Ingo Dahn
Laurent Vigneron (Eds.)

First-order Theorem Proving

4th International Workshop, FTP 2003
Valencia, Spain, June 12-14, 2003
Proceedings

Volume Editors

Ingo Dahn
Universität Koblenz-Landau, Germany
Email: dahn@uni-koblenz.de

Laurent Vigneron
LORIA - Université Nancy 2, France
Email: vigneron@loria.fr

Proceedings of the 4th International Workshop on First-order Theorem Proving, FTP'03.
Valencia, Spain, June 12-14, 2003

Depósito Legal:

Impreso en España.

Technical Report DSIC-II/10/03,
http://www.dsic.upv.es
Departamento de Sistemas Informáticos y Computación,
Universidad Politécnica de Valencia, 2003.

Preface

FTP'2003 is the fourth in a series of workshops intended to focus effort on First-Order Theorem Proving as a core theme of Automated Deduction, and to provide a forum for presentation of recent work and discussion of research in progress. The previous workshops of this series were held at Schloss Hagenberg, Austria (1997), Vienna, Austria (1998), St Andrews, Scotland (2000). In 2001, FTP was part of the IJCAR Conference, held in Siena, Italy.

FTP'2003 is one of the three main events of the Federated Conference on Rewriting, Deduction and Programming (RDP'03), together with RTA (the 14th International Conference on Rewriting Techniques and Applications), and TLCA (the 6th International Conference on Typed Lambda Calculi and Applications). FTP'2003 has hold on June 12-14, 2003.

The technical program of FTP'2003 consists of three invited talks, twelve regular papers, two system descriptions and two position papers. The topics of these papers match very well those of the workshop which cover theorem proving in first-order classical, many-valued, modal and description logics, including non-exclusively: resolution, equational reasoning, term-rewriting, model construction, constraint reasoning, unification, description logics, propositional logic, specialized decision procedures; strategies and complexity of theorem proving procedures; implementation techniques and applications of first-order theorem provers to verification, artificial intelligence, mathematics and education.

We sincerely thank everyone who contributed to make this workshop possible. First of all, we would like to thank all the authors who contributed the papers to FTP'2003. We received 19 papers and accepted for presentation 16 of them. We thank the members of the Program Committee and the additional reviewers for their excellent job. We also thank the Steering Committee, and Maria Paola Bonacina in particular, for their advice throughout all phases of the workshop. Finally, we owe a lot to Salvador Lucas and his collaborators of the University of Valencia, who did a very good work at preparing the local arrangements, and publishing these proceedings as Technical Report DSCI-II/10/03 of the Universidad Politécnica of Valencia.

The papers included in this report are preliminary versions. For most of them, the final version is published in the ENTCS series (Electronic Notes in Theoretical Computer Science, `http://www.math.tulane.edu/~entcs/`), volume 86 no.1.

Valencia, Spain
June 2003

Ingo Dahn and Laurent Vigneron

Program Committee Chairs

Ingo Dahn — University of Koblenz-Landau, Germany
Laurent Vigneron — LORIA - Université Nancy 2, France

Program Committee

Maria Paola Bonacina — Univ. of Verona, Italy
Ricardo Caferra — LEIBNIZ-IMAG, Grenoble, France
Bernhard Gramlich — TU Wien, Vienna, Austria
Paliath Narendran — SUNY at Albany, USA
David Plaisted — UNC at Chapel Hill, USA
Christophe Ringeissen — LORIA - INRIA Lorraine, Nancy, France
Albert Rubio — UPC, Barcelona, Spain
John Slaney — ANU, Canberra, Australia
Tomàs Uribe — SRI Int., Menlo Park, USA
Luca Viganò — ETHZ, Zürich, Switzerland
Christoph Weidenbach — Opel - MPI Saarbrücken, Germany
Hantao Zhang — Univ. of Iowa, Iowa City, USA

Additional Referees

Alessandro Armando	Florent Jacquemard	Silvio Ranise
Jürgen Avenhaus	Alexandre Miquel	Sophie Tison
David Déharbe	Raúl Monroy	Femke van Raamsdonk
Didier Galmiche	Robert Nieuwenhuis	Lida Wang

FTP Steering Committee

Alessandro Armando — Univ. di Genova, Italy
Peter Baumgartner — Univ. Koblenz, Germany
Maria Paola Bonacina, Chair — Univ. of Verona, Italy
Ricardo Caferra — LEIBNIZ-IMAG, Grenoble, France
Domenico Cantone — Univ. di Catania, Italy
David Crocker — Escher Technologies Ltd., UK
Ingo Dahn — Univ. of Koblenz-Landau, Germany
Bernhard Gramlich — Technische Univ. Wien, Austria
Reiner Hähnle — Chalmers Univ. of Technology, Göteborg, Sweden
Alexander Leitsch — Technische Univ. Wien, Austria
Paliath Narendran — Univ. at Albany - SUNY, Albany, NY, USA
Christoph Weidenbach — Opel - MPI Saarbrücken, Germany

Local Organization

María Alpuente	Javier Oliver	Germán Vidal
Salvador Lucas (chair)	María José Ramírez	

and all the ELP group at the Universidad Politécnica de Valencia.

Table of Contents

Deduction as an Engineering Science

Dieter Hutter [1]

German Research Center for Artificial Intelligence
Stuhlsatzenhausweg 3
66123 Saarbrücken, Germany

Abstract

Although in recent years a considerable progress has been made in the theory of automated theorem proving, the use of theorem provers in practice is still more or less restricted to a limited number of academic groups. A lot of effort has been spent in techniques to optimize the underlying logic engine by, for instance, developing efficient datastructures or controlling redundancy in large search spaces (see [6]). However, the development of techniques and methodologies to integrate such a logic engine into an overall proof assistant has gained less attraction. In this abstract we discuss the related research problems briefly and will explore possible ways to tackle these problems in the extended version of this paper.

Automated deduction has been an active area since the 1950s. However, the ultimate goal of fully mechanizing the proof capability of a mathematician is still distant. To overcome the combinatorial explosion in proof search, specialized theorem provers have been developed that are restricted to specific domains. Deduction engineering, which denotes the process of adjusting a functioning deductive system for improved performance, has become the source of many improvements in the area of automated deduction. As Loveland states in [4], *deduction engineering covers, for instance, the process of strategy formulations, having a range of outcomes, from publishable restrictions of considerable sophistication to simple delaying of the use of clauses that have many free variables. Most progress is found now by augmenting existing systems rather than implementing new basic procedures. Successful systems are composites and will become more so. Resources are needed to let the systems grow in capability, for instance, to allow for adding new heuristics or strategies.*

Originally supposed to provide **the** reasoning capabilities for Artificial Intelligence application, the focus for applying deduction has mainly shifted to the area of formal methods. To increase the reliabity of complex software systems, the use of formal software development and program verification in particular becomes

[1] Email: hutter@dfki.de

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

more and more popular. While nowadays the use of formal methods is state of the art in industrial hardware design, still verification techniques have not penetrated software manufacturing industry but their utilization is restricted to a rather limited number of academic groups. The issues that emerge when aiming at a more widespread application of deduction in software development are manifold.

Proof Engineering as an Evolutionary Process.

Developing proofs in formal methods applications is a lengthy and error-prone task. Since even the verification of small-sized industrial developments requires several person months, specification errors revealed in late verification phases pose an incalculable risk for the overall project costs. In all applications so far, development steps turned out to be flawed and errors had to be corrected. The search for formally correct software and the corresponding proofs is more like a *formal reflection* of partial developments rather than just a way to assure and prove more or less evident facts. There is a need for tool support coping with an *evolutionary development of proofs*. The issues are twofold: first, to adequately react on changes caused by detecting specification errors and second, to evolve proofs for sophisticated tasks by gradually proving more and more detailed versions of it in order to reuse proofs of previous versions as skeletons for following versions.

The Role of the User.

Today, the tackling of proof obligations in formal methods require user interactions and will do it in the foreseeable future. While in automated theorem proving user interaction is restricted to the fine-tuning of various parameters of a system (with mostly unpredictable results), tactical theorem proving requires the adequate selection of series of tactic/tactical calls to find a proof. Failures of such a system to find a proof, typically give rise to a laborious investigation of possible reasons and require far-reaching knowledge of the user how the underlying proof calculus and proof procedure will explore the search space. There is a need for techniques to allow for a more abstract communication between systems and users about the achievements, the goals, and the intentions in the ongoing proof work. Prerequisites are the development of appropriate abstractions to provide a language for communication and the design of appropriate user interfaces that eases the communication.

Scalability.

Proof obligations increase dramatically in size when tackling more complex (industrial scaled) verification problem. However, in many system a once successful proof search is endangered when enlarging the set of axioms with additional (redundant) facts. One of the main obstacles of using techniques for automated theorem proving in practice is the lack of their scalability. Analogously to software engineering there is a need for modularity and structured design in proof engineering. Supporting the proof design in early phases is an untackled problem. Using

automated theorem provers successfully, usually requires to know already the key steps of the desired proof in advance. Depending on the way of specifying the initial problem, a theorem prover will be able to prove a problem or not. However, the knowledge about the *successful* way is usually restricted to the developers or very experienced users of the system.

Integrating Application Knowledge.

One way to solve the problem of scalability is to transfer and use the structuring mechanisms of the application (giving rise to the proof obligation under consideration) to structure also the proof search. Generally speaking, semantic knowledge about the application can often be used to guide the proof process and reduce the search space dramatically. Rippling [2] as a specific technique to guide inductive proofs is a paradigm of this principle. However, incorporating application knowledge into a theorem proving system requires that it is also able to maintain this knowledge and deduce the consequences for new facts that have been derived with the help of the logical calculus.

Combining Different Proof Techniques.

Successful systems are composites like for instance the PVS-system [5]. They combine special purpose procedures (like model checking, arithmetic procedures, computer algebra systems, etc.) to make use of the special knowledge that is incorporated into these procedures. All of them have special restrictions to the kind of problems they can tackle and the main problem of making effective use of them is to reformulate problems occurring during the proof search such that they fit the restrictions of the designated special purpose procedure (e.g. [1]). While there is a lot of progress concerning techniques to connect different systems wrt. technical means (see [3]), lemma speculation and generalization techniques are still necessary to formulate the needs of a system in an appropriate language of the tool to be used.

References

[1] Boyer, R.S. and Moore, J S. *Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic*, Machine Intelligence **(**11),pp. 83-124, Clarendon Press, Oxford, (1988)

[2] Bundy, A., Basin, D., Hutter, D., Ireland, A. *Rippling: Meta-level Guidance for Mathematical Reasoning*, Cambridge University Press, (2003)

[3] Kerber, M., Kohlhase, M. (eds) *Symbolic Computation and Automated Reasoning – The Calculemus-2000 Symposium*, A K Peters Publishers, USA, (2000)

[4] Loveland, D., *Automated Deduction: Looking ahead*, AI-Magazine. **20**(1) (1999).

[5] Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M. K. *PVS: Combining specification, proof checking, and model checking*, Proceedings of the 18th International Conference on Computer Aided Verification CAV, Springer, LNCS 1102, New Brunswick, NJ, USA, (1996)

[6] Voronkov, A., *Algorithms, Datastructures, and Other Issues in Efficient Automated Deduction*, Proccedings of the 1st International Joint Conference on Automated Reasoning, IJCAR 2001, Springer, LNAI 2083, (2001)

Citius altius fortius: Lessons learned from the Theorem Prover WALDMEISTER

Thomas Hillenbrand [1]

Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany

Abstract

In the last years, the development of automated theorem provers has been advancing in a so to speak Olympic spirit, following the motto "faster, higher, stronger"; and the WALDMEISTER system has been a part of that endeavour. We will survey the concepts underlying this prover, which implements Knuth-Bendix completion in its unfailing variant. The system architecture is based on a strict separation of active and passive facts, and is realized via specifically tailored representations for each of the central data structures: indexing for the active facts, set-based compression for the passive facts, successor sets for the hypotheses. In order to cope with large search spaces, specialized redundancy criteria are employed, and the empirically gained control knowledge is integrated to ease the use of the system. We conclude with a discussion of strengths and weaknesses, and a view of future prospects.

[1] Email: hillen@mpi-sb.mpg.de

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

SAT and Beyond SAT

Enrico Giunchiglia [1]

DIST – University of Genova
Viale Causa 13
16145 Genova, Italy

Abstract

In the last few years we have seen a tremendous boost in the performance and capacity of SAT solvers, both on randomly generated instances and on instances coming from real-world problems. This boost has fostered the solution of real-world problems via compilation to SAT and/or via the construction of SAT-based engines.

In this talk, I will first survey the state-of-the-art on SAT solvers technology (possibly reporting also on the results of the last SAT competition), and then I will discuss how it is possible to define SAT-based decision procedures for more expressive formalisms.

[1] Email: giunchiglia@unige.it

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Quantifier Elimination and Provers Integration

Silvio Ghilardi [1]

Dipartimento di Scienze dell'Informazione
Università degli Studi
Milano, Italy

Abstract

We exploit quantifier elimination in the global design of combined decision and semi-decision procedures for theories over non-disjoint signatures, thus providing in particular extensions of Nelson-Oppen results.

Keywords: Combination, Nelson-Oppen Combination Schema, Fusion, Superposition Calculus, Quantifier Elimination, Model Completions.

1 Introduction and Background

Quantifier elimination has been considered, since the early times of modern symbolic logic, a powerful technique for decision procedures. Even in actual approaches to combination problems (see e.g. [9]), specific quantifier elimination algorithms are often invoked as specialized reasoners to be integrated within a flexible general setting dealing with multiple theories. This happens, in particular, whenever numerical constraints problems need to be adequately addressed: examples of such specialized reasoners are the Fourier-Motzkin quantifier elimination procedure for linear rational arithmetic or Cooper's quantifier elimination procedure for integer Presburger arithmetic.

In contrast to this *local call* for quantifier elimination algorithms, we shall address in this paper quantifier elimination as a *global design* opportunity for integrated provers: we shall show in particular how it can be used in order to *extend Nelson-Oppen combination procedure* [11], [13], [16] *to non-disjoint signatures*. Detailed proofs of the results presented here, as well as additional information, can be found in [6].

A *signature* Σ is a set of functions and predicate symbols (each of them endowed with the corresponding arity). We assume the binary equality predicate

[1] Email: ghilardi@dsi.unimi.it

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

symbol $=$ to be always present in Σ. The signature obtained from Σ by the addition of a set of new constants (= 0-ary function symbols) X is denoted by $\Sigma \cup X$ or by Σ^X. We have the usual notions of Σ-*term*, (full first order) *-formula*, *-atom*, *-literal*, *-clause*, *-positive clause*, etc.: e.g. atoms are just atomic formulas, literals are atoms and their negations, clauses are disjunctions of literals, positive clauses are disjunctions of atoms. Letters $\phi, \psi, \ldots$ are used for formulas, whereas letters $A, B, \ldots$ are used for literals and letters $C, D, \ldots$ are used for clauses. Terms, literals and clauses are called *ground* whenever variables do not appear in them. Formulas without free variables are called *sentences*. A Σ-*theory* T is a set of sentences (called the axioms of T) in the signature Σ; however when we write $T \subseteq T'$ for theories, we may mean not just set-theoretic inclusion but the fact that all the axioms for T are logical consequences of the axioms for T'.

From the semantic side, we have the standard notion of a Σ-*structure* $\mathcal{A}$: this is nothing but a support set endowed with an arity-matching interpretation of the predicate and function symbols from Σ. We shall notationally confuse, for the sake of simplicity, a structure with its support set. Truth of a Σ-formula in $\mathcal{A}$ is defined in any one of the standard ways (so that truth of a formula is equivalent to truth of its *universal* closure). A Σ-structure $\mathcal{A}$ is a *model* of a Σ-theory T (in symbols $\mathcal{A} \models T$) iff all axioms of T are true in $\mathcal{A}$; for models of a Σ-theory T we shall preferably use the letters $\mathcal{M}, \mathcal{N}, \ldots$ to distinguish them from arbitrary Σ-structures. If ϕ is a formula, $T \models \phi$ ('ϕ *is a logical consequence of* T') means that ϕ is true in any model of T. A Σ-theory T is *complete* iff for every Σ-sentence ϕ, either ϕ or $\neg\phi$ is a logical consequence of T; T is *consistent* iff it has a model (i.e. iff $T \not\models \bot$).

An *embedding* between two Σ-structures $\mathcal{A}$ and $\mathcal{B}$ is any map $f : \mathcal{A} \longrightarrow \mathcal{B}$ among the corresponding support sets satisfying the condition

$$(*) \qquad \mathcal{A} \models A \quad \text{iff} \quad \mathcal{B} \models A$$

for all $\Sigma^{\mathcal{A}}$ atoms A (here $\mathcal{A}$ is regarded as a $\Sigma^{\mathcal{A}}$-structure by interpreting each $a \in \mathcal{A}$ into itself and $\mathcal{B}$ is regarded as a $\Sigma^{\mathcal{A}}$-structure by interpreting each $a \in \mathcal{A}$ into $f(a)$). Notice that, as we have identity in the language, an embedding is an injective function (it also must preserve the interpretation of the function symbols and, in case it is just an inclusion, the interpretation of the predicate symbols in the smaller structure must be the restriction of the corresponding interpretation in the bigger structure). In case $(*)$ holds for all first order formulas, the embedding is said to be *elementary*.

The main problems we deal with are *word problems*, more precisely, given a Σ-theory T:
- the *word problem* for T is that of deciding whether $T \models A$ holds for a Σ-atom A;
- the *conditional word problem* for T is that of deciding whether $T \models C$ holds for a Horn Σ-clause C;
- the *clausal word problem* for T is that of deciding whether $T \models C$ holds for a Σ-clause C;
- the *elementary word problem* for T is that of deciding whether $T \models \phi$ holds for a

first order Σ-formula ϕ.

A formula is *quantifier-free* iff it does not contain quantifiers. A Σ-theory T is said to *eliminate quantifiers* iff for every formula $\phi(\underline{x})$ [2] there is a quantifier-free formula $\phi'(\underline{x})$ such that $T \models \phi(\underline{x}) \leftrightarrow \phi'(\underline{x})$. There are many well-known theories [4] eliminating quantifiers, we give here some examples which can be of interest for software verification.

Example 1.1 Linear integer arithmetic (i.e. the theory of the structure of integer numbers in the signature $+, 0, 1, \leq, \equiv_n$) eliminates quantifiers; so does rational linear arithmetic (i.e. the theory of rational numbers in the signature $+, 0, \leq$). Another well-known classical example from Tarski is real arithmetic (i.e. the theory of real numbers in the signature $+, 0, \cdot, 1, \leq$).

Example 1.2 The theory of acyclic binary lists L [13], [14] eliminates quantifiers (see [6]).

The main ingredient of this paper is the well-known notion of a *model completion* of a theory. There are good chapters on that in all texbooks from Model Theory. We shall recall here just the essential definitions *for the only case of universal theories* [3] which is the relevant one for the purposes of this paper (readers may consult e.g. [4], [10], [18] for further information).

Let T be a universal Σ-theory and let $T^* \supseteq T$ a further Σ-theory; we say that T^* is a model completion of T iff i) every model of T has an embedding into a model of T^* and ii) T^* eliminates quantifiers.

It can be shown that a model completion T^* of a theory T is unique, in case it exists, and moreover that T^* has a set of $\forall\exists$-axioms, see [4].

Example 1.3 The theory of an infinite set is the model completion of pure equality theory; the theory of dense total orders without endpoints is the model completion of the theory of total orders.

Example 1.4 There are many classical examples from algebra: the theory of algebraically closed fields is the model completion of the theory of intergral domains, the theory of divisible torsion free abelian groups is the model completion of the theory of torsion free abelian groups, etc.

Example 1.5 The theory of atomless Boolean algebras [4] is the model completion of the theory of Boolean algebras (for model completions arising in the algebra of logic, see the book [8]).

Example 1.6 An old result in [18] says, in particular, that universal Horn theories T in finite signatures always have a model completion, provided the following two

[2] By this notation, we mean that ϕ contains free variables only among the finite set $\underline{x}$.

[3] Recall that a universal theory T is a theory having as axioms only universal closures of quantifier-free formulas.

[4] We recall that an atom in a Boolean algebra is a minimal non-zero element; a Boolean algebra is atomless iff it has no atoms.

conditions are satisfied: a) finitely generated models of T are all finite; b) amalgamation property holds for models of T. This fact can be used in order to prove the existence of a model completion for theories axiomatizing many interesting discrete structures (like graphs, posets, etc.).

Example 1.7 It follows from the quantifier elimination result reported in [6] that the theory L of acyclic binary lists is the model completion of itself.

Example 1.8 If a theory T^* has elimination of quantifiers, then it is the model completion of the theory T axiomatized by the set of universal sentences which are logical consequences of T, see [4].

2 Compatibility

The key ingredient for our combination procedures is the following notion:

Definition 2.1 Let T be a theory in the signature Σ and let T_0 be a universal theory in a subsignature $\Sigma_0 \subseteq \Sigma$. We say that T is T_0-compatible iff

(i) $T_0 \subseteq T$;

(ii) T_0 has a model-completion T_0^*;

(iii) every model of T embeds into a model of $T \cup T_0^*$.

Condition (iii) can be equivalently given in a slightly different form, by saying that every quantifier-free Σ-formula which is false in a model of T is false also in a model of $T \cup T_0^*$.

Example 2.2 According to this remark, it is evident that T_0-compatibility *reduces to the standard notion of stable infiniteness* (used in the disjoint Nelson-Oppen combination procedure) in case T_0 is the pure theory of equality:[5] recall in fact that in this case T_0^* (i.e. the model completion of the pure equality theory) is the theory of an infinite set.

Example 2.3 Every theory including the theory L of acyclic binary lists is compatible with L, because L is universal and $L = L^*$.

Example 2.4 If T_0 has a model completion T_0^* and if $T \supseteq T_0^*$, then T is certainly T_0-compatible: this trivial case is often interesting (we may take e.g. T_0 to be the theory of linear orders and T to be real aritmetic or rational linear arithmetic).

Example 2.5 Let T_0 be a universal theory having a model completion T_0^*; if T is any extension of T_0 with free function symbols only, then T is T_0-compatible.

More examples will be supplied in section 4. An interesting feature of T_0-compatibility is that it is a *modular* property:

[5] By the 'pure theory of equality' we mean the empty theory in the signature containing only the equality predicate.

Proposition 2.6 *Let T_1 be a Σ_1-theory and let T_2 be a Σ_2-theory; suppose they are both compatible with respect to a Σ_0-theory T_0 (where $\Sigma_0 := \Sigma_1 \cap \Sigma_2$). Then $T_1 \cup T_2$ is T_0-compatible too.*

3 Combining compatible theories

Let us progressively fix our main data for the whole paper.

Assumption (I). *T_1 is a theory in the signature Σ_1 and T_2 is a theory in the signature Σ_2; Σ_0 is the signature $\Sigma_1 \cap \Sigma_2$.*

Our main aim is that of (semi)deciding the clausal word problem for $T_1 \cup T_2$, given that the corresponding clausal word problems for T_1 and T_2 are (semi)decidable. Equivalently, this amounts to (semi)decide the consistency of

$$T_1 \cup T_2 \cup \Gamma,$$

where Γ is a finite set of ground literals in the signature $\Sigma_1 \cup \Sigma_2$, expanded with a finite set of new Skolem constants.

Γ can be *purified*: as usual, we can abstract alien subterms and add equations involving further new free constants, in such a way that our problem is reduced to the problem of establishing the consistency of a set of sentences like

$$(T_1 \cup \Gamma_1) \cup (T_2 \cup \Gamma_2), \tag{1}$$

where Γ_1, Γ_2 are as explained in the following:

Assumption (II). *For finitely many new free constants $\underline{a}$, Γ_1 is a finite set of ground literals in the signature $\Sigma_1^{\underline{a}}$ and Γ_2 is a finite set of ground literals in the signature $\Sigma_2^{\underline{a}}$.*

For trivial reasons, the consistency of (1) cannot follow from the mere separate consistency of $T_1 \cup \Gamma_1$ and of $T_2 \cup \Gamma_2$. We need some *information exchange* between a reasoner dealing with $T_1 \cup \Gamma_1$ and a reasoner dealing with $T_2 \cup \Gamma_2$.

Craig's interpolation theorem for first order logic ensures that the inconsistency of (1) can be detected by the information exchange of a single $\Sigma_0^{\underline{a}}$-sentence ϕ such that $T_1 \cup \Gamma_1 \models \phi$ and $T_2 \cup \Gamma_2 \cup \{\phi\} \models \bot$. However, as pointed out in [15], this observation is not very useful, as ϕ might be any first-order formula, whereas we would like - at least - ϕ to be quantifier-free.

Unfortunately, information exchange of quantifier-free $\Sigma_0^{\underline{a}}$-formulas alone is not sufficient, even for syntactically simple T_1 and T_2, to establish the inconsistency of (1) (see section 5 below for a counterexample). We so need a further assumption in order to get limited information echange without affecting refutational completeness (this is the relevant assumption we make, the other two being mere notational conventions):

Assumption (III). *There is a universal Σ_0-theory T_0 such that both T_1 and T_2 are T_0-compatible.*

A finite list $C_1, \ldots, C_n$ of positive ground $\Sigma_0^{\underline{a}}$-clauses such that for every $k = 1, \ldots, n$, there is $i = 1, 2$ such that

$$T_i \cup \Gamma_i \cup \{C_1, \ldots, C_{k-1}\} \models C_k.$$

is called a *positive residue chain*. We can now formulate our combination results (see [6] for proofs):

Theorem 3.1 *In the above assumptions, $(T_1 \cup \Gamma_1) \cup (T_2 \cup \Gamma_2)$ is inconsistent iff there is a positive residue chain $C_1, \ldots, C_n$ such that C_n is the empty clause.*

Thus inconsistency can be detected by repeated exchanges of positive ground clauses only; if we allow information exchange consisting on ground quantifier free formulas, a single exchange step is sufficient:

Theorem 3.2 *In the above assumptions, $(T_1 \cup \Gamma_1) \cup (T_2 \cup \Gamma_2)$ is inconsistent iff there is a ground quantifier-free $\Sigma_0^{\underline{a}}$-sentence ϕ such that*

$$T_1 \cup \Gamma_1 \models \phi \qquad \textit{and} \qquad T_2 \cup \Gamma_2 \cup \{\phi\} \models \bot.$$

Following [15], we say that our T_i's are Σ_0-convex iff whenever it happens that $T_i \cup \Gamma_i \models A_1 \vee \cdots \vee A_n$ (for $n \geq 1$ and for ground $\Sigma_0^{\underline{a}}$-atoms $A_1, \ldots, A_n$), then there is $k = 1, \ldots, n$ such that $T_i \cup \Gamma_i \models A_k$.[6] For Σ_0-convex theories, Theorem 3.1 refines in the following way:

Corollary 3.3 *In addition to the above assumptions, suppose also that T_1, T_2 are both Σ_0-convex. Then $(T_1 \cup \Gamma_1) \cup (T_2 \cup \Gamma_2)$ is inconsistent iff there is a positive residue chain $C_1, \ldots, C_n$ in which $C_1, \ldots, C_{n-1}$ are all ground Σ_0-atoms and C_n is $\bot$.*

4 The locally finite case

We say that a Σ_0-universal theory T_0 is *locally finite* iff Σ_0 is finite and for every finite set $\underline{a}$ of new free constants, there are finitely many $\Sigma_0^{\underline{a}}$-ground terms $t_1, \ldots, t_{k_{\underline{a}}}$ such that for every further $\Sigma_0^{\underline{a}}$-ground term u, we have $T_0 \models u = t_i$ (for some $i = 1, \ldots, k_{\underline{a}}$).[7] As we are mainly dealing with computational aspects, we consider part of the definition the further request that such $t_1, \ldots, t_{k_{\underline{a}}}$ are effectively computable from $\underline{a}$. Examples of locally finite theories are the theory of graphs, of partial orders (more generally, any theory whose signature does not contain function symbols), of commutative idempotent monoids, of Boolean algebras, etc.

[6] Among Σ_0-convex theories we have the important class of universal Horn theories, see [15] again.

[7] Local finiteness is a much weaker requirement than the notion of 'finitary modulo a renaming' introduced in [2]. The reason is because the number $k_{\underline{a}}$ depends on the cardinality of $\underline{a}$; on the contrary a Σ_0-theory T is said to be finitary modulo a renaming iff there is a finite set of Σ_0-terms S such that for every Σ_0-term u there are $t \in S$ and a renaming σ such that $T \models u = t\sigma$. Consequently, for instance, locally finite theories (like Boolean algebras) in which the number $k_{\underline{a}}$ grows more than polynomially in the cardinality of $\underline{a}$ cannot be finitary modulo a renaming.

In a locally finite theory T_0, there are restricted *finite* classes which are representatives, up to T_0-equivalence, of the whole classes of Σ_0^a-ground literals, clauses, quantifier-free sentences, etc. (they are just the ground literals, clauses, quantifier-free sentences, etc. containing only the above mentioned terms $t_1, \ldots, t_{k_a}$). As it is evident that we can limit information exchange to ground positive clauses and quantifier-free sentences in that restricted class, both Theorems 3.1, 3.2 yield *combined decision procedures for the clausal word problem in* $T_1 \cup T_2$ in case the above assumptions (I) and (III) are satisfied and in case T_0 is locally finite. In particular, Theorem 3.1 suggest the following extension of the Nelson-Oppen procedure [13]:

Algorithm 1

Step 1: Negate, skolemize and purify the universal closure of the **input clause** C *thus producing a set* Γ_1 *of ground* Σ_1^a*-literals and a set* Γ_2 *of ground* Σ_2^a*-literals (then* $\Gamma_1 \cup \Gamma_2$ *is* $T_1 \cup T_2$*-equisatisfiable with* $\neg\forall \underline{x} C$*). During the next Steps loop, positive ground* Σ_0^a*-clauses are added to* Γ_1, Γ_2*.*

Step 2: Using the decision procedures for T_1, T_2*, check whether* $T_1 \cup \Gamma_1$ *and* $T_2 \cup \Gamma_2$ *are consistent or not (if one of them is not,* **return** *'*$T_1 \cup T_2 \models C$*').*

Step 3: If $T_i \cup \Gamma_i$ *entails some positive ground* Σ_0^a*-clause (atom in the* Σ_0*-convex case) not entailed by* $T_j \cup \Gamma_j$ *(*$j \neq i$*) add this positive ground clause (atom) to* Γ_j *and go back to Step 2.*

Step 4: If this step is reached, **return** *'*$T_1 \cup T_2 \not\models C$*'.*

Example 4.1 Let T_1 be rational linear arithmetic and let T_2 be the theory of total orders endowed with a strict monotonic function f. We take as T_0 the theory of total orders (recall that its model completion T_0^* is the theory of dense total orders without endpoints). T_1 is known to be decidable and the clausal word problem for T_2 is decidable too. As $T_1 \supseteq T_0^*$, T_1 is certainly T_0-compatible. T_2 is also T_0-compatible (to embed a model $\mathcal{M}$ of T_2 into a model $\mathcal{M}'$ of $T_0^* \cup T_2$, take as $\mathcal{M}'$ the lexicographic product of $\mathcal{M}$ with e.g. the poset of rational numbers). Thus our combination results apply and we obtain the decidability of the clausal word problem for rational linear arithmetic endowed with a strict monotonic function.

Example 4.2 A *modal algebra* is a Boolean algebra $\mathcal{B} = \langle B, \cap, 1, \cup, 0, (-)' \rangle$ endowed with an operator $\Box$ preserving binary meets and the top element. Let now Σ_1 be the signature of Boolean algebras augmented with a unary function symbol $\Box_1$ and let Σ_2 be the signature of Boolean algebras augmented with a unary function symbol $\Box_2$. T_1 is the equational theory of a variety V_1 of modal algebras and T_2 is the equational theory of another variety V_2 of modal algebras. For $i = 1, 2$, T_i is a universal Horn theory, hence it is Σ_i-convex: this means in particular that the solvability of the conditional word problem for T_i implies the solvability of the clausal word problem for T_i. As every model of T_i embeds into a model whose Boolean reduct is atomless, we can conclude that the solvability of the conditional word problem for T_1 and T_2 implies the solvability of the conditional word prob-

lem for $T_1 \cup T_2$. Also, in case the modal operators $\Box_1, \Box_2$ are both *transitive*, [8] the solvability of the word problem for T_1 and T_2 implies the solvability of the word problem for $T_1 \cup T_2$.

We underline that the last observation, once read in terms of logics, *means exactly fusion decidability for normal extensions of* $K4$. Although this does not entirely cover Wolter's fusion decidability results [19], it puts some substantial part of them into the appropriate general combination context. For new results (based on a refinement of the combination schema explained in this section) concerning fusion of modal logics sharing a universal modality and nominals, see [7].

5 Pure deductions

In this section we give some further suggestions about a possible use of the ideas explained in section 3 within saturation-based theorem proving. We show that whenever T_0-compatibility holds it is possible to cut in a deduction the inferences which are not pure, still retatining refutational completeness. An inference among $(\Sigma_1 \cup \Sigma_2)^{\underline{a}}$-clauses

$$\frac{C_1, \ldots, C_n}{C}$$

is *pure* iff there is $i = 1, 2$ such that all the clauses $C_1, \ldots, C_n, C$ are $\Sigma_i^{\underline{a}}$-clauses. Similarly, a deduction is pure iff all inferences in it are pure. Usually pure deductions are not able to detect inconsistency of (the skolemization of) sets of sentences like $T_1 \cup \Gamma_1 \cup T_2 \cup \Gamma_2$, however we shall see that this may happen when the T_0-compatibility conditions are satisfied.

In order to realize this program, we first need to skolemize the theories T_1, T_2, thus passing to theories T_1^{sk}, T_2^{sk} in extended signatures $\Sigma_1^{sk}, \Sigma_2^{sk}$; Skolem functions will *not* be considered shared symbols, hence we still have that $\Sigma_0 = \Sigma_1^{sk} \cap \Sigma_2^{sk}$. The first problem we meet is the following: if T_i is T_0-compatible, is T_i^{sk} still T_0-compatible? We do not have a general answer for that, however there is a relevant case in which the answer is affirmative:

Proposition 5.1 *Let T be a Σ-theory which is compatible with respect to a Σ_0-theory T_0 (here Σ_0 is a subsignature of Σ). If the axioms of T are all $\forall\exists$-sentences, then T^{sk} is T_0-compatible too.*

The previous Proposition motivates the following extra assumption (in addition to those from section 3):

Assumption (IV). *T_1, T_2 are axiomatized by $\forall\exists$-sentences; T_1^{sk}, T_2^{sk} are their skolemizations.*

[8] The modal operator $\Box_i$ is said to be transitive iff $T_i \models \Box_i x \cap \Box_i \Box_i x = \Box_i x$. For transitive modal operators it is easily seen that the conditional word problem reduces to the word problem.

We take into consideration here the *Superposition Calculus* $\mathcal{I}$ (see [3], [12]). We fix a *lexicographic path ordering* [9] induced by a total precedence on the symbols of $\Sigma_1^{sk} \cup \Sigma_2^{sk} \cup \{\underline{a}\}$; assuming for simplicity that our signatures are finite, this induces a reduction ordering $>$ which is total on ground terms. We give to symbols in Σ_0^a *lower precedence* than to symbols in $\Sigma_1^{sk} \setminus \Sigma_0$ and in $\Sigma_2^{sk} \setminus \Sigma_0$. This is essential: as a consequence, ground Σ_0^a-clauses will be smaller in the twofold multiset extension of $>$ than all ground clauses containing a proper Σ_1 or Σ_2-symbol.

Theorem 5.2 *In the above assumptions (I)-(IV), the set of sentences* $T_1 \cup \Gamma_1 \cup T_2 \cup \Gamma_2$ *is inconsistent iff there is a pure* $\mathcal{I}$*-derivation of the empty clause from* $T_1^{sk} \cup \Gamma_1 \cup T_2^{sk} \cup \Gamma_2$.

A possible direction for future reseach should try to take advantage from Theorem 5.2 in decision procedures based on Superposition Calculus: in fact for interesting (intrinsecally non locally finite) theories, the Superposition Calculus terminates whenever it has to test satisfiability of finite sets of ground literals [1].

Before concluding this section, we shall provide an example in which the assumptions of Theorem 5.2 are satisfied and an example in which such assumptions fail.

Example 5.3 Let T_1, T_2 be both the theory of Boolean algebras; we assume that symbols of the bounded distributivity lattice language (namely $\cap, \cup, 0, 1$) are shared but that the two complements n_1, n_2 are not. We want to prove that $T_1 \cup T_2 \models \forall x(n_1(x) = n_2(x))$. If we take T_0 to be the theory of bounded distributive lattices (i.e. of distributive lattices with 0 and 1), we see that T_1, T_2 are T_0-compatible. Skolemization and purification give for instance the two sets of literals $\Gamma_1 = \{a = n_1(c), a \neq b\}$ and $\Gamma_2 = \{b = n_2(c), a \neq b\}$. A pure $\mathcal{I}$-refutation exists: the prover SPASS produces a pure $\mathcal{I}$-refutation consisting on 28 steps. However, the system is not programmed in order to avoid impure inferences, so that, during saturation, it impurely derives also (useless) 'mixed' clauses containing both n_1 and n_2. One of them, namely the atom $b \cap n_1(n_2(a)) = b$, is also selected as a given clause.

Example 5.4 Let T_1 be the theory of Boolean algebras and let T_2 be the theory of pseudocomplemented distributive lattices; these are bounded distributive lattices endowed with a unary operator $(-)^*$ satisfying the condition $\forall x \forall y\ (x \cap y = 0 \leftrightarrow y \leq x^*)$. This condition expresses the properties of intuitionistic negation, hence in the union theory $T_1 \cup T_2$, the operator $(-)^*$ collapses into the classical complement. This means that $T_1 \cup \Gamma_1 \cup T_2 \cup \Gamma_2$ is inconsistent, where Γ_1 is empty and Γ_2 is $\{(a^*)^* \neq a\}$. A SPASS refutation takes 43 lines and it is highly impure. In fact a pure refutation cannot exist: the Σ_0- (and even the Σ_0^a)-ground clauses deducible from either $T_1 \cup \Gamma_1$ or $T_2 \cup \Gamma_2$ are insufficient to detect inconsistency, because they are all subsumed by the three negative literals $0 \neq 1, a \neq 1, a \neq 0$. Notice that T_2 is not T_0-compatible.

[9] It is not clear whether the results explained in this section hold in case a Knuth-Bendix ordering is adopted.

6 Conclusions and related work

In this paper we have extended Nelson-Oppen combination procedure to the case of theories T_1, T_2 over non-disjoint signatures, in presence of compatibility conditions over a common universal subtheory T_0. The extension we proposed applies to examples of real interest giving, as shown in section 4, combined decidability in case T_0 is locally finite. Whenever T_0 is not locally finite, our method can be used in order to limit residue exchange (see section 3) or in order to forbid impure inferences in saturation-based theorem proving, thus yielding restrictions on the search space during refutation derivations (see section 5).

It should be noticed that quantifier-elimination *plays only an indirect role* in the paper: in this sense, the existence of a model completion for a universal theory T_0 *guarantees a certain behaviour* in combination problems *by itself*, independently on how quantifier elimination in the model completion is established (this can be established also by semantic non constructive arguments, as largely exemplified in the model-theoretic literature). In principle, the quantifier elimination complexity/decidability *has nothing to do* with the complexity/decidability of our combination methods, simply because *quantifier elimination algorithms do not enter into them.* This is crucial, because most quantifier elimination algorithms are subject to heavy complexity lower bounds, which are often structural lower bounds for the decision of the elementary word problem in the corresponding theories [5].

One may wonder how severe is the crucial condition of T_0-compatibility used in the paper: let us discuss it for a while. T_0-compatibility involves two aspects, namely the existence of a model completion T_0^* for T_0 and the embeddability of models of T_i into models of $T_i \cup T_0^*$. As we have shown in the examples, the existence of a model completion seems to be frequent for theories commonly used in software verification. On one side, numeric constraint theories often enjoy this property, in the sense that they eliminate quantifiers (thus being model completions of the theories axiomatized by their respective universal consequences). On the other side, acyclic binary lists might probably be the paradigm of situations arising in theories axiomatizing natural datatypes. Finally, notice that quantifier elimination strictly depends on the choice of the language: every theory trivially has quantifier elimination in an extended language with infinitely many definitional axioms, hence the problem of obtaining quantifier elimination seems to be mostly a problem of choosing a sufficiently rich but still natural and manageable language.

The question concerning embeddability of models of T_i into models of $T_i \cup T_0^*$ looks more problematic, in the sense that it can fail in significant situations and, in addition, it does not look to be mechanizable. Further research is necessary on this point, however we underline that there is a relevant case in which the problem disappears. This is the case in which T_i is an extension of T_0^*: we have seen an example in section 4 where T_i is rational linear arithmetic and T_0 is the theory of linear orders. Another example is the theory of acyclic lists L (which coincides with L^*): any extension of the theory of acyclic lists with significant extra structures matches our requirements and the advantages of our method (limited residue

exchange, elimination of impure inferences, etc.) apply to all combinations of theories obtained in this way.

There have been many efforts in the literature trying to extend Nelson-Oppen combination method to theories sharing function and predicate symbols (different from equality). The starting point of any attempt to generalize Nelson-Oppen procedure to the non-disjoint case should preliminarly answer the following question: what is the specific feature of the stable infiniteness requirement that we want to generalize? In the present paper we answered the question by saying that infinite models are just *existentially closed* models of the pure theory of equality and based our further investigations on this observation. On the contrary, in other approaches (see e.g. [17]), it is emphasized that infinite models are just *free* models of the pure theory of equality with infinitely many generators. This leads to completely different results, because the notion of infinitely generated free and of existentially closed structure are quite divergent and their coincidence for the pure theory of equality must be considered a rather exceptional fact.

Before closing, we would like to remark that the idea (suggested in [15]) of using interpolation theorems in order to limit residue exchange in partial theory reasoning (whenever the background reasoner has to deal with combined theories) inspired some of the material presented in section 3 above. Notice however the following difference with respect to [15]: there the input theories T_1, T_2 were assumed to share all functions symbols (alien function symbols belonging to one theory being considered as free Skolem functions for the other), whereas we tried to keep function symbols separated too, as much as possible. This is essential in our context, because otherwise e.g. local finiteness of the common subtheory T_0 would be lost (and decidability of the combined problems presented in section 4 would not be achieved as a consequence).

Aknowledgements: I wish to thank Silvio Ranise and Cesare Tinelli for e-mail discussions on the subject of this paper.

References

[1] Armando A., Ranise S., Rusinowitch M., *Uniform Derivation of Superposition Based Decision Procedures*, "Proceedings of the Annual Conference on Computer Science Logic" (CSL01), Paris, France, pp. 513-527, (2001).

[2] Baader F., Tinelli C., *Combining Decision Procedures for Positive Theories Sharing Constructors*, in Sophie Tison (ed.) "Rewriting Techniques and Applications", 13th International Conference (RTA02), Springer LNCS 2378, pp. 352-366, (2002).

[3] Bachmair L., Ganzinger H. *Equational Reasoning in Saturation-Based Theorem Proving*, in Bibel L., Schmitt P.H. (eds.) "Automated Deduction - A Basis for Applications", vol. I, pp. 353-397, Kluwer (1998).

[4] Chang C.C., Keisler H.J., *Model Theory*, IIIrd edition, North Holland (1990).

[5] Ferrante J., Rackoff C.W., *The Computational Complexity of Logical Theories*, Springer Lecture Notes in Mathematics 718, (1979).

[6] Ghilardi S., *Reasoners Cooperation and Quantifier Elimination*, Rapporto Interno n. 288-03, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano (2003).

[7] Ghilardi S., Santocanale L., *Algebraic and Model Theoretic Techniques for Fusion Decidability in Modal Logic*, preprint (2003).

[8] Ghilardi S., Zawadowski M., *Sheaves, Games and Model Completions*, Trends in Logic Series, Kluwer (2002).

[9] Janicìc P., Bundy A., *A General Setting for Flexibly Combining and Augmenting Decision Procedures*, Journal of Automated Reasoning, 28, pp.257-305 (2002).

[10] MacIntyre A., *Model Completeness*, in Barwise J. (ed.), "Handbook of Mathematical Logic", North Holland, pp. 139-180 (1977).

[11] Nelson G., Oppen D., *Simplification by Cooperating Decision Procedures*, ACM Transactions on Programming Languages and Systems, 1(2), pp. 245-257 (1979).

[12] Nieuwhenhuis R., Rubio A., *Paramodulation-Based Theorem Proving*, in Robinson A., Voronkov A., (eds.) "Handbook of Automated Reasoning", vol. I, Elsevier/MIT, pp. 371-533 (2001).

[13] Oppen D., *Complexity, Convexity and Combination of Theories*, Theoretical Computer Science, 12, pp. 291-302 (1980).

[14] Oppen D., *Reasoning about Recursively Defined Data Structures*, Journal of the ACM, 27, 3, pp. 403-411 (1980).

[15] Tinelli C., *Cooperation of Background Reasoners in Theory Reasoning by Residue Sharing*, Journal of Automated Reasoning, 2003 (to appear).

[16] Tinelli C., Harandi M., *A New Correctness Proof of the Nelson-Oppen Combination Procedure*, in Baader F., Schulz K. (eds.) "1st International Workshop on Frontiers of Combining Systems (FroCos'96)", Applied Logic Series, vol. 3, Kluwer Academic Publishers, pp. 103-120 (1996).

[17] Tinelli C., Ringeissen C., *Unions of Non-Disjoint Theories and Combination of Satisfiability Procedures*, Theoretical Computer Science 290(1), pp.291-353, (2003).

[18] Wheeler W. H., *Model-Companions and Definability in Existentially Complete Structures*, Israel Journal of Mathematics, 25, pp.305-330 (1976).

[19] Wolter F., *Fusions of Modal Logics Revisited*, in Kracht M., De Rijke M., Wansing H., Zakharyaschev M. (eds.) "Advances in Modal Logic", CSLI, Stanford (1998).

Combining Non-Stably Infinite Theories

Cesare Tinelli [1]

Department of Computer Science
University of Iowa
Iowa City, IA 52242

Calogero G. Zarba [2]

Computer Science Department
Stanford University
Stanford, CA 94305, USA

Abstract

The Nelson-Oppen combination method combines decision procedures for first-order theories over disjoint signatures into a single decision procedure for the union theory. To be correct, the method requires that the component theories be stably infinite. This restriction makes the method inapplicable to many interesting theories such as, for instance, theories having only finite models.

In this paper we provide a new combination method that can combine any theory that is not stably infinite with another theory, provided that the latter is what we call a *shiny* theory. Examples of shiny theories include the theory of equality, the theory of partial orders, and the theory of total orders.

An interesting consequence of our results is that any decision procedure for the satisfiability of quantifier-free Σ-formulae in a Σ-theory T can always be extended to accept inputs over an arbitrary signature $\Omega \supseteq \Sigma$.

1 Introduction

An important research problem in automated reasoning asks how we can modularly combine decision procedures for theories T_1 and T_2 into a decision procedure for a combination of T_1 and T_2.

The most successful and well-known method for combining decision procedures was invented in 1979 by Nelson and Oppen [8]. This method is at the heart

[1] Email: tinelli@cs.uiowa.edu.
[2] Email: zarba@theory.stanford.edu.

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

of the verification systems CVC [12], ESC [3], EVES [2], and SDVS [6], among others.

The Nelson-Oppen method allows us to decide the satisfiability of quantifier-free formulae in a combination T of a theory T_1 and a theory T_2, by using as black boxes the decision procedures for the satisfiability of quantifier-free formulae in T_1 and in T_2. To be correct, the Nelson-Oppen method requires that the theories T, T_1, and T_2 satisfy the following restrictions:

- T is logically equivalent to $T_1 \cup T_2$;
- the signatures of T_1 and T_2 are disjoint;
- T_1 and T_2 are both stably infinite. [3]

There are several interesting combination problems that do not satisfy all these restrictions.

In this paper we concentrate on the issue of relaxing the stable infiniteness restriction. This is an important research problem at the theoretical level because it allows us to better understand the foundations of combination problems, and to prove more decidability results by combination techniques. But it is also interesting at a practical level because (i) proving that a given theory is stably infinite is not always easy, and (ii) many interesting theories, such as those admitting only finite models, are not stably infinite.

We show that when one component theory satisfies a stronger property than stable infiniteness, which we call *shininess*, [4] then the other component theory does not need to be stably infinite for their decision procedures to be combinable. We do that by providing and proving correct an extension of the Nelson-Oppen method that, in addition to propagating equality constraints between the component decision procedures, also propagates certain cardinality constraints.

Examples of shiny theories include the theory of equality, the theory of partial orders, and the theory of total orders. In particular, the fact that the theory of equality is shiny leads to a notable side result:

> **Result 1.** *If the satisfiability in a Σ-theory T of quantifier-free Σ-formulae is decidable, then the satisfiability in T of quantifier-free formulae over any arbitrary signature $\Omega \supseteq \Sigma$ is also decidable.*

Result 1 was proven by Policriti and Schwartz [11] for theories T that are universal. It was also known for theories T that are stably infinite, since in this case one can use the Nelson-Oppen method to combine the decision procedure for T with one for the theory of equality over the symbols in $\Omega \setminus \Sigma$. In this paper we prove that Result 1 holds regardless of whether T is universal or not, and regardless of whether T is stably infinite or not.

[3] See Definition 2.2.
[4] See Definition 2.5.

1.1 Related work.

Several researchers have worked on relaxing the requirements of the Nelson-Oppen combination method. The disjointness problem was addressed by Ghilardi [4], Tinelli [13], Tinelli and Ringeissen [15] and Zarba [20]. The stably infiniteness requirement was addressed by Baader and Tinelli [1] for combinations problems concerning the word problem, and by Zarba [17–19] for combinations of integers with lists, sets, and multisets. (The latter works by Zarba consider combination problems other than simple set-theoretic union.)

1.2 Organization of the paper

The paper is organized as follows. In Section 2 we introduce some preliminary notions, including the notion of a shiny theory. In Section 3 we describe our combination method. In Section 4 we provide two examples showing our method in action. In Section 5 we prove that our method is correct. In Section 6 we prove that the theory of equality is shiny. We conclude in Section 7 with directions for further research.

In order to focus on the main results, we omit here the proofs that the theories of partial and total orders are shiny. They can be found in the long version of this paper [16].

2 Preliminaries

A *signature* Σ is composed by a set Σ^{C} of constants, a set Σ^{F} of function symbols, and a set Σ^{P} of predicate symbols. We use the standard notions of (Σ-)term, atom, literal, formula, and sentence. We use $\approx$ to denote the equality logical symbol. We abbreviate with $s \not\approx t$ the negation of a literal $s \approx t$, and we identify a conjunction of formulae $\varphi_1 \wedge \cdots \wedge \varphi_n$ with the set $\{\varphi_1, \ldots, \varphi_n\}$.

If φ is a term or a formula, $vars(\varphi)$ denotes the set of variables occurring in φ. Similarly, if Φ is a set of terms or a set of formulae, $vars(\Phi)$ denotes the set of variables occurring in Φ.

For a signature Σ, a Σ-*interpretation* $\mathcal{A}$ with domain A over a set V of variables is a map which interprets each variable x as an element $x^{\mathcal{A}} \in A$, each constant $c \in \Sigma^{\mathrm{C}}$ as an element $c^{\mathcal{A}} \in A$, each function symbol $f \in \Sigma^{\mathrm{F}}$ of arity n as a function $f^{\mathcal{A}} : A^n \to A$, and each predicate symbol $P \in \Sigma^{\mathrm{P}}$ of arity n as a subset $P^{\mathcal{A}}$ of A^n. We adopt the convention that calligraphic letters $\mathcal{A}$, $\mathcal{B}$, ... denote interpretations, while the corresponding Roman letters A, B, ... denote the domains of the interpretations.

Let $\mathcal{A}$ be a Σ-interpretation over a set V of variables. For a Σ-term t over V, we denote with $t^{\mathcal{A}}$ the evaluation of t under the interpretation $\mathcal{A}$. Likewise, for a Σ-formula φ over V, we denote with $\varphi^{\mathcal{A}}$ the truth-value of φ under the interpretation $\mathcal{A}$. If T is a set of Σ-terms over V, we denote with $T^{\mathcal{A}}$ the set $\{t^{\mathcal{A}} \mid t \in T\}$.

A formula φ is *satisfiable*, if it is true under some interpretation, and *unsatisfi-*

able otherwise.

We use the standard model-theoretic notions of *embedding* and of *isomorphism* between interpretations [5].

Definition 2.1 Let Σ be a signature, and let $\mathcal{A}$ and $\mathcal{B}$ be Σ-interpretations over some set V of variables. A map $h : A \to B$ is an EMBEDDING of $\mathcal{A}$ into $\mathcal{B}$ if the following conditions hold:

- h is injective;
- $h(u^{\mathcal{A}}) = u^{\mathcal{B}}$ for each variable or constant $u \in V \cup \Sigma^{\mathrm{C}}$;
- $h(f^{\mathcal{A}}(a_1, \ldots, a_n)) = f^{\mathcal{B}}(h(a_1), \ldots, h(a_n))$, for each n-ary function symbol $f \in \Sigma^{\mathrm{F}}$ and $a_1, \ldots, a_n \in A$;
- $(a_1, \ldots, a_n) \in P^{\mathcal{A}}$ if and only if $(h(a_1), \ldots h(a_n)) \in P^{\mathcal{B}}$, for each n-ary predicate symbol $P \in \Sigma^{\mathrm{P}}$ and $a_1, \ldots, a_n \in A$.

An ISOMORPHISM of $\mathcal{A}$ into $\mathcal{B}$ is a surjective (and therefore bijective) embedding of $\mathcal{A}$ into $\mathcal{B}$.

A Σ-*theory* is any set of Σ-sentences. Given a Σ-theory T, a T-*model* is a Σ-interpretation that satisfies all sentences in T. A formula φ is T-*satisfiable* if it is satisfied by some T-model, and it is T-*unsatisfiable* otherwise. Given a set L of formulae, the *satisfiability problem* of T with respect to L is the problem of deciding, for each formula φ in L, whether or not φ is T-satisfiable. When we do not specify L, it is implicitly assumed that L is the set of all Σ-formulae. However, when we say "quantifier-free satisfiability problem", without specifying L, then we implicitly assume that L is the set of all quantifier-free Σ-formulae.

We use the usual notion of stable infiniteness for a theory, together with its "dual" one, which we call stable finiteness.

Definition 2.2 A Σ-theory T is STABLY INFINITE (respectively, STABLY FINITE) if every quantifier-free Σ-formula φ is T-satisfiable if and only if it is satisfied by a T-interpretation $\mathcal{A}$ whose domain A is infinite (respectively, finite).

Examples of stably infinite theories include the theory of equality, [5] the theory of integer arithmetic, the theory of rational arithmetic, the theory of lists, and the theory of arrays. Examples of stably finite theories include the theory of equality, all theories satisfied only by finite interpretations, and all theories finitely axiomatized by formulae in the Bernays-Schönfinkel-Ramsey class.

Note that a theory can be both stably finite and stably infinite. We will show that in Section 6 for the theory of equality.

Definition 2.3 A Σ-theory T is SMOOTH if for every quantifier-free Σ-formula φ, for every T-model $\mathcal{A}$ satisfying φ, and for every cardinal number $\kappa > |A|$ there exists a T-model $\mathcal{B}$ satisfying φ such that $|B| = \kappa$.

[5] Since we regard $\approx$ as a logical symbol, for us the theory of equality and the empty theory are the same theory.

A direct consequence of Definition 2.3 is that every smooth theory is stably infinite. The following proposition is useful when proving that a theory is smooth.

Proposition 2.4 *A Σ-theory T is smooth if and only if for every quantifier-free Σ-formula φ and every finite T-model $\mathcal{A}$ of φ, there exists a T-model $\mathcal{B}$ of φ such that $|B| = |A| + 1$.*

Given a theory T and a T-satisfiable quantifier-free formula φ, we denote with $mincard_T(\varphi)$ the smallest cardinality of a T-model satisfying φ. Note that if T is a stably finite theory then, for every T-satisfiable formula φ, $mincard_T(\varphi)$ is a natural number.

Definition 2.5 A Σ-theory T is SHINY if it is both smooth and stably finite, and such that $mincard_T$ is computable.

3 The combination method

Let S be a shiny Σ-theory and let T be an Ω-theory such that $\Sigma \cap \Omega = \emptyset$ and the quantifier-free satisfiability problems of S and of T are decidable. We now describe a method for combining decision procedures for the quantifier-free satisfiability problems of S and T into a single decision procedure for the quantifier-free satisfiability problem of $S \cup T$.

Since every quantifier free formula is logically equivalent to its disjunctive normal form, without loss of generality we restrict ourselves to conjunctions of literals. In addition, we consider only conjunctions of the form $\Gamma_1 \cup \Gamma_2$, which we call a *separate form*, where Γ_1 contains only Σ-literals and Γ_2 contains only Ω-literals. The latter restriction is also without loss of generality, as every conjunction Γ of $(\Sigma \cup \Omega)$-literals can be effectively converted into an equisatisfiable separate form $\Gamma_1 \cup \Gamma_2$ with the help of new auxiliary variables.

Let $\Gamma = \Gamma_1 \cup \Gamma_2$ be a conjunction of literals in separate form. The combination method consists of two phases, described below.

Decomposition phase. Nondeterministically guess an equivalence relation E over the set $V = vars(\Gamma_1) \cap vars(\Gamma_2)$ of variables shared by Γ_1 and Γ_2.

Check phase. Where E is the guessed equivalence relation over V, perform the following steps:

1. Construct the *arrangement* of V induced by E, defined by

$$arr(V, E) = \{x \approx y \mid x, y \in V, x \text{ and } y \text{ are distinct, and } (x, y) \in E\} \cup \{x \not\approx y \mid x, y \in V \text{ and } (x, y) \notin E\}.$$

2. If $\Gamma_1 \cup arr(V, E)$ is S-satisfiable go to the next step; otherwise output `fail`.
3. Compute $n = mincard_S(\Gamma_1 \cup arr(V, E))$.
4. Construct a set δ_n of literals whose purpose is to force models with cardinality at least n. More precisely, let $\delta_n = \{w_i \not\approx w_j \mid 1 \leq i < j \leq n\}$, where $w_1, \ldots, w_n$ are new variables not occurring in $\Gamma_1 \cup \Gamma_2$.

5. If $\Gamma_2 \cup arr(V, E) \cup \delta_n$ is T-satisfiable output `succeed`; otherwise output `fail`.

In Section 5 we will prove that (i) if the check phase outputs `succeed` for some equivalence relation E over V, then Γ is $(S \cup T)$-satisfiable, and (ii) if the check phase outputs `fails` for each equivalence relation E over V, then Γ is $(S \cup T)$-unsatisfiable.

Our combination method differs from the Nelson-Oppen method as follows. In the check phase, the Nelson-Oppen method omits steps 3 and 4, and in step 5 it checks the T-satisfiability of $\Gamma_2 \cup arr(V, E)$ only. Note that this is enough in the Nelson-Oppen method because there T is assumed to be stably infinite, and therefore the constraint δ_n is guaranteed to hold.

Note that our method applies just as well in case T is stably-infinite. [6] However, if one knows that T is stably infinite, resorting to the original Nelson-Oppen method is more appropriate, as it lets one avoid the cost of computing $mincard_S$.

4 Examples

In this section we discuss two examples of theories that are not combinable with the Nelson-Oppen method but are combinable with ours. In both examples we combine the theory S of equality over a signature Σ with a non-stably infinite theory T over a signature Ω disjoint from Σ. In the first case, T is not stably infinite because it only admits finite models. In the second case, T is not stably infinite even if it has infinite models. The examples are adapted from [14] and [1], respectively, where they are used to show that the Nelson-Oppen method is in fact incorrect on non-stably infinite theories.

Example 4.1 Let $\Sigma = \{f\}$ and $\Omega = \{g\}$ be signatures, where f and g are distinct unary function symbols. Let S be the theory of equality over the signature Σ, and let T be an Ω-theory such that all T-interpretations have cardinality at most two. Since T is not stably infinite, we cannot use the Nelson-Oppen combination method. But since S is shiny, we can use our method.

Let $\Gamma = \Gamma_1 \cup \Gamma_2$, where

$$\Gamma_1 = \{f(x) \not\approx f(y)\,,\ f(x) \not\approx f(z)\} \text{ and } \Gamma_2 = \{g(y) \not\approx g(z)\}\,.$$

Note that Γ is $(S \cup T)$-unsatisfiable. In fact, Γ implies $x \not\approx y \wedge x \not\approx z \wedge y \not\approx z$, and therefore every interpretation satisfying Γ must have cardinality at least three. Since every $(S \cup T)$-interpretation has at most two elements, it follows that Γ is $(S \cup T)$-unsatisfiable.

Let us apply our combination method to Γ. Since $vars(\Gamma_1) \cap vars(\Gamma_2) = \{y, z\}$, there are only two equivalence relations available for guessing: either $(y, z) \in E$ or $(y, z) \notin E$.

[6] Recall that S is already stably infinite, since it is shiny.

If $(y, z) \in E$ we have that $\Gamma_1 \cup \{y \approx z\}$ is S-satisfiable and that $\Gamma_2 \cup \{y \approx z\}$ is T-unsatisfiable. Thus, we will output `fail` when reaching step 4 of the check phase.

If instead $(y, z) \notin E$ then $\Gamma_1 \cup \{y \not\approx z\}$ is S-satisfiable. In addition, we have $mincard_S(\Gamma_1 \cup \{y \not\approx z\}) = 3$. To see this, first observe that $\Gamma_1 \cup \{y \not\approx z\}$ implies $x \not\approx y \wedge x \not\approx z \wedge y \not\approx z$, and therefore $mincard_S(\Gamma_1 \cup \{y \not\approx z\}) \geq 3$. In addition, we can construct an interpretation $\mathcal{A}$ of cardinality 3 satisfying $\Gamma_1 \cup \{y \not\approx z\}$ by letting $A = \{a_1, a_2, a_3\}$, $x^{\mathcal{A}} = a_1$, $y^{\mathcal{A}} = a_2$, $z^{\mathcal{A}} = a_3$, and $f^{\mathcal{A}}(a) = a$, for each $a \in A$.[7] In the third step of the check phase we introduce three new variables w_1, w_2, w_3, and construct δ_3 as the set $\{w_1 \not\approx w_2, w_1 \not\approx w_3, w_2 \not\approx w_3\}$. Since $\Gamma_2 \cup \{y \not\approx z\} \cup \delta_3$ is T-unsatisfiable, in the fourth step we output `fail`. We can therefore declare that Γ is $(S \cup T)$-unsatisfiable.

Example 4.2 Let $\Sigma = \{k\}$ and $\Omega = \{f, g, h\}$ be signatures, where k, f and g are distinct unary function symbols. Let S be again the theory of equality over the signature Σ, and let T be the following equational theory:

$$T = \left\{ \begin{array}{l} (\forall x)(\forall y)(x \approx f(g(x), g(y))), \\ (\forall x)(\forall y)(f(g(x), h(y)) \approx y) \end{array} \right\}.$$

Using simple term rewriting arguments, it is possible to show that T admits models of cardinality greater than one, and so admits models of infinite cardinality.[8] However, T is not stably infinite.

In fact, consider the set quantifier-free formula $g(z) \approx h(z)$. This formula is T-satisfiable because both the formula and T admit a trivial model, that is, a model with just one element. Now let $\mathcal{A}$ be any T-model of $g(z) \approx h(z)$, let $a_0 = z^{\mathcal{A}}$, and let $a \in A$. Because of T's axioms, we have that

$$a = f^{\mathcal{A}}(g^{\mathcal{A}}(a), g^{\mathcal{A}}(a_0)) = f^{\mathcal{A}}(g^{\mathcal{A}}(a), h^{\mathcal{A}}(a_0)) = a_0 \,.$$

Given that a is arbitrary, this entails that $|A| = 1$. Thus, $g(z) \approx h(z)$ is only satisfiable in trivial models of T, and therefore the theory T is not stably infinite.

For an application of our combination method to S and T, let $\Gamma = \Gamma_1 \cup \Gamma_2$, where

$$\Gamma_1 = \{g(z) \approx h(z)\} \text{ and } \Gamma_2 = \{k(z) \not\approx z\} \,.$$

The conjunction Γ is $(S \cup T)$-unsatisfiable, because $g(z) \approx h(z)$ is satisfiable only in trivial models of $S \cup T$ (for being satisfiable only in trivial models of T, as seen above), while $k(z) \not\approx z$ is clearly satisfiable only in non-trivial models of $S \cup T$.

Let us apply our combination method to Γ. Since $vars(\Gamma_1) \cap vars(\Gamma_2) = \{z\}$, in the check phase there are no equivalence relations to examine, therefore we

[7] We will see how to effectively compute $mincard_S$ in Section 6.

[8] This is because the set of models of an equational theory is closed under direct products.

generate the empty arrangement. Clearly, Γ_1 is S-satisfiable, and in models of cardinality at least 2. Therefore, we have that $mincard_S(\Gamma_1) = 2$.

In the third step of the check phase, we then compute δ_2 as the set $\{w_1 \not\approx w_2\}$ for some fresh variables w_1, w_2. For what we argued above, $\Gamma_2 \cup \delta_2$ is T-unsatisfiable, so in the fourth step we output `fail`, as needed.

5 Correctness

In this section we prove that our combination method is correct.

Clearly, our combination method is terminating. This follows from the fact that, since there is only a finite number of equivalence relations over a finite set V of variables, the nondeterministic decomposition phase is finitary. Thus, we only need to prove that our method is also partially correct.

We will use the following theorem which is a special case of a more general combination result given in [15] for theories with possibly non-disjoint signatures. A direct proof of this theorem can be found in [7].

Theorem 5.1 (Combination Theorem for Disjoint Signatures) *Let Φ_i be a set of Σ_i-formulae, for $i = 1, 2$, and let $\Sigma_1 \cap \Sigma_2 = \emptyset$.*

Then $\Phi_1 \cup \Phi_2$ is satisfiable if and only if there exists an interpretation $\mathcal{A}$ satisfying Φ_1 and an interpretation $\mathcal{B}$ satisfying Φ_2 such that:

(i) $|A| = |B|$,

(ii) $x^{\mathcal{A}} = y^{\mathcal{A}}$ *if and only if* $x^{\mathcal{B}} = y^{\mathcal{B}}$, *for every* $x, y \in vars(\Phi_1) \cap vars(\Phi_2)$.

The following proposition proves that our method is partially correct.

Proposition 5.2 *Let S be a shiny Σ-theory and let T be an Ω-theory such that $\Sigma \cap \Omega = \emptyset$. Let Γ_1 be a conjunction of Σ-literals and Γ_2 a conjunction of Ω-literals. Where $V = vars(\Gamma_1) \cap vars(\Gamma_2)$, the following are equivalent:*

(i) $\Gamma_1 \cup \Gamma_2$ *is $(S \cup T)$-satisfiable.*

(ii) *There exists an equivalence relation E over V such that $\Gamma_1 \cup arr(V, E)$ is S-satisfiable and $\Gamma_2 \cup arr(V, E) \cup \delta_n$ is T-satisfiable, with $n = mincard_S(\Gamma_1 \cup arr(V, E))$.*

Proof. ($1 \Rightarrow 2$). Assume that $\Gamma_1 \cup \Gamma_2$ is $(S \cup T)$-satisfiable, and let $\mathcal{F}$ be one of its $(S \cup T)$-models. Let $E = \{(x, y) \mid x, y \in V \text{ and } x^{\mathcal{F}} = y^{\mathcal{F}}\}$.

Clearly, $\mathcal{F}$ is an $(S \cup T)$-model of $\Gamma_1 \cup \Gamma_2 \cup arr(E, V)$. It follows that $\mathcal{F}$ is also an S-model of $\Gamma_1 \cup arr(E, V)$. In addition, $\mathcal{F}$ is a T-model of $\Gamma_2 \cup arr(E, V)$. Let $\kappa = |F|$, and let $n = mincard_S(\Gamma_1 \cup arr(V, E))$. By definition of $mincard_S$, we have $n \leq \kappa$, which implies that $\mathcal{F}$ is also a T-model of $\Gamma_2 \cup arr(E, V) \cup \delta_n$.

($2 \Rightarrow 1$). Let $V_1 = vars(\Gamma_1)$ and $V_2 = vars(\Gamma_2 \cup \delta_n)$, and observe that $V_1 \cap V_2 = V$. Assume there is an equivalence relation E of V such that $\Gamma_1 \cup arr(V, E)$ is S-satisfiable and $\Gamma_2 \cup arr(V, E) \cup \delta_n$ is T-satisfiable, where $n = mincard_S(\Gamma_1 \cup$

$arr(V, E)$). Then there exist an S-model $\mathcal{A}$ of $\Gamma_1 \cup arr(V, E)$ and a T-model $\mathcal{B}$ of $\Gamma_2 \cup arr(V, E) \cup \delta_n$.

Since $\mathcal{B}$ satisfies δ_n, we have $|B| \geq n$. Thus, by the smoothness of S, we can assume without loss of generality that $|A| = |B|$. In addition, because both $\mathcal{A}$ and $\mathcal{B}$ satisfy $arr(V, E)$, we have that $x^{\mathcal{A}} = y^{\mathcal{A}}$ if and only if $x^{\mathcal{B}} = y^{\mathcal{B}}$, for all $x, y \in V$. By Theorem 5.1, $S \cup T \cup \Gamma_1 \cup \Gamma_2 \cup arr(V, E) \cup \delta_n$ is satisfiable. Thus, $\Gamma_1 \cup \Gamma_2$ is $(S \cup T)$-satisfiable. □

Combining Proposition 5.2 with the fact that our combination method is terminating, we obtain the following decidability result.

Theorem 5.3 *Let S be a shiny Σ-theory and let be T an Ω-theory such that $\Sigma \cap \Omega = \emptyset$. If the quantifier-free satisfiability problems of S and of T are decidable, then the quantifier-free satisfiability problem of $S \cup T$ is also decidable.*

6 The theory of equality

It is known that the theory of equality (over an arbitrary signature) is stably infinite and has a decidable quantifier-free satisfiability problem [10]. We show here that it is also shiny.

We will use the following basic lemma of model theory [5].

Lemma 6.1 *Let $\mathcal{A}, \mathcal{B}$ be two interpretations such that there is an embedding of $\mathcal{A}$ into $\mathcal{B}$, and let φ be a quantifier-free formula. Then φ is satisfied by $\mathcal{A}$ if and only if it is satisfied by $\mathcal{B}$.*

Proposition 6.2 *Let φ be a quantifier-free formula, and let $\mathcal{A}$ be a finite model of φ. Then there exists a model $\mathcal{B}$ of φ such that $|B| = |A| + 1$.*

Proof. Let $k = |A|$. We construct a Σ-model $\mathcal{B}$ of φ such that $|B| = k + 1$ as follows. Let $B = A \cup \{b\}$, where $b \notin A$. Then, fix an arbitrary element $a_0 \in B$, and let

- for variables and constants: $u^{\mathcal{B}} = u^{\mathcal{A}}$,
- for function symbols of arity n:

$$f^{\mathcal{B}}(a_1, \ldots, a_n) = \begin{cases} f^{\mathcal{A}}(a_1, \ldots, a_n)\,, & \text{if } a_1, \ldots, a_n \in A\,, \\ a_0\,, & \text{otherwise,} \end{cases}$$

- for predicate symbols of arity n:

$$(a_1, \ldots, a_n) \in P^{\mathcal{B}} \iff a_1, \ldots, a_n \in A \text{ and } (a_1, \ldots, a_n) \in P^{\mathcal{A}}\,.$$

We have $|B| = k + 1$. In addition, the map $h : A \to B$ defined by $h(a) = a$, for each $a \in A$, is an embedding of $\mathcal{A}$ into $\mathcal{B}$. Since $\mathcal{A}$ satisfies φ, by Lemma 6.1 it follows that $\mathcal{B}$ also satisfies φ. □

Input: An S-satisfiable conjunction Γ of Σ-literals
Output: $mincard_S(\Gamma)$
1: **if** Γ is empty **then**
2: **return** 1
3: **else**
4: $U \leftarrow \text{TERMS}(\Gamma)$
5: $\Gamma' \leftarrow \Gamma$
6: **for** $s, t \in U$ **do**
7: **if** $\Gamma' \cup \{s \approx t\}$ is S-satisfiable **then**
8: $\Gamma' \leftarrow \Gamma' \cup \{s \approx t\}$
9: **end if**
10: **end for**
11: $E \leftarrow \{(s,t) \mid s \approx t \in \Gamma'\}$
12: $C \leftarrow \text{CONG-CLOSURE}(E)$
13: **return** $\text{CARD}(U/_C)$
14: **end if**

Fig. 1. A procedure for $mincard_S$.

Combining Propositions 2.4 and 6.2, we obtain the smoothness of the theory of equality.

Proposition 6.3 *For every signature Σ, the Σ-theory of equality is smooth.*

Next, we show that $mincard_S(\varphi)$ is computable when S is the theory of equality. A procedure that computes $mincard_S$ is given in Figure 1.

In the procedure, the function TERMS returns the set of all terms and subterms occurring in its input Γ. For instance, if $\Gamma = \{f(g(x)) \approx g(f(y))\}$ then $\text{TERMS}(\Gamma)$ returns the set $\{x, g(x), f(g(x)), y, f(y), g(f(y))\}$. The function CONG-CLOSURE computes the congruence closure of the binary relation E over the signature of Γ. [9] $U/_C$ denotes the quotient of U with respect to the congruence relation C.

Both C and $U/_C$ can be computed using any standard congruence closure algorithm [9]. The complexity of such algorithms is (no more than) $\mathcal{O}(n^2)$, where n is the cardinality of U. The test in line 7 can be performed by the same congruence closure algorithm used for computing C. Since the procedure in Figure 1 is clearly terminating, it then follows that its complexity is $\mathcal{O}(n^4)$.

We show below that the procedure is also partially correct.

Proposition 6.4 *For every input Γ, the procedure shown in Figure 1 returns $mincard_S(\Gamma)$.*

Proof. If Γ is empty then Γ is satisfied by every interpretation. Thus, in this case the procedure returns the correct value $mincard_S(\Gamma) = 1$.

Let us consider the case in which Γ is not empty. Let U, Γ', E, and C be as computed by the procedure. Moreover, let k be the value returned in line 13. Note

[9] Given a binary relation E, the congruence closure of E is the smallest congruence C containing E.

that Γ' is S-satisfiable, and that $\Gamma \subseteq \Gamma'$. Thus, every model of Γ' is also a model of Γ. Finally, since Γ is not empty, then U is also not empty. It follows that the quotient $U/_C$ is not empty, hence $k \geq 1$.

Let $\mathcal{A}$ be any model of Γ', and consider the set $B = \{t^{\mathcal{A}} \mid t \in U\}$.

We claim that $|B| = k$. To see this, suppose, for a contradiction, that $|B| \neq k$. Assume first that $|B| < k$. Since k is equal to the number of equivalence classes of C, there exist two terms $s, t \in U$ such that $(s,t) \notin C$ and $s^{\mathcal{A}} = t^{\mathcal{A}}$. But then $\Gamma' \cup \{s \approx t\}$ is satisfied by $\mathcal{A}$, which implies that $s \approx t \in \Gamma'$. It follows that $(s,t) \in E$, and therefore $(s,t) \in C$, a contradiction.

Next, suppose that $|B| > k$. Then there exist distinct terms $t_1, \ldots, t_n$, with $n > k$, such that $t_i^{\mathcal{A}} \neq t_j^{\mathcal{A}}$, for $i < j$. Since C is the congruence closure of E, it follows that, for every term s, t, if $(s,t) \in C$ then $s^{\mathcal{A}} = t^{\mathcal{A}}$. But then, for every term s, t, if $s^{\mathcal{A}} \neq t^{\mathcal{A}}$ then $(s,t) \notin C$. Thus, $(t_i, t_j) \notin C$, for $i < j$. It follows that C has more than k equivalence classes, a contradiction.

Since $|B| = k$, by the generality of $\mathcal{A}$, we can conclude that every model of Γ has at least k elements.

We now construct a model $\mathcal{B}$ of Γ with domain B. The proposition's claim will then follow from the fact that $|B| = k$.

Let b be some element of B. We define

- for variables and constants:

$$u^{\mathcal{B}} = \begin{cases} u^{\mathcal{A}}\,, & \text{if } u^{\mathcal{A}} \in B\,, \\ b\,, & \text{otherwise}\,, \end{cases}$$

- for function symbols of arity n:

$$f^{\mathcal{B}}(b_1, \ldots, b_n) = \begin{cases} f^{\mathcal{A}}(b_1, \ldots, b_n)\,, & \text{if } f^{\mathcal{A}}(b_1, \ldots, b_n) \in B\,, \\ b\,, & \text{otherwise,} \end{cases}$$

- for predicate symbols of arity n:

$$(b_1, \ldots, b_n) \in P^{\mathcal{B}} \quad \Longleftrightarrow \quad (b_1, \ldots, b_n) \in P^{\mathcal{A}}\,.$$

By structural induction, one can show that $t^{\mathcal{B}} = t^{\mathcal{A}}$ for all terms $t \in U$, and that $\ell^{\mathcal{B}} = \ell^{\mathcal{A}}$ for all literals $\ell \in \Gamma'$. It follows that $\mathcal{B}$ satisfies Γ'. Since $\Gamma \subseteq \Gamma'$, $\mathcal{B}$ also satisfies Γ. □

As an immediate corollary of Proposition 6.4, we obtain the following result.

Proposition 6.5 *For every signature Σ, the Σ-theory of equality is stably finite.*

Putting together Propositions 2.4, 6.4, and 6.5, we obtain the shininess of the theory of equality.

Proposition 6.6 *For every signature Σ, the Σ-theory of equality is shiny.*

Proposition 6.6 is relevant because, together with our combination method in Section 3, it tells us that any procedure that decides the quantifier-free satisfiability problem for a Σ-theory T can be extended to accept inputs Γ containing arbitrary free symbols in addition to the symbols in Σ.

This fact was already known for theories T that are universal [11]. It was also known for theories T that are stably-infinite, since in this case one can use the Nelson-Oppen method to combine the decision procedure for T with one for the theory of equality over the symbols of Γ that are not in Σ. Thanks to Proposition 6.6 and our combination method, we are able to lift the universal and/or stable-infiniteness requirement for T altogether.

More formally, we have the following theorem.

Theorem 6.7 *Let T be a Σ-theory such that the quantifier-free satisfiability problem of T is decidable. Then, for every signature $\Omega \supseteq \Sigma$, the quantifier-free satisfiability problem of T with respect to Ω-formulae is decidable.*

7 Conclusion

We have addressed the problem of extending the Nelson-Oppen combination method to pairs of theories that are not stably infinite. We provided a modification of the Nelson-Oppen method in which it is possible to lift the stable infiniteness requirement from one theory, provided that the other one satisfies a stronger condition, which we called shininess.

Examples of shiny theories include the theory of equality, the theory of partial orders, and the theory of total orders.

In particular, the shininess of the theory of equality yields an interesting useful result: Any decision procedure for the quantifier-free satisfiability problem of a theory T can be extended to accept input formulae over an arbitrary signature. The usefulness of this result stems from the fact that, in practice, satisfiability problems in a theory T often contain free function symbols in addition to the original symbols of T. These function symbols are typically introduced by skolemization or abstraction processes. Our result says that these symbols can be always dealt with properly, no matter what T is.

The Nelson-Oppen method is applicable to an arbitrary number of stably infinite and pairwise signature-disjoint theories. Similarly, our method can be extended to the combination of one arbitrary theory and $n > 1$ shiny theories, all pairwise signature-disjoint. In is unlikely that our method can be extended to allow more than one arbitrary theory. In fact, if this were the case, we would be able to combine two arbitrary theories.

The correctness proof of both the Nelson-Oppen method and our method relies on the Combination Theorem for Disjoint Theories (Theorem 5.1). That theorem requires that the two parts of a separate form of an input formula be satisfied in models of the respective theories having the same cardinality. This requirement is impossible to check in general [15]. Considering only stably infinite theories, as

done in the original method, allows one to completely forgo the check, because stably infinite theories always satisfy it. Our method deals with the cardinality requirement by assuming enough on one theory, the shiny one, so that a simpler cardinality check, the one represented by δ_n, can be performed on the other.

We plan to continue our research on relaxing the stable infiniteness requirement by aiming at finding general sufficient conditions for shininess, and at identifying additional specific examples of shiny theories.

References

[1] Baader, F. and C. Tinelli, *A new approach for combining decision procedures for the word problem, and its connection to the Nelson-Oppen combination method*, in: W. McCune, editor, *Automated Deduction – CADE-14*, Lecture Notes in Computer Science **1249** (1997), pp. 19–33.

[2] Craigen, D., S. Kromodimoeljo, I. Meisels, B. Pase and M. Saaltink, *EVES: An overview*, in: S. Prehen and H. Toetenel, editors, *Formal Software Development Methods*, Lecture Notes in Computer Science **552** (1991), pp. 389–405.

[3] Detlefs, D. L., K. R. M. Leino, G. Nelson and J. B. Saxe, *Extended static checking*, Technical Report 159, Compaq System Research Center (1998).

[4] Ghilardi, S., *Quantifier elimination and provers integration*, Electronic Notes in Theoretical Computer Science **86** (2003).

[5] Hodges, W., "A Shorter Model Theory," Cambridge University Press, 1997.

[6] Levy, B., I. Filippenko, L. Marcus and T. Menas, *Using the state delta verification system (SDVS) for hardware verification*, in: T. F. Melham, V. Stavridou and R. T. Boute, editors, *Theorem Prover in Circuit Design: Theory, Practice and Experience* (1992), pp. 337–360.

[7] Manna, Z. and C. G. Zarba, *Combining decision procedures*, in: *Formal Methods at the Cross Roads: From Panacea to Foundational Support*, Lecture Notes in Computer Science (2003), to appear.

[8] Nelson, G. and D. C. Oppen, *Simplification by cooperating decision procedures*, ACM Transactions on Programming Languages and Systems **1** (1979), pp. 245–257.

[9] Nelson, G. and D. C. Oppen, *Fast decision procedures based on congruence closure*, Journal of the ACM **27** (1980), pp. 356–364.

[10] Oppen, D. C., *Complexity, convexity and combination of theories*, Theoretical Computer Science **12** (1980), pp. 291–302.

[11] Policriti, A. and J. T. Schwartz, *T-theorem proving I*, Journal of Symbolic Computation **20** (1995), pp. 315–342.

[12] Stump, A., C. W. Barret and D. L. Dill, *CVC: A cooperating validity checker*, in: E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification*, Lecture Notes in Computer Science **2404**, 2002, pp. 500–504.

[13] Tinelli, C., *Cooperation of background reasoners in theory reasoning by residue sharing*, Journal of Automated Reasoning **30** (2003), pp. 1–31.

[14] Tinelli, C. and M. T. Harandi, *A new correctness proof of the Nelson-Oppen combination procedure*, in: F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems*, Applied Logic Series **3** (1996), pp. 103–120.

[15] Tinelli, C. and C. Ringeissen, *Unions of non-disjoint theories and combinations of satisfiability procedures*, Theoretical Computer Science **290** (2003), pp. 291–353.

[16] Tinelli, C. and C. G. Zarba, *Combining non-stably infinite theories*, Technical report, University of Iowa (2003), electronically available at `ftp://ftp.cs.uiowa.edu/pub/tinelli/papers/TinZar-RR-03.pdf`.

[17] Zarba, C. G., *Combining lists with integers*, in: R. Goré, A. Leitsch and T. Nipkow, editors, *International Joint Conference on Automated Reasoning: Short Papers*, Technical Report DII 11/01, Università di Siena, 2001, pp. 170–179.

[18] Zarba, C. G., *Combining multisets with integers*, in: A. Voronkov, editor, *Automated Deduction – CADE-18*, Lecture Notes in Computer Science **2392** (2002), pp. 363–376.

[19] Zarba, C. G., *Combining sets with integers*, in: A. Armando, editor, *Frontiers of Combining Systems*, Lecture Notes in Computer Science **2309** (2002), pp. 103–116.

[20] Zarba, C. G., *A tableau calculus for combining non-disjoint theories*, in: U. Egly and C. G. Fermüller, editors, *Automated Reasoning with Analytical Tableaux and Related Methods*, Lecture Notes in Computer Science **2381** (2002), pp. 315–329.

Can Decision Procedures be Learnt Automatically?

Mateja Jamnik [1,2]

University of Cambridge Computer Laboratory
J.J. Thomson Avenue, Cambridge, CB3 0FD, England, UK
www.cl.cam.ac.uk/~mj201

Predrag Janičić [3]

Faculty of Mathematics, University of Belgrade, Studentski trg 16
11000 Belgrade, Serbia and Montenegro
www.matf.bg.ac.yu/~janicic

Abstract

In this paper we present an investigation into whether and how can decision procedures be learnt automatically. Our approach consists of two stages. First, a refined brute-force search procedure applies exhaustively a set of given elementary methods to try to solve a corpus of conjectures generated by a stochastic context-free grammar. The successful proof traces are saved. In the second stage, a learning algorithm (by Jamnik et al.) tries to extract a required supermethod (i.e., decision procedure) from the given traces. In the paper, this technique is applied to elementary methods that encode the operations of the Fourier-Motzkin's decision procedure for Presburger arithmetic on rational numbers. The results of our experiment are encouraging.

1 Introduction

Learning proof methods and programs is a challenging task. Jamnik and colleagues [7] devised a framework for proof planning [4] systems where new proof methods can be learnt automatically (the implementation of this framework is called LEARNΩMATIC [8]). In this approach, a proof planning system is used to

[1] The first author was supported by the EPSRC Advanced Research Fellowship, and the second author was supported by EPSRC grant GR/R52954/01 and by the Serbian Ministry of Science research grant 1379.

[2] Email: mateja.jamnik@cl.cam.ac.uk

[3] Email: janicic@matf.bg.ac.yu

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

construct examples of proofs that use similar reasoning patterns. These proofs consist of low level inference steps or proof methods that are available to the system initially. The goal is to learn a procedure which uses these methods in some structured and efficient way. In order to learn such a procedure, a series of example proofs is generated automatically. The traces of example proofs are then fed into the learning mechanism which learns the so-called *method outline*, which captures the pattern common to all of the example proofs. Finally, the representation of a learnt method outline is enriched into a fully fleshed proof method so that it can be used by a specific proof planning system of choice. Such a learnt proof method is then used in subsequent proof planning attempts for other conjectures.

In this paper, we discuss how the learning approach in LEARNΩMATIC (for background, see §2.1) can be extended and used for a wider range of domains and procedures. In particular, we apply LEARNΩMATIC to developing decision procedures (for background, see §2.2). This is a challenging task as the learnt method should be terminating, sound and complete. Learning decision procedures automatically would be beneficial for a reasoning system, especially for user defined theories or when for some theory a decision procedure is not available. So, our main motivation is a mechanisation of learning and discovery of new decision procedures (while learning existing decision procedures serves as an illustration of an important step towards the final goal). Learning new decision procedures automatically can reduce the time required for developing them, it can prevent human implementation flaws, and presents a generic approach (that is independent of the theory) to generating decision procedures. We propose the programme and demonstrate how it can yield one specific procedure — Fourier-Motzkin's decision procedure [12] (the proposed framework can, of course, be used for other proof methods as well).

While our larger aim is to discover new procedures, we start by learning an existing procedure. This is a difficult task, since even if the idea of the required procedure is know and all the building blocks are available, it is still very challenging to combine them correctly into the require decision procedure. Our framework does not provide full automation (or guarantees formal properties, such as termination), however, it can be used as a very useful mechanised assistant. The user needs to provide the necessary building blocks and also some guidance to refine the brute force search according to the specific theory, in order to construct examples for automatic learning which generates the decision procedure.

In the research presented in this paper, we used the system LEARNΩMATIC [8], while all other discussed/used algorithms and modules were newly developed (and serve as an extension to LEARNΩMATIC).

Our programme (which also reflects the structure of this paper) consists of the following steps (we illustrate our approach with the example of linear arithmetic and the Fourier-Motzkin's procedure):

- the methods that can make up a decision procedure are provided (§3);
- the examples of proofs using the given methods are constructed (§4); this requires:

  - a number of conjectures is generated randomly (4.1);
  - implementing a simple PROLOG deduction system (which essentially carries out a brute force search) that applies the given methods (4.2);
  - grouping and ordering of methods to direct the brute force search and to prevent non-termination in the process of generating proof examples (4.3);
  - all example proofs are divided into groups according to a number of variables; from each group the most illustrative proofs are taken; all these selected proofs make the learning set (4.4).
- the selected example proofs are input into the learning mechanism which learns a procedure that captures the pattern of reasoning employed in all of the example proofs (§5);
- on the basis of the learnt pattern, a PROLOG mechanism automatically generates a corresponding supermethod (also in PROLOG), which is our required decision procedure (§6);
- the learnt procedure is tested on the original set of examples (§7).

We finish the paper with a brief discussion of related work in §8, and conclusions and future directions in §9.

2 Background

2.1 Automatic learning

Jamnik et al [7] devised a framework within which a proof planning [4] system can learn frequently occurring patterns of reasoning automatically from a number of typical examples, and then use them in proving new theorems [9]. The availability of such patterns, captured as proof methods in a proof planning system, reduces search and proof length. Jamnik et al implemented this learning framework for the proof planner ΩMEGA [2] – they call the system LEARNΩMATIC. The entire process of learning and using new proof methods in LEARNΩMATIC consists of the following steps:

(i) The user chooses informative examples and gives them to ΩMEGA to be automatically proved. Traces of these proofs are stored.

(ii) Proof traces of typical examples are given to the learning mechanism which automatically learns so-called *method outlines*.

(iii) Method outlines are automatically enriched by adding to them additional information and performing search for information that cannot be reconstructed in order to get fully fleshed proof methods that ΩMEGA can use in proofs of new theorems.

The methods LEARNΩMATIC aims to learn are complex and are beyond the complexity that can typically be tackled in the field of machine learning. Therefore, LEARNΩMATIC learns *method outlines*, which are expressed in the following lan-

guage L, where P is a set of known identifiers of primitive methods used in a method that is being learnt:

- for any $p \in P$, let $p \in L$,
- for any $l_1, l_2 \in L$, let $[l_1, l_2] \in L$,
- for any $l_1, l_2 \in L$, let $[l_1|l_2] \in L$,
- for any $l \in L$, let $l^* \in L$,
- for any $l \in L$ and $n \in \mathbf{N}$, let $l^n \in L$,
- for any $list$ such that all $l_i \in list$ are also $l_i \in L$, let $T(list) \in L$.

"[" and "]" are auxiliary symbols used to separate subexpressions, "," denotes a *sequence*, "|" denotes a *disjunction*, "$*$" denotes a *repetition* of a subexpression any number of times (including 0), n a fixed number of times, and T is a constructor for a branching point ($list$ is a list of branches), i.e., for proofs which are not sequences but branch into a tree. For more information on the expressiveness of this language, the reader is referred to [9].

Our learning technique considers some typically small number of positive examples which are represented in terms of sequences of identifiers for primitive methods, and generalises them so that the learnt pattern is in language L. The pattern is of *smallest size* with respect to a defined heuristic measure of *size* [9], which essentially counts the number of primitives in an expression. The pattern is also *most specific* (or equivalently, least general) with respect to the definition of specificity *spec*. *spec* is measured in terms of the number of nestings for each part of the generalisation [9]. Again, this is a heuristic measure.

The algorithm is based on the generalisation of the simultaneous compression of well-chosen examples. Here is just an abstract description of the learning algorithm, but the detailed steps with examples of how they are applied can be found in [9]:

(i) Split every example trace into sublists of all possible lengths.

(ii) If there is any branching in the examples, then recursively repeat this algorithm on every element of the list of branches.

(iii) For each sublist in each example find consecutive repetitions, i.e., patterns, and compress them using exponent representation.

(iv) Find compressed patterns that match in all examples.

(v) If there are no matches in the previous step, then generalise the examples by joining them disjunctively.

(vi) For every match, generalise different exponents to a Kleene star, and the same exponents to a constant.

(vii) For every matching pattern in all examples, repeat the algorithm on both sides of the pattern.

(viii) Choose the generalisations with the smallest size and largest specificity.

The learning algorithm is implemented in SML of NJ v.110. Its inputs are the sequences of methods extracted from proofs. Its output are method outlines.

2.2 Decision procedures

A theory $\mathcal{T}$ is *decidable* if there is an algorithm (which we call a *decision procedure*) such that for an input $\mathcal{T}$-sentence F, it returns *true* if and only if F is valid in $\mathcal{T}$ (i.e., $\mathcal{T} \models F$) and returns *false* otherwise. The role of decision procedures is often very important in theorem proving (e.g., see [10]). Decision procedures can reduce the search space of heuristic components of a prover and increase its abilities. Decision procedures can usually be much more efficient than some other proving strategies (e.g., induction). There are many decision procedures in standard use, including decision procedures for fragments of arithmetics, theories of lists, theory of equality etc. Due to its importance in hardware and software verification, decision procedures for fragments of arithmetic (like PRA — Presburger Rational Arithmetic) are of particular interest.

Instead of using basic inference rules, decision procedures are usually built from some higher-level building blocks. We start with methods in the spirit of Bundy's proof plans for normalisation [5].

We look at the ideas from Fourier-Motzkin's decision procedure [12] (which is essentially the same as the well known implementation of Hodes' decision procedure for Presburger arithmetic [6]). Fourier-Motzkin's algorithm is a decision procedure for rational numbers, but it is also often used (because of its better efficiency) as sound (but incomplete) procedure for the universal fragment of PIA – Presburger Integer Arithmetic (see, for instance, [3]).

3 Building blocks

We use a simple stand-alone PROLOG implementation of a deduction system based on the proof–planning paradigm, but it is simplified as it does not require preconditions and postconditions of methods.

Decision procedures can be implemented as compact, optimised procedures or they can be built from separate methods (some of which can be general-purpose methods, i.e., methods used also within other procedures). The latter approach often leads to additional overhead processing and is thus less efficient. However, it is much more flexible and gives easily understandable algorithms, and hence we use it in our programme.

We use the following sorts of normalisation methods (in the spirit of Bundy's proof plans for normalisations [5]):

Remove is a normalisation method used to eliminate a certain function symbol, predicate symbol or a quantifier from a formula. For instance, we can eliminate a connection $\Rightarrow$ by exhaustive application of the following rewrite rule: $f_1 \Rightarrow f_2 \longrightarrow \neg f_1 \vee f_2$.

Stratify is a normalisation method used to stratify a class of formulae into two (or more) syntactical layers containing just *some* specific predicate symbols, function symbols or connectives. For instance, *stratify* puts a formula into prenex normal form, moves negations inside disjunctions and conjunctions, moves con-

junctions inside disjunctions etc.

Thin is a normalisation method that exhaustively applies thinning rewrite rules, such as elimination of multiple negations: $\neg\neg f \longrightarrow f$ or elimination of multiple unary minus symbols: $--t \longrightarrow t$.

Reduce is a method that reduces the number of occurrences (to at most one) of a certain function symbol, predicate symbol or a connective in a formula. For instance, it reduces the number of symbols $\top$ and $\bot$ in a formula being proved.

Left Association is one of the normalisation methods for reorganisation within a class. If a syntactical class contains only one function symbol and if that function symbol is both binary and associative, then members of this class can be put into left associative form. For instance, we can use this method for left association of addition and multiplication (given the needed rewrite rules).

Poly-form is a method which we will use for putting a formula into polynomial normal form. It uses rewrite rules such as: $i_1 \cdot i_2 \longrightarrow i_3$ where i_1, i_2, i_3 represent numbers and $i_1 \cdot i_2 = i_3$.

Reorder is one of the methods for reorganisation within one syntactical class. If a class contains only one function and if that function is commutative and associative, this method is used to reorder arguments within a term (which is supposed to be in left associative form). We can use it to reorder arguments in a term which is in polynomial normal form or in a formula in disjunctive normal form. This transformation requires an ordering on variables as an additional device.

Collect is a method which we will use to reduce multiple occurrences of some variable in a term.

Isolate is a method which we use to isolate a specific variable in an atomic formula.

The methods described above are general ones. Clearly, some theories may require more specific methods. [4] However, even if all the necessary methods (general or theory-specific) are available, it may still be very challenging to combine them correctly into a required decision procedure.

4 Generating solved examples

We generated a set of solved examples in several stages: we generated a corpus, grouped and ordered the methods, ran brute force search for proofs and chose solved examples.

4.1 Generating corpus

We generated 1000 Presburger arithmetic conjectures by using the stochastic context-free grammar [5] given in Table 1. The probabilities used were chosen *ad-*

[4] For example, in order to learn the Fourier-Motzkin's procedure, we need a method which performs *cross-multiply and add* step [12] (see also §4.3).

[5] A stochastic context–free grammar is a context–free grammar with a stochastic component which attaches a probability to each of the production rules and controls its use.

hoc (a similar stochastic grammar was used in [11]). We believe that choosing different probabilities would give similar final results to the ones we got in this study. For simplicity, we generated only quantifier-free formulae, [6] and then took their universal closure.

#	*Rule*	*Probability*
1.	$\langle \text{formula} \rangle := \langle \text{atomic formula} \rangle$	0.5
2.	$\langle \text{formula} \rangle := (\neg \langle \text{formula} \rangle)$	0.125
3.	$\langle \text{formula} \rangle := (\langle \text{formula} \rangle \vee \langle \text{formula} \rangle)$	0.125
4.	$\langle \text{formula} \rangle := (\langle \text{formula} \rangle \wedge \langle \text{formula} \rangle)$	0.125
5.	$\langle \text{formula} \rangle := (\langle \text{formula} \rangle \Rightarrow \langle \text{formula} \rangle)$	0.125
6.	$\langle \text{atomic formula} \rangle := (\langle \text{term} \rangle = \langle \text{term} \rangle)$	0.20
7.	$\langle \text{atomic formula} \rangle := (\langle \text{term} \rangle < \langle \text{term} \rangle)$	0.20
8.	$\langle \text{atomic formula} \rangle := (\langle \text{term} \rangle \leq \langle \text{term} \rangle)$	0.20
9.	$\langle \text{atomic formula} \rangle := (\langle \text{term} \rangle > \langle \text{term} \rangle)$	0.20
10.	$\langle \text{atomic formula} \rangle := (\langle \text{term} \rangle \geq \langle \text{term} \rangle)$	0.20
11.	$\langle \text{term} \rangle := (\langle \text{term} \rangle + \langle \text{term} \rangle)$	0.20
12.	$\langle \text{term} \rangle := 1$	0.20
13.	$\langle \text{term} \rangle := 0$	0.20
14.	$\langle \text{term} \rangle := var$	0.40
15.	$\langle \text{var} \rangle := x$	0.30
16.	$\langle \text{var} \rangle := y$	0.25
17.	$\langle \text{var} \rangle := z$	0.20
18.	$\langle \text{var} \rangle := u$	0.15
19.	$\langle \text{var} \rangle := v$	0.10

Table 1
A stochastic grammar for the quantifier-free fragment of Presburger arithmetic.

4.2 Search for proofs

We implemented (in PROLOG) a simple mechanism for brute-force search for proofs of the given conjectures. The mechanism works as follows:

- if the current formula is equal to $\top$ or $\bot$, then stop the search;

[6] Note that closed formulae without redundant quantifiers cannot be generated by a context-free grammar. However, this restriction is not critical. Namely, most quantifier elimination procedures (including the Fourier-Motzkin's procedure) eliminate universal quantifiers by reducing them to existential quantifiers. So, the learning process would be the same if we considered full Presburger arithmetic. Moreover, the learnt procedure (presented in §5) is a decision procedure for full Presburger arithmetic.

- if the current list of applied methods exceeds the given limit, then stop the search;
- try to apply one of the available methods to the current formula; if the method changes the current formula, add that method to the list of applied methods and try to prove the obtained (now new current) formula.

If a current formula is transformed to $\top$ or $\bot$, we consider it solved and we call a sequence of applied methods *a proof trace*. We put the limit (100) for the number of applied methods in order to prevent infinite loops in this search. Some of the generated formulae were huge (one of them had 409 functions symbols, predicate symbols and connectives) so we also put a time limit for solving each conjecture. We used the time limit of 1 minute. [7]

4.3 Grouping methods and ordering of methods

On the basis of the generic normalisation methods discussed in §3, we implemented (in PROLOG) a set of arithmetic-specific methods. We also added the method for elimination of an existentially quantified (and isolated) variable based on Fourier-Motzkin's *cross-multiply and add* step [12]. For the sake of simplicity, we grouped some of these methods (in a natural, expected way), yielding the following set of 9 methods (some of them compound):

`M1`: remove $\Rightarrow$

`M2`: remove $\neq, >, <, \geq$

`M3`: adjust the innermost quantifier (transforms $\forall x F$ to $\neg\exists x \neg F$)

`M4`: stratify $\neg$s beneath $\vee$s and $\wedge$s; thin $\neg$, remove $\neg$

`M5`: delete the innermost redundant quantifier (*cross-multiply and add* step)

`M6`: isolate the innermost variable (provided it is isolated in each atomic formula)

`M7`: stratify $\cdot$ beneath $+$, left-assoc $\cdot$, left-assoc $+$, poly-form

`M8`: stratify $\wedge$s beneath $\vee$s and eliminate the innermost variable

`M9`: reduce $\top$ and $\bot$

Despite having only 9 methods after grouping, a simple depth first search over them does not always produce proofs, because 9 methods still give a large search space [8] and, more importantly, some rules cancel each other out, which can lead to non-termination. Namely, most of the available methods consist of sets of rewrite rules. Even though each set of these sets of rewrite rules is terminating (but not always confluent), the union of sets is not necessarily terminating. Therefore, our set of methods is not terminating. Hence, in order to simplify and direct search, we also had to change the ordering of methods.

The two strategies just described, i.e., grouping and ordering, involve some

[7] All modules were implemented in SWI Prolog; experiments were ran on a 64Mb PC 466Mhz. All source files are available upon request from the authors.

[8] The situation is even worse if we consider low level inference rules, rather than higher level methods (since the proofs would be much longer, and the search space would be much larger).

human knowledge based on experiments in this context, and present a control information for search for proofs.

Methods are tried on given goals in the following order: `M1`, `M2`, `M3`, `M4`, `M5`, `M6`, `M7`, `M8`, `M9`. This ordering is *ad-hoc* and in our experiments we tried several orderings. We chose this as the most appropriate one. Notice that the ordering and grouping phase is not expected to provide the termination argument for the learnt procedure. It can be viewed as a heuristic which directs and improves the brute force search. Moreover, ordering and grouping can be helpful when considering the properties (such as termination and completeness) of the generated procedure (see §5).

4.4 Running brute force search and choosing examples

We ran the described search engine on the set of 1000 generated conjectures/ examples. 76.8% of conjectures were solved (proved or disproved) by this engine; results are given in Table 2. Table 2 also shows how the percentage of solved examples decreases as the number of variables increases. This is reasonable as the search space is rather big and the brute-force search is practically lost on very complex conjectures.

# of variables	0	1	2	3	4	5	total
total	121	340	249	118	77	95	1000
solved	121	301	189	77	45	35	768
% solved	100	88.5	75.9	65.2	58.4	36.8	76.8
longest trace	5	10	15	18	23	30	N/A
# of examples with longest traces	6	8	10	4	5	2	35

Table 2
Results of the brute force method

Having 768 solved examples, we needed to choose the subset of examples which would be used in the learning process (well-chosen examples are essential for this phase of the programme). Good examples are demonstrative examples, i.e., the ones that involve as many methods as possible that should be in the decision procedure that we are learning. But these methods should be used in a concise way in good examples. The search for a proof (given our set and ordering of methods) stops as soon it reaches $\top$ or $\bot$. Thus, the available proofs are the shortest ones that the brute force engine can find. Amongst such proofs of different conjectures, we select as the most illustrative and descriptive proofs the longest ones. Namely, in some cases some methods (that form some parts of the procedure we are learning) leave certain formulae under consideration unchanged, but in other cases they transform (rewrite) them. So, such methods must be considered in order for the system to learn a (general) decision procedure. To learn such pieces of our sought procedure it was sensible to choose examples that use as many of the relevant meth-

ods as possible (i.e., examples that are the most difficult and demanding, and not trivial or easy ones). In other words, in a sense we choose the longest amongst the shortest proofs.

Since the number of variables has a critical role in proving Presburger arithmetic conjectures (the same holds for almost all theories), we separated all solved examples into groups according to the number of variables. We considered formulae with 0, 1, 2, 3, 4 and 5 variables. From each group we selected the longest proof traces (see Table 2).

Within the groups of formulae with 0, 1, and 2 variables all conjectures with the longest proof traces had the same traces (respectively):

```
[M1, M2, M4, M7, M9]
[M1, M2, M3, M4, M6, M8, M5, M4, M7, M9]
[M1, M2, M3, M4, M6, M8, M5, M3, M4, M6, M8, M5, M4, M7, M9]
```

Within the groups of formulae with 3 and 4 variables there were 4 and 5 conjectures with the longest proof traces, but these traces were not equal (within each respective group). Since it is not clear which amongst these are the most descriptive ones, we did not use them for learning. [9] Within the group of formulae with 5 variables there were 2 conjectures with the (same) longest proof trace. Finally, we took the longest traces for formulae with 0, 1 and 2 variables and put them into the learning mechanism.

5 Learning and generating supermethods

From the given sequences, the learning mechanism (described in §2.1) learnt the following general pattern: [10]

$$[M1, M2, [M3, M4, M6, M8, M5]^*, M4, M7, M9].$$

We notice that in each run of the loop ($[M3, M4, M6, M8, M5]^*$), one quantifier is eliminated. Since their number is finite in any conjecture, this process eventually terminates. Provided that all the used primitive methods are sound, the generated supermethod is also sound. Provided the methods are complete, then each conjecture is transformed by the above supermethod to $\bot$ or $\top$, and hence, the learnt procedure is a decision procedure for PRA. Although our proposed programme does not provide a guarantee about the properties of a learnt procedure (such as termination, soundness and completeness), often these properties can be easily proved (as we can see in the above informal discussion).

[9] Namely, considering a possibly very complex procedure, it is not likely that within 1000 formulae we will have conjectures with 3, 4, 5,... variables whose proofs contain all the needed steps of the procedure in all iterations. Larger corpus would perhaps contain such conjectures (but then we may want to consider more variables, so the problem remains).

[10] As expected, it turns out that if examples with 5 variables were used for learning as well, then this learnt pattern would still be the same.

6 Automatic programming for learnt methods

We implemented (in PROLOG) a system for automatic generation of PROLOG predicates on the basis of sequences provided from the learning mechanism. The system supports all constructions that the LEARNΩMATIC system can make (see §2.1), and can generate corresponding PROLOG code. Given the sequence $[M1, M2, [M3, M4, M6, M8, M5]^*, M4, M7, M9]$, our system generated the following PROLOG code (which we finally applied to the original set of conjectures):

```
pa(Fa,FF):-
method('M1',Fa,Fb),
method('M2',Fb,Fc),
pb(Fc,Fd),
method('M4',Fd,Fe),
method('M7',Fe,Ff),
method('M9',Ff,FF).

pb(Fa,FF) :-
method('M3',Fa,Fb),
method('M4',Fb,Fc),
method('M6',Fc,Fd),
method('M8',Fd,Fe),
method('M5',Fe,Ff),
pb(Ff,FF),!.
pb(F,F).
```

7 Evaluation

Given the learnt method and the generated PROLOG program, we ran it on the original set of 1000 generated conjectures. While the brute force method solved 768 conjectures (within the given time limit), the learnt decision procedure solved 991 conjectures (see Table 3). Nine unsolved examples had hundreds of symbols and the method had not failed to solve them, but exceeded the time limit. For each conjecture solved by the brute force search, we measured the speed-up when using the newly generated procedure (see Table 3). The overall speed-up average was 1.0619. However, the main gain from the learnt procedure is in 223 conjectures that were not solved at all by the brute force method. We can see in Table 3 that the speed-up increases as the number of variables increases. The speed-up for 5-variable case would probably be higher if we used a higher time limit.

8 Related work

The work presented in this paper uses the learning mechanism of LEARNΩMATIC, which is related to the least general generalisation, and to some more recent work on learning regular expressions, grammar inference and sequence learning [13]. For details, see [9].

Our work is related to ideas from [5]. In Bundy's programme a decision pro-

# of variables	0	1	2	3	4	5	total
total	121	340	249	118	77	95	1000
solved	121	340	249	118	77	86	991
% solved	100	100	100	100	100	90.5	99.1
speed-up	1	1.0001	1.0287	1.0990	1.4394	1.4181	1.0619

Table 3
Results of the learnt method

cedure should be synthesised given all needed rewrite rules and several general patterns for normalising formulae. Considering automatic derivation of decision procedures our work is also related to work presented in [1] which is aimed at deriving decision procedures using superposition.

9 Conclusions and future work

Our conclusion is that learning decision procedures is not an easy task (even when all the needed primitive methods are given), but it is possible. It is difficult to have the process of learning a complex decision procedure fully automated, so at some stages human interaction and human help is needed. We presented a methodology consisting of a number of steps, techniques and ideas (including a mechanism for generating a corpus of conjectures, a controlled brute force search, strategies for choosing examples, learning mechanism, and the system for automatic programming based on the learnt sequences). Automation in this field is important as it can prevent human flaws in analysing decision procedures or in implementing them. We believe that this methodology (and learning decision procedures in general) can be useful, especially for new or user defined theories. Here are some of the main lessons we learnt during the development of the proposed programme:

- Despite the fact that the implementation of decision procedures based on autonomous, independent methods is less efficient, we find that this approach is flexible and suitable for both analysing and synthesising decision procedures.
- Given a set of methods sufficient to solve any conjecture of a given theory, it is still not a trivial task to build a decision procedure for that theory. The brute force search can solve a number of conjectures, but it is difficult to make a brute force search complete, efficient and terminating (even when all the building blocks are terminating).
- Even if the idea of the required procedure is known and all the necessary building blocks are available, it may still be a non-trivial task to correctly implement the procedure. Automatic assistance in this can be very important.
- In order to make a brute force search more efficient, it is useful to provide some sort of control information. We used grouping and ordering of methods (where it was sensible to do so). This task requires human assistance.

- Having a number of solved examples, it is essential to make a good selection of examples to be used in the learning process. Our strategy was the following: we selected the longest proofs among the shortest proofs found by the brute force search. The rationale is that the most demanding conjectures are the most illustrative ones for learning.
- Provided that we have good examples and a choice of good methods, the learning mechanism can learn a decision procedure from just a few example proofs.
- A system can be made which for a given learnt proof sequence generates a corresponding implementation.
- The learnt method outperforms the brute force search both in the number of conjectures solved and in the CPU time spent.
- We believe that the methodology presented in this study is very well suited to the proof planning paradigm (or its simplified version, as described here), and can be applied to other environments as well.

It is difficult to provide a characterisation of theories for which the proposed approach is successful, since some very deep theory-specific knowledge may be required. However, we can give a characterisation of decision procedures which cannot be learnt: the proposed framework cannot learn procedures which cannot be expressed with the language used in LEARNΩMATIC. All other procedures can potentially be learnt. At the moment, LEARNΩMATIC covers a wide range of languages, while further extensions are under consideration. Learning procedures expressed in another language would require that we replace in our framework LEARNΩMATIC's learning mechanism with another one that uses the desired language, but the other modules of our framework (e.g., generating examples, automatic generation of code from the learnt pattern) can remain unchanged. We also plan to extend the learning approach and the realm of covered languages so that the mechanism could learn recursive methods, which would enable automatic learning of a new range of decision procedures.

Another limitation of our proposed programme is that it may require non-trivial human assistance (e.g., in ordering and grouping). We plan to further develop our methodology and to try to automate (at least to some extent) the steps which now need human interaction.

A comparison between a direct implementation of the decision procedure and a learnt decision procedure would be interesting for further work. But this is out of the scope of the present paper, as we are interested in a larger picture of discovering new decision procedures, rather than in efficient implementations of the existing ones. Mechanised learning of existing decision procedures is an important step towards mechanised learning and discovery of decision procedures. In this sense, the work presented in this paper is an encouraging preliminary step towards discovery. Our hope is that such a framework will be used as a useful assistant in such a process, and moreover, it will lead to automatic discovery of new decision procedures.

References

[1] A. Armando, S. Ranise, and M. Rusinowitch. Uniform Derivation of Decision Procedures by Superposition. CSL 15, LNCS 2142. Springer, 2001.

[2] C. Benzmüller et al. ΩMEGA: Towards a mathematical assistant. CADE 14, LNCS 1249, Springer, 1997.

[3] R. S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. *Machine Intelligence 11*, 1988.

[4] A. Bundy. The use of explicit plans to guide inductive proofs. CADE 9, LNCS 310, Springer.

[5] A. Bundy. The use of proof plans for normalization. In *Essays in Honor of Woody Bledsoe*, Kluwer, 1991.

[6] L. Hodes. Solving problems by formula manipulation in logic and linear inequalities. IJCAI 2, William Kaufmann, 1971.

[7] M. Jamnik, M. Kerber, and M. Pollet. Automatic learning in proof planning. ECAI 15, 2002.

[8] M. Jamnik, M. Kerber, and M. Pollet. LEARNΩMATIC: System description. CADE 18, LNCS 2392, Springer, 2002.

[9] M. Jamnik, M. Kerber, M. Pollet, and C. Benzmüller. Automatic learning of proof methods in proof planning. Technical Report CSRP-02-5, School of Computer Science, University of Birmingham, 2002. Submitted to Journal of AI.

[10] Predrag Janičić and Alan Bundy. A general setting for the flexible combining and augmenting decision procedures. *Journal of Automated Reasoning*, 28(3), 2002.

[11] Predrag Janičić, Ian Green, and Alan Bundy. A comparison of decision procedures in Presburger arithmetic. LIRA '97, Univ. of Novi Sad, 1997.

[12] J.-L. Lassez and M.J. Maher. On Fourier's algorithm for linear arithmetic constraints. *Journal of Automated Reasoning*, 9(3), 1992.

[13] Sun, R., Giles, L., eds.: Sequence Learning: Paradigms, Algorithms, and Applications. LNAI 1828, Springer, 2000.

A Decision Procedure for a Sublanguage of Set Theory Involving Monotone, Additive, and Multiplicative Functions [1]

Domenico Cantone [2]

Dipartimento di Matematica e Informatica
Università degli Studi di Catania
95125 Catania, Italy

Jacob T. Schwartz [3]

Courant Institute of Mathematical Sciences
New York University
New York, NY 10012, USA

Calogero G. Zarba [4]

Computer Science Department
Stanford University
Stanford, CA 94305, USA

Abstract

MLSS is a decidable sublanguage of set theory involving the predicates membership, set equality, set inclusion, and the operators union, intersection, set difference, and singleton.

In this paper we extend **MLSS** with constructs for expressing monotonicity, additivity, and multiplicativity properties of set-to-set functions. We prove that the resulting language is decidable by reducing the problem of determining the satisfiability of its sentences to the problem of determining the satisfiability of sentences of **MLSS**.

[1] This research has been partially supported by MURST Grant prot. 2001017741 under project "Ragionamento su aggregati e numeri a supporto della programmazione e relative verifiche".
[2] Email: cantone@dmi.unict.it.
[3] Email: schwartz@cs.nyu.edu.
[4] Email: zarba@theory.stanford.edu.

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Since many mathematical facts can be expressed in set-theoretic terms, it is useful to design and implement a proof system based on the powerful formalism of set theory. However, the expressive power of the full language of set theory comes at the price of undecidability. It is therefore more practical to concentrate on sublanguages of set theory.

Computable set theory [2,4] is that area of mathematics and computer science which studies the decidability properties of sublanguages of set theory. It was initiated in 1980 by the seminal paper of Ferro, Omodeo, and Schwartz [7], who proved the decidability of:

- a *multi-level syllogistic* (**MLS**) involving membership, set equality, set inclusion, union, intersection, and set difference;
- a *multi-level syllogistic with singleton* (**MLSS**) extending **MLS** with the singleton operator.

In this paper we introduce the sublanguage of set theory **MLSSmf** (*multi-level syllogistic with singleton and monotone functions*), which extends **MLSS** with uninterpreted set-to-set function symbols and several constructs for expressing monotonicity, additivity, and multiplicativity properties of set-to-set functions.

We prove that **MLSSmf** is decidable by providing a reduction algorithm which maps each sentence of **MLSSmf** into an equisatisfiable sentence of **MLSS**. Then the decidability of **MLSSmf** will follow from the decidability of **MLSS**.

Our reduction algorithm is an *augmentation* method, that is, a method that uses as a black box a decision procedure for a language L in order to obtain a decision procedure for a nontrivial extension L' of L. Other augmentation methods for set-theoretic languages can be found in [9,10].

The literature abounds with decidability results for extensions of **MLSS** involving uninterpreted function symbols. Ferro, Omodeo, and Schwartz [8] and Beckert and Hartmer [1] proved the decidability of an extension of **MLSS** with uninterpreted function symbols, but with no monotonicity, additivity, and multiplicativity constructs. Cantone and Zarba [6] proved the decidability of a sublanguage of set theory with urelements [5] and stratified sets involving monotonicity constructs, but no additivity and multiplicativity constructs.

A preliminary version of this paper, not addressing multiplicativity constructs, can be found in [5].

This paper is organized as follows. In Section 2 we formally define the syntax and semantics of the languages **MLS**, **MLSS** and **MLSSmf**, and we give other useful notions which will be needed subsequently. In Section 3 we present our reduction algorithm for mapping sentences of **MLSSmf** into equisatisfiable sentences of **MLSS**. In Section 4 we prove that our reduction algorithm is correct and we assess its complexity. Finally, in Section 5 we draw conclusions from our work.

[5] Urelements (also known as atoms or individuals) are objects which contain no elements but are distinct from the empty set. "Ur" is a German prefix meaning "primitive" or "original".

2 Preliminaries

2.1 Multi-level syllogistic

MLS (*multi-level syllogistic*) is the unquantified set-theoretic language containing:

- an enumerable collection of variables;
- the constant $\emptyset$ (*empty set*);
- the operators $\cup$ (*union*), $\cap$ (*intersection*), and $\setminus$ (*set difference*);
- the predicates $\in$ (*membership*), $=$ (*set equality*), and $\subseteq$ (*set inclusion*);
- the propositional connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$.

The semantics of **MLS** is based upon the standard von Neumann hierarchy of sets $\mathcal{V}$ defined by:

$$\begin{aligned} \mathcal{V}_0 &= \emptyset\,, \\ \mathcal{V}_{\alpha+1} &= \mathcal{P}(\mathcal{V}_\alpha)\,, && \text{for each ordinal } \alpha\,, \\ \mathcal{V}_\lambda &= \bigcup_{\mu<\lambda} \mathcal{V}_\mu\,, && \text{for each limit ordinal } \lambda\,, \\ \mathcal{V} &= \bigcup_{\alpha\in\mathcal{O}} \mathcal{V}_\alpha\,, \end{aligned}$$

where $\mathcal{P}$ denotes the power-set operator, and $\mathcal{O}$ is the class of all ordinals.

An *assignment* $\mathcal{A}$ over a collection of variables V is any map from V into the von Neumann hierarchy of sets $\mathcal{V}$. Given an **MLS**-formula φ over a collection V of variables, and an assignment $\mathcal{A}$ over V, we denote with $\varphi^{\mathcal{A}}$ the truth-value of φ obtained by interpreting each variable $x \in V$ with the set $x^{\mathcal{A}}$, and interpreting the set symbols and logical connectives according to their standard meaning. A *model* of an **MLS**-formula φ is an assignment $\mathcal{A}$ such that $\varphi^{\mathcal{A}}$ is true. An **MLS**-formula φ is *satisfiable* if it has a model.

The satisfiability problem for **MLS** is the problem of determining whether or not an **MLS**-formula φ is satisfiable. This problem is decidable [7].

2.2 Extensions of **MLS**

MLSS (*multi-level syllogistic with singleton*) is the unquantified set-theoretic language extending **MLS** with the singleton operator $\{\cdot\}$. The semantics of **MLSS** is defined similarly to the semantics of **MLS**. The satisfiability problem for **MLSS** is decidable [7].

In this paper we focus on the satisfiability problem for the unquantified set-theoretic language **MLSSmf** (*multi-level syllogistic with singleton and monotone functions*), which extends **MLSS** with an enumerable collection of unary set-to-set function symbols and the predicates inc, dec, add, mul, and $\preceq$.

The semantics of **MLSSmf** is defined similarly to the semantics of **MLS** and **MLSS**, with the only difference that if $\mathcal{A}$ is an (**MLSSmf**-)assignment and f is

a function symbol then $f^{\mathcal{A}}$ is a class function from $\mathcal{V}$ into $\mathcal{V}$. Moreover, for any assignment $\mathcal{A}$ we agree that:

- $inc(f)$ holds in $\mathcal{A}$ if and only if $f^{\mathcal{A}}$ is *increasing*, that is, $s \subseteq t \rightarrow f^{\mathcal{A}}(s) \subseteq f^{\mathcal{A}}(t)$, for all sets s, t;
- $dec(f)$ holds in $\mathcal{A}$ if and only if $f^{\mathcal{A}}$ is *decreasing*, that is, $s \subseteq t \rightarrow f^{\mathcal{A}}(t) \subseteq f^{\mathcal{A}}(s)$, for all sets s, t;
- $add(f)$ holds in $\mathcal{A}$ if and only if $f^{\mathcal{A}}$ is *additive*, that is, $f^{\mathcal{A}}(s \cup t) = f^{\mathcal{A}}(s) \cup f^{\mathcal{A}}(t)$, for all sets s, t;
- $mul(f)$ holds in $\mathcal{A}$ if and only if $f^{\mathcal{A}}$ is *multiplicative*, that is, $f^{\mathcal{A}}(s \cap t) = f^{\mathcal{A}}(s) \cap f^{\mathcal{A}}(t)$, for all sets s, t;
- $f \preceq g$ holds in $\mathcal{A}$ if and only if $f^{\mathcal{A}}(s) \subseteq g^{\mathcal{A}}(s)$, for every set s.

2.3 Normalized literals

In order to simplify details, we will often consider conjunctions of *normalized* **MLSSmf**-literals of the form:

$$\begin{array}{llll} x = y\,, & x \neq y\,, & x = y \cup z\,, & x = y \setminus z\,, \\ x = \{y\}\,, & x = f(y)\,, & inc(f)\,, & dec(f)\,, \\ add(f)\,, & mul(f)\,, & f \preceq g\,. & \end{array} \tag{1}$$

Let φ be an **MLSSmf**-formula. By suitably introducing new variables, it is possible to convert φ into an equisatisfiable formula $\psi = \psi_1 \vee \ldots \vee \psi_k$ in disjunctive normal form, where each ψ_i is a conjunction of normalized **MLSSmf**-literals of the form (1). Thus, we have the following result.

Lemma 2.1 *The satisfiability problem for* **MLSSmf***-formulae is equivalent to the satisfiability problem of conjunctions of normalized* **MLSSmf***-literals of the form* (1).

Similar results as the one in Lemma 2.1 also hold for the languages **MLS** and **MLSS**, although with different groups of normalized literals.

Lemma 2.2 *The satisfiability problem for* **MLS***-formulae is equivalent to the satisfiability problem of conjunctions of normalized* **MLS***-literals of the form:*

$$x = y\,, \qquad x \neq y\,, \qquad x = y \cup z\,, \qquad x = y \setminus z\,, \qquad x \in y\,. \tag{2}$$

Lemma 2.3 *The satisfiability problem for* **MLSS***-formulae is equivalent to the satisfiability problem of conjunctions of normalized* **MLSS***-literals of the form:*

$$x = y\,, \qquad x \neq y\,, \qquad x = y \cup z\,, \qquad x = y \setminus z\,, \qquad x = \{y\}\,. \tag{3}$$

Unless otherwise specified, in the rest of this paper the word normalized refers to literals of the form (1).

3 The reduction algorithm

Let $\mathcal{C}$ be a conjunction of normalized **MLSSmf**-literals, and denote with $V = \{x_1, \ldots, x_n\}$ and F the collections of variables and function symbols occurring in $\mathcal{C}$, respectively. In this section we describe a reduction algorithm for converting $\mathcal{C}$ into an equisatisfiable conjunction $\mathcal{C}^*$ of **MLSS**-formulae.

We will use the following notation. Given a set a, $\mathcal{P}^+(a)$ denotes the set $\mathcal{P}(a) \setminus \{\emptyset\}$. Moreover, we denote with ℓ_j the set $\{\alpha \in \mathcal{P}^+(\{1, \ldots, n\}) : j \in \alpha\}$, for $1 \leq j \leq n$.

The reduction algorithm is shown in Figure 1, and consists of three steps. In the first step, we generate new variables whose intuitive meaning is as follows:

- for each $\alpha \in \mathcal{P}^+(\{1, \ldots, n\})$, the new variable v_α is intented to represent the Venn region $\bigcap_{i \in \alpha} x_i \setminus \bigcup_{j \notin \alpha} x_j$;
- for each $\ell \subseteq \mathcal{P}^+(\{1, \ldots, n\})$, the new variable $w_{f,\ell}$ is intended to represent the value of the function f over the set $\bigcup_{\alpha \in \ell} v_\alpha$.

In the second step, we add to $\mathcal{C}$ appropriate **MLSS**-formulae whose purpose is to model the variables v_α and $w_{f,\ell}$ according to their intuitive meaning. In particular, the variables $w_{f,\ell}$ are modeled by noticing that for each $\ell, m \subseteq \mathcal{P}^+(\{1, \ldots, n\})$, if $\bigcup_{\alpha \in \ell} v_\alpha = \bigcup_{\beta \in m} v_\beta$ then $f\left(\bigcup_{\alpha \in \ell} v_\alpha\right) = f\left(\bigcup_{\beta \in m} v_\beta\right)$.

Finally, in the third step we remove from $\mathcal{C}$ all literals involving function symbols. This is done by replacing all literals of the form $x_i = f(x_j)$, $inc(f)$, $dec(f)$, $add(f)$, $mul(f)$, and $f \preceq g$ with **MLSS**-literals involving only the variables x_i and the new variables $w_{f,\ell}$.

We claim that our reduction algorithm is correct. More specifically, we claim that if $\mathcal{C}^*$ is the result of applying to $\mathcal{C}$ our reduction algorithm then:

- the reduction is *sound*, namely, if $\mathcal{C}$ is satisfiable, so is $\mathcal{C}^*$;
- the reduction is *complete*, namely, if $\mathcal{C}^*$ is satisfiable, so is $\mathcal{C}$.

The next section proves that our reduction algorithm is sound and complete, and therefore it yields a decision procedure for **MLSSmf**.

4 Correctness

4.1 Soundness

Let $\mathcal{C}$ be a satisfiable conjunction of normalized **MLSSmf**-literals, and let $\mathcal{C}^*$ be the result of applying to $\mathcal{C}$ the reduction algorithm in Figure 1. The key idea of the soundness proof is that, given a model $\mathcal{A}$ of $\mathcal{C}$, a model $\mathcal{B}$ of $\mathcal{C}^*$ can be constructed in the most natural way if we remember the intuitive meaning of the variables v_α and $w_{f,\ell}$.

Lemma 4.1 (Soundness) *Let $\mathcal{C}$ be a conjunction of normalized* **MLSSmf***-literals, and let $\mathcal{C}^*$ be the result of applying to $\mathcal{C}$ the reduction algorithm in Figure 1. Then if $\mathcal{C}$ is satisfiable, so is $\mathcal{C}^*$.*

Reduction algorithm

Input: a conjunction $\mathcal{C}$ of normalized **MLSSmf**-literals.

Output: a conjunction $\mathcal{C}^*$ of **MLSS**-formulae.

Notation:

- $V = \{x_1, \ldots, x_n\}$ is the collection of variables occurring in $\mathcal{C}$;
- F is the collection of function symbols occurring in $\mathcal{C}$;
- $\mathcal{P}^+(a) = \mathcal{P}(a) \setminus \{\emptyset\}$, for each set a;
- ℓ_j stands for the set $\{\alpha \in \mathcal{P}^+(\{1, \ldots, n\}) : j \in \alpha\}$, for $1 \leq j \leq n$.

Step 1. Generate the following new variables:

$$\begin{array}{ll} v_\alpha\,, & \text{for each } \alpha \in \mathcal{P}^+(\{1,\ldots,n\})\,, \\ w_{f,\ell}\,, & \text{for each } f \in F \text{ and } \ell \subseteq \mathcal{P}^+(\{1,\ldots,n\})\,. \end{array}$$

Step2. Add to $\mathcal{C}$ the following **MLSS**-formulae:

$$v_\alpha = \bigcup_{i\in\alpha} x_i \setminus \bigcap_{j\notin\alpha} x_j\,, \qquad \text{for each } \alpha \in \mathcal{P}^+(\{1,\ldots,n\})\,,$$

and

$$\bigcup_{\alpha\in\ell} v_\alpha = \bigcup_{\beta\in m} v_\beta \rightarrow w_{f,\ell} = w_{f,m}\,, \quad \text{for each } f \in F \text{ and } \ell, m \subseteq \mathcal{P}^+(\{1,\ldots,n\})\,.$$

Step 3. Replace literals in $\mathcal{C}$ containing function symbols with **MLSS**-formulae as follows:

$$\begin{array}{lcl} x_i = f(x_j) & \Longrightarrow & x_i = w_{f,\ell_j} \\ inc(f) & \Longrightarrow & \bigwedge_{\ell\subseteq m} (w_{f,\ell} \subseteq w_{f,m}) \\ dec(f) & \Longrightarrow & \bigwedge_{\ell\subseteq m} (w_{f,m} \subseteq w_{f,\ell}) \\ add(f) & \Longrightarrow & \bigwedge_{\ell,m} (w_{f,\ell\cup m} = w_{f,\ell} \cup w_{f,m}) \\ mul(f) & \Longrightarrow & \bigwedge_{\ell,m} (w_{f,\ell\cap m} = w_{f,\ell} \cap w_{f,m}) \\ f \preceq g & \Longrightarrow & \bigwedge_{\ell} (w_{f,\ell} \subseteq w_{g,\ell}) \end{array}$$

Fig. 1. The reduction algorithm.

Proof. Let $\mathcal{A}$ be a model of $\mathcal{C}$, and denote with $V = \{x_1, \ldots, x_n\}$ and F the collections of variables and function symbols occurring in $\mathcal{C}$, respectively.

It is easy to verify that the assignment $\mathcal{B}$ defined by:

$$\begin{aligned} x_i^{\mathcal{B}} &= x_i^{\mathcal{A}}\,, && \text{for each } i \in \{1, \ldots, n\}\,, \\ v_\alpha^{\mathcal{B}} &= \bigcap_{i \in \alpha} x_i^{\mathcal{A}} \setminus \bigcup_{j \notin \alpha} x_j^{\mathcal{A}}\,, && \text{for each } \alpha \in \mathcal{P}^+(\{1, \ldots, n\})\,, \\ w_{f,\ell}^{\mathcal{B}} &= f^{\mathcal{A}}\left(\bigcup_{\alpha \in \ell} v_\alpha^{\mathcal{B}}\right)\,, && \text{for each } f \in F \text{ and } \ell \subseteq \mathcal{P}^+(\{1, \ldots, n\}) \end{aligned}$$

is a model of $\mathcal{C}^*$. □

4.2 Completeness

Let $\mathcal{C}$ be a conjunction of normalized **MLSSmf**-literals. As before, let us denote with $V = \{x_1, \ldots, x_n\}$ and F the collections of variables and function symbols occurring in $\mathcal{C}$, respectively. Also, let $\mathcal{C}^*$ be the result of applying to $\mathcal{C}$ the reduction algorithm in Figure 1. To show the completeness of our reduction algorithm, we need to prove that if $\mathcal{C}^*$ is satisfiable, so is $\mathcal{C}$.

To do so, let $\mathcal{B}$ be a model of $\mathcal{C}^*$, and let us start to define an assignment $\mathcal{M}$ over the variables and function symbols in $\mathcal{C}$ by letting

$$x_i^{\mathcal{M}} = x_i^{\mathcal{B}}\,, \qquad \text{for each } i = 1, \ldots, n\,.$$

In order to define $\mathcal{M}$ over the function symbols in F, let us recall that the intuitive meaning of a variable of the form $w_{f,\ell}$ is to represent the expression $f(\bigcup_{\alpha \in \ell} v_\alpha)$. Thus, our definition of $f^{\mathcal{M}}$ should satisfy the property that $f^{\mathcal{M}}(\bigcup_{\alpha \in \ell} v_\alpha^{\mathcal{B}}) = w_{f,\ell}^{\mathcal{B}}$, for every $\ell \subseteq \mathcal{P}^+(\{1, \ldots, n\})$. But how do we define $f^{\mathcal{M}}(a)$ in the more general case in which a is not the union of sets of the form $v_\alpha^{\mathcal{B}}$? The idea is to define opportunely a *discretization function* $\lambda : \mathcal{V} \to \mathcal{P}(\mathcal{P}^+(\{1, \ldots, n\}))$, and then let

$$f^{\mathcal{M}}(a) = w_{f,\lambda(a)}^{\mathcal{B}}\,, \qquad \text{for each } f \in F \text{ and each set } a\,.$$

To achieve completeness, we need a *good* discretization function.

Definition 4.2 Let $\mathcal{C}$ be a conjunction of normalized **MLSSmf**-literals and let $\mathcal{C}^*$ be the result of applying to $\mathcal{C}$ the reduction algorithm in Figure 1. A discretization function $\lambda : \mathcal{V} \to \mathcal{P}(\mathcal{P}^+(\{1, \ldots, n\}))$ is GOOD with respect to a model $\mathcal{B}$ of $\mathcal{C}^*$ if the following conditions hold:

(A) λ is increasing;

(B) λ is additive;

(C) λ is multiplicative;

(D) if $a = \bigcup_{\alpha \in \ell} v_\alpha^{\mathcal{B}}$ then $a = \bigcup_{\alpha \in \lambda(a)} v_\alpha^{\mathcal{B}}$, for each $\ell \subseteq \mathcal{P}^+(\{1, \ldots, n\})$.

Lemma 4.3 *Let $\mathcal{C}$ be a conjunction of normalized* **MLSSmf***-literals, let $\mathcal{C}^*$ be the result of applying to $\mathcal{C}$ the reduction algorithm in Figure 1, and let $\mathcal{B}$ be a model of $\mathcal{C}^*$. Assume that there exists a discretization function $\lambda : \mathcal{V} \to \mathcal{P}(\mathcal{P}^+(\{1,\ldots,n\}))$ which is good with respect to $\mathcal{B}$.*

Then $\mathcal{C}$ is satisfiable.

Proof. As before, let $V = \{x_1,\ldots,x_n\}$ and F be the collections of variables and function symbols occurring in $\mathcal{C}$, respectively.

It is an easy matter to verify that the assignment $\mathcal{M}$ defined by:

$$\begin{aligned} x_i^{\mathcal{M}} &= x_i^{\mathcal{B}}\,, && \text{for each } i = 1,\ldots,n\,,\\ f^{\mathcal{M}}(a) &= w^{\mathcal{B}}_{f,\lambda(a)}\,, && \text{for each } f \in F \text{ and each set } a\,, \end{aligned}$$

is a model of $\mathcal{C}$. □

Lemma 4.3 shows that the existence of good discretization functions is enough to ensure the completeness of our reduction algorithm. But how do we define good discretization functions?

As a first attempt, given an arbitrary model $\mathcal{B}$ of $\mathcal{C}^*$, let us put

$$\lambda^+_{\mathcal{B}}(a) = \{\alpha \in \mathcal{P}^+(\{1,\ldots,n\}) : v^{\mathcal{B}}_\alpha \cap a \neq \emptyset\}\,, \qquad \text{for each set } a\,.$$

It is easy to see that $\lambda^+_{\mathcal{B}}$ satisfies properties (A), (B), and (D) of Definition 4.2. However, in general $\lambda^+_{\mathcal{B}}$ is not multiplicative. As a counter-example, assume that there exist two disjoint sets a, b and some $\alpha \subseteq \mathcal{P}^+(\{1,\ldots,n\})$ such that $a \cap v^{\mathcal{B}}_\alpha \neq \emptyset$ and $b \cap v^{\mathcal{B}}_\alpha \neq \emptyset$. Then $\alpha \in \lambda^+_{\mathcal{B}}(a) \cap \lambda^+_{\mathcal{B}}(b)$ but $\lambda^+_{\mathcal{B}}(a \cap b) = \emptyset$.

Note that in the proof of Lemma 4.3 the hypothesis that $\lambda^+_{\mathcal{B}}$ is multiplicative is used only to show that the literals of the form $mul(f)$ in $\mathcal{C}$ are satisfied by $\mathcal{M}$. Therefore, if we define **MLSSmf**$^+$ to be the language obtained from **MLSSmf** by removing the symbol mul, we get the following partial result.

Lemma 4.4 *Let $\mathcal{C}$ be a conjunction of normalized* **MLSSmf**$^+$*-literals and let $\mathcal{C}^*$ be the result of applying to $\mathcal{C}$ the reduction algorithm in Figure 1. Then if $\mathcal{C}^*$ is satisfiable, so is $\mathcal{C}$.*

Combining Lemma 4.1 and Lemma 4.4 we obtain immediately the decidability of **MLSSmf**$^+$.

Theorem 4.5 *The satisfiability problem for* **MLSSmf**$^+$ *is decidable.*

As a second attempt to find a good discretization function, let us put

$$\lambda^\times_{\mathcal{B}}(a) = \{\alpha \in \mathcal{P}^+(\{1,\ldots,n\} : \emptyset \neq v^{\mathcal{B}}_\alpha \subseteq a\}\,, \qquad \text{for each set } a\,.$$

It is easy to see that $\lambda^\times_{\mathcal{B}}$ satisfies properties (A), (C), and (D) of Definition 4.2. However, in general $\lambda^\times_{\mathcal{B}}$ is not additive. As a counter-example, assume that there exist two sets a, b and some $\alpha \subseteq \mathcal{P}^+(\{1,\ldots,n\})$ such that $v^{\mathcal{B}}_\alpha \subseteq a \cup b$, $v^{\mathcal{B}}_\alpha \not\subseteq a$, and $v^{\mathcal{B}}_\alpha \not\subseteq b$. Then $\alpha \in \lambda^\times_{\mathcal{B}}(a \cup b)$ but $\alpha \notin \lambda^\times_{\mathcal{B}}(a) \cup \lambda^\times_{\mathcal{B}}(b)$.

Note that in the proof of Lemma 4.3 the hypothesis that $\lambda^{\times}_{\mathcal{B}}$ is additive is used only to show that the literals of the form $add(f)$ in $\mathcal{C}$ are satisfied by $\mathcal{M}$. Therefore, if we define **MLSSmf**$^{\times}$ to be the language obtained from **MLSSmf** by removing the symbol add, we get the following partial result.

Lemma 4.6 *Let $\mathcal{C}$ be a conjunction of normalized* **MLSSmf**$^{\times}$*-literals and let $\mathcal{C}^*$ be the result of applying to $\mathcal{C}$ the reduction algorithm in Figure 1. Then if $\mathcal{C}^*$ is satisfiable, so is $\mathcal{C}$.*

Combining Lemma 4.1 and Lemma 4.6 we obtain at once the decidability of **MLSSmf**$^{\times}$.

Theorem 4.7 *The satisfiability problem for* **MLSSmf**$^{\times}$ *is decidable.*

So far, it appears as neither $\lambda^{+}_{\mathcal{B}}$ nor $\lambda^{\times}_{\mathcal{B}}$ are good discretization functions. However, assume that we have a model $\mathcal{B}$ of $\mathcal{C}^*$ such that:

$$|v^{\mathcal{B}}_{\alpha}| \leq 1\,, \qquad \text{for each } \alpha \in \mathcal{P}^{+}(\{1,\ldots,n\})\,. \tag{4}$$

Then it is easy to see that in this case $\lambda^{+}_{\mathcal{B}}$ and $\lambda^{\times}_{\mathcal{B}}$ coincide, and therefore they are both additive and multiplicative. Thus, both $\lambda^{+}_{\mathcal{B}}$ and $\lambda^{\times}_{\mathcal{B}}$ are good discretization functions with respect to any model $\mathcal{B}$ of $\mathcal{C}^*$ satisfying (4).

But do models of $\mathcal{C}^*$ satisfying (4) exist? The following lemma gives an affirmative answer to this question. [6]

Lemma 4.8 *Let $\mathcal{C}$ be a conjunction of normalized* **MLSSmf***-literals, and let $\mathcal{C}^*$ be the result of applying to $\mathcal{C}$ the reduction algorithm in Figure 1. Assume also that $\mathcal{C}^*$ is satisfiable. Then there exists a model $\mathcal{B}$ of $\mathcal{C}^*$ such that $|v^{\mathcal{B}}_{\alpha}| \leq 1$, for each $\alpha \in \mathcal{P}^{+}(\{1,\ldots,n\})$.*

Combining Lemma 4.8 with Lemma 4.3 we can finally obtain the completeness of our reduction algorithm, using either $\lambda^{+}_{\mathcal{B}}$ or $\lambda^{\times}_{\mathcal{B}}$ as a good discretization function with respect to a model $\mathcal{B}$ of $\mathcal{C}^*$ satisfying (4).

Lemma 4.9 (Completeness) *Let $\mathcal{C}$ be a conjunction of normalized* **MLSSmf***-literals and let $\mathcal{C}^*$ be the result of applying to $\mathcal{C}$ the reduction algorithm in Figure 1. Then if $\mathcal{C}^*$ is satisfiable, so is $\mathcal{C}$.*

Combining Lemma 4.1 and Lemma 4.9, we obtain the decidability of **MLSSmf**.

Theorem 4.10 (Decidability) *The satisfiability problem for* **MLSSmf** *is decidable.*

4.3 Complexity issues

Let $\mathcal{C}$ be a conjunction of normalized **MLSSmf**-literals containing n distinct variables and m distinct function symbols. It turns easily out that the formula $\mathcal{C}^*$,

[6] A proof of Lemma 4.8 will be reported in the extended version of the present paper.

which results by applying to $\mathcal{C}$ the reduction algorithm in Figure 1, involves $\mathcal{O}(2^n)$ variables of type v_α, with $\alpha \in \mathcal{P}^+(\{1,\ldots,n\})$, and $\mathcal{O}(m \cdot 2^{2^n})$ variables of type $w_{f,\ell}$, where f is a function symbol in $\mathcal{C}$ and $\ell \subseteq \mathcal{P}^+(\{1,\ldots,n\})$. Moreover, the collective size of all formulae generated in Step 2 is

$$\mathcal{O}(n \cdot 2^n) \;+\; \mathcal{O}(m \cdot 2^n \cdot 2^{2^{n+1}})\,,$$

and the collective size of all formulae generated in Step 3 is bounded by

$$\mathcal{O}(p \cdot 2^{2^{n+1}})\,,$$

where p is the number of literals in $\mathcal{C}$ of type

$$x = f(y)\,, \qquad inc(f)\,, \qquad dec(f)\,, \qquad add(f)\,, \qquad mul(f)\,, \qquad f \preceq g\,.$$

Thus, if we denote with K the size of $\mathcal{C}$, since $m, n, p \leq K$, we have the following upper bound on the size of $\mathcal{C}^*$:

$$\mathcal{O}(K \cdot 2^K \cdot 2^{2^{K+1}})\,.$$

Finally, to estimate the complexity of our decision procedure, we must take into account that the formula $\mathcal{C}^*$ must then be tested for satisfiability, and it is know that the satisfiability problem for **MLSS** is *NP*-complete [3]. Though the satisfiability test for **MLSS** is quite efficient in practice, it becomes very expensive when run on such large formulae as $\mathcal{C}^*$.

On the other hand, preliminary results show that, in favorable cases, the reduction algorithm in Figure 1 can be "factorized" over a suitable partition of the function symbols present in the input **MLSSmf**-conjunction $\mathcal{C}$. This, roughly speaking, is the case for formulae which do not contain both literals $add(f)$ and $mul(f)$ for the same function f. In such cases, it turns out that $\mathcal{C}$ can be reduced to an equisatisfiable **MLSS**-formula $\mathcal{C}^\times$ having size comparable to that of $\mathcal{C}$, thus making the overall decision process for $\mathcal{C}$ practical.

5 Conclusion

We presented a decision procedure for the set-theoretic sublanguage of set theory **MLSSmf** extending **MLSS** with constructs for expressing monotonicity, additivity, and multiplicativity properties of set-to-set functions. The decision procedure consists of a reduction algorithm which maps each sentence of **MLSSmf** into an equisatisfiable sentence of **MLSS**. Then the decidability of **MLSSmf** follows from the decidability of **MLSS**.

Our work can have applications in an interactive proof environment in which the user helps the system by telling which expressions are monotonic, while our decision procedure performs the tedious combinatoric steps. For instance, when

proving the validity of the formula

$$\{f(x) : x \in a \setminus v\} \subseteq \{f(x) : x \in (a \cup b) \setminus v\}\,, \tag{5}$$

the user can instruct the system with the insight that the function

$$F(u) = \{f(x) : x \in u \setminus v\}$$

is increasing in u. Then, the system would conclude that to prove that (5) is valid, it suffices to prove that

$$inc(F) \;\rightarrow\; F(a) \subseteq F(a \cup b) \tag{6}$$

is valid. Since (6) is an **MLSSmf**-formula, its validity can be automatically proven by our decision procedure.

Future directions of research may involve extensions of our decision procedure to handle other constructs related to set-to-set functions, such as injectivity and surjectivity of functions, as well as a fixed-point operator on monotone functions. Moreover, we are currently working on the identification of convenient syntactic restrictions which allow a speed-up of the reduction process.

References

[1] Beckert, B. and U. Hartmer, *A tableau calculus for quantifier-free set theoretic formulae*, in: H. C. M. de Swart, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, Lecture Notes in Computer Science **1397** (1998), pp. 93–107.

[2] Cantone, D., A. Ferro and E. G. Omodeo, "Computable Set Theory," International Series of Monographs on Computer Science **6**, Clarendon Press, 1989.

[3] Cantone, D., E. G. Omodeo and A. Policriti, *The automation of syllogistic. II. Optimization and complexity issues*, Journal of Automated Reasoning **6** (1990), pp. 173–187.

[4] Cantone, D., E. G. Omodeo and A. Policriti, "Set Theory for Computing. From Decision Procedures to Logic Programming with Sets," Monographs in Computer Science, Springer, 2001.

[5] Cantone, D., J. T. Schwartz and C. G. Zarba, *Decision procedures for fragments of set theory with monotone and additive functions*, in: G. Rossi and B. Jayaraman, editors, *Declarative Programming with Sets*, Technical report 200, Università di Parma, 1999, pp. 1–8.

[6] Cantone, D. and C. G. Zarba, *A tableau calculus for integrating first-order reasoning with elementary set theory reasoning*, in: R. Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, Lecture Notes in Computer Science **1847** (2000), pp. 143–159.

[7] Ferro, A., E. G. Omodeo and J. T. Schwartz, *Decision procedures for elementary sublanguages of set theory. I. Multi-level syllogistic and some extensions*, Communications on Pure and Applied Mathematics **33** (1980), pp. 599–608.

[8] Ferro, A., E. G. Omodeo and J. T. Schwartz, *Decision procedures for some fragments of set theory*, in: W. Bibel and R. A. Kowalski, editors, *5th Conference on Automated Deduction*, Lecture Notes in Computer Science **87** (1980), pp. 88–96.

[9] Zarba, C. G., *Combining multisets with integers*, in: A. Voronkov, editor, *Automated Deduction – CADE-18*, Lecture Notes in Computer Science **2392** (2002), pp. 363–376.

[10] Zarba, C. G., *Combining sets with integers*, in: A. Armando, editor, *Frontiers of Combining Systems*, Lecture Notes in Computer Science **2309** (2002), pp. 103–116.

On leaf permutative theories and occurrence permutation groups

Thierry Boy de la Tour [1] Mnacho Echenim [2]

LEIBNIZ laboratory, IMAG - CNRS
INPG, 46 avenue Félix Viallet F-38031 Grenoble Cedex, France

Abstract

Leaf permutative theories contain variable-permuting equations, so that rewriting a term may lead to an exponential number of terms which are only permutations of it. In [1] Avenhaus and Plaisted propose to represent such sets by *stratified* terms, and to deduce directly with these. Our aim is to use computational group theory to analyse the complexity of the corresponding problems, and hopefully devise better algorithms than [1]. In order to express stratified sets as orbits, we adopt a representation of terms based on *occurrences*, which can conveniently be permuted. An algorithm solving a basic equivalence problem is presented.

1 Introduction

When dealing with an equational theory, it is often the case that *leaf permutative* equations are produced, i.e. equations that are invariant under some permutation of variables. For example, suppose an equational theory E contains the two following equations: $f(x, y, g(z, t)) = f(y, x, g(z, t))$ and $f(x, y, g(z, t)) = f(y, z, g(t, x))$. Then from a clause $C[f(t_1, t_2, g(t_3, t_4))]$ we can deduce any clause $C[f(t_{1^\sigma}, t_{2^\sigma}, g(t_{3^\sigma}, t_{4^\sigma}))]$, where σ is a permutation in the symmetric group $\mathrm{Sym}(4)$, of cardinality $4!$. Consequently, the number of resolvents modulo E can grow exponentially (depending on E). Different methods have been devised to handle this sort of problem, either by designing specialised unification algorithms, or by adding constraints to the theory (see e.g. [8]).

In [1], Avenhaus and Plaisted devise a new way to handle such theories, which intuitively consists in reasoning with a member of an equivalence class of terms instead of the terms themselves, thus avoiding the exponential overhead. More

[1] Email: Thierry.Boy-de-la-Tour@imag.fr
[2] Email: Mnacho.Echenim@imag.fr

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

precisely, any formula is considered modulo its consequences modulo leaf permutative equational theories, which are finite, and are defined in [1] through "stratified" rewriting. Of course, reasoning modulo such equivalence classes requires the modification of basic deduction algorithms such as unification, subsumption or factorisation. Unfortunately, the algorithms in [1] all have exponential time complexity.

It is an essential idea of [1] that leaf permutative equations provide generators for permutation groups. Our aim is to exploit further this idea by using group-theoretic tools (see e.g. [6] or [4]). We first focus on defining a suitable group and action (on group actions see [6], or the simple introduction in [3], section 3) so that these equivalence classes appear as orbits.

But defining an action on terms gives rise to a first difficulty: how do we determine precisely the image of a term by a given permutation? For example, given a commutative function symbol g, how can we identify the basic operation of swapping a and b in the two equivalent terms $g(g(a,b),c)$ and $g(c,g(a,b))$, since $g(a,b)$ occurs at different positions? In [1] a distinctive mark labels each occurrence of g, which allows to keep track of "travelling" positions. This makes the swapping of a and b *relative* to a mark, which is not very convenient to retain a simple group-theoretic framework.

This is why our formalism departs from [1], starting with a simple observation: considering the usual computer representation of the two terms above (e.g. in LISP), they may both contain the *same* pointer to $g(a,b)$. Hence the action we are looking for boils down to a simple kind of pointer manipulations. This justifies the notion of terms developed in section 2, where occurrences are the fundamental objects.

In section 3 we add leaf permutative theories as labels in terms, yielding (our version of) the *stratified terms* of [1]. In this context we give a simple definition of stratified rewriting (section 4) and reach our aim of expressing equivalence classes as orbits. This result is limited to our formalism, which becomes clear in section 5, devoted to the issue of computing the cardinality of these classes. Section 6 presents an algorithm for the equivalence problem on stratified terms, in a rather abstract way that should serve as a schema for further algorithms, and hints to solving the complexity issues.

A word on notations. The operation in permutation groups is the inverse of function composition, i.e. $\sigma\sigma' = \sigma' \circ \sigma$. The exponential notation (e.g. T^π) is usual for group actions, including function application, i.e. if σ is an integer permutation and i an integer, then $i^\sigma = \sigma(i)$. id is the identity function and $\mathrm{I} = \{\mathrm{id}\}$ is the trivial group. Permutations are written in cycle notations and implicitly extended with fix-points, e.g. $3^{(1\ 2)} = 3$.

2 Occurrence terms

Definition 2.1 *A* Σ-term t *is a finite algebra* (O, s, a)*, with* $s : O \to \Sigma$ *and* $a : O \to O^\star$ *such that* $\forall v \in O$*, the length of the word* $a(v)$ *is the arity of* $s(v)$*, and the*

directed graph $(O, \{\langle v, v' \rangle \in O^2 / v' \text{ occurs in } a(v)\})$ *is a tree.*

We call the root *of* t *the root of this tree, and note it* $\mathrm{root}(t)$*. The formula*

$$\forall v, v_1, \ldots, v_n \in O, a(v) = v_1 \ldots v_n \Rightarrow \mathrm{desc}_t(v) = v.\mathrm{desc}_t(v_1) \ldots \mathrm{desc}_t(v_n)$$

defines a unique function from O *to* $O^\star$*, such that all the elements of* O *occur once and only once in* $\mathrm{desc}_t(\mathrm{root}(t))$*, which we note* $\mathrm{desc}(t)$*. The elements of* O *are the* occurrences *of* t*. The subscripts will be dropped if no ambiguities arise. We may use* $\mathrm{desc}(v)$ *to denote the set of occurrences appearing in* $\mathrm{desc}(v)$*; the same holds for other strings, such as* $a(v)$*.*

If $t = (O, s, a)$ *is a* Σ*-term and* $v \in O$*, we note* $t.v = (\mathrm{desc}(v), s', a')$ *the* subterm of t at v*, where* s' *and* a' *are the restrictions of* s *and* a *to* $\mathrm{desc}(v)$*. The reader may check that* $t.v$ *is a* Σ*-term, and that* $\mathrm{root}(t.v) = v$*,* $t.\mathrm{root}(t) = t$*, and* $\forall u \in \mathrm{desc}(v), (t.v).u = t.u$*.*

This definition is closely linked to the way a term could be represented in a computer: an occurrence v can be considered as the address of a structure that contains a label, $s(v)$, and a list of addresses of (i.e. pointers to) other occurrences: $a(v)$.

It should be noted that terms in the usual sense (say, terms-as-strings) have a unique mathematical representation, like integers. This is not the case here, and two different terms may represent the same term-as-string. Strictly speaking, terms-as-strings may be identified with *isomorphism classes* of terms.

Definition 2.2 *For a term* $t = (O, s, a)$ *and a bijection* $\eta : O \to O'$*, we define* $\eta(t) = (O', s', a')$ *by* $\forall v \in O, s'(\eta(v)) = s(v)$ *and* $a'(\eta(v)) = \eta(a(v))$ *(i.e. the function* η *is applied to each letter in* $a(v)$*). Two terms* t *and* t' *are* isomorphic*, noted* $t \simeq t'$ *iff there is an* isomorphism η from t to t'*, noted* $\eta : t \simeq t'$*, that is a bijection such that* $t' = \eta(t)$*.*

For any Σ*-term* t*, with* $r = \mathrm{root}(t)$*, if* $a(r) = v_1 \ldots v_n \neq \varepsilon$ *(the empty string), let* $\mathrm{string}(t) = s(r)(\mathrm{string}(v_1), \ldots, \mathrm{string}(v_n))$*, otherwise* $\mathrm{string}(t) = s(r)$*.*

Isomorphism testing can be performed in linear time; it is easy to see that $t \simeq t'$ iff $\mathrm{string}(t) = \mathrm{string}(t')$. Two isomorphic terms are joined by a unique isomorphism, so that $\eta(t) = \eta'(t)$ iff $\eta = \eta'$.

Example 2.3 *Let* $f(a, g(b, c))$ *be a* Σ*-term in the usual sense (i.e. a term-as-string). This term is composed of two binary functions* f *and* g*, and three constant symbols,* a, b *and* c*. Take* $O = \{0, \ldots, 4\}$ *and define the functions* s *and* a *by:*

	0	1	2	3	4
s	f	a	g	b	c
a	1.2	ε	3.4	ε	ε

Then $t = (O, s, a)$ *is one representation with occurrences of the term,* $\mathrm{desc}(t) = 0.1.2.3.4$*, and* $\mathrm{string}(t) = f(a, g(b, c))$*. Figure 1 is a graphical representation of* t*.*

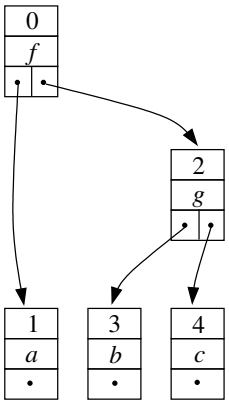


Fig. 1. A representation of $f(a, g(b, c))$

Definition 2.4 *A* Σ-context c *is a* $(\Sigma \uplus \{[]\})$*-term, where* $[]$ *is a constant known as "the hole".*

From a term $t = (O, s, a)$ *and* $H \subseteq O$ *such that* $\forall h, h' \in H, \mathrm{desc}(h) \cap \mathrm{desc}(h') = \emptyset$ *(i.e.* H *is an* antichain *in* T*), we define the context* $t \setminus H$ *by replacing the occurrences in* H *by holes:* $t \setminus H = (O' \cup H, s', a')$*, where* $O' = O \setminus \bigcup_{h \in H} \mathrm{desc}(h)$*, and* $\forall v \in O', s'(v) = s(v), a'(v) = a(v)$*, and* $\forall h \in H, s'(h) = [], a'(h) = \varepsilon$*.*

A term t *is an* instance *of* c *iff there is a* H *such that* $t \setminus H \simeq c$*; this set* H *is of course unique.*

Example 2.5 *It is easy to build a* Σ*-context* c *corresponding to the string* $f([], g([], []))$*. The term* t *of Example 2.3 is an instance of* c*, since with* $H = \{1, 3, 4\}$*, we have* $t \setminus H \simeq c$*.*

3 Stratified terms

The manner of controlling rewriting adopted in [1] is to rewrite so-called stratified terms, where function symbols are labelled with permutative equational theories and unique integers to avoid possible ambiguities. Our focus on occurrences makes this integer superfluous, and the theory is essentially a context and a group permuting the holes of this context.

Definition 3.1 *Let* Σ' *be the set of* $\langle f, c, G\rangle$ *where* $f \in \Sigma$*,* c *is a* Σ*-context such that* $s(\mathrm{root}(c)) = f$*, and* G *is a subgroup of* $\mathrm{Sym}(m)$*, where* m *is the number of holes in* c*. A* stratified term T *is a* $(\Sigma \uplus \Sigma')$*-term such that* $\forall v \in T$*, if* $s(v) \in \Sigma'$*, let* $\langle f, c, G\rangle = s(v)$*, then* $T.v$ *is an instance of* c'*, which is* c *with* $s(\mathrm{root}(c))$ *replaced by* $\langle f, c, G\rangle$*.*

Let um *be the projection from* $\Sigma \uplus \Sigma'$ *onto* Σ *defined by* $\forall \langle f, c, G\rangle \in \Sigma', \mathrm{um}(\langle f, c, G\rangle) = f$*. It can trivially be extended to a projection from* $(\Sigma \uplus \Sigma')$*-terms to* Σ*-terms, also noted* um*.*

Example 3.2 *We consider the leaf permutative theory E axiomatised by the equation $\forall x, y, z, f(x, g(y, z)) = f(y, g(z, x))$. Using the context from Example 2.5, we can write it $c[x, y, z] = c[y, z, x]$. This means that the first hole moves to the third position, the second to the first, the third to the second. So we represent the equation by c and the permutation $\sigma = (1\ 3\ 2)$. The theory E is represented by the symbol $F = \langle f, c, G\rangle$, where G is the group generated*[3] *by σ. By replacing f by F in Figure 1, we obtain a stratified term T. Indeed, $T \setminus H$ is isomorphic to the context c', with $\mathrm{string}(c') = F([], g([], []))$. We have $\mathrm{um}(T) = t$.*

Each symbol in Σ' corresponds to a leaf permutative theory, and may appear at several occurrences in a stratified term. However, each occurrence labelled with a symbol of Σ' does not need the full generality of the theory, and refers only to specific occurrences. Hence the following definition:

Definition 3.3 *Given a stratified term $T = (O, s, a)$ we first define a function H_T from O to $O^\star$: $\forall v \in O$, if $s(v) \in \Sigma$ then $\mathrm{H}_T(v) = \varepsilon$, else let $\langle f, c, G\rangle = s(v)$, and c' be the context obtained from c as in definition 3.1, then $\exists! H \subseteq O$ such that $T \setminus H \simeq c'$, let $v_1, \ldots, v_m$ be the elements of H, given in the order in which they appear in $\mathrm{desc}(T)$, then we let $\mathrm{H}_T(v) = v_1 \ldots v_m$.*

Next we define for any $v \in O$ a function Φ^v_T from $\mathrm{Sym}(m)$, where $m = |\mathrm{H}_T(v)|$, to $\mathrm{Sym}(O)$: let $v_1 \ldots v_m = \mathrm{H}_T(v)$, and $\pi \in \mathrm{Sym}(O)$ defined by $\forall i, (v_i)^\pi = v_{i^\sigma}$, we let $\Phi^v_T(\sigma) = \pi$. The reader may check that Φ^v_T is a group isomorphism.

Finally, for any $v \in O$ we define a group $\mathrm{G}_T(v)$, equal to I if $s(v) \in \Sigma$, and to $\Phi^v_T(G)$ if $s(v) = \langle f, c, G\rangle$.

Example 3.4 *Following our example, we have $\mathrm{H}_T(0) = 1.3.4$, so $v_1 = 1, v_2 = 3$ and $v_3 = 4$. Let $\pi = \Phi^0_T(\sigma)$, we have $1^\pi = v_{1^\sigma} = v_3 = 4, 3^\pi = v_{2^\sigma} = 1$ and $4^\pi = v_{3^\sigma} = 3$, so $\pi = (1\ 4\ 3)$.*

We may now define an action of the group $\mathrm{G}_T(v)$ on all terms built on O and s.

Definition 3.5 *For $T = (O, s, a)$, $v \in O$ and $\pi \in \mathrm{G}_T(v)$, let $T^\pi = (O, s, a^\pi)$, where $\forall v \in O, a^\pi(v) = a(v)^\pi$, i.e. π is applied to each letter in $a(v)$.*

Example 3.6 *We may apply $\pi = (1\ 4\ 3)$ to T, simply by replacing a by a^π. We have $a^\pi(0) = (1.2)^\pi = 4.2$ and $a^\pi(2) = (3.4)^\pi = 1.3$. See the result on Figure 2.*

The reader may check that T^π is a term, since a^π still defines a tree-like structure; this is due to the fact that $\mathrm{G}_T(v)$ only permutes occurrences from $\mathrm{H}_T(v)$, which is an antichain in T. It is obvious that $\forall \pi, \rho \in \mathrm{G}_T(v)$, we have $T^{\pi\rho} = (T^\pi)^\rho$, and that $T^{\mathrm{id}} = T$. Hence we have defined an action of $\mathrm{G}_T(v)$ on stratified terms built on O and s, if however we are able to prove that T^π is indeed a stratified term. This is not obvious since a random permutation on occurrences may disrupt the contexts (try for instance $\pi = (1\ 2)$ in Example 3.6; then $\mathrm{string}(T^\pi) = F(g(b, c), a)$ is not stratified). We first prove a lemma.

Lemma 3.7 $\forall u, v \in O, \forall \pi \in \mathrm{G}_T(v)$,

[3] An implementation would contain a concise representation of G, i.e. basically some set of generators.

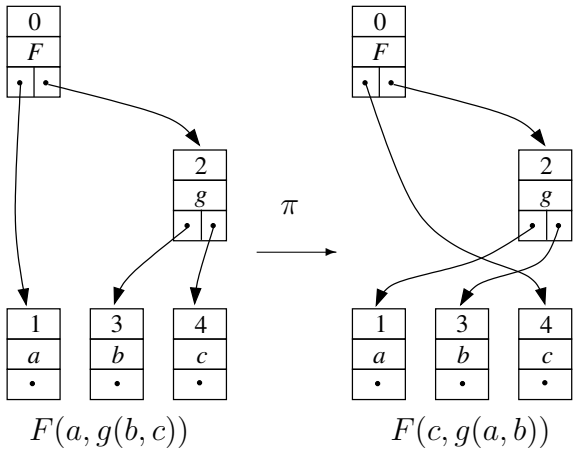


Fig. 2. The action of $\pi = (1\ 4\ 3)$ on T

(i) *if* $u \neq v$ *and* $s(u) \in \Sigma'$*, then* $T^\pi.u \setminus \mathrm{H}_T(u) = T.u \setminus \mathrm{H}_T(u)$,

(ii) $\pi(T.v \setminus \mathrm{H}_T(v)) = T^\pi.v \setminus \mathrm{H}_T(v)$.

Proof. If $s(v) \in \Sigma$, then $\mathrm{G}_T(v) = \mathrm{I}$, hence $\pi = \mathrm{id}$ and $T^\pi = T$, and both (i) and (ii) hold. If $s(v) \in \Sigma'$, with $\langle f, c, G\rangle = s(v)$,

(i) we first suppose that $T.u$ is a (strict) subterm of $T.v$; then $\exists w \in \mathrm{H}_T(v)$ such that $T.u$ is a subterm of $T.w$ (since $s(u) \in \Sigma'$ and c is a Σ-context), so $a(u) \cap \mathrm{H}_T(v) = \emptyset$, hence $a^\pi(u) = a(u)$, and by induction we conclude $T^\pi.u = T.u$. Suppose now that $T.u$ is not a subterm of $T.v$, then $\mathrm{H}_T(v) \cap \mathrm{desc}(T^\pi.u \setminus \mathrm{H}_T(u)) = \emptyset$, and we clearly get $T^\pi.u \setminus \mathrm{H}_T(u) = T.u \setminus \mathrm{H}_T(u)$.

(ii) $\forall u \in \mathrm{desc}(T.v \setminus \mathrm{H}_T(v))$, either $u \in \mathrm{H}_T(v)$, and then $\pi(u) \in \mathrm{H}_T(v)$ and $s(u) = s(\pi(u)) = []$; or $u \notin \mathrm{H}_T(v)$, and then $\pi(u) = u$, hence $s(\pi(u)) = s(u)$ and $a^\pi(\pi(u)) = a^\pi(u) = \pi(a(u))$. This proves $\pi(T.v \setminus \mathrm{H}_T(v)) = T^\pi.v \setminus \mathrm{H}_T(v)$.

Theorem 3.8 $\forall v \in O, \forall \pi \in \mathrm{G}_T(v)$, T^π *is a stratified term and* $\mathrm{H}_{T^\pi} = \pi \circ \mathrm{H}_T$.

Proof. $\forall u \in O$, if $u \in \Sigma'$ then by Lemma 3.7 we get $T^\pi.u \setminus \mathrm{H}_T(u) \simeq T.u \setminus \mathrm{H}_T(u)$, which is isomorphic to the required context since T is stratified, and therefore so is T^π.

Remark that if $u \neq v$, we have $\pi(T.u \setminus \mathrm{H}_T(u)) = T.u \setminus \mathrm{H}_T(u)$. Hence by Lemma 3.7 we have $\pi(T.u \setminus \mathrm{H}_T(u)) = T^\pi.u \setminus \mathrm{H}_T(u)$, so that $\pi(\mathrm{H}_T(u)) = \mathrm{H}_{T^\pi}(u)$.

4 Stratified rewriting

We therefore have an action of $\mathrm{G}_T(v)$ on stratified terms built on O and s, which we use to define the rewriting of T *at* an occurrence v, through the theory specified in $s(v)$.

Definition 4.1 *We say that* T rewrites at v *into* T'*, noted* $T \to^{v,\star} T'$*, iff* $\exists \pi \in \mathrm{G}_T(v), T' = T^\pi$*. We note* $\to$ *for* $\bigcup_{v \in O} \to^{v,\star}$*, and* $\to^\star$ *its reflexive and transitive*

closure [4] *. Since the relation* $\rightarrow^{v,\star}$ *is symmetric, so are* $\rightarrow$ *and* $\rightarrow^\star$*.*

The stratified set *of* T*, noted* $[T]_{\rm s}$*, is the equivalence class of* T *modulo* $\rightarrow^\star$*. We also define* $\mathrm{S}[T] = \mathrm{string}(\mathrm{um}([T]_{\rm s}))$.

Our aim is now to build a group that will yield stratified sets as orbits. We construct it from the $\mathrm{G}_T(v)$'s, but we must first establish the following commutativity.

Lemma 4.2 $\forall u, v \in O, \mathrm{G}_T(u)\mathrm{G}_T(v) = \mathrm{G}_T(v)\mathrm{G}_T(u)$.

Proof. If $u = v$ this is obvious, so suppose that $u \neq v$, then $\mathrm{H}_T(u) \cap \mathrm{H}_T(v) = \emptyset$, so that $\forall \pi \in \mathrm{G}_T(u), \forall \pi' \in \mathrm{G}_T(v)$, π and π' have disjoint cycles, hence $\pi\pi' = \pi'\pi$.

This allows the definition of the following product.

Definition 4.3 *Let* $\mathrm{G}(T) = \prod_{v \in O} \mathrm{G}_T(v)$*. If* $\mathrm{desc}(T) = v_1 \dots v_n$*, and* $\pi = \prod_{i=1}^n \pi_i \in \mathrm{G}(T)$*, where* $\pi_i \in \mathrm{G}_T(v_i)$*, we define* $T^\pi = (\cdots (T^{\pi_1})^{\pi_2} \cdots)^{\pi_n}$.

Example 4.4 *In our example,* $\mathrm{G}(T)$ *is the group generated by* π*, thus containing the three elements* $\pi, \pi^2 = (1\ 3\ 4)$ *and* $\pi^3 = \mathrm{id}$*. Hence* $\mathrm{S}[T] = \{f(a, g(b, c)), f(c, g(a, b)), f(b, g(c, a))\}$.

It is obvious from Theorem 3.8 that $\forall \pi \in \mathrm{G}(T), T^\pi$ is a stratified term. We now prove that we have defined a semi-regular action of $\mathrm{G}(T)$ on the set of stratified terms built on O and s.

Theorem 4.5 (i) *We have* $T^{\mathrm{id}} = T$ *and* $\forall \pi, \pi' \in \mathrm{G}(T), (T^\pi)^{\pi'} = T^{\pi\pi'}$.

(ii) *If* $T^\pi = T^{\pi'}$ *then* $\pi = \pi'$.

Proof.

(i) $T^{\mathrm{id}} = T$ is obvious. Let $v_1 \dots v_n = \mathrm{desc}(T)$, and $\pi, \pi' \in \mathrm{G}(T)$, with $\pi = \prod_{i=1}^n \pi_i$, $\pi' = \prod_{i=1}^n \pi'_i$ where $\forall i, \pi_i, \pi'_i \in \mathrm{G}_T(v_i)$.

As in the proof of Lemma 4.2, if $i \neq j$ then $\pi_i\pi_j = \pi_j\pi_i$. Hence $\pi\pi' = \prod_{i=1}^n \pi_i\pi'_i$, and by definition $T^{\pi\pi'} = (\cdots T^{\pi_1\pi'_1} \cdots)^{\pi_n\pi'_n}$. Remark that in $(T^\pi)^{\pi'}$, the π'_i's are applied in the order specified in $\mathrm{desc}(T^\pi)$. However, the order is irrelevant, since obviously $\forall u \in O, \forall \rho \in \mathrm{G}_T(v_i), \forall \rho' \in \mathrm{G}_T(v_j), (a^\rho)^{\rho'}(u) = a^{\rho\rho'}(u) = a^{\rho'\rho}(u)$ (by Lemma 4.2), thus $(T^\rho)^{\rho'} = (T^{\rho'})^\rho$ (if $i \neq j$). Successive applications of this swapping rule yields $(T^\pi)^{\pi'} = ((\cdots (T^{\pi_1})^{\pi'_1} \cdots)^{\pi_n})^{\pi'_n} = T^{\pi\pi'}$.

(ii) If $T^\pi = T^{\pi'}$ then $\forall u \in O$, if $u_1 \dots u_m = a(u)$, then $u_1^\pi \dots u_m^\pi = a^\pi(u) = a^{\pi'}(u) = u_1^{\pi'} \dots u_m^{\pi'}$, so that $\forall i, u_i^\pi = u_i^{\pi'}$. The only occurrence that does not occur in a $a(u)$ is $\mathrm{root}(T)$, which is a fix-point for both π and π'. Hence $\forall u \in O, u^\pi = u^{\pi'}$, i.e. $\pi = \pi'$.

This proves that rewritings at different occurrences are essentially independent, even if one occurrence appears in a subterm of another one. We are now going to use the $\mathrm{G}(T)$-orbit of T, but we must first remark that it is not quite standard to consider the orbit of an element w.r.t. a group that *depends* on this element.

[4] The relation $\rightarrow^{v,\star}$ corresponds to the relation $\rightarrow^{{\rm s},\star}{}_i$ of [1], but is defined on occurrence terms rather than terms-as-strings.

We may not speak of the orbit partition of the set of stratified terms, since many different groups are involved. One thing we need however, is to make sure that we keep the same group on the whole orbit.

Lemma 4.6 $\forall \pi \in \mathrm{G}(T), \forall v \in O, \mathrm{G}_{T^\pi}(v) = \mathrm{G}_T(v)$, *and* $\mathrm{G}(T^\pi) = \mathrm{G}(T)$.

Proof. We first prove it for $\pi \in \mathrm{G}_T(u)$, for any $u \in O$. If $s(v) \in \Sigma$, then $\mathrm{G}_{T^\pi}(v) = \mathrm{I} = \mathrm{G}_T(v)$. If $s(v) \in \Sigma'$, let G be the group in $s(v)$, we have $\mathrm{G}_T(v) = \Phi^v_T(G)$, $\mathrm{G}_{T^\pi}(v) = \Phi^v_{T^\pi}(G)$ and by Theorem 3.8 $\mathrm{H}_{T^\pi}(v) = \pi(\mathrm{H}_T(v))$.

If $u \neq v$, since π is a permutation of $\mathrm{H}_T(u)$ disjoint from $\mathrm{H}_T(v)$, we have $\mathrm{H}_{T^\pi}(v) = \mathrm{H}_T(v)$, so that $\Phi^v_{T^\pi} = \Phi^v_T$, hence $\mathrm{G}_{T^\pi}(v) = \mathrm{G}_T(v)$.

If $u = v$, let $v_1 \dots v_m = \mathrm{H}_T(v)$, and for all $\sigma \in G$, let $\rho = \Phi^v_{T^\pi}(\sigma)$, by definition we have $\pi(v_i)^\rho = \pi(v_{i^\sigma})$, thus $v_i^{\pi\rho\pi^{-1}} = v_{i^\sigma}$, which proves that $\pi\rho\pi^{-1} = \Phi^v_T(\sigma)$, i.e. $\Phi^v_{T^\pi}(\sigma) = \pi^{-1}\Phi^v_T(\sigma)\pi$. Hence $\mathrm{G}_{T^\pi}(v) = \pi^{-1}\mathrm{G}_T(v)\pi = \mathrm{G}_T(v)$ since $\pi \in \mathrm{G}_T(v)$.

If $\pi \in \mathrm{G}(T)$, then π decomposes into a product of elements of the $\mathrm{G}_T(u)$'s, and the result follows by induction. $\mathrm{G}(T^\pi) = \mathrm{G}(T)$ is a trivial consequence.

In this proof, if we express $\pi = \Phi^v_T(\sigma')$, we see that $\Phi^v_T(\sigma')\Phi^v_{T^\pi}(\sigma) = \Phi^v_T(\sigma)\Phi^v_T(\sigma') = \Phi^v_T(\sigma\sigma')$, i.e. if we apply σ on the term obtained *after* applying σ', we obtain the same result as applying the product of σ and σ' in *reverse* order. This is a source of confusion that explains why the subscript T in Φ^v_T is mandatory. The previous lemma proves that the group, i.e. the image of these morphisms, is invariant under π, even though the morphisms are not. We can now obtain stratified sets as orbits.

Theorem 4.7 $[T]_{\mathrm{s}} = T^{\mathrm{G}(T)}$.

Proof. We first show by induction that $\forall i \in \mathbb{N}$, if $T \to^i T'$ then $\exists \pi \in \mathrm{G}(T)$ such that $T' = T^\pi$. This is trivial for $i = 0$, with $\pi = \mathrm{id}$. If true for i, and $T \to^{i+1} T'$, then by induction hypothesis $\exists \pi \in \mathrm{G}(T), \exists v \in O$ such that $T \to^i T^\pi \to^{v,\star} T'$, and by definition $\exists \rho \in G_{T^\pi}(v)$ such that $T' = T^{\pi\rho}$. By Lemma 4.6, we have $\rho \in \mathrm{G}(T^\pi) = \mathrm{G}(T)$, which completes the induction, and proves $[T]_{\mathrm{s}} \subseteq T^{\mathrm{G}(T)}$.

Conversely, $\forall \pi \in \mathrm{G}(T)$, let $\{v_1, \dots, v_n\} = O$, then $\forall i, \exists \pi_i \in \mathrm{G}_T(v_i)$ such that $\pi = \pi_1 \cdots \pi_n$. Let $T_1 = T$ and $T_{i+1} = T_i^{\pi_i}$ for $i = 1 \dots n$, suppose $T \to^\star T_i$ (which is true for $i = 1$), since $\pi_i \in \mathrm{G}_T(v_i) = G_{T_i}(v_i)$ by Lemma 4.6, then $T_i \to^{v_i,\star} T_i^{\pi_i}$, thus $T_i \to T_{i+1}$, and therefore $T \to^\star T_{i+1}$. This proves by induction that $T \to^\star T_{n+1} = T^\pi$, yielding $T^\pi \in [T]_{\mathrm{s}}$.

5 On cardinality

Since the action is semi-regular, the cardinality of $[T]_{\mathrm{s}}$ is the order of the group $\mathrm{G}(T)$, which can be computed in polynomial time from the generators of the groups in Σ' (see [6]). However, the stratified set as defined in [1] is $\mathrm{string}([T]_{\mathrm{s}})$, which is equipotent to the quotient $[T]_{\mathrm{s}}/\simeq$. Let us consider an example.

Example 5.1 *We consider a ternary function symbol g and two constant symbols*

A and B. The theory axiomatised by $\forall xyz, g(x,y,z) = g(y,z,x)$ is represented by the context $c = g([],[],[])$ and the group G generated by $(1\ 2\ 3)$. Let $g' = \langle g, c, G\rangle$, $O = \{0,\dots,9,\beth,\daleth,\gimel\}$, we define the functions s and a by:

	0	$\beth$	$\daleth$	$\gimel$	1	2	3	4	5	6	7	8	9
s	g'	g'	g'	g'	A	A	A	B	B	B	A	A	B
a	$\beth.\daleth.\gimel$	1.2.3	4.5.6	7.8.9	ε	ε	ε	ε	ε	ε	ε	ε	ε

This defines a stratified term $T = (O, s, a)$, whose root is 0. We have $\mathrm{string}(\mathrm{um}(T)) = g(g(A,A,A), g(B,B,B), g(A,A,B))$, and $\mathrm{G}(T)$ is the group generated by $\{(\beth\ \daleth\ \gimel), (1\ 2\ 3), (4\ 5\ 6), (7\ 8\ 9)\}$, which has 81 elements, and so does $[T]_{\mathrm{s}}$. We now list the elements in $\mathrm{S}[T]$, each followed by the number of times it is obtained as a $\mathrm{string}(\mathrm{um}(T^\pi))$ for $\pi \in \mathrm{G}(T)$.

$$
\begin{array}{ll}
g(g(B,A,A), g(A,A,A), g(B,B,B)) & 9 \\
g(g(A,B,A), g(A,A,A), g(B,B,B)) & 9 \\
g(g(A,A,B), g(A,A,A), g(B,B,B)) & 9 \\
g(g(B,B,B), g(B,A,A), g(A,A,A)) & 9 \\
g(g(B,B,B), g(A,B,A), g(A,A,A)) & 9 \\
g(g(B,B,B), g(A,A,B), g(A,A,A)) & 9 \\
g(g(A,A,A), g(B,B,B), g(B,A,A)) & 9 \\
g(g(A,A,A), g(B,B,B), g(A,B,A)) & 9 \\
g(g(A,A,A), g(B,B,B), g(A,A,B)) & 9
\end{array}
$$

This means that $[T]_{\mathrm{s}}/\simeq$ is an equipartition of $[T]_{\mathrm{s}}$. We are now going to prove that this is always the case. This is probably an important result toward computing the number of partitions.

Definition 5.2 *For $u \in O \setminus \{\mathrm{root}(T)\}$ we define an integer $\mathrm{pos}(T,u)$ (the* position *of u in T) and an occurrence $\mathrm{sup}(T,u)$, by:*

- *If $\exists v \in O$ such that $u \in \mathrm{H}_T(v)$, then v is unique, let $v_1 \dots v_m = \mathrm{H}_T(v)$, then $\mathrm{pos}(T,u) = j$, where $v_j = u$, and $\mathrm{sup}(T,u) = v$.*
- *If u is not in a $\mathrm{H}_T(v)$, then $\exists! v \in O$ such that $u \in a(v)$, and we let $\mathrm{pos}(T,u) = j$ where u appears as the j^{th} letter in $a(v)$, and $\mathrm{sup}(T,u) = v$.*

Example 5.3 *Considering the term T defined in Example 3.2, we have $\mathrm{sup}(T,2) = 0, \mathrm{pos}(T,2) = 2$ and $\mathrm{sup}(T,4) = 0, \mathrm{pos}(T,4) = 3$.*

Lemma 5.4 *$\forall \eta \in \mathrm{Sym}(O), \forall u \in O \setminus \{\mathrm{root}(T)\}$, we have*

(i) $\mathrm{sup}(\eta(T), \eta(u)) = \eta(\mathrm{sup}(T,u))$,

(ii) $\mathrm{pos}(\eta(T), \eta(u)) = \mathrm{pos}(T, u)$,
(iii) $\eta(\mathrm{root}(T)) = \mathrm{root}(\eta(T))$.

We leave the proof of this lemma to the reader.

Lemma 5.5 *$\forall u \in O \setminus \{\mathrm{root}(T)\}, \pi \in \mathrm{G}(T)$, let $v = \sup(T, u)$ and σ be the inverse image by Φ^v_T of the restriction of π to $\mathrm{H}_T(v)$, then $\mathrm{pos}(T^\pi, u) = \mathrm{pos}(T, u)^{\sigma^{-1}}$.*

Proof. If $s(v) \in \Sigma$ then $\sigma = \mathrm{id}$ and the result is trivial. If $s(v) \in \Sigma'$, let $v_1 \dots v_m = \mathrm{H}_T(v)$, then $\mathrm{H}_{T^\pi}(v) = \pi(v_1 \dots v_m) = v_{1^\sigma} \dots v_{m^\sigma}$, so if $u = v_{j^\sigma}$ then $\mathrm{pos}(T^\pi, u) = j$, and $\mathrm{pos}(T, u) = j^\sigma$.

Theorem 5.6 *$\forall \pi, \nu \in \mathrm{G}(T), \forall \eta \in \mathrm{Sym}(O)$, if $\eta(T) = T^\pi$ then $\eta(T^\nu) \in T^{\mathrm{G}(T)}$.*

Proof. For all $v \in O$, let σ_v be the inverse image by Φ^v_T of the restriction of ν to $\mathrm{H}_T(v)$, and $\mu_v = \Phi^{\eta(v)}_{T^\pi}(\sigma_v)$, we have $\mu_v \in \mathrm{G}_{T^\pi}(\eta(v)) = \mathrm{G}_T(\eta(v))$ by Lemma 4.6. Let $\mu = \prod_{v \in O} \mu_v$, we have $\mu \in \mathrm{G}(T)$, and we will prove that $\eta(T^\nu) = T^{\pi\mu}$. Remark that μ_v is the restriction of μ to $\mathrm{H}_{T^\pi}(\eta(v))$.

We clearly have $\mathrm{root}(T^{\pi\mu}) = \mathrm{root}(T)$, and according to Lemma 5.4 (iii) we have

$$\begin{aligned}
\mathrm{root}(\eta(T^\nu)) &= \eta(\mathrm{root}(T^\nu)) \\
&= \eta(\mathrm{root}(T)) \\
&= \mathrm{root}(\eta(T)) \\
&= \mathrm{root}(T^\pi) \qquad \text{(since } \eta(T) = T^\pi\text{)} \\
&= \mathrm{root}(T),
\end{aligned}$$

so the two terms have the same root. Moreover, $\forall u \in O \setminus \{\mathrm{root}(T)\}$, let $v = \sup(T, u)$, then by Lemma 5.4 (i) we have $\eta(v) = \sup(\eta(T), \eta(u)) = \sup(T^\pi, \eta(u))$, so that

$$\begin{aligned}
\mathrm{pos}(T^{\pi\mu}, \eta(u)) &= \mathrm{pos}(T^\pi, \eta(u))^{\sigma_v^{-1}} && \text{(by Lemma 5.5)} \\
&= \mathrm{pos}(\eta(T), \eta(u))^{\sigma_v^{-1}} \\
&= \mathrm{pos}(T, u)^{\sigma_v^{-1}} && \text{(by Lemma 5.4 (ii))} \\
&= \mathrm{pos}(T^\nu, u) && \text{(by Lemma 5.5)} \\
&= \mathrm{pos}(\eta(T^\nu), \eta(u)) && \text{(by Lemma 5.4 (ii)).}
\end{aligned}$$

So all occurrences have the same positions in these two terms. By induction, they must be equal. Carrying this last induction requires some more formalism, of which we have decided to exempt the reader.

Corollary 5.7 *$[T]_{\mathrm{s}}/\simeq$ is an equipartition of $[T]_{\mathrm{s}}$.*

Proof. Consider any element $C = \{T_1, \dots, T_n\} \in [T]_{\mathrm{s}}/\simeq$, then $\exists \pi_1, \dots, \pi_n \in \mathrm{G}(T)$ such that $\forall i, T_i = T_1^{\pi_i}$, and $\exists \eta_1, \dots, \eta_n \in \mathrm{Sym}(O)$ such that $\eta_i(T_1) = T_i$. Since $\forall j \neq i, T_j \neq T_i$, then $\eta_j \neq \eta_i$.

Consider now any $C' \in [T]_s/\simeq$, $T' \in C'$ and $\nu \in G(T)$ such that $T' = T_1^\nu$. Since $\forall i, \eta_i(T_1) = T_1^{\pi_i}$, then by the previous theorem $\eta_i(T') = \eta_i(T_1^\nu) \in T_1^{G(T_1)} = [T]_s$. Therefore $\eta_i(T') \in C'$, and $\forall j \neq i, \eta_j(T') \neq \eta_i(T')$. This proves that $|C'| \geq n = |C|$.

This is true for any C, C', so we also have $|C| \geq |C'|$, hence $|C| = |C'|$.

6 The equivalence problem

Definition 6.1 *The* equivalence problem, *given T_1 and T_2, is the problem $\exists T \in [T_1]_s$ such that $T \simeq T_2$; we note it $T_1 \bowtie T_2$.*

On any stratified term T we consider a binary relation $\sim$ on O defined by $\forall u, v \in O, u \sim v \Leftrightarrow \exists \pi \in G(T.u), T^\pi.u \simeq T.v$.

We also define the height *of v in T, noted $h_T(v)$, as follows: if $s(v) \in \Sigma$, then $h_T(v) = 1$ if $a(v) = \varepsilon$, and $h_T(v) = 1 + \max\{h_T(u)/u \in a(v)\}$ if $a(v) \neq \varepsilon$; if $s(v) \in \Sigma'$, then $h_T(v) = 1 + \max\{h_T(u)/u \in H_T(v)\}$.*

Given two stratified terms on disjoint occurrences $T_i = (O_i, s_i, a_i)$ for $i = 1, 2$, we consider a new occurrence α and a new binary function symbol @, let $O = O_1 \uplus O_2 \uplus \{\alpha\}$, the function s equal to s_i on O_i and $s(\alpha) =$ @, and the function a equal to a_i on O_i and $a(\alpha) = root(T_1)root(T_2)$; then $T_1 @ T_2 = (O, s, a)$ is a stratified term.

Example 6.2 *Considering the term T defined in Example 3.2, we have $h_T(0) = h_T(2) = 2$, even though $T.2$ is a subterm of $T.0$.*

It is easy to see that the height is invariant under both isomorphisms and the action of $G(T)$, i.e. $h_{\eta(T)}(\eta(v)) = h_T(v)$, and $h_{T^\pi}(v) = h_T(v)$. As a consequence we have $u \sim v \Rightarrow h_T(u) = h_T(v)$ (and $s(u) = s(v)$ as well), which justifies the following induction.

Theorem 6.3 *$\sim$ is an equivalence relation.*

Proof. Reflexivity is trivial. Suppose $\sim$ is an equivalence relation on occurrences of height strictly less than h, and consider $u, v \in O$ such that $h_T(u) = h_T(v) = h$.

If $u \sim v$ then $\exists \pi \in G(T.u), T^\pi.u \simeq T.v$. In the case $s(u) \in \Sigma$, let $a(u) = u_1 \dots u_n$ and $a(v) = v_1 \dots v_n$, we can write $\pi = \pi_1 \cdots \pi_n$ such that $\forall i, \pi_i \in G(T.u_i)$, and since $a^\pi(u) = a(u)$ we easily get $T^{\pi_i}.u_i \simeq T.v_i$. Therefore $u_i \sim v_i$, and by induction hypothesis $v_i \sim u_i$, i.e. $\exists \pi'_i \in G(T.v_i), T^{\pi'_i}.v_i \simeq T.u_i$. Let $\pi' = \pi'_1 \cdots \pi'_n \in G(T.v)$, we obviously have $T^{\pi'}.v \simeq T.u$, (since $a^{\pi'}(v) = a(v)$), hence $v \sim u$.

We now consider the case where $s(u) \in \Sigma'$, let $u_1 \dots u_m = H_T(u)$ and $v_1 \dots v_m = H_T(v)$, we can write $\pi = \mu\pi_1 \cdots \pi_m$ such that $\forall i, \pi_i \in G(T.u_i)$ and $\mu \in G_T(u)$. Let σ be the inverse image of μ by Φ_T^u and $\mu' = \Phi_T^v(\sigma^{-1})$, as above we have $u_{i^\sigma} \sim v_i$, so that $\exists \pi'_i \in G(T.v_i), T^{\pi'_i}.v_i \simeq T.u_{i^\sigma}$, and we let $\pi' = \mu'\pi'_1 \cdots \pi'_m \in G(T.v)$. We have $T^{\pi'_j}.v_j \simeq T.u_i$, where $j = i^{\sigma^{-1}}$, and by Lemma 5.5 we have $pos(T^{\pi'}, v_j) = pos(T, v_j)^\sigma = j^\sigma = i = pos(T, u_i)$, hence $T^{\pi'}.v \simeq T.u$. This proves that $\sim$ is symmetric.

For transitivity we add a $w \in O$ such that $\mathrm{h}_T(w) = \mathrm{h}_T(u)$, and the hypothesis $v \sim w$, so that $\exists \rho \in \mathrm{G}(T.v), T^\rho.v \simeq T.w$. We skip the easy case $s(u) \in \Sigma$ and suppose $s(u) \in \Sigma'$, with $w_1 \dots w_m = \mathrm{H}_T(w)$. We can write $\pi = \mu\pi_1 \cdots \pi_m$ and $\rho = \nu\rho_1 \cdots \rho_m$ where $\mu \in \mathrm{G}_T(u), \nu \in \mathrm{G}_T(v), \pi_i \in \mathrm{G}(T.u_i)$ and $\rho_i \in \mathrm{G}(T.v_i)$, and if σ is the inverse image of μ by Φ^u_T, and τ is the inverse image of ν by Φ^v_T, then we get $u_{i^\sigma} \sim v_i$ and $v_{i^\tau} \sim w_i$. By induction hypothesis we have $u_{i^{\tau\sigma}} \sim w_i$, so that $\exists \pi'_i \in \mathrm{G}(T.u_{i^{\tau\sigma}}), T^{\pi'_i}.u_{i^{\tau\sigma}} \simeq T.w_i$. Let $\pi' = \Phi^u_T(\tau\sigma)\pi'_1 \cdots \pi'_m \in \mathrm{G}(T.u)$, we have $\mathrm{pos}(T^{\pi'}, u_{i^{\tau\sigma}}) = \mathrm{pos}(T, u_{i^{\tau\sigma}})^{(\tau\sigma)^{-1}} = i = \mathrm{pos}(T, w_i)$, hence $T^{\pi'}.u \simeq T.w$, and the induction is complete.

This is clearly related to the equivalence problem:

Lemma 6.4 *$T_1 \bowtie T_2$ iff $\mathrm{root}(T_1) \sim \mathrm{root}(T_2)$ in $T = T_1 @ T_2$.*

Proof. Let $r_i = \mathrm{root}(T_i)$, we have by definition of $\sim$

$$\begin{aligned} r_1 \sim r_2 &\Leftrightarrow \exists \pi \in \mathrm{G}(T.r_1), T^\pi.r_1 \simeq T.r_2 \\ &\Leftrightarrow \exists \pi \in \mathrm{G}(T_1), T_1^\pi \simeq T_2 \\ &\Leftrightarrow T_1 \bowtie T_2 \qquad \text{by Theorem 4.7.} \end{aligned}$$

A trivial consequence is that $\bowtie$ is an equivalence relation among stratified terms. But the point is that we can compute $\bowtie$ by determining the $\sim$-classes of occurrences of a suitable term (built in linear time), which can be performed recursively. It is rather trivial to see that $\forall u, v \in O$, if $s(u) = s(v) \in \Sigma$, then $u \sim v$ iff $\forall i, u_i \sim v_i$, where $a(u) = u_1 \dots u_n$ and $a(v) = v_1 \dots v_n$. But if $s(u) = s(v) \in \Sigma'$, we must resort to more complex notions.

Definition 6.5 *If $s(u) \in \Sigma'$, let* $\mathrm{E}(u) = \prod_{C \in \mathrm{H}_T(u)/\sim} \mathrm{Sym}(C)$.

Example 6.6 *Considering the term T defined in Example 5.1, we have* $\mathrm{H}_T(\text{ℶ})/\sim = \{\{7,8\},\{9\}\}$, *so that* $\mathrm{E}(\text{ℶ}) = \mathrm{Sym}(\{7,8\})\mathrm{Sym}(\{9\}) = \{\mathrm{id}, (7\ 8)\}$.

The group $\mathrm{E}(u)$ has a particularly simple structure, and a generating set can easily be computed, supposing as we do that we have determined $\sim$ on the subterms of u and v. We then show that determining whether $u \sim v$ holds or not reduces to a group theoretic problem.

Lemma 6.7 *$\forall u, v \in O$ such that $s(u) = s(v) \in \Sigma'$, if $\mathrm{H}_T(u) = u_1 \dots u_m$ and $\mathrm{H}_T(v) = v_1 \dots v_m$, then*

(i) *$u \sim v$ iff $\exists \sigma \in \mathrm{Sym}(m)$ such that conditions* (7) *and* (8) *below hold,*

$$\forall i, u_{i^\sigma} \sim v_i \tag{7}$$

$$\Phi^u_T(\sigma)\mathrm{E}(u) \cap \mathrm{G}_T(u) \neq \emptyset \tag{8}$$

(ii) *if $\exists \sigma, \sigma' \in \mathrm{Sym}(m)$ such that $\forall i, u_{i^\sigma} \sim v_i$ and $u_{i^{\sigma'}} \sim v_i$, we have $\Phi^u_T(\sigma)\mathrm{E}(u) = \Phi^u_T(\sigma')\mathrm{E}(u)$.*

$Test(u, v) =$
  assert $s(u) = s(v) \wedge \mathrm{h}_T(u) = \mathrm{h}_T(v)$;
  if $s(u) \in \Sigma$ **then**
    let $u_1 \dots u_n = a(u)$ **and** $v_1 \dots v_n = a(v)$ **in**
    return $\forall i \in \{1, \dots, n\}, u_i \sim v_i$
  else let $u_1 \dots u_m = \mathrm{H}_T(u)$ **and** $v_1 \dots v_m = \mathrm{H}_T(v)$ **in**
    let $J = \{1, \dots, m\}$ **and** $\sigma = \emptyset$ **and** $c1 = true$ **in**
    for $i = 1$ **to** m **do**
($\star$) **if** $\exists j \in J / u_i \sim v_j$ **then**
        $\sigma := \sigma \cup \{\langle i, j \rangle\}; J := J \setminus \{j\}$
      else $c1 := false$
    done;
    return $c1 \wedge \Phi^u_T(\sigma)\mathrm{E}(u) \cap \mathrm{G}_T(u) \neq \emptyset$

Fig. 3. Algorithm $Test$

Proof.

(i) If $u \sim v$ then by definition $\exists \pi \in \mathrm{G}(T.u)$ such that $T^\pi.u \simeq T.v$. Let μ be the restriction of π to $\mathrm{H}_T(u)$, and σ its inverse image by Φ^u_T; since $\mu \in \mathrm{G}_T(u)$ we trivially have $\mu \in \Phi^u_T(\sigma)\mathrm{E}(u) \cap \mathrm{G}_T(u)$. And since $\mathrm{pos}(T^\pi, u_{i^\sigma}) = i = \mathrm{pos}(T, v_i)$ we have $u_{i^\sigma} \sim v_i$.

Conversely, we suppose there is a $\sigma \in \mathrm{Sym}(m)$ such that (7) and (8) hold, and let $\mu = \Phi^u_T(\sigma)$. There is a $\rho \in \mathrm{E}(u)$ such that $\mu\rho \in \mathrm{G}_T(u)$, and by definition of $\mathrm{E}(u)$ we have $\forall i, (u_{i^\sigma})^\rho \sim u_{i^\sigma} \sim v_i$. If τ is the inverse image of ρ by Φ^u_T we get $(u_{i^\sigma})^\rho = u_{i^{\sigma\tau}} \sim v_i$ by Theorem 6.3. Hence $\exists \pi_i \in \mathrm{G}_T(u_{i^{\sigma\tau}})$ such that $T^{\pi_i}.u_{i^{\sigma\tau}} \simeq T.v_i$, and if we let $\pi = \mu\rho\pi_1 \cdots \pi_m \in \mathrm{G}(T.u)$ we have $\mathrm{pos}(T^\pi, u_{i^{\sigma\tau}}) = i = \mathrm{pos}(T, v_i)$, and therefore $T^\pi.u \simeq T.v$, i.e. $u \sim v$.

(ii) Since $u_{i^{\sigma^{-1}\sigma'}} \sim v_{i^{\sigma^{-1}}} \sim u_i$, then $\Phi^u_T(\sigma^{-1}\sigma') \in \mathrm{E}(u)$, and therefore $\Phi^u_T(\sigma') \in \Phi^u_T(\sigma)\mathrm{E}(u)$.

The condition (8) is an instance of the coset intersection emptiness problem, noted CIE, defined on two subgroups G, H of $\mathrm{Sym}(n)$, given by generators, and on a permutation $\pi \in \mathrm{Sym}(n)$, and deciding $\pi H \cap G \neq \emptyset$.

Supposing that we are given an oracle for CIE, according to Lemma 6.7 (i) we can compute $u \sim v$ with the polynomial algorithm $Test(u, v)$, given in Figure 3 in pseudo-CAML. In this algorithm, the value of σ obviously depends on the choices of j on line ($\star$), but its existence (i.e. the value of $c1$) does not, hence according to Lemma 6.7 (ii) the value of $Test(u, v)$ is independent of these choices. We can therefore compute $T_1 \bowtie T_2$ with the polynomial algorithm $Equiv(T_1, T_2)$, given in Figure 4.

We have therefore established a polynomial Turing reduction (see [5]) from the equivalence problem on stratified terms to CIE. It is shown in [6] that CIE is polynomially equivalent to a number of problems on groups, including the problem of computing generators for the intersection of two groups, and the setwise stabiliser problem SET_STAB. This problem is defined on a subgroup G of $\mathrm{Sym}(n)$ and a

$Equiv(T_1, T_2) =$
  let $T = T_1 @ T_2$ **in**
  for $h = 1$ **to** $\mathrm{h}_T(\mathrm{root}(T_1))$ **do**
    let $V = \{v \in O / \mathrm{h}_T(v) = h\}$ **in**
    while $V \neq \emptyset$ **do**
      choose $v \in V$;
      let $U = \{u \in V / s(u) = s(v)\}$ **in**
      $V := V \setminus U$;
      for $\langle u', v' \rangle \in U^2$ **do** $u' \sim v' := Test(u', v')$ **done**
    done
  done;
  return $\mathrm{root}(T_1) \sim \mathrm{root}(T_2)$

Fig. 4. Algorithm $Equiv$

subset A of $\{1, \ldots, n\}$, and consists in computing a set of generators for the group $G_A = \{\sigma \in G / A^\sigma = A\}$.

Although no polynomial algorithm is known for these problems, efficient techniques exploiting group theoretic properties can be used, see e.g. [7]. Problems that are computationally close to the graph isomorphism problem GI (it is shown in [6] that GI polynomially reduces to the problems in the class of CIE) often exhibit efficient average case algorithms (see [2]). On the peculiar status of GI's complexity, see [9, chapters 16 to 18].

From a theoretical point of view it also important that we provide a polynomial reduction in the reverse direction, i.e. from an isomorphism-hard problem to our equivalence problem, hence making sure (as much as can be) that $T_1 \bowtie T_2$ can not be solved in polynomial time. This is not as obvious as Lemma 6.7 (i) suggests, due to the particularly simple structure of the group $\mathrm{E}(u)$, i.e. it seems that we do not need the full generality of CIE. This may well lower its complexity, since asserting special properties on groups sometimes yields polynomial algorithms (as for instance for computing the intersection of a group with a p-group, see [6]).

However, an analysis of some proofs in [6], which we cannot carry here, led us to a polynomial reduction from SET_STAB to this subproblem of CIE, and therefore to our equivalence problem, hence showing that it is polynomially equivalent to SET_STAB, CIE, etc.

One may wonder why we focus on the equivalence problem on stratified terms. It is not one of the problems considered in [1], which are more complex in the sense that they are about the sets $\mathrm{S}[T]$ rather than about the orbits $[T]_\mathrm{s}$. The set $\mathrm{S}[T]$ can be considered as the *meaning* of the stratified term T. We feel however that the algorithms and techniques developed above can be useful for solving problems on $\mathrm{S}[T]$ (like unification) under reasonable hypotheses linking $\mathrm{S}[T]$ and $[T]_\mathrm{s}$. These hypotheses and links are still to be devised.

References

[1] J. Avenhaus and D. Plaisted. General algorithms for permutations in equational inference. *Journal of Automated Reasoning*, 26:223–268, April 2001.

[2] László Babai, Paul Erdõs, and Stanley M. Selkow. Random graph isomorphism. *SIAM Journal on Computing*, 9(3):628–635, aug 1980.

[3] Thierry Boy de la Tour. A note on symmetry heuristics in SEM. In Andrei Voronkov, editor, *Proceedings of CADE-18*, Lecture Notes in Artificial Intelligence 2392, pages 181–194, Copenhagen, Denmark, july 2002. Springer Verlag.

[4] G. Butler. *Fundamental algorithms for permutation groups*. Lecture Notes in Computer Science 559. Springer Verlag, 1991.

[5] M. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of* **NP***-completeness*. Freeman, San Francisco, California, 1979.

[6] C. Hoffmann. *Group-theoretic algorithms and graph isomorphism*. Lecture Notes in Computer Science 136. Springer Verlag, 1981.

[7] Jeffrey S. Leon. Permutation group algorithms based on partitions, I: Theory and algorithms. *Journal of Symbolic Computation*, 12(4–5):533–583, 1991.

[8] D. Plaisted. Equational reasoning and term rewriting systems. *Handbook of Logic in Artificial Intelligence and Logic Programming*, 1:273–364, 1993.

[9] Uwe Schöning and Randall Pruim. *Gems of Theoretical Computer Science*. Springer-Verlag, 1998.

Manipulating Tree Tuple Languages by Transforming Logic Programs [3]

Sébastien Limet [1]

Laboratoire d'Informatique Fondamentale d'Orléans (LIFO)
Université d'Orléans
BP 6759
F-45067 Orléans Cedex 2, France

Gernot Salzer [2]

Institut für Computersprachen
Technische Universität Wien
Favoritenstraße 9/E1852
A-1040 Wien, Austria

Abstract

We introduce inductive definitions over language expressions as a framework for specifying tree tuple languages. Inductive definitions and their sub-classes correspond naturally to classes of logic programs, and operations on tree tuple languages correspond to the transformation of logic programs. We present an algorithm based on unfolding and definition introduction that is able to deal with several classes of tuple languages in a uniform way. Termination proofs for clause classes translate directly to closure properties of tuple languages, leading to new decidability and computability results for the latter.

1 Introduction

First-order terms and term tuples (also called tree tuples) are the basic data structure in many areas of logic and computer science. What integers and reals are to numeric computing, terms are to formal verification, automated deduction, logic programming, and many other fields. Usually we are confronted not just with single terms or tree tuples but with infinite sets thereof when e.g. describing models in

[1] E-mail: limet@lifo.univ-orleans.fr
[2] E-mail: salzer@logic.at
[3] Extended abstract; see http://www.logic.at/css/ftp03.pdf for the proofs.

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

automatic deduction and logic programming, approximating calculi, or detecting infinite loops.

For theoretical and practical reasons we are interested in finite presentations of these infinite sets and in ways to manipulate the finite presentations in place of the infinite sets. Typical problems to solve are

- how to construct finite representations from the initial problem either statically by syntactic transformations, or dynamically by providing e.g. loop detection mechanisms during a computation, and
- how to perform operations like union or intersection on finite representations and how to test for properties like membership or emptiness.

The main difficulty is to find a good balance between the expressiveness of the chosen formalism and the tractability of the operations defined on it.

A prominent example are tree automata which define the class of regular tree languages (tree here means ground term). They are closed under standard set operations like union, intersection, and complement, both the membership and the emptiness problem are decidable, and the computational complexity of these operations is low. Therefore tree automata are widely used and have found applications in all areas mentioned above [1,13]. The drawback of regular tree languages is their weak expressiveness, which has lead to many extensions: tree automata with constraints [2], tree set automata [6], regular relations [1], or synchronized languages [9,11]. Most extensions lose either closure under some operations or the nice complexity of the operations, or both.

The work presented in this paper grew out of the study of synchronized languages. They were introduced in [11] to obtain finite representations of the solutions of certain R-unification problems, and were subsequently applied to R-disunification and one-step-rewriting [15]. Moreover, synchronized languages are able to represent the transitive closure of the relation induced by certain process algebras with communications and so can be used for model checking in this context [9].

Originally, synchronized languages were specified by means of Tree Tuple Synchronized Grammars (TTSGs), which are regular tree grammars with packs of productions that have to be applied at the same time. TTSGs are able to specify non-regular tree languages, but are a bit unwieldy; therefore grammar productions were subsequently replaced by set constraints [9]. In general, synchronized languages do not possess the same nice properties as regular tree language do, but they are considerably more expressive, and some relevant sub-classes are closed under operations like join or projection which are sufficient for many applications.

In this paper we go beyond constraint systems and introduce *inductive definitions over language expressions* (Section 2). Language expressions specify tuple languages by the usual set operations and by two term tuple operations, construction and filtering. Construction builds more complicated tuples out of simpler ones, while filtering selects certain tuples from a language. After fixing some notations from logic programming in Section 3, we translate inductive definitions to Horn

logic and vice versa in Section 4. This translation allows to deal with constraint systems in a uniform framework using notions and notations from clause logic. Section 5 defines a rule system that transforms any logic program into a – not necessarily finite – cs-program, the Horn-equivalent of a constraint system. Section 6 identifies several classes of programs for which the derivation process provably terminates, leading to new decidability results for tree tuple languages.

2 Tree Tuple Languages

Let Σ be a finite set of symbols with arities, and let T_Σ denote the set of all ground terms over Σ. Any subset of ${T_\Sigma}^n$ is called an *n-ary tree tuple language over Σ* (n-tuple language for short). Union, intersection, and Cartesian product are denoted by $\cup$, $\cap$, and $\times$, respectively. The notation $A_1 \times \cdots \times A_k$ is shorthand for $\{()\}$ (the set containing just the zero-tuple) if $k = 0$, for A_1 if $k = 1$, and for $A_1 \times (A_2 \times \cdots \times A_k)$ otherwise.

A *template* is a ground term that additionally may contain positive integers as constant symbols. Formally, if ω denotes the set of positive integers, then $T_{\Sigma\cup\omega}$ is the set of templates over Σ.[4] The integers are called *indices* and are used to refer to particular components of tuple languages. As usual we denote substitutions as sets, i.e., $A\{s_1 \mapsto t_1, \ldots, s_l \mapsto t_l\}$ is obtained from A by replacing simultaneously all occurrences of s_i by t_i. The arity of any object O will be denoted by $\mathrm{ar}(O)$, where O may be a function or predicate symbol, a language variable, or a language expression.

Let A be an n-tuple language, and let $\Box$ be a k-tuple of templates with l denoting the largest index occurring in $\Box$. The operations *construction* and *filtering* are defined as:

$$\Box \circ A \stackrel{\text{def}}{=} \{\, \Box\{1 \mapsto t_1, \ldots, l \mapsto t_l\} \mid (t_1, \ldots, t_{n+l}) \in A \times {T_\Sigma}^l \,\}$$
$$A/\Box \stackrel{\text{def}}{=} \{\, (t_1, \ldots, t_l) \in {T_\Sigma}^l \mid \Box\{1 \mapsto t_1, \ldots, l \mapsto t_l\} \in A \,\}$$

To some extent construction and filtering are inverse to each other: if the arity of A equals the largest index in $\Box$, i.e. $n = l$, and if $\Box$ contains all indices up to l then $(\Box \circ A)/\Box = A$.

Example 2.1 Let $\Sigma = \{a/0, f/1, g/2\}$, $A = \{(a, f(a)),\ (f(a), a)\}$, and let $\Box = [g(1,3), 2]$ be a pair of templates. We obtain:

$$\Box \circ A = \{\, (g(a,t), f(a)) \mid t \in T_\Sigma \,\} \cup \{\, (g(f(a),t), a) \mid t \in T_\Sigma \,\}$$
$$A/\Box = \emptyset$$
$$(\Box \circ A)/\Box = \{\, (a, f(a), t) \mid t \in T_\Sigma \,\} \cup \{\, (f(a), a, t) \mid t \in T_\Sigma \,\}$$

[4] For the sake of notational convenience we do not distinguish between integers and the symbols denoting them. Thus in a given context a variable i may represent a symbol when occurring in a formal entity like a template and at the same time the corresponding integer when occurring in a mathematical expression.

To specify interesting relations like one-step rewriting or the semantics of process algebras by term tuple languages we need some iterative or recursive mechanism. We propose inductive definitions with subset relations over language expressions as general framework. Let $\mathcal{X}$ be a set of language variables, each supplied with an arity. The set of *language expressions* is defined by the grammar

$$e ::= A \mid \{()\} \mid (e \times e) \mid (\Box \circ e) \mid (e/\Box)$$

for language variables A and template tuples $\Box$. An *inductive definition* is a set of constraints of form $A \supseteq e$ such that the arities of A and e coincide. [5] The languages defined by an inductive definition $\mathcal{D}$ are the least fixed point of the constraints. The least fixed point solution of variable A, i.e. the language associated with A, is denoted by $\mathcal{L}_{\mathcal{D}}(A)$.

Example 2.2 Operations on term tuples like forming sets of term tuples, computing union, intersection, joins, projections, or testing for membership can be expressed by proper language expressions:

$$\begin{aligned}
\{\boldsymbol{t}_1, \ldots, \boldsymbol{t}_n\} &= [\boldsymbol{t}_1] \circ \{()\} \cup \cdots \cup [\boldsymbol{t}_n] \circ \{()\} \\
e \cup f &= A \qquad \text{with the constraints } A \supseteq e \text{ and } A \supseteq f \\
e \cap f &= (e \times f)/[1, \ldots, m, 1, \ldots, n] \qquad \text{for } m = n \\
e \bowtie_{i,j} f &= (e \times f)/[1, \ldots, m+j-1, i, m+j, \ldots, m+n-1] \\
e \bowtie_{i,n} f &= (e \times f)/[1, \ldots, m+n-1, i] \\
\Pi_{i_1,\ldots,i_k} e &= [i_1, \ldots, i_k] \circ e \\
\boldsymbol{t} \in A &\equiv (A/[\boldsymbol{t}] \neq \emptyset) \equiv (A/[\boldsymbol{t}] = \{()\})
\end{aligned}$$

where $\boldsymbol{t}, \boldsymbol{t}_1, \ldots, \boldsymbol{t}_n$ are term tuples, e and f are m- and n-ary language expressions, respectively, and $1 \leq i, i_1, \ldots, i_k \leq m$, $1 \leq j < n$.

Example 2.3 Let $\Sigma = \{a/0, f/2\}$ and $\mathcal{X} = \{Id_2/2, Sym/2\}$. The constraints

$$\begin{aligned}
Id_2 &\supseteq \{(a, a)\} \cup [f(1,3), f(2,4)] \circ (Id_2 \times Id_2) \\
Sym &\supseteq \{(a, a)\} \cup [f(1,3), f(4,2)] \circ (Sym \times Sym)
\end{aligned}$$

define the languages $\mathcal{L}(Id_2) = \{\, (t, t) \mid t \in T_\Sigma \,\}$ and $\mathcal{L}(Sym) = \{\, (t, t') \mid t \in T_\Sigma$, t' is the symmetric tree of $t \,\}$.

The languages definable by inductive definitions are exactly the recursively enumerable tuple languages; there is a direct translation of constraints to Horn clauses and vice versa (see below). The drawback of such expressiveness is that properties like emptiness and membership tests get undecidable. Therefore we restrict the formalism of inductive definitions to single out tractable subclasses.

Constraint systems (CSs) were introduced in [9] as a simplified version of tree-tuple synchronized grammars. They can be viewed as inductive definitions without filter operations. In this case all constraints can be normalized to the form $A \supseteq$

[5] The arity of a language expression is the number of components in the tuples of the language defined by it; it can be determined in a straightforward manner since the arity of language variables is given.

$\Box \circ (A_1 \times \cdots \times A_k)$ for language variables $A, A_1, \ldots, A_k$. For constraint systems it is convenient to write indices as pairs of integers, $i.j$, referring to the j-th component of A_i. A constraint $A \supseteq \Box \circ (A_1 \times \cdots \times A_k)$ is called

- *linear* iff no index occurs twice in $\Box$; [6]
- *horizontal* iff for any two indices $i.j$ and $i'.j'$ in $\Box$, $i = i'$ implies that $i.j$ and $i'.j'$ occur in $\Box$ at positions of the same depth;
- *regular* iff it is linear and for some mapping $\pi\colon \omega \mapsto \omega$, each component of $\Box$ is of the form $f(i_1.j_1, \ldots, i_{\mathrm{ar}(f)}.j_{\mathrm{ar}(f)})$, where $\pi(i_l) = l$ for $l = 1, \ldots, \mathrm{ar}(f)$.

A CS is called linear (horizontal, regular) iff all constraints have the respective property. Obviously regularity implies horizontality and, by definition, linearity.

3 Logic Programming

We assume the reader to be familiar with the basics of logic programming, in particular with the operational and declarative semantics of definite logic programs (see e.g. [12]). We give a few definitions to fix notation.

Let $T_{\Sigma,V}$ denote the set of first-order terms over signature Σ and an infinite set of first-order variables, V. If P is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an atomic formula (*atom* for short). A term, an atom or a set of atoms is called *linear* if no variable occurs more than once in it. The most general (simultaneous) unifier of two (tuples of) terms, s and t, is denoted by $\mathrm{mgu}(s, t)$. The set of variables occurring in an object O will be denoted by $\mathrm{var}(O)$, where O may be a term, an atom, or a set of terms or atoms.

For atoms $H, B_1, \ldots, B_k$, the expression $H \leftarrow B_1, \ldots, B_k$ is called *program clause*, with H being the *head* and $B_1, \ldots, B_k$ the *body* of the clause; for $k = 0$ the clause is called a *fact*. A *logic program* is a finite set of program clauses. A *query* is a program clause without head. Program clauses and queries are summarized as *Horn clauses*.

The depth of a variable is 0, the depth of a constant is 1, and the depth of a functional term is the maximal depth of its arguments plus 1. The depth of an atom is the maximal depth of its arguments.

The least Herbrand model of a program $\mathcal{P}$ is denoted by $\mathcal{M}(\mathcal{P})$. The set of term tuples, for which a predicate P is true in $\mathcal{M}(\mathcal{P})$, is defined as

$$\mathcal{M}(\mathcal{P})|_P \stackrel{\mathrm{def}}{=} \{\, (t_1, \ldots, t_n) \mid P(t_1, \ldots, t_n) \in \mathcal{M}(\mathcal{P}) \,\} \ .$$

4 Inductive Definitions vs. Horn Logic

In this section we define translations from language expressions to semantically equivalent Horn clauses and vice versa. This enables us to discuss closure properties and decidability issues of constraint systems in a purely clausal setting and

[6] In [9], this property is called *non-copying*.

$$\frac{\top}{\{()\} \rightsquigarrow \langle (),\, \{\}\rangle} \qquad \frac{\top}{A \rightsquigarrow \langle (x_1, \ldots, x_{\mathrm{ar}(A)}),\, \{P_A(x_1, \ldots, x_{\mathrm{ar}(A)})\}\rangle}$$

$$\frac{e \rightsquigarrow \langle (s_1, \ldots, s_m),\, \mathcal{G}\rangle \qquad f \rightsquigarrow \langle (t_1, \ldots, t_n),\, \mathcal{H}\rangle}{(e \times f) \rightsquigarrow \langle (s_1, \ldots, s_m, t_1, \ldots, t_n),\, \mathcal{G} \cup \mathcal{H}\rangle}$$

$$\frac{e \rightsquigarrow \langle (s_1, \ldots, s_m),\, \mathcal{G}\rangle}{(\Box \circ e) \rightsquigarrow \langle \Box\{1 \mapsto s_1, \ldots, m \mapsto s_m, m{+}1 \mapsto x_1, m{+}2 \mapsto x_2, \ldots\},\, \mathcal{G}\rangle}$$

$$\frac{e \rightsquigarrow \langle (s_1, \ldots, s_m),\, \mathcal{G}\rangle}{(e/\Box) \rightsquigarrow \langle (x_1, \ldots, x_l)\mu,\, \mathcal{G}\mu\rangle}$$

if l is the maximal index occurring in $\Box$ and
$\mu = \mathrm{mgu}((s_1, \ldots, s_m), \Box\{1 \mapsto x_1, \ldots, l \mapsto x_l\})$ exists.

Table 1
Converting language expressions to clause logic

to use results from the areas of logic program transformations and clausal theorem proving.

For any language variable A of arity n, let P_A denote an n-ary predicate symbol uniquely associated with A. The clause corresponding to a constraint is defined as

$$\mathrm{horn}(A \supseteq e) \stackrel{\mathrm{def}}{=} \begin{cases} \{P_A(s_1, \ldots, s_n) \leftarrow \mathcal{G}\} & \text{if } e \rightsquigarrow \langle (s_1, \ldots, s_n),\, \mathcal{G}\rangle \\ \{\} & \text{otherwise} \end{cases}$$

Relation $\rightsquigarrow$ is specified in Table 1; the first-order variables x_i introduced by the rules are assumed to be fresh variables occurring nowhere else. For an inductive definition $\mathcal{D}$, i.e., for a set of constraints, the corresponding logic program is defined as $\mathrm{horn}(\mathcal{D}) \stackrel{\mathrm{def}}{=} \bigcup_{C \in \mathcal{D}} \mathrm{horn}(C)$.

Example 4.1 Consider the following inductive definition where $\Sigma = \{a/0, s/1\}$.

$$X \supseteq [a, 1, 1] \circ \{()\} \qquad X \supseteq [s(1), 2, s(3)] \circ X$$
$$Y \supseteq [a, 1, a] \circ \{()\} \qquad Y \supseteq [s(4), 1, 3] \circ ((X \times Y)/[1, 2, 3, 4, 1, 2])$$

X and Y are the languages $(s^m(a), s^n(a), s^{m+n}(a))$ and $(s^m(a), s^n(a), s^{m*n}(a))$, respectively. The second constraint for Y can be read as

> If $(s^m(a), s^n(a), s^{m+n}(a)) \in X$, $(s^k(a), s^l(a), s^{k*l}) \in Y$, $s^m(a) = s^l(a)$, and $s^n(a) = s^{k*l}(a)$, then $(s^{k+1}(a), s^m(a), s^{m+k*m}(a)) \in Y$.

We obtain the following Horn clauses for the inductive definition:

$$P_X(a, x_1, x_1) \leftarrow \qquad P_X(s(x_1), x_2, s(x_3)) \leftarrow P_X(x_1, x_2, x_3)$$
$$P_Y(a, x_1, a) \leftarrow \qquad P_Y(s(x_4), x_1, x_3) \leftarrow P_X(x_1, x_2, x_3), P_Y(x_4, x_1, x_2)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\top}{\{()\} \rightsquigarrow \langle (), \{\}\rangle}}{(a,1,1) \circ \{()\} \rightsquigarrow \langle (a, x_1, x_1), \{\}\rangle} \qquad \dfrac{\dfrac{\top}{\{()\} \rightsquigarrow \langle (), \{\}\rangle}}{(a,1,a) \circ \{()\} \rightsquigarrow \langle (a, x_1, a), \{\}\rangle}}{\dfrac{\dfrac{\top}{X \rightsquigarrow \langle (x_1,x_2,x_3), \{P_X(x_1,x_2,x_3)\}\rangle}}{(s(1),2,s(3)) \circ X \rightsquigarrow \langle (s(x_1),x_2,s(x_3)), \{P_X(x_1,x_2,x_3)\}\rangle}}{\dfrac{\dfrac{\dfrac{\top}{X \rightsquigarrow \langle (x_1,x_2,x_3), \{P_X(x_1,x_2,x_3)\}\rangle} \qquad \dfrac{\top}{Y \rightsquigarrow \langle (x_4,x_5,x_6), \{P_Y(x_4,x_5,x_6)\}\rangle}}{X \times Y \rightsquigarrow \langle (x_1,x_2,x_3,x_4,x_5,x_6), \{P_X(x_1,x_2,x_3), P_Y(x_4,x_5,x_6)\}\rangle}}{X \times Y / [1,2,3,4,1,2] \rightsquigarrow \langle (x_1,x_2,x_3,x_4,x_1,x_2), \{P_X(x_1,x_2,x_3), P_Y(x_4,x_1,x_2)\}\rangle}}{[s(4),1,3] \circ (X \times Y / [1,2,3,4,1,2]) \rightsquigarrow \langle (s(x_4),x_1,x_3), \{P_X(x_1,x_2,x_3), P_Y(x_4,x_1,x_2)\}\rangle}$$

Table 2
Conversion of sample expressions to clause logic (Example 4.1)

The transformation of the language expressions to Horn logic is given in Table 2.

For every predicate symbol P of arity n, let A_P denote an n-ary language variable uniquely associated with P, and let $\sigma = \{x_1 \mapsto 1, x_2 \mapsto 2, \dots\}$ be some fixed substitution replacing every first-order variable by a unique positive integer. The constraint corresponding to a program clause is defined as

$$\begin{aligned}&\mathrm{indef}(\, P(s_1, s_2, \dots) \leftarrow P_1(t_{11}, t_{12}, \dots), P_2(t_{21}, t_{22}, \dots), \dots)\\ &\stackrel{\text{def}}{=} \{A_P \supseteq [s_1\sigma, s_2\sigma, \dots] \circ ((A_{P_1} \times A_{P_2} \times \cdots) \,/\, [t_{11}\sigma, t_{12}\sigma, \dots, t_{21}\sigma, t_{22}\sigma, \dots])\}\end{aligned}$$

For a logic program, $\mathcal{P}$, the corresponding inductive definition is defined as $\mathrm{indef}(\mathcal{P}) \stackrel{\text{def}}{=} \bigcup_{C \in \mathcal{P}} \mathrm{indef}(C)$. The following proposition states that inductive definitions and logic programs are essentially the same and that the transformations above preserve equivalence with respect to the generated tuple languages.

Proposition 4.2 *Let $\mathcal{D}$ be any inductive definition and $\mathcal{P}$ any logic program.*

(a) $\mathcal{L}_{\mathcal{D}}(A) = \mathcal{M}(\mathrm{horn}(\mathcal{D}))|_{P_A}$ *for every language variable* A.

(b) $\mathcal{M}(\mathcal{P})|_P = \mathcal{L}_{\mathrm{indef}(\mathcal{P})}(A_P)$ *for every predicate symbol* P.

As corollary we obtain that the tuple languages definable by inductive definitions are exactly the recursively enumerable tuple languages.

A program clause $H \leftarrow B_1, \dots, B_k$ is a *cs-clause* if $B_1, \dots, B_k$ is linear and contains no function symbols, i.e., if all arguments of the B_i are variables occurring nowhere else in the body. A cs-clause is called

- *linear* iff the head is linear;
- *horizontal* iff any two head variables that are arguments of the same body atom occur at the same depth in the head;
- *regular* iff the head is linear and for some mapping $\pi\colon \omega \mapsto \omega$, each argument of the head is of the form $f(x_1, \dots, x_{\mathrm{ar}(f)})$, where x_i occurs in a body atom B_j

such that $\pi(j) = i$.

A logic program is a *cs-program* iff all its clauses are cs-clauses. It is linear (horizontal, regular) iff all its clauses are. The next proposition states that the subclasses of inductive definitions correspond to their counterparts in clause logic.

Proposition 4.3

(a) If $\mathcal{D}$ is a (linear, horizontal, or regular) constraint system, then $\mathrm{horn}(\mathcal{D})$ is a (linear, horizontal, or regular) cs-program.

(b) If $\mathcal{P}$ is a (linear, horizontal, or regular) cs-program then $\mathrm{indef}(\mathcal{P})$ is a (linear, horizontal, or regular) constraint system.

5 Transforming Logic Programs to CS-Programs

This section presents two rules, *unfolding* and *definition introduction*, that transform logic programs to equivalent cs-programs; they are particular instances of rules studied in the field of logic program transformation [14]. Typically the starting point of the transformation is a cs-program satisfying properties like linearity, and a single non-conforming clause that specifies e.g. the intersection of two sets. If the transformation process terminates the resulting cs-program represents the intersection by a cs-program of a particular kind.

The rules transform states $\langle \mathcal{P}, \mathcal{D}_{\mathrm{new}}, \mathcal{D}_{\mathrm{done}}, \mathcal{C}_{\mathrm{new}}, \mathcal{C}_{\mathrm{out}} \rangle$ where $\mathcal{P}$ is a logic program that remains unchanged, $\mathcal{D}_{\mathrm{new}}$ are definitions not yet unfolded, $\mathcal{D}_{\mathrm{done}}$ are definitions already processed but still used for simplifying clauses, $\mathcal{C}_{\mathrm{new}}$ are clauses generated from definitions by unfolding, and $\mathcal{C}_{\mathrm{out}}$ are the cs-clauses generated so far. Syntactically, definitions are written as clauses, but from the semantic point of view they are equivalences. A set of definitions, $\mathcal{D}$, is *compatible with* $\mathcal{P}$, if all predicate symbols occurring in the heads of the definitions occur in exactly one head and nowhere else in $\mathcal{D}$ and $\mathcal{P}$; the only exception are tautological definitions of the form $P(\boldsymbol{x}) \leftarrow P(\boldsymbol{x})$ where P may occur without restrictions throughout $\mathcal{D}$ and $\mathcal{P}$. The predicate symbols in the heads of $\mathcal{D}$ are called the *predicate symbols defined by* $\mathcal{D}$. Tautological definitions are convenient to trigger the transformation of the clauses defining P without having to introduce a new predicate symbol; the alternative would be to replace the tautology by $P'(\boldsymbol{x}) \leftarrow P(\boldsymbol{x})$ for some new predicate symbol P'.

We write $S \Rightarrow S'$ if S' is a state obtained from state S by one application of rule *unfolding* or rule *definition introduction* (see below for their description). The reflexive and transitive closure of $\Rightarrow$ is denoted by $\stackrel{*}{\Rightarrow}$. An *initial state* is of the form $\langle \mathcal{P}, \mathcal{D}, \emptyset, \emptyset, \emptyset \rangle$ where $\mathcal{D}$ is compatible with $\mathcal{P}$; a *final state* is of the form $\langle \mathcal{P}, \emptyset, \mathcal{D}', \emptyset, \mathcal{P}' \rangle$. $\mathcal{P}$ and $\mathcal{D}$ are called the input of a derivation, $\mathcal{P}'$ its output. A derivation is called *complete* if its last state is a final one.

Unfolding.

Pick a definition not yet processed, select one or more of its body atoms (according to some selection rule), and unfold them with all matching clauses from

the input program. Formally:

$$\frac{\langle \mathcal{P},\, \mathcal{D}_{\text{new}} \,\dot\cup\, \{L \leftarrow \mathcal{R}\dot\cup\{A_1,\ldots,A_k\}\},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}},\, \mathcal{C}_{\text{out}}\rangle}{\langle \mathcal{P},\, \mathcal{D}_{\text{new}},\, \mathcal{D}_{\text{done}} \cup \{L \leftarrow \mathcal{R}\cup\{A_1,\ldots,A_k\}\},\, \mathcal{C}_{\text{new}} \cup \mathcal{C},\, \mathcal{C}_{\text{out}}\rangle}$$

where

$$\mathcal{C} = \{\, (L \leftarrow \mathcal{R}\cup\mathcal{B}_1\cup\cdots\cup\mathcal{B}_k)\mu \mid H_i{\leftarrow}\mathcal{B}_i \in \mathcal{P} \text{ for } i = 1,\ldots,k \text{ such that } \mu = \text{mgu}((A_1,\ldots,A_k),(H_1,\ldots,H_k)) \text{ exists} \,\}$$

Note that as usual prior to unfolding the clauses from $\mathcal{P}$ have to be properly renamed such that they share variables neither with each other nor with $L \leftarrow \mathcal{R} \cup \{A_1,\ldots,A_k\}$.

Definition introduction.

Pick a clause not yet processed, decompose its body into minimal variable-disjoint components, and replace every component that is not yet a single linear atom without function symbols by an atom that is either looked up in the set of old definitions, or if this fails is built of a new predicate symbol and the component variables. For every failed lookup introduce a new definition associating the new predicate symbol with the replaced component. Formally:

$$\frac{\langle \mathcal{P},\, \mathcal{D}_{\text{new}},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}} \,\dot\cup\, \{H \leftarrow \mathcal{B}_1\dot\cup\cdots\dot\cup\mathcal{B}_k\},\, \mathcal{C}_{\text{out}}\rangle}{\langle \mathcal{P},\, \mathcal{D}_{\text{new}} \cup \mathcal{D},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}},\, \mathcal{C}_{\text{out}} \cup \{H \leftarrow L_1,\ldots,L_k\}\rangle}$$

where $\mathcal{B}_1,\ldots,\mathcal{B}_k$ is a maximal decomposition of $\mathcal{B}_1 \cup \cdots \cup \mathcal{B}_k$ into non-empty variable-disjoint subsets,

$$L_i = \begin{cases} L\eta^{-1} & \text{if } L{\leftarrow}\mathcal{B}_i\eta \in \mathcal{D}_{\text{done}} \text{ for some variable renaming } \eta \\ P_i(x_1,\ldots,x_n) & \text{otherwise, with } \{x_1,\ldots,x_n\} = \text{var}(\mathcal{B}_i) \end{cases}$$

for $1 \le i \le k$ and new predicate symbols P_i, and where $\mathcal{D}$ is the set of all $L_i{\leftarrow}\mathcal{B}_i$ such that L_i contains a new predicate symbol. [7]

Example 5.1 Let $\mathcal{P}$ be the cs-program

$$\begin{array}{ll} M(x,a,x) \leftarrow & E(a) \leftarrow \\ M(s(x),s(y),z) \leftarrow M(x,y,z) & E(s(s(x))) \leftarrow E(x) \end{array}$$

and let $\mathcal{D} = \{R(x,z) \leftarrow M(x,y,z), E(y)\}$. Predicate M defines subtraction (minus), E evenness, and R defines all pairs of numbers with an even difference. Note that the definition is no cs-clause since its body atoms share variables.

[7] A substitution η is a variable renaming for a set of atoms $\mathcal{R}$, if there exists a substitution η^{-1} such that $\mathcal{R}\eta\eta^{-1} = \mathcal{R}$.

$\mathcal{D}_{\text{new}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	
$R(x,y,z) \leftarrow$ $M(x,y,z), E(y)$			U_M
	$R(x,a,x) \leftarrow E(a)$ $R(s(x),s(y),z) \leftarrow$ $M(x,y,z), E(s(y))$		D
$E' \leftarrow E(a)$	$R(s(x),s(y),z) \leftarrow$ $M(x,y,z), E(s(y))$	$R(x,a,x) \leftarrow E'$	U_E
	$E' \leftarrow$ $R(s(x),s(y),z) \leftarrow$ $M(x,y,z), E(s(y))$		D
	$R(s(x),s(y),z) \leftarrow$ $M(x,y,z), E(s(y))$	$E' \leftarrow$	D
$R'(x,y,z) \leftarrow$ $M(x,y,z), E(s(y))$		$R(s(x),s(y),z) \leftarrow$ $R'(x,y,z)$	U_E
	$R'(x,s(y),z) \leftarrow$ $M(x,s(y),z), E(y)$		D
$R''(x,y,z) \leftarrow$ $M(x,s(y),z), E(y)$		$R'(x,s(y),z) \leftarrow$ $R''(x,y,z)$	U_M
	$R''(s(x),y,z) \leftarrow$ $M(x,y,z), E(y)$		D
		$R''(s(x),y,z) \leftarrow$ $R(x,y,z)$	

Table 3
Transformation of a sample program to cs-clauses (Example 5.1)

Table 3 gives a complete derivation starting with input $\mathcal{P}$ and $\mathcal{D}$. We omit $\mathcal{D}_{\text{done}}$ since it consists just of the definitions in $\mathcal{D}_{\text{new}}$. Moreover, we list definitions and clauses in columns $\mathcal{D}_{\text{new}}$ and $\mathcal{C}_{\text{out}}$ only once in the step that adds them. Unfolding always selects the leftmost atom of maximal term depth. The third column, $\mathcal{C}_{\text{out}}$, lists the generated cs-clauses. The last column gives the applied rule: U_P means unfolding the leftmost atom of maximal depth with the clauses for predicate P, and D means definition introduction applied to the first clause in $\mathcal{C}_{\text{new}}$.

Theorem 5.2 (Correctness) *Let $\mathcal{P}$ be a logic program and $\mathcal{D}$ be a set of definitions compatible with $\mathcal{P}$. If $\langle \mathcal{P}, \mathcal{D}, \emptyset, \emptyset, \emptyset \rangle \stackrel{*}{\Rightarrow} \langle \mathcal{P}, \emptyset, \mathcal{D}', \emptyset, \mathcal{P}' \rangle$, then $\mathcal{P}'$ is a cs-program with the property that $\mathcal{M}(\mathcal{P}')|_P = \mathcal{M}(\mathcal{P} \cup \mathcal{D})|_P$ for all predicate symbols P defined by $\mathcal{D}$.*

Remark 5.3 For any state derivable from an initial state, $\mathcal{D}_{\text{new}} \cup \mathcal{D}_{\text{done}}$ is compatible with $\mathcal{P} \cup \mathcal{C}_{\text{new}}$. This means in particular that newly introduced predicate symbols do not occur in the bodies of definitions and clauses in $\mathcal{D}_{\text{new}}$, $\mathcal{D}_{\text{done}}$, and $\mathcal{C}_{\text{new}}$.

Remark 5.4 In general cs-programs generated by $\Rightarrow$-derivations contain unproductive clauses, i.e., clauses that do not contribute to minimal models. Such clauses can be removed by the following procedure. Starting with the facts, mark each clause $P(\boldsymbol{s}) \leftarrow Q_1(\boldsymbol{t}_1), \ldots, Q_k(\boldsymbol{t}_k)$ as productive and its predicate symbol P as non-empty provided that all Q_i have been marked as non-empty before. Repeat the process until no more clauses and predicate symbols can be marked. Clauses not

marked productive can be removed without affecting the semantics of the program. In fact this pruning of clauses is sufficient to test for emptiness of minimal models: the set of clauses is empty after removal of unproductive clauses if and only if the minimal Herbrand model is empty. As corollary, the emptiness problem of constraint systems is decidable.

Remark 5.5 To compute a cs-program equivalent to an arbitrary logic program, $\mathcal{P}$, let $\mathcal{D}_\mathcal{P}$ be the set of all tautologies $P(\boldsymbol{x})\leftarrow P(\boldsymbol{x})$ such that P occurs in $\mathcal{P}$. The output of every complete derivation starting from $\mathcal{P}$ and $\mathcal{D}$ is a cs-program equivalent to $\mathcal{P}$ on all its predicates.

6 Termination and Other Properties

An algorithm that is able to compute cs-programs for arbitrary programs has to loop necessarily on certain inputs. If it would not we could use it to decide emptiness of minimal Herbrand models for logic programs, known to be an undecidable problem: just compute an equivalent cs-program and test its minimal model for emptiness according to Remark 5.4 above. In fact, the rules presented in the last section do not terminate for quite simple inputs; take e.g. program $\{P(x) \leftarrow P(s(x))\}$ and definition $P(x)\leftarrow P(x)$. However, in this section we will show that the derivation process terminates for several interesting classes. The point is that the same rules cope with all of them, i.e., we have a uniform approach to solve various problems considered in the area of tree tuple grammars and constraint systems.

A derivation is bound to be finite if from some point onwards no further definitions are added to $\mathcal{D}_{\text{new}}$ because all required definitions are already there. To show this property for all programs of a class it is sufficient to prove that the variable-disjoint subsets $\mathcal{B}_i$ in rule *definition introduction* satisfy two conditions:

- the number of atoms in $\mathcal{B}_i$ is bounded;
- the maximal depth of atoms in $\mathcal{B}_i$ is bounded.

In this case there is only a finite number of potential $\mathcal{B}_i$'s and therefore also only a finite number of potential definitions up to variable renaming. Note that according to Remark 5.3 the $\mathcal{B}_i$'s are built over the original signature, i.e., the number of occurring predicate symbols does not grow.

The following class is a generalization of cs-programs that allows function symbols also in the body of clauses.

Definition 6.1 A clause is *quasi-cs*, if the body is linear and for every variable that occurs both in the body and the head, the depth of its occurrence in the body is smaller than or equal to the depth of all occurrences in the head. A program is quasi-cs if all its clauses are.

Every quasi-cs program can be transformed to an equivalent finite cs-program.

Theorem 6.2 *Let $\mathcal{P}$ be a quasi-cs program, and let $\mathcal{D}_\mathcal{P}$ be the set of all tautologies $P(\boldsymbol{x})\leftarrow P(\boldsymbol{x})$ such that P occurs in $\mathcal{P}$. Any $\Rightarrow$-derivation with input $\mathcal{P}$ and $\mathcal{D}_\mathcal{P}$ is*

finite.

Corollary 6.3 *The membership test for constraint systems as well as the test for the emptiness of linear filter operations is decidable.*

A clause (or definition) $H \leftarrow B_1, \ldots, B_k$ is called a *join-clause* (or *join-definition*) if it has the following properties:

- The atoms $H, B_1, \ldots, B_k$ are linear and do not contain function symbols.
- The variables of H are among those of the body.

Join-clauses are in general no cs-clauses since the B_i's may share variables, i.e., the body of a join-clause need not be linear though each atom is. This property allows to represent the intersection or join of two or more tuple languages as join-clauses.

The next theorem shows that $\Rightarrow$-derivations can be used to compute regular cs-programs for the intersection of regular tuple languages.

Theorem 6.4 *Let $\mathcal{P}$ be a regular cs-program, and let D be a join-definition compatible with $\mathcal{P}$. Then any complete derivation with input $\mathcal{P}$ and $\{D\}$ that unfolds in each unfolding step all atoms simultaneously is finite and its output is a regular cs-program.*

Corollary 6.5 *Tuple languages defined by regular constraint systems are closed under intersection and joins. A regular constraint system representing the result of the operation can be computed via $\Rightarrow$-derivations.*

The closure properties of regular constraint systems are well-known facts [1]. But the corollary shows that our approach subsumes these results which one should expect of a general framework as we claim logic programs and $\Rightarrow$-derivations are. Moreover we can use the same algorithm for all sorts of joins and (partial) intersections.

7 Related Work

The tight connection between set constraints and tree automata on the one hand and logic programs on the other is quite natural and is indeed used frequently in one way or another (see e.g. [5,16]).

More interesting parallels to our work can be found in the area of type inference and type checking for logic programs. The type of a variable in any programming language can be defined as the set of values the variable may take. In logic programming this leads to types given by sets of ground first-order terms. Frühwirth et al. [4] show that the types for a logic program can be constructed and described systematically as unary-predicate programs consisting of clauses $H(t) \leftarrow P_1(t_1), \ldots, P_n(t_n)$ where t is linear, and each t_i is either a subterm of t, a strong superterm of t, or variable-disjoint from t. Unfolding techniques simplify these type programs to regular unary-predicate programs which admit type checking (i.e., membership tests). The approach partially extends to higher-order terms [7] and AC tree automata [8].

A main difference between type checking in logic programming and our work is the arity of the languages and predicates considered. While we are interested in tree tuple languages of arbitrary arity and in the interaction of tuple components, types are unary languages. As a consequence our cs-programs may contain predicates of any arity, but require linear clause bodies. Unary-predicate symbols on the other hand are compatible with shared variables and nested terms.

8 Conclusion

The transformation rules presented in this paper provide a uniform framework to handle tree tuple languages: on the practical side we obtain a single algorithm for computing with tree tuple languages, [8] on the theoretical side the proof of closure properties for classes of tuple languages reduces to giving bounds on the length and depth of the generated clauses.

Our work might also be viewed as contributing to two other fields: clausal model building and logic program transformation. Starting from failed proof attempts clausal model building constructs finite presentations of counter-models like sets of atoms, and to compute with these presentations (see e.g. [3,10]). Our algorithm transforms logic programs to cs-programs, which are presentations of the minimal Herbrand model; operations like testing for membership correspond to operations on models like checking the validity of an atom. Regarding program transformations, our algorithm is an instance of the 'rules+strategy' approach [14], and the termination results constitute a successful application of the principles of logic program transformation to the field of tree tuple languages.

The results in the last section are just the beginning. Further properties can be proved in a similar way, like closure properties of weakly regular relations [15]. We believe that these results can be generalized to new classes of cs-programs corresponding to new classes of constraint systems. Though we focused on constraint systems, our approach also applies to other tree tuple formalisms like tree automata with equality constraints. This could allow to mix different classes of tree tuple languages in a single scheme.

References

[1] Comon, H., M. Dauchet, R. Gilleron, D. Lugiez, S. Tison and M. Tommasi, "Tree Automata Techniques and Applications (TATA)," `http://www.grappa.univ-lille3.fr/tata`, 1997.

[2] Dauchet, M. and S. Tison, *Structural complexity of classes of tree languages*, in: M. Nivat and A. Podelski, editors, *Tree Automata and Languages*, North-Holland, Amsterdam, 1992 pp. 327–353.

[8] A prototype implementation of the algorithm in Prolog is available from `http://www.logic.at/css/`.

[3] Fermüller, C. and A. Leitsch, *Hyperresolution and automated model building*, Journal of Logic and Computation **2** (1996), pp. 173–203.

[4] Frühwirth, T. W., E. Y. Shapiro, M. Y. Vardi and E. Yardeni, *Logic programs as types for logic programs*, in: *Logic in Computer Science*, 1991, pp. 300–309.

[5] Gallagher, J. P. and G. Puebla, *Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs*, in: *4th International Symposium, PADL 2002*, LNCS **2257** (2002), pp. 243–261.

[6] Gilleron, R., S. Tison and M. Tommasi, *Set constraints and automata*, Information and Computation **1** (1999), pp. 1–41.

[7] Goubault-Larrecq, J., *Higher-order positive set constraints*, in: *Proc. 16th Int. Workshop Computer Science Logic (CSL'2002), Edinburgh, Scotland, Sep. 2002*, LNCS **2471** (2002), pp. 473–489.

[8] Goubault-Larrecq, J. and K. N. Verma, *Alternating two-way AC-tree automata*, Research Report LSV-02-11, Lab. Specification and Verification, ENS de Cachan, Cachan, France (2002), 21 pages.

[9] Gouranton, V., P. Réty and H. Seidl, *Synchronized tree languages revisited and new applications*, in: *Proceedings of 6th Conference on Foundations of Software Science and Computation Structures, Genova (Italy)*, LNCS **2030** (2001), pp. 214–229.

[10] Leitsch, A., *Decision procedures and model building, or how to improve logical information in automated deduction.*, in: R. Caferra and G. Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, LNCS **1761** (2000), pp. 62–79.

[11] Limet, S. and P. Réty, *E-unification by means of tree tuple synchronized grammars*, Discrete Mathematics and Theoretical Computer Science **1** (1997), pp. 69–98.

[12] Lloyd, J., "Foundations of Logic Programming," Springer Verlag, 1984.

[13] Matzinger, R., "Computational Representations of Models in First-Order Logic," Dissertation, Technische Universität Wien, Austria (2000).

[14] Pettorossi, A. and M. Proietti, *Transformation of logic programs*, Handbook of Logic in Artificial Intelligence and Logic Programming **5**, Oxford University Press, 1998 pp. 697–787.

[15] Réty, P., "Langages synchronisés d'arbres et applications," Habilitation thesis (in French), LIFO, Université d'Orléans (2001).

[16] Saubion, F. and I. Stéphan, *A unified framework to compute over tree synchronized grammars and primal grammars*, Discrete Mathematics and Theoretical Computer Science **5** (2002), pp. 227–262.

A Resolution-based Model Building Algorithm for a Fragment of $\mathcal{OCC}1\mathcal{N}_{=}$ (Extended Abstract)

Nicolas Peltier

Centre National de la Recherche Scientifique
Laboratoire LEIBNIZ-IMAG
46, Avenue Félix Viallet 38031 Grenoble Cedex - FRANCE
E-mail: Nicolas.Peltier@imag.fr
Phone: (33) 4 76 57 48 05

Abstract

$\mathcal{OCC}1\mathcal{N}$ [11] is a decidable subclass of first-order clausal logic without equality. [7] shows that $\mathcal{OCC}1\mathcal{N}$ becomes undecidable when equational literals are allowed, but remains decidable if equality is restricted to ground terms only.

First, we extend this decidability result to some non ground equational literals. By carefully restricting the use of the equality predicate we obtain a new decidable class, called $\mathcal{OCC}1\mathcal{N}^*_{=}$. We show that existing paramodulation calculi do not terminate on $\mathcal{OCC}1\mathcal{N}^*_{=}$ and we define a new simplification rule which allows to ensure termination. Second, we show that the automatic extraction of Herbrand models is possible from saturated sets in $\mathcal{OCC}1\mathcal{N}^*_{=}$ not containing $\Box$. These models are represented by certain finite sets of (possibly equational and non ground) linear atoms. The difficult point here is to show that this formalism is suitable as a model representation mechanism, i.e. that the evaluation of *arbitrary* non equational first-order formulae in such interpretations is a decidable problem.

1 Introduction

Since the satisfiability problem is undecidable (semi-decidable) for first order logic, identifying syntactic subclasses for which this problem is decidable is a major issue [6]. Traditionally, most works in this field were dealing with classes of prenex first order formulae defined by syntactic conditions on the quantifier prefix and/or on the matrix. Then, with the development of the Resolution method [25], some attention has been payed to clausal classes, i.e. classes of formulae in conjunctive normal form, without any existential quantifier, but possibly containing function symbols. [18] showed that the resolution calculus may be used as a decision procedure for several classes of clause sets. The idea is to exhibit (refutationally complete) refinements of the resolution calculus (using for example ordering restrictions and/or

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

selection functions) and to show that these refinements *terminate* on the considered classes. This approach has the advantage that efficient and uniform decision procedures can be obtained with little programming effort, simply by using existing theorem provers. The reader may consult [11,16,10] for exemplification of this technique for various kinds of resolution refinements. In particular this principle has been used to prove the decidability of several interesting classes using the hyperesolution rule [24] as a decision procedure. Such classes include $\mathcal{PVD}$ (**P**ositively **V**ariable **D**ominated), $\mathcal{KPOD}$ (**K**rom **P**ositively **O**ccurrence **D**ominated), or more generally all T-dominated classes [19]. We can also mention the recent class $\mathcal{BU}$, which is an extension of classes of clause sets obtained by translation from modal logics into first-order clausal logic (see [13] and also [14,17]).

This technique can be extended to first-order logic with equality, using refinements of the paramodulation calculus [1]. For example [2] presents a decision procedure for the monadic class with equality based on refinements of the superposition calculus. In [9,26,23], decidable extensions of $\mathcal{PVD}$ to equational logic are presented. Nevertheless, still only little is known about termination of paramodulation calculi. In most of the cases, the termination results do not extend to the equational case, and even if the class is still decidable, sophisticated refinements are needed to ensure termination. On the other hand, the use of the equality predicate is mandatory for many applications.

The class $\mathcal{OCC}1\mathcal{N}$ [11] is a class of clause sets which is decidable via hyperresolution. An interesting feature of $\mathcal{OCC}1\mathcal{N}$ is that it may contain clauses that are **not** range-restricted, i.e. that contain variables not occurring in the negative literals (such as, for example, the clause $\neg R(x) \vee P(x, y)$). This property is not shared by the other classes on which hyperresolution is currently known to terminate (such that T-dominated classes, $\mathcal{PVD}$, $\mathcal{KPOD}$, $\mathcal{BU}$, etc.) and makes the termination proof more difficult (since hyperresolution may generate non ground clauses, condensing is *needed* to decide $\mathcal{OCC}1\mathcal{N}$ whereas it is useless for the other classes such as $\mathcal{PVD}$ or $\mathcal{BU}$). As a consequence, this implies that the hyperresolution rule may generate *non ground clauses* (this is not the case if all the clauses are range-restricted). This makes the termination proof more difficult (in particular, condensing is *needed* to decide $\mathcal{OCC}1\mathcal{N}$, whereas it is useless for the other classes such as $\mathcal{PVD}$ or $\mathcal{BU}$). In [7], it is shown that $\mathcal{OCC}1\mathcal{N}$ becomes undecidable if equational literals are allowed, but that the class remains decidable if only *ground* equations are considered. The class of $\mathcal{OCC}1\mathcal{N}$ clause sets with ground equality is called $\mathcal{OCC}1\mathcal{N}^{g}_{=}$.

In this paper, we consider a new decidable extension of $\mathcal{OCC}1\mathcal{N}$, called $\mathcal{OCC}1\mathcal{N}^{*}_{=}$, strictly containing $\mathcal{OCC}1\mathcal{N}^{g}_{=}$, in which certain non ground equations are considered. We show that existing refinements of paramodulation or superposition calculi do not terminate on $\mathcal{OCC}1\mathcal{N}^{*}_{=}$ and we provide a new simplification rule which allows to ensure termination. Since this rule preserves refutational completeness, this entails that $\mathcal{OCC}1\mathcal{N}^{*}_{=}$ is decidable. Then, we show how to extract models from satisfiable saturated clause sets in $\mathcal{OCC}1\mathcal{N}^{*}_{=}$. The interpretations are built on the Herbrand universe and are specified by sets of linear atoms, called

EEAR (**E**lementary **E**quational **A**tomic **R**epresentations). We show that the evaluation of non equational first-order formulae in an interpretation specified by a EEAR is effectively decidable by reducing this problem to the emptiness problem for finite tree automata [5], which makes this representation mechanism suitable for applications.

2 Preliminaries

In this section we introduce the necessary notions and notations. We assume that the reader is familiar with the basic definitions and with the usual terminology in Logic and Automated Deduction (see for example [21,12]).

The sets of *terms* and *atoms* are built on a set of function symbols Σ, on a set of predicate symbols Ω and on a set of variables $\mathcal{X}$. We assume that $\Omega, \Sigma, \mathcal{X}$ share no element and that Ω contains the equality predicate, that is denoted by $\approx$ in order to avoid confusion with semantic equality (in infix notation).

In this paper, we often use vectors for simplifying notations. For example, $f(\boldsymbol{t})$ denotes a term of the form $f(t_1, \ldots, t_n)$ where $\boldsymbol{t} = (t_1, \ldots, t_n)$.

A *literal* is either an atom (positive literal) or the negation of an atom (negative literal). A negative literal of the form $\neg(t \approx s)$ is usually denoted by $t \not\approx s$. A literal (atom) that is of the form $t \approx s$ or $t \not\approx s$ is said to be *equational*.

A *clause* is a finite multiset of literals (often denoted as a disjunction). If C is a clause, then C^+ (resp. C^-) denotes the set of positive (resp. negative) literals in C. C_e denotes the set of equational literals in C. A clause is said to be *equational* if $C_e \neq \emptyset$, non equational otherwise.

The notion of *substitution* is defined as usual. The image of a term, atom, literal etc. t by a substitution σ is denoted by $t\sigma$.

If E is an expression (term, atom, clause, etc.), then $\mathit{Var}(E)$ denotes the set of variables occurring in E. An expression is called *ground* if it contains no variables.

We introduce the notion of *condensing*, which will be crucial for the termination of the calculus.

Definition 2.1 Let C be a clause of the form $\bigvee_{i=1}^{n} L(\boldsymbol{t}_i) \vee R$ (with $n \geq 2$) such that θ is a m.g.u. of $\{(\boldsymbol{t}_1, \boldsymbol{t}_i) \mid i \in [2..n]\}$. The clause $D = L(\boldsymbol{t}_1)\theta \vee R\theta$ is a *factor* of C (if D is a factor of C then any factor of D is also a factor of C).

A clause C is called *condensed* if there exists no factor of C which is a subclause of C. If C' is a condensed factor of C s.t. $C' \subseteq C$ then C' is called a *condensation* of C. Condensations are unique up to renaming (see [18]).

The notions of interpretations, models, satisfiability etc. are defined as usual.

A *position* is a finite sequence of natural number. The empty position is denoted by ϵ and the concatenation of two positions p and q is denoted by $p.q$. If p is a position then $|p|$ denotes the length of p. A position p is said to *occur in* a term or atom t if p is ϵ or if $p = i.q$ and t is of the form $f(t_1, \ldots, t_n)$ where $i \in [1..n]$ and q is a position in t_i. The set of positions occurring in t is denoted by $\mathit{Pos}(t)$. Let t, s be two terms (or atoms), let $p \in \mathit{Pos}(t)$. $t_{|p}$ denotes the term (atom) occurring

at position p in t and $t[s]_p$ denotes the term (atom) obtained by replacing the term at position p in t by s. The notion of position may be extended to negative literals by the relations: $\mathit{Pos}(\neg L) \stackrel{def}{=} \mathit{Pos}(L)$, $(\neg L)_{|p} \stackrel{def}{=} \neg L_{|p}$ and $(\neg L)[t]_p \stackrel{def}{=} \neg L[t]_p$.

3 Definition of the decidable class

In this section we give the definition of $\mathcal{OCC}1\mathcal{N}^*_{=}$. We need to introduce a few additional definitions.

Let t be a term (or atom) and x be a variable. $\mathit{Occ}(x,t)$ denotes the set of occurrences of x in t, i.e. the set of positions p such that $t_{|p} = x$. $\mathit{Occ}(x,t)$ is extended to negative literals by the relation $\mathit{Occ}(x,\neg A) \stackrel{def}{=} \mathit{Occ}(x,A)$. If C is a clause then $\mathit{Occ}(x,C)$ denotes the *multiset* of positions p such that there exists $L \in C$ with $\mathit{Occ}(x,L) = p$. For ex. $\mathit{Occ}(x, p(x) \vee q(x)) = \{1,1\}$.

An expression (term, clause, etc.) t is said to be *linear* iff for all variables x, $|\mathit{Occ}(x,t)| \leq 1$.

$\tau_{min}(x,C)$ and $\tau_{max}(x,C)$ denote respectively the minimal and maximal depth of the occurrences of x in C. More formally we have $\tau_{min}(x,C) \stackrel{def}{=} \min\{|p| \mid p \in \mathit{Occ}(x,C)\}$ and $\tau_{max}(x,C) \stackrel{def}{=} \max\{|p| \mid p \in \mathit{Occ}(x,C)\}$.

For any expression E, $\tau(E)$ denotes the depth of E (if t is a term, then $\tau(t) \stackrel{def}{=} \max\{|p| \mid p \in \mathit{Pos}(t)\}$).

We firstly recall the definition of the class $\mathcal{OCC}1\mathcal{N}$, originally defined for clause sets without equality [11].

$\mathcal{OCC}1\mathcal{N}$ is the set of all sets of clauses S such that for all $C \in S$:

(i) $|\mathit{Occ}(x,C^+)| \leq 1$ for all $x \in \mathcal{X}$ and

(ii) $\tau_{max}(x,C^+) \leq \tau_{min}(x,C^-)$ for all $x \in \mathit{Var}(C^+) \cap \mathit{Var}(C^-)$.

$\mathcal{OCC}1\mathcal{N}$ [11] is defined by the following two conditions: for any clause C, we must have $|\mathit{Occ}(x,C^+)| \leq 1$, for all $x \in \mathcal{X}$ (i.e. there is at most one occurrence of each variable in the positive part of C) and $\tau_{max}(x,C^+) \leq \tau_{min}(x,C^-)$, for all $x \in \mathit{Var}(C^+) \cap \mathit{Var}(C^-)$ (i.e. for any variable x occurring in the negative part of C, all the occurrences of x in the positive part of C must be of lower depth than the occurrences of x in the negative part of C). In the non equational case $\mathcal{OCC}1\mathcal{N}$ is decidable and hyperresolution (with condensing) terminates on $\mathcal{OCC}1\mathcal{N}$. However, in the equational case, additional restrictions are needed for ensuring decidability. [7] shows that it is sufficient to restrict the use of the equality predicate to ground terms only. In this paper, we propose a less restrictive criteria. We observe that the problem is related to the reflexivity axiom $(\forall x)(x \approx x)$ which allows to impose equality conditions on the variables occurring in C^+. The following example will help to clarify this point and will give an intuition of why the class becomes undecidable if equational literals are allowed.

Example 3.1 We consider the following set of clauses:

$$\{\neg p(x) \vee x \not\approx f(y) \vee p(y), p(a)\}$$

The reader can easily check that the clauses in S namely $\neg p(x) \vee x \not\approx f(y) \vee p(y)$ and $p(a)$ fulfill the above conditions hence S belongs to $\mathcal{OCC}1\mathcal{N}$. However, after applying the resolution rule with the literal $x \not\approx f(y)$ and the reflexivity axiom $x \approx x$ we get the clause:

$$\neg p(x) \vee p(f(x))$$

that is not in $\mathcal{OCC}1\mathcal{N}$ (since $\tau_{max}(x, p(f(x))) = 2 > \tau_{min}(x, p(x)) = 1$). Clearly, using this last clause and the clause $p(a)$ we can generate an infinite number of distinct clauses of the form $p(f^n(a))$ (for $n \in \mathbb{N}$). This shows that positive resolution calculi do not terminate on $\mathcal{OCC}1\mathcal{N}$ in the equational case.

This principle can be generalized: actually it is possible to show that *any* clause set S can be transformed into an equivalent clause set in $\mathcal{OCC}1\mathcal{N}$ (possibly containing equational literals). This is done by linearizing and flattening the terms when needed. The corresponding additional conditions on the variables are expressed by adding new equational literals.

For instance the clause $\neg p(x) \vee q(x, f(x))$ which is not in $\mathcal{OCC}1\mathcal{N}$, may be transformed into the clause: $\neg T(p(x)) \vee T(q(x, y)) \vee y \not\approx f(x)$.

Thus, we propose to restrict the class by forbidding such conditions. This is done by introducing the notion of safe literals: A literal L is said to be *unsafe* iff it is of the form $t \not\approx s$ where t, s are non ground. If C is a clause, then C^u denotes the set of unsafe literals in C. If a negative literal $t \not\approx s$ is safe then the reflexivity rule cannot be applied in a non trivial way on $t \not\approx s$. Indeed, since either t or s is ground, all the variables in $t \not\approx s$ will be instantiated by ground terms occurring in the original set of clauses.

We introduce the following definition:

Definition 3.2 $\mathcal{OCC}1\mathcal{N}^*_{=}$ is the set of all sets of clauses S such that for all $C \in S$:

(i) $|\mathit{Occ}(x, C^+)| \leq 1$ for all $x \in \mathcal{X}$ and

(ii) $\tau_{max}(x, C^+) \leq \tau_{min}(x, C^-)$ for all $x \in \mathit{Var}(C^+) \cap \mathit{Var}(C^-)$ and

(iii) $\mathit{Var}(C^+) \cap \mathit{Var}(C^u) = \emptyset$ and

(iv) $\mathit{Var}(C^+_e) \cap \mathit{Var}(C^-) = \emptyset$.

Note that in the non-equational case, we have $C_e = C^u = \emptyset$, thus Definition 3.2 coincides with the usual definition of $\mathcal{OCC}1\mathcal{N}$ in this case.

We shall prove that the satisfiability problem is decidable for $\mathcal{OCC}1\mathcal{N}^*_{=}$. In the next section, we introduce the calculus that is used for this purpose.

4 The calculus

We use a hyperresolution calculus with additional rules to handle equational literals.

Ordering: We assume given a reduction ordering $<$ which is total on ground terms. $<$ is extended to literals using the following relation: $L_1 < L_2 \Leftrightarrow \phi(L_1) <_{mult} \phi(L_2)$, where $<_{mult}$ denotes the multiset extension of $<$. ϕ is defined as follows: $\phi(t \approx s) \stackrel{def}{=} \{\{t, s\}\}$ and $\phi(t \not\approx s) \stackrel{def}{=} \{\{t\}, \{s\}\}$. $<$ is extended to clauses using the multiset extension of the ordering on literals. A literal L is said to be *maximal* in a clause C iff for all literals $L' \in C$, $L \not< L'$.

We use the following inference rules.

Hyper resolution

$$\frac{\bigvee_{i=1}^{n} \neg p_i(\boldsymbol{t}_i) \vee R \qquad p_1(\boldsymbol{s}_1) \vee R_1, \ldots, p_n(\boldsymbol{s}_n) \vee R_n}{(\bigvee_{i=1}^{n} R_i \vee R)\theta}$$

If: θ is the[1] m.g.u. of $\{(\boldsymbol{t}_i, \boldsymbol{s}_i) \mid i \in [1..n]\}$, R is positive and for all i, R_i is positive and $p_i(\boldsymbol{s}_i)\theta$ is maximal in $(p_i(\boldsymbol{s}_i) \vee R_i)\theta$.

Paramodulation (1)

$$\frac{L \vee R \qquad t \approx s \vee R'}{(L[s]_p \vee R \vee R')\theta}$$

If: p is a non variable position in L, θ is the m.g.u. of $\{(L_{|p}, t)\}$, $t\theta \not< s\theta$, $L \vee R$, R' are positive, $L\theta$ and $(t \approx s)\theta$ are respectively maximal in $(L \vee R)\theta$ and $(t \approx s \vee R')\theta$.

Paramodulation (2)

$$\frac{\neg L \vee R \qquad t \approx s \vee R'}{(\neg L[s]_p \vee R \vee R')\theta}$$

If: p is a non variable position in L,θ is the m.g.u. of $L_{|p}$ and t, $t\theta \not< s\theta$, R' is positive, $(t \approx s)\theta$ is maximal in $(t \approx s \vee R')\theta$.

Remark 4.1 Note that additional, more restrictive, conditions could be added. For instance, we could require that $L\theta$ is the maximal negative literal in $(\neg L \vee R)\theta$ (this preserves refutational completeness). For the sake of clarity, and to increase the generality of the results, we prefer to state the weakest possible conditions insuring termination.

Factorization

$$\frac{L \vee L' \vee R}{(L \vee R)\theta}$$

If: $L \vee L' \vee R$ is positive, θ is the m.g.u. of L and L'.

[1] It is well known that m.g.u.'s are unique up to a renaming.

We denote by $\Pi_<$ the calculus defined by these 4 rules: Hyperresolution, Paramodulation (1 and 2), Factorization. If S is a set of clauses, we denote by $\Pi_<(S)$ the set of clauses that can be deduced from $S \cup \{x \approx x\}$ by applying one of the rules in $\Pi_<$.

We define the following resolution operators (see [20] for details on this technique). $\Pi^0_<(S) \stackrel{def}{=} S$, $\Pi^{i+1}_<(S) \stackrel{def}{=} \Pi_<(\Pi^i_<(S)) \cup \Pi^i_<(S)$. $\Pi^\infty_<(S)$ denotes the limit of this sequence, i.e. the set: $\Pi^\infty_<(S) \stackrel{def}{=} \bigcup_{i=0}^\infty \Pi^i_<(S)$.

$\Pi_<$ is sound and refutationally complete (this follows from the results in [3], see also [4]) hence $\Pi^\infty_<(S)$ contains $\Box$ iff S is unsatisfiable. Unfortunately, $\Pi_<$ does not necessarily terminate on equational sets of clauses in $\mathcal{OCC}1\mathcal{N}^*_=$, even if usual simplification rules such as subsumption are used to prune the search space. The following example will suffice to convince the reader:

Example 4.2 We consider the following set of clauses:

$$1 \quad f(x) \approx h(y)$$
$$2 \quad h(x) \approx g(f(y))$$

The reader can easily check that S belongs to $\mathcal{OCC}1\mathcal{N}^*_=$. Note that whatever the ordering $<$ may be, we have $f(x) \not< h(y)$ and $h(x) \not< g(f(y))$ (since $<$ is a reduction ordering and x may be replaced by $h(y)$ and $g(f(y))$ respectively). We deduce:

$$3 \quad f(x) \approx g(f(y)) \quad \text{(paramodulation 1, clause 2 into 1)}$$
$$4 \quad f(x) \approx g(h(y)) \quad \text{(paramodulation 1, clause 1 into 3)}$$
$$5 \quad f(x) \approx g(g(f(y))) \quad \text{(paramodulation 1, clause 2 into 4)}$$
$$6 \quad \ldots$$

It is obvious that an infinite number of distinct clauses, of the form $f(x) \approx g^n(f(y))$, can be deduced.

Therefore, a more sophisticated calculus is mandatory. In the next section, we introduce a new simplification rule that is sufficient to ensure that $\Pi_<$ terminates on any clause set in $\mathcal{OCC}1\mathcal{N}^*_=$.

5 The renaming rule

The basic principle of the simplification rule is to dynamically introduce new function symbols to "rename" some of the functional terms occurring in the clause set. The goal is to eliminate "irrelevant" parameters from the equations. For example, assume that an equation $f(x, y) \approx g(x, z)$ is generated. Then it is clear that the values of $f(x, y)$ and $g(x, z)$ do not depend on y, z but *only* on x.

Therefore, a new unary function h may be introduced, mapping each term x to $f(x,y)$ and $g(x,z)$ (since $f(x,y) \approx g(x,z)$ holds for all y,z the values of y,z are irrelevant). $f(x,y) \approx g(x,z)$ may be deleted, and replaced by the conjunction: $f(x,y) \approx h(x) \wedge g(x,z) \approx h(x)$. In order to ensure that the equation $f(x,y) \approx g(x,z)$ will not be generated again by paramodulation, we assume that $h(x)$ is strictly lower than $f(x,y)$ and $g(x,z)$. In our case, all the equations that we consider are linear. Thus any equation $t \approx s$ may be replaced by two equations $t \approx a$ and $s \approx a$, where a is a new constant symbol strictly lower than t and s.

From now, we assume that Σ contains an infinite set of constant symbols $\mathcal{C}$ not occurring in the initial set of clauses S. We also assume that each constant symbol a occurring in $\mathcal{C}$ is strictly smaller than any ground term whose head symbol is not in $\mathcal{C}$, i.e. for all $a \in \mathcal{C}$, for all n-ary function symbols $f \in \Sigma \setminus \mathcal{C}$ and for all terms $(t_1, \ldots, t_n)$, we have $f(t_1, \ldots, t_n) > a$.

An equation $t \approx s$ is said to be *elementary* iff $s \in \mathcal{C}$, t is linear and $t > s$.

The renaming rule is formally defined as follows:

$$\frac{S \cup \{(t \approx s) \vee R\}}{S \cup \{(t \approx c) \vee R, (s \approx c) \vee R\}}$$

If c is a new constant symbol in $\mathcal{C}$, not occurring in $S \cup \{(t \approx s) \vee R\}$, and neither t nor s occur in $\mathcal{C}$.

Note that the renaming rule does not merely deduce new clauses, but actually *deletes* existing clauses and *replaces* them by new clauses.

Lemma 5.1 *Let S be a set of clauses in $\mathcal{OCC}1\mathcal{N}^*_=$. Let S' be a set of clauses obtained by applying the Renaming rule on S.*

(i) *S' is in $\mathcal{OCC}1\mathcal{N}^*_=$.*

(ii) *If S is satisfiable then S' is satisfiable.*

(iii) *If $\mathcal{M}$ is a model of S', then $\mathcal{M} \models S$.*

Lemma 5.2 *Let S be a set of clauses in $\mathcal{OCC}1\mathcal{N}^*_=$. Indeterministic application of the Renaming rule terminates on S.*

If S is a set of clauses in $\mathcal{OCC}1\mathcal{N}^*_=$, we denote by $R^*(S)$ an (arbitrarily chosen) normal form of S w.r.t. the Renaming rule. By repeated applications of Lemma 5.1, we know that:

(i) $R^*(S) \in \mathcal{OCC}1\mathcal{N}^*_=$ and

(ii) S is satisfiable iff $R^*(S)$ is satisfiable, and

(iii) any model of $R^*(S)$ is a model of S.

6 Decidability proof

In this section, we prove that $\Pi_<$ terminates on $\mathcal{OCC}1\mathcal{N}^*_=$ provided that the Renaming rule is applied on the clause set at hand, thus showing that the satisfiability

problem is decidable for $\mathcal{OCC}1\mathcal{N}^*_=$. We need to introduce some further definitions.

Let S_1, S_2 be two sets of clauses. We denote by $S_1 \times S_2$ the set of clauses of the form: $C_1 \vee C_2$, where C_1 is a renaming of a clause in S_1, C_2 is a renaming of a clause in S_2 and $\mathit{Var}(C_1) \cap \mathit{Var}(C_2) = \emptyset$.

Similarly, if S is a set of clauses, we denote by S^k the set $\underbrace{S \times \ldots \times S}_{n \text{ times}}$ and by S^* the set $\bigcup_{i=0}^{\infty} S^i$.

Lemma 6.1 *Let S be a finite set of clauses. S^* is finite (up to condensing).*

For any set of clauses S, we denote by $\mathit{eq}(S)$ the set of equations occurring in a clause in S, i.e. $\mathit{eq}(S) \stackrel{def}{=} \{(t \approx s) \mid C \in S, (t \approx s) \in C\}$.

The following lemma states some properties of the equations that may be generated from the set of equations occurring in $R^*(S)$ using the inference rules in $\Pi_<$.

Lemma 6.2 *Let $S \in \mathcal{OCC}1\mathcal{N}^*_=$. For any $k \in \mathbb{N}$, and for any equation $E \in \Pi^k_<(\mathit{eq}(R^*(S)))$, E is of the form $t \approx c$, where $t > c$, t is linear, $\tau(t) \leq \tau(\mathit{eq}(R^*(S)))$ and $c \in \mathcal{C}$.*

We immediately deduce the following:

Corollary 6.3 *Let $S \in \mathcal{OCC}1\mathcal{N}^*_=$. $\Pi^\infty_<(\mathit{eq}(R^*(S)))$ is finite, up to a renaming of variables.*

Now, the following lemma states some useful properties of the clauses generated during the proof process. In particular, it shows that applying the rules in $\Pi_<$ to clauses belonging to $\mathcal{OCC}1\mathcal{N}^*_=$ only produces clauses that are still in $\mathcal{OCC}1\mathcal{N}^*_=$.

Lemma 6.4 *Let $S \in \mathcal{OCC}1\mathcal{N}^*_=$. For any $C \in \Pi^\infty_<(R^*(S))$, we have:*

(i) $|\mathit{Occ}(x, C^+)| \leq 1$ *for all* $x \in \mathcal{X}$ *and*

(ii) $\tau_{max}(x, C^+) \leq \tau_{min}(x, C^-)$ *for all* $x \in \mathit{Var}(C^+) \cap \mathit{Var}(C^-)$ *and.*

(iii) $\mathit{Var}(C^+) \cap \mathit{Var}(C^u) = \emptyset$.

(iv) *If* $C \equiv (t \approx s) \vee C'$ *then* $t \approx s \in \Pi^\infty_<(\mathit{eq}(R^*(S)))$ *and* $\mathit{Var}(t \approx s) \cap \mathit{Var}(C') = \emptyset$.

Let S be a set of clauses and let E be a set of equations. We denote by $\Pi_<(S, E)$ the set of clauses deduced by applying the paramodulation rule from clauses in E into clauses in S (clauses in S are not used for paramodulation). The set $\Pi^k_<(S, E)$ is inductively defined as follows.

- $\Pi^0_<(S, E) \stackrel{def}{=} S$.
- $\Pi^{k+1}_<(S, E) = \Pi^k_<(S, E) \cup \Pi_<(\Pi^k_<(S, E), E)$.
- $\Pi^\infty_<(S, E) \stackrel{def}{=} \bigcup_{k=0}^{\infty} \Pi^k_<(S, E)$.

The following lemma shows that $\Pi^\infty_<(S, E)$ is finite if E is finite and only contains elementary equations.

Lemma 6.5 *Let S be a finite set of clauses in $\mathcal{OCC}1\mathcal{N}^*_=$ and let E be a finite set of elementary equations. $\Pi^\infty_<(S, E)$ is finite.*

We need to introduce a new notation. If d is an integer, we denote by $\mathrm{pc}(d)$ the set of clauses C such that C is positive, linear and $\tau(C) \leq d$.

The following lemma gives the general form of the clauses generated from S by applying the rules in $\Pi_<$.

Lemma 6.6 *Let S be a clause set in $\mathcal{OCC}1\mathcal{N}^*_=$ and let*

$$S' = \Pi^\infty_<(S, \Pi^\infty_<(\mathit{eq}(S))).$$

$$\Pi^\infty_<(S) \subseteq S'^* \times \mathit{pc}(\tau(S'))$$

This entails the following:

Lemma 6.7 *Let S be a set of equational clauses in $\mathcal{OCC}1\mathcal{N}^*_=$. $\Pi^\infty_<(R^*(S))$ is finite (up to condensing).*

Corollary 6.8 *$\mathcal{OCC}1\mathcal{N}^*_=$ is decidable.*

7 Model Building

For many applications, detecting satisfiability is not sufficient and it is also important to be able to construct explicitly a model of the formula, in case it is satisfiable [3]. In the non equational case, $\mathcal{OCC}1\mathcal{N}$ is known to be finitely controllable (i.e. any satisfiable clause set in $\mathcal{OCC}1\mathcal{N}$ has a finite model). Actually [8] presents a procedure for extracting automatically a finite model of certain clause sets S in case hyperresolution terminates on S without detecting a contradiction. This is done by first constructing a Herbrand model represented by a finite set of linear non equational atoms (using a kind of "splitting" of clauses, but without backtracking, in contrast to SATCHMO-like algorithms [22]) and then by "projecting" the model on a finite domain. Unfortunately, this algorithm does not work for the equational (and non ground) case, because the reflexivity axiom $x \approx x$ is not compatible with the projection. In this section, we provide an algorithm for building Herbrand models of satisfiable sets of clauses in $\mathcal{OCC}1\mathcal{N}^*_=$.

Since the interpretations we build are infinite, they cannot be represented as usual by truth tables hence a suitable representation mechanism has to be provided. In [8], Herbrand interpretations are represented by finite sets of non equational atoms (ARM). Such a representation is suitable for applications because the evaluation problem (i.e. the problem of finding the truth value of a given formula in the represented interpretation) is decidable [15]. However the evaluation problem is known to be undecidable for equational ARM, thus further restrictions on atomic representations are mandatory in our case. In [9], models are represented by sets of *ground* equational atoms. In this paper, we extend this technique by considering sets of elementary equational atoms.

Definition 7.1 A *Elementary Equational Atomic Representation (EEAR)* is a set containing only non equational linear atoms or elementary equations. We say that a EEAR E *represents* an interpretation $\mathcal{I}$ iff for all ground atoms A $\mathcal{I} \models A$ iff $E \models A$.

If E is a EEAR, we denote by $\mathcal{M}_E$ the interpretation represented by E ($\mathcal{M}_E$ is obviously unique).

The following key theorem shows that the evaluation problem is decidable for the interpretations specified by EEARs.

Theorem 7.2 *Let E be an EEAR. The problem of finding the truth value of a first-order formula ϕ without equality in the interpretation $\mathcal{M}_E$ is decidable.*

Now, it remains to show how to construct a EEAR from a saturated set of clauses in $\mathcal{OCC}1\mathcal{N}^*_=$. We denote by $\Pi^{mb}_<$ the calculus $\Pi_<$ enriched by the following splitting rule.

$$\frac{S \cup \{C \vee D\}}{S \cup \{C\} \qquad S \cup \{D\}}$$

If $C \vee D$ is positive.

The splitting rule transforms a set of clauses into a disjunction of clause sets. It is correct, since any positive clause is linear: we have $\mathit{Var}(C) \cap \mathit{Var}(D) = \emptyset$ and $S \cup \{C \vee D\}$ is satisfiable iff one of the set $S \cup \{C\}$ or $S \cup \{D\}$ is satisfiable. Moreover, any model of $S \cup \{C\}$ (resp. $S \cup \{D\}$) is a model of $S \cup \{C \vee D\}$.

Obviously the splitting rule does not affect the termination behavior of the calculus. Thus, if S is a satisfiable set of clauses in $\mathcal{OCC}1\mathcal{N}^*_=$, then there exists at least one set of clauses $S' = \Pi^{mb*}_<(S)$ such that S does not contain $\Box$ and $\Pi^{mb}_<(S') = S'$ (note that since splitting is a branching rule, there may exist several clause sets having this property).

We denote by $\mathit{Mod}(S)$ the set of positive unit clauses in $\Pi^{mb*}_<(S)$. Since any positive clause in $\Pi^{mb}_<(S)$ is linear and all equations are elementary, $\mathit{Mod}(S)$ must be a EEAR. The following lemma states the correctness of our construction.

Lemma 7.3 *Let S be a clause set in $\mathcal{OCC}1\mathcal{N}^*_=$. If S is satisfiable then $\mathcal{M}_{\mathit{Mod}(S)} \models S$.*

8 Conclusion

We defined a decidable extension of $\mathcal{OCC}1\mathcal{N}$ to clause sets possibly containing non ground equational literals. We provided a resolution-based decision procedure for $\mathcal{OCC}1\mathcal{N}$ and a model extraction algorithm for satisfiable clause sets. Beside containing equality, this class has the interesting feature that, in contrast to similar existing classes such as $\mathcal{PVD}$ or $\mathcal{BU}$, it may contain non range-restricted clauses, thus non ground (possibly equational) clauses may be generated during the proof process.

The definition of $\mathcal{OCC}1\mathcal{N}^*_=$ is mainly based on the depth of the occurrence of the variables. In [19] more general termination results are considered, using different kinds of complexity measures. However all the clause sets considered in [19] are range-restricted. An interesting possibility would be to extend the definition of $\mathcal{OCC}1\mathcal{N}$ in order to deal with other kinds of complexity measures **and** with non range-restricted clauses. This would result in more expressive (hopefully decidable) classes, mixing $\mathcal{OCC}1\mathcal{N}$ with the T-dominated classes in [19].

References

[1] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[2] L. Bachmair, H. Ganzinger, and U. Waldmann. Superposition with simplification as a decision procedure for the monadic class with equality. In *Computational Logic and Proof Theory, KGC 93*, pages 83–96. Springer, LNCS 713, 1993.

[3] C. Bourely, R. Caferra, and N. Peltier. A Method for Building Models Automatically. Experiments with an extension of OTTER. In *Proceedings of CADE-12*, pages 72–86. Springer LNAI 814, 1994.

[4] R. Caferra, A. Leitsch, and N. Peltier. Automated model building. Submitted, 2002.

[5] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997.

[6] B. Dreben and W. D. Goldfarb. *The Decision Problem, Solvable Classes of Quantificational Formulas*. Addison-Wesley, 1979.

[7] C. Fermueller and G. Moser. Have spass with occ1n$^{=}_{g}$. In *LPAR'2000*, pages 114–130. Springer, 2000. Reunion, 2000.

[8] C. Fermüller and A. Leitsch. Hyperresolution and automated model building. *Journal of Logic and Computation*, 6(2):173–203, 1996.

[9] C. Fermüller and A. Leitsch. Decision procedures and model building in equational clause logic. *Journal of the IGPL*, 6(1):17–41, 1998.

[10] C. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. Resolution decision procedures. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 25, pages 1791–1850. Elsevier, 2001.

[11] C. Fermüller, A. Leitsch, T. Tammet, and N. Zamov. *Resolution Methods for the Decision Problem*. LNAI 679. Springer, 1993.

[12] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[13] L. Georgieva, U. Hustadt, and R. Schmidt. A new clausal class decidable by hyperresolution. In A. Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 258–272. Springer-Verlag, July 27-30 2002.

[14] L. Georgieva, U. Hustadt, and R. A. Schmidt. Hyperresolution for guarded formulae. In P. Baumgartner and H. Zhang, editors, *Proceedings of the Third International Workshop on First-Order Theorem Proving (FTP 2000)*, volume 5/2000 of *Fachberichte Informatik*, pages 101–112, Koblenz, Germany, 2000. Institut für Informatik, Universität Koblenz-Landau.

[15] G. Gottlob and R. Pichler. Working with ARMs: Complexity results on atomic representations of Herbrand models. *Information and Computation*, 2001. 25 pages, to appear.

[16] U. Hustadt and R. A. Schmidt. Maslov's class K revisited. In H. Ganzinger, editor, *Automated Deduction—CADE-16*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 172–186. Springer, 1999.

[17] U. Hustadt and R. A. Schmidt. Using resolution for testing modal satisfiability and building models. *Journal of Automated Reasoning*, 28(2):205–232, Feb. 2002.

[18] W. Joyner. Resolution strategies as decision procedures. *Journal of the ACM*, 23:398–417, 1976.

[19] A. Leitsch. Deciding clause classes by semantic clash resolution. *Fundamenta Informaticae*, 18:163–182, 1993.

[20] A. Leitsch. *The resolution calculus*. Springer. Texts in Theoretical Computer Science, 1997.

[21] D. W. Loveland. *Automated Theorem Proving: A Logical Basis*, volume 6 of *Fundamental Studies in Computer Science*. North Holland, 1978.

[22] R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In *Proc. of CADE-9*, pages 415–434. Springer, LNCS 310, 1988.

[23] N. Peltier. On the decidability of the PVD class with equality. *Logic Journal of the IGPL*, 9(4):601–624, 2001.

[24] J. Robinson. Automatic deduction with hyperresolution. *Intern. Journal of Computer Math.*, 1:227–234, 1965.

[25] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 12:23–41, 1965.

[26] T. Rudlof. SHR tableaux - A Framework for Automated Model Generation. *Journal of Logic and Computation*, 10(6):107–155, 2000.

Transforming equality logic to propositional logic

Hans Zantema [1] and Jan Friso Groote [2]

Department of Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract

We investigate and compare various ways of transforming equality formulas to propositional formulas, in order to be able to solve satisfiability in equality logic by means of satisfiability in propositional logic. We propose *equality substitution* as a new approach combining desirable properties of earlier methods, we prove its correctness and show its applicability by experiments.

1 Introduction

We consider *equality formulas* being propositional formulas in which the atoms are equalities between variables. Two such formulas are called *equality equivalent*, denoted by $\simeq_E$, if for any interpretation of the variables in any domain they yield the same result. For instance, we have

$$x = y \wedge x = z \quad \simeq_E \quad x = y \wedge y = z$$

since for both formulas the result is true if and only if the variables x, y, z all three have the same interpretation. On the other hand, in propositional logic they are not equivalent: writing p, q, r for $x = y$, $x = z$, $y = z$, respectively, we do not have $p \wedge q \equiv p \wedge r$.

The main question we address is the question of how to check whether two (big) equality formulas are equality equivalent automatically. A direct observation shows that

$$\phi \simeq_E \psi \iff \neg(\phi \leftrightarrow \psi) \simeq_E \mathsf{false},$$

[1] Email: h.zantema@tue.nl
[2] Email: jfg@win.tue.nl

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

hence checking equivalence of two formulas can be done by checking whether a formula is equivalent to false. The latter is called *satisfiability*, hence we are interested in satisfiability of equality formulas.

This problem plays an important role in hardware verification. In fact there one is interested in a slightly more extensive logic: the logic of equality with uninterpreted functions (UIF, [5]). However, by Ackermann's transformation ([1]) the problem of deciding validity of a formula in UIF is reduced to satisfiability of equality formulas. More recently, an improved transformation serving the same goal was proposed in [2].

One approach was presented in [7], where a variant of BDD-technology (EQ-BDDs) was developed for satisfiability of equality formulas. The given method is complete in the sense that their algorithm always terminates, and decides whether the given formula is satisfiable. Unfortunately, in EQ-BDDs there is no unique representation as is the case in ordinary BDDs for propositional formulas. Another method is proposed in [11]. There a resolution-like method was developed for checking satisfiability formulas in CNF.

A different approach is first transform the equality formula to a propositional formula and then analyze this propositional formula. For propositional formulas a lot of work has been done for efficient satisfiability checking, yielding a variety of efficient and usable implementations. In this paper we concentrate on transformations Ψ from equality formulas to propositional formulas by which satisfiability of equality formulas is transformed to satisfiability of propositional formulas, i.e.,

$$\phi \simeq_E \mathsf{false} \iff \Psi(\phi) \equiv \mathsf{false}.$$

Having such a transformation Ψ then checking satisfiability of an equality formula ϕ proceeds as follows: compute $\Psi(\phi)$ and decide whether $\Psi(\phi) \equiv \mathsf{false}$ by a standard satisfiability checker for propositional formulas. For such a transformation Ψ a number of properties is desirable:

- the size of $\Psi(\phi)$ is not too big;
- the structure of $\Psi(\phi)$ reflects the structure of ϕ;
- the variables of $\Psi(\phi)$ represent equalities in ϕ.

The main goal of these properties is that checking (propositional) satisfiability of $\Psi(\phi)$ by standard techniques is feasible for a reasonable class of formulas ϕ. Roughly speaking two main approaches can be distinguished:

(i) Addition of transitivity. In this approach it is analyzed which transitivity properties may be relevant for ϕ, and Ψ is defined by

$$\Psi(\phi) = \phi \wedge T,$$

where T is the conjunction of the relevant transitivity properties. This approach is followed in [6,3,4].

(ii) Bit vector encoding. In this approach $\lceil \log(\#A) \rceil$ boolean variables x_i are

introduced for every variable x, where $\#A$ is the size of the set A of variables, and $\Psi(\phi)$ is obtained from ϕ by replacing every $x = y$ by

$$\bigwedge_i (x_i \leftrightarrow y_i).$$

In [6] this is already mentioned as a folklore method. Closely related is range allocation [9,10]. In this approach a formula structure is analyzed to define a small domain for each variable, preferably smaller than $\#A$. Then a standard BDD based tool is used to check satisfiability of the formula under the domain.

By addition of transitivity the variables of $\Psi(\phi)$ represent equalities in ϕ, but the structure of $\Psi(\phi)$ does not reflect the structure of ϕ. For instance, if ϕ is a formula over n variables then the size of T is $\Theta(n^3)$ which can be much bigger than the size of ϕ itself. On the other hand by bit vector encoding the structure of $\Psi(\phi)$ reflects the structure of ϕ, but the variables of $\Psi(\phi)$ do not represent equalities in ϕ. Moreover, although the size of the transformed formula is small, it often turns out that the efficiency of proving unsatisfiability of this formula by standard approaches is very bad.

In this paper we define *equality substitution* eqs as an alternative transformation that combines both desired properties. The emphasis is on proving correctness: both for the earlier approaches and equality substitution we prove the basic correctness property $\phi \simeq_E \mathsf{false} \iff \Psi(\phi) \equiv \mathsf{false}$. We are not aware of earlier full proofs for the earlier approaches. In the last section we report some experiments showing that equality substitution outperforms the bit vector encoding for a class of formulas similar to the pigeon hole formulas. Comparison of equality substitution to addition of transitivity shows a similar performance, but equality substitution yields much smaller formulas.

2 Basic definitions and properties

Let A be a finite set of variable symbols. We define an *equality formula* by the syntax

$$V ::= x \mid y \mid z \mid \cdots \quad \text{where } A = \{x, y, z, \ldots\}$$
$$E ::= V = V \mid \mathsf{true} \mid \mathsf{false} \mid \neg E \mid (E \vee E) \mid (E \wedge E) \mid (E \rightarrow E) \mid (E \leftrightarrow E)$$

Hence an equality formula consists of *equations* $x = y$ for $x, y \in A$ and usual boolean connectives. As usual redundant parentheses will be omitted. For instance, if $x, y, z \in A$ then $(x = y \wedge y = z) \rightarrow x = z$ is an equality formula.

A *domain* D is defined to be a non-empty set. For any domain D we call a function $\epsilon : A \rightarrow D$ an *assignment* to D. For any assignment ϵ we define its interpretation $\overline{\epsilon}$ on equality formulas inductively as follows:

$$\overline{\epsilon}(x=y) = \begin{cases} \mathsf{true} & \text{if } \epsilon(x) = \epsilon(y) \\ \mathsf{false} & \text{if } \epsilon(x) \neq \epsilon(y) \end{cases}$$

$$\overline{\epsilon}(\mathsf{true}) = \mathsf{true}$$
$$\overline{\epsilon}(\mathsf{false}) = \mathsf{false}$$

$$\begin{aligned} \overline{\epsilon}(\neg\phi) &= \neg\overline{\epsilon}(\phi) \\ \overline{\epsilon}(\phi \vee \psi) &= \overline{\epsilon}(\phi) \vee \overline{\epsilon}(\psi) \\ \overline{\epsilon}(\phi \wedge \psi) &= \overline{\epsilon}(\phi) \wedge \overline{\epsilon}(\psi) \\ \overline{\epsilon}(\phi \to \psi) &= \overline{\epsilon}(\phi) \to \overline{\epsilon}(\psi) \\ \overline{\epsilon}(\phi \leftrightarrow \psi) &= \overline{\epsilon}(\phi) \leftrightarrow \overline{\epsilon}(\psi) \end{aligned}$$

Two equality formulas ϕ, ψ are called *equality equivalent*, denoted as $\phi \simeq_E \psi$, if $\overline{\epsilon}(\phi) = \overline{\epsilon}(\psi)$ for every domain D and every assignment ϵ to D. For instance, one can check that

$$(x = y \wedge y = z) \to x = z \ \simeq_E \ \mathsf{true}.$$

We will concentrate on the question how to decide whether $\phi \simeq_E \psi$ for arbitrary equality formulas ϕ, ψ. It is easily checked that

$$\phi \simeq_E \psi \iff \neg(\phi \leftrightarrow \psi) \simeq_E \mathsf{false}$$

hence we may and shall concentrate on the question whether $\phi \simeq_E \mathsf{false}$ for a given equality formula ϕ.

Fix a total order $<$ on A. For an equality formula ϕ write $R(\phi)$ for the equality formula obtained from ϕ by replacing every $x = x$ by true and replacing $x = y$ by $y = x$ if $y < x$, for all $x, y \in A$. Clearly $R(\phi) \simeq_E \phi$ for every equality formula ϕ. An equality formula ϕ is called *reduced* if $\phi = R(\phi)$, i.e., it only contains equations $x = y$ satisfying $x < y$. By applying this reduction our question of deciding $\phi \simeq_E \mathsf{false}$ for arbitrary equality formulas reduces to the question of deciding $\phi \simeq_E \mathsf{false}$ for a reduced equality formula ϕ.

We write $\equiv$ for logical equivalence in the sense of propositional logic; if applied to equality formulas this means that an equation $x = y$ is considered as a propositional atom.

Write T for the conjunction of all formulas

$$\neg R(x = y) \vee \neg R(y = z) \vee R(x = z)$$

for which $x, y, z \in A$ are all three distinct.

Theorem 2.1 *Let ϕ be a reduced equality formula. Then $\phi \simeq_E \mathsf{false}$ if and only if $\phi \wedge T \equiv \mathsf{false}$.*

Proof. First assume that $\phi \wedge T \equiv \mathsf{false}$. Let $\epsilon : A \to D$ be arbitrary; we have to prove that $\overline{\epsilon}(\phi) = \mathsf{false}$. By transitivity of equality in D we obtain that

$$\overline{\epsilon}(\neg R(x = y) \vee \neg R(y = z) \vee R(x = z)) = \mathsf{true}.$$

As a consequence we obtain $\overline{\epsilon}(T) = \mathsf{true}$. Hence

$$\overline{\epsilon}(\phi) = \overline{\epsilon}(\phi) \wedge \overline{\epsilon}(T) = \overline{\epsilon}(\phi \wedge T) = \mathsf{false};$$

the last step follows from $\phi \wedge T \equiv \mathsf{false}$ and the definition of $\overline{\epsilon}$.

Conversely assume that $\phi \simeq_E \mathsf{false}$ holds and $\phi \wedge T \not\equiv \mathsf{false}$; we have to derive a contradiction. Since $\phi \wedge T$ is satisfiable there is an assignment δ on the atoms of the shape $x = y$ to the booleans such that $\overline{\delta}(\phi \wedge T) = \mathsf{true}$, where $\overline{\delta}$ is the interpretation corresponding to δ. Hence $\overline{\delta}(\phi) = \overline{\delta}(T) = \mathsf{true}$. Define the relation $\simeq$ in A as follows:

$$x \simeq y \iff \overline{\delta}(R(x = y)).$$

From the definition of R it follows that $\simeq$ is reflexive and symmetric; since $\overline{\delta}(T) = \mathsf{true}$ we conclude that $\simeq$ is transitive. Hence $\simeq$ is an equivalence relation. By injectively mapping the equivalence classes of $\simeq$ to some domain D we obtain an assignment $\epsilon : A \to D$ satisfying

$$x \simeq y \iff \epsilon(x) = \epsilon(y).$$

By construction we now have $\overline{\epsilon}(\phi) = \overline{\delta}(\phi) = \mathsf{true}$, contradicting the assumption $\phi \simeq_E \mathsf{false}$. □

Theorem 2.1 shows that addition of transitivity is a valid approach for transforming equality formulas to propositional formulas by which satisfiability of equality formulas is transformed to satisfiability of propositional formulas. The next theorem states validity of the bit vector encoding approach.

Fix N to be the smallest number satisfying $2^N \geq \#A$. For every $x \in A$ introduce N boolean variables $x_1, \ldots, x_N$. Write A_N for the set of all of these $N * \#A$ boolean variables. The *bit vector encoding* bve transforming equality formulas over A to propositional formulas over A_N is defined as follows:

$$\mathsf{bve}(x = y) = \bigwedge_{i=1}^{N} (x_i \leftrightarrow y_i),$$

$$\mathsf{bve}(\mathsf{true}) = \mathsf{true}, \quad \mathsf{bve}(\mathsf{false}) = \mathsf{false}, \quad \mathsf{bve}(\neg\phi) = \neg\mathsf{bve}(\phi),$$
$$\mathsf{bve}(\phi \diamond \psi) = \mathsf{bve}(\phi) \diamond \mathsf{bve}(\psi)$$

for $x, y \in A$, $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$.

Theorem 2.2 *Let ϕ be an equality formula over A. Then $\phi \simeq_E \mathsf{false}$ if and only if $\mathsf{bve}(\phi) \equiv \mathsf{false}$.*

Proof. For the 'if' part we take an arbitrary assignment $\epsilon : A \to D$ satisfying $\overline{\epsilon}(\phi) = \mathsf{true}$ and we prove that this gives rise to a satisfying assignment for $\mathsf{bve}(\phi)$. Since $\#\epsilon(A) \leq \#A \leq 2^N$ there exists an injective map $\alpha : \epsilon(A) \to \{\mathsf{false}, \mathsf{true}\}^N$. Define $\overline{\alpha} : A_N \to \{\mathsf{false}, \mathsf{true}\}$ by

$$\alpha(\epsilon(x)) = (\overline{\alpha}(x_1), \ldots, \overline{\alpha}(x_N))$$

for all $x \in A$. Extend $\overline{\alpha}$ to propositional formulas over A_N by defining

$$\overline{\alpha}(\mathsf{true}) = \mathsf{true}, \quad \overline{\alpha}(\mathsf{false}) = \mathsf{false}, \quad \overline{\alpha}(\neg\phi) = \neg\overline{\alpha}(\phi),$$

$$\overline{\alpha}(\phi \diamond \psi) = \overline{\alpha}(\phi) \diamond \overline{\alpha}(\psi)$$

for $x, y \in A$, $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$. For $x, y \in A$ we obtain

$$\begin{aligned}
\overline{\epsilon}(x = y) = \mathsf{true} &\iff \epsilon(x) = \epsilon(y) \\
&\iff \alpha(\epsilon(x)) = \alpha(\epsilon(y)) \quad \text{(since } \alpha \text{ is injective)} \\
&\iff (\overline{\alpha}(x_1), \ldots, \overline{\alpha}(x_N)) = (\overline{\alpha}(y_1), \ldots, \overline{\alpha}(y_N)) \\
&\iff \overline{\alpha}(x_1) = \overline{\alpha}(y_1) \wedge \cdots \wedge \overline{\alpha}(x_N)) = \overline{\alpha}(y_N) \\
&\iff \overline{\alpha}(\textstyle\bigwedge_{i=1}^{N}(x_i \leftrightarrow y_i)) = \mathsf{true} \\
&\iff \overline{\alpha}(\mathsf{bve}(x = y)) = \mathsf{true}.
\end{aligned}$$

This holds for every equality $x = y$. Hence,

$$\overline{\alpha}(\mathsf{bve}(\phi)) = \overline{\epsilon}(\phi) = \mathsf{true}.$$

So, we have a satisfying assignment $\overline{\alpha}$ for $\mathsf{bve}(\phi)$, which we had to prove.

For the converse assume $\overline{\alpha} : A_N \rightarrow \{\mathsf{false}, \mathsf{true}\}$ is a satisfying assignment for $\mathsf{bve}(\phi)$. Let $D = \{\mathsf{false}, \mathsf{true}\}^N$. Define $\epsilon : A \rightarrow D$ by $\epsilon(x) = (\overline{\alpha}(x_1), \ldots, \overline{\alpha}(x_N))$. Similarly as above we obtain $\overline{\alpha}(\mathsf{bve}(x = y)) = \mathsf{true} \iff \overline{\epsilon}(x = y) = \mathsf{true}$, hence from $\overline{\alpha}(\mathsf{bve}(\phi)) = \mathsf{true}$ we may conclude $\overline{\epsilon}(\phi) = \mathsf{true}$, contradicting the assumption $\phi \simeq_E \mathsf{false}$. □

The requirement $2^N \geq \#A$ is essential for the validity of Theorem 2.2 as is shown by the following example. Let $A = \{x_1, \ldots, x_n\}$ and $n > 2^N$. Then

$$\bigwedge_{1 \leq i < j \leq n} \neg(x_i = x_j) \quad \not\simeq_E \quad \mathsf{false},$$

while

$$\mathsf{bve}(\bigwedge_{1 \leq i < j \leq n} \neg(x_i = x_j)) \quad = \quad \bigwedge_{1 \leq i < j \leq n} \neg(\bigwedge_{k=1}^{N}(x_{ik} \leftrightarrow x_{jk})) \quad \equiv \quad \mathsf{false}.$$

3 Equality substitution

In this section equality substitution eqs is introduced for transforming equality formulas to propositional formulas, combining desired properties of the two transformations considered until now. Just like in bit vector encoding a substitution is applied on the equalities in the formula, and the rest of the formula remains unchanged. The main point is to define $\mathsf{eqs}(x = y)$ for variables x, y such that $\phi \simeq_E \psi \iff \mathsf{eqs}(\phi) \equiv \mathsf{eqs}(\psi)$.

Let $<$ on A be the order that we already fixed for defining R. It is convenient to number the elements of A with respect to this order, i.e., we assume

$A = \{x_1, x_2, \ldots, x_n\}$ for $n = \#A$, satisfying

$$x_i < x_j \iff i < j.$$

For every i, j satisfying $1 \leq i < j \leq n$ we introduce a fresh propositional variable p_{ij}; the set of all these $\frac{n(n-1)}{2}$ variables is denoted by P_A.

For $1 \leq k \leq i < j \leq n$ we define $P(k, i, j)$ inductively by

$$P(i, i, j) = p_{ij}$$

for all i, j satisfying $1 \leq i < j \leq n$, and

$$P(k, i, j) = (p_{ki} \wedge p_{kj}) \vee (\neg p_{ki} \wedge \neg p_{kj} \wedge P(k+1, i, j))$$

for all k, i, j satisfying $1 \leq k < i < j \leq n$. We will use these formulas only for $k = 1$; the formula $P(1, i, j)$ is a propositional formula over P_A of size $O(i)$. For instance, $P(1, 3, 5)$ is equal to

$$(p_{13} \wedge p_{15}) \vee (\neg p_{13} \wedge \neg p_{15} \wedge ((p_{23} \wedge p_{25}) \vee (\neg p_{23} \wedge \neg p_{25} \wedge p_{35}))).$$

We define the transformation eqs from equality formulas over A to propositional formulas over P_A as follows:

$$\mathsf{eqs}(x_i = x_j) = \begin{cases} \mathsf{true} & \text{if } i = j, \\ P(1, i, j) & \text{if } i < j, \\ P(1, j, i) & \text{if } j < i, \end{cases}$$

$$\mathsf{eqs}(\mathsf{true}) = \mathsf{true}, \quad \mathsf{eqs}(\mathsf{false}) = \mathsf{false}, \quad \mathsf{eqs}(\neg\phi) = \neg\mathsf{eqs}(\phi),$$

and

$$\mathsf{eqs}(\phi \diamond \psi) = \mathsf{eqs}(\phi) \diamond \mathsf{eqs}(\psi)$$

for $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$.

It is hard to give an intuition for eqs other than what follows directly from its definition; surprisingly the original intuition we had for eqs turned out to be wrong. Many modifications of eqs turned out to violate the essential property below.

Theorem 3.1 *Let ϕ, ψ be arbitrary equality formulas over A. Then*

$$\phi \simeq_E \psi \iff \mathsf{eqs}(\phi) \equiv \mathsf{eqs}(\psi).$$

Indeed, $\mathsf{eqs}((x_1 = x_2 \wedge x_2 = x_3) \rightarrow x_1 = x_3)$ is equal to

$$(p_{12} \wedge ((p_{12} \wedge p_{13}) \vee (\neg p_{12} \wedge \neg p_{13} \wedge p_{23}))) \rightarrow p_{13}$$

which is logically equivalent to $\mathsf{eqs}(\mathsf{true}) = \mathsf{true}$.

In the remainder of this section we prove Theorem 3.1. We start by proving

$$\phi \simeq_E \psi \iff \mathsf{eqs}(\phi) \equiv \mathsf{eqs}(\psi).$$

We assume that

$$\overline{\delta}(\mathsf{eqs}(\phi)) = \overline{\delta}(\mathsf{eqs}(\psi))$$

for all $\delta : P_A \to \mathsf{Bool}$, and we have to prove that $\overline{\epsilon}(\phi) = \overline{\epsilon}(\psi)$ for every domain D and every assignment $\epsilon : A \to D$. This follows from the following lemma, proving the $\Leftarrow$-part of Theorem 3.1.

For an assignment $\epsilon : A \to D$ we define $\delta_\epsilon : P_A \to \mathsf{Bool}$ by

$$\delta_\epsilon(p_{ij}) \iff \epsilon(x_i) = \epsilon(x_j).$$

Lemma 3.2 *Let ϕ be an equality formula and let $\epsilon : A \to D$ be any assignment. Then*

$$\overline{\epsilon}(\phi) = \overline{\delta}_\epsilon(\mathsf{eqs}(\phi)).$$

Proof. Due to the compositional definition of eqs it suffices to prove this for ϕ being of the shape $x_i = x_j$. In case of $i = j$ this holds since $\overline{\epsilon}(x_i = x_i) = \mathsf{true} = \overline{\delta}_\epsilon(\mathsf{true}) = \overline{\delta}_\epsilon(\mathsf{eqs}(x_i = x_i))$. In the remaining case $i \neq j$ we may assume $i < j$ by

$$\overline{\epsilon}(x_i = x_j) = \overline{\epsilon}(x_j = x_i)$$

and symmetry in the definition of eqs. Since $\mathsf{eqs}(x_i = x_j)$ is equal to $P(1, i, j)$ it remains to prove

$$\overline{\epsilon}(x_i = x_j) \iff \overline{\delta}_\epsilon(P(1, i, j)).$$

We prove this by proving the stronger claim

$$\overline{\epsilon}(x_i = x_j) \iff \overline{\delta}_\epsilon(P(k, i, j)).$$

for all $k = 1, 2, \ldots, i$ by reverse induction on k. For $k = i$ this holds by definition. As the induction hypothesis we now assume

$$\epsilon(x_i) = \epsilon(x_j) \iff \overline{\epsilon}(x_i = x_j) \iff \overline{\delta}_\epsilon(P(k+1, i, j)).$$

Now we have

$\overline{\delta}_\epsilon(P(k, i, j)) \quad \iff \quad$ (by definition)

$\overline{\delta}_\epsilon((p_{ki} \wedge p_{kj}) \vee (\neg p_{ki} \wedge \neg p_{kj} \wedge P(k+1, i, j))) \quad \iff \quad$ (by definition)

$(\epsilon(x_k) = \epsilon(x_i) \wedge \epsilon(x_k) = \epsilon(x_j)) \vee (\epsilon(x_k) \neq \epsilon(x_i) \wedge \epsilon(x_k) \neq \epsilon(x_j) \wedge \overline{\delta}_\epsilon(P(k+1, i, j)))$

$\iff$ (by the induction hypothesis)

$(\epsilon(x_k) = \epsilon(x_i) \wedge \epsilon(x_k) = \epsilon(x_j)) \vee (\epsilon(x_k) \neq \epsilon(x_i) \wedge \epsilon(x_k) \neq \epsilon(x_j) \wedge \epsilon(x_i) = \epsilon(x_j))$

$\iff$ (by transitivity of $=$)

$(\epsilon(x_k) = \epsilon(x_i) \wedge \epsilon(x_i) = \epsilon(x_j)) \vee (\epsilon(x_k) \neq \epsilon(x_i) \wedge \epsilon(x_k) \neq \epsilon(x_j) \wedge \epsilon(x_i) = \epsilon(x_j))$

$\iff$ (proposition logic)

$(\epsilon(x_k) = \epsilon(x_i) \vee \epsilon(x_k) \neq \epsilon(x_j) \vee \epsilon(x_i) \neq \epsilon(x_j)) \wedge \epsilon(x_i) = \epsilon(x_j)$

$\iff$ (by transitivity of $=$)

$\epsilon(x_i) = \epsilon(x_j) \iff \overline{\epsilon}(x_i = x_j)$

which we had to prove. □

The hard part of Theorem 3.1 is the $\Rightarrow$-part. For that we need a lemma.

Lemma 3.3 *Let T be the conjunction of all formulas*

$$\neg R(x = y) \vee \neg R(y = z) \vee R(x = z)$$

for which $x, y, z \in A$ are all three distinct. Then $\mathsf{eqs}(T) \equiv \mathsf{true}$.

Proof. We have to prove that $\mathsf{eqs}(\neg R(x = y) \vee \neg R(y = z) \vee R(x = z)) \equiv \mathsf{true}$. Let j, k, m satisfying $1 \leq j < k < m \leq n$ be the numbers of the variables x, y, z in some order. Then the required property is one of the following three propositional equivalences:

$$\neg P(1, j, k) \vee \neg P(1, j, m) \vee P(1, k, m) \equiv \mathsf{true},$$

$$\neg P(1, j, k) \vee P(1, j, m) \vee \neg P(1, k, m) \equiv \mathsf{true},$$
$$P(1, j, k) \vee \neg P(1, j, m) \vee \neg P(1, k, m) \equiv \mathsf{true}.$$

We will prove the more general property that for every i satisfying $1 \leq i \leq j$ the following three propositional equivalences hold:

$$\neg P(i, j, k) \vee \neg P(i, j, m) \vee P(i, k, m) \equiv \mathsf{true},$$

$$\neg P(i, j, k) \vee P(i, j, m) \vee \neg P(i, k, m) \equiv \mathsf{true},$$
$$P(i, j, k) \vee \neg P(i, j, m) \vee \neg P(i, k, m) \equiv \mathsf{true}.$$

First assume that the first equivalence does not hold. Then there is an assignment such that the propositions

$$P(i, j, k) = (p_{ij} \wedge p_{ik}) \vee (\neg p_{ij} \wedge \neg p_{ik} \wedge P(i + 1, j, k)),$$

$$P(i, j, m) = (p_{ij} \wedge p_{im}) \vee (\neg p_{ij} \wedge \neg p_{im} \wedge P(i + 1, j, m)),$$
$$\neg P(i, k, m) = (\neg p_{ik} \vee \neg p_{im}) \wedge (p_{ik} \vee p_{im} \vee \neg P(i + 1, k, m))$$

all three hold. If p_{ij} holds then we conclude from the validity of the first two propositions that p_{ik} and p_{im} both hold too, contradicting the validity of the third proposition. Hence $\neg p_{ij}$ holds. Then by validity of all three propositions we conclude that $\neg p_{ik}$, $\neg p_{im}$, $P(i + 1, j, k)$, $P(i + 1, j, m)$ and $\neg P(i + 1, k, m)$ all hold. Repeating the same argument $j - i$ times yields that

$$P(j, j, k) = p_{jk}, \quad P(j, j, m) = p_{jm},$$

$$\neg P(j,k,m) = (\neg p_{jk} \vee \neg p_{jm}) \wedge (p_{jk} \vee p_{jm} \vee \neg P(j+1,k,m))$$

all three hold, contradiction. Hence the first equivalence to be proved holds.

Next assume that the second equivalence does not hold. Then in a similar way after $j - i$ steps we obtain that

$$P(j,j,k) = p_{jk}, \quad \neg P(j,j,m) = \neg p_{jm},$$

$$P(j,k,m) = (p_{jk} \wedge p_{jm}) \vee (\neg p_{jk} \wedge \neg p_{jm} \wedge P(j+1,k,m))$$

all three hold, contradiction.

Finally assuming that the third equivalence does not hold yields in a similar way that

$$\neg P(j,j,k) = \neg p_{jk}, \quad P(j,j,m) = p_{jm},$$
$$P(j,k,m) = (p_{jk} \wedge p_{jm}) \vee (\neg p_{jk} \wedge \neg p_{jm} \wedge P(j+1,k,m))$$

all three hold, contradiction. □

Now we prove the $\Rightarrow$-part of Theorem 3.1.

Assume $\phi \simeq_E \psi$. Then $\neg(\phi \leftrightarrow \psi) \simeq_E \mathsf{false}$. From Theorem 2.1 we conclude that $\neg(\phi \leftrightarrow \psi) \wedge T \equiv \mathsf{false}$. In this equivalence the equalities are considered as propositional variables. Since eqs has been defined as a substitution on these variables we conclude $\mathsf{eqs}(\neg(\phi \leftrightarrow \psi) \wedge T) \equiv \mathsf{false}$. We obtain

$$\begin{aligned}
\neg(\mathsf{eqs}(\phi) \leftrightarrow \mathsf{eqs}(\psi)) &\equiv \mathsf{eqs}(\neg(\phi \leftrightarrow \psi)) \\
&\equiv \mathsf{eqs}(\neg(\phi \leftrightarrow \psi)) \wedge \mathsf{eqs}(T) \text{ (by Lemma 3.3)} \\
&= \mathsf{eqs}(\neg(\phi \leftrightarrow \psi) \wedge T) \\
&\equiv \mathsf{false}
\end{aligned}$$

hence $\mathsf{eqs}(\phi) \equiv \mathsf{eqs}(\psi)$, which concludes the proof of Theorem 3.1.

4 Experimental results

In this section we report some experimental results comparing addition of transitivity, bit vector encoding and equality substitution, all three in combination with various propositional satisfiability provers.

We consider the formulas form_n from [11] that are related to the pigeon hole formulas in proposition calculus. Just like pigeon hole formulas these are parameterized by a number n, they are easily seen to be contradictory by a meta argument, and each of the formulas is the conjunction of two subformulas. The formulas are defined as follows.

$$\mathsf{form}_n \equiv \Big(\bigwedge_{1 \le i < j \le n} x_i \neq x_j\Big) \wedge \bigwedge_{j=1}^{n} \Big(\bigvee_{i \in \{1,\ldots,n\}, i \neq j} x_i = y\Big)$$

There are $n+1$ variables $x_1, \ldots, x_n, y$. The first subformula states that all values of $x_1, \ldots, x_n$ are different.

The second subformula states that the value of y occurs in every subset

of size $n-1$ of $\{x_1, \ldots, x_n\}$, hence it will occur at least twice in $\{x_1, \ldots, x_n\}$, contradicting the property of the first subformula. Hence the total formula is unsatisfiable. This is a non-trivial kind of unsatisfiability in the following sense: the whole formula is a conjunction of a great number of formulas, and for every of these conjuncts it holds that the formula is satisfiable after removing the conjunct. Moreover, for every pair of variables the equality between these variables occurs in the formula, either positively or negatively. Since pigeon hole like formulas are well-known to be notoriously hard in propositional logic, we consider this formula to be an interesting candidate for experiments for techniques for checking satisfiability of equality formulas. We did our experiments on the formula form_n for n having the values 10, 15, 20, 30, 40, 50, 60.

We used three different propositional satisfiability checkers. The first one consists of computing the BDD using the package CUDD, see `http://supportweb.cs.bham.ac.uk/documentation/cudd/`. In the table this checker is denoted by 'bdd'. The second one first transforms the formula to CNF using Tseitin's transformation and then applies zChaff, see `http://ee.princeton.edu/~chaff/zchaff.php`. In the table this checker is denoted by 'ch'. The last one is the checker HeerHugo ([8]), denoted by 'hh'. All experiments are carried out under Linux on a 1Ghz. pentium 4.

The following table reports the results. Times are in seconds; '-' means that more than 600 seconds were required. Size indicates the number of binary symbols in the propositional formula.

	add transitivity				bit vector encoding				equality substitution			
n	size	bdd	ch	hh	size	bdd	ch	hh	size	bdd	ch	hh
10	1619	1	0	0	1079	56	1	113	794	0	0	0
15	5354	-	0	0	2519	-	7	-	2554	1	0	1
20	12539	-	0	0	5699	-	91	-	5889	20	0	1
30	41759	-	0	1	13049	-	-	-	19284	-	0	4
40	98279	-	1	3	28079	-	-	-	44979	-	1	16
50	191099	-	2	6	44099	-	-	-	86974	-	2	49
60	329219	-	4	11	63719	-	-	-	149269	-	5	123

About the bdd experiments with addition of transitivity we note that the order in which the big conjunction is computed is of great influence on the result. In the table we first computed the bdds of form_n and T separately and then computed the conjunction, as is suggested by the the shape of the formula. Only computing the bdd of T is already very expensive: for 12 variables the resulting bdd has over one million nodes. However, by computing the bdd of form_n and then consecutively taking conjunction with each of the transitivity properties gives a much better result: then unsatisfiability of form_{60} is proved in 62 seconds.

As a conclusion from the table we may state that the best results are obtained by the two transformations addition of transitivity and equality substitution, both in combination with zChaff: then unsatisfiability of form_{60} is proved in only a few seconds. Among these two transformations equality substitution gives rise to the smallest formulas. Although bit vector encoding gives rise to much smaller formulas, it gives a very bad performance on proving unsatisfiability.

5 Concluding Remarks

We proposed equality substitution as a new transformation by which the satisfiability problem for equality logic is transformed to the satisfiability problem for propositional logic. Both for earlier approaches and for this new approach we gave proofs for correctness. We did some experiments on pigeon hole like formulas showing that equality substitution serves well for proving unsatisfiability of equality formulas in combination with the propositional prover zChaff. Although this involves only one particular class of formulas, it is an indication for practical applicability.

References

[1] Ackermann, W., "Solvable cases of the decision problem," Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam, 1954.

[2] Bryant, R., S. German, and M. Velev, *Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic*, ACM Transactions on Computational Logic **2** (2001), pp. 93–134.

[3] Bryant, R. and M. Velev, *Boolean satisfiability with transitivity constraints*, in: E. Emerson and A. Sistla, editors, *Computer-Aided Verification (CAV'00)*, LNCS **1855** (2000), pp. 85–98.

[4] Bryant, R. and M. Velev, *Boolean satisfiability with transitivity constraints*, ACM Transactions on Computational Logic **3** (2002), pp. 604–627.

[5] Burch, J. and D. Dill, *Automated verification of pipelined microprocesoor control*, in: D. Dill, editor, *Computer-Aided Verification (CAV'94)*, LNCS **818** (1994), pp. 68–80.

[6] Goel, A., K. Sajid, H. Zhou, A. Aziz and V. Singhal, *BDD based procedures for a theory of equality with uninterpreted functions*, in: *Proceedings of Conference on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science **1427** (1998), pp. 244–255.

[7] Groote, J. F. and J. C. van de Pol, *Equational binary decision diagrams*, in: M. Parigot and A. Voronkov, editors, *Logic for Programming and Reasoning (LPAR)*, Lecture Notes in Artificial Intelligence **1955** (2000), pp. 161–178.

[8] Groote, J. F. and J. P. Warners, *The propositional formula checker HeerHugo*, Journal of Automated Reasoning **24** (2000), pp. 101–125.

[9] Pnueli, A., Y. Rodeh, O. Shtrichman and M. Siegel, *Deciding equality formulas by small domains instantiations*, in: *Computer Aided Verification (CAV'99)*, LNCS **1633** (1999), pp. 455–469.

[10] Rodeh, Y. and O. Shtrichman, *Finite instantiations in equivalence logic with uninterpreted functions*, in: *Computer Aided Verification (CAV'01)*, LNCS **2102** (2001), pp. 144–154.

[11] Tveretina, O. and H. Zantema, *A proof system and a decision procedure for equality logic*, Technical Report CS-report 03-02, Eindhoven University of Technology (2003), available via `http://www.win.tue.nl/~hzantema/TZ.pdf`.

Light-Weight Theorem Proving for Debugging and Verifying Pointer Manipulating Programs

Silvio Ranise [1]

LORIA & INRIA-Lorraine, Nancy (France)

David Déharbe [2,3]

LORIA & INRIA-Lorraine, Nancy (France) and DIMAp/UFRN, Natal (Brasil)

Abstract

We describe a combination of BDDs and superposition theorem proving, called *light-weight theorem proving*, and its application to the flexible and efficient automation of the reasoning activity required to debug and verify pointer manipulating programs. This class of programs is notoriously challenging to reason about and it is also interesting from a programming point of view since pointers are an important source of bugs. The implementation of our technique (in a system called haRVey) scales up significantly better than state-of-the-art tools such as E (a superposition prover) and *Simplify* (a prover based on the Nelson and Oppen combination schema of decision procedures which is used in ESC/Java) on a set of proof obligations arising in debugging and verifying C functions manipulating pointers.

1 Introduction

We are interested in debugging and verifying imperative programs. It is well-known that properties of programs can be expressed by formulae in some logical formalism so that debugging amounts to finding counter-examples for the validity of such formulae and checking the correctness of programs to finding proofs of their validity. Our goal is to build reasoning tools which provide an adequate theorem proving support for both debugging and verification of programs. This task has proven to be quite difficult since both a *high degree of automation* and the *capability of handling rich theories* are mandatory to build useful tools. Automation allows non-experts

[1] Email: Silvio.Ranise@loria.fr

[2] Email: david@consiste.dimap.ufrn.br

[3] This work was realized while the second author was on a post-doctoral stay at LORIA, thanks in part to CAPES grant BEX0006/02-5.

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

in theorem proving to productively use such tools. Handling rich theories allows users to introduce definitions to structure their specifications. Existing state-of-the-art reasoning systems are not completely successful in fulfilling both requirements.

On the one hand, propositional satisfiability solvers are highly efficient but are severely limited in expressiveness. Reductions from various decidable first-order theories are known (see e.g. [6]) but it is not clear how to handle user-defined symbols with this technique. When more general reductions are designed (such as the translation of first-order relational logic to propositional logic for program debugging of [15]), it is not clear how to use them to check the correctness of programs since only finite instances of the problem can be handled this way. On the other hand, first-order (or higher-order) theorem provers mechanise rich logics. Unfortunately, first-order (and *a fortiori* higher-order) theorem proving is not a "push-button" technology: tuning either the strategies or interactively guiding the system towards a proof is necessary. This is perhaps one of the main reasons why theorem proving has not been widely adopted in industry. Between these two extremes, there are tools called *validity checkers* (such as CVC, [4] ICS, [5] and *Simplify* [6]) which provide a high degree of automation to check the entailment of quantifier-free formulae by some fixed (combination of) theories. Unfortunately, they are not flexible enough to support the extension of the background theories by user-defined symbols, which are thus treated as uninterpreted. A solution would be to put the (conjunction of the) symbol definitions in the antecedent of an implication with consequent the formula to be proved. This makes the formula to be checked no longer quantifier-free and systems such as CVC and ICS cannot handle these kinds of problems as they are; quantifiers must be preliminary eliminated for the tools to handle the problem. Notice that this pre-processing can be non-trivial in the presence of interpreted symbols (see e.g. [12]). Furthermore, quantifier-elimination can significantly increase the size of the formula, possibly making the problem out of the reach of the validity checker. *Simplify* provides support for handling quantifiers by means of a heuristic matching mechanism to find instantiations of variables (see [17] for details). Unfortunately, given its heuristic nature, the mechanism may produce false negatives also for small formulae (for examples, see Table 2).

The main contribution of this paper is a technique called *lightweight theorem proving*, which combines BDDs and superposition theorem proving for the flexible and efficient implementation of the reasoning activity required to debug and verify imperative programs which manipulate pointers. This class of programs is difficult to reason about (see [21] for a discussion on this issue) and it is also interesting from a programming point of view since pointers are notoriously a source of bugs. It is thus a challenging and interesting application domain to test the viability of our approach. Section 2 describes how proof obligations for debugging are extracted from an (annotated) program. Section 3 describes how to build satisfiability procedures based on superposition theorem proving (see [2]) and how BDDs are used to

[4] `http://verify.stanford.edu/CVC/`
[5] `http://ics.csl.sri.com/`
[6] `http://research.compaq.com/SRC/esc/Simplify.html`

case-split on the boolean structure of formulae. We also describe how to eliminate cases which are subsumed by others already considered: this is one of the keys of the effectiveness of our approach. Finally, Section 4 reports a comparison between an implementation of our technique, called haRVey, and the state-of-the-art validity checker *Simplify*. We consider proof obligations arising in both debugging and verifying programs. It turns out that haRVey performs better than *Simplify* both in terms of efficiency and the correctness of the results. For example, in the verification of the Union-Find program of [18] (cf. Table 2), haRVey returns no false negative against the 6 of *Simplify* (out of 12 proof obligations). Furthermore, haRVey outperforms superposition theorem provers (in automatic mode), thereby achieving a better trade-off between expressivity and level of automation.

2 Debugging and Verifying Software

Our goal is to build a tool which assists the development of C programs manipulating linked lists. In particular, we consider the usual constructs of C such as variable declarations, assignments, conditionals, while loops, and memory allocation operations (i.e. `malloc` and `free`). Furthermore, we assume the following declaration of a linked list data type:

`typedef struct list_struct {` τ `car; struct list *cdr; } list;`

where τ is any data type over which equality is defined. We extend programs with annotations of the form `require` ϕ, `ensure` ϕ, `invariant` ϕ which stands for pre-, post-conditions, and loop invariants, where ϕ is a formula of first-order logic with equality.

For the *verification* of an (annotated) program π, we follow the standard approach of generating formulae whose validity implies the correctness of ϕ (see e.g. [13]). For lack of space, we will not say more about this and we will concentrate on our methodology to (symbolically) *debug* programs (see, also [16]). Debugging is particularly important for mainly two reasons. First, when developing a program, the programmer is more interested in finding bugs in it; only after gaining confidence, is he interested in proving correctness, which is a difficult and time consuming activity. Second, for debugging, it is not necessary to annotate loops with invariants whose invention is known to be a daunting task.

To detect a bug we only need a particular execution of the program which exposes it: finitely many iterations of the loops are sufficient. So, one of the inputs to our debugging tool is the number of iterations a loop must be (symbolically) executed. (It has been observed that usually few iterations suffice to detect a bug [15].) Given the number of iterations n, we unroll n times the loop `while` B S to `if` B { S`; while` (B) S }, where B is a boolean expression and S a statement. Notice that the number of possible execution paths to be considered may increase exponentially with the number n of loop iterations. In order to cope with this, we extend first-order logic with the ternary operator ite which can build either formulae or terms. The meaning of ite is as follows:

$$\mathsf{ite}(A, B, C) \text{ rewrites to } (A \Rightarrow B) \wedge (\neg A \Rightarrow C) \tag{9}$$

$$B[\mathsf{ite}(A, t_1, t_2)] \text{ rewrites to } \mathsf{ite}(A, B[t_1], B[t_2]), \tag{10}$$

where A, B, C are formulae and t_1, t_2 are terms. Using the ite construct is quite common for reasoning about program semantics (see e.g. [9]).

Since we want to reason about programs manipulating pointers, we represent the memory as a first-order concept so that we can build predicates which express both local and global properties of the storage. The idea is to represent the memory as a symbolic mapping from addresses to values by exploiting the theory of arrays (see e.g. [21]), denoted below with $\mathcal{A}$. The binary function symbol rd and the ternary wr are in the signature of $\mathcal{A}$. The axioms $Ax(\mathcal{A})$ of $\mathcal{A}$ are $\mathsf{rd}(\mathsf{wr}(A, I, E), I) = E$ and $I \neq J \Rightarrow \mathsf{rd}(A, J) = \mathsf{rd}(\mathsf{wr}(A, I, E), J)$, where A, I, J, E are implicitly universally quantified variables. In this way, program variables are first-order constants representing addresses in the memory and pointer variables are constants representing addresses in the memory whose content is an address in memory. Since each variable in a program must be stored at a distinct address, we assume the following set $\Delta := \{v_i \neq v_j \mid 1 \leq i \neq j \leq n\}$ of axioms, where $v_1, ..., v_n$ are the variables declared in the program. Furthermore, to indicate that a variable v is declared but not yet assigned, we use a distinguished symbol $?$ which is assumed to be the value stored in v, i.e. we assume the following singleton set $\Upsilon := \{\mathsf{rd}(m_0, V) = ?\}$ of axioms, where m_0 is the state of the memory before the execution of the first command and V is an implicitly universally quantified variable.

C assignments are easily encoded as nested applications of wr's and rd's. For example, consider the assignment `c=2` and let m be the state of the memory before its execution; then the state of the memory after its execution is $m' = \mathsf{wr}(m, \mathtt{c}, 2)$. Notice that reasoning about side-effects, which can be quite subtle in presence of pointers, can easily be done in the theory $\mathcal{A}$. To illustrate, consider the following program fragment: ℓ_1 `*i=3;` ℓ_2 `*j=2;` ℓ_3 `c=*i;`, where ℓ_i ($i = 1, 2, 3$) is a unique statement identifier. Let $m_{\ell_{i-1}}$ (m_{ℓ_i}) be the state of the memory before (after, resp.) the execution of the command labelled with ℓ_i ($i = 1, 2, 3$). We will have the following conjunction of literals characterising the state of the memory after the execution of the program fragment above:

$$\begin{aligned} & m_{\ell_1} = \mathsf{wr}(m_{\ell_0}, \mathsf{rd}(m_{\ell_0}, \mathtt{i}), 3) \wedge m_{\ell_2} = \mathsf{wr}(m_{\ell_1}, \mathsf{rd}(m_{\ell_1}, \mathtt{j}), 2) \\ \wedge\ & m_{\ell_3} = \mathsf{wr}(m_{\ell_2}, \mathtt{c}, \mathsf{rd}(m_{\ell_2}, \mathsf{rd}(m_{\ell_2}, \mathtt{i}))) \end{aligned} \tag{11}$$

By using the set $Ax(\mathcal{A})$ and (11), it is easy to see that after the execution of the three statements above, `c` gets the value of `3` if `i`$\neq$`j` and `2`, otherwise.

The linked list data type `list` declared above is modelled by using the theory of lists, denoted with $\mathcal{L}$. The unary function symbols car and cdr, the binary function symbol cons, and the constant symbol null are in the signature of $\mathcal{L}$. The set $Ax(\mathcal{L})$ of the axioms of $\mathcal{L}$ contains $\mathsf{car}(\mathsf{cons}(X, Y)) = X$, $\mathsf{cdr}(\mathsf{cons}(X, Y)) = Y$, $\mathsf{null} \neq \mathsf{cons}(X, Y)$, where X, Y are implicitly universally quantified variables. In order to guarantee that $?$ is a distinguished value, we need the following set $\Theta := \{? \neq \mathsf{cons}(X, Y), \mathsf{null} \neq ?, v_i \neq ?, v_i \neq \mathsf{null} \mid i = 1, ..., n\}$ of axioms, where

$$
\begin{array}{rl}
\text{Lvalue} & \left\{\begin{array}{rcl}
\mathcal{S}_\lambda(\texttt{name}, m) & = & \texttt{name} \\
\mathcal{S}_\lambda(*t, m) & = & \mathsf{rd}(m, \mathcal{S}_\lambda(t))
\end{array}\right. \\
\text{Expression} & \left\{\begin{array}{rcl}
\mathcal{S}_\epsilon(\texttt{var}, m) & = & \mathsf{rd}(m, \texttt{var}) \\
\mathcal{S}_\epsilon(\texttt{const}, m) & = & \texttt{const} \\
\mathcal{S}_\epsilon(*e, m) & = & \mathsf{rd}(m, \mathcal{S}_\epsilon(e)) \\
\mathcal{S}_\epsilon(e.\texttt{cdr}, m) & = & \mathsf{cdr}(\mathcal{S}_\epsilon(e))
\end{array}\right. \\
\text{Statement} & \left\{\begin{array}{rcll}
\mathcal{S}_\sigma(s;s', m) & = & \mathcal{S}_\sigma(s', \mathcal{S}_\sigma(s)) & \\
\mathcal{S}_\sigma(\texttt{if}\, e\, \texttt{then}\, s\, \texttt{else}\, s', m) & = & \mathsf{ite}(\mathcal{S}_\epsilon(e, m), \mathcal{S}_\sigma(s), \mathcal{S}_\sigma(s')) & \\
\mathcal{S}_\sigma(t{=}e, m) & = & \mathsf{wr}(m, \mathcal{S}_\lambda(t), \mathcal{S}(e)) & \\
\mathcal{S}_\sigma(t{=}\,\texttt{malloc}(\texttt{sizeof}(\texttt{list})), m) & = & \mathsf{wr}(m, \mathcal{S}_\lambda(t), c) & (c \text{ is a fresh constant}) \\
\mathcal{S}_\sigma(\texttt{free}(t), m) & = & \mathsf{wr}(m, \mathcal{S}_\lambda(t), ?) &
\end{array}\right.
\end{array}
$$

Fig. 1. Symbolic execution semantics for debugging.

$$
\begin{array}{rcll}
\mathcal{V}_\pi(\texttt{require}\, e;s, m_0) & = & m \Rightarrow v & \text{where } \langle m, v\rangle = \mathcal{V}_\sigma(s, \langle \mathcal{S}_\epsilon(e, m_0), \top\rangle) \\
\mathcal{V}_\sigma(\texttt{ensure}\, e, \langle m, v\rangle) & = & \langle m, v \wedge \mathcal{S}_\epsilon(e, m)\rangle & \\
\mathcal{V}_\sigma(s, \langle m, v\rangle) & = & \langle \mathcal{S}_\sigma(s, m), v\rangle, & (\text{if } s \text{ is not a } \textit{require}) \\
\mathcal{V}_\sigma(s_1 s_2, \langle m, v\rangle) & = & \mathcal{V}_\sigma(s_2, \mathcal{V}_\sigma(s_1)) &
\end{array}
$$

Fig. 2. Proof obligation generation for debugging.

X, Y are implicitly quantified first-order variables, $v_1, ..., v_n$ are the variables declared in the program (represented as first-order constants), and the constant null represents the null pointer. An excerpt of our semantics for debugging of programs manipulating `lists` is given in Figure 1. It should be clear that such a semantics is a mapping which associates a first-order term to a program π. The term represents the state of the memory after the execution of π in terms of m_0, the state of the memory before π begins its execution.

Annotations for debugging. As already noted above, we allow programs to be annotated with pre- and post-conditions. Indeed, this is already interesting since the programmer has the freedom to specify a wide range of properties. However, annotations denoting frequently occurring programming errors can be automatically added to programs. Below, we consider three such properties which go under the name of *cleanness conditions* (see, e.g. [8]). First, **read undefined** is checking whether *every variable has been assigned before it is used.* This condition is taken in consideration by adding the annotation $\mathsf{rd}(m, l) \neq ?$ before a statement S in which the program variable l occurs in an expression (where m is the state of the

memory before the execution of S). Second, **null dereferencing** is checking that *no dereferenced pointer is equal to null*. To consider this condition, one should add the annotation $\mathsf{rd}(m, p) \neq \mathsf{null}$ before any command dereferencing a pointer p. Third, **memory leakage** is more complex: *when a pointer is "killed", either its value is null or undefined, or another pointer points to the same location.*[7] The assertion for this condition is that, either the value of the pointer variable p is one of null or $?$, or that there is a variable p' (distinct from p) such that the value of p' is the same as that of p (i.e. there is aliasing). It is easy to write a predicate which characterises a state of the memory without aliasing as $\forall m, p.(\mathsf{noalias}(m, p) \Leftrightarrow \forall p'.(p' \neq p \Rightarrow \mathsf{rd}(m, p) \neq \mathsf{rd}(m, p')))$, so that the condition for memory leakage can be written as $\mathsf{noalias}(m, p) \vee \mathsf{rd}(m, p) = ? \vee \mathsf{rd}(m, p) = \mathsf{null}$.

Proof obligations for debugging. We are left with the problem of extracting the proof obligations from the annotated (either automatically or by the user) program. To this end, we apply the function $\mathcal{V}_\pi$, defined in Figure 2, which generates such proof obligations by exploiting the symbolic execution semantics defined above. $\mathcal{V}_\pi$ takes as input an annotated program and a first-order formula (containing m_0) which encodes a description of the initial state of the memory and it returns a first-order formula ϕ whose invalidity signals a bug. Notice that if ϕ is valid, then we are only allowed to conclude that the program is bug-free for all the executions of finite length which can be obtained by unrolling the loops in the program a given number of times.

Summary of the approach to debugging. Let π be an annotated program containing only one loop (for simplicity) and n a natural number. First, we unroll n-times the loop in π and we obtain a loop-free program π'. Let ϕ be the first-order formula returned by $\mathcal{V}_\pi(\pi', m_0)$. It remains to check that

$$\mathcal{T} \models \phi, \tag{12}$$

where $Ax(\mathcal{T}) =_{\text{def}} Ax(\mathcal{A}) \cup Ax(\mathcal{L}) \cup \Delta \cup \Upsilon \cup \Theta \cup \Lambda$ is the axiomatisation of the background theory, and $\mathcal{A}$ is the theory of arrays, $\mathcal{L}$ is the theory of lists, Δ encodes the pairwise distinctness of program variables, Υ characterises the initial value of the store, Θ characterises the special symbol $?$, and Λ is a (possibly empty) set of properties which the user adds in order to specify some symbols used in the assertions of the programs. Some remarks are in order. First, ϕ may contain ite's even at the term level and it can be huge (the size is increasing with the number n of loop unrolling considered) thereby making it difficult (from a computational viewpoint) to apply standard automated theorem proving techniques. In fact, as already remarked in [7] and confirmed by our experiments (see Section 4), the approach of eliminating ites in favour of the traditional logical connectives and then translating the negation of the resulting formula to conjunctive normal form is not viable in practice, although theoretically possible. This suggests that only selected sub-problems should be tackled by automated theorem proving. Second,

[7] By definition, a pointer p is said to be killed whenever it gets out of scope, or it is parameter to a C `free` statement, or it is assigned a new value.

checking the entailment in (12) requires a high-degree of flexibility w.r.t. reasoning in the background theory since Λ can vary widely according to the needs of the user. This level of flexibility is not provided by most state-of-art validity checkers.

3 Light-Weight Theorem Proving

We design a technique, which we call *light-weight theorem proving*, to efficiently check the entailment in (12). Our method is based on refutation, i.e. we will check the unsatisfiability of $Ax(\mathcal{T}) \cup \{\neg\phi\}$, where $Ax(\mathcal{T}) := Ax(\mathcal{A}) \cup Ax(\mathcal{L}) \cup \Delta \cup \Upsilon \cup \Theta \cup \Lambda$. A direct consequence of (9) which is useful for (un-)satisfiability testing is

$$\neg\mathsf{ite}(A, B, C) \text{ rewrites to } (A \wedge \neg B) \vee (\neg A \wedge \neg C). \tag{13}$$

Notice that ite on the propositional level is a logical basis (see e.g. [20]) and that $\neg\phi$ can be assumed to be quantifier-free since quantified-subformulae can be eliminated by renaming (this is sufficient for refutation, see e.g. [22]). We call *proof obligation* the pair $(Ax(\mathcal{T}), \neg\phi)$. Our flexible and efficient technique to discharge proof obligations of the form $(Ax(\mathcal{T}), \neg\phi)$ consists of three steps. First, *lift* ites at the term to the propositional level by exhaustively applying (10) to $\neg\phi$, thereby obtaining a formula ϕ' where the only occurrences of ites are at the propositional level. For example, $a = \mathsf{ite}(b = c, d, e)$ is converted to $\mathsf{ite}(b = c, a = d, a = e)$. Second, *build the BDD* of ϕ' by abstracting the ground (first-order) atoms of ϕ' to propositional letters. Let β be such a BDD containing ite as the only logical connective. It is easy to see that the first argument of any ite in β is an atom. Third, exhaustively apply (13) to $\neg\beta$ (recall that negation can be done in constant time with BDDs) to build its disjunctive normal form (DNF) and then check the unsatisfiability of each disjunct w.r.t. the background theory $\mathcal{T}$. In our method, the adjective *light-weight* is added to theorem proving since automated deduction techniques are used only on selected sub-tasks, identified by exploiting the boolean structure of the formula. Some important remarks are in order.

First, an efficient implementation of lifting is mandatory to attack real verification problems. This is not difficult since standard coding techniques such as sharing of common expressions and memoization of intermediate computations can be used. Second, the number of paths in a BDD can be exponential in the number of atoms occurring in it. As a consequence, in the third step of our technique, a very large number of proof obligations can be generated for the prover. In order to avoid this problem we interleave the generation of a proof obligation with the activity of pruning those paths in the BDD which are unsatisfiable modulo the background theory. Our method for pruning unsatisfiable paths is inspired by [11] and it is related to tableaux with lemmata (see e.g. [20]). Let us consider the BDD β of $\neg\phi$ and assume that one of its disjuncts (say δ) has been proved unsatisfiable. In this case, the superposition prover has produced a proof of the empty clause from $Ax(\mathcal{T}) \cup \delta$. Such a proof contains a "small" (in general, not guaranteed to be minimal) set of the literals in δ which is necessary to derive the empty clause. Such literals can be used as constraints to simplify β by exploiting standard BDD

techniques (see, e.g. [14]). In many verification problems, pruning subsumed paths modulo $\mathcal{T}$ can greatly reduce the number of proof obligations sent to the prover, thereby dramatically improving performances.

Third, we need to provide an automatic and efficient support to reasoning in the background theory $\mathcal{T}$. This is not easy since $\mathcal{T}$ is usually an extension of a combination of some theories which are ubiquitously used in verification with the properties of user-defined function or predicate symbols. Widening the scope of applicability of decision procedures to take into account user-defined symbols is a challenging problem since the seminal work [5] of Boyer and Moore on the integration of a decision procedure in rewriting. In the rewriting approach to satisfiability procedures described in [2], it seems particularly easy to extend the procedures. In fact, a satisfiability procedure amounts to the exhaustive application of the rules of the superposition calculus [19,3] (implemented in many state-of-art automated theorem provers) to $Ax(\mathcal{T})$ and the checking of the ground literals for (un-)satisfiability. Termination of this process is shown in [2] for some equational theories such as the theory of arrays and of lists. In order to obtain automatic support for the background theory extended with user-defined properties, we simply add such definitions to the axioms of the decidable theory. Although the theoretical results in [2] about the termination of saturation do not hold for the extended theory, it turns out that, in practice, termination is frequently achieved. It is well-known [4] that definitions dramatically enlarge the search space of theorem provers. However, in our experiments, the naïve clausification of definitions turned out to be sufficient to obtain high performances. Furthermore, not only definitions but also lemmas about user-defined symbols can be used in our approach. This allows us to use first-order approximation of inductive predicates which are frequently sufficient to prove many interesting properties. For example, in [18], a set of eight axioms is listed which characterises the inductive predicate encoding the reachability of an element in a singly linked list and the correctness of a Union-Find algorithm is proved with such set (cf. Table 2).

Summary of light-weight theorem proving. Let $\neg\phi$ be a first-order ground formula (possibly containing ites) and $Ax(\mathcal{T})$ be a set of first-order equational clauses. Our light-weight theorem proving procedure can be summarised as follows.

First step. Exhaustively apply the rule $B[\mathsf{ite}(A, t_1, t_2)] \rightarrow \mathsf{ite}(A, B[t_1], B[t_2])$ to $\neg\phi$. Let ϕ' be the resulting formula which does not contain any occurrence of ite in a first-order term.

Second step. Abstract each atom in ϕ' to a "fresh" propositional letter and then build a BDD of such abstracted formula. Let ϕ'' be the resulting BDD and f be the bijective mapping associating a ground atom with a new propositional letter.

Third step. For each path δ from the root of the BDD ϕ'' to the node labelled by true (intended conjunctively):

- exhaustively apply the rules of the superposition calculus to $f^{-1}(\delta) \cup Ax(\mathcal{T})$;[8] let $\mathcal{C}$ be the resulting set of clauses;

[8] $f^{-1}(\delta)$ abbreviates the set $\{l \mid p \in \delta \text{ and } l = f(p)\}$.

- if the empty clauses is not in $\mathcal{C}$, then exit the loop and return that ϕ is not valid.
  Otherwise, consider a proof P of the empty clause and the set π of (unit) clauses which are in $P \cap \delta$. Simplify ϕ'' under the assumption that each literal in π is false. Let ϕ''' be the resulting BDD. Repeat the third step by assigning ϕ''' to ϕ''.

It is easy to see that the procedure above terminates if the process of closing the set $f^{-1}(\delta) \cup Ax(\mathcal{T})$ under the rules of the superposition calculus terminates. This is frequently the case in practice since the clauses in $f^{-1}(\delta) \cup Ax(\mathcal{T})$ do not contain a lot of redundant information which is present in the whole conjunctive normal form of $\neg\phi$ and $Ax(\mathcal{T})$. Furthermore, such redundancy is eliminated by reducing the number of paths in the BDD by simplifying it under the assumption that the literals in a (usually small) subset of $f^{-1}(\delta)$ are $\mathcal{T}$-unsatisfiable.

Implementation. We have implemented the three-step method described above in a system called haRVey [9] which integrates the ATerm library [10] to implement the lifting of ites from the term to the propositional level, D. Long's library [11] for the BDD construction and simplification, and the E prover [12] for checking the unsatisfiability modulo $\mathcal{T}$. The input syntax of haRVey is LISP-like. It takes as input a proof obligation of the form $(Ax(\mathcal{T}), \neg\phi)$ and it returns whether the formula is valid or a "counter-example", namely a set of literals which is satisfiable. The E prover is invoked on a proof obligation in automatic mode. So far in our experiments, we did not feel the need to specify particular strategies for saturation to obtain high performances, thereby achieving one of the goal of the present work, i.e. making superposition theorem proving usable also by non-experts.

4 Results

To evaluate the flexibility and the effectiveness of our approach, we consider two different test sets. The former is generated (following the method of Section 2) by symbolically debugging the reference programs which manipulate linked lists of [8,15] (c.f. Table 1). This set of proof obligations allows us to investigate the scalability of our approach. Larger and larger formulae are obtained by simply unrolling the loops in the programs an increasing number of times (indicated by the superscript after the name of the program in the first column of Table 1). More precisely, we consider two issues: *user-defined properties* (cf. Table 1 (a)) and *cleanness properties* (cf. Table 1 (b),(c), and (d)). As an example of the user-defined property, we have considered the property expressing the fact that all the cells containing a certain value in a list are removed from the list by the program named $remove$.

[9] `http://www.loria.fr/equipes/cassis/softwares/haRVey`
[10] `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ATermLibrary`
[11] `http://www-2.cs.cmu.edu/~modelcheck/bdd.html`
[12] `http://www4.informatik.tu-muenchen.de/~schulz/WORK/eprover.html`

Table 1
Experiments with **debugging**

(a) user-defined property

	haRVey	E	*Simplify*
search[1]	0.08	**0.03**	**0.03**
search[2]	0.17	**0.17**	**0.10**
search[3]	0.68	**0.16**	0.71
search[4]	**0.16**	0.36	1.78
search[5]	**0.17**	1.22	21.36
search[6]	**0.23**	5.48	mem.out
remove[1]	0.15	**0.02**	**0.02**
remove[2]	0.19	0.39	**0.14**
remove[3]	**0.21**	11.40	2.18
remove[4]	**0.37**	40.55	mem.out

(b) read undefined

	haRVey	E	*Simplify*
fumble[3]	0.06	1.62	**0.04**
insert[3]	0.34	111.89	**0.29**
merge[2]	**0.19**	time out	3.61
remove[3]	**0.04**	time out	3.46
remove-all[3]	0.06	**0.03**	**0.03**
reverse[3]	0.26	2.64	**0.08**
rotate[3]	0.48	10.80	**0.11**
search[3]	**0.33**	145.40	0.79
swap[3]	0.21	0.10	**0.04**

(c) null pointer dereferencing

	haRVey	E	*Simplify*
insert[6]	**0.83**	time out	3.22
merge[4]	**5.83**	sp. out	41.71
remove[4]	**0.05**	sp. out	9.69
remove-all[3]	0.06	0.05	**0.02**
reverse[8]	**0.20**	time out	0.51
rotate[8]	**0.39**	time out	0.85
swap[3]	0.25	0.12	**0.04**

(d) memory leakage

	haRVey	E	*Simplify*
fumble[2]	**0.10**	time out	38.08
insert[3]	**0.07**	time out	25.93
merge[1]	**0.48**	sp. out	mem. out
remove[2]	**0.11**	sp. out	mem. out
remove-all[3]	**0.05**	508.14	0.65
reverse[2]	**1.20**	sp. out	mem. out
rotate[2]	**1.68**	sp. out	mem. out
search[3]	**0.05**	time out	3.64
swap[3]	**0.05**	time out	1.17

Legenda: Timings are in seconds and are collected on a Pentium IV 2GHz running Linux. "time out" means that execution time is > 600 seconds, "mem. out" that main memory usage is > 256 Mb., and "space out" that the disk space used is > 1 Gb. For haRVey, time includes lifting, BDD manipulations, and saturations. For E and *Simplify*, time includes only the activity of checking the unsatisfiability of the negation of the formula without lifting and the activity of renaming sub-expressions, which was necessary for *Simplify* to handle some proof obligations.

Table 2
Experiments with **verification**

	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12
haRVey	0.03	0.06	0.03	0.03	0.06	0.14	0.11	0.33	0.16	2.46	0.31	1.50
E prover	0.03	0.03	0.10	23.16	0.03	0.04	0.04	0.18	time out	time out	0.13	0.65
Simplify	—	0.01	—	0.04	0.01	0.01	0.20	—	0.01	—	—	—

Legenda: Timings are in seconds and are collected on a Pentium IV 2GHz running Linux. "—" means that *Simplify* fails to check the validity (w.r.t. the background theory) of the formula.

The proof obligations in the second set were obtained in [18] while verifying the correctness of a Union-Find program (c.f. Table 2). Also in this case, linked lists are manipulated by the program but their axiomatisation is different from the one described in Section 2. Our approach allows the necessary flexibility to efficiently handle also this situation. As a further complication, an inductive predicate

characterising reachability is extensively used in the specification of the correctness of the Union-Find algorithm. To reason about such a predicate, we use the list of first-order axioms given in [18] which turns out to be sufficient in many situations. Notice that, although the proof obligations listed in [18] are smaller than those considered in the first test set, they require non-trivial handling of the quantifiers for the axioms characterising reachability.

We compare haRVey against the E prover (version 0.7), and *Simplify*, the well-engineered validity checker at the heart of the ESC/Java program analysis tool [10]. It is important to notice that *Simplify* is one of the few validity checkers capable of handling the proof obligations considered in our experiments as they are, thanks to its heuristics mechanism to find instantiations of axioms added to the background theory. As already noted, other checkers (such as CVC or ICS) cannot directly handle our proof obligations since they are limited to their built-in theories.

In all but the simplest problems of Table 1, haRVey outperforms the E prover, and performs significantly better than *Simplify*. Furthermore, haRVey successfully checks all proof obligations in Table 2 whereas *Simplify* returns false negatives for half of them. These experiments show that light-weight theorem proving offers a higher degree of automation and a better scalability than state-of-the-art validity checkers on proof obligations arising in the context of program debugging and verification. It also offers a new context where to apply refutation theorem provers without asking the user to fine tune their strategies, which is an untenable requirement for "push-button" applications.

5 Conclusion and Future Work

We have described a combination of BDDs and superposition theorem proving to automate the reasoning activity required to debug and verify imperative programs which manipulate pointers, in a flexible and efficient manner. An implementation of the technique compares well with state-of-the-art tools like the E prover and *Simplify*.

At the time of writing, we are using haRVey to discharge the proof obligations arising in the verification of safety properties of B machines [1]. This requires to reason in a fragment of set theory which is considered challenging for automated reasoning tools. Preliminary experiments show encouraging results.

Our future work will focus on three issues. First, we want to integrate arithmetic reasoning by using some well-known combination techniques like the Nelson and Oppen schema in order to handle pointer arithmetics. Second, we will interleave the construction of the BDD with superposition theorem proving so to tackle larger proof obligations. Third, we plan to integrate model building in our technique so that a counter-example can be returned when a formula is not proved valid.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] Alessandro Armando, Silvio Ranise, and Michaël Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, to appear, 2003.

[3] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. of Logic and Comp.*, 4(3):217–247, 1994.

[4] R. Boyer, E. Lusk, W. McCune, R. Overbeek, and M. Stickel nad L. Wos. Set Theory in First-Order Logic: Clauses for Godel's Axioms. *J. of Automated Reasoning*, 2:287–326, 1986.

[5] R.S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. *Machine Intelligence*, 11:83–124, 1988.

[6] R. E. Bryant and M. N. Velev. Processor Verification Using Efficient Reductions of the logic of Uninterpreted Functions to Propositional Logic. *ACM Trans. on Computational Logic (TOCL)*, 2(1):93–134, 2001.

[7] K. Claessen, R. Hähnle, and J. Maartensson. Verification of Hardware Systems with First-Order Logic. In *In VERIFY'02 (Workshop affiliated to FloC'02*, 2002.

[8] N. Dor, M. Rodeh, and S. Sagiv. Checking Cleanness in Linked Lists. In *Static Analysis Symposium*, pages 115–134, 2000.

[9] Peter J. Downey and Ravi Sethi. Assignment commands with array references. *Journal of the Association of Computing Machinery*, 25(4):652–666, October 1978.

[10] C. Flanagan, K. R. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proc. of the 2002 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI), Berlin, Germany*, pages 234–245, 2002.

[11] Pascal Fontaine and E. Pascal Gribomont. Using bdds with combinations of theories. In *9th Intl. Conf. on Logic for Programming and Automated Reasoning (LPAR'2002)*, 2002.

[12] E. P. Gribomont and P. Fontaine. Decidability of Invariant Validation for Parameterized Systems. In *Proc. of 9th Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, 2003. To appear.

[13] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12(10):156–164, 1969.

[14] Y. Hong, P. Beerel, Jerry Burch, and Kenneth McMillan. Safe bdd minimization using don't cares. In 34^{th} *Design Automation Conference*, 1997.

[15] D. Jackson and M. Vaziri. Finding Bugs with a Constraint Solver. In *Proc. of International Symposium on Software Testing and Analysis*, Portland, OR, USA, 2000.

[16] J. C. King. Symbolic Execution and Program Testing. *Comm. ACM*, 19(7):385–394, 1976.

[17] G. Nelson. Techniques for Program Verification. Technical Report CSL–81–10, Xerox, Palo Alto Research Center, 1981.

[18] G. Nelson. Verifying Reachability Invariants of Linked Structures. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages (POPL'83)*, pages 38–47, Austin, Texas, USA, 1983.

[19] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Hand. of Automated Reasoning*. 2001.

[20] J. Posegga and P. H. Schmidt. Automated Deduction with Shannon Graphs. *J. of Logic and Computation*, 5(6):697–729, 1995.

[21] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of the 17th IEEE Symposium on Logic in Computer Science (LICS'02)*, Copenhagen, Denmark, 2002.

[22] C. Weidenbach and A. Nonnengart. Small clause normal form. In A. Robinson and A. Voronkov, editors, *Hand. of Automated Reasoning*. 2001.

Exact Algorithms for MAX-SAT

Hantao Zhang [1,2] Haiou Shen [3]

Computer Science Department
The University of Iowa
Iowa City, IA 52242, USA

Felip Manyà [4]

Department of Computer Science
Universitat de Lleida
Jaume II, 69, 25001-Lleida, Spain

Abstract

The maximum satisfiability problem (MAX-SAT) is stated as follows: Given a Boolean formula in CNF, find a truth assignment that satisfies the maximum possible number of its clauses. MAX-SAT is MAX-SNP-complete and received much attention recently. One of the challenges posed by Alber, Gramm and Niedermeier in a recent survey paper asks: Can MAX-SAT be solved in less than 2^n "steps"? Here, n is the number of different variables in the formula and a step may take polynomial time of the input. We answered this challenge positively by showing that a popular algorithm based on branch-and-bound is bounded by $O(b2^n)$ in time, where b is the maximum number of occurrences of any variable in the input.

When the input formula is in 2-CNF, that is, each clause has at most two literals, MAX-SAT becomes MAX-2-SAT and the decision version of MAX-2-SAT is still NP-complete. The best bound of the known algorithms for MAX-2-SAT is $O(m2^{m/5})$, where m is the number of clauses. We propose an efficient decision algorithm for MAX-2-SAT whose time complexity is bound by $O(n2^n)$. This result is substantially better than the previously known results. Experimental results also show that our algorithm outperforms any algorithm we know on MAX-2-SAT.

[1] Partially supported by the National Science Foundation under Grant CCR-0098093.
[2] Email: hzhang@cs.uiowa.edu
[3] Email: hshen@cs.uiowa.edu
[4] Email: felip@eup.udl.es
This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

In recent years, there has been considerable interest in the satisfiability problem (SAT) and the maximum satisfiability problem (MAX-SAT) of propositional logic. The input instance is a boolean formula in conjunctive normal form (CNF), and the problem is to find a truth assignment that satisfies all the clauses for SAT and the maximum number of clauses for MAX-SAT. The decision version of MAX-SAT is NP-complete, even if the clauses have at most two literals (so called the MAX-2-SAT problem). One of the major results in theoretical computer science in recent years is that if there is a polynomial time approximation scheme for MAX-SAT, then P = NP [5].

Because the MAX-SAT problem is fundamental to many practical problems in computer science [14] and electrical engineering [23], efficient methods that can solve a large set of instances of MAX-SAT are eagerly sought. One important application of MAX-2-SAT is that NP-complete graph problems such as Maximum Cut and Independent Set can be reduced to special instances of MAX-2-SAT [8,18]. Many of the proposed methods for MAX-SAT are based on approximation algorithms [9], some of them are based on branch-and-bound methods [14,7,6,16,15,13,20], and some of them are based on transforming MAX-SAT into SAT [23,2].

Regarding the problems for formulas in CNF, most authors consider bounds with respect to three parameters: the length L of the input formula (i.e., the number of literals in the input), the number m of its clauses, and the number n of variables occurring in it. The best currently known bounds for SAT are $O(2^{m/3.23})$ and $O(2^{L/9.7})$ [16]. Nothing better than trivial $O(m2^n)$ is known when n is concerned. However, for 3-SAT, we have algorithms of complexity $O(m1.481^n)$ [9]. For MAX-SAT, the best bounds are $O(L2^{m/2.36})$ and $O(L2^{L/6.89})$ [6]. For MAX-2-SAT, the best bounds have been improved from $O(m2^{m/3.44})$ [6], to $O(m2^{m/2.88})$ [20], and recently to $O(m2^{m/5})$ [13]. It was posed as an open problem in [19,1,13] to seek exact algorithms bounded by $O(L2^n)$ for MAX-SAT.

Since an algorithm which enumerates all the 2^n assignments and then counts the number of true clauses in each assignment will take exactly time $O(L2^n)$, we assume that the challenge posed in [19,1,13] would ask for an algorithm better than $O(L2^n)$. In this paper, we present a simple algorithm based on branch-and-bound whose time complexity is only $O(b2^n)$, where b is the maximum number of occurrences of any variable in the input. Typically, $b \simeq L/n$. In particular, for MAX-2-SAT, $b \leq 2n$ and the bound becomes $O(n2^n)$. When $m = 4n^2$, the bound is $O(\sqrt{m}1.414^{\sqrt{m}})$, which is substantially better than the best known bound $O(m2^{m/5})$ [13].

Our branch-and-bound algorithm works in a similar way as the well-known Davis-Putnam-Logemann-Loveland (DPLL) procedure [10,11], which is a depth-first search procedure. To the best of our knowledge, there are only three implementations of exact algorithms for MAX-SAT that are variants of the DPLL method. One is due to Wallace and Freuder (implemented in Lisp) [22], one is due

to Borchers and Furman [7] (implemented in C and publicly available) and the last is due to Alsinet, Manyà and Planes [3] (a substantial improvement over Borchers and Furman's implementation). However, no attempts were made to analyze the complexities of the algorithms used in these implementations [22,7,3].

In this paper, we will present a rigorous analysis of exact algorithms for MAX-SAT and MAX-2-SAT. This kind of analysis, missing in [22,7,3], was used in [6,20,13] but the results presented there are not as strong as ours. It involves the design of data structures for clauses and the implementations of various operations on clauses. We also present experimental results to show that our algorithms are faster than other known algorithms not only in theory but also in practice.

2 Preliminary

We assume the reader is familiar with the basic concepts of the SAT problems, such as variable, literal, clause, CNF, assignment, and satisfiability. For every literal x, we use variable(x) to denote the variable appearing in x. That is, variable$(x) = x$ for positive literal x and variable$(\overline{x}) = x$ for negative literal $\overline{x}$. If y is a literal, we use $\overline{y}$ to denote x if $y = \overline{x}$. A literal x in a set S of clauses is said to be *pure* in S if its complement does not appear in S. A partial (complete) assignment can be represented by a set of literals (or unit clauses) in which each variable appears at most (exactly) once and each literal is meant to be true in the assignment. If a variable x does not appear in a partial assignment A, then we say literals x and $\overline{x}$ are *unassigned* in A.

Given a set S of clauses and a literal x, we use $S[x]$ to denote the set of clauses obtained from S by removing the clauses containing x from S and removing $\overline{x}$ from the clauses in S. Given a set of literals $A = \{x_1, x_2, ..., x_k\}$, $S[A] = S[x_1][x_2] \cdots [x_n]$. Given a variable x, let $\#(x, S)$ be the number of clauses containing x (either positively or negatively).

3 A Branch-and-Bound Algorithm for MAX-SAT

Before presenting an algorithm for MAX-SAT, we make the following assumption. We regard the input clauses S_0 as a multiset. When some literals are assigned a truth value, this multiset is simplified by removing the assigned literals. If a unit clause is generated in this process, we remove this unit clause from the multiset and store this information in one variable, $u(x)$, associated with each literal x. That is, $u(x)$ records the numbers of unit clauses x generated during the search. If there are no unit clauses in the input, these variables are initialized to zero.

Our new algorithm for MAX-SAT is illustrated in Fig. 1. The variable min_false_clauses in the function branch_bound_max_sat and the recursive procedure bb_max_sat is a global variable and is initialized with a maximum integer. In practice, this variable can be initialized with the minimum number of false clauses found by a local search procedure [7]. After the execution of bb_max_sat,

Fig. 1. A branch-and-bound algorithm for MAX-SAT.

```
function branch_bound_max_sat ( S: clause set ) return integer
    // initiation
    for each literal y do u(x) := 0; end for
    min_false_clauses := MAX_INT;
    bb_max_sat(S, 0, ∅);
    return min_false_clauses;
end function

procedure bb_max_sat ( S: clause set, k: integer, A: assignment )
    if |A| = n then // n is the number of variables.
        if (k < min_false_clauses) then
            print_model(A);
            min_false_clauses := k;
        end if
    else
        pick an unassigned literal x in S;
        // decide if we want to set literal x to false
        if (is_not_pure(x) ∧ u(x) + k < min_false_clauses) then
            S' := record_unit_clauses(S[x̄]);
            bb_max_sat(S', k + u(x), A ∪ {x̄});
            undo_record_unit_clauses(S[x̄]);
        end if
        // decide if we want to set literal x to true
        if (u(x̄) + k < min_false_clauses) then
            S' := record_unit_clauses(S[x]);
            bb_max_sat(S', k + u(x̄), A ∪ {x});
            undo_record_unit_clauses(S[x]);
        end if
    end if
end procedure

function record_unit_clauses ( S: clause set ) return clause set
    for each unit clause y ∈ S do S := S − {y}; u(y) := u(y) + 1; end for;
    return S;
end function

procedure undo_record_unit_clauses ( S: clause set )
    for each unit clause y ∈ S do S := S − {y}; u(y) := u(y) − 1; end for;
end procedure
```

min_false_clauses records the minimum number of false clauses under any assignment.

The parameter A in bb_max_sat(S, k, A) records a set of literals as a partial assignment (making every literal in A true). Note that A can be omitted if there is no need to print out an assignment whenever a better assignment is found. However, A is useful in the analysis of this algorithm.

The algorithm presented in Fig. 1 is certainly not the most efficient as many optimization techniques such as the pure-literal lookahead rule [21] or the *dominating unit-clause rule* [20] can be used. We made our best effort to present it as simple as possible so that it is easy to analyze.

Theorem 3.1 *Suppose min_false_clauses is initialized with a maximum integer. Then after the execution of* bb_max_sat$(S_0, 0, \emptyset)$*, min_false_clauses records the minimum number of false clauses in S_0 under any assignment.*

Proof. Let us consider the following pre-conditions of bb_max_sat(S, k, A).

- A is a partial assignment for variables appearing S_0.
- Let the multiset $S_0[A]$ be divided into empty clauses E, unit clauses U and non-unit clauses N. Then
  (i) $S = N$;
  (ii) $u(y)$ is the number of unit clauses y in U for any unassigned literal y under A;
  (iii) $k = |E|$, the number of false clauses in S_0 under A.

At first, these conditions are true for the first call bb_max_sat$(S_0, 0, \emptyset)$, assuming neither unit clauses nor empty clauses are in S_0. Suppose these conditions are true for bb_max_sat(S, k, A). If we want to assign the literal x to true, then this action creates $u(\overline{x})$ empty clauses. If $u(\overline{x}) + k \geq$ min_false_clauses, then the current A will not lead to a better assignment. Otherwise, we will try to extend A further by assigning x to true and compute $S' = S[x]$. The procedure record_unit_clauses will update $u(y)$ for the newly created unit clauses for each literal y and remove these unit clauses from S'. It is easy to see that the pre-conditions for the call bb_max_sat$(S, k + u(\overline{x}), A \cup \{x\})$ are all true. Similarly, the pre-conditions for the call bb_max_sat$(S, k + u(x), A \cup \{\overline{x}\})$ are all true when we assign literal x to false. By an inductive reasoning, the preconditions are true for any multiset S. Finally when $|A|$ becomes empty, A is a complete assignment for S_0: If $k <$ min_false_clauses, then A is a better assignment and we update min_false_clauses by k accordingly.

The search conducted by bb_max_sat is an exhaustive one because every assignment is tried except the cases when we know the number of false clauses under that assignment exceeds min_false_clauses. This justifies the correctness of bb_max_sat(S, k, A). □

Besides the correctness of the algorithm, we show that the time complexity of the algorithm is bounded by $O(b2^n)$, where b is the maximum number of occurrences of any variable. For any variable x, recall that $\#(x, S)$ is the number of

occurrences of x (both positively and negatively) in S.

Theorem 3.2 *If it takes $O(b)$ to* pick an unassigned literal x in S*, then the time complexity of branch_bound_max_sat(S_0) is $O(b2^n)$, where n is the number of variables in S_0 and $b = \max_x \#(x, S_0)$, the maximum number of occurrences of any variable in S_0.*

Proof. The number of calls to bb_max_sat is bounded by 2^n because the tree representing the relation between recursive calls of bb_max_sat is a binary tree (each internal node has at most two children) and the height of the tree is n.

We need to show that computing $S[x]$ and identifying new unit clauses in $S[x]$ can be done in $O(\#(x, S))$. To achieve this, we can use the data structure suggested by Freeman [12] as follows: For each clause $c \in S_0$, let count(c) be the number of unassigned literals in c and flag(c) be true if and only if one of the literals of c is assigned true. For each variable v, let pos(v) be the list of clauses in S_0 in which v appears positively and neg(v) be the list of clauses in which v appears negatively. To compute $S[x]$ from S, if x is positive, then for every clause c in pos(x), we assign true to flag(c) and for every clause c in neg(x) such that flag(c) is not true and count$(c) > 1$, we decrease count(c) by one. If count$(c) = 1$ after decreasing, we have obtained a new unit clause. The case when x is negative is similar. So the total cost for computing $S[x]$ and identifying new unit clauses is $O(\#(x, S_0))$.

In other words, record_unit_clauses and undo_record_unit_clauses can be done $O(\#(x, S_0))$. Since $\#(x, S_0)) \geq b$ and there are at most 2^n nodes, the total cost is bounded by $O(b2^n)$. □

Note that popular literal selection heuristics such as MOMS [12] and Jeroslow-Wang (JW) [17] take more than $O(b)$. MOMS is used by Borchers and Furman [7] and JW is used by Alsinet et al. [3] for MAX-SAT. In the next section, we will present an efficient decision algorithm for MAX-2-SAT in which it takes a constant time to select literals.

4 An Efficient Decision Algorithm for MAX-2-SAT

The decision version of MAX-SAT takes the following form:

Instance: A formula S in CNF and a nonnegative integer g.

Question: Is there a truth assignment for the variables in S such that at most g clauses in S are false under this assignment?

It is well-known that if the decision version of MAX-SAT can be solved in time T, then the optimization version of MAX-SAT can be solved in time $lg(m)T$, where m is the number of input clauses. For MAX-2-SAT, $m \leq 4n^2$ if no duplicate clauses are allowed.

Before presenting the algorithm for MAX-2-SAT, we define the following data structure for binary clauses. We assume that the n propositional variables are named (and ordered in the obvious way) from 1 to n. For each variable i, we define the following two sets:

Fig. 2. A decision algorithm for MAX-2-SAT.

```
function max_2_sat2 ( S_0: clause set, n: variable, g_0: integer) return boolean
    // initiation
    for i := 1 to n
        compute B_0(i) and B_1(i) from S_0;
        u(i) := u_i(ī) := 0; // assuming no unit clauses in S_0
    end for
    return dec_max_2_sat(1, g_0, ∅);
end function

function dec_max_2_sat( i: variable, g: integer, A: assignment ) return Boolean
    if i > n then print_model(A); return true; end if // end of the search tree
    // decide if we want to set variable i to true
    if (u(ī) ≤ g) then
        record_unit_clauses(i, 0);
        if (dec_max_2_sat(i + 1, g − u(ī), A ∪ {i}) then return true; end if
        undo_record_unit_clauses(i, 0);
    end if
    // decide if we want to set variable i to false
    if (u(i) ≤ g) then
        record_unit_clauses(i, 1);
        if (dec_max_2_sat(i + 1, g − u(i), A ∪ {ī}) then return true; end if
        undo_record_unit_clauses(i, 1);
    end if
    return false;
end function

procedure record_unit_clauses ( i: variable, s: boolean )
    for y ∈ B_s(i) do u(y) := u(y) + 1 end for;
end procedure

procedure undo_record_unit_clauses ( i: variable, s: boolean )
    for y ∈ B_s(i) do u(y) := u(y) − 1 end for;
end procedure
```

$$B_0(i) = \{y \mid (\bar{i} \vee y) \in S, i < \text{variable}(y)\}$$

$$B_1(i) = \{y \mid (i \vee y) \in S, i < \text{variable}(y)\}$$

Intuitively, $B_0(i)$ is an economic representation of neg(i) and $B_1(i)$ is an economic representation of pos(i). The decision algorithm for MAX-2-SAT is illustrated in Fig. 2.

Theorem 4.1 *Suppose S_0 is a set of binary clauses. Then max_2_sat2(S_0, n, g_0) returns true if and only if there exists an assignment under which at most g_0 clauses*

in S are false.

Proof. The proof is analogous to that of Theorem 3.1. The pre-conditions considered here for dec_max_2_sat(i, g, A) are the following.

- A is a partial assignment for variables 1 to $i-1$.
- g is equal to g_0 minus the number of false clauses in S under A, where g_0 is the parameter in the decision problem and the first call to dec_max_2_sat.
- For any literal y, $i \leq \text{variable}(y) \leq n$, $u(y)$ is the number of unit clauses y in $S_0[A]$.

An inductive proof on i will prove these pre-conditions. □

Theorem 4.2 *The time complexity of dec_max_sat(S_0, n, g_0) is $O(n2^n)$ and the space complexity is $L/2 + O(n)$, where L is the size of the input.*

Proof. The time complexity is analogous to that of Theorem 3.2. Since for MAX-2-SAT, the maximum number of occurrences of any variable is bound by $O(n)$ and the algorithm takes constant time on literal selection, the total time is bounded by $O(n2^n)$.

For the space complexity, since only one literal in each binary clause is stored in the algorithm, we need $L/2$ to store the input. Adding the space for u and local variables in recursive calls gives us the result. □

To the best of our knowledge, the space complexity of other algorithms is bounded by cL, where $c > 1$, for the algorithms in [22,7,3].

5 Experimental Results

To obtain an efficient decision procedure, we have considered several techniques. One such technique is the so-called pure-literal deletion to prune some futile branches. A literal is said to be *pure* in the current clause set if its negation does not occur in the clause set. There are no need to assign false to a pure literal because doing so will not find an assignment which makes less clauses false.

For the data structure used in our algorithm, let us assume $b_0(i) = |B_0(i)|$ and $b_1(i) = |B_1(i)|$. Then a positive literal i is pure if $u(\bar{i}) + b_0(i) = 0$. Similarly, a negative literal $\bar{i}$ is pure if $u(i) + b_1(i) = 0$. We can easily check this condition before branching.

In [20], Niedermeier and Rossmanith used a rule called the *dominating unit-clause rule* to prune some search space. The dominating unit-clause rule can be easily checked using our data structure: There is no need to assign variable i false (true) if $u(i) \geq u(\bar{i}) + b_0(i)$ ($u(\bar{i}) > u(i) + b_1(i)$). This rule covers the pure-literal checking because if literal i is pure, then $u(\bar{i}) + b_0(i) = 0$, hence $u(i) \geq u(\bar{i}) + b_0(i)$.

In dec_max_2_sat(i, g, A), if $(\sum_{j=i}^{n} min(u_i(\bar{j}), u(j))) > g$, then there is no solution (dec_max_2_sat(i, g, A) will return false). We found later that this technique of pruning is also used in [3,4] (named LB2) and is crucial to the high performance of their implementation.

Fig. 3. Modification to function dec_max_2_sat.

```
function dec_max_2_sat( i: variable, g: integer ) return Boolean
    if i > n then return true; // end of the search tree
    if (∑_{j=i}^{n} min(u(j̄), u(j)) > g) then return false end if
    // decide if we want to set variable i to true
    if (u(ī) ≤ g ∧ u(ī) < u(i) + b1(i)) then
        record_unit_clauses(i, 0);
        if (dec_max_2_sat(i + 1, g − u(ī))) then return true; end if
        undo_record_unit_clauses(i, 0);
    end if
    // decide if we want to set variable i to false
    if (u(i) ≤ g ∧ u(i) ≤ u(ī) + b0(i)) then
        record_unit_clauses(i, 1);
        if (dec_max_2_sat(i + 1, g − u(i))) then return true; end if
        undo_record_unit_clauses(i, 1);
    end if
    return false;
end function
```

Combining the ideas in the above discussion, we present in Fig. 3 a modified dec_max_2_sat(i, k, A). The same ideas apply to bb_max_sat(i, k, A) in Fig. 1 as well.

We have implemented the algorithm presented in Fig. 3 in C++ and preliminary experimental results seem promising. Table 1 shows some results of Borchers and Furman's program (BF) [7], Alsinet et al.'s (AMP, the option LB2-I+JW is used), and our implementation (New) on the problems of 50 variables distributed by Borchers and Furman. Note that min_false_clauses in our algorithm is at first set to the number found by the first phase of Borchers and Furman's local search procedure and then decreased by one until the optimal value is decided.

Problems p100-p500 have 50 variables and problems p2200-p2400 have 100 variables. Times (in seconds) are collected on a Pentium 4 linux machine with 256Mb memory. "–" indicates an incomplete run after running for two hours. It is clear that our algorithm runs consistently faster than both Borchers and Furman's program and Alsinet et al.'s modification. We have also generated several thousands of random instances of MAX-2-SAT and the results remains the same. Some of the experimental results are depicted in Fig. 4.

Note that our algorithm takes a fixed order, i.e., from 1 to n, to assign truth value to variables. We found that it is helpful to sort the variables first according their occurrences in the input in non-increasing order and then assign variables in that order. Fig 5 shows the impact of sorting n variables for $n = 30, 40$ variables and $m = 80, ..., 360$ clauses.

Another pre-processing technique we found useful is to delete pure literals from the input when $c = m/n \leq 3$, where n is the number of variables and m is the number of clauses. Fig. 5 shows the impact of using pure literal deletion (PLD) at

Table 1
Experimental results on Borchers and Furman's examples.

Problem	false clauses	BF	AMP	New
p100	4	0.035	0.03	0.02
p150	8	0.091	0.04	0.02
p200	16	6.425	0.51	0.12
p250	22	37	0.36	0.05
p300	32	530	6.53	0.85
p350	41	3866	14	1.45
p400	45	3467	6.05	0.54
p450	63	–	86	4.68
p500	66	–	25	1.78
p2200	5	0.191	0.18	0.53
p2300	15	763	41	7.67
p2400	29	–	1404	172

Fig. 4. Running time for BF [7], AMP [4], and our new algorithm (New). We considered the following cases: $n = 50$ variables and $m = cn$ clauses, where $c = 1.6, 1.7, ..., 5.9, 6$. For each case, we generated 100 random problems. The total run time of 100 instances is depicted (excluding the time reading the input from the file).

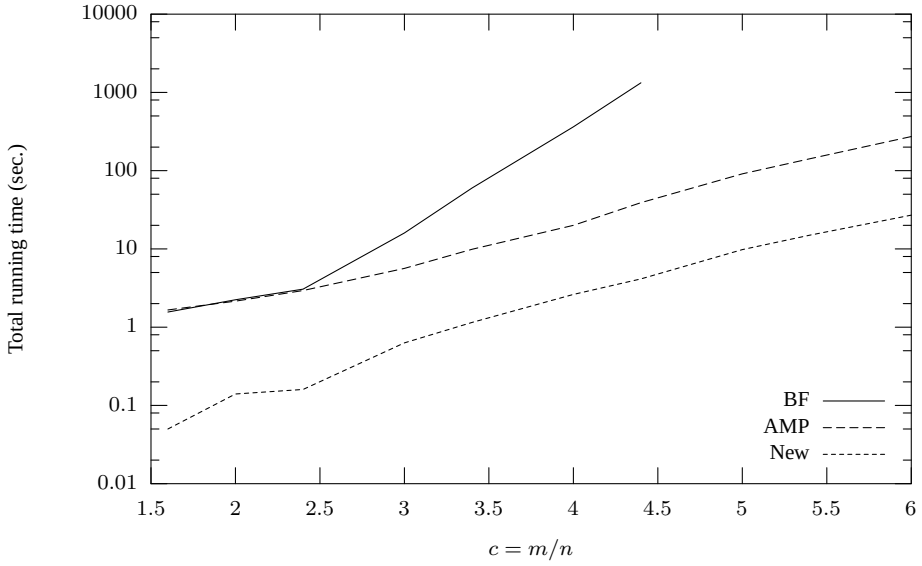
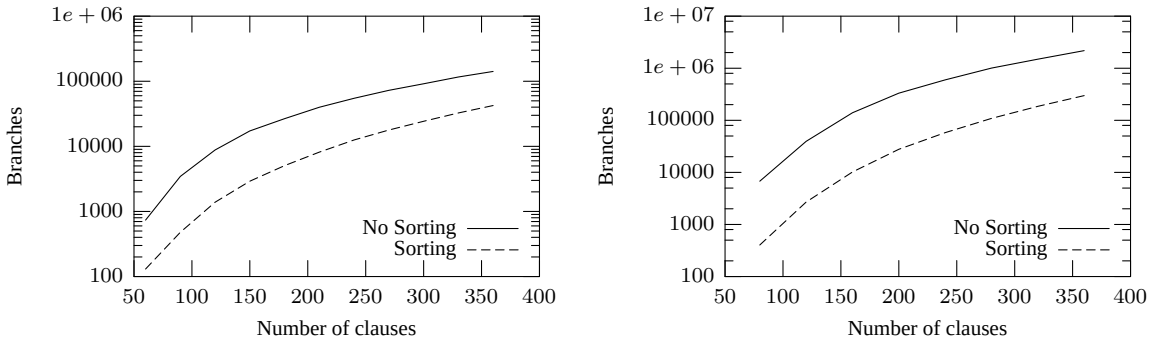
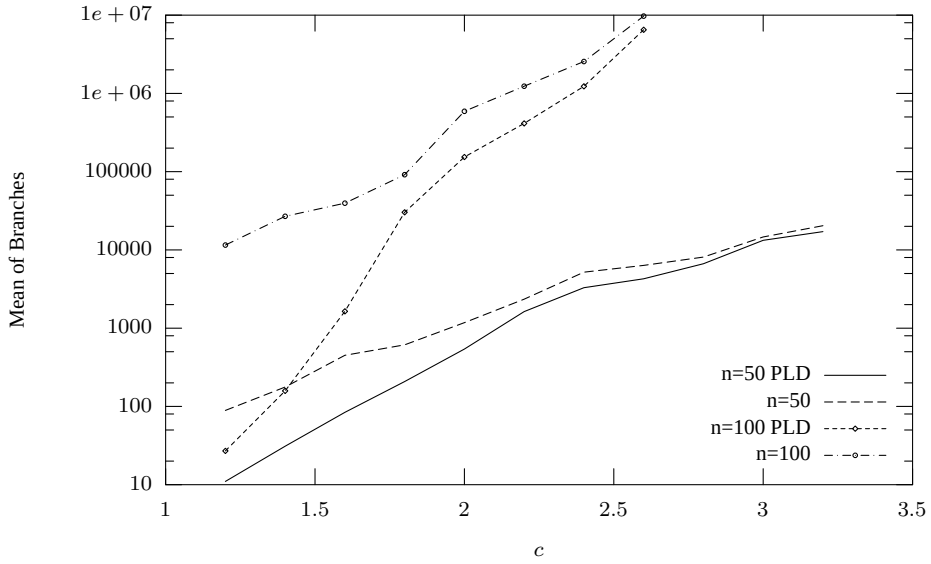


Fig. 5. Run time comparison: no sorting vs. sorting at pre-processing.



(a) $n = 30$ **(b)** $n = 40$

Fig. 6. Computing cost for our decision algorithm with and without pure literal deletion at pre-processing. We generated 100 random problems for each case. The computing cost is the mean of branches of the search tree.



pre-processing when $n = 50, 100$. After removing pure literals from the input, the remaining variables are sorted according to their occurrences. This ordering of the variables is then used in the algorithm in Fig. 3. From the figure, we see clearly that the smaller the value of c, the more the size of the search tree (number of branches) is reduced. The running time, which is proportional to the size of the search tree, is also reduced. This figure also shows the importance of literal selection heuristics. In [3,4], it shown that MOM and JW work well in some cases of MAX-SAT. We plan to further investigate some effective, easy to compute heuristics.

6 Conclusion

We have analyzed a branch-and-bound algorithm for MAX-SAT and showed that the complexity of this algorithm is substantially better than the known results. We also presented an efficient decision algorithm for MAX-2-SAT and showed that it is fast both in theory and in practice. The high performance of our algorithm for MAX-2-SAT may be due to the fact that we have special data structure for binary clauses. As future work, we will implement the algorithm in Fig. 1 for general MAX-SAT (and weighted MAX-SAT [7]) and compare our implementation with [7,3]. We will also use them to study properties of MAX-SAT.

References

[1] J. Alber, J. Gramm, R. Niedermeier, Faster exact algorithms for hard problems: A parameterized point of view. Preprint, submitted to Elsevier, August, 2000

[2] F.A. Aloul, A. Ramani, I.L. Markov, K.A. Sakallah, Generic ILP versus specialized 0-1 ILP: An update. Technical Report CSE-TR-461-02, University of Michigan, Ann Arbor, Michigan, Aug., 2002.

[3] T. Alsinet, F. Manyà, J. Planes, Improved branch and bound algorithms for Max-SAT. Proc. of 6th International Conference on the Theory and Applications of Satisfiability Testing, SAT2003, pages 408-415.

[4] T. Alsinet, F. Manyà, J. Planes, Improved branch and bound algorithms for Max-2-SAT and Weighted Max-2-SAT. Submitted.

[5] S. Arora, C. Lund. Hardness of approximation. In D. Hochbaum (ed.): Approximation algorithms for NP-hard problems, Chapter 10, pages 399-446. PWS Publishing Company, Boston, 1997.

[6] N. Bansal, V. Raman, Upper bounds for MaxSat: Further improved. In Aggarwal and Rangan (eds.): *Proceedings of 10th Annual conference on Algorithms and Computation*, ISSAC'99, volume 1741 of Lecture Notes in Computer Science, pages 247-258, Springer-Verlag, 1999.

[7] B. Borchers, J. Furman, A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization*, 2(4):299-306, 1999.

[8] J. Cheriyan, W.H. Cunningnham, L. Tuncel, Y. Wang. A linear programming and rounding approach to Max 2-Sat. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 26:395–414, 1996.

[9] E. Dantsin, A. Goerdt, E.A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, U. Schöning. A deterministic $(2-2/(k+1))^n$ algorithm for k–SAT based on local search. *Theoretical Computer Science*, 2002.

[10] M. Davis, H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery 7,* 3 (July 1960), 201–215.

[11] M. Davis, G. Logemann, D. Loveland, A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 7 (July 1962), 394–397.

[12] J.W. Freeman, Improvements to propositional satisfiability search algorithms. Ph.D. Dissertation, Dept. of Computer Science, University of Pennsylvania, 1995.

[13] J. Gramm, E.A. Hirsch, R. Niedermeier, P. Rossmanith: New worst-case upper bounds for MAX-2-SAT with application to MAX-CUT. Preprint, submitted to Elsevier, May, 2001

[14] P. Hansen, B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279-303, 1990.

[15] E.A. Hirsch. A new algorithm for MAX-2-SAT. In *Proceedings of 17th International Symposium on Theoretical Aspects of Computer Science*, STACS 2000, vol. 1770, Lecture Notes in Computer Science, pages 65-73. Springer-Verlag.

[16] E.A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397-420, 2000.

[17] R.G. Jeroslow, J. Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1:167-187, 1990.

[18] M. Mahajan, V. Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms*, 31:335–354, 1999.

[19] R. Niedermeier, Some prospects for efficient fixed parameter algorithms. In *Proc. of the 25th Conference on Current Trends in Theory and Practice of Informatics* (SOFSEM'98), Springer, LNCS 1521, pages 168–185, November, 1998.

[20] R. Niedermeier, P. Rossmanith. New upper bounds for maximum satisfiability. *Journal of Algorithms*, 36:63-88, 2000.

[21] I. Schiermeyer. Pure literal look ahead: An $O(1,497^n)$ 3-satisfiability algorithm. In Franco et al (eds.) Proceedings of *Workshop on the Satisfiability Problem*, Report No. 96-230, Universität zu Köln, pages 127-136, Siena, April 1996.

[22] R. Wallace, E. Freuder. Comparative studies of constraint satisfaction and Davis-Putnam algorithms for maximum satisfiability problems. In D. Johnson and M. Trick (eds.) *Cliques, Coloring and Satisfiability*, volume 26, pages 587-615, 1996.

[23] H. Xu, R.A. Rutenbar, K. Sakallah, sub-SAT: A formulation for related boolean satisfiability with applications in routing. ISPD'02, April, 2002, San Diego, CA.

Canonicity [1]

Nachum Dershowitz [2]

School of Computer Science, Tel Aviv University, Tel-Aviv 69978, Israel

Abstract

We explore how different proof orderings induce different notions of saturation. We relate completion, paramodulation, saturation, redundancy elimination, and rewrite system reduction to proof orderings.

They are not capable to ground a canonicity of universal consistency.

—Alexandra Deligiorgi (ΠΑΙΔΕΙΑ, 1998)

1 Introduction

We show how to define the *canonical* basis of an abstract deductive system in three distinct ways: (1) Formulæ appearing in minimal proofs; (2) non-redundant lemmata; (3) minimal trivial theorems. Well-founded orderings of proofs [1] are used to distinguish between cheap "direct" proofs, those that are of a computational flavor (e.g. rewrite proofs), and expensive "indirect" proofs, those that require search to find. This approach suggests generalizations of the concepts of "redundancy" and "saturation", as elaborated by Nieuwenhuis and Rubio in [6]. Saturated, for us, means that all *cheap* proofs are supported. By considering different orderings on proofs, one gets different kinds of saturated sets.

This work continues our development of an abstract theory of "canonical inference", initiated in [5]. Although we will use ground equations as an illustrative example, the framework applies equally well in the first-order setting, whether equational or clausal. Though our motivation is primarily æsthetic; our expectation is that practical applications will follow.

[1] This research was supported in part by the Israel Science Foundation (grant no. 254/01).

[2] Email: Nachumd@tau.ac.il

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

2 Proof Systems

Let $\mathbb{A}$ be the set of all formulæ (ground equations and disequations, in our examples) over some fixed vocabulary. Let $\mathbb{P}$ be the set of all (ground equational) proofs. These sets are linked by two functions: $\Gamma : \mathbb{P} \to 2^{\mathbb{A}}$ gives the assumptions in a proof, and $\Delta : \mathbb{P} \to \mathbb{A}$ gives its conclusion. Both are extended to sets of proofs in the usual fashion. (We assume for simplicity that proofs use only a finite number of assumptions.)

The framework proposed here is predicated on two *well-founded* partial orderings over $\mathbb{P}$: a *proof ordering* $\geq$ and a *subproof relation* $\trianglerighteq$. They are related by a monotonicity requirement given below (20). We will assume for convenience that the proof ordering only compares proofs with the same conclusion ($p \geq q \Rightarrow \Delta\, p = \Delta\, q$), rather than mention this condition each time we will have cause to compare proofs.

We will use the term *presentation* to mean a set of formulæ, and *justification* to mean a set of proofs. We reserve the term *theory* for deductively closed presentations. Let A^* denote the *theory* of presentation A, that is, the set of conclusions of all proofs with assumptions A:

$$A^* \quad := \quad \Delta\ \Gamma^{-1} A \ = \ \{\Delta\, p \colon p \in \mathbb{P},\ \Gamma\, p = A\} \tag{14}$$

We assume the following three standard properties of Tarskian consequence relations:

$$A^* \ \subseteq \ (A \cup B)^* \tag{15}$$

$$A \ \subseteq \ A^* \tag{16}$$

$$A^{**} \ = \ A^* \tag{17}$$

Thus, -* is a closure operation. We say that presentation A is a *basis* for theory C if $A^* = C$. Presentations A and B are *equivalent* if their theories are identical: $A^* = B^*$.

As a very simple running example, let the vocabulary consist of the constant 0 and unary symbol s. Abbreviate tally terms $s^i 0$ as numeral i. The set $\mathbb{A}$ consists of all *unordered* equations $i = j$ (so symmetry is built into the structure of proofs). We postpone dealing with disequations for the time being. An equational inference system for this vocabulary might consist of the following inference rules:

$$\frac{\Box}{0 = 0}\ \mathbf{Z} \qquad \frac{\boxed{i = j}}{i = j}\ \mathbf{I}_{i=j}$$

$$\frac{i = j \ \ j = k}{i = k}\ \mathbf{T} \qquad \frac{i = j}{si = sj}\ \mathbf{S}$$

where **Z** is an (assumptionless) axiom, **I** introduces assumptions, **S** infers that $i + 1 = j + 1$ from a proof of $i = j$, and proof tree branches of the transitivity rule **T** are *unordered*. To accommodate (15), and ignore unneeded assumptions, we also

need projection:

$$\frac{a \;\; c}{c}\;\mathbf{P}$$

For example, if $A = \{4 = 2, 4 = 0\}$, then $A^* = \{i = j : i \equiv j (\mathrm{mod}\ 2)\}$.

Consider the proof schemata:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Box}{0=0}}{1=1}}{\vdots}}{i=i} \qquad \dfrac{\dfrac{\dfrac{\boxed{4=0} \quad \boxed{4=2}}{4=0} \quad \dfrac{}{4=2}}{2=0}}{\vdots}}{i+2=i} \qquad \dfrac{\dfrac{\dfrac{\boxed{4=0} \quad \boxed{4=2}}{4=0} \quad \dfrac{}{4=2}}{2=0} \qquad \dfrac{\dfrac{\dfrac{\vdots}{i-j-2=0}}{i-j-1=1}}{i-j=2}}{i-j=0}}{\vdots}}{i=j}$$

Let's use proof terms for proofs, denoting the above three trees by $S^i Z$, $S^i T(I(4,0), I(4,2))$, and $S^j T(T(I(4,0), I(4,2)), SS(\nabla_{i-j-2=0}))$, respectively. With a recursive path ordering [4] to order proofs, precedence $Z < S < T < I < P < 0 < 1 < 2 < \cdots$, and multiset "status" for I, minimal proofs of the theorems in A^* must take one of these two forms, or the form of one of their subproofs.

We call a proof *trivial* when it proves only itself and has no subproofs other than itself, that is, if $\Gamma\, p = \{\Delta\, p\}$ and $p \trianglerighteq q \;\Rightarrow\; p = q$. We denote by $\widehat{a}$ such a trivial proof of $a \in \mathbb{A}$ and by $\widehat{A}$ the set of trivial proofs of each $a \in A$. For example, $I(4,0) = \widehat{4{=}0}$.

We assume that proofs use their assumptions, that subproofs don't use non-existent assumptions, and that proof orderings are monotonic with respect to subproofs. Specifically, for all proofs p, q, r and formulæ a:

$$a \in \Gamma\, p \;\;\Rightarrow\;\; p \trianglerighteq \widehat{a} \tag{18}$$

$$p \trianglerighteq q \;\;\Rightarrow\;\; \Gamma\, p \supseteq \Gamma\, q \tag{19}$$

$$p \rhd q > r \;\;\Rightarrow\;\; \exists v \in \mathbb{P}.\; p > v \rhd r \tag{20}$$

We make no other assumptions regarding proofs or their structure.

Postulate (20) is the most significant for the development that follows. It states that $>$ (restricted to proofs with the same conclusion) and $\rhd$ commute (i.e. $\rhd \circ > \subseteq > \circ \rhd$). In other words, "replacing" a subproof q of a proof p with a smaller proof r "results" in a proof v that is smaller than the original p. All proof orderings in the literature obey this monotonicity requirement. On account of (18), this also means that proofs are monotonic with respect to any inessential assumptions they refer to, should the latter admit smaller proofs.

Every formula a admits a trivial proof $\widehat{a}$ by (16,18). Let $\Sigma p = \{q : p \trianglerighteq q\}$ denote the subproofs of p, and likewise $\Sigma P = \cup_{p \in P} \Sigma p$. This way, (18) can be abbreviated

$\widehat{\Gamma\, p} \subseteq \Sigma p$.

It may be convenient to think of a proof-tree "leaf" as a subproof with only itself as a subproof; other subproofs are the "internal nodes". There are two kinds of leaves: trivial proofs $\widehat{a}$ (such as inferences **I**), and vacuous proofs $\bar{a}$ with $\Gamma\, \bar{a} = \emptyset$ and $\Delta\, \bar{a} = a$ (such as **Z**). By well-foundedness of $\trianglerighteq$, there are no infinite "paths" in proof trees. It follows from (20) that the transitive closure of $> \cup \triangleright$ is also well-founded.

3 Canonical Systems

Denote the set of all proofs using assumptions of A by:

$$\Pi A \quad := \quad \{p \in \mathbb{P} \colon\ \Gamma\, p \subseteq A\}$$

and define the *minimal* proofs in a set of proofs as:

$$\mu P \quad := \quad \{p \in P : \neg\exists q \in P.\ q < p\}$$

On account of well-foundedness, minimal proofs always exist.

Note that Γ, Δ, *, and Π are all monotonic with respect to set inclusion, but $\mu\Pi$ is not.

Proposition 3.1 *For all justifications P:*

$$P \ \subseteq\ \Pi\, \Gamma\, P \tag{21}$$

And for all presentations A, B:

$$\Gamma\, \Pi A \ = \ A \tag{22}$$
$$\Sigma \mu \Pi A \ = \ \mu \Pi A \tag{23}$$
$$\Pi A = \Pi B \ \Leftrightarrow \ A = B \tag{24}$$

Proof. (21) follows from the definitions, as does one direction of (22). $A \subseteq \Gamma\, \Pi A$ is a consequence of reflexivity (16). (23) is a consequence of (20). The interesting direction of (24) follows immediately from (22) and monotonicity of Γ. □

We say that presentation A is *reduced* when $A = \Gamma\, \mu\Pi A$. Our main definition is:

Definition 3.2 [Canonical Presentation] The *canonical presentation* contains those formulæ that appear as assumptions of minimal proofs, allowing as assumptions any lemma of the theory:

$$A^\sharp \quad := \quad \Gamma\, \mu\Pi A^*$$

So, we say that A is *canonical* if $A = A^\sharp$.

Proof orderings are lifted to sets of proofs, as follows:

Definition 3.3 Justification Q is *better* than justification P if:

$$P \sqsupseteq Q \quad :\equiv \quad \forall p \in P.\, \exists q \in Q.\, p \geq q$$

It is *much better* if:

$$P \sqsupset Q \quad :\equiv \quad \forall p \in P. \exists q \in Q.\, p > q$$

Justifications are *similar* if:

$$P \simeq Q \quad :\equiv \quad P \sqsupseteq Q \sqsupseteq P$$

Transitivity of these three relations follows from the definitions. They are compatible: $\sqsupseteq \circ \sqsupset \subseteq \sqsupset$, $\sqsupseteq \circ \simeq \subseteq \sqsupseteq$, etc. Since it is also reflexive, $\sqsupseteq$ is a quasi-ordering.

Proposition 3.4 *For all justifications P, Q:*

$$P \sqsupseteq \mu P \tag{25}$$
$$P \sqsupseteq Q \Leftrightarrow \mu P \sqsupseteq \mu Q \tag{26}$$
$$P \sqsupset Q \Leftrightarrow \mu P \sqsupset \mu Q \tag{27}$$
$$P \simeq Q \Leftrightarrow \mu P = \mu Q \tag{28}$$

Proof. Well-foundedness ensures that minimal proofs exist (25). Suppose $P \sqsupseteq Q$. Trivially, $\mu P \sqsupseteq P$; by (25), $Q \sqsupseteq \mu Q$; so $\mu P \sqsupseteq \mu Q$. For the other direction of (26): $P \sqsupseteq \mu P \sqsupseteq \mu Q \sqsupseteq Q$. (27) is similar. For (28), Suppose $p \in \mu P \simeq \mu Q$. There must be $q \in \mu Q$ and $p' \in \mu P$ such that $p \geq q \geq p'$. Since p is minimal, $p = p' = q \in \mu Q$. By symmetry, $\mu P = \mu Q$. □

Proposition 3.5 *The relation $\sqsupseteq$ is a* partial *ordering of* minimal *proofs.*

Proof. By (28). □

This "better than" quasi-ordering on proofs is lifted to a "simpler than" quasi-ordering on (equivalent) sets of formulæ, as follows:

Definition 3.6 Presentation B is said to be *simpler* than an equivalent presentation A when B provides better proofs than does A:

$$A \gtrsim B \quad :\equiv \quad A^* = B^* \wedge \Pi A \sqsupseteq \Pi B$$

Presentations are *similar* if their proofs are:

$$A \approx B \quad :\equiv \quad \Pi A \simeq \Pi B$$

These relations are also compatible.

Proposition 3.7 *For all presentations A, B:*

$$\Pi A \sqsupseteq \Pi(A \cup B) \tag{29}$$
$$\mu\Pi A = \mu\Pi B \Leftrightarrow A \approx B \tag{30}$$
$$A \subseteq B \wedge A^* = B^* \Rightarrow A \gtrsim B \tag{31}$$
$$A \subseteq B \wedge \Pi B \sqsupseteq \Pi A \Rightarrow A \approx B \tag{32}$$

Proof. (29) is a consequence of the monotonicity of Π; (30) is a direct consequence of (28); and (31) is a consequence of (29) and the definitions. If $A \subseteq B$ and $\Pi A \sqsupseteq \Pi B$, as on the left of (32), then $\Pi A \simeq \Pi B$, again by (29). Hence, their theories are the same, and, by definition, $A \approx B$. □

Proposition 3.8 *The relation $\succsim$ is a quasi-ordering and $\approx$ is its associated equivalence relation.*

The function $\_^\sharp$ is "canonical" with respect to equivalence of presentations. That is: $A^{\sharp *} = A^*$; $A^* = B^* \Leftrightarrow A^\sharp = B^\sharp$; and $A^{\sharp\sharp} = A^\sharp$. This justifies the terminology of Definition 3.2.

We conclude this section by showing that the canonical presentation is indeed the simplest:

Lemma 3.9 $A \succsim A^\sharp$.

Proof. By (16) and (29), we have $A \succsim A^*$. It can be shown [5, Lemma 2] that $\mu\Pi A^\sharp = \mu\Pi A^*$, so $\Pi A^* \sqsupseteq \mu\Pi A^* = \mu\Pi A^\sharp \sqsupseteq \Pi A^\sharp$. In other words, $A^* \succsim A^\sharp$. □

4 Saturated Systems

By a "normal-form proof", we will mean a proof in $\mu\Pi A^*$, the minimal proofs allowing any theorem as a lemma. Recall (20) that all subproofs of normal-form proofs are also in normal form. We propose the following definitions:

Definition 4.1 [Saturation] A presentation A is *saturated* if it supports all possible normal form proofs: $\Pi A \supseteq \mu\Pi A^*$. A presentation A is *complete* if every theorem has a normal form proof: $A^* = \Delta\,(\Pi A \cap \mu\Pi A^*)$.

In fact, a presentation is saturated iff $\mu\Pi A = \mu\Pi A^*$.

A presentation is complete if it is saturated, but for the converse, we need a further hypothesis: *minimal proofs are unique* if for all theorems $c \in \Pi A$ there is exactly one minimal proof in $\mu\Pi A^*$ with conclusion c.

Proposition 4.2 *If minimal proofs are unique, then a presentation is saturated iff it is complete.*

For example, suppose all rewrite (valley) proofs are minimal but incomparable. Then any Church-Rosser system is complete, since every identity has a rewrite prof, but only the full deductive closure is saturated.

Theorem 4.3 ([5]) *A presentation A is saturated iff it contains its own canonical presentation: $A \supseteq A^\sharp$. In particular, $A^\sharp$ is saturated. Moreover, the canonical presentation $A^\sharp$ is the smallest saturated set: No equivalent proper subset of $A^\sharp$ is saturated; if A is saturated, then every equivalent superset also is.*

Corollary 4.4 *Presentation A is saturated iff $A^* \approx A$.*

Proof. It is always the case that $A \succsim A^* \succsim A^\sharp$. If A is saturated, then $A \supseteq A^\sharp$ and, therefore, $A^* \succsim A^\sharp \succsim A$. For the other direction, suppose $p \in \mu\Pi A^*$. Since A is similar, there must be a proof $q \in \Pi A \subseteq \Pi A^*$, such that $q \leq p$. But $q \not< p$, so $p \in \Pi A$. It follows that $\mu\Pi A^* \subseteq \Pi A$, and A is saturated. □

Lemma 4.5 *Similar presentations are either both saturated or neither is; similar presentations are either both complete or neither is.*

Proof. The first claim follows directly from the previous result. For the second, one can verify that $A \approx B$ implies:

$$B^* \;=\; A^* \;=\; \Delta\,(\Pi A \cap \mu\Pi A^*) \;=\; \Delta\,(\mu\Pi A \cap \mu\Pi A^*)$$
$$=\; \Delta\,(\mu\Pi B \cap \mu\Pi B^*) \;=\; \Delta\,(\Pi B \cap \mu\Pi B^*)$$

□

Formulæ that can be removed from a presentation—without making proofs worse—are "redundant":

Definition 4.6 [Redundancy] A set R of formulæ is *(globally) redundant* with respect to a presentation A when: $A \cup R \succsim A \setminus R$. The set of all *(locally) redundant* formulæ of a given presentation A will be denoted ρA:

$$\rho A \quad := \quad \{r \in A \colon A \succsim A \setminus \{r\}\}$$

A presentation A is *irredundant* if $\rho A = \emptyset$.

Proposition 4.7 ([5]) *The following facts hold for all presentations A:*

$$\rho A^\sharp \;=\; \emptyset \tag{33}$$
$$A \;\approx\; A \setminus \rho A \tag{34}$$
$$A^\sharp \;=\; A^* \setminus \rho A^* \tag{35}$$
$$A^\sharp \;=\; \Delta\,(\mu\Pi A^* \cap \widehat{A^*}) \tag{36}$$
$$\widehat{A^\sharp} \;=\; \mu\Pi A^* \cap \widehat{A^*} \tag{37}$$

It is thanks to well-foundedness of $>$ that the set of all *locally* redundant formulæ in ρA is *globally* redundant (Eq. 34). Thus, it can be shown that A is reduced iff it is irredundant. The alternate definition of the canonical set (36) is made possible by the properties of subproofs. For details, see [5].

Theorem 4.8 *A presentation is canonical iff it is saturated and reduced.*

Proof. One direction follows immediately from Theorem 4.3 and (33). For the other direction, let A be saturated and reduced. We aim to show that $A = A^\sharp$. By Lemma 3.9, $A \succsim A^\sharp$ and the two presentations are equivalent. If A is saturated, then by Theorem 4.3, $A \supseteq A^\sharp$. By (31), for any $r \in A \setminus A^\sharp$, $A \succsim A^\sharp \succsim A \setminus \{r\}$. But $\rho A = \emptyset$, since A is reduced, so it cannot be that $r \in A$. In other words, $A \setminus A^\sharp = \emptyset$, and A is canonical. □

Returning to our simple example, we can add three inference rules for disequalities:

$$\frac{i = j \quad j \neq k}{i \neq k}\ \mathbf{T} \qquad \frac{i \neq i}{j = k}\ \mathbf{F}_{j=k} \qquad \frac{\boxed{i \neq j}}{i \neq j}\ \mathbf{I}_{i \neq j}$$

With them, one can infer, for example, $0 \neq 0$ from $1 \neq 1$. If F is smaller than other proof combinators, and I nodes are incomparable, then the canonical basis of any inconsistent set is $\{i \neq j \colon i, j \in \mathbf{N}\}$. All positive equations are redundant.

5 Variations

Consider the above inference rules for ground equality and disequality: S, T, F, I, Z, with S extended to apply to all function symbols of any arity. Suppose we are using something like the recursive path ordering for proof terms.

Refutation.

If the inference rule F is the cheapest in the proof ordering, $T < I$, and $I(i, j)$ nodes are measured by the values of i and j, then the canonical basis of any inconsistent presentation is a (smallest) trivial disequation $\{t \neq t\}$.

Deduction.

If the proof ordering prefers direct application I of axioms over all other inferences (including Z), then trivial proofs are best. In that case, $\rho A^* = \emptyset$ and the canonical basis includes the whole theory $A^\sharp = A^*$.

Paramodulation.

If the proof ordering makes functional reflexivity S smaller than I, but the only ordering on leaves is $I(u, t) \leq I(c[u], c[t])$ for any context c, then the canonical basis will be the congruence closure, as generated by paramodulation: $\rho A = \{f(u_1, \ldots, u_n) = f(t_1, \ldots, t_n) \colon u_1 = t_1, \ldots, u_n = t_n \in A^*\}$. The theory A^* is the closure under functional reflexivity of the basis $A^\sharp$. If A is as in our first example, then $A^\sharp = \{2j = 0 \colon j > 0\}$.

Completion.

On the other hand, if the ordering on leaves compares terms in some simplification ordering $\gtrsim$, then the canonical basis will be the fully reduced set, as generated by (ground) completion: $\rho A = \{u = u\} \cup \{u = t \colon t = v \in A^*,\ t \gg v,\ v \text{ is not } u\}$. For our first example, $A^\sharp = \{2 = 0\}$. For another example, if $A = \{a = c, sa = b\}$ and $sa \gg sb \gg sc \gg a \gg b \gg c$, then $I(sa, b) > T(S(I(a, c)), I(sc, b))$, and hence $A^\sharp = \{a = c, sc = b\}$.

Superposition.

If one distinguishes between T steps based on the weight of the shared term j, making $T > I$ when j is the smallest, and $T < I$ otherwise, then the canonical basis is also closed under paramodulation into the larger side of equations.

6 Derivations

Theorem proving with simplification (cf. [3, Chap. 2]) entails two processes: **Expansion**, whereby any sound deductions (anything in E^*) may be added to the set of derived theorems; and **Contraction**, whereby any redundancies (anything in ρE) may be removed.

A sequence of presentations $E_0 \rightsquigarrow E_1 \rightsquigarrow \cdots$ is called a *derivation*. Let $E_* = \cup_i E_i$. The *result* of the derivation is, as usual [1], its *persisting* formulæ:

$$E_\infty \quad := \quad \liminf_{j\to\infty} E_j$$

We will say that a proof p *persists* when $\Gamma\, p \subseteq E_\infty$. Thus, if a proof persists, so do its subproofs (by 19). By (29), we have $\Pi E_i \sqsupseteq \Pi E_*$.

Definition 6.1 A derivation $E_0 \rightsquigarrow E_1 \rightsquigarrow \cdots$ is *good* if $E_i \succsim E_{i+1}$ for all i.

We are only interested in good derivations. From here on in, only good derivations will be considered. It is easy to see that:

Lemma 6.2 *Derivations, the steps of which are expansions and contractions, are good.*

Proposition 6.3 *If a derivation is good, then the limit supports the best proofs:* $E_* \approx E_\infty$.

Proof. One direction, namely $\Pi E_\infty \sqsupseteq \Pi E_*$, follows by (29) from the fact that $E_\infty \subseteq E_*$. To establish that $\Pi E_* \sqsupseteq \Pi E_\infty$, we show that $\mu\Pi E_* \sqsupseteq \Pi E_\infty$ and rely on (25). Suppose $p \in \mu\Pi E_*$. It follows from (18,23) that $\widehat{\Gamma\, p} \subseteq \Sigma p \subseteq \mu\Pi E_* \subseteq \mu\Pi E_*$. By goodness, each $a \in \Gamma\, p$ persists from some E_i on. Hence, $\Gamma\, p \subseteq E_\infty$, and $p \in \Pi E_\infty$. □

Definition 6.4 A good derivation is *fair* if $C(E_\infty) \sqsupset \Pi E_*$ where $C(E)$ is the set of *critical proof obligations*:

$$C(E) \quad := \quad \{p \in \Pi E : p \notin \mu\Pi E^*,\ \forall q \lhd p.\ q \in \mu\Pi E^*\} \tag{38}$$

It is *clean* if $\rho E_* \cap E_\infty = \emptyset$.

Critical obligations are proofs that are not in normal form but all of whose proper subproofs are already in normal form. Fairness means that all persistent obligations are eventually "subsumed" by a strictly smaller proof.

Lemma 6.5 *If a derivation is clean, then its limit is reduced.*

Proof. Suppose, on the contrary, that some $r \in \rho E_\infty \subseteq E_\infty \subseteq E_*$. Consider $\widehat{r}$, and compare it to a smaller proof $p \in \Pi E_\infty$. Let $A = \Gamma\, p \subseteq E_\infty \subseteq E_*$. Let $q \in \mu\Pi E_*$. Were $r \in \Gamma\, q$, then replacing $\widehat{r}$ as a subproof of q with p, would by (20) result in a smaller proof than q. It follows that $r \in \rho E_*$, which contradicts cleanliness. □

Lemma 6.6 *If a derivation is fair, then its limit is complete.*

Proof. Any presentation A is complete if $\Pi A \sqsupseteq \Pi A \cap \mu\Pi A^*$. since $a \in A^*$ implies $a \in \Delta\,(\Pi A \cap \mu\Pi A^*)$, whence completeness. Let $A = E_*$ be all formulæ proved at any stage in the derivation. We show that A is complete in the above manner. Completeness of E_∞ follows from Lemma 4.5. Consider any proof in $p \in \Pi A$ of a. Let $p_\infty \in \Pi E_\infty \subseteq \Pi A$ be the persisting proof of a, for which $p_\infty \leq p$ by the

previous proposition. If $p_\infty \in \mu\Pi A^*$, we're done. Otherwise, p_∞ has a minimal (with respect to $\trianglelefteq$) non-normal-form (possibly trivial) subproof q, all subproofs of which (persist and) are in normal form. By fairness, there is a proof $r \in \Pi A$ of the same theorem as q such that $p_\infty \trianglerighteq q > r$. By (20), there is therefore a better proof $p' < p_\infty \leq p$. By induction, there is a $p'' \leq p'$ in both ΠA and $\mu\Pi A^*$, also proving a. □

For example, suppose a proof ordering makes $\widehat{c} > \frac{\hat{a}}{c}$ and $\frac{\hat{c}}{a} > \widehat{a}$. Start with $E_0 = \{c\}$, and consider $\widehat{c}$. Were $\widehat{c}$ to persist, then by fairness a better proof would evolve, the better proof being $\frac{\hat{a}}{c}$. If $\widehat{a}$ is in normal form, then $a \in E_\infty$ and both minimal proofs persist. Another example: $\mu\mathbb{P} = \{\widehat{a}, \widehat{c}, \frac{\hat{a}}{c}\}$ and $E = \{a\}$, then $E \rightsquigarrow E \rightsquigarrow \cdots$ is fair, since $E_\infty = E$ and $C(E_\infty) = \emptyset$. The result is complete but unsaturated (c is missing).

Together, these lemmata and Proposition 4.2 yield:

Theorem 6.7 *If minimal proofs are unique and a derivation is fair and clean, then its limit is canonical.*

By (36), this also means that each $e \in E_\infty$ is its own ultimate proof $\widehat{e}$, so is not susceptible to contraction.

Returning to our main example, if projection P is the most expensive type of inference, then no minimal proof includes it. And if proofs are compared in a simplification ordering (subproofs are always smaller than their superproofs), then minimal proofs will never have superfluous transitivity inferences of the form

$$\frac{u = t \;\; t = t}{u = t}\ \mathbf{T}$$

Let $\gtrdot$ be a total simplification-ordering of terms, let $P > I > T > S > Z$ in the precedence, let proofs be greater than terms, and compare proof trees in the corresponding total recursive path simplification-ordering. *Ground completion* is an inference mechanism consisting of the following inference rules:

Deduce: $E \cup \{w = t[u]\} \rightsquigarrow E \cup \{w = t[v]\}$ if $u = v \in E$ and $u \gg v$

Delete: $E \cup \{t = t\} \rightsquigarrow E$

Furthermore, operationally, completion implements these inferences "fairly": No persistently enabled inference rule is ignored forever.

Corollary 6.8 (Completeness of Completion) *Ground completion results—at the limit—in the canonical, Church-Rosser basis:* $E_\infty = E_0^\sharp$.

Proof. Ground completion is good, since **Deduce** and **Delete** don't increase proofs ($\rightsquigarrow \subseteq \gtrsim$). In particular, $I(w, t[u]) > T(I(w, t[v]), S^n(I(u, v)))$ if $u \gg v$, since

$t[u] \gg t[v]$ and $t[u] \gtrdot u \gg v$. Ground completion is fair and clean. For example, the critical obligation

$$\frac{w = t \;\; t = v}{w = v}\ \mathbf{T}$$

when $t \gg w, v$, is resolved by **Deduce**. Also, since $T > S$, non-critical cases resolve naturally:

$$\frac{\dfrac{w = t}{fw = ft} \quad \dfrac{t = v}{ft = fv}}{fw = fv} \;>\; \frac{\dfrac{w = t \;\; t = v}{w = v}}{fw = fv}$$

□

7 Discussion

We have suggested here that proof orderings, rather than formula orderings, take center stage in theorem proving with contraction (simplification and deletion of formulæ). Given a proof ordering that distinguishes "good proofs" from "bad proofs", it makes sense to define completeness of a set of formulæ as the claim that all theorems enjoy a smallest ("best") proof. Then an inference system is complete if it has the ability to generate all formulæ needed for such ideal proofs. Given a formula ordering, one can, of course, choose to compare proofs by simply comparing the multiset of their assumptions.

The notion of "saturation" in theorem proving, in which superfluous deductions are not necessary for completeness, was suggested by Rusinowitch [7, pp. 99–100] in the context of a Horn-clause resolution calculus. In our terminology: A presentation was said to be saturated when all inferrible formulæ are syntactically subsumed by formulæ in the presentation. This concept was refined by Bachmair and Ganzinger (see, most recently, [2]) and by Nieuwenhuis and Rubio [6, pp. 29–42]. They define saturation in terms of a more general kind of redundancy: An inference is redundant if its conclusion can be inferred from smaller formulæ; a presentation is saturated if every inference is redundant.

We propose alternate definitions of saturation and redundancy, defining both in terms of the proof ordering. This appears to be more flexible, since it allows small proofs to use large assumptions. The definition of redundancy in [6] coincides with ours when proofs are measured first by their maximal assumption. The one given in [3, Def. 2.4.4]—a sentence is redundant if adding it to the set of assumptions does not decrease any minimal proof—is equivalent.

In [1], a completion sequence is deemed fair if all persistent critical inferences are generated. In [6, fn. 8], an inference sequence is held to be fair if all persistent inferences are either generated or become redundant. The definition of fairness propounded here combines the two ideas. But fairness only earns completeness,

not saturation. (A stronger version of fairness is needed for saturation when the proof ordering is partial.) Our definition of critical obligations also allows one to incorporate "critical pair criteria".

References

[1] Leo Bachmair and Nachum Dershowitz. Equational inference, canonical proofs, and proof orderings. *J. of the Association for Computing Machinery*, 41(2):236–276, 1994.

[2] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science, 2001.

[3] Maria Paola Bonacina. *Distributed automated deduction*. PhD thesis, Department of Computer Science, State University of New York at Stony Brook, December 1992.

[4] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982.

[5] Nachum Dershowitz and Claude Kirchner. Abstract saturation-based inference. In *Proceedings of the 18th Annual Symposium on Logic in Computer Science*, Ottawa, June 2003. IEEE Computer Society Press.

[6] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.

[7] Michaël Rusinowitch. *Démonstration Automatique: Techniques de Réécriture*. Science Informatique. InterEditions, Paris, 1989.

Reachability in Conditional Term Rewriting Systems

Guillaume Feuillade [1] Thomas Genet [2]

IRISA
Université de Rennes 1 & ENS Cachan
Campus de Beaulieu
F-35042 Rennes

Abstract

In this paper, we study the reachability problem for conditional term rewriting systems. Given two ground terms s and t, our practical aim is to prove $s \not\rightarrow^*_{\mathcal{R}} t$ for some join conditional term rewriting system $\mathcal{R}$ (possibly not terminating and not confluent). The proof method we propose relies on an over approximation of reachable terms for unrestricted join conditional term rewriting systems. This approximation is computed using an extension of the tree automata completion algorithm to the conditional case.

Introduction

In [8], we proposed a technique for approximating the set of reachable terms: given a Term Rewriting System (TRS for short) $\mathcal{R}$ and a regular set of terms E recognized by a tree automaton $\mathcal{A}$, we compute another tree automaton $\mathcal{A}'$ recognizing a super set of terms reachable by rewriting terms of E with $\mathcal{R}$, i.e. $\mathcal{R}^*(E)$. Then, given two terms $s, t \in \mathcal{T}(\mathcal{F})$ and $s \in E$, if t is not recognized by $\mathcal{A}'$ then we have a proof that $s \not\rightarrow^*_{\mathcal{R}} t$. This technique is implemented in the Timbuk tool [11] and have some direct applications in verification where $\mathcal{R}$ is used to model a program behavior, E a set of initial configurations and the super set of reachable terms represent an approximation of every possible execution. An interesting aspect w.r.t. verification is that no assumption is made over $\mathcal{R}$, in particular termination and confluence are not needed. This approach proved to be successful for the verification of cryptographic protocols [9] and was recently applied to verify a cryptographic protocol for pay TV developed by Thomson Multimedia [10].Cryptographic protocol verification consists in proving that for a fixed set of possible intruder actions,

[1] Email: {Guillaume.Feuillade}@irisa.fr
[2] Email: {Thomas.Genet}@irisa.fr

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

whatever the combination of those actions maybe, the intruder is not able to obtain any secret information or to break authentication between agents. In our setting, we model the protocol by a term rewriting system $\mathcal{R}$ where for every rule, the left-hand side represents the message expected by an agent and the right-hand side models its answer. This term rewriting system is extended by a set of rules describing the intruder actions: usually listening to any message, encrypting and decrypting with any key he has, building and deconstructing messages, etc. The set E contains the initial messages of the protocol and the set $\mathcal{R}^*(E)$ contains every possible message exchanged during the protocol with an intruder listening, replaying, encrypting, decrypting, etc. With Timbuk we can automatically build an over-approximation of $\mathcal{R}^*(E)$ provided that the user gives some approximation rules by hand. Using user defined approximation rules rather than a fixed automatic approximation methodology permits in particular to adapt the precision of the approximation to the property to be proven. Then, we define a set F of forbidden messages representing flaws in the protocol and prove that the intersection between the over-approximation of $\mathcal{R}^*(E)$ and F is empty, proving that no flaw can occur from an initial message whatever the combination of intruder action may be. This technique allowed us to prove automatically security properties on cryptographic protocols under complex verification assumptions: sessions interleaving, unbounded number of sessions, unbounded number of agents.

Now, our aim is to provide in Timbuk some approximation of reachable terms for an extended specification language: Conditional Term Rewriting Systems (CTRS for short). The first motivation for this extension is that CTRS provide a more 'user-friendly' specification language for programs and protocols. A second motivation is that programming languages based on rewriting like Elan [2] or Maude [3] do integrate conditions. Thus to prove properties on Elan and Maude programs using approximations, it is necessary to take conditions into account. Note that these languages also rely on strategies and that there are already some approaches for dealing with reachability under some strategy for some restricted classes of TRS [13].

A first and natural idea to approximate reachability for conditional term rewriting systems is to encode CTRS into TRS and thus reduce the problem of reachability for a conditional term rewriting system to the problem of reachability for a non conditional one. Suprisingly, this is not the more easy and natural way. Thus, in this paper, we propose an extension of the tree automata completion algorithm of [8] to the conditional case.

In section 1, we define TRS, CTRS and tree automata. In section 2, we shortly recall the tree automata completion algorithm for TRS and the approximation construction. In section 3, we define the extension of this algorithm to the CTRS case and show that it produces a tree automaton recognizing an over approximation of reachable terms for join CTRSs. In section 4, we show that the extended algorithm should give better results than using an encoding of CTRS into TRS and the existing algorithm. Finally, we conclude in section 5.

1 TRS, CTRS and tree automata

In this section we shortly present definitions and tools used in this paper. For details about TRS and CTRS one can refer to [5,6] and to [4] for tree automata.

Let $\mathcal{F}$ be a set of function symbols with an arity in $\mathbb{N}$, $\mathcal{X}$ be a set of variables. $\mathcal{T}(\mathcal{F},\mathcal{X})$ is the set of terms over $\mathcal{F}$ and $\mathcal{X}$, $\mathcal{T}(\mathcal{F})$ is its subset of ground terms. $Var(t)$ designates the set of variables of a term t. The set of *positions* of a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ is a word over $\mathbb{N}$ defined by:

(i) $\mathcal{P}os(t) = \{\epsilon\}$ if $t \in \mathcal{X}$,

(ii) $\mathcal{P}os(f(t_1,\ldots,t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in \mathcal{P}os(t_i)\}$ if $f \in \mathcal{F}$, $arity(f) = n$ and $t_1,\ldots,t_n \in \mathcal{T}(\mathcal{F},\mathcal{X})$, where ϵ is the empty sequence of integers.

If $p \in \mathcal{P}os(t)$ then the subterm of t at position p is denoted by $t|_p$. The term obtained by replacing $t|_p$ in t at position p by the term s is denoted by $t[s]_p$. $\mathcal{P}os(t)$ is a partially ordered set whose order is defined by $p \leq p' \Leftrightarrow \exists q \in \mathcal{P}os(t|_p)$ s.t. $p' = p.q$.

Definition 1.1 A *substitution* is an application $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F},\mathcal{X})$ that one can extend to $\mathcal{T}(\mathcal{F},\mathcal{X})$ in an endomorphism: $\sigma : \mathcal{T}(\mathcal{F},\mathcal{X}) \mapsto \mathcal{T}(\mathcal{F},\mathcal{X})$. The result of the *application* of a substitution σ to a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ is the term denoted by $t\sigma$.

Definition 1.2 A context is a term $C[\,]$ in $\mathcal{T}(\mathcal{F} \cup \{\Box\},\mathcal{X})$ where the new constant symbol $\Box \notin \mathcal{F}$ appears only once. For all context $C[\,]$ and all term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $C[t]$ is the term obtained replacing $\Box$ by t in $C[\,]$.

1.1 TRS and CTRS

A term rewriting system (TRS) over a set of ground terms $\mathcal{T}(\mathcal{F})$ is a set $\mathcal{R}$ of pairs $(t_l,t_r) \in \mathcal{T}(\mathcal{F},\mathcal{X})$ called *rules* (and denoted by $t_l \rightarrow t_r$) such that $Var(t_r) \subseteq Var(t_l)$. In the following (r) will designate a rule in $\mathcal{R}$. A term $t \in \mathcal{T}(\mathcal{F})$ can be rewritten by the rule $(r) : t_l \rightarrow t_r$ at position $p \in \mathcal{P}os(t)$ iff there exists a substitution σ such that $t|_p = t_l\sigma$ and the result of the rewriting is $t[t_r\sigma]_p$. We denote this rewriting by $t \xrightarrow{(r)}_{\mathcal{R}} t[t_r\sigma]_p$. Thus a rewriting system $\mathcal{R}$ defines a *rewriting relation* which is a binary relation $\rightarrow_{\mathcal{R}}$ between terms in $\mathcal{T}(\mathcal{F})$ such that $t \rightarrow_{\mathcal{R}} t'$ iff there exists a rule (r), a position $p \in \mathcal{P}os(t)$ and a substitution σ such that $t \xrightarrow{(r)} t'$. $\rightarrow^*_{\mathcal{R}}$ is the transitive closure of $\rightarrow_{\mathcal{R}}$.

A conditional term rewriting system (CTRS) over a set of ground terms $\mathcal{T}(\mathcal{F})$ is a set $\mathcal{R}$ of conditional rules $(r)t_l \rightarrow t_r$ if $cond$, where $t_l,t_r \in \mathcal{T}(\mathcal{F},\mathcal{X})$ and $cond$ designates a conjunction of conditions that must be checked before rewriting. In this paper, conditions are pairs of terms denoted by $c_1 \downarrow c_2$ where $c_1,c_2 \in \mathcal{T}(\mathcal{F},\mathcal{X}), (Var(c_1) \cup Var(c_2)) \subseteq Var(t_l)$; these are join conditions. Such a condition is said to be true for a substitution σ if there exists a term $u \in \mathcal{T}(\mathcal{F})$

such that $c_1\sigma$ and $c_2\sigma$ can be both rewritten by the CTRS $\mathcal{R}$ into u in a finite number of steps. Then, the rule $t_l \rightarrow t_r$ if $c_1 \downarrow c_2$ can be applied to the term $t \in \mathcal{T}(\mathcal{F})$ at position p as for a TRS. $\rightarrow_{\mathcal{R}}$ also defines a rewriting relation on $\mathcal{T}(\mathcal{F})$.

For a TRS or a CTRS $\mathcal{R}$ and a set of terms $E \subseteq \mathcal{T}(\mathcal{F})$, we define the set of $\mathcal{R}$-descendants of E, denoted by $\mathcal{R}^*(E)$ which is the set of reachable terms from E by $\rightarrow^*_{\mathcal{R}}$, i.e. $\mathcal{R}^*(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid s \in E \text{ and } s \rightarrow^*_{\mathcal{R}} t\}$.

1.2 Tree automata

Tree automata are tools to represent finitely sets of trees and then of terms. A *tree automaton* is a tuple $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, where $\mathcal{Q}$ is a finite set of symbols with arity 0 that are the *states* of the automaton. $\mathcal{Q}_f$ is the subset of $\mathcal{Q}$ of final states and Δ is a set of *normalized transitions*. A normalized transition is a rewriting rule where the left term is $f(q_1, q_2, \ldots, q_n)$ where $f \in \mathcal{F}$ s.t. $arity(f) = n$, $\{q_1, \ldots, q_n\} \subseteq \mathcal{Q}$ and the right term is a state $q \in \mathcal{Q}$. The rewriting relation induced by Δ is denoted by $\rightarrow_\Delta$. The language recognized by $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists q \in \mathcal{Q}_f \text{ s.t. } t \rightarrow^*_\Delta q\}$. The language recognized by a state $q \in Q$ is the set of terms that rewrite into q and is denoted by $\mathcal{L}(\mathcal{A}, q)$. Note that $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in \mathcal{Q}_f} \mathcal{L}(\mathcal{A}, q)$. A set of terms is said to be *regular* if there exists a tree automaton that recognize it.

2 The tree automaton completion algorithm

We first shortly present the algorithm used to approximate $\mathcal{R}^*(E)$ when $\mathcal{R}$ is a TRS [8]. Informally, the algorithm consists in a completion of the tree automaton recognizing E w.r.t the rules of $\mathcal{R}$ to get -step by step- the $\mathcal{R}$-reachable terms from E. Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton over $\mathcal{T}(\mathcal{F})$.

Definition 2.1 A *regular language substitution over* $\mathcal{A}$ is an application $\sigma_{\mathcal{L}} : \mathcal{X} \mapsto \mathcal{Q}$. We can extend this definition to a morphism $\sigma_{\mathcal{L}} : \mathcal{T}(\mathcal{F}, \mathcal{X}) \mapsto \mathcal{T}(\mathcal{F}, \mathcal{Q})$.

For $E = \mathcal{L}(\mathcal{A})$ and for all $q \in \mathcal{Q}$, the completion consists in extending $\mathcal{L}(\mathcal{A}, q)$, the language recognized by q, to the transitive closure of $\mathcal{L}(\mathcal{A}, q)$ by $\mathcal{R}$. To do so, we have to find $s, t \in \mathcal{T}(\mathcal{F})$ and $s \in \mathcal{L}(\mathcal{A}, q)$ such that $(l \rightarrow r) \in \mathcal{R}$, $s = l\sigma \rightarrow_{\mathcal{R}} r\sigma = t$, and $t \notin \mathcal{L}(\mathcal{A}, q)$. Then, we can add t to $\mathcal{L}(\mathcal{A}, q)$ by adding a set of transitions to Δ s.t. $t \rightarrow^*_\Delta q$.

In practice, considering every possible $s \in \mathcal{T}(\mathcal{F})$ such that $s \in \mathcal{L}(\mathcal{A}, q)$ is not possible since it can be infinite. However, it is equivalent to consider every rule $l \rightarrow r$ of $\mathcal{R}$ and every possible regular language substitution $\sigma_{\mathcal{L}}$ (which are in finite number) over $\mathcal{A}$ such that $l\sigma_{\mathcal{L}} \rightarrow^*_\Delta q$ and $r\sigma_{\mathcal{L}} \not\rightarrow^*_\Delta q$. In this case, $l\sigma_{\mathcal{L}} \rightarrow^*_\Delta q$ and $l\sigma_{\mathcal{L}} \rightarrow^*_{\mathcal{R}} r\sigma_{\mathcal{L}}$ is what we call a *critical pair*. Note that it is equivalent to consider every possible $l\sigma_{\mathcal{L}}$ instead of every possible $l\sigma \in \mathcal{T}(\mathcal{F})$ because if $l\sigma \rightarrow^*_\Delta q$ then there exists a particular $\sigma_{\mathcal{L}}$ s.t. $l\sigma \rightarrow^*_\Delta l\sigma_{\mathcal{L}} \rightarrow^*_\Delta q$. When such a critical pair is found, adding a set of normalized transitions to perform the rewriting $r\sigma_{\mathcal{L}} \rightarrow^*_\Delta q$ is equivalent to adding a set of normalized transitions to perform the rewriting $r\sigma \rightarrow^*_\Delta q$

in $\mathcal{A}$ for the same reason. Non left or right linear rules may cause problems due to the filtering process to get the regular language substitution (a same term can be recognized by two different states, then the non-linearity should not apply to states); however there exist a method to go trough this issue described in [11]. When it terminates, this algorithm produces a tree automaton $\mathcal{A}'$ such that for every state q, $\mathcal{L}(\mathcal{A}, q)$ contains $\mathcal{L}(\mathcal{A}, q)$ as well as an *approximation* of its successors by $\mathcal{R}$.However, in many cases the completion produces an infinite number of new states and thus may not terminate. This is due to the fact that transitions added during completion need to be normalized. Normalization of transitions is necessary in order to be able to perform complementation and intersection operations to exploit the result of the completion. Adding transitions to the automaton adds states to the automaton: the left-hand side of $r\sigma \to q$ may be of depth 2 or more, thus it may be necessary to normalize it before adding it to $\mathcal{A}$. In that case, we have to create some new states to recognize subterms of $r\sigma$. Repeatedly adding new states may lead completion to diverge. On the opposite, it is very easy to force termination by limitating the number of new states that can be used and re-use existing states.

Example 2.2 Let us consider the following transition to be normalized in a automaton $\mathcal{A}$ where $q_1 \in \mathcal{Q}$, q_2 is a new state and there is no transition with q_1 in its left hand side:

$$g(f(q_1)) \to q_2$$

The usual way to normalize it is to create a new state, say q_3, and to add the transitions $f(q_1) \to q_3$ and $g(q_3) \to q_2$. An approximation would be, for example, to add these rules instead: $f(q_1) \to q_1$ and $g(q_1) \to q_2$. This would produce, in the new automaton $\mathcal{A}'$ the language $\mathcal{L}(\mathcal{A}', q_2) = \{g(f^n(\mathcal{L}(\mathcal{A}, q_1))) \mid n \in \mathbb{N}\}$ which is a superset of the exact one which is $\{g(f(\mathcal{L}(\mathcal{A}, q_1)))\}$

The approximation we obtain are very close to the widening operations in abstract interpretation. In such techniques, the widening is defined once and it ensures termination in any case. However, if a widening is not precise enough for a given property nothing can be done. Our setting is different, the user is able to give some approximation rules well adapted to its model and to the property he wants to prove. The approximation rules are rewrite rules used to merge together terms or states in order to minimize the number of new states. Standard widening operations on languages can easily be described using a small number of approximation rules [11].

3 Reachability in CTRS by tree automata completion

The purpose of this section is to adapt the existing algorithm of approximation of reachable terms [8] to CTRS. Let $E \subseteq \mathcal{T}(\mathcal{F})$ be a regular entry set of ground terms. We want to get an over approximation of the set $\mathcal{R}^*(E)$ for a given CTRS $\mathcal{R}$. Note that we only focus on left-linear CTRS, since every non left-linear CTRS can be transformed into an equivalent left-linear CTRS: every rule of the form $C[x, x, x] \to \ldots$ can be transformed into $C[x, y, z] \to \ldots$ if $\downarrow (x, y, z)$

where $\downarrow (c_1, \ldots, c_n)$ is an extended joinability condition. Given a substitution σ, $\downarrow (c_1, \ldots, c_n)$ is true if there exists a term u s.t. $c_1\sigma \rightarrow^*_{\mathcal{R}} u$, ..., $c_n\sigma \rightarrow^*_{\mathcal{R}} u$. This technique extends to any number of non linear occurrence of variables and the extended joinability condition can easily be integrated in the following completion algorithm. Similarly, for sake of simplicity we chose to present the algorithm on rules with a unique joinability condition but it can straightforwardly be lifted to any conjunction of joinability conditions.

3.1 Reachable terms for conditions in a CTRS

Let us consider a conditional-join rule of type $l \rightarrow r$ if $s \downarrow t$. For a substitution σ, the term $l\sigma$ rewrites into $r\sigma$ if and only if exists a term u such that $s\sigma \rightarrow^* u$ and $t\sigma \rightarrow^* u$ with the considered CTRS. Note that this is similar to $l \rightarrow r$ if and only if $\mathcal{R}^*(s\sigma) \cap \mathcal{R}^*(t\sigma) \neq \emptyset$. This condition becomes $\mathcal{R}^*(c_1\sigma) \cap \ldots \cap \mathcal{R}^*(c_n\sigma) \neq \emptyset$ for an extended joinability condition $\downarrow (c_1, \ldots, c_n)$. Similarly for any conjunction of conditions $c_1 \downarrow c'_1 \wedge \ldots c_n \downarrow c'_n$, it becomes $\mathcal{R}^*(c_1\sigma) \cap \mathcal{R}^*(c'_1\sigma) \neq \emptyset \wedge \ldots \wedge \mathcal{R}^*(c_n\sigma) \cap \mathcal{R}^*(c'_n\sigma) \neq \emptyset$. Consequently conditions can be evaluated using reachable terms. The way to verify the truth of a condition should be to build the sets $\mathcal{R}^*(s\sigma)$ and $\mathcal{R}^*(t\sigma)$ in order to check if they have common reachable terms or not. Let us remark that performing such a computation with an over-approximation may lead to assign true for a condition that may be false. However this is coherent with an over-approximation of $\mathcal{R}^*(E)$; informally, since conditional rules may apply more often in an over approximation than in the exact case, more descendants are produced, which is still an over approximation of $\mathcal{R}^*(E)$.

Lemma 3.1 *Let $\mathcal{R}$ be a left-linear TRS, $\mathcal{A}'$ be the result of computation of the completion algorithm applied to a set $E = \mathcal{L}(\mathcal{A})$, then $\mathcal{A}'$ is closed by rewriting w.r.t $\mathcal{R}$, i.e: if $l \rightarrow^*_{\mathcal{R}} r$ and $\exists q \in \mathcal{Q}_{\mathcal{A}}, l \in \mathcal{L}(\mathcal{A}, q)$ then $r \in \mathcal{L}(\mathcal{A}', q)$.*

The proof of this corollary is described in [8] and is part of the proof for the correctness of the completion method over a TRS. We intend to reuse this intermediate result to compute separately the reachable terms from useful conditional terms.

3.2 Completion over regular set of terms for a CTRS

We first define a rewriting relation $t \xrightarrow{\downarrow n}_{\mathcal{R}} s$ meaning that to rewrite t into s, it is necessary to evaluate at most n conditions (n is called the depth of the derivation in [6]).

Definition 3.2 For a CTRS $\mathcal{R}$ with a subset $\mathcal{R}_{nc}$ of non conditional rules, we note $\xrightarrow{\downarrow n}_{\mathcal{R}}$ the relation defined by:

- $\xrightarrow{\downarrow 0}_{\mathcal{R}} = \rightarrow_{\mathcal{R}_{nc}}$

- $a \xrightarrow{\downarrow n+1}_{\mathcal{R}} b \Leftrightarrow a \xrightarrow{\downarrow 0}_{\mathcal{R}} b$ or $\exists \sigma$ substitution, $p \in \mathcal{P}os(a)$ and $(l \to r$ if $s \downarrow t) \in \mathcal{R}$ such that $a|_p = l\sigma, b = a[r\sigma]_p$ and $\exists u \in \mathcal{T}(\mathcal{F})$ such that $s\sigma \xrightarrow{\downarrow n}{}^*_{\mathcal{R}} u$ and $t\sigma \xrightarrow{\downarrow n}{}^*_{\mathcal{R}} u$.

Note that $l \to^*_{\mathcal{R}} r$ means that $\exists n \in \mathbb{N}$ s.t. $l \xrightarrow{\downarrow n}{}^*_{\mathcal{R}} r$.

Let $\mathcal{A}_0$ be the tree automaton whose language E is the entry set of terms for the left-linear CTRS R. Let us consider the following algorithm, where we complete at each step the automaton $\mathcal{A}_i$ to an automaton A_{i+1}. The set of state Q_i is partitioned into three set of states: $Q_0 \cup Q_{i,new} \cup Q_{i,cond}$. Q_0 is the set of states of $\mathcal{A}_0$, $Q_{i,new}$ is a set of states produced by transition normalization and indexed by naturals, $Q_{i,cond}$ is a set of conditional states indexed by terms of $\mathcal{T}(\mathcal{F}, Q_i)$.

(i) from $\mathcal{A}_i = \langle \mathcal{F}, \mathcal{Q}_i, \mathcal{Q}_f, \Delta_i \rangle$, the i^{th} step of completion, we compute the automaton $\mathcal{A}_{i+1} = \langle \mathcal{F}, \mathcal{Q}_{i+1}, \mathcal{Q}_f, \Delta_{i+1} \rangle$ with the initialization: $\mathcal{Q}_{i+1} = \mathcal{Q}_i, \Delta_{i+1} = \Delta_i$.

(ii) Let us consider each *critical pair* without considering the condition of the rule. A pair (q, r) of $\mathcal{Q} \times \mathcal{T}(\mathcal{F})$ is said to be critical for a rule (α) either non conditional $l \to r$, or conditional $l \to r$ if $c_1 \downarrow c_2$ where $\sigma_{\mathcal{L}}$ is a regular language substitution $\sigma_{\mathcal{L}} = \{x_1 \to q_{i_1}, x_2 \to q_{i_2}, \ldots, x_n \to q_{i_n}\}$, where $\{x_1, x_2, \ldots, x_n\} = var(l)$, if $l\sigma_{\mathcal{L}} \to^*_{\Delta_i} q$ and $r\sigma_{\mathcal{L}} \not\to^*_{\Delta_i} q$.

(iii) for all of these critical pairs, (α) is either:
- a conditional rule: $l \to r$ if $c_1 \downarrow c_2$. There are two possibilities :
  · there are no state indexed by $c_1\sigma_{\mathcal{L}}$ or $c_2\sigma_{\mathcal{L}}$ in the conditional subset of states of $\mathcal{Q}_i$ ($q_{c_1\sigma_{\mathcal{L}}} \notin \mathcal{Q}_{i,cond}$ or $q_{c_2\sigma_{\mathcal{L}}} \notin \mathcal{Q}_{i,cond}$), then we create these two states (or the one missing) and we add to the automaton $\mathcal{A}_{i+1}$ the following transitions:

$$c_1\sigma_{\mathcal{L}} \to^*_{\Delta_{i+1}} q_{c_1\sigma_{\mathcal{L}}} \text{ and } c_2\sigma_{\mathcal{L}} \to^*_{\Delta_{i+1}} q_{c_2\sigma_{\mathcal{L}}}$$

  · there exists two states $q_{c_1\sigma_{\mathcal{L}}}$ and $q_{c_2\sigma_{\mathcal{L}}}$ in $\mathcal{Q}_i$. We have to calculate $\mathcal{L}(\mathcal{A}_i, q_{c_1\sigma_{\mathcal{L}}}) \cap \mathcal{L}(\mathcal{A}_i, q_{c_2\sigma_{\mathcal{L}}})$. If this set is empty, the condition is, for this completion step, considered as false. If it is not empty, then the condition is true and we go on processing the critical pair as if the rule were not conditional.
- a non conditional one (or it is conditional and the condition has been found true in the previous step), then we add to the automaton the transition $r\sigma_{\mathcal{L}} \to^*_{\Delta_{i+1}} q$.

(iv) the new automaton $\mathcal{A}_{i+1} = \langle \mathcal{F}, \mathcal{Q}_{i+1}, \mathcal{Q}_{f,i+1}, \Delta_{i+1} \rangle$ is the result of one step of completion of $\mathcal{A}_i$.

If there exists $i \in \mathbb{N}$ such that $\mathcal{A}_i = \mathcal{A}_{i+1}$, then $\mathcal{A}_i$ is the result. Remember that each time we add a transition to the automaton, we have to normalize it with new states (index by naturals and added in $Q_{i,new}$) and then the opportunity to make an approximation in order to limit the number of new states created for the normalization. As in the non conditional case, this completion may not have a

fixed point: we may produce infinitely many new states. However, approximation techniques similar to those of section 2 apply: let Q_{cond} be the set of new states $q_{c_1\sigma_\mathcal{L}}$ and $q_{c_1\sigma_\mathcal{L}}$ produced by conditions, $\mathcal{Q}_{new}$ the set of new states used to normalize the transitions, one may restrict in any way the set $\mathcal{Q}_{new}$ to force completion to terminate. Note that there is no need to limit the number of states of Q_{cond}, since the number of possible conditions c_1, c_2 is finite and the number of possible $\sigma_\mathcal{L}$ is finite if $\mathcal{Q}_{new}$ is.

Theorem 3.3 *Let A_0 be a tree automaton such that $\mathcal{L}(\mathcal{A}_0) \supseteq E$ and $\mathcal{R}$ a left linear CTRS. If $\mathcal{A}'$ is the result of the completion of $\mathcal{A}_0$ w.r.t $\mathcal{R}$, then $\mathcal{L}(\mathcal{A}')$ is closed with respect to $\mathcal{R}$ and $\mathcal{R}^*(E) \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A}')$*

Proof. Let $\mathcal{A}' = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$. We prove that $\forall t \in \mathcal{T}(\mathcal{F})$ s.t. $\exists q \in \mathcal{Q}', t \in \mathcal{L}(\mathcal{A}', q)$, $\forall u \in \mathcal{T}(\mathcal{F})$ s.t. $t \to_\mathcal{R}^* u$, we have $u \in \mathcal{L}(\mathcal{A}', q)$. We prove by induction that $\forall n \in \mathbb{N}, q \in \mathcal{Q}', t \in \mathcal{L}(\mathcal{A}', q)$, u s.t. $t \xrightarrow{\downarrow n}{}_\mathcal{R}^* u$, then $u \in \mathcal{L}(\mathcal{A}', q)$

- If $q \in \mathcal{Q}', t \in \mathcal{L}(\mathcal{A}', q)$, and $t \xrightarrow{\downarrow 0}{}_\mathcal{R}^* u$ then we trivially have $u \in \mathcal{L}(\mathcal{A}', q)$. Indeed, $\xrightarrow{\downarrow 0}{}_\mathcal{R}^*$ means that we consider non conditional R_{nc} subset of rules of $\mathcal{R}$ and then the proof follows from lemma 3.1.
- now suppose that for a given n: $\forall k \leq n, t \xrightarrow{\downarrow k}{}_\mathcal{R}^* u$ and $t \to_{\Delta'}^* q \Rightarrow u \to_{\Delta'}^* q$. We want to show that:

$$t \xrightarrow{\downarrow n+1}{}_\mathcal{R}^* u \text{ and } t \to_{\Delta'}^* q \Rightarrow u \to_{\Delta'}^* q$$

$t \xrightarrow{\downarrow n+1}{}_\mathcal{R}^* u$ means that exists $\{t_1, t_2, \ldots, t_j\} \subset \mathcal{T}(\mathcal{F})$ such that

$$t_0 = t \xrightarrow{\downarrow n+1}_\mathcal{R} t_1 \xrightarrow{\downarrow n+1}_\mathcal{R} t_2 \xrightarrow{\downarrow n+1}_\mathcal{R} \ldots \xrightarrow{\downarrow n+1}_\mathcal{R} t_{j-1} \xrightarrow{\downarrow n+1}_\mathcal{R} t_j = u$$

Now we show that for every t_i, if $t_i \to_{\Delta'}^* q$ then $t_{i+1} \to_{\Delta'}^* q$, this leads to two cases:

· $t_i \xrightarrow{\downarrow n}_\mathcal{R} t_{i+1}$, then using the induction hypothesis, $t_{i+1} \to_{\Delta'}^* q$.

· $t_i \not\xrightarrow{\downarrow n}_\mathcal{R} t_{i+1}$ and $t_i \xrightarrow{\downarrow n+1}_\mathcal{R} t_{i+1}$, so there exists a rule $(k)\, l \to r$ if $c_1 \downarrow c_2 \in \mathcal{R}$ a closed context $C[]$, and a substitution σ such that:

$$t_i = C[l\sigma] \to_\mathcal{R} C[r\sigma] = t_{i+1} \text{ if } c_1\sigma \downarrow c_2\sigma$$
$$\text{and } \exists c \text{ s.t. } c_1\sigma \xrightarrow{\downarrow n}{}_\mathcal{R}^* c, \text{ and } c_2\sigma \xrightarrow{\downarrow n}{}_\mathcal{R}^* c$$

Since no critical pair between $\mathcal{R}$ and Δ' exists, the automaton is a fixed point for the completion and we necessarily have that $\exists q_{c_1\sigma}, q_{c_2\sigma} \in \mathcal{Q}'$. Thus, we have:

$$c_1\sigma \xrightarrow{\downarrow n}{}_\mathcal{R}^* c \text{ and } c_1 \to_{\Delta'} q_{c_1\sigma}$$
$$c_2\sigma \xrightarrow{\downarrow n}{}_\mathcal{R}^* c \text{ and } c_1 \to_{\Delta'} q_{c_2\sigma}$$

The induction hypothesis leads to $c \in \mathcal{L}(\mathcal{A}', q_{c_1\sigma})$ and $c \in \mathcal{L}(\mathcal{A}', q_{c_2\sigma})$. Consequently, since $t_i \to_{\Delta'}^* q$, since the condition

$\mathcal{L}(\mathcal{A}', q_{c_1,\sigma}) \cap \mathcal{L}(\mathcal{A}', q_{c_2,\sigma}) \neq \emptyset$ is true, and since $\mathcal{A}$ is a fixed point for the completion for automaton $\mathcal{A}$, we necessarily have $t_{i+1} \to^*_{\Delta'} q$.

We have $t = t_0 \in \mathcal{L}(\mathcal{A}', q)$, so by induction $\forall i \leq n, t_i \in \mathcal{L}(\mathcal{A}', q)$, in particular $u = t_j$.

We get the result that $t \xrightarrow{\downarrow n+1}{}^*_{\mathcal{R}} u$ and $t \to^*_{\Delta} q$ implies $u \to^*_{\Delta} q$

So $\forall n \in \mathbb{N}, t \xrightarrow{\downarrow n}{}^*_{\mathcal{R}} u$ and $t \to^*_{\Delta'} q$ implies $u \to^*_{\Delta'} q$, then $t \to^*_{\mathcal{R}} u$ and $t \to^*_{\Delta'} q$ implies $u \to^*_{\Delta'} q$. This leads us to $\forall q \in \mathcal{Q}', \mathcal{L}(\mathcal{A}', q)$ is closed under rewriting by $\mathcal{R}$, in particular for $q \in \mathcal{Q}_f$, thus $\mathcal{L}(\mathcal{A}')$ is closed under rewriting by $\mathcal{R}$. Since completion is incremental, we have the inequalities $\Delta \subseteq \Delta'$ and thus $E \subseteq \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$, and finally $\mathcal{R}^*(E) \subseteq \mathcal{L}(\mathcal{A}')$.

□ □

4 Regular approximations of conditional descendants

The purpose of this section is to compare the automaton we obtain using the completion algorithm on CTRS w.r.t. the automaton obtained by the initial completion algorithm over a TRS encoding a CTRS. We first give an encoding of CTRS into TRS.

4.1 Encoding a CTRS into a TRS

Many encodings of CTRS into TRS were proposed (See for instance [12,1,14]). All those encodings are based on the extension or transformation of the alphabet $\mathcal{F}$: some symbols are added to model conditions or the arity of symbols of $\mathcal{F}$ is extended in order to store some conditions. We require the encoded CTRS to have the same reachable terms on $\mathcal{T}(\mathcal{F})$ though it rewrites over $\mathcal{T}(\mathcal{F}')$, i.e. if $\mathcal{R}$ is a CTRS over $\mathcal{F}$, and $\mathcal{R}'$ is its TRS encoding over $\mathcal{F}'$ (with a function Π mapping (encoded) terms of $\mathcal{T}(\mathcal{F})'$ to terms of $\mathcal{T}(\mathcal{F})$) then for all terms $s, t \in \mathcal{T}(\mathcal{F})$ and $s', t' \in \mathcal{T}(\mathcal{F})'$ such that $s = \Pi(s')$ and $t = \Pi(t')$, we have: $s \to^*_{\mathcal{R}} t$ if and only if $s' \to^*_{\mathcal{R}'} t'$. As far as we know the encoding of P. Viry [14] is the only one to preserve reachability. This is mainly due to the fact encodings are generally designed for a different purpose in a different context: most of them are designed to transform a confluent and terminating conditional term rewriting system into a terminating and confluent non conditional term rewriting system having the same normal forms. This is rather different from our setting where systems are not necessarily confluent or terminating.

With regards to approximation construction, we will also require the encoding to minimize the number of 'new' terms (representing condition evaluation and verification). We will also try to minimize their depth as well as the arity of their symbols, since each of these elements increases the complexity of the approximation construction. Intuitively, a 'new' term is a term that is not already recognized by the approximation automaton. Each subterm of a 'new' term requires one state of the automaton to be recognized. Thus minimizing new terms and their depth

minimizes the size of the automaton and the complexity – in time and space – of approximation construction. This is why, although Viry's encoding is preserving reachability, it is not fully satisfactory for our purpose. In [7], we propose another encoding of CTRS into TRS:

$$(i)l \to r \text{ if } c_1 \downarrow c_2$$

is transformed into:

$$\begin{aligned}(i_1)\ & l && \to c_i(l, r, c_1, c_2)\\ (i_2)\ & c_i(x, y, z, z) && \to y\end{aligned}$$

where $x, y, z \in \mathcal{X}$. Since this encoding is dedicated to reachability proof for CTRS (in particular it does not have to preserve termination), it is very simple and has good properties for approximations: no arity extension and fewer new terms. For instance, in our setting, it is more interesting to store l itself in $c_i(l, ...)$ than every variable of l – say $c_i(x_1, \ldots, x_n, \ldots)$ for instance – like in [1,14] which increases the arity of c_i. Similarly, since l is the left-hand side of the rule, then the subterm l of $c_i(l, \ldots)$ will be shared in the approximation and thus will not be constructed twice: in $c_i(l, \ldots)$, l is not a 'new' term, it will be recognized by the same state. Note that termination of the system is not required for the approximation, and we will show that recursive applications of the (i_1) rule can easily be solved.

Since this encoding of a CTRS $\mathcal{R}$ into a TRS $\mathcal{R}'$ (over an alphabet $\mathcal{F}' = \mathcal{F} \cup \{c_i | i = 1 \ldots n\}$) preserves reachability, one can apply the existing completion algorithm for unrestricted TRS [11]. The completion algorithm constructs a tree automaton $\mathcal{A}$ whose language is $\mathcal{L}(\mathcal{A}) \supseteq \mathcal{R}'^*(E)$. To approximate $\mathcal{R}^*(E)$, it is then necessary to project the language from $\mathcal{T}(\mathcal{F})'$ into $\mathcal{T}(\mathcal{F})$. This can easily be done by removing all the transitions of the automaton with a c_i as top symbol.

4.2 Example

In order to avoid technical details about approximation rules (see [11]) and to focus on the comparison between the extended completion algorithm and the encoding approach, we choose an example where the completion terminates without approximation.

Example 4.1 For this example, we consider an automaton recognizing the membership of 'b' in lists of 'a' and 'b'. mb designates membership. Let $\mathcal{F} = \{tt, nil, a, b : 0, cons, mb : 2\}$, $\mathcal{X} = \{x, y, l\}$ and $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, where $\mathcal{Q} = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\mathcal{Q}_f = \{q_5\}$, and

$$\Delta = \left\{ \begin{array}{l} nil \to q_0, b \to q_1, a \to q_2, cons(q_2, q_0) \to q_3 \\ cons(q_2, q_3) \to q_3, cons(q_1, q_3) \to q_4, cons(q_2, q_4) \to q_4 \\ cons(q_1, q_4) \to q_4, mb(q_1, q_3) \to q_5, mb(q_1, q_4) \to q_6 \end{array} \right.$$

$$\mathcal{R} = \begin{cases} mb(x, cons(y, l)) & \to tt \text{ if } mb(x, l) \downarrow tt \\ mb(x, cons(y, l)) & \to tt \text{ if } x \downarrow y \\ mb(x, nil) & \to \bot \end{cases}$$

$$\mathcal{R}' = \begin{cases} mb(x, cons(y, l)) & \to c(mb(x, cons(y, l)), tt, mb(x, l), tt) \\ mb(x, cons(y, l)) & \to c(mb(x, cons(y, l)), tt, x, y) \\ c(x, y, z, z) & \to y \\ mb(x, nil) & \to \bot \end{cases}$$

Note that state q_5 tests membership of 'b' in lists of 'a', and q_6 tests membership of 'b' in lists of 'a' and 'b', with 'a' for last elements. Completion over $\mathcal{A}$ with the CTRS completion method on the conditional system gives:

applied rule	**state**	**conditions**	**conditions state**	**new rules**
$mb(q_1, cons(q_2, q_0)) \to tt$	q_5	$mb(q_1, q_0) \downarrow tt$	new	$mb(q_1, q_0) \to q_{c1}, tt \to q_{c2}$
$mb(q_1, cons(q_2, q_0)) \to tt$	q_5	$q_1 \downarrow q_2$	$false$	
$mb(q_1, cons(q_2, q_3)) \to tt$	q_5	$mb(q_1, q_3) \downarrow tt$	$false$	
$mb(q_1, cons(q_2, q_3)) \to tt$	q_5	$q_1 \downarrow q_2$	$false$	
$mb(q_1, cons(q_2, q_4)) \to tt$	q_6	$mb(q_1, q_4) \downarrow tt$	$false$	
$mb(q_1, cons(q_2, q_4)) \to tt$	q_6	$mb(q_1, q_4) \downarrow tt$	$false$	
$mb(q_1, cons(q_1, q_3)) \to tt$	q_6	$mb(q_1, q_3) \downarrow tt$	$false$	
$mb(q_1, cons(q_1, q_3)) \to tt$	q_6	$q_1 \downarrow q_1$	$true$	$tt \to q_6$
$mb(q_1, nil) \to \bot$	q_{c1}			$\bot \to q_{c1}$

Applied rule and state are the elements of the considered critical pair, conditions are the conditions for the rule to be effectively applicable, condition state is the current state of the condition regarding the existence of the corresponding conditional states in the automaton (*new* if there are not both present), and their intersection (*false* if it is the empty set, *true* otherwise), and new rules refer to the rules added in the automaton after normalization.

Completion over $\mathcal{A}$ with the TRS method on the translated system:

applied rule	**state**	**transition to normalize**	**new rules**
$mb(q_1, cons(q_2, q_0)) \to tt$	q_5	$c(mb(q_1, cons(q_2, q_0)), tt, mb(q_1, q_0), tt)$	$mb(q_1, q_0) \to q_7$
			$tt \to q_8$
			$c(q_5, q_8, q_7, q_8) \to q_5$
$mb(q_1, cons(q_2, q_0)) \to tt$	q_5	$c(mb(q_1, cons(q_2, q_0)), tt, q_1, q_2)$	$c(q_5, q_8, q_1, q_2) \to q_5$
$mb(q_1, cons(q_2, q_3)) \to tt$	q_5	$c(mb(q_1, cons(q_2, q_3)), tt, mb(q_1, q_3), tt)$	$c(q_5, q_8, q_5, q_8) \to q_5$
$mb(q_1, cons(q_1, q_3)) \to tt$	q_6	$c(mb(q_1, cons(q_1, q_3)), tt, mb(q_1, q_3), tt)$	$c(q_6, q_8, q_4, q_8) \to q_6$
$mb(q_1, cons(q_1, q_3)) \to tt$	q_6	$c(mb(q_1, cons(q_1, q_3)), tt, q_1, q_1)$	$c(q_6, q_8, q_1, q_1) \to q_6$
$mb(q_1, cons(q_2, q_4)) \to tt$	q_6	$c(mb(q_1, cons(q_2, q_4)), tt, mb(q_1, q_4), tt)$	$c(q_6, q_8, q_6, q_8) \to q_6$
$mb(q_1, cons(q_2, q_4)) \to tt$	q_6	$c(mb(q_1, cons(q_2, q_4)), tt, q_1, q_2)$	$c(q_6, q_8, q_1, q_2) \to q_6$
$mb(q_1, nil) \to \bot$	q_7	$\bot \to q_7$	$\bot \to q_7$
$c(q_6, q_8, q_1, q_1) \to q_8$	q_6	$q_8 \to q_6$	$q_8 \to q_6$

Applied rule, state and new rules denote the same informations as in the previous completion, while transition to normalize is given in order to show the complexity of the rule to be added with this method and to insist on the approximation. Note that the construction of $\mathcal{A}$ reflect an abstract interpretation choice (every list of 'a' is recognized by the same state q_3, etc). The second method uses approximation to converge while the first one does not need to. The TRS method over the translation produces deep terms to normalize (such as $c(mb(q_1, cons(q_2, q_0)), tt, mb(q_1, q_0), tt)$) we have chosen to normalize using existing states whenever possible. Even with this approximation, the completion produces odd terms such as $c(q_6, q_8, q_6, q_8)$ which are meaningless (q_6 rewrites into q_8 if q_6 rewrites into q_8). For both examples, we obtain $tt \in \mathcal{L}(\mathcal{A}, q_6)$ and $tt \notin \mathcal{L}(\mathcal{A}, q_5)$, and then prove that $mb(b, l) \not\to^*_{\mathcal{R}} tt$ when l is a list of $'a'$.
This example also shows that we can replace the two rules:

$$mb(x, cons(y, l)) \to tt \text{ if } mb(x, l) \downarrow tt$$
$$mb(x, cons(y, l)) \to tt \text{ if } x \downarrow y$$

by one rule :

$$mb(x, cons(y, l)) \to tt \text{ if } mb(x, l) \downarrow tt \text{ or } x \downarrow y$$

and test both conditions in the same step, i.e allow disjunctions of conditions with the CTRS method, while this is not easily possible with a translation.

5 Conclusion

In this paper, we tackle the problem of approximating reachable terms for any join Conditional Term Rewriting System. As far as we know, this is the first time that this problem is addressed. We proposed an algorithm extending the existing automata completion algorithm – implemented in the Timbuk tool – for dealing with CTRS. This extension is rather natural w.r.t. the existing algorithm and uses similar techniques, in particular for approximation construction.

We compared the automata produced by this algorithm with what could be obtained using an encoding of CTRS into TRS and the completion algorithm on TRS. This comparison is in favor of the first one. Even if the encoding was chosen so as to limit the number of new terms, limit their depth as well as their arity, the automaton produced by completion on the encoding is bigger: it recognizes more terms (right-hand side of the rules even if the condition is not true) and contains some alien information (c_i). One should note that each of these additional terms is likely to make the completion diverge and thus needs to be approximated. Hence, the extension of the completion algorithm to CTRS seems to give a more convincing answer to approximate reachable terms for CTRS and should be chosen soon for implementation in Timbuk.

References

[1] I. Alouini and C. Kirchner. Toward the concurrent implementation of computational systems. In *Algebraic and Logic Programming*, pages 1–31, 1996.

[2] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In *Proc. 1st WRLA*, volume 4 of *ENTCS*, Asilomar (California), 1996.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[4] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata/`, 2002.

[5] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990. Also as: Research report 478, LRI.

[6] N. Dershowitz, M. Okada, and G. Sivakumar. Canonical Conditional Rewrite Systems. In *Proc. 9th CADE Conf., Argonne (Ill., USA)*, volume 310 of *LNCS*, pages 538–549. Springer-Verlag, 1988.

[7] G. Feuillade and T. Genet. Reachability in conditional term rewriting systems. Technical report, Irisa, Apr. 2003. `http://www.irisa.fr/lande/genet/publications.html`.

[8] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proc. 9th RTA Conf., Tsukuba (Japan)*, volume 1379 of *LNCS*, pages 151–165. Springer-Verlag, 1998.

[9] T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *Proc. 17th CADE Conf., Pittsburgh (Pen., USA)*, volume 1831 of *LNAI*. Springer-Verlag, 2000.

[10] T. Genet, Y.-M. Tang-Talpin, and V. Viet Triem Tong. Verification of Copy Protection Cryptographic Protocol using Approximations of Term Rewriting Systems. Workshop on Issues in the Theory of Security, 2003.

[11] T. Genet and V. Viet Triem Tong. Reachability Analysis of Term Rewriting Systems with *timbuk*. In *Proc. 8th LPAR Conf., Havana (Cuba)*, volume 2250 of *LNAI*, pages 691–702. Springer-Verlag, 2001.

[12] C. Hintermeier. A transformation of canonical conditional trs's into equivalent canonical trs's. In *Proceedings of the 4th International Workshop on Conditional Rewriting Systems, Jerusalem (Israel)*, June 1994.

[13] P. Réty and J. Vuotto. Regular Sets of Descendants by some Rewrite Strategies. In *Proc. 13th RTA Conf., Copenhagen (Denmark)*, volume 2378 of *LNCS*. Springer-Verlag, 2002.

[14] P. Viry. Elimination of conditions. *Journal of Symbolic Computation*, 28(3):381–401, 1999.

MPTP 0.1 - System Description

Josef Urban [1]

Dept. of Theoretical Computer Science
Charles University
Malostranské nám. 25, Praha, Czech Republic

Abstract

MPTP (Mizar Problems for Theorem Proving) is a system for translating the Mizar Mathematical Library (MML) into untyped first order format suitable for automated theorem provers, allowing generating theorem proving problems corresponding to MML. The first version generates about 30000 problems from complete proofs of Mizar theorems, and about 630000 problems from the simple (one-step) justifications done by the Mizar checker. We describe the design and structure of the system, some limitations, and planned future extensions.

1 Availability

Mizar problems for theorem proving (MPTP) is available online at `http://alioth.uwb.edu.pl/twiki/bin/view/Mizar/MpTP`, the main packed distribution has about 70 MB and unpacks to about 100MB. It is possible to download only the basic distribution (about 300 kB) without libraries, and build the main (possibly customized) libraries from the Mizar system. MPTP should run on all sorts of Unix-like systems, where Perl and Berkeley DB are installed. You may additionally need to install the standard Perl DB_File module, providing Berkeley DB interface, which is not always included with Perl distributions. For creation of customized libraries, Mizar Linux distribution (www.mizar.org) is needed, and it is limited to Linux x86 architectures.

2 Motivation

We want to have a system that would allow closer cooperation between theorem provers and large mathematical formalization projects. It seems, that the Mizar [Rudnicky 92] Mathematical Library (MML), containing nearly 800 formalized articles with the total of about 60 MB of mathematical theories, is currently the largest

[1] Email: `urban@kti.ms.mff.cuni.cz`

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

and most mathematically oriented corpus of formalized mathematics, available for this task.

This should allow things like consistent domain-based optimization and training of theorem provers, learning of lemma conjecturing, introduction of definitions and their unfolding, and provide incentive for dealing with more "real" mathematics in theorem provers e.g. by implementing efficient type handling algorithms, or integrating some efficient decision or evaluation procedures, learning how to choose premises for solving given problem from a vast repository of available assertions, or doing theorem discovery in advanced mathematical theories. Having such optimized systems could in turn further boost the feasibility and popularity of the formalization efforts, with the hope of making computers more helpful in the science that gave birth to them.

3 Description

3.1 Overview of MPTP

MPTP 0.1 at the time of writing this description consists of the following parts:

- the main Mizar-to-ATP translation tool (fo_tool)
- Makefiles and some very simple scripts creating the translated library from fo_tool's output
- the translated Mizar library, accessible both as Prolog files and as Berkeley DB files
- Perl scripts accessing the library as Berkeley DB files, generating proof problems and providing other important functionality, like signature filtering or results parsing
- the generated proof problems
- additionally, SQL (MySQL) database of results with web interface is used for collecting and analysis of provers' results, however this is not a part of the system distribution

The system is designed so that the transition between the Mizar format of problems and generated ATP problems is as simple, transparent, and fast, as possible. This is necessary, because the translation of the more complex Mizar logic into untyped first order format is nontrivial, and requires a lot of feedback from provers run on the translated problems (see section 3.5), and also because MML (and also Mizar) is very much "living" project, growing with new articles and changing almost every week. We also want to keep the problem generating scripts simple and documented, so that it is easy for others to experiment with creating their own customized proof problems from the library.

3.2 fo_tool

fo_tool is a standalone program, based on the Mizar implementation (written in objective Pascal). Since the source code of the Mizar checker and related utilities is only available to members of the Association of the Mizar Users, we only distribute a Linux binary, executable on x86 architectures. It is not needed, unless you want to generate your own customized library. fo_tool takes a Mizar article as input, and produces several files containing the translated information about various Mizar constructors, theorems, definitions and clusters exported from the article (see [Urban 03] for details of the translation, which mainly consists in relativization of the Mizar types). Additionally, fo_tool also collects necessary information about complete proofs of all (about 30000) exported Mizar theorems. Even though we thus collect just the minimal set of references used in the Mizar proof, problems generated from these proofs can be quite hard, because of their length and some automation (e.g. type handling) in Mizar. That's why we also export the simplest Mizar single-step justifications - checker problems. There are about 630000 of them, and they should be quite easy since the strength of the Mizar checker is limited [2] .

Direct translation into the DFG format [Hähnle et all 96] used by the SPASS prover [Weidenbach et all 99] was chosen for the first MPTP version, since we need the strongest prover for testing the translation, and it seems, that SPASS performs best on the translated problems, probably because of its autodetection of sort theories and use of semantic blocking [Ganzinger et all 97] of ill-typed inferences. The dfg2tptp tool (available in SPASS distribution) can be used now to translate DFG tasks to TPTP format, but it is possible, that we will make TPTP the default format in the future, or will support more than one output format.

3.3 The Exported Library

The library is now a collection of several files, usually containing formulas in DFG format, expressing some part of the translated MML structure. So all translated theorems are in one file, all definitions in another, etc., and these files are Prolog readable (though sometimes quite big). We keep small index file (also in Prolog format), telling for each library file F, and each Mizar article A, at which point of F the translated items from A are placed. Since most Mizar items (e.g. theorems, definitions, constructors, etc.) are already numbered by the Mizar system (e.g. REAL_1:70 is 70th theorem in article REAL_1), and the naming scheme used by our translation respects this numbering (again, the naming scheme is dealt with in more detail in [Urban 03]), it is thus usually very simple and fast (constant time) to compute a position of some item in a library file.

This approach now takes care of most of the indexation problems, necessary for fast access into the library files. The memory efficiency is solved by accessing the

[2] The probably most detailed available description of the methods used by the Mizar checker is in [Wiedijk 00]. Some specific methods are also discussed in [Naumowicz and Byliński 02].

library files as simple Berkeley DB databases of the RECNO (record number) type.

Additionally, the library contains input files for checker problems, again in a Prolog format. Because of the number of checker problems, these files can be quite large (several MB) even for a single article, and would occupy about 1 GB, if not compressed. So for space efficiency, we keep them compressed in a special directory. Decompression and cleanup of these files are handled by the problem generating scripts.

3.4 Problem Generating Scripts

Problems creation is implemented in about 2000 lines of documented Perl modules and scripts. The basic Perl module MPTPUtils.pm provides database access to the translated library, functions for creating the basic background theory for articles, based on their environment directives, and functions for problem printing.

The Perl module MPTPSgnFilter.pm is an implementation of a signature filter, based on reasonings about the Mizar checker, and used for cutting unnecessary context (background) formulas from the problems. The criteria for adding new background formulas are derived from close inspection of the Mizar checker's work with this information, which is quite a nontrivial matter. However, the number of background formulas cut in this way from the problem is usually pretty high, and it improves the prover's chances quite significantly.

The Perl script mkproblem.pl gets names of problems, or names of articles (preceded with "-t" for theorem problems and "-c" for checker problems) as input, and produces the problem files in DFG format. It takes about 8 minutes and about 500 MB on P4 to produce all theorem problems (about 30000), producing all checker problems takes about 10 GB. Because of this speed and the problem sizes, we do not distribute the generated problems now.

3.5 The Database of Results

To facilitate the analysis of the results of provers run on MPTP, experimental SQL (MySQL) database has been set up for them (see [MPTPResults] for its SQL structure). A web interface to the database allowing arbitrary SQL selects is at `http://lipa.ms.mff.cuni.cz/phpMyAdmin-2.4.0` . This is now mainly used to look for suspicious spots in the translation, e.g. by comparing the length of a Mizar proof, with the length of the proof found by a theorem prover.

We would like to encourage MPTP users to contribute their results into the database, however, it is necessary to say, that the structure of the database may still change a lot in the early versions.

3.6 First Results

Because of limited resources, the database of results now only contains results of the SPASS prover (version 2.0), run with 4 second timelimit on P4 on all theorem problems. For about one third of the problems (8727 out of 27298 tried), the proof

was found within the timelimit. We also tried the E prover [Schulz 99] (version 0.7) with the same timelimit (and in the automatic mode), solving 7737 problems. These results are not in the database yet, because of problems with proof extraction.

Two more experiments with SPASS have been conducted recently. In the first one, the signature filtering was not used, yielding problems that are on average very large (570 clauses/problem). Only about 2740 problems were proved within the 4 second timelimit. In the second experiment, a cluster of P3 was used on the filtered versions, raising the timelimit to 120 seconds. The number of proved problems raised to 10810. About 630 completions were found in this experiment, see the next section for some discussion of their possible causes and planned improvements.

Because of their number, large-scale statistics is not yet available on checker problems.

4 Problems, Limitations and Future Extensions

There are now several problems and limitations when using MPTP, their up-to-date description and suggested workarounds are present in files README_MPTP.txt and MPTPFAQ.txt distributed with the system.

Some problems arise from second order features of the Mizar language, used to deal with the infinite axiomatizations used by MML (Tarski-Grothendieck set theory). They can be solved by instantiation, and we are planning to do that in future versions. Other problems are caused by various automatizations implemented by Mizar. They can be dealt with either on the level of fo_tool, by watching such automatizations in Mizar, or later on the level of problem generating scripts, e.g. with methods like signature filtering. The longterm solution e.g. to arithmetical evaluations, seems to be integration of such procedures directly in ATP systems.

ATP systems that want to perform well on MPTP should also implement fast type-handling procedures, otherwise the provers may spend a lot of time, just to prove, that certain terms are properly typed.

Possible MPTP extensions include even more detailed export of the proof structure, allowing creation of harder and harder problems by following the proof structure (expanding lemmas), more experiments with the translation of Mizar types, (see [Dahn 98] for some suggestions), and maybe even direct translation to CNF format, to have consistency of skolem symbols. However the main goal remains to have at least one prover optimized very much for MML tasks, so that it can be of real use to Mizar authors.

References

[Dahn 98] Ingo Dahn. Interpretation of a Mizar-like Logic in First Order Logic. Proceedings of FTP 1998. pp. 137-151.

[Ganzinger et all 97] H. Ganzinger, C. Meyer, C. Weidenbach: Soft Typing for Ordered Resolution. In Proc. CADE-14, pp. 321-335, Springer, 1997.

[Hähnle et all 96] R. Hähnle, M. Kerber, and C. Weidenbach. Common Syntax of the DFGSchwerpunktprogramm Deduction. Technical Report TR 10/96, Fakultät für Informatik, Universät Karlsruhe, Karlsruhe, Germany, 1996.

[MPTPResults] MPTPResults.sql - SQL structure of the MPTP result database, published online at `http://alioth.uwb.edu.pl/twiki/bin/view/Mizar/MpTP`.

[Naumowicz and Byliński 02] Adam Naumowicz and Czesław Byliński, Basic Elements of Computer Algebra in MIZAR, Mechanized Mathematics and Its Applications Vol. 2(1), August 2002.

[Rudnicky 92] Rudnicki, P., An Overview of the Mizar Project, Proceedings of the 1992 Workshop on Types for Proofs and Programs, Chalmers University of Technology, Bastad, 1992.

[Schulz 99] Schulz S., System abstract: E 0.3. In H. Ganzinger, ed., 16th International Conference on Automated Deduction, CADE-16, Vol. 1632 of LNAI, Springer, pp. 297-301.

[Suttner and Sutcliffe 98] C. Suttner and G. Sutcliffe. The TPTP problem library (TPTP v2.2.0). Technical Report 9704, Department of Computer Science, James Cook University, Townsville, Australia, 1998.

[Urban 03] Josef Urban. Translating Mizar for First Order Theorem Provers. In Andrea Asperti, Bruno Buchberger, James Davenport (eds.), Mathematical Knowledge Management, Proceedings of MKM 2003, LNCS 2594, 2003.

[Weidenbach et all 99] Weidenbach C., Afshordel B., Brahm U., Cohrs C., Engel T., Keen R., Theobalt C. and Topic D., System description: Spass version 1.0.0, in H. Ganzinger, ed., '16th International Conference on Automated Deduction, CADE-16', Vol. 1632 of LNAI, Springer, pp 314-318

[Wiedijk 00] Freek Wiedijk. CHECKER - at `http://www.cs.kun.nl/\~{}freek/mizar/by.dvi`

VOTE : Group Editors Analyzing Tool

Abdessamad Imine [a], Pascal Molli [a], Gérald Oster [a] and Pascal Urso [b]

[a] *LORIA, INRIA - Lorraine*
Campus Scientifique, 54506 Vandoeuvre-Lès-Nancy Cedex, France
{imine,molli,oster}@loria.fr

[b] *Ecole Supérieure en Sciences Informatiques.*
930, Route des Colles, 06903 Sophia Antipolis, France
urso@essi.fr

Abstract

We present an initial version of a tool VOTE [1], for detecting copies inconsistency in group editors. As input, our tool takes an algorithmic-description which consists of the group editor behaviour and the transformation algorithm. VOTE translates this description into rewrite rules. As a verification back-end we use SPIKE, an automated induction-based theorem prover, which is suitable for reasoning about conditional theories. The effectiveness of our tool is illustrated on several case studies.

1 Motivations

A *group editor* is a system that allows for two or more users (sites) to simultaneously edit a document (a text, an image, a graphic, etc.) without the need for physical proximity and enables them to synchronously observe each others changes. In order to achieve good responsiveness, the shared document is *replicated* at the local memory of each participating user. Every operation is executed locally first and then *broadcasted* for execution at other sites. So, the operations are applied in different orders at different *replicas* (or copies) of the document. This potentially leads to *inconsistent* (or different) replicas – an undesirable situation for group editors. Let us consider the following group text editor scenario (see the figure 1): there are two sites working on a shared document represented by a string of characters. Initially, all the copies hold the string ''efect''. The document is modified with the operation $Ins(p, c)$ for inserting a character c at position p. Users 1 and 2 generate two concurrent operations: $op_1 = Ins(2, \text{''f''})$ and $op_2 = Ins(6, \text{''s''})$ respectively. When op_1 is received and executed on site 2, it produces the expected

[1] VOTE can be found at http://www-sop.inria.fr/coprin/urso/logiciels/.

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

string ''effects''. But, when op_2 is received on site 1, it does not take into account that op_1 has been executed before it. So, we obtain an *inconsistency* between sites 1 and 2.

How to maintain consistency? One proposed solution is the operational transformation approach [2]. It consists of an algorithm T, called *transformation algorithm*, which takes two concurrent operations op_1 (remote) and op_2 (local) defined on the same state and returns op'_1 which is equivalent to op_1 but defined on a state where op_2 has been applied. In Figure 2, we illustrate the effect of T on the previous example. Indeed, when op_2 is received on site 1, op_2 needs to be transformed according to op_1 as follows: $T((Ins(6,s), Ins(2,f)) = Ins(7,s)$. The insertion position of op_2 is incremented because op_1 has inserted a character at position 2, which is before the character inserted by op_2.. Next, op'_2 is executed on site 1. In the same way, when op_1 is received on site 2, it is transformed as follows: $T(Ins(2,f), Ins(6,s)) = Ins(2,f)$; op_1 remains the same because ''f'' is inserted before ''s''. Intuitively we can write the transformation T as follows:

T(*Ins*(p_1,c_1),*Ins*(p_2,c_2)) = **if** ($p_1 < p_2$) **return** *Ins*(p_1,c_1)
else return *Ins*($p_1 + 1$,c_1)
endif;

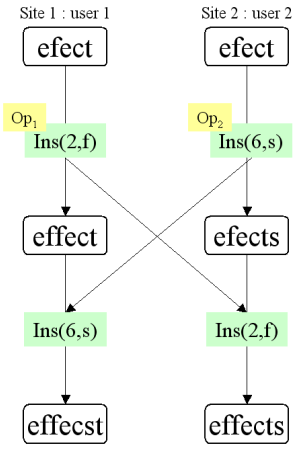


Fig. 1. Incorrect integration

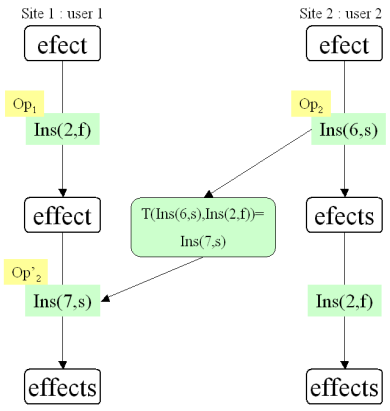


Fig. 2. Integration with transformation

However, according to [5,8,7] the transformation algorithm needs to fulfill the following conditions, in order to achieve copies consistency (we use the symbol $\circ$ to represent the sequence of operations):

Condition C_1: Let op_1 and op_2 be two concurrent operations defined on the same state. T satisfies C_1 iff:
$op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2)$. ($\equiv$ denotes a state equivalence).

Condition C_2: For any operations op_1, op_2, op_3, T satisfies C_2 iff:
$T(op_3, op_1 \circ T(op_2, op_1)) = T(op_3, op_2 \circ T(op_1, op_2))$.

Finding such a transformation algorithm for an group editor application and proving that it satisfies conditions C_1 and C_2 is not an easy task. This proof is often difficult to produce by hand and unmanageably complicated. Moreover, C_2 is particularly difficult to meet even on a simple string object. *Consequently, to be able to develop the transformational approach with simple or more complex objects, proving conditions on transformation algorithm must be automatic.*

In this paper, we present an initial version of a tool, `VOTE` (Validation of Operational Transformation Environment), for automatically checking these conditions. The input of our tool consists of a formal specification written in algorithmic style; it specifies the system behaviour in the *situation calculus* – that allows the developer to concisely describe the effects of operations on the state object without representing its inner structure explicitly – and the functional description of the transformation algorithm. The tool builds an algebraic specification described in terms of conditional equations. As a verification back-end we use `SPIKE` , an automated induction-based theorem prover, which is suitable for reasoning about conditional theories.

2 Architecture

The organization of the tool is depicted in figure 3. The main entry is a "humanly readable"description of a group editor (behaviour and transformation algorithm). The consistency conditions, C_1 and C_2, are automatically generated with respect to the input description.

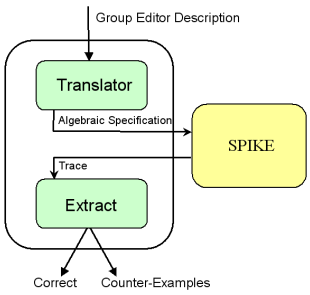


Fig. 3. Tool architecture.

Input description

More formally a group editor system is a structure of the form $G =< S_t, O, Tr >$ where:

(i) S_t is the structure of the shared object (*i.e.*, string, XML document, CAD object), (ii) O is the set of operations applied on the shared object, (iii) Tr is the transformation algorithm.

Since group editors are in essence dynamic systems, the situation calculus is especially well-suited for formalizing them [4]. This formalism allows us to reason about operations concealing the structure of the shared object. In fact, the situations

are finite sequences of operations. Starting with an initial situation, operations possible in a current situation are executed to get new situations. We observe the behaviour of the group editor through the situations. In other words, we define only the effect of each operation on the characteristics of the shared object. These characteristics are observed by *fluents* (or *observers*) which are inductively defined upon the situation by *successor state axioms*. The state of a situation is defined as being the set of fluents that hold in that situation. Accordingly, *two situations s and s' have the same state, and we denote it by $s \approx_{state} s'$ if the set of fluents that hold is the same*.

As a first step, the user describes the group editor system in algorithmic-style. Firstly, the user declares sorts of used data and the signatures of observers and operations. Every operation is preceded by a boolean expression indicating when this operation is enabled. Next, the user defines the transformation rules. This definition is complete, *i.e.* all cases should be given. Finally, the user gives the observation rules, *i.e.* successor state axioms, for every observer and operation.

Algebraic specification

In the second step, VOTE translates the above description into algebraic specification. Let π be a group editor system. Two sorts are used: sit and opn for situations and operations respectively. Let S, $S_{bs} = \{sit, opn\}$ and $S_{is} = S \setminus S_{bs}$ be the set of all sorts, the set of basic sorts and the set of individuals sorts, respectively. We use $\natural(\omega, s)$ for denoting the number of occurrences of the sort s in the sequence ω. Then, π is modeled by an algebraic specification $SP^{\pi} = (\Sigma^{\pi}, \mathcal{A}^{\pi}x)$ where:

- $\Sigma^{\pi} = (F, X)$ is a signature. F is defined as $C \cup D$, where C and D are constructor and non-constructor (or defined) functions, such that: (i) $C_{\epsilon,sit} = \{S_0\}$, $C_{opn\ sit,sit} = \{\bullet\}$ and $C_{\omega,s} = \emptyset$ if $s \in sit$ or ω contains an element of S_{bs}. (ii) $D_{opn\ opn,opn} = \{T\}$, $D_{opn\ sit,bool} = \{poss\}$ and $D_{\omega,s} = \emptyset$ if either $s \in sit$, ω contains an element of sort opn, or $\natural(\omega, sit) > 1$. (iii) X is S-indexed family of sets.
- $\mathcal{A}^{\pi}x = \mathcal{D}_{S_0} \cup \mathcal{D}_P \cup \mathcal{D}_{SS} \cup \mathcal{D}_T$ is the set of axioms (written as conditional equations) such that: (i) $\mathcal{D}_{S_0}$ is the set of axioms describing the initial situation, S_0; (ii) $\mathcal{D}_P$ is the set of operation precondition axioms, *i.e.* $poss$; (iii) $\mathcal{D}_{SS}$ is the set of successor state axioms for every fluent; (iv) $\mathcal{D}_T$ contains axioms corresponding to the transformation function T.

The sort sit has two constructor functions: the constant constructor S_0 and the constructor symbol $\bullet$. The set $C_{\omega,s}$ ($\omega \in S_{is}^*$) contains all constructor operations which represent the operation types of π. All the necessary conditions for the execution of an operation are given by $\mathcal{D}_P$. The set $D_{\omega\ sit,s}$ contains all fluent symbols, where $\omega \in S_{is}^*$ and s is an element of S_{is}; these ones are used to define the observations related to the characteristics of the shared object. Precisely when π evolving, the change of these characteristics is described by the set of successor state axioms, $\mathcal{D}_{SS}$. Finally, the transformation algorithm used by π is given as a set

of axioms $\mathcal{D}_T$.

Proving consistency conditions

As a verification back-end we use SPIKE [1,6], first-order implicit induction prover. SPIKE was chosen for the following reasons: (i) its high automation degree, (ii) its ability on case analysis (to deal with multiple operations and many case of transformations), (iii) its *refutational completeness* (to find counter-examples), (iv) its incorporation of *decision procedures* (to automatically eliminate arithmetic tautologies produced during the proof attempt [2]). In the sequel, we use the following notations: (i) $[b_1, \ldots, b_n] \bullet s = b_n \bullet ([b_1, \ldots, b_{n-1}] \bullet s)$, and, (ii) $Legal([b_1, \ldots, b_n], s) = poss(b_1, s) \wedge \ldots \wedge poss(b_n, [b_1, \ldots, b_{n-1}] \bullet s)$, where $b_1, \ldots, b_n$ are terms of sort opn and s is of sort sit. We use also $\models_{Ind}$ for denoting the inductive consequence.

The consistency conditions are formulated as theorems to be proved. Let $SP^\pi = (\Sigma^\pi, \mathcal{A}^\pi x)$ be an algebraic specification modeling an group editor system π. The first condition C_1 expresses a *semantic equivalence* between two operation sequences. Given two operations op_1 and op_2, the sequences $[op_1, T(op_2, op_1)]$ and $[op_2, T(op_1, op_2)]$ must produce the same state.

Theorem 2.1 (Condition C_1).
If for all operations op_1 and op_2, and for all $n+1$-ary fluent f:

$$\begin{aligned}
&\mathcal{A}^\pi x \models_{Ind} Legal([op_1, T(op_2, op_1)], s_1) = true \\
&\wedge Legal([op_2, T(op_1, op_2)], s_2) = true \\
&\wedge f(x_1, \ldots, x_n, s_1) = f(x_1, \ldots, x_n, s_2) \\
&\Longrightarrow f(x_1, \ldots, x_n, [op_1, T(op_2, op_1)] \bullet s_1) = \\
&f(x_1, \ldots, x_n, [op_2, T(op_1, op_2)] \bullet s_2)
\end{aligned}$$

holds then,

$$\begin{aligned}
&Legal([op_1, T(op_2, op_1)], s_1) = true \wedge Legal([op_2, T(op_1, op_2)], s_2) = true \\
&\wedge s_1 \approx_{state} s_2 \Longrightarrow [op_1, T(op_2, op_1)] \bullet s_1 \approx_{state} [op_2, T(op_1, op_2)] \bullet s_2
\end{aligned}$$

also holds.

The second condition C_2 stipulates a *syntactic equivalence* between two operation sequences. Given three operations op_1, op_2 and op_3, transforming op_3 with respect two sequences $[op_1, T(op_2, op_1)]$ and $[op_2, T(op_1, op_2)]$ must give the same operation.

Theorem 2.2 (Condition C_2).
For all operations op_1, op_2 and op_3: $\mathcal{A}^\pi x \models_{Ind} T(op_3, [op_1, T(op_2, op_1)]) = T(op_3, [op_1, T(op_1, op_2)])$.

All axioms of $\mathcal{A}^\pi x$ are automatically oriented into rewrite rules by SPIKE . For proving theorem 2.1 (resp. 2.2), SPIKE replaces first the variables op_1 and op_2 (resp. op_1, op_2 and op_3) with the elements of the test set describing the sort

[2] like $x + z > y = false \wedge z + x < y = false \Longrightarrow x + z = y$

opn. This replacement generates many instances of the theorem to be verified, enabling to cover all possible cases. Next, SPIKE simplifies these instances by rewriting. The proof of C_1 and C_2 is either successful and transformation algorithm is verified, or failed and the SPIKE 's proof-trace is used by VOTE to extract the problematic cases to the user. In the later case, there are two possibilities. The first one concerns valid conjectures where appear undefined auxiliary functions or arithmetic symbols which SPIKE 's decision procedure cannot manage; in this case, the user can introduce lemmas. The second one concerns cases violating condition C_1 or C_2. VOTE gives the scenario (operation and conditions) of each cases to help user to rectify its transformations.

3 Experiments

We have detected a lot of bugs in well-known group editors such that GROVE [2], Joint Emacs [5], REDUCE [3] [8] and SAMS [4] [3] which are based on transformational approach for maintaining consistency of shared data. The results of our experiments are reported in table 1. GROVE, Joint Emacs and REDUCE are group text editor whereas SAMS is XML document-based group editor. $S5$ [5] is a file synchronizer which uses a transformation algorithm for synchronizing many file systems replicas.

Group editors	C_1	C_2
GROVE	violated	violated
Joint Emacs	violated	violated
REDUCE	correct	violated
SAMS	correct	violated
$S5$	correct	violated

Table 1
Case studies.

Let consider the group text editor GROVE designed by Ellis and Gibbs – the pioneers of the operational transformation. The text is modified by two operations: (i) $Ins(p, c, pr)$ to insert a character c at position p. (ii) $Del(p, pr)$ to delete the character located at position p. The pr parameter represents the priority (site identifier where the operation is generated). Let us consider the following transformations:

T(*Ins*(p_1, c_1, pr_1), *Del*(p_2, pr_2)) =
  if ($p_1 < p_2$) **then return** *Ins*(p_1, c_1, pr_1)

[3] http://www.cit.gu.edu.au/$\backsim$scz/projects/reduce
[4] http://woinville.loria.fr/sams
[5] http://woinville.loria.fr/S5

else return *Ins*$(p_1 - 1, c_1, pr_1)$
endif;

T(*Del*(p_1, pr_1), *Ins*(p_2, c_2, pr_2)) =
if $(p_1 < p_2)$ **then return** *Del*(p_1, pr_1)
else return *Del*$(p_1 + 1, pr_1)$
endif;

After submitting this system to VOTE , it has detected that condition C_1 is violated by giving the counter-example depicted in figure 4. The counter-example is simple: (i) $user_1$ inserts x in position 2 (op_1) while $user_2$ concurrently deletes the character at the same position (op_2). (ii) When op_2 is received by site 1, op_2 must be transformed according to op_1. So $T(Del(2), Ins(2, x))$ is called and $Del(3)$ is returned. (iii) In the same way, op_1 is received on site 2 and must be transformed according to op_2. $T(Ins(2, x), Del(2))$ is called and return $Ins(1, x)$. Condition C_1 is violated. Accordingly, the final results on both sites are different.

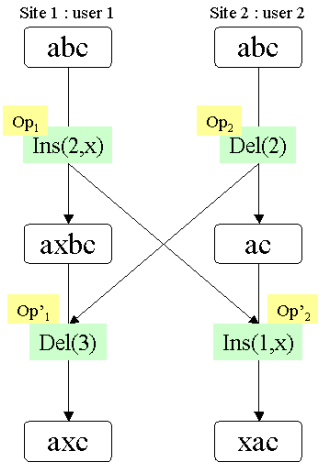


Fig. 4. Counter-example violating condition C_1

The error comes from the definition of $T(Ins(p_1, c_1, pr_1), Del(p_2, pr_2))$. The condition $p_1 < p_2$ should be rewritten $p_1 \leq p_2$.

4 Conclusion

This tool is a first step towards to assist the development of correct transformation algorithms in order to ensure copies consistency in group editors. We have detected bugs in many well-known systems. So, we think that our approach is very valuable because: (i) it can help significantly to increase confidence in a transformation algorithm; (ii) having the theorem prover ensures that all cases are considered and quickly produces counter-example scenarios;

Many features are planned to deal effective and large systems. We plan to ensure the correct composition of many transformation algorithms for handling composed objects. Finally, we intend to improve strategy proofs underlying to

SPIKE for increasing more the degree of automation.

References

[1] Bouhoula, A., E. Kounalis and M. Rusinowitch, *Automated Mathematical Induction*, Journal of Logic and Computation **5(5)** (1995), pp. 631–668.

[2] Ellis, C. A. and S. J. Gibbs, *Concurrency Control in Groupware Systems*, , **18**, 1989, pp. 399–407.

[3] Molli, P., H. Skaf-Molli, G. Oster and S. Jourdain, *SAMS: Synchronous, Asynchronous, Multi-synchronous Environments*, in: *The Seventh International Conference on CSCW in Design*, Rio de Janeiro, Brazil, 2002.

[4] Pirri, F. and R. Reiter, *Some Contributions to the Metatheory of the Situation Calculus*, Journal of the ACM **46(3)** (1999), pp. 325–361.

[5] Ressel, M., D. Nitsche-Ruhland and R. Gunzenhauser, *An Integrating, Transformation-oriented Approach to Concurrency Control and Undo in Group Editors*, in: *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'96)*, Boston, Massachusetts, USA, 1996, pp. 288–297.

[6] Stratulat, S., *A General Framework to Build Contextual Cover Set Induction Provers*, Journal of Symbolic Computation **32** (2001), pp. 403–445.

[7] Suleiman, M., M. Cart and J. Ferrié, *Concurrent Operations in a Distributed and Mobile Collaborative Environment*, in: *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA* (1998), pp. 36–45.

[8] Sun, C., X. Jia, Y. Zhang, Y. Yang and D. Chen, *Achieving Convergence, Causality-preservation and Intention-preservation in Real-time Cooperative Editing Systems*, ACM Transactions on Computer-Human Interaction (TOCHI) **5** (1998), pp. 63–108.

Automatic Theorem Proving in Calculi with Cut

Elmar Eder [1]

Department of Scienctific Computing
University of Salzburg
Salzburg, Austria

Abstract

In automated theorem proving, some proof calculi such as W. Bibel's connection method or E. Beth's and R. Smullyan's tableau calculus are based on backward reasoning in G. Gentzen's sequent calculus. For efficiency of search, these calculi must keep the search tree finitely branching, and therefore have to use the sequent calculus without the cut rule. Since the cut rule allows an immense reduction of the length of proof for some classes of formulas, it would be desirable, however, to use a calculus with cut such as the full sequent calculus or Frege-Hilbert calculi. The rules in such calculi have formula schemes as premises and conclusions. One way to avoid an infinitely branching search tree is to lift the idea of unification from terms to formula schemes. This idea allows to define an operation of composition of rules of the calculus. Since formula schemes do not admit a single most general unifier, only a partial unification can be made, and a condition has to be added to the resulting rule restricting its applicability and thus guaranteeing the equivalent of full unification. The goal of the present research is to obtain a better characterization and understanding of rules of such calculi and of their composition, in order to build automatic or interactive theorem provers for forward, backward, and bidirectional reasoning.

In automated theorem proving in first order predicate logic, proof calculi derived from Gerhard Gentzen's sequent calculus [3] without cut, have been frequently used successfully. Examples of such calculi are the various variants of E. Beth's and R. Smullyan's tableau calculus and W. Bibel's connection method. The idea behind these calculi is, roughly speaking, to construct a sequent calculus proof of a formula in a backward direction, starting from the formula to be proved, and ending at the axioms. Thus, at each instant of time, there is an agenda list of formulas yet to be proved valid. A proof step consists in choosing a formula from the agenda list, in finding a rule of the sequent calculus such that the chosen formula is the conclusion of an instance of this rule, and in replacing the chosen formula in the agenda list by the premises of this rule instance.

[1] Email: eder@cosy.sbg.ac.at

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The reason that such a proof procedure is successful in automated theorem proving, is that it is analytic. This means that a proof step consists essentially of splitting a formula (occurring in the proof constructed so far) to its immediate subformulas. The proof step is uniquely determined by choosing such a subformula. So, at each proof step, there is only a finite number of possibilities of choice. The search tree is finitely branching, which is an important condition for a successful automatic proof procedure.

On the other hand, there are proof calculi for first order predicate logic which have a cut rule such as modus ponens $\frac{A \quad A \to B}{B}$. Examples of such calculi are Frege-Hilbert calculi and Gentzens sequent calculus with cut. Gentzen [3] has proved that the cut rule can be eliminated in his calculus. But Statman [6] and Orevkov [4] have shown that there are classes of formulas for which this is only possible at the cost of immense [2] (non-elementary) increase of the length of the proof. So it seems to be highly desirable to be able to do automated theorem proving in calculi with cut. The sequent calculus and Frege-Hilbert calculi can simulate each other at low polynomial cost, as has been shown in [1]. So it does not matter very much which of the two types of cut calculi we choose.

However, a cut rule such as modus ponens is not easily applied backward automatically. For, the premises A and $A \to B$ are not uniquely determined by the conclusion B. You would have to guess the formula A, and there are an infinite number of possibilities of choosing A. This makes automated backward reasoning in calculi with cut a difficult job.

In [2], it has been shown how this difficulty could be overcome. The idea is to look at the way that automatic theorem provers treat a rule such as $\frac{F(t)}{\exists x\, F(x)}$. From the conclusion $\exists x\, F(x)$, the premise $F(t)$ is not uniquely determined, since t is unknown. Automatic theorem provers circumvent this difficulty by constructing the non-closed formula $F(x)$ as a premise and determining the value t for the variable x later through unification. A similar idea of unification, but not on the level of terms but on the level of formulas, works for the cut rule. Here variables have to be introduced to denote formulas instead of terms. Using such metavariables for formulas, for ground terms, for constants, and for object variables, *formula schemes* can be built. Replacing the metavariables with formulas, ground terms, constants, and object variables, resp., in a formula scheme yields a formula.

In this formulation, a rule in the Frege-Hilbert calculus has as its premises and conclusion formula schemes. It is possible to construct compositions of rules (and axioms) of Frege-Hilbert calculi using this idea of unification. However, for formula schemes it is in general not possible to find a single mgu as it is for terms.

The sort of problems occurring in unification may be seen with a simple example. In a Frege-Hilbert calculus, a rule may have a premise $F(s)$, and another rule

[2] There is a sequence (F_n) of formulas such that the following holds. There is a polynomial p such that each F_n has a proof of length $\leq p(n)$ in the full sequent calculus with the cut rule. But the shortest proof of F_n in the sequent calculus without cut has a length $\geq \underbrace{2^{2^{\cdots^{2^2}}}}_{n \text{ times}}$.

a conclusion $G(t)$, where F and G are metavariables. Now, if we want to combine these two rules by joining the premise of the first rule to the conclusion of the second rule, we have to unify the formula scheme $F(s)$ with the formula scheme $G(t)$. One way to do this is to substitute $H(*,t)$ for $F(*)$, and $H(s,*)$ for $G(*)$. This substitution unifies the formula schemes $F(s)$ and $G(t)$ to $H(s,t)$. But we might, instead, substitute $f(t)$ for s and $F(f(*))$ for $G(*)$. It would unify the two formula schemes $F(s)$ and $G(t)$ to $F(f(t))$. A more general substitution than these two substitutions is to substitute $f(t)$ for s, $J(*,*,t)$ for $F(*)$, and $J(f(*),s,*)$ for $G(*)$. It unifies the formula schemes $F(s)$ and $G(t)$ to $J(f(t),f(t),t)$, but it is not a most general unifier.

In order not to have to deal with a great or possibly infinite number of unifiers, we unify only the overall logical structures of formula schemes. This *pseudounification* does not guarantee that two formula schemes become identical. A finite number of conditions must be explicitly stated in each rule to ensure this. Such conditions are eigenconstant conditions stating that some constant must not occur in some formula, as well as conditions stating that substituting some variables with some terms in one formula yields the same result as substituting some other variables with some other terms in another formula.

For example, a combination of axioms and rules of a Frege-Hilbert calculus yields that $\exists x \forall u\, F \rightarrow \forall y \exists z\, G$ holds if there are distinct constants a and b not occurring in F or G, and terms s and t, such that the result of substituting x with a and u with t in F is identical to the result of substituting y with b and z with s in G.

If $A_1, \ldots, A_n, B$ are given formulas such that B follows semantically from the universal closures of $A_1, \ldots, A_n$, then we call the figure $\frac{A_1 \ldots A_n}{B}$ *valid*. For example, the figures $\frac{P(y)}{\forall x\, P(x)}$ and $\frac{P(a)}{\exists x\, P(x)}$ are valid. They give rise to Frege-Hilbert rules $\frac{F(a)}{\forall x\, F(x)}$ and $\frac{F(t)}{\exists x\, F(x)}$, respectively. An instance of the Frege-Hilbert rule is obtained from the valid figure by replacing constants with terms, bound variables with bound variables, free variables with constants, and function and predicate symbols with nominal forms in the sense of Schütte [5]. Free variables give rise to eigenconstant conditions. The example of the last paragraph cannot be generated from a valid figure. Thus the combination of rules leads to a more general class of rules than valid figures do.

My research interests are to obtain a better characterization of rules and their combination. This would allow to build an automatic and/or interactive theorem prover which could use forward, backward, and bidirectional reasoning for Frege-Hilbert calculi and also for sequent calculi.

References

[1] Eder, E., "Relative Complexities of First Order Calculi," Artificial Intelligence, Vieweg, Wiesbaden, 1992, (Wolfgang Bibel and Walther von Hahn, editors).

[2] Eder, E., *Backward reasoning in systems with cut*, in: *Artificial Intelligence and Symbolic Computation, International Conference, AISMC-3*, Lecture Notes in

Computer Science 1138 (1996), pp. 339–353.

[3] Gentzen, G., *Untersuchungen über das logische Schließen*, Mathematische Zeitschrift **39** (1935), pp. 176–210, 405–431.

[4] Orevkov, V. P., *Lower Bounds for Increasing Complexity of Derivations after Cut Elimination*, Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta im V. A. Steklova AN SSSR **88** (1979), pp. 137–161, english translation in *J. Soviet Mathematics*, 2337–2350, 1982.

[5] Schütte, K., "Proof Theory," Grundlehren der mathematischen Wissenschaften 225, Springer-Verlag, Berlin, Heidelberg, New York, 1977, transl. from German.

[6] Statman, R., *Lower bounds on Herbrand's theorem*, Proc. AMS **75** (1979).

Dialogue Games for Modelling Proof Search in Non-classical Logics

Christian G. Fermüller

Technische Universität Wien, Austria

To provide a general, rigorous, and useful mathematical analysis of proof search strategies is a great challenge; in particular if also the degree of possible parallelization of proof search is in focus. For classical logic, models for some types of proof search, including distributed search, have been presented (see, e.g., [4,5]); but the topic still seems to be far from fully explored.

For non-classical logics the situation is worse: proof search methods are often presented without paying much attention to strategies. For many logics researchers are content with the presentation of any calculus that is sound and complete for the logic in question and promises to be a reasonable base for proof search algorithms solely in virtue of the analyticity of its rules.

We suggest that dialogue games provide a uniform, versatile and elegant tool for the modelling of proof search based on analytic (tableau style) calculi for a wide range of non-classical logics. Logical dialogue games come in many forms and versions, nowadays. Instead of using more recent formulations in the style of Blass [2], Abramsky [1] and others, we refer to Paul Lorenzen's original idea (see, e.g., [10]) to identify logical validity of a formula F with the existence of a winning strategy for a *proponent* $\mathbf{P}$ in an idealized confrontational dialogue, in which $\mathbf{P}$ tries to uphold F against systematic doubts by an *opponent* $\mathbf{O}$. (For more recent literature on Lorenzen style dialogue games see [6], [9], [11].)

Recent results (see [7,8]) show that various intermediate logics — including intuitionistic and classical logic themselves — can be characterized by parallel versions of Lorenzen style dialogues. Work that is still in progress indicates that a much wider range of (analytic) logics can be analyzed in this way.

To see how dialogue games are useful for modelling proof search, one better interprets a dialogue as a collaborative effort to check the validity of a formula, rather than as a confrontational dialogue between the two players. Borrowing terminology from Andreas Blass [3] (who in turn credits unpublished work of Dexter Kozen) we may speak of a *client* $\mathbf{C}$, who wants to check whether an 'initial formula' F follows from given assumptions $G_1, \ldots, G_n$, that are provided initially by a *server* $\mathbf{S}$.

Both, client and server, may send requests and provide answers. However, we stipulate that the client $\mathbf{C}$ is the 'scheduler' of the dialogue. By this we mean that

This is a preliminary version. The final version will be published in volume 86 no. 1 of Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

S always has to answer immediately to the last request by **C**, or — if **C**'s last move was not a request but an answer — has to send a request to **C**, that refers to the formula stated as answer by **C** in her last move. In contrast, answers to requests by **S** can be delayed by **C**. I.e., **C** chooses at each step in the dialogue, in which it is her turn, whether to send a new request to **S** or to answer one of the requests by **S**, that she has not yet answered at this stage.

For a wide range of logics the 'logical rules' of the dialogue game are identical and can be summarized as follows. (The rules are symmetric: if **X** is the client **C** then **Y** refers to the server **S** and *vice versa*.)

provided by **X**:	request by **Y**	answer by **X**
$A \wedge B$	$\mathsf{l}?$ or $\mathsf{r}?$ (**Y** chooses)	$A!$ or $B!$, accordingly
$A \vee B$	$\mathsf{one\text{-}of}(A, B)?$	$A!$ or $B!$ (**X** chooses)
$A \supset B$	$A!\colon\ B?$	$B!$

An exclamation mark after the formula means that this formula is provided to the dialogue partner. The signs $\mathsf{l}?, \mathsf{r}?, \mathsf{one\text{-}of}(.,.)?$, as well as formulas followed by a question mark merely serve as indicators for the type of request.

Formulas $\neg A$ are understood as abbreviations for $A \supset \bot$, where $\bot$ denotes *falsum*, i.e., a formula that cannot be validated.

Observe that there are no logical rules referring to atomic formulas. However it is convenient to think of all atomic formulas that are stated by **C** during the course of a dialogue as 'requested' by **S**.

A client–server session (dialogue) proceeds by strictly alternating moves (i.e., requests or answers) between **C** and **S**. **S** starts the dialogue with a request that refers to **C**'s initial formula. A state of a dialogue is denoted by a *dialogue sequent* $\Pi \vdash C$, where Π denotes the set of formulas provided by the server **S** so far (up to this state); and C denotes the last formula to which **S** has referred in a request sent to **C**, but which has not yet been answered by **C**. C_i is called *current formula*.

A client–server session constrained by the just described protocol is called **I**-*dialogue*. It ends with **C** 'winning', if one of the following situations arise:

(i) the answer to a request by **S** (i.e., a request referring to a formula which **C** has stated) has already been provided by **S** itself in a previous move, or

(ii) a formula requested by **S** has already been initially provided by **S**, or

(iii) **S** is providing $\bot$.

More concisely: **C** wins the dialogue if it is in a state of form $\Pi \cup \{A\} \vdash A$ or $\Pi \cup \{\bot\} \vdash A$.

Winning strategies for **C** in the sketched game correspond to analytic tableau proofs for intuitionistic logic. *Proof search strategies*, i.e., strategies that guarantee the (efficient) construction of a closed tableau, if one exists, correspond to *uniform winning strategies* for **C** in the dialogue game. I.e., generic winning strategies that are parameterized by the formulas on which a concrete instance of the game is

to be played. E.g., the strategies for intuitionistic proof search described in [12] can easily be modeled as preferences of choice among **C**'s possible moves in a dialogue.

The aptness of the dialogue game framework for the formalization of proof search strategies for a *wider variety of logics* only becomes apparent once *parallel* dialogues with different rules for synchronizing (or rather: merging) dialogues are taken into account. We consider parallel versions of **I**-dialogue games, that share the following features:

(i) The rules for **I**-dialogues remain unchanged. Indeed, ordinary **I**-dialogues appear as sub-case of the more general parallel framework.

(ii) The client **C** may initiate additional client–server sessions (i.e., **I**-dialogues) by simply 'cloning' the dialogue sequent of one of the parallel component **I**-dialogues of the game.

(iii) To win a set of parallel dialogues the client **C** has to win at least one of the component dialogues.

Parallel dialogue games refer to *global dialogue states*

$$\{\Pi_1 \vdash_{\iota 1} C_1, \ldots, \Pi_n \vdash_{\iota n} C_n\},$$

where the $\Pi_i \vdash_{\iota i} C_i$ are **I**-dialogue sequents denoting the states of the indivdual *component dialogues* of the parallel game.

Synchronization rules that are adequate for intuitionistic logic, Gödel-Dummett logic, Jankov's logic of weak excluded middle, and all finite valued Gödel logics (including classical logic) are presented in [8].

To provide a concrete example we state the synchronization rule that allows to characterize Gödel-Dummett logic **LC**:

LC-*merge– client part***:** **C** picks two components $\Pi_1 \vdash_{\iota 1} C_1$ and $\Pi_2 \vdash_{\iota 2} C_2$ from the current global state, which she wants to merge ('synchronize'). She thus indicates that $\Pi_1 \cup \Pi_2$ will be the formulas provides by the server in the resulting merged **I**-dialogue.

LC-*merge– server part***:** In response to the client's move, **S** chooses either C_1 or C_2 as the current formula of the merged **I**-dialogue (which is indexed by $\iota 1$ or $\iota 2$, correspondingly).

Synchronization rules for other intermediate logics take a similar form. If one wants to model 'resource bounded' logics, like contraction-free versions of intuitionistic logic, Łukasiewicz logic or fragments of linear logic, then specific additional constraints on the protocol for the underlying **I**-dialogues have to be introduced.

In current and future research we (will) demonstrate that parallel dialogue games allow to formalize and compare different *proof search strategies* for the mentioned logics. In particular, these games facilitate the study of algorithms that distribute proof obligations over parallel processors.

References

[1] S. Abramsky, R. Jagadeesan: Games and Full Completeness for Multiplicative Linear Logic. *J. Symbolic Logic*, 59(2) (1994), 543–574.

[2] A. Blass: A Game Semantics for Linear Logic. *Annals of Pure and Applied Logic*, 56(1992), 183–220.

[3] A. Blass: Is Game Semantics Necessary? In: Computer Science Logic – 7th Workshop, CSL '93, Selected Papers, Springer LNCS 832, 1994, 66–77.

[4] M.P. Bonacina: A taxonomy of parallel strategies for deduction, *Annals of Mathematics and Artificial Intelligence* 29(1–4), 223–257, 2000.

[5] M.P. Bonacina, J. Hsiang: On the modelling of search in theorem proving – Towards a theory of strategy analysis, *Information and Computation* 147, 171–208, 1998

[6] W. Felscher: Dialogues as Foundation for Intuitionistic Logic. In: D. Gabbay and F. Günther (eds.), *Handbook of Philosophical Logic, III*, Reidel, 1986, 341–372.

[7] C.G. Fermüller, A. Ciabattoni: From Intuitionistic Logic to Gödel-Dummett Logic via Parallel Dialogue Games. 33rd Intl. Symp. on Multiple-Valued Logic, Tokyo May 2003, IEEE Press, to appear.

[8] C.G. Fermüller: Parallel Dialogue Games and Hypersequents for Intermediate Logics, Proceedings of *TABLEAUX 2003*, Automated Reasoning with Analytic Tableaux and Related Methods. Roma, Italy, 9-12 September 2003, Springer, to appear.

[9] E.C.W. Krabbe: Formal Systems of Dialogue Rules. *Synthese*, 63(1985), 295–328.

[10] P. Lorenzen: Logik und Agon. In: *Atti Congr. Internat. di Filosofia*, Vol. 4 (Sansoni, Firenze, 1960), 187–194.

[11] S. Rahman: *Über Dialoge, Protologische Kategorien und andere Seltenheiten*. Europäische Hochschulschriften, Peter Lang, 1993.

[12] N. Shankar: Proof Search in the Intuitionistic Sequent Calculus. Proceedings 11th Intl. Conf. on Automated Deduction, CADE'92, LNCS 607, Springer, 1992.

Author Index