)$$

then for any prefix mapping, $m$,

$$\mathcal{G}(v(\sigma, \tau)_m) = \mathcal{G}(\sigma_m) \cap \mathcal{G}(\tau_m)$$

For any possible combination of pairs of prefixes, $\langle h_1|\sigma\rangle, \langle h_2|\tau\rangle$, we only provide the rule for one ordering of the pair. Given the rule for $\langle h_1|\sigma\rangle, \langle h_2|\tau\rangle$, the rule for $\langle h_2|\tau\rangle, \langle h_1|\sigma\rangle$ is self evident.

Also in this particular presentation we do not retain the implied substitution(s). Instead, any substitutions required are applied during the unification process and we only return the final unifications.

If $v(\sigma, \tau) \neq \emptyset$ we say that $\sigma$ and $\tau$ unify. For convenience, we will also use $v$ as a predicate such that $v(\sigma, \tau)$ iff $\sigma$ and $\tau$ unify.
Definition 14. We define our unification algorithm, $v$ as follows:

\[
v(\emptyset, \emptyset) = \{\emptyset\}
\]

\[
v(h, g) = v(h, g) \quad \text{if } h \notin \{C_*, C_\uparrow\}
\]

\[
v(h, g) = v(h, g) \quad \text{if } h \in V_\downarrow \cup V_\uparrow
\]

\[
v(h, g) = \{\emptyset\}
\]

\[
v(h, g) = \{h\}; v(g, h) \quad \text{if } h \in \mathbb{N} \cup \mathbb{V}
\]

\[
v(h, g) = \{\emptyset\}; v(h, g) \quad \text{otherwise}
\]

$v((i\sigma), (i\tau)) = v((i\sigma), (i\tau)) \cup \{i\}; v((i\tau), (i\sigma))$

$v((i\sigma), (C_i\tau)) = \{\emptyset\}; v((i\tau), (C_i\sigma))$

$v((i\sigma), (C_i\rho)) = \{\emptyset\}; v((i\tau), (C_i\rho))$

$v((i\sigma), (C_i\tau)) = \{\emptyset\}; v((i\tau), (C_i\sigma))$

$v((i\sigma), (v_i\tau)) = \{\emptyset\}; v((i\tau), \{v_i\} \cup \{t\}; v((i\tau), (v_i\tau))$

$v((i\sigma), (v_i\rho)) = \{\emptyset\}; v((i\tau), \{v_i\} \cup \{t\}; v((i\tau), (v_i\rho))$

$v((i\sigma), (C_i\tau)) = \{\emptyset\}; v((i\tau), (C_i\sigma))$

$v((i\sigma), (C_i\tau)) = \{\emptyset\}; v((i\tau), (C_i\sigma))$

$v((i\sigma), (C_i\tau)) = \{\emptyset\}; v((i\tau), (C_i\sigma))$

$v((i\sigma), (v_i\tau)) = \{\emptyset\}; v((i\tau), \{v_i\} \cup \{t\}; v((i\tau), (v_i\tau))$

$v((i\sigma), (v_i\rho)) = \{\emptyset\}; v((i\tau), \{v_i\} \cup \{t\}; v((i\tau), (v_i\rho))$

$v((i\sigma), (C_i\tau)) = \{\emptyset\}; v((i\tau), (C_i\sigma))$

$v((i\sigma), (C_i\tau)) = \{\emptyset\}; v((i\tau), (C_i\sigma))$

$v((i\sigma), (v_i\tau)) = \{\emptyset\}; v((i\tau), \{v_i\} \cup \{t\}; v((i\tau), (v_i\tau))$

$v((i\sigma), (v_i\rho)) = \{\emptyset\}; v((i\tau), \{v_i\} \cup \{t\}; v((i\tau), (v_i\rho))$

$v((i\sigma), (C_i\tau)) = \{\emptyset\}; v((i\tau), (C_i\sigma))$

$v((i\sigma), (C_i\tau)) = \{\emptyset\}; v((i\tau), (C_i\sigma))$

$v((i\sigma), (v_i\tau)) = \{\emptyset\}; v((i\tau), \{v_i\} \cup \{t\}; v((i\tau), (v_i\tau))$

$v((i\sigma), (v_i\rho)) = \{\emptyset\}; v((i\tau), \{v_i\} \cup \{t\}; v((i\tau), (v_i\rho))$

$v((i\sigma), (C_i\tau)) = \{\emptyset\}; v((i\tau), (C_i\sigma))$

$v((i\sigma), (C_i\tau)) = \{\emptyset\}; v((i\tau), (C_i\sigma))$
\[
v(C_\sigma, \langle v, \tau \rangle) = v(\sigma, \langle v, \tau \rangle) \cup \{\langle v \rangle\}; v(C_\sigma, \tau)
\]
\[
v(C_\sigma, \langle v, u \rangle) = v(\sigma, \langle v, u \rangle) \cup v(\sigma, \langle u, v \rangle) \cup v(C_\sigma, \tau)
\]
\[
v(C_\sigma, \langle v, \tau \rangle) = v(\sigma, \langle v, \tau \rangle) \cup \{\langle v \rangle\}; v(C_\sigma, \tau)
\]
\[
v(C_\sigma, \langle v, u \rangle) = v(\sigma, \langle v, u \rangle) \cup \{\langle v \rangle\}; v(C_\sigma, \tau)
\]
\[
v(C_\sigma, \langle v, \tau \rangle) = v(\sigma, \langle v, \tau \rangle) \cup \{\langle v \rangle\}; v(C_\sigma, \tau)
\]
\[
v(C_\sigma, \langle v, u \rangle) = v(\sigma, \langle v, u \rangle) \cup \{\langle v \rangle\}; v(C_\sigma, \tau)
\]
\[
v(C_\sigma, \langle v, \tau \rangle) = v(\sigma, \langle v, \tau \rangle) \cup \{\langle v \rangle\}; v(C_\sigma, \tau)
\]

where \(u, u_+, u_+, u, v, w, w_+, \) and \(w_+\) are fresh variables, \(i\) and \(j\) are distinct numbers and \(v\) and \(v'\) are distinct variables.

4 Comparison With Other Prefix Systems

4.1 Fitting’s prefixes

In his prefixed tableau system ([2]), Fitting defined a prefix as a non-empty finite sequence of positive integers. In our notation this would be \(\Sigma_{\mathbb{Z}^+}\). In terms of expressivity, this is the same as \(\Sigma_{\mathbb{N}^+}\).

Fitting doesn’t actually talk about the idea of unification. Instead he refers to accessibility. The definition of accessibility varies depending on the logic under consideration. The table below describes different types of accessibility.

- general \(\sigma n\) is accessible from \(\sigma\) for every \(n\)
- symmetry \(\sigma\) is accessible from \(\sigma n\) for every \(n\)
- reflexivity \(\sigma\) is accessible from itself
- transitivity \(\tau\) is accessible from \(\sigma\) if \(\sigma\) is a proper segment of \(\tau\)
- universal any prefix is accessible from any prefix

We can express this form of accessibility using unification.
general \( \sigma \) is accessible from \( \tau \) if \( v(\sigma, \tau; \langle c \rangle) \)
symmetry \( \sigma \) is accessible from \( \tau \) if \( v(\sigma; \langle c \rangle, \tau) \)
reflexivity \( \sigma \) is accessible from \( \tau \) if \( \sigma = \tau \)
transitivity \( \sigma \) is accessible from \( \tau \) if \( v(\sigma, \tau; \langle C_+ \rangle) \)
universal \( \sigma \) is accessible from \( \tau \) if \( v(\sigma, \tau; \langle C_+ \rangle) \)

4.2 Massacci’s SST

In Single Step Tableaux (SST [6]), Massacci uses the same prefix notation as Fitting, but removes the need to determine anything other than immediate successors of a particular prefix. In which case, with regards to our \( \Sigma_{X,Y} \) prefixes, SST can be presented in the same way as Fitting’s prefixes except we only ever need to apply the general accessibility rule.

4.3 leanK

The Free Variable SST (leanK) system ([1]) enriches Fitting style prefixes with variables that capture all the worlds that are immediately accessible from another. Beckert and Goré present two forms of variable which they refer to as universal and free. A variable, \( x \), is universal if an instantiation of \( x \) on one branch need not affect the value of \( x \) on another. Universal variables may be instantiated multiple times. Free variables may only be instantiated once.

It is clear that universal variables relate to our \( C \) symbols and free variables to the variables in \( V \). As we are only interested in immediate successors, we need only consider the \( C \) symbol, \( c \) and variables of the type \( V \). In this case we can see that their notation is equivalent to \( \Sigma_{\{c\} \uparrow V} \).

Beckert and Goré do use unification in their work. If we translate their prefixes into \( \Sigma_{\{c\} \uparrow V} \) prefixes through a sensible mapping of universal variables to \( c \) and free variables to members of \( V \), our unification algorithm captures their requirements. The converse is not the case, however. leanK prefixes do not have the same expressivity as general prefixes. For example, the general prefix \( \langle 1, C_+ \rangle \) represents an infinite set of leanK prefixes \( \{1, \langle 1, u_1 \rangle, \langle 1, u_1, u_2 \rangle, \ldots \} \), where each \( u_n \) is a universal variable.

4.4 \( \Box \)KE

The prefix system described in this paper was inspired by the work of Pitt and Cunningham in their \( \Box \)KE system ([10]). Although the syntax has been changed somewhat and the underlying semantics have been expanded, the prefixes used in \( \Box \)KE are essentially the same as general \( \Sigma_{\{V \cup C_+ \}} \) prefixes.

4.5 KEM

It is KEM ([3]) that we have most difficulty relating our work to. KEM’s philosophy is that whilst the process of encoding modal formulas into prefixed ones is straightforward, it is at the point of unification that we need to deal with the semantics of the logic in question. This is an opposing point of view to how we wish to deal with unification\(^1\). That isn’t to say that there isn’t a correspondence between the two systems, as one

\(^1\) KEM has more in common with Otten and Kreitz’s T-string unification [9]
could take a KEM unification problem and translate the prefixes into general prefixes
taking into account the current modal semantics. However, this would be a cumbersome
process and of questionable benefit.

5 Properties of $\Sigma_{XY}$ unification

5.1 Termination

In [9], Otten and Kreitz, describe a class of strings that satisfy certain restrictions.
They call these strings, T-strings\(^2\). Two strings \(s\) and \(t\) have the T-string property iff

1. no character is repeated either in \(s\) or in \(t\), and
2. equal characters occur only at the beginning of \(s\) and \(t\).

Although, up until now we have placed no such restriction on a prefixes, we find that,
in practice, prefixes generated by our modal to prefix translation procedure ([12]) have
a similar property\(^3\). Because of the use of our \(C\) symbols, we can actually have the
same symbol (character) being repeated. However, we can simply revise the T-string
property to reflect prefixes.

**Definition 15.** Let \(\sigma\) and \(\tau\) be two prefixes. \(\sigma\) and \(\tau\) have the T-string property iff,

\[
\begin{align*}
1. & \forall_{1 \leq i, j \leq |\sigma|} : i \neq j \rightarrow (\sigma(i) \neq \sigma(j) \text{ or } \sigma(i) \in \mathcal{C}) \text{ and } \\
2. & \exists_{1 \leq k \leq |\sigma|} : (\forall_{1 \leq i \leq k} : \sigma(i) = \tau(i) \text{ and } \\
& \forall_{j \leq i \leq |\sigma| \& j \leq s \leq |\tau|} : (\sigma(i) \neq \tau(k) \text{ or } \sigma(i) \in \mathcal{C}))
\end{align*}
\]

**Theorem 1.** For any two T-prefixes \(\sigma\) and \(\tau\), \(\Sigma_{XY}\) unification will terminate on \(\sigma\)
and \(\tau\).

**Proof.** For any two prefixes, there is only one rule that can be applied at a particular
level.

In the case that neither \(\sigma\) or \(\tau\) contain variables then we can see that after each
application of our rules for \(v\) our subgoal(s) are simpler than our previous goal. Eventu-
ally, either of the first two rules will apply and we terminate.

In the case that we do have to deal with variables, we need to take into account the
properties of a T-prefixes. As we know that no variable will appear more than once in
a prefix and that if the same variable appears in both prefixes then it will only appear
as part of a common initial prefix, no substitutions made during unification will affect
the structure of a subgoal. Therefore, once again we can show that our subgoal(s) will
always be simpler than our previous goal.

5.2 Soundness

We define soundness for our most important classes of prefixes. Although the definition
of soundness for \(\Sigma_s\) is applicable to all sub-classes, the definitions of soundness for \(\Sigma_{VN}\) and \(\Sigma_{DC}\) are clearer and aid understanding.

\(^2\) Olholt ([12]) uses a similar idea which he calls Prefix-Stability

\(^3\) This translation procedure was inspired by the prefixed tableaux expansions in [10]
\( \Sigma_{N \nu} \) prefixes

The unification algorithm described in 3.2 does not give us the unifiers of two prefixes directly. Instead, for each unification that is produced there is an implied unifier that was required to make that unification. When necessary we will write \((\pi, \theta) \in v(\sigma, \tau)\) if \(\pi \in v(\sigma, \tau)\) and \(\theta\) was implicitly constructed during \(\pi\)’s discovery. We use the term implied to refer to any unifier that is implicit in a particular unification process.

Our unification algorithm is sound with respect to \( \Sigma_{N \nu} \) prefixes if for all \(\sigma\) and \(\tau\), if \(v(\sigma, \tau)\) implies \(\theta\) then \(\sigma\theta = \tau\theta\)

\( \Sigma_{NC} \) prefixes

Our unification algorithm is sound with respect to \( \Sigma_{NC} \) prefixes if for all \(\sigma\) and \(\tau\), if for all \(\pi \in v(\sigma, \tau)\), \(G(\pi) \subseteq G(\sigma) \cap G(\tau)\)

If we extend \(G\) to apply on sets of prefixes such that

\[ G(\Pi) = \bigcup_{\pi \in \Pi} G(\pi) \]

we can restate the above as:

Our unification algorithm is sound with respect to \( \Sigma_{NC} \) prefixes if for all \(\sigma\) and \(\tau\), \(G(v(\sigma, \tau)) \subseteq G(\sigma) \cap G(\tau)\)

\( \Sigma \) prefixes

\(v\) is sound with respect to \( \Sigma \) prefixes if for all \(\sigma\) and \(\tau\), if \((\pi, \theta) \in v(\sigma, \tau)\) then there exists a prefix mapping, \(m\) s.t. \(G(\pi_m) \subseteq G(\sigma \theta_m) \cap G(\tau \theta_m) \subseteq G(\sigma_m) \cap G(\tau_m)\)

5.3 Completeness

\( \Sigma_{N \nu} \) prefixes

\(v\) is complete with respect to \( \Sigma_{N \nu} \) if for all \(\sigma, \tau\) and \(\theta\), if \(\sigma \theta = \tau \theta\) then there exists a substitution, \(\theta'\) s.t. \(v(\sigma, \tau)\) implies \(\theta'\) and \(\theta\) is an instance of \(\theta'\).

\( \Sigma_{NC} \) prefixes

\(v\) is complete with respect to \( \Sigma_{NC} \) if for all \(\sigma\) and \(\tau\), \(G(\sigma) \cap G(\tau) \subseteq G(v(\sigma, \tau))\)

\( \Sigma \) prefixes

\(v\) is complete with respect to \( \Sigma \) prefixes if for all \(\sigma\) and \(\tau\), if \((\pi, \theta) \in v(\sigma, \tau)\) then for all prefix mappings, \(m\), \(G(\sigma_m) \cap G(\tau_m) \subseteq G(\sigma \theta_m) \cap G(\tau \theta_m) \subseteq G(\pi_m)\)

Theorem 2. \(v\) is sound and complete w.r.t. \( \Sigma_{NC} \) prefixes, i.e.

\[ G(v(\sigma, \tau)) = G(\sigma) \cap G(\tau) \]

Proof. As we have already noted, each step of our unification introduces simpler subgoals. Our proof is a straightforward induction over the length of a prefix pair s.t. \((\sigma, \tau) < (\pi, \psi)\) if \(|\sigma| < |\pi|\) and \(|\tau| \leq |\psi|\) or \(|\sigma| \leq |\pi|\) and \(|\tau| < |\psi|\). Our rules satisfy the property that if our subgoals satisfy the induction hypothesis \(G(v(\sigma, \tau)) = G(\sigma) \cap G(\tau)\) then so does our goal.
6 Discussion

The problem of unifying two prefixes is essentially a word unification problem with a single equation and additional constraints. Although, there has been significant work in solving general word equation problems (e.g., [5,11,4]), it is generally accepted that the problem is too hard to be solved in practical applications. Instead, we address a problem that seems to be a useful but tractable subset. The prefix notation presented in this paper was initially proposed as one aspect of a formalisation of □KE. As work progressed it became clear that its ability to encode accessibility relations was strong enough to allow for a straightforward extension to clausal resolution that could be applied to prefixed formulas. This prefixed resolution system then became the basis of a ‘modal Prolog’ ([13]).

There are a number of directions that future work could take. For instance, we have yet to determine the complexity of the unification process when applied to different classes of prefix. Theoretically, the use of C symbols significantly reduces the number of possible unifications of two particular prefixes when compared to equivalent prefixes containing variables. However, whether this reduction is reflected in the worst case or just in the average case is unclear. As it stands the unification algorithm described in 3.2 does not reflect this potential reduction of solutions. It doesn’t always find the simplest set of unifications and can find the same unification multiple times. For example, the unification of the prefixes, σ = (C*, C*) and τ = (C*, C*), finds the set of unifications {⟨⟩, ⟨c⟩, ⟨cc⟩} and it finds the unification ⟨c⟩ 4 times. Obviously the set {⟨C*, C*⟩} would be a preferable solution.

To further confuse matters, there are properties of the C symbols that have been observed but which need more consideration. For example, the prefixes (C*, C*), (c, C*), and (C*, C*) all represent the same set of basic prefixes. It would appear that given three prefixes, σ, σ' and τ, if σ and σ' are equivalent and |σ| < |σ'| the process of unifying σ and τ is simpler than unifying σ' and τ.

Let us briefly demonstrate the impact simplifications could make. Consider a prefix, σ, which is simply a chain of n C symbols. If we look at our unification algorithm it is clear that any attempt to unify σ with another prefix, τ, is going to have an exponential search space with respect to n. However, we can demonstrate that σ is equivalent to the prefix σ'\langle C*\rangle, where σ' is a chain of n − 1 c symbols. As we can see, the search space implied by c is linear. If we ignore the complexity implied by the structure of τ we have essentially reduced the search space from O(c^n) to O(c) = O(1)

It is clear that a better understanding of these issues would result in a more efficient unification algorithm.

In addition to issues of efficiency we may also wish to increase our expressivity. As it stands we do not have the ability to capture the essence of a multi-modal logic. The semantics of our unification algorithm assumes that we only have one accessibility relation. In order to deal with multiple modalities, we would need each element of a prefix to be paired with the relation it encodes.

It is our hope that work in such areas will result in an efficient and useful logic programming language for (multi)modal logics.

References

Author’s Index

S. Anantharaman, 4
J. Cheney, 15
H. Comon-Lundh, 20
J. Cunningham, 81
A. Foret, 24
G. Hains, 4
D. Kapur, 38
M. Kohlhase, 50
T. Kutsia, 56
M. Marin, 64
D. Miller, 1
P. Narendran, 38
J. Niehren, 78
W. Plandowski, 3
T. Priesnitz, 78
R. Treinen, 20
C. Urban, 15
L. Wang, 38
A. Williams, 81