Proceedings of the 6th International Workshop on First-Order Theorem Proving FTP 2007

September 12–13, 2007
University of Liverpool
Liverpool, United Kingdom

Editor:
Silvio Ranise

Technical Report ULCS-07-018
Department of Computer Science
University of Liverpool

UNIVERSITY OF LIVERPOOL

Supported by
EPSRC Grant EP/F014058/1

Preface

This volume contains the proceedings of FTP'07, the sixth workshop on First-Order Theorem Proving, held September 12 and 13, 2007, in Liverpool, England (UK). As for the previous events of this series, the focus of this workshop is on first-order theorem proving as a core theme of Automated Deduction, and its aim is to provide a forum for presentation of recent work and discussion of research in progress. The workshop was co-located with the sixth International Symposium on Frontiers of Combining Systems (FroCoS'07), held September 10-12, 2007 also in Liverpool. On September 12, 2007, there was also a joint session with Viorica Sofronie-Stokkermans as (joint) invited speaker.

These proceedings contain seven regular papers and the abstract of three "presentation-only" papers (i.e. papers submitted or accepted for publication elsewhere), each of which was reviewed by three referees. The regular papers present and discuss various topics related to first-order theorem proving such as a calculus for clauses admitting existential quantifiers (called geometric resolution), a reasoning system based on Manna and Waldinger's tableaux to be used in education, theorem proving for functional programming, simplification of complex proof obligations obtained from program verification, automated proofs of a modularity result for the termination of term rewriting systems, inductive theorem proving in the context of term rewriting systems modulo some equational theories. The three "presentation-only" papers selected by the program committee present simulation results for propositional modal logic calculi by first-order resolution, discuss the support that can be provided by first-order resolution for theorem proving in the monodic fragment of first-order temporal logic, and introduce a calculus for an extension of temporal logic. For these, only abstracts have been included in the proceedings.

I welcomed three invited lectures: one by Viorica Sofronie-Stokkermans on "Hierarchical and Modular Reasoning in Complex Theories: The Case of Local Theory Extensions" (joint with FroCoS'07), one by Martin Giese on "Aspects of First-order Reasoning in the KeY system", and one by Bernd Fischer on "Applying FTPs in Formal Software Safety Certification.". Short abstracts of all talks are included in this volume.

I would like to thank the members of the program committee and one external referee for their care and time in reviewing the submitted papers. I would also like to thank the members of the local organisation committee. In particular, I express my gratitude to the local organisation committee chair, Ullrich Hustadt, for his help and support in all phases of the workshop.

Finally, I gratefully acknowledge the financial support provided by EPSRC grant EP/F014058/1 to this workshop.

Silvio Ranise
LORIA and INRIA-Lorraine
Nancy, August 2007

Program Chair

Silvio Ranise	LORIA and INRIA-Lorraine, Nancy, Fance

Program Committee

Peter Baumgartner	NICTA, Canberra, Australia
Bernhard Beckert	University of Koblenz-Landau, Koblenz, Germany
Reiner Hähnle	Chalmers University of Technology, Göteborg, Sweden
Ullrich Hustadt	University of Liverpool, England, UK
Alexander Leitsch	TU Vienna, Austria
William McCune	University of New Mexico, Albuquerque, New Mexico, USA
Hans de Nivelle	University of Wroclaw, Poland
Nicolas Peltier	CNRS, Grenoble, France
David A. Plaisted	University of North Carolina at Chapel Hill, USA
Christophe Ringeissen	LORIA and INRIA-Lorraine, Nancy, Fance
Albert Rubio	Technical University of Catalonia, Barcelona, Spain
Luca Viganó	Università di Verona, Italy
Jian Zhang	Institute of Software, Chinese Academy of Sciences, Beijing, China

Additional Referees

Jia-Huai You	University of Alberta, Edmonton, Alberta, Canada

Local Organisation Chair

Ullrich Hustadt	University of Liverpool, UK

Steering Committee

Alessandro Armando	Università di Genova, Italy
William McCune	University of New Mexico, Albuquerque, New Mexico, USA
Ingo Dahn	University of Koblenz-Landau, Koblenz, Germany
Ullrich Hustadt	University of Liverpool, UK
Paliath Narendran	University at Albany - SUNY, Albany, New York, USA
Nicolas Peltier	CNRS, Grenoble, France
Silvio Ranise	LORIA and INRIA-Lorraine, France
Stephan Schulz	RISC-Linz, Austria
Gernot Stenz	Munich University of Technology, Germany
Cesare Tinelli	University of Iowa, Iowa City, Iowa, USA
Luca Viganó	Università di Verona, Italy
Laurent Vigneron	LORIA - University Nancy 2, France

Table of Contents

Hierarchical and Modular Reasoning in Complex Theories: The Case of Local Theory Extensions

Viorica Sofronie-Stokkermans

Max-Planck-Institut für Informatik, Campus E1 4, D-66123 Saarbrücken, Germany
sofronie@mpi-inf.mpg.de

Abstract

Many problems in computer science can be reduced to proving the satisfiability of conjunctions of literals w.r.t. a background theory. This can be a concrete theory (e.g. the theory of real or rational numbers), the extension of a theory with additional functions (free, monotone, or recursively defined) or a combination of theories. It is therefore very important to have efficient procedures for checking the satisfiability of conjunctions of ground literals in such theories.

We give an overview of results on hierarchical and modular reasoning in complex theories (cf. e.g. [1,2,3,4]). We show that for a special type of extensions of a base theory, which we call *local*, hierarchical reasoning is possible (i.e. proof tasks in the extension can be hierarchically reduced to proof tasks w.r.t. the base theory). Many theories important for computer science or mathematics fall into this class (examples are theories of data structures, theories of free or monotone functions, functions occurring in mathematical analysis, but also complex extensions, in which various types of functions or data structures are taken into account at the same time). We show how local theory extensions can be identified and under which conditions locality is preserved when combining theories, and we investigate possibilities of efficient reasoning in local theory extensions and combinations. We also present several examples of application domains where local theory extensions occur in a natural way. We show, in particular, that various phenomena analyzed in the verification literature can be explained in a unified way using the notion of locality.

References

1. S. Jacobs, V. Sofronie-Stokkermans. Applications of hierarchical reasoning in the verification of complex systems. *Electronic Notes in Theoretical Computer Science*, 174(8):39–54, 2007.
2. V. Sofronie-Stokkermans. Hierarchic reasoning in local theory extensions. In *20th International Conference on Automated Deduction (CADE-20)*, LNAI 3632, pages 219–234, 2005. Springer.
3. V. Sofronie-Stokkermans. Interpolation in local theory extensions. In *Proc. of the International Joint Conference on Automated Reasoning (IJCAR 2006)*, LNAI 4130, pages 235–250. Springer, 2006.
4. V. Sofronie-Stokkermans, C. Ihlemann. Automated reasoning in some local extensions of ordered structures. Proc. of ISMVL'07, IEEE Press, paper 1, 2007.

Aspects of First-order Reasoning in the KeY System

Martin Giese

Research Institute for Symbolic Computation, Johannes Kepler University
Altenbergerstr. 69, A-4040 Linz, Austria
`martin.giese@risc.uni-linz.ac.at`

Abstract

The deductive program verification system developed as part of the KeY project [4,3,1] is based on a sequent calculus for a certain dynamic logic, which is specially tailored to the verification of Java-Card programs. The sequent calculus symbolically executes programs, until proof obligations in first-order logic are obtained. To simplify reasoning about objects of a Java program, the first-order logic of KeY has strongly typed terms and subtyping [7]. In the context of program verification, it is desirable to have an integrated interactive and automated theorem prover [5], and moreover, an automated proof procedure that does not require backtracking [6]. Theory-specific reasoning in KeY is done by enhancing the prover with theory-specific rules, known as taclets [2], which can easily be formulated in a special rule language. Alternatively, KeY may call various external SMT solvers to handle common datatypes.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.
2. B. Beckert, M. Giese, E. Habermalz, R. Hähnle, A. Roth, P. Rümmer, and S. Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas (RACSAM)*, 98(1):17–53, 2004.
3. B. Beckert, M. Giese, R. Hähnle, V. Klebanov, P. Rümmer, S. Schlager, and P. H. Schmitt. The KeY system 1.0 (deduction component). In *Proceedings of CADE-21, Bremen*, pages 379–384, 2007.
4. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.
5. M. Giese. Integriertes automatisches und interaktives Beweisen: Die Kalkülebene. Diploma Thesis, Fakultät für Informatik, Universität Karlsruhe, June 1998.
6. M. Giese. Incremental closure of free variable tableaux. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. IJCAR, Siena, Italy*, volume 2083 of *LNCS*, pages 545–560. Springer, 2001.
7. M. Giese. A calculus for type predicates and type coercion. In B. Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, Tableaux 2005*, volume 3702 of *LNAI*, pages 123–137. Springer, 2005.

Applying First-Order Theorem Provers in Formal Software Safety Certification

Bernd Fischer

School of Electronics and Computer Science, University of Southampton
Southampton SO17 1BJ, United Kingdom
b.fischer@ecs.soton.ac.uk

Abstract

Formal software safety certification approaches like proof-carrying code use Hoare-style techniques to prove that programs satisfy a variety of safety properties. The properties range from simple language-specific properties like initialization-before-use or array-bounds safety to more complex domain-specific properties as for example frame safety, which is specific to the navigation domain. All of these properties are simpler than full functional correctness, so that the emerging proof obligations are simpler as well, and come within reach of the capabilities of current fully automated first-order theorem provers.

In this talk, I will describe our approach to safety certification of automatically generated code, where we exploit its idiomatic structure to construct the annotations (e.g., loop invariants) necessary for fully automatic proofs. The annotations can be constructed during the code generation process, by embedding annotation templates into the code templates, or during a completely separate post-generation inference phase, where aspect-oriented techniques are used to annotate the crucial code fragments. We have implemented both techniques and integrated them into our AutoBayes and AutoFilter program generation systems; in ongoing work, we apply the inference technique to code generated from Matlab-models. Here I will focus on our experience in integrating first-order theorem provers as "off-the-shelf" components into such systems. I will outline the requirements this application puts on the provers and present the results we achieved with different provers.

Joint work with Ewen Denney and Johann Schumann, RIACS/NASA Ames Research Center.

References

1. E. Denney and B. Fischer. Certifiable program generation. In *Proc. Conf. Generative Programming and Component Engineering (GPCE'05)*, *LNCS 3676*, pp. 17–28. Springer, 2005.
2. E. Denney and B. Fischer. A generic annotation inference algorithm for the safety certification of automatically generated code. In *Proc. Conf. Generative Programming and Component Engineering (GPCE'06)*, pp. 121–130. ACM Press, 2006.
3. E. Denney, B. Fischer, and J. Schumann. An empirical evaluation of automated theorem provers in software certification. *International Journal of AI Tools*, 15(1):81–107, February 2006.

Inductive Proof Search Modulo

Fabrice Nahon[1], Claude Kirchner[2], Hélène Kirchner[2]

[1] LORIA[⋆]
[2] INRIA & LORIA
Nancy, France

Abstract. We present an original narrowing-based proof search method for inductive theorems in equational rewrite theories given by a rewrite system $\mathcal{R}$ and a set E of equalities. It has the specificity to be grounded on deduction modulo and to rely on narrowing to provide both induction variables and instantiation schemas. Whenever the equational rewrite system $(\mathcal{R}, E)$ has good properties of termination, sufficient completeness, and when E is constructor and variable preserving, narrowing at defined-innermost positions leads to consider only unifiers which are constructor substitutions. This is especially interesting for associative and associative-commutative theories for which the general proof search system is refined. The method is shown to be sound and refutationaly complete.

Keywords: Deduction modulo, Noetherian induction, equational rewriting, equational narrowing.

Introduction

Proof by induction is a main reasoning principle and is of prime interest in informatics. Typically in hardware and software verification problems, when dealing with security protocols or safety properties of embedded systems, reasoning on complex data structures with infinite data or states makes a prominent use of induction.

Three main approaches have been developed for mechanizing inductive proofs: (i) explicit induction, used in proof assistants like Nqthm-ACL2 [KM96], Coq[BC04], Isabelle[NPW02] or Inka [AHMS99], (ii) implicit induction by rewriting used in automated theorem provers like RRL [KZ95] or Spike [BKR92] and that should *not* be confused with the third one, (iii) induction by consistency, as clearly emphasized in [Com01, section 1.3] where the interested reader can also find all the relevant references on that last approach. As a bridge between the two first trends, a proof search mechanism for such inductive proofs has been explored in [DKKN03,Dep02,KKN07] relying on the deduction modulo approach [DHK03]. Although already quite expressive, the latter approach is designed for theories expressed as rewrite rules and is thus limited by the fact that axioms like commutativity cannot be oriented as a rule without loosing termination of the underlying rewrite system.

[⋆] UMR 7503 CNRS-INPL-INRIA-Nancy2-UHP

The solution consists then of using equational rewriting (also called rewriting modulo) as pioneered by [PS81] and [JK86] and to extend the proof search method developed in [DKKN03] in order to perform induction in theories containing such non orientable axioms. This extension should also be compared to implicit induction techniques used for induction modulo associativity and commutativity as done in [BBR96] and [Aot06] who generalises [Red90]. The following example is helpful to make this comparison and show how our method is essentially different from the previous ones. Assume that we want to prove

– **Sorts:** nat;
– **constructors:** $0 : \to nat$ $\quad s : nat \to nat$
– **defined functions:** $+ : nat \times nat \to nat$ $\quad * : nat \times nat \to nat$
– **rules**:

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
x+0 & \to & x & x*0 & \to & 0 & exp(x,0) & \to & s(0) \\
x+s(y) & \to & s(x+y) & x*s(y) & \to & x*y+x & exp(x,s(y)) & \to & x*exp(x,y)
\end{array}$$

Fig. 1. Simple arithmetic

the proposition $\forall x, y, n\ exp(x*y, n) \approx exp(x,n)*exp(y,n)$, where $+$ and $*$ are also assumed to be associative and commutative (AC). The method developed in [Ber97] is based on *induction schemes*. More precisely, it computes a subset of variables of the goal, the *induction variables*, and a set of terms, the *test set*. The induction variables are replaced by elements of the test set, and such replacements produce new conjectures which are simplified by rewrite rules of the specification and smaller instances of the original conjecture (the induction hypothesis). The proof is completed when all newly generated conjectures are simplified into known or trivial inductive theorems. Algorithms are provided to compute induction variables and test sets. In the example above, the induction variables are x, y, and n, and the test set is $\{0, s(x)\}$. Therefore, a *test instance* is $exp(s(x')*s(y'), s(n')) \approx exp(s(x'), s(n'))*exp(s(y'), s(n'))$. However, this last equality can be reduced by rules of the specification into $s(x')*s(y')*exp(s(x'+y'+x'*y'), n') \approx exp(s(x'), n')*exp(s(y'), n')$, which cannot be simplified by the induction hypothesis, and the proof attempt may fail. One can avoid this difficulty if the set of induction variables is restricted. That is why [Ber97] have defined an heuristic in order to select *good* induction variables relying on observations of the Nqthm-ACL2 system. Using this strategy in the example above, only the variable n is instantiated and the proof search succeeds. However, the method does not remain refutationaly complete under such an heuristic.

In our approach, the induction step is performed by narrowing at *defined-innermost* positions, when the theory is axiomatized by a sufficiently complete and terminating equational rewrite system. More precisely, it suffices to perform the narrowing step at only one defined-innermost position. In the situation above, these defined innermost positions are 1.1, 2.1 and 2.2. Now, since $*$ is commutative, the goal remains equivalent by permuting the variables x and y, therefore two possibilities remain: narrowing at the defined-innermost position 1.1 where the symbol $*$ occurs, or 2.1 where the symbol *exp* occurs. Considering the latter better, since it further creates more reductions, we choose to narrow at the position 2.1. After normalization, we obtain the trivial subgoal $s(0) \approx s(0)$ and $x * y * exp(x * y, n) \approx x * y * exp(x, n) * exp(y, n)$ which can be reduced by the induction hypothesis.

It is important to emphasize that the latter strategy for selecting one defined-innermost position to perform the narrowing step remains refutationaly complete, whenever the specification has *good* properties: more precisely, this is the case when, given a rewrite system $\mathcal{R}$ and a set E of equalities, the rewrite relation $\mathcal{R}, E$ of Peterson and Stickel [PS81,JK86] is terminating and sufficiently complete modulo E, and when E is *constructor preserving*. Furthermore, under those conditions, narrowing at defined-innermost positions leads to consider only unifiers which are constructor substitutions. Hence, serious difficulties, related to the size of complete sets of unifiers, can be avoided. For instance, it becomes possible to perform induction modulo non finitary theories like associativity.

The paper is structured as follows. Section 1 recalls basic results about rewriting and narrowing, and introduces the concepts of *constructor preserving* theories, *defined-innermost* positions and *complete sets of constructor unifiers* that are used in the following. In Section 2, we explain how deduction modulo manages the Noetherian induction principle and we present the proof search system for inductive proofs modulo a general theory E, which is proved sound and refutationaly complete. Section 3 deals with the special case of associative-commutative theories or associative theories. The proof system of Section 2 is instantiated in these cases with more operational proof steps.

1 Basic ingredients

For the main notations and classical results on term rewriting, we refer for instance to [BN98] or [KK99].

We assume given a many sorted signature $(\mathcal{S}, \Sigma)$ (or simply Σ, for short) where $\mathcal{S}$ is a set of *sorts* and Σ is a set of function symbols, each symbol f given with a rank $f : S_1 \times \ldots \times S_n \to S$, where $S_1, \ldots, S_n, S \in \mathcal{S}$ and n is the arity of f. We assume moreover that the signature Σ comes in two parts, $\mathcal{S} = \mathcal{C} \cup \mathcal{D}$, where $\mathcal{C}$ is a set of *constructor symbols*, and $\mathcal{D}$ is a set of *defined symbols*. A constructor term is a term built only with constructor symbols. Let $\mathcal{X}$ be a family of sorted variables. The set of well-sorted terms over Σ (resp. well-sorted constructor terms) with variables in $\mathcal{X}$ will be denoted by $\mathcal{T}(\Sigma, \mathcal{X})$ (resp. $\mathcal{T}(\mathcal{C}, \mathcal{X})$). The subset of $\mathcal{T}(\Sigma, \mathcal{X})$ (resp. $\mathcal{T}(\mathcal{C}, \mathcal{X})$) of variable-free terms, or *ground* terms, is denoted $\mathcal{T}(\Sigma)$ (resp. $\mathcal{T}(\mathcal{C})$). A term $t \in \mathcal{T}(\Sigma, \mathcal{X})$ is identified as usual to a function from its set of *positions* (strings of positive integers) $\mathcal{D}om(t)$ to symbols of Σ and $\mathcal{X}$. We note ε the empty string (root position).

The *subterm* of t at position ω is denoted by $t_{|\omega}$. The result of replacing $t_{|\omega}$ with s at position ω in t is denoted by $t[s]_\omega$. This notation is also used to indicate that s is a subterm of t and, in this case, the position ω may be omitted. $\mathcal{V}ar(t)$ denotes the set of (free) variables of the term t and $|\mathcal{V}ar(t)|$ its cardinality. We define $\overrightarrow{\mathcal{V}ar(t)}$ as the vector of variables assumed linearly ordered by their name. These notations are extended to equalities $t_1 \approx t_2$ seen as terms with top symbol $\approx$ of arity 2, as well as to rewrite rules. A substitution is a finite mapping $\{x_1 \to t_1, \ldots, x_n \to t_n\}$ where $x_1, \ldots, x_n \in \mathcal{X}$ and $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{X})$. We use postfix notation for substitutions application and composition. The domain of a substitution σ is the set $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid x\sigma \neq x\}$, the set of variables introduced by σ is the set $\mathcal{R}an(\sigma) = \bigcup_{x \in \mathcal{D}om(\sigma)} \mathcal{V}ar(x\sigma)$, and the image of σ is the set $Im(\sigma) = \{t \in \mathcal{T}(\Sigma, \mathcal{X}) \mid \exists x \in \mathcal{D}om(\sigma),\ t = x\sigma\}$. A substitution σ is ground whenever $Im(\sigma) \subseteq \mathcal{T}(\Sigma)$, and is constructor whenever $Im(\sigma) \subseteq \mathcal{T}(\mathcal{C}, \mathcal{X})$. Given two terms s and t, a *unifier* of s and t is a substitution σ such that $s\sigma = t\sigma$, and a *most general unifier* of s and t ($mgu(s,t)$ for short) is a unifier σ such that, for any unifier θ of s and t, there exists a substitution μ such that $\theta = \sigma\mu$ on the variables of s and t.
Given a relation $\to$ on $\mathcal{T}(\Sigma, \mathcal{X})$, $\xrightarrow{+}$ and $\xrightarrow{*}$ denote the transitive and the reflexive transitive closure of $\to$ respectively. A normal form of t, denoted $t\downarrow$, is such that $t \xrightarrow{*} t\downarrow$ and $t\downarrow$ cannot be reduced by the relation $\to$. The normalized form $\sigma\downarrow$ of a substitution σ is defined by $x(\sigma\downarrow) = (x\sigma)\downarrow$ for all $x \in \mathcal{D}om(\sigma)$. An equality is an expression of the form $e_1 \approx e_2$, where e_1 and e_2 are two terms of the same sort. Given a set E of equalities, $=_E$ denotes the congruence generated by E. We always understand equalities in a symmetric way, i.e. we make no difference between $e_1 \approx e_2$ and $e_2 \approx e_1$.
Given two terms s and t, an E-*unifier* of s and t is a substitution σ such that $s\sigma =_E t\sigma$, and a *complete set* of E-unifiers of s and t ($CSU_E(s,t)$ for short) is a set of E-unifiers of s and t satisfying: for any E-unifier θ of s and t, there exists a substitution μ such that $\theta =_E \sigma\mu[\mathcal{V}ar(s) \cup \mathcal{V}ar(t)]$, i.e. $\theta(x) =_E \sigma\mu(x)$ for all $x \in \mathcal{V}ar(s) \cup \mathcal{V}ar(t)$.

Definition 1.1. A set E of equalities is *regular* iff for any equality $e_1 \approx e_2 \in E$, $\mathcal{V}ar(e_1) = \mathcal{V}ar(e_2)$. A set E of equalities is *constructor preserving* whenever E is regular, and, for any equality $e_1 \approx e_2 \in E$, $e_1 \in \mathcal{T}(\mathcal{C}, \mathcal{X}) \Rightarrow e_2 \in \mathcal{T}(\mathcal{C}, \mathcal{X})$.

As a consequence of this definition, a set E of equalities is constructor preserving iff two terms cannot be E-equivalent whenever one of them is constructor and the other is not. Typically, if $+ \in \mathcal{D}$ and $0 \in \mathcal{C}$, $0 + x = x$ (as well as all non-constructor headed collapse axioms) is not constructor preserving (since $0 + 0 = 0$) but associativity or commutativity of $+$ are.

1.1 Equational rewriting and narrowing

We recall some basic notions introduced in [JK86]. A rewrite rule is an ordered pair of terms $l \to r$ such that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ and l is not a variable. A conditional rewrite rule $c \Rightarrow l \rightarrow r$ moreover satisfies $\mathcal{V}ar(c) \subseteq \mathcal{V}ar(l)$. A rewrite system $\mathcal{R}$ is a set of rewrite rules. An *equational rewrite system* is given by a set of rewrite rules $\mathcal{R}$ and a set of equalities E. Let $\rightarrow_{\mathcal{R}/E}$ ($\mathcal{R}/E$ for short) be the relation $=_E \circ \underset{\mathcal{R}}{\rightarrow} \circ =_E$ which simulates the relation induced by $\mathcal{R}$ in E-equivalence classes.

Definition 1.2. An equational rewrite system $(\mathcal{R}, E)$ is *terminating modulo E* iff the relation $\mathcal{R}/E$ is Noetherian, i.e. there is no infinite sequence of the form $t_0 =_E t'_0 \underset{\mathcal{R}}{\rightarrow} t_1 \ldots t_n =_E t'_n \underset{\mathcal{R}}{\rightarrow} t_{n+1} \rightarrow \ldots$. It is *ground terminating modulo E* if it is terminating modulo E over the set of ground terms.

Given an equational rewrite system $(\mathcal{R}, E)$, the rewriting modulo E relation $\rightarrow_{\mathcal{R},E}$ ($\mathcal{R}, E$ for short) and the narrowing modulo E relation $\rightsquigarrow_{\mathcal{R},E}$ are defined as follows:

Definition 1.3. Given two terms $s,\ t \in \mathcal{T}(\Sigma, \mathcal{X})$, *$s$ rewrites modulo E to t*, denoted $s \rightarrow_{\mathcal{R},E} t$, whenever there exist a rewrite rule $l \rightarrow r \in \mathcal{R}$, a position $\omega \in \mathcal{D}om(t)$, and a substitution σ, such that $s_{|\omega} =_E l\sigma$ and $t = s[r\sigma]_\omega$. In this case, s is said $\mathcal{R}, E$-reducible. In addition, for a conditional rule $c \Rightarrow l \rightarrow r$, $c\sigma$ must evaluate to true when applying the rule. Also, *s narrows modulo E into t*, denoted $s \rightsquigarrow_{\mathcal{R},E} t$, whenever there exist a rewrite rule $l \rightarrow r \in \mathcal{R}$, a position $\omega \in \mathcal{D}om(t)$, and a substitution σ, such that $s_{|\omega}\sigma =_E l\sigma$ and $t = (s[r]_\omega)\sigma$.

Since $\rightarrow_{\mathcal{R}} \subseteq \rightarrow_{\mathcal{R},E} \subseteq \rightarrow_{\mathcal{R}/E}$, termination of $\mathcal{R}/E$ implies termination of $\rightarrow_{\mathcal{R}}$ and $\rightarrow_{\mathcal{R},E}$. Sufficient completeness is a fundamental property which states that it is always possible to rewrite any ground non-constructor term into a constructor one:

Definition 1.4. A relation $\rightarrow$ is *sufficiently complete modulo E* when, for any $s \in \mathcal{T}(\Sigma)$, there exists $t \in \mathcal{T}(\mathcal{C})$, such that $s \overset{*}{\rightarrow} t$. The equational rewrite system $(\mathcal{R}, E)$ is *sufficiently complete modulo E* if the relation $\rightarrow_{\mathcal{R},E}$ is.

For ground terminating and sufficiently complete modulo E rewrite systems, it is possible to specify particular positions in terms where reductions must apply, and where case analysis by rewriting can usefully be done.

Definition 1.5. For any $t \in \mathcal{T}(\Sigma, \mathcal{X})$, a position ω in t is called *defined-innermost*, and we denote $\omega \in DI(t)$, if $t(\omega) \in \mathcal{D}$ and $t(\omega') \in \mathcal{C} \cup \mathcal{X}$ whenever $\omega < \omega'$.

For instance, considering the Peano's integers defined in the simple arithmetic example of Fig. 1, in $s((0+0) + s(0 + s(x)))$, the positions 1.1 and 1.2.1 are defined-innermost but 1 is not.
The following proposition states that defined-innermost positions are ground $\mathcal{R}, E$-reducible under appropriate assumptions:

Proposition 1.1. Assume that $(\mathcal{R}, E)$ is sufficiently complete modulo E and that E is constructor preserving. Then, for any term t, for any ground $\mathcal{R}, E$-normalized substitution α, and for any $\omega \in DI(t)$, $t\alpha$ is $\mathcal{R}, E$-reducible at the position ω.

1.2 Constructor E-unifiers

A main difference between previous narrowing or superposition-based approaches and the one proposed in this paper, is that the unification used here to perform narrowing is quite restricted. For instance, when reasoning modulo associativity, instead considering potentialy infinite sets of unifiers, we can safely restrict to finitely many ones.

For a given set E of equalities, *constructor E-unifiers* are a key to tame the proof search system IndNarrowModE presented below. *Complete sets of constructor E-unifiers* are generating sets of constructor unifiers:

Definition 1.6. Let $s,\ t \in \mathcal{T}(\Sigma, \mathcal{X})$, a substitution σ is a constructor E-unifier of s and t if $s\sigma =_E t\sigma$ and $Im(\sigma) \subseteq \mathcal{T}(\mathcal{C}, \mathcal{X})$. Given two terms $s, t \in \mathcal{T}(\Sigma, \mathcal{X})$, $CSUC_E(s,t)$ is a *complete set of constructor E-unifiers* of s and t, if:

Correctness: every σ of $CSUC_E(s,t)$ is a constructor E-unifier of s and t;
Completeness: for any constructor E-unifier of s and t, there exist $\sigma \in CSUC_E(s,t)$ and a substitution μ, such that $\theta =_E \sigma\mu\ [\mathcal{V}ar(s) \cup \mathcal{V}ar(t)]$;
Domain: for any $\sigma \in CSUC_E(s,t)$, $\mathcal{R}an(\sigma) \cap \mathcal{D}om(\sigma) = \emptyset$.

If E is constructor preserving and satisfy syntactic conditions detailled in [Nah07], the subset of all constructor elements of $CSU_E(s,t)$ is a complete set of constructor E-unifiers of s and t. This is in particular the case when considering AC of A theories involving only defined symbols. More precisely, when E is an AC theory involving only defined symbols, if s and t are terms and ω is a defined-innermost position in s, then $CSUC_E(s_{|\omega}, t)$ is $CSUC_F(s_{|\omega}, t)$, where F denotes the subset of commutativity axioms of E. In other words, in this case AC constructor unification reduces to C constructor unification. Similarly if E is an associative theory involving only defined symbols, although $CSU_E(s_{|\omega}, t)$ may be infinite, $CSUC_E(s_{|\omega}, t)$ is $CSUC_\emptyset(s_{|\omega}, t)$ which of course simplifies considerably the induced proof space.

To conclude this section, the following proposition shows that, whenever E is constructor preserving and $(\mathcal{R}, E)$ is sufficiently complete modulo E, the narrowing step at defined-innermost positions is performed with constructor substitutions:

Proposition 1.2. Assume that $(\mathcal{R}, E)$ is sufficiently complete modulo E and that E is constructor preserving. Then, for all $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$, for any $f \in \mathcal{D}$, for any ground $\mathcal{R}, E$-irreducible instantiation α of $f(t_1, \ldots, t_n)$, and for any set V such that $\mathcal{D}om(\alpha) \subseteq V$, there exist a rewrite rule $l \to r \in \mathcal{R}$, a substitution $\sigma \in CSUC_E(f(t_1, \ldots, t_n), l)$ and a substitution μ such that: $\sigma\mu =_E \alpha\ \ [V]$.

Thanks to these settings, we now present an inductive proof search system, relying on a main induction rule that uses narrowing to choose both the induction variables and the instantiation schema.

2 A proof search system for induction modulo

The proof search system IndNarrowModE for inductive proofs introduced in this section is based on (restricted) narrowing and rewriting. The main rule, called **Induce**, performs the induction step. Its intuition is the following: in order to apply the induction hypothesis, one should decrease the size of the goal by rewriting it using a noetherian rewrite system. Whenever the goal does not rewrite, it should be first instantiated to be then rewritten, i.e. it should be narrowed. By expressing this in the sequent calculus modulo, we provide an explicit and constructive bridge between the rewrite-based implicit and explicit approaches of induction.

2.1 The proof search system IndNarrowModE

Let $<$ be a Noetherian order on a set τ, i.e. such that there is no infinite sequence of elements of the form $a_0 > a_1 > \ldots > a_n > \ldots$. The *Noetherian induction principle* states that a proposition P holds for any element x of τ if P holds for all b in τ with $b < x$. Formally, if $\forall x\, x \in \tau \wedge (\forall b\, b \in \tau \wedge b < x \Rightarrow P(b)) \Rightarrow P(x)$, then P holds for all x in τ. Hence, if we write $Noeth(<, \tau)$ to state that $<$ is a Noetherian relation over τ, and $NoethInd(P, <, \tau)$ the proposition above, the Noetherian induction principle is the right-hand side of the following implication:

$$NI: \quad \forall < \forall \tau\, [Noeth(<, \tau) \Rightarrow \forall P\, (NoethInd(P, <, \tau) \Rightarrow \forall x\, P(x))]$$

To emphasize the order condition $b < x$, and since b is universally quantified, we rename b into $\underline{x}$. From now on, we instantiate τ by the set of ground terms $\mathcal{T}(\Sigma)$, P by an equality predicate $\approx$ and $<$ by the proper part of a quasi ordering $\leqslant$ defined on the set of terms $\mathcal{T}(\Sigma, \mathcal{X})$. The induction hypothesis becomes therefore $\forall \underline{x}(\underline{x} < x \Rightarrow s(\underline{x}) \approx t(\underline{x}))$, with x any variable of $s \approx t$. It is also possible to define an induction hypothesis with respect to *all* variables of $s \approx t$. Let $\overrightarrow{x} \in \mathcal{X}^n$ denote the vector of variables of $s \approx t$. In order to compare n-tuples of terms, we use the standard extension on the Cartesian product $\leqslant_n$ of $\leqslant$: $\forall \overrightarrow{u}, \overrightarrow{v} \in \mathcal{T}(\Sigma, \mathcal{X})^n \quad \overrightarrow{u} \leqslant_n \overrightarrow{v} \Leftrightarrow (\forall i\; 1 \leq i \leq n \Rightarrow u_i \leqslant v_i)$. In which case the induction hypothesis becomes:

$$\mathcal{RE}_{ind}(s \approx t, <_n, \mathcal{T}(\Sigma)^n): \quad (\underline{\overrightarrow{x}} \in \mathcal{T}(\Sigma)^n \wedge \underline{\overrightarrow{x}} <_n \overrightarrow{x}) \Rightarrow s(\underline{\overrightarrow{x}}) \approx t(\underline{\overrightarrow{x}})$$

and $\overrightarrow{x}$ is therefore the vector of free variables of $\mathcal{RE}_{ind}(s \approx t, <_n, \mathcal{T}(\Sigma)^n)$. In order to simplify the notations, and when no confusion can occur, we denote it simply $\mathcal{RE}_{ind}(s \approx t, <)$. The following notation, where σ is any substitution, will also be used:

$$\mathcal{RE}_{ind}(s \approx t, <)\sigma: \quad (\underline{\overrightarrow{x}} \in \mathcal{T}(\Sigma)^n \wedge \underline{\overrightarrow{x}} <_n \overrightarrow{x}\sigma) \Rightarrow s(\underline{\overrightarrow{x}}) \approx t(\underline{\overrightarrow{x}})$$

There is no space here to detail how an inductive proof (that requires at least second-order logic) can be formalized in $HOL_{\lambda\sigma}$[DHK01]. This is in particular detailled in [Dep02] whose main idea relies on deduction modulo [DHK03], where the computational and deduction parts of a proof, as well as their interactions, are identified as such. In first-order logic, for example based on the sequent calculus, a congruence on *propositions* models computation and often consists of a confluent term rewrite system, rewriting terms to terms and atomic propositions to propositions. For instance, modulo such a congruence $\sim$, the right rule for the conjunction in sequent calculus modulo is written:

$$\frac{\Gamma \vdash_{\sim} A, \Delta \quad \Gamma \vdash_{\sim} B, \Delta}{\Gamma \vdash_{\sim} D, \Delta} \text{ if } D \sim A \wedge B.$$

In order to provide the notational support for expressing our proof search methodology, this is further refined by writing the sequents $\Gamma_1 | \Gamma_2 \vdash_{\mathcal{RE}_1 | \mathcal{RE}_2} Q$, where Γ_1 is the deductive part of the user definitions, $\mathcal{RE}_1$ is their computational part; Γ_2 is the deductive part for other statements, $\mathcal{RE}_2$ is their computational part; Q is an equational goal. The distinction between $\Gamma_1, \mathcal{RE}_1$ and $\Gamma_2, \mathcal{RE}_2$ is needed because only $\mathcal{RE}_1$ will be used for narrowing. For simplicity, we assume that $\mathcal{RE}_1$ contains only unconditional rules or equalities, and we assume from now on, that Γ_1 contains a constructor preserving theory E, such that $(\mathcal{RE}_1, E)$ is terminating and sufficiently complete modulo E. Γ_2 is initialized with the proposition NI defined above, with the theory of equality $Th_{\approx}$ satisfied by the binary relation $\approx$, and, if the goal and the rules in $\mathcal{RE}_2$ contain n free variables, with the proposition $Noeth(<_n, \mathcal{T}(\Sigma)^n)$, and may contain other lemmas. $\mathcal{RE}_2$ will receive the induction hypotheses provided by application of the proof search rules, so $\mathcal{RE}_2$ may contain conditional equalities.

Example 2.1. Assume that $\mathcal{RE}_1$ contains the rules of simple arithmetic given in Figure 1. $\mathcal{RE}_1$ is terminating and sufficiently complete modulo associativity and commutativity of the $*$ and $+$ operators (denoted $AC(+,*)$). Let $\Gamma_1 = AC(+,*)$, $\Gamma_2 = Th_{\approx} \cup \{NI,\ Noeth(<_4, \mathcal{T}(\Sigma)^4)\}$, and $Q = (x_1 + x_2 + x_3) * x_4 \approx x_1 * x_4 + x_2 * x_4 + x_3 * x_4$. Then, we can consider the goal $\Gamma_1 | \Gamma_2 \vdash_{\mathcal{RE}_1 | \emptyset} Q$.

The proof search rules are presented in Figure 2.

Sequents are gathered in a multiset structure modeled with the $\bullet$ operator that is an AC operator on sequents with $\diamond$ as neutral element.

The rule **Induce** performs the induction step. It uses narrowing to choose both the induction variable(s) and the instantiation schema. Narrowing is applied only at defined innermost positions $DI(Q')$ of a goal Q' E-equivalent to the current goal Q. Indeed Q' may be Q itself, and this will be the case for the derived inference systems where E is A or AC.

The other rules are doing the following: **Trivial** eliminates a trivial equation, **Rewrite** (1 or 2) rewrites using a rule, an equation, or a smaller instance of a previous goal. **Rewrite** is duplicated because of the $\Gamma_1, \mathcal{RE}_1$ and $\Gamma_2, \mathcal{RE}_2$ distinction.

This inference rule set is generic and prepares to more operational versions tailored for AC and A-theories.

Induce $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q \rightarrowtail$

$\bullet_{\substack{l \to r \in \mathcal{RE}_1 \\ \sigma' \in CSUC_E(Q'_{|\omega'}, l)}} \quad \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2\sigma' \cup \{\mathcal{RE}_{ind}(Q,<)\sigma'\}} (Q'[r]_{\omega'})\sigma'$

if $Q' =_E Q$ and $\omega' \in DI(Q')$

Rewrite$_1$ $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q \rightarrowtail \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q'$

if $Q \to_{\mathcal{RE}_1/E} Q'$

Rewrite$_2$ $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q \rightarrowtail \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q'$

if $Q \to_{\mathcal{RE}_2/E} Q'$

Trivial $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} t \approx t' \rightarrowtail \diamond$

if $t =_E t'$

Refutation $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q \rightarrowtail$ Refutation

when no other rules can be applied

Fig. 2. The proof search system IndNarrowModE

2.2 Properties of IndNarrowModE

From now on, let us assume that $(\mathcal{R}, E)$ is terminating and sufficiently complete modulo E, and that E is constructor preserving.

Soundness: Proving soundness amounts showing that for each rule of the proof search system IndNarrowModE of the form $S \rightarrowtail S'$, if S' is derivable in the sequent calculus modulo, then one can also build a proof of S. The main delicate point is to prove this result for the **Induce** rule, as stated in the next theorem.

Theorem 2.1. *If the sequent* $\Gamma_1|\Gamma_2, \overrightarrow{x_{\sigma'}} \in \mathcal{T}(\Sigma)^{n_{\sigma'}} \vdash_{\mathcal{RE}_1|\mathcal{RE}_2\sigma' \cup \{\mathcal{RE}_{ind}(Q)\sigma'\}} (Q'[r]_{\omega'})\sigma'$ *is derivable in the sequent calculus modulo, where:*

1. $Q =_E Q'$ *and* $\omega' \in DI(Q')$*;*
2. $l \to r \in \mathcal{RE}_1$ *and* $\sigma' \in CSUC_E(Q'_{|\omega'}, l)$*;*
3. $\mathcal{RE}_2\sigma'$ *is the rewrite system obtained by the replacement of each free variable* x *of any rewrite rule in* $\mathcal{RE}_2$ *by a corresponding* $x\sigma'$*;*
4. $\overrightarrow{x_{\sigma'}} \in \mathcal{X}^{n_{\sigma'}}$ *is the vector of free variables of* $\mathcal{RE}_2\sigma' \cup \{Q\sigma'\}$*;*

then, one can build a proof in the sequent calculus modulo of $\Gamma, \overrightarrow{x} \in \mathcal{T}(\Sigma)^n \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q$ *where* $\overrightarrow{x} \in \mathcal{X}^n$ *denotes the vector of free variables of* $\mathcal{RE}_2 \cup \{Q\}$*.*

Refutational correctness: Proving refutational correctness amounts showing that for each rule of the proof search system IndNarrowModE of the form $S \rightarrowtail S'$, if S is derivable in the sequent calculus modulo, then one can also build a proof of S'. Again the main delicate point is for the **Induce** rule, and is stated as follows.

Theorem 2.2. *If the sequent* $\Gamma_1|\Gamma_2, \overrightarrow{x} \in \mathcal{T}(\Sigma)^n \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q$ *where* $\overrightarrow{x} \in \mathcal{X}^n$ *is the vector of free variables of* $\mathcal{RE}_2 \cup \{Q\}$*, admits a proof in the sequent calculus modulo, then one can build a proof of:*

$$\Gamma_1|\Gamma_2, \overrightarrow{x_{\sigma'}} \in \mathcal{T}(\Sigma)^{n_{\sigma'}} \vdash_{\mathcal{RE}_1|\mathcal{RE}_2\sigma' \cup \{\mathcal{RE}_{ind}(Q,<)\sigma'\}} (Q'[r]_{\omega'})\sigma'$$

where $Q' =_E Q$*,* $l \to r \in \mathcal{RE}_1$*,* $\omega' \in DI(Q')$*,* $\sigma' \in CSUC_E(Q'_{|\omega'}, l)$*, and* $\overrightarrow{x_{\sigma'}} \in \mathcal{X}^{n_{\sigma'}}$ *is the vector of free variables of* $\mathcal{RE}_2\sigma' \cup \{Q\sigma'\}$*.*

Refutational completeness: Proving refutational completeness is achieved thanks to the **Refutation** rule which applies when no other rule of IndNarrow can be applied.

Theorem 2.3. *If* $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q \overset{*}{\rightarrowtail}$ *Refutation then the sequent* $\Gamma_1|\Gamma_2, \overrightarrow{x} \in \mathcal{T}(\Sigma)^n \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q$ *has no proof in the sequent calculus modulo.*

3 Induction modulo AC and A

The general IndNarrowModE proof search system is indeed working directly on equivalence classes modulo E, a situation not directly implementable for most theories E. To focus on more operational proof search systems where instead of working with $\to_{R/E}$, we use the operational rewrite relation $\to_{R,E}$, we focus in this section on the case of associative-commutative or associative theories. We introduce two proof search systems IndNarrowModAC and IndNarrowModA as special instances of IndNarrowModE with specific improvements and illustrating examples. Soundness and refutational correctness and completeness of these systems will be consequences of the properties of IndNarrowModE.

3.1 More about flattened terms

In associative and associative-commutative theories, equivalence classes of terms are often represented by flattened terms. We refer for the basic definitions and results about positions and subterms to [Mar93]. Intuitively flattening a term amounts to recursively replace $f(f(s,t),u)$ or $f(s,f(t,u))$ by $f(s,t,u)$ if f is an associative symbol. This is the key point bridging the proof search systems IndNarrowModAC and IndNarrowModA on the one hand, and IndNarrowModE on the other hand.

From now on, we assume that some function symbols in a subset $\mathcal{V}$ of Σ may have an *unbounded arity*. Let $A(\mathcal{V}) = \{f(fxy)z \approx fx(fyz) \mid f \in \mathcal{V}\}$ and $AC(\mathcal{V}) = A(\mathcal{V}) \cup \{fxy \approx fyx \mid f \in \mathcal{V}\}$. In the following, we assume that all symbols in $\mathcal{V}$ are defined symbols, that constructor symbols do *not* have unbounded arities, and that $+$ and $*$ denote symbols with unbounded arity.

[Mar93] defines a transformation which associates to each position in a given term t a position in the flattening $\overline{t}$ of t, also called the *flattening* of this position. However, a position in $\overline{t}$ is not always the flattening of some position in t, and this led us to introduce the following definition:

Definition 3.1. For a flattened term $\overline{s}$, a position $\omega \in \mathcal{D}om(\overline{s})$ is *flattened* if there exist $i, k \in \mathbb{N}$, and a word ω_0, s.t. $\omega = \omega_0.i$ or $\omega = \omega_0.\{i, i+1, \ldots, i+k\}$.

The above flattened positions are precisely the flattening of positions [Nah07]. To define a rewrite relation on the set of flattened terms, the notion of replacement has to be generalized:

Definition 3.2. Given two flattened terms $\overline{s} = f\overline{s_1}\ldots\overline{s_n}$, $\overline{t}$, and a position $\omega \in \overline{s}$, the replacement by $\overline{t}$ in $\overline{s}$ at the position ω is inductively defined by:

- $\overline{s}[\overline{t}]_\varepsilon = \overline{t}$
- If $\omega \in \{1, \ldots, n\}$
  - Case 1: there exist $i,\ k \in \mathbb{N}$, such that $\omega = \{i, i+1, \ldots, i+k\}$.
    $\overline{s}[\overline{t}]_\omega = \overline{f\overline{s_1}\ldots\overline{s_{i-1}}\ \overline{t}\ \overline{s_{i+k+1}}\ldots\overline{s_n}}$
  - Case 2: otherwise, let $\{i_1, \ldots, i_k\} = \{1, \ldots, n\} - \omega$.
    $\overline{s}[\overline{t}]_\omega = \overline{f\overline{s_{i_1}}\ldots\overline{s_{i_k}}\ \overline{t}}$.
- $\overline{s}[\overline{t}]_{i.\omega_i} = \overline{f\overline{s_1}\ldots\overline{s_i}[\omega_i \leftarrow \overline{t}]\ldots\overline{s_n}}$.

Now, we introduce a rewrite relation on the set of flattened terms as follows:

Definition 3.3. Given a rewrite system $\mathcal{R}$, we define the relation $\rightarrow_{\overline{\mathcal{R}}}$ on the set of flattened terms by $\overline{s} \rightarrow_{\overline{\mathcal{R}}} \overline{t}$ whenever there exist a rule $c \Rightarrow l \rightarrow r \in \mathcal{R}$, a flattened position $\omega \in \mathcal{D}om(\overline{s})$ and a substitution σ such that:

- $\overline{s}_{|\omega} = \overline{l\sigma}$, $\overline{t} = \overline{s}[\overline{r\sigma}]_\omega$
- and the condition $c\sigma$ is true.

If $\equiv_p$ denotes the classical equivalence induced on the set of flattened terms by permutation of the arguments of symbols in $\mathcal{V}$, we consider the extension $\overline{\mathcal{R}}/\equiv_p$ of $\overline{\mathcal{R}}$ on the set of $\equiv_p$-equivalences. As previously, in order to perform induction by narrowing at *defined-innermost* positions, we must define such positions for flattened terms:

Definition 3.4. For any $s \in \mathcal{T}(\Sigma, \mathcal{X})$, and for any $\omega \in \mathcal{D}om(\overline{s})$, the position ω is called *defined-innermost* whenever there exist $f \in \Sigma$ and terms $s_1, \ldots,\ s_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$, such that $\overline{s}_{|\omega} = f\overline{s_1}\ldots\overline{s_n}$, and moreover $n = 2$ if $f \in \mathcal{V}$.

Intuitively, the position ω in $\overline{s}$ is defined-innermost when $\overline{s}_{|\omega}$ coincides with its flattened form.

InduceAC $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q} \rightarrowtail$

$$\bullet_{\substack{l \to r \in \mathcal{RE}_1 \\ \sigma \in CSUC_{AC}(\overline{Q}_{|\omega},\, l)}} \quad \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2\sigma \cup \{\mathcal{RE}_{ind}(Q,<)\sigma\}} (\overline{Q}[\overline{r}]_\omega)\sigma$$

if $\omega \in DI(\overline{Q})$

Rewrite$_1$AC $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q} \rightarrowtail \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q'}$

if $\overline{Q} \to_{\overline{\mathcal{RE}_1}/\equiv_p} \overline{Q'}$

Rewrite$_2$AC $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q} \rightarrowtail \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q'}$

if $\overline{Q} \to_{\overline{\mathcal{RE}_2}/\equiv_p} \overline{Q'}$

TrivialAC $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{t} \approx \overline{t'} \rightarrowtail \diamond$

if $\overline{t} \equiv_p \overline{t'}$

RefutationAC $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q} \rightarrowtail$ Refutation

when no other rules can be applied

Fig. 3. The proof search system IndNarrowModAC

InduceA $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q} \rightarrowtail$

$$\bullet_{\substack{l \to r \in \mathcal{RE}_1 \\ \sigma \in CSUC_{A}(\overline{Q}_{|\omega},\, l)}} \quad \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2\sigma \cup \{\mathcal{RE}_{ind}(Q,<)\sigma\}} (\overline{Q}[\overline{r}]_\omega)\sigma$$

if $\omega \in DI(\overline{Q})$ and ω flattened.

Rewrite$_1$A $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q} \rightarrowtail \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q'}$

if $\overline{Q} \to_{\overline{\mathcal{RE}_1}} \overline{Q'}$

Rewrite$_2$A $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q} \rightarrowtail \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q'}$

if $\overline{Q} \to_{\overline{\mathcal{RE}_2}} \overline{Q'}$

TrivialA $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{t} \approx \overline{t'} \rightarrowtail \diamond$

if $\overline{t} \neq \overline{t'}$

RefutationA $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q} \rightarrowtail$ Refutation

when no other rules can be applied

Fig. 4. The proof search system IndNarrowModA

3.2 The proof search systems IndNarrowModAC and IndNarrowModA

The specific proof search systems IndNarrowModAC and IndNarrowModA are respectively given in Figure 3 and Figure 4.

Soundness, refutational correctness and completeness of IndNarrowModAC and IndNarrowModA are consequences of the following proposition that states a correspondence between a deduction on a goal Q using IndNarrowModE and a deduction on the corresponding flattened goal using IndNarrowModAC or IndNarrowModA.

Theorem 3.1. *Let $E = AC(\mathcal{V})$ (resp.$E = A(\mathcal{V})$).*

1. *If $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q \rightarrowtail_{\mathsf{IndNarrowModE}} \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE'}_2} R$, then $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q} \rightarrowtail_{\mathsf{IndNarrowModAC}} \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE'}_2} \overline{R}$. (resp. $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q} \rightarrowtail_{\mathsf{IndNarrowModA}} \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE'}_2} \overline{R}$).*

2. *If $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q} \rightarrowtail_{\mathsf{IndNarrowModAC}} \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE'}_2} \overline{R}$ (resp. $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} \overline{Q} \rightarrowtail_{\mathsf{IndNarrowModA}} \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE'}_2} \overline{R}$), there exists R' such that $R' =_{AC} R$ (resp. $R' =_A R$), and $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q \rightarrowtail_{\mathsf{IndNarrowModE}} \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE'}_2} R'$*

3.3 Two simple examples

In order to get a better intuition on the way these sets of rules are working, let us look at two examples. In the following, we always refer to the specification and the set of rewrite rules given in Figure 1. The first example of proof uses AC properties of $+$ and $*$ induction modulo AC, and the second one uses the same rules but just associativity of these two symbols.

An $AC-$example: In the context of Example 2.1, let us consider the following sequent: $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\emptyset} Q$ and first apply the rule **InduceAC**. The innermost positions in Q are $1.1.\{1,2\}$, $1.1.\{1,3\}$, $1.1.\{2,3\}$, 2.1, 2.2 and 2.3. Since the goal remains equivalent by permutation of the variables x_1, x_2 and x_3, only two possibilities remain: narrowing at a position where the symbol $+$ occurs, or where the symbol $*$ occurs. Since the last choice creates more reductions than the first one, we arbitrarily choose to narrow at the position 2.1 of the goal. Therefore, we must compute the set $CSUC_{AC}(x_1 * x_4, l)$ for any rewrite rule $l \rightarrow r$ of $\mathcal{RE}_1$. This restricts to rules such that $l(\varepsilon) = *$, and we obtain:

l	$CSUC_{AC}(x_1 * x_4, l)$
$x * 0$	$\sigma_1 = \{x_1 \rightarrow y_1; x \rightarrow y_1; x_4 \rightarrow 0\}$ $\sigma_2 = \{x_1 \rightarrow 0; x \rightarrow y_4; x_4 \rightarrow y_4\}$
$x * s(y)$	$\sigma_3 = \{x_1 \rightarrow y_1; x \rightarrow y_1; y \rightarrow y_4; x_4 \rightarrow s(y_4)\}$ $\sigma_4 = \{x_1 \rightarrow s(y_1); x \rightarrow y_4; y \rightarrow y_1; x_4 \rightarrow y_4\}$

After normalization, this leads us to prove the four sequents:

$\Gamma_1\|\Gamma_2 \vdash_{\mathcal{RE}_1\|\mathcal{RE}_{ind}(Q)\sigma_1}$	$0 \approx 0$
$\Gamma_1\|\Gamma_2 \vdash_{\mathcal{RE}_1\|\mathcal{RE}_{ind}(Q)\sigma_2}$	$(x_2 + x_3) * y_4 \approx x_2 * y_4 + x_3 * y_4$
$\Gamma_1\|\Gamma_2 \vdash_{\mathcal{RE}_1\|\mathcal{RE}_{ind}(Q)\sigma_3}$	$(y_1 + x_2 + x_3) * y_4 + y_1 + x_2 + x_3$ $\approx y_1 * y_4 + y_1 + x_2 * y_4 + x_2 + x_3 * y_4 + x_3$
$\Gamma_1\|\Gamma_2 \vdash_{\mathcal{RE}_1\|\mathcal{RE}_{ind}(Q)\sigma_4}$	$(y_1 + x_2 + x_3) * y_4 + y_4 \approx y_1 * y_4 + y_4 + x_2 * y_4 + x_3 * y_4$

Trivial gets rid of the first one. Since $(y_1, x_2, x_3, y_4) <_4 (y_1, x_2, x_3, s(y_4))$, **Rewrite**$_2$ can be applied on the third one, and since $(y_1, x_2, x_3, y_4) <_4 (s(y_1), x_2, x_3, y_4)$, **Rewrite**$_2$ can be applied on the fourth one. Hence we get:

$\Gamma_1 \| \Gamma_2 \vdash_{\mathcal{RE}_1 \| \mathcal{RE}_{ind}(Q)\sigma_2}$	$(x_2 + x_3) * y_4 \approx x_2 * y_4 + x_3 * y_4$
$\Gamma_1 \| \Gamma_2 \vdash_{\mathcal{RE}_1 \| \mathcal{RE}_{ind}(Q)\sigma_3}$	$y_1 * y_4 + x_2 * y_4 + x_3 * y_4 + y_1 + x_2 + x_3 \approx y_1 * y_4 + y_1 + x_2 * y_4 + x_2 + x_3 * y_4 + x_3$
$\Gamma_1 \| \Gamma_2 \vdash_{\mathcal{RE}_1 \| \mathcal{RE}_{ind}(Q)\sigma_4}$	$y_1 * y_4 + x_2 * y_4 + x_3 * y_4 + y_4 \approx y_1 * y_4 + y_4 + x_2 * y_4 + x_3 * y_4$

Trivial gets rid of the two last subgoals. The application of **Induce** to the first one at position 2.1 generates four subgoals. **Trivial** gets rid of the two first ones, the application of **Rewrite**$_2$ to the last ones creates two new subgoals which are trivial and we are done.

An A-example: Assume that $\mathcal{RE}_1$ contains the rules of simple arithmetic given in Figure 1. $\mathcal{RE}_1$ is terminating and sufficiently complete modulo associativity of the $*$ and $+$ operators (denoted $A(+,*)$) Let us prove that distributivity of $*$ over $+$ is an inductive theorem.
Let $\Gamma_1 = A(+,*)$, $\Gamma_2 = Th_{\approx} \cup \{NI,\ Noeth(<_3, \mathcal{T}(\Sigma)^3)\}$, and $Q = x_1 * (x_2 + x_3) \approx x_1 * x_2 + x_1 * x_3$. Let us start from the sequent: $\Gamma_1 | \Gamma_2 \vdash_{\mathcal{RE}_1 | \emptyset} Q$.
We can apply **InduceA** at the innermost positions 1.2, 2.1 and 2.2 in Q and Theorem 2.1 ensures that each of these choices is correct. Since narrowing at position 2.1 creates less further reductions than the ones at positions 1.2 or 2.2, we choose to narrow at the position 2.2 of the goal. Thus, we need to compute $CSUC_A(x_1 * x_3, l)$ for any rewrite rule $l \to r$ of $\mathcal{RE}_1$. This restricts to rules such that $l(\varepsilon) = *$, and we obtain:

l	$CSUC_A(x_1 * x_3, l)$
$x * 0$	$\sigma_1 = \{x_1 \to y_1; x \to y_1; x_3 \to 0\}$
$x * s(y)$	$\sigma_2 = \{x_1 \to y_1; x \to y_1; y \to y_3; x_3 \to s(y_3)\}$

After normalization, we obtain the subgoals:

$\Gamma_1 \| \Gamma_2 \vdash_{\mathcal{RE}_1 \| \mathcal{RE}_{ind}(Q)\sigma_1} y_1 * x_2 \approx y_1 * x_2$
$\Gamma_1 \| \Gamma_2 \vdash_{\mathcal{RE}_1 \| \mathcal{RE}_{ind}(Q)\sigma_2} y_1 * (x_2 + y_3) + y_1 \approx y_1 * x_2 + y_1 * y_3 + y_1$

Trivial gets rid of the first subgoal. Since $(y_1, x_2, y_3) <_3 (y_1, x_2, s(y_3))$, **Rewrite**$_2$**A** can be applied on the second one. Hence, we get:

$$\boxed{\Gamma_1 | \Gamma_2 \vdash_{\mathcal{RE}_1 | \mathcal{RE}_{ind}(Q)\sigma_2} y_1 * x_2 + y_1 * y_3 + y_1 \approx y_1 * x_2 + y_1 * y_3 + y_1}$$

and **Trivial** gets rid of this last subgoal.

4 Conclusion

We have extended the inductive proof search method based on narrowing to the case where theories contain non-orientable axioms. The main inference rule is based on a restricted application of narrowing at defined-innermost positions and with a restricted notion of equational unifiers based only on constructors. This general approach is proved correct and refutationaly complete. We then applied it to the specific case of rewriting modulo AC or A axioms and show on two examples how the method safely restricts the proof search space. This provides a significant improvement on the current inductive proof search approaches.

An interesting side result of our approach is the introduction of a new kind of E-unifiers that we called constructor E-unifiers. In the case of associative and associative commutative theories E, they have the nice property to considerably reduce the number of unifiers to be considered in a complete set of unifiers that may be huge or even infinite in these theories. A natural and challenging question is to build a unification theory for these specific unifiers.

First motivated by the wish to provide a bridge between explicit and implicit induction, our approach achieves this goal through a specific instance of the sequent calculus modulo [DHK01] that clarifies the respective roles and uses of the noetherian induction principle and of equational rewriting. As a consequence, we plan to have an automated construction of such proofs into the sequent calculus for insertion into proof assistants like lemuridæ which is based on superdeduction [BHK07]. Such proof assistants will therefore rely on an implementation of our inference systems, a task that still remains to be done.

References

AHMS99. S. Autexier, D. Hutter, H. Mantel, and A. Schairer. System description: Inka 5.0 – a logic voyager. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *LNAI*, pages 207–211, Trento, Italy, july 1999. Springer.

Aot06. T. Aoto. Dealing with Non-orientable Equations in Rewriting Induction. In F. Pfenning, editor, *Proceedings of the 17th International Conference on Rewriting Techniques and Applications*, volume 4098 of *Lecture Notes in Computer Science*, pages 242–256, Nara (Japan), apr 2006. Springer.

BBR96. N. Berregeb, A. Bouhoula, and M. Rusinowitch. Automated verification by induction with associative-commutative operators. In R. Alur and T. A. Henzinger, editors, *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 220–231. Springer, 1996.

BC04. Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

Ber97. N. Berregeb. *Preuves par induction implicite : cas des théories associatives-commutatives et observationnelles*. Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, June 1997.

BHK07. P. Brauner, C. Houtmann, and C. Kirchner. Principle of superdeduction. In L. Ong, editor, *Proceedings of LICS*, pages 41–50, jul 2007.

BKR92. A. Bouhoula, E. Kounalis, and M. Rusinowitch. Spike: An automatic theorem prover. In *Proceedings of the 1st International Conference on Logic Programming and Automated Reasoning, St. Petersburg (Russia)*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 460–462. Springer-Verlag, July 1992.

BN98. F. Baader and T. Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

Com01. H. Comon. Inductionless induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 14, pages 914–959. Elsevier Science, 2001.

Dep02. E. Deplagne. *Système de preuve modulo récurrence.* Thèse de doctorat, Université Nancy 1, November 2002.

DHK01. G. Dowek, T. Hardin, and C. Kirchner. HOL-$\lambda\sigma$ an intentional first-order expression of higher-order logic. *Mathematical Structures in Computer Science*, 11(1):21–45, 2001.

DHK03. G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, Nov 2003.

DKKN03. E. Deplagne, C. Kirchner, H. Kirchner, and Q.-H. Nguyen. Proof search and proof check for equational and inductive theorems. In F. Baader, editor, *Proceedings of CADE-19*, Miami, Florida, July 2003. Springer-Verlag.

JK86. J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986.

KK99. C. Kirchner and H. Kirchner. Rewriting, solving, proving. A preliminary version of a book available at www.loria.fr/~ckirchne/rsp.ps.gz, 1999.

KKN07. C. Kirchner, H. Kirchner, and F. Nahon. Narrowing based inductive proof search: Definition and optimisations. Research report, LORIA, Mars 2007.

KM96. M. Kaufmann and J. S. Moore. ACL2: An industrial strength version of nqthm. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

KZ95. D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *J. Computer and Mathematics with Applications*, 29(2):91–114, 1995.

Mar93. C. Marché. *Réécriture modulo une théorie présentée par un système convergent et décidabilité du problème du mot dans certaines classes de théories équationnelles.* Thèse de Doctorat d'Université, Université de Paris-Sud, Orsay (France), October 1993.

Nah07. F. Nahon. *Preuves par induction dans le calcul des séquents modulo.* PhD thesis, UHP, to appear, October 2007.

NPW02. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

PS81. G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:233–264, 1981.

Red90. U. Reddy. Term rewriting induction. In M. E. Stickel, editor, *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, volume 449 of *Lecture Notes in Computer Science*, pages 162–177. Springer-Verlag, 1990.

Extending Poitín to Handle Explicit Quantification

H. Kabir and G.W. Hamilton

School of Computing
Dublin City University
Dublin 9
IRELAND

Abstract. We have previously shown how the distillation program transformation algorithm can be used to prove inductive theorems in which there is no explicit quantification, and all variables are assumed to be implicitly universally quantified. These techniques were implemented in the theorem prover Poitín. In this paper, we show how Poitín can be extended to prove inductive theorems which contain explicit universal and existential quantifiers. This extension has also been implemented and added to Poitín; we give the results of applying the resulting theorem prover to a number of example conjectures.

1 Introduction

A wide range of inductive theorem proving systems have been developed (for example, NQTHM [4], CLAM [7], INKA [2], RRL [15]), but these tend to concentrate mainly on universal quantification. The inclusion of existential quantification is very problematic and greatly complicates the theorem proving process. The usual approach to proving existential inductive conjectures is to try to constructively find witnesses and prove that they satisfy the respective inductive property. However, the finding of such witnesses often requires the use of higher-order unification, which is in general undecidable. In this paper, we present an alternative approach to proving existential conjectures which performs a pure existence proof without the need to construct existential witnesses.

In previous work [11], we have presented the *distillation* program transformation algorithm, which was originally devised with the goal of eliminating intermediate data structures from functional programs. The distillation algorithm was largely influenced by Turchin's supercompilation [21], but improves greatly upon it. For example, supercompilation can only produce a linear speedup in programs, while distillation can produce a superlinear speedup. Turchin has shown how supercompilation can be used in inductive theorem proving [20], which also influenced our work on showing how the distillation algorithm can be used in inductive theorem proving in our theorem prover Poitín [10].

Our previous work on Poitín did not include any explicit quantification; all free variables within the input conjecture were assumed to be implicitly universally quantified. In this paper, we show how Poitín can be extended to prove inductive theorems which contain explicit universal and existential quantifiers. This extension has also been implemented and added to Poitín. We apply the resulting theorem prover to a number of example conjectures, the results of which are very promising.

The remainder of this paper is organised as follows. In Section 2, we give a brief overview of the distillation algorithm. In Section 3, we define rules to show how the Poitín theorem prover can be extended to handle explicit quantification and give examples of the application of these rules. In Section 4, we give the results of applying our extended version of Poitín to a number of inductive conjectures. In Section 5, we consider related work, and Section 6 concludes.

2 Distillation

In this section, we define the language used throughout this paper and we give a brief overview of the distillation algorithm. Due to space constraints, we cannot give much detail of the algorithm here; full details can be found in [11].

Definition 1 (Language). *The language used throughout this paper is a simple higher-order functional language as shown in Fig. 1.* □

$prog ::= e_0 \ \mathbf{where} \ f_1 = e_1; \ \ldots; \ f_n = e_n;$	program
$e ::= v$	variable
$\mid c \ e_1 \ \ldots \ e_n$	constructor application
$\mid \lambda v.e$	lambda abstraction
$\mid f$	function variable
$\mid e_0 \ e_1$	application
$\mid \mathbf{case} \ e_0 \ \mathbf{of} \ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k$	case expression
$\mid \mathbf{let} \ v = e_0 \ \mathbf{in} \ e_1$	let expression
$\mid \mathbf{letrec} \ f = e_0 \ \mathbf{in} \ e_1$	letrec expression
$p ::= c \ v_1 \ldots v_n$	pattern

Fig. 1. Language

Programs in the language consist of an expression to evaluate and a set of function definitions. The intended operational semantics of the language is normal order reduction. It is assumed that the language is typed using the Hindley-Milner polymorphic typing system (so erroneous terms such as $(c \ e_1 \ldots e_n) \ e$ and $\mathbf{case} \ (\lambda v.e) \ \mathbf{of} \ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$ cannot occur). The variables in the patterns of **case** expressions and the arguments of λ-abstractions are *bound*; all other variables are *free*. We use the notation $e[e_1/v_1 \ldots e_n/v_n]$ to represent the simultaneous substitution of the sub-expressions $e_1, \ldots, e_n$ for the free occurrences of variables $v_1, \ldots, v_n$, respectively, within e. We require that each function has exactly one definition and that all variables within a definition are bound. The propositional operators (*and*, *or*, *implies*, etc.) are implemented as functions in this language.

Each constructor has a fixed arity; for example *Nil* has arity 0 and *Cons* has arity 2. Within the expression **case** e_0 **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$, e_0 is called the *selector*, and $e_1 \ldots e_k$ are called the *branches.* The patterns in **case** expressions may not be nested. Methods to transform **case** expressions with nested patterns to ones without nested patterns are described in [1,23]. No variables may appear more than once within a pattern. We assume that the patterns in a **case** expression are non-overlapping and exhaustive.

Distillation is a powerful program transformation technique to remove intermediate data structures from higher-order functional programs and is significantly more powerful than the supercompilation algorithm [21]. This extra power is obtained through the use of a stronger form of matching prior to folding. In supercompilation, matching is performed on flat terms only; functions are considered to match only if they have the same name. In the distillation algorithm, matching is also performed on recursive terms, so different functions are considered to match if their corresponding recursive definitions also match.

The transformation rules in distillation are of the form $\mathcal{T}[\![e]\!]\ \rho\ \phi$ where e is the expression to be transformed, ρ is the set of previously encountered expressions and ϕ is the set of function definitions. These rules essentially perform normal-order reduction. *Folding* is performed when an expression is encountered which is an *instance* of a previously encountered expression, and *generalization* is performed to ensure termination of the transformation process. This generalization is performed when an expression is encountered which is an *embedding* of a previously encountered expression. The form of embedding which we use to guide this generalization is the *homeomorphic* embedding relation which was derived from results by Higman [12] and Kruskal [18] and was defined within term rewriting systems [8] for detecting the possible divergence of the term rewriting process.

Definition 2 (Distilled Form). *The expressions resulting from the distillation of a boolean expression are in the* distilled form *dt as defined in Fig. 2.* □

In addition, all of the functions within a distilled expression are terminating, as all possibly non-terminating functions are replaced by $\perp$ during distillation.

In order to use the distillation algorithm within our inductive theorem prover Poitín, we apply distillation to the input conjecture. The result of this transformation will be in distilled form. Inductive proof rules are then applied to this expression to try and prove it. The output from these proof rules will be either *True* indicating that the conjecture is true, or else $\perp$ which provides no information.

$$\begin{aligned} dt :=\ & v \\ &\mid True \\ &\mid False \\ &\mid \bot \\ &\mid \textbf{case}\ v\ \textbf{of}\ p_1 \ \Rightarrow\ dt'_1 \mid \ldots \mid p_k \ \Rightarrow\ dt'_k \\ &\mid \textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.dt\ \textbf{in}\ f\ v'_1 \ldots v'_n \\ &\mid f\ e_1 \ldots e_n \end{aligned}$$

Fig. 2. Distilled Form of Boolean Expressions

3 Explicit Quantification in Poitín

In this section, we show how the theorem prover Poitín can be extended to handle explicit quantification. We add quantifiers of the form ALL $v.e$ and EX $v.e$ to our language and, later in this section, we define sets of rules $\mathcal{A}$ for handling universal quantifiers and $\mathcal{E}$ for handling existential quantifiers. A proof of the soundness of these rules can be found in a longer version of this paper [14]. The transformation rules $\mathcal{T}$ for distillation are extended to be able to handle quantifiers as shown in Fig. 3.

$$\begin{aligned} &\mathcal{T}[\![\text{ALL}\ v_1 \ldots v_n.e]\!]\ \rho\ \phi\ = \mathcal{A}[\![e']\!]\ \{\}\ \{v_1 \ldots v_n\} \qquad (\mathcal{T}1) \\ &\text{where} \\ &e' = \mathcal{T}[\![e]\!]\ \{\}\ \phi \\ \\ &\mathcal{T}[\![\text{EX}\ v_1 \ldots v_n.e]\!]\ \rho\ \phi\ = \mathcal{E}[\![e']\!]\ \{\}\ \{v_1 \ldots v_n\} \qquad (\mathcal{T}2) \\ &\text{where} \\ &e' = \mathcal{T}[\![e]\!]\ \{\}\ \phi \end{aligned}$$

Fig. 3. Distillation Rules for Quantifiers

Rule ($\mathcal{T}1$) handles universally quantified expressions. Within a universally quantified expression ALL $v_1 \ldots v_n.e$, the subterm e (within which the variables $v_1 \ldots v_n$ are therefore free) is first of all transformed using the rules $\mathcal{T}$ for distillation. The proof rules defined by $\mathcal{A}$ are then applied to the resulting expression. In a similar way, existential quantification is handled by application of the rule ($\mathcal{T}2$). Within an existentially quantified expression EX $v_1 \ldots v_n.e$, the subterm e is first of all transformed using the rules $\mathcal{T}$ for distillation. The proof rules defined by $\mathcal{E}$ are then applied to the resulting expression.

If there are a number of nested quantifiers within the conjecture to be proved, then the proof rules will be applied to the innermost quantified term first. These inner quantified terms may contain free variables, which will be bound by another quantifier in some outer scope. The term resulting from the application of these proof rules may therefore also contain free variables if these were present in the original term. We therefore construct a hierarchy of transformations which correspond to *metasystem transitions* [22,9].

3.1 Proving Universally Quantified Conjectures.

The rules for proving a universally quantified conjecture e are defined by $\mathcal{A}[\![e]\!]\ \rho\ \phi$ as shown in Fig. 4, where the parameter ρ is the set of previously encountered function calls and ϕ is the set of universally quantified variables.

$$\begin{array}{ll}
\mathcal{A}[\![v]\!]\ \rho\ \phi = \bot, \text{ if } v \in \phi & (\mathcal{A}1)\\
\qquad\qquad = v,\ \text{otherwise} &
\end{array}$$

$$\mathcal{A}[\![True]\!]\ \rho\ \phi\ = True \qquad (\mathcal{A}2)$$

$$\mathcal{A}[\![False]\!]\ \rho\ \phi\ = \bot \qquad (\mathcal{A}3)$$

$$\mathcal{A}[\![\bot]\!]\ \rho\ \phi\ = \bot \qquad (\mathcal{A}4)$$

$$\begin{array}{ll}
\mathcal{A}[\![\textbf{case}\ v\ \textbf{of}\ p_1\ :\ e_1 \mid \ldots \mid p_n\ :\ e_n]\!]\ \rho\ \phi & (\mathcal{A}5)\\
= (\mathcal{A}[\![e_1]\!]\ \rho\ (\phi \cup \{v_{11} \ldots v_{1k_1}\}))\ \wedge \ldots \wedge\ (\mathcal{A}[\![e_n]\!]\ \rho\ (\phi \cup \{v_{n1} \ldots v_{nk_n}\})), & \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{if } v \in \phi & \\
= \textbf{case}\ v\ \textbf{of}\ p_1\ :\ (\mathcal{A}[\![e_1]\!]\ \rho\ \phi) \mid \ldots \mid p_n\ :\ (\mathcal{A}[\![e_n]\!]\ \rho\ \phi), \text{otherwise} & \\
\text{where} & \\
p_i\ =\ c_i\ v_{i1} \ldots v_{ik_i} &
\end{array}$$

$$\begin{array}{ll}
\mathcal{A}[\![\textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.e_0\ \textbf{in}\ f\ v_1 \ldots v_n]\!]\ \rho\ \phi & (\mathcal{A}6)\\
= \mathcal{A}[\![e_0]\!]\ (\rho \cup \{f\ v_1 \ldots v_n\})\ \phi, \text{ if } \{v_1 \ldots v_n\} \subseteq \phi & \\
= \textbf{letrec}\ f\ =\ \lambda v'_1 \ldots v'_k.(\mathcal{A}[\![e_0]\!]\ (\rho \cup \{f\ v_1 \ldots v_n\})\ \phi)\ \textbf{in}\ f\ v'_1 \ldots v'_k, & \\
\qquad\qquad\qquad\qquad \text{otherwise} & \\
\text{where} & \\
v'_1 \ldots v'_k = \{v_1 \ldots v_n\} \setminus \phi &
\end{array}$$

$$\begin{array}{ll}
\mathcal{A}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi & (\mathcal{A}7)\\
= True, \qquad\qquad\qquad \text{if } \{v_1 \ldots v_n\} \subseteq \phi & \\
= (f\ v'_1 \ldots v'_k)[e_1/v_1 \ldots e_n/v_n], \text{otherwise} & \\
\text{where} & \\
(f\ v_1 \ldots v_n) \in \rho & \\
v'_1 \ldots v'_k = \{v_1 \ldots v_n\} \setminus \phi &
\end{array}$$

Fig. 4. Proof Rules for Universal Quantification

Note that these rules will only be applied to terms which are in distilled form. Using these rules, the universally quantified variables contained within ϕ are eliminated, and a simplified expression defined over the remaining free variables is obtained. If there are no free variables, then the input conjecture is either reduced to *True* indicating that it is true, or else $\perp$ which provides no information. In rule ($\mathcal{A}1$), if a universally quantified variable is encountered, then since it must be a Boolean, the value $\perp$ is returned as the variable cannot always be *True*. If a free variable is encountered, then it is left unchanged. In rule ($\mathcal{A}2$), if the value *True* is encountered, then the value *True* is returned. In rules ($\mathcal{A}3$) and ($\mathcal{A}4$), if the values *False* or $\perp$ are encountered, then the value $\perp$ is returned. In rule ($\mathcal{A}5$), if we encounter a **case** expression, then since this expression will be in distilled form, the redex must be a variable. If this variable is free, then it remains in the resulting term, and the proof rules are further applied to the branches of the **case** expression. If the selector variable is universally quantified, then a *case split* is performed in which we prove the current term separately for each of the possible values of the selector, and then return the conjunction of the resulting values. The different possible values of the selector are simply the patterns within the **case** expression. In rule ($\mathcal{A}6$), if we encounter a **letrec** function definition and all the parameters in the initial application of this function are universally quantified, then this function application is a potential inductive hypothesis. Since at least one of these parameters must be decreasing, this parameter can be used as the *induction variable*. If we subsequently encounter a recursive call of this function in rule ($\mathcal{A}7$), then we have re-encountered this inductive hypothesis, so the value *True* is returned. If a function definition contains free variables, then the function is re-defined over these free variables.

Example 1. Consider the following conjecture:

$$\begin{array}{l}
\text{ALL } xs\ ys.eqnum\ (length\ (append\ xs\ ys))\ (plus\ (length\ xs)\ (length\ ys))\\
\textbf{where}\\
\begin{array}{lll}
eqnum & = \lambda x.\lambda y.\,\textbf{case}\ x\ \textbf{of} & \\
 & \qquad\qquad Zero \Rightarrow \textbf{case}\ y\ \textbf{of} & \\
 & \qquad\qquad\qquad\qquad Zero \Rightarrow True & \\
 & \qquad\qquad\qquad\qquad \mid Succ\ y' \Rightarrow False & \\
 & \qquad\qquad \mid Succ\ x' \Rightarrow \textbf{case}\ y\ \textbf{of} & \\
 & \qquad\qquad\qquad\qquad Zero \Rightarrow False & \\
 & \qquad\qquad\qquad\qquad \mid Succ\ y' \Rightarrow eqnum\ x'\ y' & \\
length & = \lambda xs.\,\textbf{case}\ xs\ \textbf{of} & \\
 & \qquad\qquad Nil \Rightarrow Zero & \\
 & \qquad\qquad \mid Cons\ x\ xs' \Rightarrow Succ\ (length\ xs') & \\
append & = \lambda xs.\lambda ys.\,\textbf{case}\ xs\ \textbf{of} & \\
 & \qquad\qquad Nil \Rightarrow ys & \\
 & \qquad\qquad \mid Cons\ x\ xs' \Rightarrow Cons\ x\ (append\ xs'\ ys) & \\
plus & = \lambda x.\lambda y.\,\textbf{case}\ x\ \textbf{of} & \\
 & \qquad\qquad Zero \Rightarrow y & \\
 & \qquad\qquad \mid Succ\ x' \Rightarrow Succ\ (plus\ x'\ y) &
\end{array}
\end{array}$$

This conjecture states that the length of appending two lists is equal to the sum of their individual lengths. The result of distilling this term is as follows:

```
letrec
f0 = λxs. case xs of
              Nil          ⇒ case ys of
                                 Nil          ⇒ True
                                 Cons y ys′ ⇒
                                    letrec
                                    f1 = λy.λys′. case ys′ of
                                                      Nil            ⇒ True
                                                    | Cons y′ ys″ ⇒ f1 y′ ys″
                                    in f1 y ys′
            | Cons x xs′ ⇒ f0 xs′
in f0 xs
```

The proof of this term proceeds as shown in Figs 5 and 6.

```
𝒜⟦f0 xs⟧ {} {xs, ys}
= (by 𝒜6)
𝒜⟦case xs of
      Nil          ⇒ case ys of
                         Nil          ⇒ True
                         Cons y ys′ ⇒
                            letrec
                            f1 = λy.λys′. case ys′ of
                                              Nil            ⇒ True
                                            | Cons y′ ys″ ⇒ f1 y′ ys″
                            in f1 y ys′
    | Cons x xs′ ⇒ f0 xs′⟧ {f0 xs} {xs, ys}
= (by 𝒜5)
(𝒜⟦case ys of
       Nil          ⇒ True
     | Cons y ys′ ⇒ letrec f1 = λy.λys′. case ys′ of
                                               Nil            ⇒ True
                                             | Cons y′ ys″ ⇒ f1 y′ ys″
                      in f1 y ys′⟧ {f0 xs} {xs, ys})
∧ (𝒜⟦f0 xs′⟧ {f0 xs} {xs, ys, x, xs′})
= (by 𝒜5)
(𝒜⟦True⟧ {f0 xs} {xs, ys})
∧ (𝒜⟦letrec f1 = λy.λys′. case ys′ of
                              Nil            ⇒ True
                            | Cons y′ ys″ ⇒ f1 y′ ys″
      in f1 y ys′⟧ {f0 xs} {xs, ys, y, ys′}) ∧ (𝒜⟦f0 xs′⟧ {f0 xs} {xs, ys, x, xs′})
```

Fig. 5. Universal Proof

$$
\begin{array}{l}
= (\text{by } \mathcal{A}2)\\
\mathit{True} \ \wedge \ (\mathcal{A}[\![\,\mathbf{letrec}\ f1 \ = \ \lambda y.\lambda ys'.\,\mathbf{case}\ ys'\ \mathbf{of}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Nil} \qquad\quad \Rightarrow \mathit{True}\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mid \mathit{Cons}\ y'\ ys'' \Rightarrow \mathit{f1}\ y'\ ys''\\
\qquad\qquad \mathbf{in}\ \mathit{f1}\ y\ ys']\!]\ \{\mathit{f0}\ xs\}\ \{xs, ys, y, ys'\})\\
\qquad\quad \wedge\ (\mathcal{A}[\![\mathit{f0}\ xs']\!]\ \{\mathit{f0}\ xs\}\ \{xs, ys, x, xs'\})\\
= (\text{by } \mathcal{A}6)\\
\mathit{True} \ \wedge \ (\mathcal{A}[\![\,\mathbf{case}\ ys'\ \mathbf{of}\\
\qquad\qquad\qquad \mathit{Nil} \qquad\quad \Rightarrow \mathit{True}\\
\qquad\qquad\quad \mid \mathit{Cons}\ y'\ ys'' \Rightarrow \mathit{f1}\ y'\ ys'']\!]\ \{\mathit{f0}\ xs, \mathit{f1}\ y\ ys'\}\ \{xs, ys, y, ys'\})\\
\qquad\quad \wedge\ (\mathcal{A}[\![\mathit{f0}\ xs']\!]\ \{\mathit{f0}\ xs\}\ \{xs, ys, x, xs'\})\\
= (\text{by } \mathcal{A}5)\\
\mathit{True} \ \wedge \ (\mathcal{A}[\![\mathit{True}]\!]\ \{\mathit{f0}\ xs, \mathit{f1}\ y\ ys'\}\ \{xs, ys, y, ys'\})\\
\qquad\quad \wedge\ (\mathcal{A}[\![\mathit{f1}\ y'\ ys'']\!]\ \{\mathit{f0}\ xs, \mathit{f1}\ y\ ys'\}\ \{xs, ys, y, ys', y', ys''\})\\
\qquad\qquad \wedge\ (\mathcal{A}[\![\mathit{f0}\ xs']\!]\ \{\mathit{f0}\ xs\}\ \{xs, ys, x, xs'\})\\
= (\text{by } \mathcal{A}2)\\
\mathit{True} \ \wedge \ \mathit{True} \ \wedge \ (\mathcal{A}[\![\mathit{f1}\ y'\ ys'']\!]\ \{\mathit{f0}\ xs, \mathit{f1}\ y\ ys'\}\ \{xs, ys, y, ys', y', ys''\})\\
\qquad\qquad\qquad\quad \wedge\ (\mathcal{A}[\![\mathit{f0}\ xs']\!]\ \{\mathit{f0}\ xs\}\ \{xs, ys, x, xs'\})\\
= (\text{by } \mathcal{A}7)\\
\mathit{True} \ \wedge \ \mathit{True} \ \wedge \ \mathit{True} \ \wedge \ (\mathcal{A}[\![\mathit{f0}\ xs']\!]\ \{\mathit{f0}\ xs\}\ \{xs, ys, x, xs'\})\\
= (\text{by } \mathcal{A}7)\\
\mathit{True} \ \wedge \ \mathit{True} \ \wedge \ \mathit{True} \ \wedge \ \mathit{True}\\
= \mathit{True}
\end{array}
$$

Fig. 6. Universal Proof (continued)

3.2 Proving Existentially Quantified Conjectures.

The rules for proving an existentially quantified conjecture e are defined by $\mathcal{E}[\![e]\!]\ \rho\ \phi$ as shown in Fig. 7, where the parameter ρ is the set of previously encountered function calls and ϕ is the set of existentially quantified variables. Using these rules, the existentially quantified variables contained within ϕ are eliminated, and a simplified expression over the remaining free variables is obtained. The rules are similar to those for universal quantification, with the only differences being in rules ($\mathcal{E}1$), ($\mathcal{E}5$), ($\mathcal{E}6$) and ($\mathcal{E}7$). In rule ($\mathcal{E}1$), if an existentially quantified variable is encountered, then since it must be a Boolean, the value *True* is returned as the value of the variable can be *True*. In rule ($\mathcal{E}5$), if the selector in a **case** expression is an existentially quantified variable, then we also perform a *case split* and prove the current term separately for each of the possible values of the selector, but in this instance we return the disjunction of the resulting values. In rules ($\mathcal{E}6$) and ($\mathcal{E}7$), function applications are no longer possible inductive hypotheses as they contain existential variables. However, if all of the parameters in a function application are existentially quantified then the value $\perp$ is returned as we know that the search space of these existential variables has been exhausted.

$$
\begin{array}{ll}
\mathcal{E}[\![v]\!]\ \rho\ \phi = True, \text{ if } v \in \phi & (\mathcal{E}1)\\
\qquad\quad = v \qquad \text{otherwise} & \\[1em]
\mathcal{E}[\![True]\!]\ \rho\ \phi\ = True & (\mathcal{E}2)\\[1em]
\mathcal{E}[\![False]\!]\ \rho\ \phi\ = \bot & (\mathcal{E}3)\\[1em]
\mathcal{E}[\![\bot]\!]\ \rho\ \phi\ = \bot & (\mathcal{E}4)\\[1em]
\mathcal{E}[\![\textbf{case}\ v\ \textbf{of}\ p_1\ :\ e_1 \mid \ldots \mid p_n\ :\ e_n]\!]\ \rho\ \phi & (\mathcal{E}5)\\
= (\mathcal{E}[\![e_1]\!]\ (\phi \cup \{v_{11} \ldots v_{1k_1}\}))\ \vee \ldots \vee\ (\mathcal{E}[\![e_n]\!]\ \rho\ (\phi \cup \{v_{n1} \ldots v_{nk_n}\})), & \\
\hfill \text{if } v \in \phi & \\
= \textbf{case}\ v\ \textbf{of}\ p_1\ :\ (\mathcal{E}[\![e_1]\!]\ \rho\ \phi) \mid \ldots \mid p_n\ :\ (\mathcal{E}[\![e_n]\!]\ \rho\ \phi), \text{ otherwise} & \\
\text{where} & \\
p_i\ =\ c_i\ v_{i1} \ldots v_{ik_i} & \\[1em]
\mathcal{E}[\![\textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.e_0\ \textbf{in}\ f\ v_1 \ldots v_n]\!]\ \rho\ \phi & (\mathcal{E}6)\\
= \mathcal{E}[\![e_0]\!]\ (\rho \cup \{f\ v_1 \ldots v_n\})\ \phi, \text{ if } \{v_1 \ldots v_n\} \subseteq \phi & \\
= \textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_k.(\mathcal{E}[\![e_0]\!]\ (\rho \cup \{f\ v_1 \ldots v_n\})\ \phi)\ \textbf{in}\ f\ v_1 \ldots v_k, & \\
\qquad\qquad\qquad\qquad \text{otherwise} & \\
\text{where} & \\
v_1 \ldots v_k = \{v_1 \ldots v_n\} \setminus \phi & \\[1em]
\mathcal{E}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi & (\mathcal{E}7)\\
= \bot, \qquad\qquad\qquad \text{if } \{v_1 \ldots v_n\} \subseteq \phi & \\
= (f\ v'_1 \ldots v'_k)[e_1/v_1 \ldots e_n/v_n], \text{ otherwise} & \\
\text{where} & \\
(f\ v_1 \ldots v_n) \in \rho & \\
v'_1 \ldots v'_k = \{v_1 \ldots v_n\} \setminus \phi &
\end{array}
$$

Fig. 7. Proof Rules for Existential Quantification

Example 2. Consider the following conjecture:

$$
\begin{array}{l}
\text{ALL } x.\text{EX } y.\mathit{iff}\ (\mathit{even}\ x)\ (\mathit{eqnum}\ (\mathit{double}\ y)\ x)\\
\textbf{where}\\
\mathit{iff}\ \ = \lambda x.\lambda y.\,\textbf{case}\ x\ \textbf{of}\\
\qquad\qquad\qquad \mathit{True} \Rightarrow y\\
\qquad\qquad\quad \mid \mathit{False} \Rightarrow \textbf{case}\ y\ \textbf{of}\\
\qquad\qquad\qquad\qquad\qquad \mathit{True} \Rightarrow \mathit{False}\\
\qquad\qquad\qquad\qquad\quad\ \mid \mathit{False} \Rightarrow \mathit{True}\\
\mathit{even} = \lambda x.\,\textbf{case}\ x\ \textbf{of}\\
\qquad\qquad\quad \mathit{Zero} \quad\ \Rightarrow \mathit{True}\\
\qquad\qquad \mid \mathit{Succ}\ x' \Rightarrow \textbf{case}\ x'\ \textbf{of}\\
\qquad\qquad\qquad\qquad\quad \mathit{Zero} \quad\ \Rightarrow \mathit{False}\\
\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ x'' \Rightarrow \mathit{even}\ x''
\end{array}
$$

```
eqnum = λx.λy. case x of
                   Zero    ⇒ case y of
                                 Zero    ⇒ True
                               | Succ y' ⇒ False
                 | Succ x' ⇒ case y of
                                 Zero    ⇒ False
                               | Succ y' ⇒ eqnum x' y'
double = λx. case x of
               Zero    ⇒ Zero
             | Succ x' ⇒ Succ (Succ (double x'))
```

This conjecture states that for all values of x, there exists a y such that if x is *even*, then y is exactly half of x. The result of distilling this term is as follows:

```
letrec f1 = λx.λy. case x of
                       Zero    ⇒ case y of
                                     Zero    ⇒ True
                                   | Succ y' ⇒ False
                     | Succ x' ⇒ case x' of
                                     Zero     ⇒ case y of
                                                    Zero    ⇒ True
                                                  | Succ y' ⇒ True
                                   | Succ x'' ⇒ f1 x'' y
in f1 x y
```

As the innermost quantifier in the original conjecture is an existential quantifier, existential proof rules are applied first to this term as shown in Figs 8 and 9.

```
ℰ⟦letrec f1 = λx.λy. case x of
                         Zero    ⇒ case y of
                                       Zero    ⇒ True
                                     | Succ y' ⇒ False
                       | Succ x' ⇒ case x' of
                                       Zero     ⇒ case y of
                                                      Zero    ⇒ True
                                                    | Succ y' ⇒ True
                                     | Succ x'' ⇒ f1 x'' y
  in f1 x y⟧ {} {y}
```

Fig. 8. Example Proof

As the outermost quantifier in the original conjecture is a universal quantifier, universal proof rules are then applied to the resulting term as shown in Fig. 10. The result of applying these rules is the value *True*, so the original conjecture has been proved.

```
= (by ℰ6)
letrec f1 = λx.ℰ⟦case x of
                   Zero    ⇒ case y of
                                 Zero    ⇒ True
                               | Succ y′ ⇒ False
                 | Succ x′ ⇒ case x′ of
                                 Zero    ⇒ case y of
                                               Zero    ⇒ True
                                             | Succ y′ ⇒ True
                               | Succ x″ ⇒ f1 x″ y⟧ {f1 x y} {y}
in f1 x
= (by ℰ5, ℰ5)
letrec
f1 = λx. case x of
           Zero    ⇒ ℰ⟦case y of
                          Zero    ⇒ True
                        | Succ y′ ⇒ False⟧ {f1 x y} {y}
         | Succ x′ ⇒ case x′ of
                          Zero    ⇒ ℰ⟦case y of
                                         Zero    ⇒ True
                                       | Succ y′ ⇒ True⟧ {f1 x y} {y}
                        | Succ x″ ⇒ ℰ⟦f1 x″ y⟧ {f1 x y} {y}
in f1 x
= (by ℰ5, ℰ5)
letrec f1 = λx. case x of
                  Zero    ⇒ (ℰ⟦True⟧ {f1 x y} {y})
                             ∨ (ℰ⟦False⟧ {f1 x y} {y})
                | Succ x′ ⇒ case x′ of
                               Zero    ⇒ (ℰ⟦True⟧ {f1 x y} {y})
                                          ∨ (ℰ⟦True⟧ {f1 x y} {y})
                             | Succ x″ ⇒ ℰ⟦f1 x″ y⟧ {f1 x y} {y}
in f1 x
= (by ℰ2, ℰ3, ℰ2, ℰ2, ℰ7)
letrec f1 = λx. case x of
                  Zero    ⇒ True
                | Succ x′ ⇒ case x′ of
                               Zero    ⇒ True
                             | Succ x″ ⇒ f1 x″
in f1 x
```

Fig. 9. Example Proof (continued)

4 Results

Some results of applying Poitín to a range of conjectures are shown in Table 1. The times given in this table are for an average of 10 runs on an Intel Pentium 4 PC with 2.40 GHz and 512 MB RAM. As can be seen, these times are all

$$
\begin{array}{l}
\mathcal{A}[\![\,\mathbf{letrec}\ \mathit{f1}\ =\ \lambda x.\,\mathbf{case}\ x\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{Zero}\quad \Rightarrow \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\ \mid \mathit{Succ}\ x' \Rightarrow \mathbf{case}\ x'\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero}\quad \Rightarrow \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mid \mathit{Succ}\ x'' \Rightarrow \mathit{f1}\ x'' \\
\quad \mathbf{in}\ \mathit{f1}\ x]\!]\ \{\}\ \{x\} \\
= (\text{by } \mathcal{A}6) \\
\mathcal{A}[\![\,\mathbf{case}\ x\ \mathbf{of} \\
\qquad\quad \mathit{Zero}\quad \Rightarrow \mathit{True} \\
\qquad\ \mid \mathit{Succ}\ x' \Rightarrow \mathbf{case}\ x'\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\quad \mathit{Zero}\quad \Rightarrow \mathit{True} \\
\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ x'' \Rightarrow \mathit{f1}\ x'']\!]\ \{\mathit{f1}\ x\}\ \{x\} \\
= (\text{by } \mathcal{A}5,\ \mathcal{A}5) \\
(\mathcal{A}[\![\mathit{True}]\!]\ \{\mathit{f1}\ x\}\ \{x\})\ \wedge\ (\mathcal{A}[\![\mathit{True}]\!]\ \{\mathit{f1}\ x\}\ \{x, x'\}) \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge\ (\mathcal{A}[\![\mathit{f1}\ x'']\!]\ \{\mathit{f1}\ x\}\ \{x, x', x''\}) \\
= (\text{by } \mathcal{A}2,\ \mathcal{A}2,\ \mathcal{A}7) \\
\mathit{True}\ \wedge\ \mathit{True}\ \wedge\ \mathit{True} \\
= \mathit{True}
\end{array}
$$

Fig. 10. Example Proof (continued)

very low, so the results are encouraging. All the conjectures were proved by performing only generalization without using any intermediate lemmas, whereas some other inductive theorem provers require both lemmas and generalizations to prove these theorems. Some of the conjectures listed in Fig. 1 were proved by SPIKE [3] using a divergence critic [24], NQTHM [4,5], CLAM [7] using rippling, and Periwinkle [16] by proposing lemmas or performing generalizations.

5 Related Work

The distillation algorithm and the Poitín theorem prover were largely inspired by Turchin's work on supercompilation [21], and its use in theorem proving [20]. Over-generalization occurs a lot more frequently when using supercompilation as opposed to distillation, thus greatly limiting its power. In order to show that the term resulting from supercompilation terminates, Turchin requires that all functions are total, so the onus is on the user to show that this really is the case. In Poitín, the termination of the term resulting from distillation is determined automatically. Turchin's use of metasystem transitions in theorem proving is analagous to the hierarchy of transformations which we use in Poitín.

A number of different approaches have been developed to identify potentially failing proof attempts, and to apply appropriate techniques to allow the proof to go through. Rippling is a powerful technique developed at the University of Edinburgh for proving theorems involving explicit induction [6]. In the step case of an inductive proof, the induction conclusion typically differs from the induction

No.	**Conjecture**	Time (in Seconds)
1.	ALL x.ALL $y.eqnum\ (plus\ x\ y)\ (plus\ y\ x)$	0.0094
2.	ALL $x.eqnum\ (plus\ x\ (Succ\ x))\ (Succ\ (plus\ x\ x))$	0.0016
3.	ALL x.ALL y.ALL $z.eqnum\ (plus\ (plus\ x\ y)\ z)\ (plus\ x\ (plus\ y\ z))$	0.0032
4.	ALL $x.eqnum\ (plus\ (plus\ x\ x)\ x)\ (plus\ x\ (plus\ x\ x))$	0.0046
5.	ALL $x.eqnum\ (gcd\ x\ x)\ x$	0.0016
6.	ALL x.ALL $y.eqnum\ (sub\ (plus\ x\ y)\ x)\ y$	0.0016
7.	ALL $x.odd\ (plus\ (Succ\ x)\ x)$	0.0015
8.	ALL x.ALL $y.eqnum\ (plus\ x\ (Succ\ y))\ (Succ\ (plus\ x\ y))$	0.0015
9.	ALL $x.even\ (plus\ x\ x)$	0.0016
10.	ALL $x.even\ (doublea\ x\ Zero)$	0.0016
11.	ALL x.ALL $y.((even\ x) \wedge (even\ y)) \Rightarrow (even\ (plus\ x\ y))$	0.0078
12.	ALL $x.(eqbool\ (even\ x)\ (True)) \Rightarrow (eqbool\ (odd\ x)\ (False))$	0.0015
13.	ALL x.EX $y.(even\ x) \Leftrightarrow (eqnum\ (double\ y)\ x)$	0.0077
14.	ALL x.EX $y.(even\ x) \Leftrightarrow (eqnum\ (mult\ y\ (Succ\ (Succ\ Zero)))\ x)$	0.0016
15.	ALL x.ALL y.EX $z.(less\ x\ y) \Rightarrow (eqnum\ (plus\ x\ z)\ y)$	0.0032
16.	ALL x.EX $y.(eqnum\ x\ Zero) \vee (eqnum\ x\ (Succ\ y))$	0.0031
17.	ALL x.EX $y.eqnum\ y\ (plus\ x\ (Succ\ Zero))$	0.0032
18.	ALL x.ALL y.EX $z.(leq\ y\ x) \Rightarrow (eqnum\ (plus\ z\ y)\ x)$	0.0046
19.	ALL x.EX $y.(eqnum\ (double\ y)\ x) \vee (eqnum\ (Succ\ (double\ y))\ x)$	0.0015
20.	ALL x.ALL y.EX $z.(eqnum\ (plus\ x\ z)\ y) \vee (eqnum\ (sub\ x\ z)\ y)$	0.022
21.	ALL xs.ALL $ys.eqnum\ (length\ (append\ xs\ ys))\ (length\ (append\ ys\ xs))$	0.0109
22.	ALL xs.ALL $ys.eqnum\ (length\ (append\ xs\ ys))\ (plus\ (length\ xs)\ (length\ ys))$	0.0016
23.	ALL xs.ALL ys.ALL $zs.eqlist\ (append\ xs\ (append\ ys\ zs))$ $(append\ (append\ xs\ ys)\ zs)$	0.0063
24.	ALL xs.ALL $ys.(even\ (length\ (append\ xs\ ys))) \leftrightarrow$ $(even\ (length\ (append\ ys\ xs)))$	0.0548

Table 1. Some Conjectures Proved by Poitín

hypothesis. Rippling uses annotations to mark these differences and applies annotated rewrite rules to remove them. In the case where no rewrite rules can be applied, the proof becomes *blocked*. In this case, *proof critics* [13] can be applied. Various critics for explicit induction have been developed that speculate missing lemmas, perform generalizations, etc. There are significant differences between the rippling approach, and the approach described here. Firstly, rippling works in an explicit induction setting, as opposed to the implicit approach described here. Secondly, in rippling, the difference matching is performed statically on the rewrite rules (although a dynamic version of rippling has also been developed [19]). In distillation, this difference matching is dynamically performed on each term as it is encountered during rewriting. Thirdly, rippling often requires the use of additional lemmas to allow the proof to go through. This may therefore require a reasonable amount of search, and possible user guidance. In Poitín, no additional lemmas are required, thus reducing the amount of search required, and allowing proofs to be performed fully automatically.

The notion of rippling has been extended to be able to deal with existentially quantified variables and synthesis in the work on *middle-out reasoning* [16]. This approach requires trying to construct existential witnesses and prove that they satisfy the required property. However, the finding of such witnesses usually requires the use of higher-order unification, which is in general undecidable. In the approach presented here we perform a pure existence proof without the need to construct existential witnesses.

In [17], rippling is combined with matrix-based constructive theorem proving. This approach is used to generate inductive specification proofs and for automating the synthesis of recursive programs. This approach does have the advantage that it guarantees that the synthesized program is correct, so this does not need to be verified afterwards. However, like the other approaches described here, it does require a high degree of user interaction, which is not the case for Poitín.

6 Conclusions

In this paper, we have shown how the Poitín theorem prover can be extended to prove inductive theorems which contain explicit universal and existential quantification. We argue that the Poitín theorem prover greatly extends the range of theorems which can be proved fully automatically without the need for intermediate lemmas. Poitín is also fully deterministic and only needs to search through a subset of previously encountered expressions, rather than through a large collection of rules and axioms. We therefore argue that Poitín is likely to be more efficient than other theorem provers which have a relatively large search space and require backtracking.

We have implemented the described extension to the Poitín theorem prover, and shown some of the results obtained by applying it to a range of inductive conjectures. Although the results are encouraging, there are still a number of fairly straightforward conjectures which cannot currently be proved by Poitín; we are currently working on these. There are a number of possible directions for further work. Firstly, the implementation of Poitín must be improved as mentioned above, and run on a wider range of test cases. This would allow a more thorough examination of the range of theorems which can be proved by Poitín, and a more detailed comparison with other theorem provers. Secondly, Poitín cannot currently extract programs from existential proofs. Work is currently under way to try to achieve this. This would then allow us to construct programs from their specifications.

References

1. L. Augustsson. Compiling Pattern Matching. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 368–381. Springer-Verlag, 1985.
2. S. Biundo, B. Hummel, D. Hutter, and C. Walther. The Karlrsruhe Induction Theorem Proving System. *Lecture Notes in Computer Science*, 230:672–674, 1987.

3. A. Bouhoula and M. Rusinowitch. Implicit Induction in Conditional Theories. *Journal of Automated Reasoning*, 14(2):189–235, 1995.
4. R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979.
5. R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, 1998.
6. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.
7. Alan Bundy, Frank Van Harmelen, Christian Horn, and Alan Smaill. The Oyster-CLAM System. In *Proceedings of the 10th International Conference on Automated Deduction*, pages 647–648, 1990.
8. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier, MIT Press, 1990.
9. Robert Glück and Morten Heine Sørensen. A Roadmap to Metacomputation by Supercompilation. In *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 137–160. Springer-Verlag, 1996.
10. G. W. Hamilton. Poitín: Distilling Theorems From Conjectures. *Electronic Notes in Theoretical Computer Science*, 151(1):143–160, 2006.
11. G.W. Hamilton. Distillation: Extracting the Essence of Programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 61–70, 2007.
12. G. Higman. Ordering by Divisibility in Abstract Algebras. *Proceedings of the London Mathemtical Society*, 2:326–336, 1952.
13. A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In *International Conference on Logic Programming and Automated Reasoning*, pages 178–189, 1992.
14. M.H. Kabir and G.W. Hamilton. Extending Poitín to Handle Explicit Quantification. Working Paper CA07-01, School of Computing, Dublin City University, 2007. http://www.computing.dcu.ie/research/papers/2007/CA-0107.pdf.
15. Deepak Kapur, G. Sivakumar, and Hantao Zhang. RRL: A Rewrite Rule Laboratory. *Lecture Notes in Computer Science*, 230:691–692, 1986.
16. I. Kraan, D. Basin, and A. Bundy. Middle-Out Reasoning For Synthesis and Induction. *Journal of Automated Reasoning*, 16(1–2):113–145, 1996.
17. Christoph Kreitz and Brigitte Pientka. Connection-Driven Inductive Theorem Proving. *Studia Logica*, 69(2):293–326, 2001.
18. J.B. Kruskal. Well-Quasi Ordering, the Tree Theorem, and Vazsonyi's Conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.
19. A. Smaill and I. Green. Higher-Order Annotated Terms for Proof Search. In *Proceedings of the International Conference on Theorem Proving in Higher Order Logics*, pages 399–413, 1996.
20. V.F. Turchin. The Use of Metasystem Transition in Theorem Proving and Program Optimization. *Lecture Notes in Computer Science*, 85:645 – 657, 1980.
21. V.F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):90–121, July 1986.
22. V.F. Turchin. Metacomputation: Metasystem Transitions plus Supercompilation. *Lecture Notes in Computer Science*, 1110:481–509, 1996.
23. P. Wadler. Efficient Compilation of Pattern Matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pages 78–103. Prentice Hall, 1987.
24. T. Walsh. A Divergence Critic for Inductive Proof. *Journal of Artificial Intelligence Research*, 4:209–236, 1996.

Developing Modal Tableaux and Resolution Methods via First-Order Resolution

Renate A. Schmidt

School of Computer Science, The University of Manchester
renate.schmidt@manchester.ac.uk

Abstract

This paper explores the use of resolution as a meta-framework for developing different deduction calculi for modal dynamic logics. Dynamic modal logics are *PDL*-like extended modal logics which are closely related to description logics. We show how tableau systems, modal resolution systems and Rasiowa-Sikorski systems, which are dual tableau systems, can be developed and studied by using standard principles and methods of first-order theorem proving. The approach is based on the translation of modal logic reasoning problems to first-order clausal form and using a suitable refinement of resolution to construct and mimic derivations of the desired proof method. The inference rules of the calculus can then be read off from the clausal form used. We show how this approach can be used to generate new proof calculi for logics that have not been considered in the literature before and prove soundness, completeness and decidability results. This slightly unusual approach allows us to gain new insight and results for familiar and less familiar logics and different proof methods, and compare them in a common framework.

Published in *Advances in Modal Logic*, Volume 6, College Publications, London, 1–16 (2006).

Reasoning Automatically about Termination and Refinement

Georg Struth

Department of Computer Science, University of Sheffield, UK
g.struth@dcs.shef.ac.uk

Abstract. We present very short mechanised proofs of Bachmair and Dershowitz's termination theorem in different variants of Kleene algebras. Through our experiments we also discover three novel refinement laws for nested infinite loops. Finally, we introduce novel divergence modules in which full automation could be achieved. These structures seem very promising for automated reasoning about infinite behaviours in programs and discrete dynamical systems.

1 Introduction

In 1986, in a fundamental study of commutation, transformation and termination properties of rewrite systems [4], Bachmair and Dershowitz proved the following, by now classical theorem: *Termination of the union of two rewrite systems can be separated into termination of the individual systems if one rewrite systems quasicommutes over the other.* In this context, rewrite systems are considered as abstract reduction systems which are essentially sets of binary relations. Quasicommutation models a quite general way of rearranging rewrite sequences that subsumes a number of interesting cases. The termination theorem yields a powerful tool for analysing termination of rewrite systems. It also provides a very general transformation and refinement law for programs, reactive and concurrent systems. The proof sketch contained in the original paper informally analyses infinite rewrite sequences.

Motivated by applications in concurrency control, Ernie Cohen posed this termination theorem as a challenge for variants of Kleene algebras at a Dagstuhl Seminar in 2001 [7], conjecturing that it *cannot* be proved in this setting, and he repeated this challenge at a DIMACS workshop [8].

Nevertheless, a proof in a variant of Kleene algebra was published in 2006 [19], but it is rather indirect and tedious. This is interesting, since statements of similar complexity could recently be proved fully automatically [14,15]. So, sharpening Cohen's challenge, can Bachmair and Dershowitz's termination theorem be proved *automatically* in variants of Kleene algebras?

This paper shows that this is indeed the case. But since the previous proof requires a series of lemmas and a direct proof from the axioms of Kleene algebras does not succeed[1], new ways must be explored. We therefore perform

[1] In all experiments, we used a Toshiba Tecra laptop under Linux with an Intel Pentium $1.73GHz$ processor with $6.5MB$ memory available.

proof experiments with the automated theorem prover SPASS [3] to "learn" the hypotheses needed for the proofs, by (manually) discarding computationally expensive axioms and by adding potentially useful lemmas in a rather random way. Hypothesis sets that are too weak can often be detected by a counterexample checker, e.g. Mace4 [2].

In the case of Bachmair and Dershowitz's termination theorem, a seemingly unrelated special unfold law for nested loops that has previously been automatically proved and used in the context of action system refinement is the key to success. With this law added to the set of hypotheses, SPASS returns a proof in less than $5min$. Retranslating this resolution proof into equational reasoning yields a fully formal proof of the termination theorem in essentially one line.

Moreover, a closer inspection of the equational proof reveals a novel refinement law for nested infinite loops, to which the termination theorem is a trivial corollary. This result is immediately applicable to concurrency control and action system refinement [5]. Since Kleene algebras are very abstract, the result holds in a variety of models relevant for programs and transition systems, including relations, traces, paths and languages.

Using this structural information, the automated proofs can easily be replayed in other variants of Kleene algebras to obtain termination and loop refinement theorems also in these settings.

The first variant—von Wright's demonic refinement algebras [21]—are appropriate for predicate transformer semantics of refinement, but not for relational models. The results obtained are comparable to the previous ones.

The second variant is based on the Kleene modules studied by Leiß [18] and by Ézik and Kuich [12]. We add a Park-style divergence operator that either models infinite iteration in the context of ω-regular languages or, in the context of discrete dynamical systems, that part of a state space from which infinite behaviour may arise. With these novel divergence modules, the proof of the loop refinement theorem can even be fully automated without any axiom restrictions and additional hypotheses in SPASS, i.e, without the loop unfold law mentioned.

The results obtained not only further confirm that variants of Kleene algebras in combination with off the shelf automated theorem provers are very useful as light-weight formal methods with heavy-weight automation for analysing programs and reactive systems. They also provide new structural insights related to Bachmair and Dershowitz's termination theorem, to action system refinement, to ω-regular languages and to discrete dynamical systems.

To make our experiment accessible and reproducible, all theory input files, all axiom restrictions, all additional hypotheses and all SPASS outputs are documented in an extended version [20]. Inputs will also be made available in TPTP-format at a web-site [1]. Since we are mainly interested in robust results for a formal methods context, we abstain from tuning the prover, avoid extreme running times and do not use extremely powerful hardware. In particular cases, stronger results can certainly be obtained by experts in theorem proving.

2 Kleene Algebras and Omega Algebras

Omega algebras provide an abstract axiomatisation of the objects and operations needed for specifying and proving Bachmair and Dershowitz's termination theorem. Relation algebras could be used as well, but omega algebras possess fewer operations and simpler axioms, which is beneficial for automated deduction. Omega algebras are simple extension of Kleene algebras that have recently emerged as foundational structures in computing.

An *idempotent semiring* is a structure $(S, +, \cdot, 0, 1)$ such that $(S, +, 0)$ is a commutative monoid with idempotent addition, $(S, \cdot, 1)$ is a monoid, multiplication distributes over addition from the left and right and 0 is a left and right zero of multiplication. Let S be a semiring. For all $x, y, z \in S$, the semiring axioms are

$$\begin{aligned} x + (y + z) &= (x + y) + z, \\ x + y &= y + x, \\ x + 0 &= x, \\ x(yz) &= (xy)z, \\ 1x &= x, \\ x1 &= x, \\ x(y + z) &= xy + xz, \\ (x + y)z &= xz + yc, \\ 0x &= 0, \\ x0 &= 0. \end{aligned}$$

As usual in algebra, we stipulate that multiplication binds more strongly than addition, and we omit the multiplication symbol. The relation $\leq$ defined by $x \leq y \Leftrightarrow x + y = x$ for all elements x, y is a partial order. Every idempotent semiring is therefore also a semilattice $(S, \leq)$ with addition as join and the following splitting law holds, which is very useful for automated deduction.

$$x \leq z \wedge y \leq z \Leftrightarrow x + y \leq z. \tag{1}$$

A *Kleene algebra* [17] is an idempotent semiring K extended by the star operation (or finite iteration operation) $^* : K \to K$ that satisfies, for all $x, y, z \in K$, the *star unfold* and *star induction* axioms

$$\begin{aligned} 1 + xx^* &\leq x^*, & 1 + x^*x &\leq x^* \\ z + xy \leq y &\Rightarrow x^*z \leq y, & z + yx \leq y &\Rightarrow zx^* \leq y. \end{aligned}$$

An *omega algebra* [6] is a Kleene algebra K extended by the omega operation (or infinite iteration operation) $^\omega : K \to K$ that satisfies, for all $x, y, z \in K$ the *omega unfold* and *omega coinduction* axiom

$$xx^\omega = x^\omega, \qquad y \leq z + xy \Rightarrow y \leq x^\omega + x^*z.$$

Kleene algebras have originally been conceived as algebras of regular events, i.e., to model the operations of addition (or union), multiplication (or concatenation) and star as they arise in language theory.

Kleene algebras also model actions (of a transition system). The constants 0 and 1 model the abortive and the ineffective action. Addition models nondeterministic choice of actions; it therefore has to be idempotent. Multiplication models the composition of actions. The star models the finite iteration of actions. The first star unfold axiom, e.g., says that a finite iteration x^* is either ineffective, whence 1, or it continues after one single x-action. By the first star induction law, x^* is the least element with that property. The omega models the strictly infinite iteration of actions. The omega unfold axiom says that prefixing actions x does not change an infinite iteration x^ω. The omega coinduction axiom implies that x^ω is the greatest element with that property; it also links finite and infinite iteration with respect to some "terminal action" z.

By the star and omega axioms, finite and infinite iteration is expressed within first-order logic with Park-style rules as least and greatest prefixed points (which are also least and greatest fixed points). Operationally, the induction axioms serve as star elimination rules at left-hand sides of equations, the coinduction axioms serve as omega elimination rules at their right-hand sides.

Encodings of omega algebras for theorem proving can be found in the research report [20] and at the web-site [1]. It follows from the definition of partial order on Kleene algebras that equational as well as order-based encodings can be used. Experience shows that the order-based encoding, although $\leq$ is treated as an ordinary predicate symbol for which no specialised inference rules are available, usually yields better results with more complex theorems. We therefore base all our arguments on the order-based encoding.

Some further facts are important for our considerations. First, the unfold axioms can be strengthened to the identities, $1 + xx^* = x^*$, $1 + x^*x = x^*$ and $x^\omega = xx^\omega$ Second, all operations are isotone with respect to the ordering $\leq$, i.e.,

$$x \leq y \Rightarrow x + z \leq y + z$$

and likewise for multiplication, star and omega. These properties are also used in the encoding of Kleene algebras for theorem proving.

3 Kleene Algebras and Abstract Reduction Systems

Kleene algebras have a rich model class that includes languages, sets of paths in a graph and sets of program traces. In the present context, however, relational models are our main interest.

So let $R = 2^{A \times A}$ denote the set of all binary relations over some set A. For all $r, s \in R$, let $r + s = r \cup s$, i.e., set union and let $r \cdot s = \{(a,b) | \exists c.(a,c) \in r \wedge (c,b) \in s\})$, i.e., the relational product of r and s. Let $0 = \emptyset$ be the empty relation and let $1 = \{(a,a) | a \in A\}$ be the unit relation. Finally, let $r^* = \bigcup_{i \geq 0} r^i$, where $r^0 = 1$ and $r^{i+1} = r \cdot r^i$. It is easy to see that r^* models the reflexive transitive closure of r. The following theorem is well-known and easy to verify.

Theorem 3.1. *$(R, +, \cdot, 0, 1, ^*)$ is a Kleene algebra.*

It is often called the *full relation Kleene algebra* over A. Obviously, R is its maximal element. It follows from basic results of universal algebra that every subalgebra of the full relation Kleene algebra is again a Kleene algebra. See, e.g., [10] for a discussion.

Now each full relation Kleene algebra is complete and, by the Knaster-Tarski theorem, the greatest fixed point of the function $\lambda y.z + xy$ exists and is equal to $x^\omega + x^* z$.

Theorem 3.2. *$(R, +, \cdot, 0, 1, ^*, ^\omega)$ is an omega algebra.*

Note, however, that x^ω is not necessarily equal to an iteration $\bigcap_{i\geq 0} x^i \cdot R$, since this would presuppose distributivity of multiplication over arbitrary infima. Nevertheless, $x^\omega = \bigcap_{i\geq 0} x^i \cdot R$ holds whenever A is finite. Finally, the following result has been shown (cf. [13] for details).

Proposition 3.3. *In every (full) relation semiring, $r^\omega = 0$ if and only if there are no infinitely ascending r-chains, that is, if and only if r terminates.*

The analysis of Bachmair and Dershowitz's termination theorem is entirely based on abstract reduction systems, i.e., it disregards the subterm property which is present in concrete term rewrite systems. Formally, an abstract reduction system is a family r_i of binary relations on some set A. Every abstract reduction system can therefore be embedded into the full relation omega algebra on A.

Corollary 3.4. *Let R be an abstract reduction system. Then $(R, +, \cdot, 0, 1, ^*, ^\omega)$, with the operations defined as before, is a relation omega algebra.*

Corollary 3.4 and Proposition 3.3 yield the general justification that termination properties of abstract reduction systems can be analysed in terms of omega algebras.

4 First Proof

Based on the general results from Section 3, we can now abstract the notions of quasicommutation and termination, and the statement of the separation theorem in omega algebra.

We assume that rewrite systems x and y are elements of some omega algebra. This is reasonable, since rewrite systems—more precisely abstract reduction systems—are relations, and since relations under union, composition, reflexive transitive closure and infinite iteration together with the empty relation and the unit relation form an omega algebra.

For specifying Bachmair and Dershowitz's termination theorem, two notions are essential. Let x and y be elements of some omega algebra. Then

- x *quasicommutes* over y if $yx \leq x(x + y)^*$;
- x *terminates* if $x^\omega = 0$.

Termination as absence of infinite x-chains has been used by Bachmair and Dershowitz. The termination theorem can now be rephrased as follows.

Theorem 4.1. *Let x and y be elements of some omega algebra and let x quasicommute over y.*

$$(x+y)^\omega = 0 \Leftrightarrow x^\omega + y^\omega = 0.$$

It is well known that the right-to-left direction does not require quasicommutation. It can be proved with SPASS in less than $0.13s$. The SPASS output of this and all other proofs together with detailed information about restrictions on the axiom set and additional hypotheses used can be found in the technical report [20]. The information provided in the report will allow readers to replay all proofs. In this particular case, no restrictions on the axiom set and no additional hypotheses are needed.

A proof of the left-to-right direction of Theorem 4.1 in a full sweep is impossible with the hardware available. Experimenting with different hypothesis sets, as mentioned in the introduction, we can find a proof from a restricted axiom set and with additional hypotheses in about $4min$. The key to success is the law

$$(x+y)^\omega = y^\omega + y^* x(x+y)^\omega, \tag{2}$$

which has previously been automatically verified and used in the context of program refinement [14].

An equational proof can be reconstructed from the resolution proofs. For its presentation, the following property is worth mentioning.

$$yx \le x(x+y)^* \Leftrightarrow y^* x \le x(x+y)^*. \tag{3}$$

The right-to-left direction can be proved in $0.33s$ without any restrictions. The right-to-left direction took about $27s$ from a reduced axiom set and an addition hypothesis.

The equational proof of Theorem 4.1 is then a one-liner:

Proof. (of Theorem 4.1)

$$(x+y)^\omega = y^\omega + y^* x(x+y)^\omega \le y^\omega + x(x+y)^*(x+y)^\omega = y^\omega + x(x+y)^\omega.$$

The first step is by Equation (2); the second step by quasicommutation and Equation (3); the third step uses the identity $x^* x^\omega = x^\omega$. Then

$$(x+y)^\omega \le x^\omega + x^* y^\omega \tag{4}$$

follows by omega coinduction. Now $(x+y)^\omega = 0$ immediately follows from $x^\omega = 0$ and $y^\omega = 0$.

The converse direction follows—without quasicommutation—from $x \le x+y$, $y \le x+y$, isotonicity of omega and the fact that $z^\omega z^\omega = z^\omega$. □

This results, obtained from experimenting with SPASS, is certainly surprising. It is much simpler and much more direct than the previous proof in [19] and its circumstantial mechanisation with Prover9 in [14]. However, from the puristic point of view, it is still not satisfactory since it relies on axiom restrictions and an additional lemma.

5 A Novel Loop Refinement Law

A closer look at the equational proof of Theorem 4.1 reveals Equation (4)—a refinement law for infinite loops in the presence of quasicommutation—which is interesting in its own right and of which the termination theorem turns out to be just a special case.

Theorem 5.1. *Let x and y be elements of some omega algebra and let x quasi-commute over y. Then*

$$(x + y)^\omega = x^\omega + x^* y^\omega. \tag{5}$$

Proof. For $(x + y)^\omega \leq x^\omega + x^* y^\omega$ replay the proof of Theorem 4.1 to equation (4). This direction depends on quasicommutation.

The converse direction follows from $x \leq x+y$, $y \leq x+y$, isotonicity of omega, $x^* x^\omega = x^\omega$ and the fact that $z^\omega z^\omega = z^\omega$. □

The left-to-right direction could be proved in $13s$ from a restricted axiom set and with additional hypotheses. The right-to-left direction could be proved in $13min35s$ without any restrictions.

Intuitively, Equation (5) says that a strictly infinite repetition of actions x or y chosen non-deterministically can be separated into the non-derministic execution of a strictly infinite repetition of x-actions or a finite (possibly empty) repetition of x-actions followed by a strictly infinite repetition of y-actions.

The assumption of quasicommutation is quite general; it is implied by other notions of commutation like $ba \leq a^+ b^*$, where $a^+ = aa^*$, $ba \leq ab$ or $ba = ab$. All these conditions model meaningful properties of systems: inequalities typically model preference or priority properties whereas equations model independence properties.

Theorem 4.1 now follows from Theorem 5.1 by setting $x^\omega = 0 = y^\omega$. The proof with SPASS takes $0.04s$ without any restrictions or additional hypotheses.

6 Demonic Refinement Algebras

We now provide an alternative proof of a variant of the above loop refinement theorem and of Bachmair and Dershowitz's termination theorem in another variant of Kleene algebra called *demonic refinement algebra* [21]. Formally, these are structures $(K, ^\infty)$ such that K is a Kleene algebra without the right zero axiom and the operation $^\infty$ of *strong iteration* is axiomatised by the *strong unfold*, the *strong coinduction* and the *isolation* axiom

$$x^\infty = 1 + xx^\infty, \qquad y \leq z + xy \Rightarrow y \leq x^\infty z, \qquad x^\infty = x^* + x^\infty 0$$

for all $x, y, z \in K$. The converse strong unfold law, $1 + x^\infty x = x^\infty$, follows from the axioms and strong iteration is isotone with respect to the ordering.

The particular axioms of demonic refinement algebra can easily be motivated from the predicate transformer model of refinement with infinite iteration, cf. [21]. It has been shown in Back and von Wright's refinement calculus [5] that

$x^\infty = x^\omega + x^*$. The same proof trivially holds in demonic refinement algebra. Therefore, strong iteration comprises finite and strictly infinite iteration. It is also immediately obvious that demonic refinement algebras do not capture the relational semantics of programs, since in the presence of the right zero axiom (which is satisfied in relational models), the isolation axiom collapses strong iteration to finite iteration. Therefore, the results of the following section are not directly related to Bachmair and Dershowitz's termination theorem. But as refinement theorems within the refinement calculus they are certainly interesting in their own right.

Also, all theorems of demonic refinement algebra that do not mention strong iterations are also theorems of Kleene algebra.

The code for demonic refinement algebras in SPASS can again be found in the research report [20].

7 Second Proof

In the context of demonic refinement algebras, quasicommutation can of course be written as before. But there are now two different notions of termination. We say that

- x *weakly terminates* if $x^\infty = x^*$;
- x *strongly terminates* if $x^\infty 0 = 0$.

Lemma 7.1. *In every demonic refinement algebra, strong termination implies weak termination, but the converse need not hold.*

The implication of weak termination by strong termination can be shown with SPASS in less than $0.06s$ without any restrictions or additional hypotheses. For the converse direction, the counterexample generator Mace4 [2] finds a counterexample with three elements. It is presented in the research report [20].

It has already been shown automatically that a variant of Equation (2) holds in demonic refinement algebra [1].

$$(x + y)^\infty = y^\infty + y^* x(x + y)^\infty. \tag{6}$$

Moreover, $x^* x^\infty = x^\infty$, so that—up to the coinduction step—the equational proof of Theorem 5.1 can be translated into demonic refinement algebra. With the strong coinduction law as a last step, we then obtain the following loop refinement law.

Theorem 7.2. *Let x and y be elements of some demonic refinement algebra and let x quasicommute over y. Then*

$$(x + y)^\infty = x^\infty y^\infty. \tag{7}$$

The right-to-left direction follows immediately from isotonicity and the identity $x^\infty x^\infty = x^\infty$. The proof from left to right with SPASS takes $0.48s$. It reuses the information of the corresponding proof of Theorem 5.1, i.e., a restricted set of axioms and additional hypotheses. The right-to-left proof takes $11s$. It also uses a restricted set of axioms and additional hypotheses.

Due to the two variants of termination, we now obtain two variants of the termination theorem as corollaries.

Theorem 7.3. *Let x and y be elements of some demonic refinement algebra and let x quasicommute over y. Then*

(i) $x^\infty = x^* \wedge y^\infty = y^* \Rightarrow (x+y)^\infty = (x+y)^*$;
(ii) $x^\infty 0 + y^\infty 0 = 0 \Rightarrow (x+y)^\infty 0 = 0$.

For (i), to show that $(x+y)^\infty \leq (x+y)^*$ follows from the hypotheses takes $13s$. It uses a restricted axiom set and an additional hypothesis. $(x+y)^* \leq (x+y)^\infty$ can be proved in $0.04s$ without any restrictions or additional hypotheses. The proof of (ii) takes $0.05s$, again without any restrictions or additions.

For the converse direction, we can prove the following statement.

Theorem 7.4. *Let x and y be elements of some demonic refinement algebra. Then*

$$(x+y)^\infty 0 = 0 \Rightarrow x^\infty 0 = 0 \wedge y^\infty 0 = 0.$$

This can be proved in $1min3s$ without any restrictions or additions.

However, a similar statement for strong termination does not hold. Mace4 yields a counterexample with three elements, which is presented in the research report [20].

8 Kleene Modules

We will now show how notions of divergence and termination can be specified in the setting of Kleene modules. Such structures were first studied by Ésik and Kuich [12] and by Leiß [18]. However, an operator for modelling program divergence as a Park-style fixed point operator has, to our knowledge, not yet been given.

Divergence and termination have already been investigated in the context of modal Kleene algebras [10,9]. Every modal Kleene algebra is also a Kleene module, but not vice versa [11]. So it is necessary to reconsider the notions of termination and divergence. We study them in this more general setting because it simplifies automated deduction and provides some structural insights.

A *Kleene module* [18] $(K, L, :)$ is a two-sorted structure of a Kleene algebra K, a semilattice $L = (L, +, 0)$ with zero and a scalar product $:$ from $K \times L$ to L that satisfies the following axioms.

$$(x+y)p = xp + yp, \qquad x(p+q) = xp + xq,$$
$$(xy)p = x(yp), \qquad 1p = p, \qquad x0 = 0,$$
$$xp + q \leq r \Rightarrow x^* q \leq r,$$

for all $x, y \in K$ and $p, q, r \in L$. We usually omit the scalar product symbol.

A *divergence module* $(K, L, :, ^\triangledown)$ is a structure such that $(K, L, :)$ is a Kleene module and $^\triangledown : K \to L$ a mapping that satisfies the *divergence-unfold* and the *divergence-coinduction* axioms

$$x^\triangledown \leq xx^\triangledown, \qquad p \leq xp + q \Rightarrow p \leq x^\triangledown + x^* q,$$

for all $x \in K$ and $p, q \in L$. Previously, the notation $\nabla(x)$ has been used to denote the divergence of x [9]. Here, we use $x^\triangledown$ to emphasise the similarity to the omega and the strong divergence operator.

This novel definition of divergence modules is very general; it admits at least two interesting interpretations.

Under the first interpretation, xp models the preimage of a set p under a relation x and $x^\triangledown$ models the set of all states from which infinite x-sequences may emanate. This divergence set is stable under x-actions and it is the greatest set with that property (0 being the lest such set). Then x *terminates* if $x^\triangledown = 0$. This definition is consistent with the standard set-theoretic notion of Noethericity. In set theory $p - xp$ models the set of x-maximal elements of p, i.e., the set of those elements from which no further x-action is possible. Now $p - xp = 0$, which is equivalent to $p \leq xp$, says that p has no x-maximal elements. Then, if x is Noetherian, the empty set 0 is the only element with that property; whence $p \leq ap \Rightarrow p = 0$. See [9] for further discussion in the setting of modal Kleene algebras.

Under the second interpretation, elements of K model finite computations or actions of a program whereas elements of L model infinite ones. The scalar product relates finite and infinite computations in a reasonable way that makes it impossible to compose an infinite element at its right-hand side with any other element. In this setting, the divergence operation maps finite elements to infinite ones. The divergence axioms are precisely typed (or sorted) variants of the unfold and coinduction axioms of omega algebra. So divergence acts as the appropriate omega operator under this interpretation and therefore, $x^\triangledown = 0$ means again absence of infinite iteration.

The two interpretations of omega make this operation very versatile and applicable in different contexts. Beyond modal reasoning, the first interpretation is interesting for the analysis of infinite behaviours in discrete dynamical systems. The second one is more compatible with the definition of ω-regular languages than omega algebra. It seems challenging to obtain a completeness theorem with respect to ω-regular languages from this setting.

Since here, relational models are again admitted, divergence modules capture again Bachmair and Dershowitz's termination theorem. The correspondence between divergence and termination is even more transparent than for omega algebra. A discussion of the general correspondence between divergence and omega (for modal Kleene algebras) can be found in [13].

As a special case, the carriers of K and L can be the same and $^\triangledown$ becomes an endomorphism. This immediately yields the following fact.

Proposition 8.1. *Every theorem of divergence modules is a theorem of omega algebra (modulo translation).*

The converse direction does, of course, not hold. $x^\omega 0 = 0$ holds in omega algebra for every element x, but a corresponding identity cannot even be written down in divergence modules.

It follows immediately from $(x+y)p = xp + yp$, from $x(p+q) = xp + xq$ and from the definition of the partial ordering that scalar products are isotone in both arguments, i.e.,

$$x \leq y \Rightarrow xp \leq yp \qquad \text{and} \qquad p \leq q \Rightarrow xq \leq xq.$$

This can easily be proved from an equational encoding of divergence modules in SPASS. We will add these properties together with the other isotonicity laws to the prover input files. Since the equational encoding is of no further interest, we neither document this encoding nor the proofs in this paper. An order-based encoding of divergence modules in SPASS can be found in the research report [20].

9 Third Proof and Full Automation

The proofs of variants of the loop refinement theorem and of Bachmair and Dershowitz's termination theorem in Kleene modules is particularly simple and can therefore be automated without any restrictions or additional hypotheses.

Analogously to the previous sections, we can prove a variant of the special unfold law for divergence modules. Since it has not yet been considered, we present it as a lemma.

Lemma 9.1. *Let x and y be elements of some divergence module. Then*

$$(x+y)^\triangledown = y^\triangledown + y^* x (x+y)^\triangledown. \tag{8}$$

The left-to-right direction takes $1min25s$; its converse $2min53s$. Both directions are proved from the full axiom set and need no additional hypotheses. This law is interesting in its own right as a refinement law, but we will not need it in further proofs.

The next statement is an analogue to the loop refinement laws Theorem 5.1 and Theorem 7.2 that hold in omega algebras and demonic refinement algebras. We display an equational proof to demonstrate that it is precisely along the lines of omega algebras.

Theorem 9.2. *Let x and y be elements of some divergence module and let x quasicommute over y. Then*

$$(x+y)^\triangledown = x^\triangledown + x^* y^\triangledown.$$

Proof. We calculate

$$(x+y)^\triangledown = y^\triangledown + y^* x (x+y)^\triangledown \leq y^\triangledown + x(x+y)^*(x+y)^\triangledown = y^\triangledown + x(x+y)^\triangledown.$$

The claim then follows from divergence-coinduction. The identity $x^\triangledown = x^* x^\triangledown$ can easily be verified. □

In contrast to previous approaches, this statement can now be proved without any restrictions on the axiom set and without any additional hypotheses with SPASS. The left-to right direction that uses quasicommutation takes $1min51s$. Its converse takes $3min6s$.

The fact that proof automation is particularly simple in divergence modules might at first sight seem surprising, since the axiom set of divergence modules is more complex than those of omega algebras and demonic refinement algebras. However, in the two-sorted setting, operations are applied to terms in a more restrictive way, especially the computationally expensive rearrangements due to associativity and commutativity of addition and to the fixed-point laws for finite and infinite iterations that allow self-substitutions can be better controlled. This certainly explains the success of SPASS which can manage sorts in an efficient way.

Theorem 9.2 is quite general and admits many different interpretations beyond rewrite systems. Under the modal interpretation, since the divergence $x^\triangledown$ models the basin of non-termination of x in the state space L, these basins of non-termination can be separated by Theorem 9.2. This is certainly relevant to the analysis of discrete dynamical systems.

Under the interpretation with finite and infinite actions, it models again loop separation, which is interesting for program verification.

A third variant of Bachmair and Dershowitz's termination theorem now follows as a corollary, as before.

Theorem 9.3. *Let x and y be elements of some divergence module and let x quasicommute over y. Then*

$$(x + y)^\triangledown = 0 \Leftrightarrow x^\triangledown + y^\triangledown = 0.$$

The left-to-right direction takes $3min9s$; its converse, assuming Theorem 9.2, $0.11s$. The proofs are again obtained from the full axiom set without additional hypotheses.

Since every modal Kleene algebra is a Kleene module, our results holds a fortiori in the former setting. The relationship between the two approaches can intuitively be described as follows. First, instead of defining Kleene modules over a semilattice, we could use a Boolean algebra in the second component. The resulting structures have been investigated in [11]; they are strongly related to dynamic algebras, which are algebraic variants of propositional dynamic logics, and they are Boolean algebras with operators in the sense of Jónsson and Tarski [16]. Second, to obtain modal Kleene algebras, the Boolean algebra can be embedded into the subalgebra bounded by 0 and 1 of the Kleene algebra such that mixed terms between Kleenean and Boolean elements can be written down. The axiom set for modal operators can then be reduced to three simple equations. The precise connection has been set up in [11].

However, due to the more complex axiomatisation of dynamic logics, Boolean algebras with operators and modal Kleene algebras, it cannot be expected to obtain a similar degree of automation. Moreover, due to the abstractness and universality of variants of Kleene algebras, our results hold in models including relations, program traces, paths and languages.

10 Conclusion

We solved a sharpened variant of Cohen's challenge by proving Bachmair and Dershowitz's termination theorem mechanically in variants of Kleene algebras and, in particular, fully automatically in the setting of divergence modules. Through our proof experiments that involve hypothesis selection, we found particularly simple proofs that could be retranslated into fully formal equational proofs with essentially one line of calculation. This is in sharp contrast to the original argument by Bachmair and Dershowitz, a formalisation of which would certainly require several pages, and which seems infeasible to automation.

The concise formalism of Kleene algebras and the discipline of proof enforced in this setting also revealed some structural insight in the setting of Bachmair and Dershowitz's theorem. Through the equational proof we discovered a new refinement theorem for nested infinite loops to which the termination theorem is a simple corollary.

Using this structural insight, we replayed our proofs in further variants of Kleene algebras and were particularly successful in the newly developed setting of divergence modules.

The simple treatment of the termination theorem in the context of Kleene algebras is based, of course, on a significant amount of abstraction. The formalisation gap between concrete rewrite systems and Kleene algebras is, however, closed once and for all by the well-known proofs that abstract reduction systems form omega algebras or divergence modules. The proofs obtained are short relative to that abstraction.

When starting our proof experiments, we used McCune's Prover9 [2], but then moved to SPASS when proofs in the two-sorted setting of divergence modules seemed infeasible. We then replayed all proofs for omega algebras and demonic refinement algebras with SPASS, to be able to present more coherent results. Since we do not want to overload this paper, we do not present a comparison of Prover9 and SPASS. Let us only mention that for omega algebras and demonic refinement algebras they were comparable.

¿From a more general point of view, this work contributes to a series of papers devoted to the automation of first-order algebraic structures with applications in program development, refinement and verification [14,15]. Our results suggest that the combination of off the shelf automated theorem proving with domain-specific algebras has considerable potential to further establish first-order theorem proving as a feasible alternative to model checking and interactive proof assistants. Due to the abstraction and universality provided by the algebras, we believe that light-weight formal methods with heavy-weight automation can be obtained.

This line of work also leads to interesting research questions in automated deduction. A first strand is the integration and implementation of solvers and decision procedures for concrete data types as they arise in verification scenarios, e.g., arithmetics, lists, queues, arrays in automated theorem provers. A second strand is the implementation of order-based reasoning through ordered chaining calculi. Order-based reasoning often highly advantageous for automated algebraic proofs but rather neglected by the theorem proving community. A third strand is the development of focused inference rules for the algebras under consideration, which would further help to guide proof search and allow one to prover relevant theorems of even greater complexity. Finally our algebraic approach provides challenging benchmarks for first-order theorem provers that are both computationally hard and practically relevant. We will therefore make all inputs available in TPTP-format [1].

While preparing the final version of this paper, Peter Höfner and the author were even able to push some of the results from this paper a step further. Bachmair and Dershowitz's termination theorem (Theorem 4.1) could now be proved in some minutes; the loop refinement theorem in demonic refinement algebra (Theorem 7.2) could be proved in a couple of hours without any axiom restrictions or additional lemmas. These unexpected results were obtained by running Prover9 with an *equational* axiomatisation of omega algebras and demonic refinement algebras. These results are documented at the web-site [1]. They are in contrast to our previous experience that an order-based approach work better with more complex theorems. Further consideration of these novel results and a comparison between different approaches is planned for an extended version of this paper.

References

1. `http://www.dcs.shef.ac.uk/~georg/ka`.
2. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9`.
3. Spass 3.0. `http://spass.mpi-inf.mpg.de/`.
4. L. Bachmair and N. Dershowitz. Commutation, transformation, and termination. In J. H. Siekmann, editor, *8th International Conference on Automated Deduction*, volume 230 of *LNCS*, pages 5–20. Springer, 1986.
5. R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer, 1998.
6. E. Cohen. Separation and reduction. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction (MPC 2000)*, volume 1837 of *LNCS*, pages 45–59. Springer, 2000.
7. E. Cohen. Omega algebra: the good, the bad, and the ugly. In R. Backhouse, D. Kozen, and B. Möller, editors, *Applications of Kleene Algebra, Report of the Dagstuhl Seminar 01081*, page 5, 2001.
8. E. Cohen. Some open problems in Kleene and omega algebras. DIMACS Workshop on Applications of Lattices and Ordered Sets to Computer Science, 2003. `http://dimacs.rutgers.edu/Workshops/Lattices/`.
9. J. Desharnais, B. Möller, and G. Struth. Termination in modal Kleene algebra. In Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell, editors, *IFIP TCS2004*, pages 647–660. Kluwer, 2004. Revised version: *Algebraic Notions of Termination.* Technical Report 2006-23, Institut für Informatik, Universität Augsburg, 2006.

10. J. Desharnais, B. Möller, and G. Struth. Kleene algebra with domain. *ACM Trans. Computational Logic*, 7(4):798–833, 2006.
11. T. Ehm, B. Möller, and G. Struth. Kleene modules. In R. Berghammer, B. Möller, and G. Struth, editors, *Relational and Kleene-Algebraic Methods in Computer Science*, volume 3051 of *LNCS*, pages 112–123. Springer, 2004.
12. Z. Ésik and W. Kuich. A semiring-semimodule generalization of ω-context-free languages. In J. Karhumäki, H. Maurer, G. Păun, and G. Rozenberg, editors, *Theory is Forever*, volume 3113 of *LNCS*, pages 68–80. Springer, 2004.
13. P. Höfner and G. Struth. Algebraic notions of non-termination. Technical Report CS-06-12, Department of Computer Science, University of Sheffield, 2006.
14. P. Höfner and G. Struth. Automated reasoning in Kleene algebra. In F. Pfenning, editor, *CADE 2007*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 279–294. Springer, 2007.
15. P. Höfner and G. Struth. Can refinement be automated? In E. Boiten, J. Derrick, and G. Smith, editors, *International Refinement Workshop—Refine 2007*, ENTCS, 2007. (To appear.).
16. B. Jónsson and A. Tarski. Boolean algebras with operators, Part I. *American Journal of Mathematics*, 73:891–939, 1951.
17. D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.
18. H. Leiß. Kleene modules and linear languages. *Journal of Logic and Algebraic Programming*, 66(2):185–194, 2006.
19. G. Struth. Abstract abstract reduction. *Journal of Logic and Algebraic Programming*, 66(2):239–270, 2006.
20. G. Struth. Reasoning automatically about termination and refinement. Technical Report CS-07-10, Department of Computer Science, University of Sheffield, 2007.
21. J. von Wright. Towards a refinement algebra. *Science of Computer Programming*, 51(1-2):23–45, 2004.

`ProofBuilder`, an Interactive Prover for Students, with Extensive Capabilities

Hugh McGuire

Grand Valley State University, Allendale, MI 49401, U.S.A.
`mcguire@cis.gvsu.edu` `http://www.cis.gvsu.edu/~mcguire/`

Abstract. `ProofBuilder` is an interactive first-order theorem-proving system. It is designed to help students learn to construct proofs by structuring proofs and their construction clearly and performing menial or error-prone symbolic manipulations, so students can concentrate on the higher-level process of deduction. This system is further designed to be usable as widely as possible by allowing different forms of input and providing a variety of popular proof methods.

1 Introduction

This paper presents an interactive first-order theorem-proving system named `ProofBuilder`, which is designed to enable student users to do basic proofs such as those for a course in Discrete Mathematics. As with other systems, `ProofBuilder`'s capabilities include rewriting for logic operators, simplification, elimination of quantifiers, substitution of equivalents or equals, and stepwise and strong induction; but what distinguishes `ProofBuilder` is that it is designed to enable users to prove theorems like the way they prove theorems manually. Such aspects of the design include a framework that clarifies which formulas in a proof are premises that are hypothesized to be true, and which formulas are goals that need to be proven; and mechanisms to apply a variety of proof methods and strategies such as forward and backward reasoning, proving by contradiction, transforming one side of an equation or equivalence to the other, and proving by cases. `ProofBuilder` is intentionally interactive — not completely automated — for pedagogical reasons: automated systems enable students to avoid the intellectual deductive work of doing proofs. But though it doesn't do all the deductive work for students, `ProofBuilder` does help them learn to construct proofs by: (i) structuring proofs and their construction clearly, avoiding confusion; (ii) showing in menus the variety of options for deductive steps; (iii) performing menial or error-prone symbolic manipulations, e.g. substituting "0" and "n+1" where appropriate when applying stepwise induction; and (iv) enforcing soundness of deductive steps.

`ProofBuilder` is written in Java, so it is usable on essentially all common modern platforms (Microsoft Windows, LINUX, Apple Macintosh, UNIX, etc.); and it has a graphical user interface, including capabilities for copying and pasting formulas from other convenient sources such as Web pages; and it uses proper mathematical characters such as "$\leq$", "$\forall$", "$\subseteq$", and "$\sum$".

2 Related Work

For one recent survey of the field of theorem-proving systems, see [1]. Regarding theorem-proving systems that are designed for educational purposes, see the *Journal of Automated Reasoning*, Vol.32 (2004) No.3 (February), which was a special issue involving education. But most theorem-proving systems have the explicit goal of automating deduction: they are cleverly designed to do as much deduction as possible automatically, working as hard as possible to minimize the amount of deduction that users must do — even if the users may be students learning to do proofs. Further, their interfaces are minimally GUI: they're really designed to 'grind' through logic formulas (which are treated like data) to automatically generate proofs, rather than to just assist users in learning to achieve construction of proofs themselves much.

Some systems such as `Omega` and `Mizar` can be used interactively and have been used in educational settings. [2,3] But consider the following:

> First, instead of supporting the language the mathematician is used to, most systems impose their own formal language on the user and require a machine-oriented formalization of the mathematical content to allow for powerful automatic inference capabilities. As a result, the line of reasoning is often unnatural and obscured. Next, the proofs are at a level of excruciating detail spelling out many logically necessary steps, which a human would nevertheless consider trivial or obvious. [4]

For example, here is part of some material for `Mizar`:

```
for k being Nat holds k + 0 = k;
```

And here is part of some material for Isabelle/Isar(/HOL):

> let ?k = n! + 1
> obtain p where prime: p ∈ prime and dvd: p dvd ?k
> using prime-factor-exists by auto
>
> [5]

By contrast, `ProofBuilder` uses more standard mathematical notation than those systems, and it involves less 'machinery' — neither libraries[1] nor automatic reasoning. Again, the purpose of `ProofBuilder` is to provide simply an environment for students to learn to construct formal proofs, as indicated in textbooks for teaching Discrete Mathematics.

There are some other systems that are supposed to help students learn to construct proofs, e.g. [6] and [7]. But those systems are designed more for checking proofs that students enter rather than really helping them construct proofs by performing logical steps that they select; and the installations are platform dependent and not very standalone, involving the automated theorem provers Otter [8] or Isabelle [9], respectively, in the background.

Some other related systems are "Deductive Tableau" [10] and "Deduct" [2], but those two systems limited their scope to material of [11]; and perhaps more significantly, the original authors of those systems no longer support them. `ProofBuilder`, presented here, may be construed as developing those latter two systems further: enabling users to avoid tedious 'low-level' steps of [11]'s formal scheme (such as applying associativity to the formula "$\neg\mathtt{P} \vee (\mathtt{P} \vee \mathtt{Q})$" to simplify it to "`true`"); providing more proper mathematical notation such as "∀" and "≤" instead of "`forall`" and "<="; handling more topics such as sets, summations, and `O`(); and formally providing more proof methods/strategies such as proving by contradiction or by cases, and transforming one side of an equation to the other.

Some other pieces of software are pedagogical or used pedagogically in contexts where students learn to construct proofs, but these pieces of software aren't really designed for general construction of proofs. For example, Maple [12] is referenced in textbooks such as [13] and [14], and the latter also references Mathematica [15]; there are applets etc. illustrating algorithms [16]; and [17] has associated Flash software enabling students to interactively work on piecing proofs together. But that proof-constructing work appears to be either merely illustrative, algebraic, or prefabricated. For example, with [17] users form proofs simply by putting several lines of prewritten text in proper order.

Incidentally, this work is not related to other software named "ProofBuilder" such as typographical software nor even the work of Brauner et al.[3]

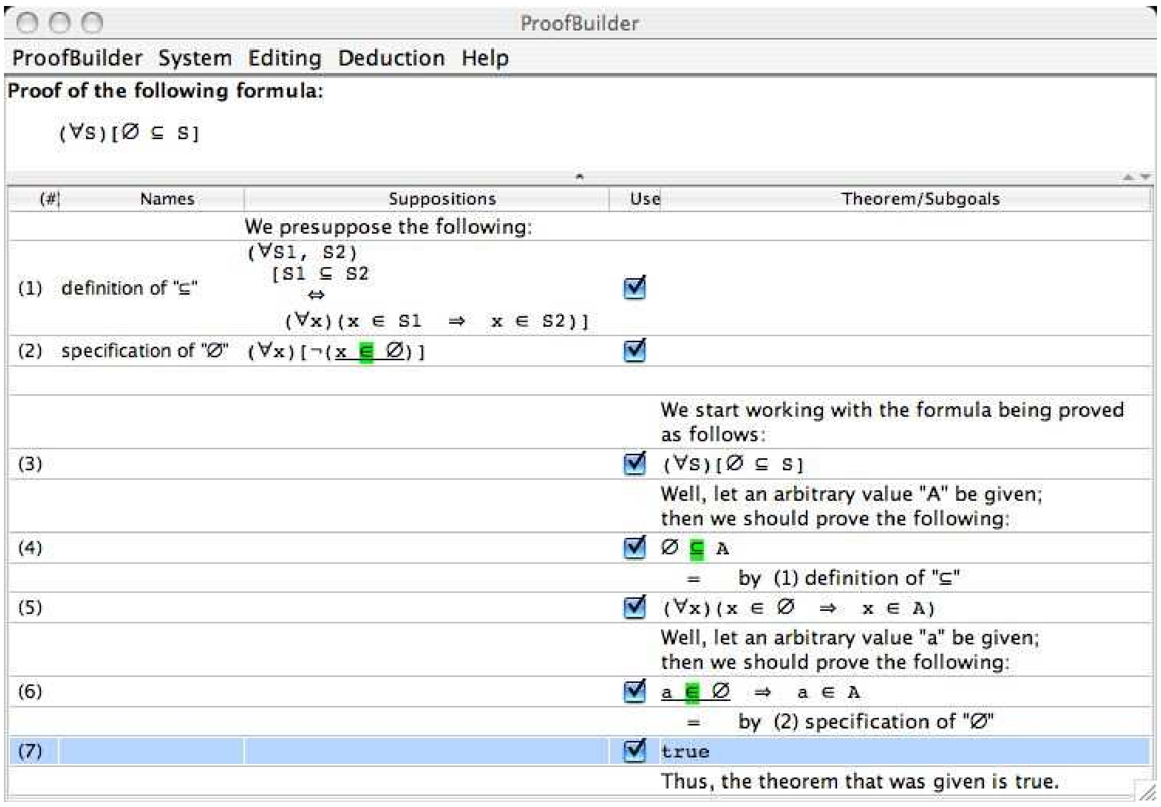
3 Illustrations of Operating `ProofBuilder`

To expedite the presentation of `ProofBuilder`, illustrations of it are used. Here is a first sample proof produced in `ProofBuilder`:

[1] "The size of the `Mizar` library is about 50 megabytes, while that of Isabelle is about 10 megabytes (including the sources of the system and the example theories)." [5]

[2] The "Deduct" software, by Michael Colón et al., has not been published, but it has been available upon request from the REACT research group under the supervision of Zohar Manna at Stanford University.

[3] Not yet published at this time.



Proof of the following formula:

(∀S)[∅ ⊆ S]

(#)	Names	Suppositions	Use	Theorem/Subgoals
		We presuppose the following:		
(1)	definition of "⊆"	(∀S1, S2) [S1 ⊆ S2 ⇔ (∀x)(x ∈ S1 ⇒ x ∈ S2)]	✓	
(2)	specification of "∅"	(∀x)[¬(x ∈ ∅)]	✓	
				We start working with the formula being proved as follows:
(3)			✓	(∀S)[∅ ⊆ S]
				Well, let an arbitrary value "A" be given; then we should prove the following:
(4)			✓	∅ ⊆ A
				= by (1) definition of "⊆"
(5)			✓	(∀x)(x ∈ ∅ ⇒ x ∈ A)
				Well, let an arbitrary value "a" be given; then we should prove the following:
(6)			✓	a ∈ ∅ ⇒ a ∈ A
				= by (2) specification of "∅"
(7)			✓	true
				Thus, the theorem that was given is true.

ProofBuilder extends the deductive-tableau proof scheme of [11]. As shown in the sample proof above, ProofBuilder constructs a proof in a table with two main columns labeled "Suppositions" and "Theorem/Subgoals" containing the logic formulas used in the proof, a column labeled "Used" containing checkboxes indicating whether formulas have been used yet in a proof, and two columns labeled "(#)" and "Names" containing numbers and names for identifying the logic formulas. And in addition to the formulas, there are also narrative texts (and blank lines) which serve to clarify the construction of the proof.

Construction of a proof in this system begins with the user entering the theorem to be proved as the first formula in the "Theorem/Subgoals" column. If desired/needed, the user can also enter axioms or lemmas as initial entries in the "Suppositions" column. For example, in the sample proof above the theorem is in the row numbered (3), and a couple of basic definitions or axioms which are used in this proof are in the rows numbered (1) and (2). Then, the user applies deductive steps — selected from the menu labeled "Deduction" — to formulas in the deductive tableau, generating additional formulas which are added to the deductive tableau, until a deductive step achieves the proof-terminating 'subgoal' of `true` (or a 'supposition' of `false`, when proving by contradiction). See Section 4 below for further details about operating ProofBuilder, including options for different symbols such as "⇒"/"→"/"`implies`"/"`IMPLIES`".

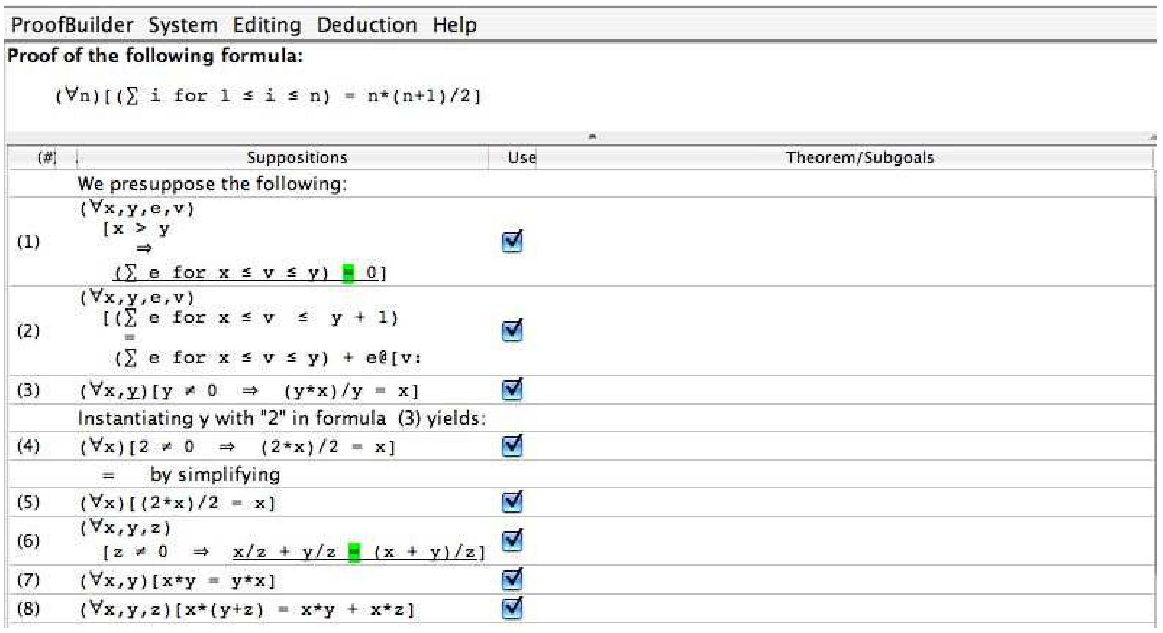
The first deductive step illustrated in the sample proof above is Quantifier Removal, which is applied to the theorem formula (3), $(\forall \mathtt{S})[\varnothing \subseteq \mathtt{S}]$, yielding the formula (4), $\varnothing \subseteq \mathtt{A}$, which is added as a new goal formula to be proved; at this point ProofBuilder marks formula (3) as used, for clarity. As shown, ProofBuilder provides 'canned' narrative text for deductive steps; naturally, these texts can be changed. Proving the validity of such a derived goal (4) would prove the validity of the original universally quantified theorem formula (3) as with classical Hilbert-style logic's deductive step of universal generalization, which specifies that proving a formula φ containing a constant symbol `c` suffices to prove the formula $(\forall \mathtt{x})\hat{\varphi}$, where $\hat{\varphi}$ is obtained from φ by replacing occurrences of `c` with `x` (under some conditions). ProofBuilder does Quantifier Removal also implicitly/automatically when handling supposition formulas that are universal quantifications. For example, in the sample proof above the supposition formula (2), $(\forall \mathtt{x})[\neg(\mathtt{x} \in \varnothing)]$, is handled as "$\neg(\mathtt{x} \in \varnothing)$". This variant of Quantifier Removal is like quantifier removal when converting formulas to clausal form. See the documentation of ProofBuilder (referenced in Section 4 below) for further details about Quantifier Removal.

The second deductive step illustrated in the sample proof above is Equivalence Substitution, which is applied to formulas (1) and (4) yielding formula (5), at which point formulas (1) and (4) are marked as used. When an equivalence formula of the form "$\varphi_1 \Leftrightarrow \varphi_2$" such as formula (1) above is available and one side of the equivalence can be safely unified with another formula φ_3 — e.g. here, equivalence formula (1)'s left-hand side, $\mathtt{S1} \subseteq \mathtt{S2}$, can be safely unified with formula (4), $\varnothing \subseteq \mathtt{A}$, via the substitutions $[\mathtt{S1} := \varnothing,\ \mathtt{S2} := \mathtt{A}]$ —

then the formula φ_3 may be replaced by the other side of the equivalence — e.g. here, the other side of the equivalence is $(\forall \mathtt{x})(\mathtt{x} \in \mathtt{S1} \Rightarrow \mathtt{x} \in \mathtt{S2})$ — subject to application of the safely unifying substitutions. The result here is formula (5), $(\forall \mathtt{x})(\mathtt{x} \in \varnothing \Rightarrow \mathtt{x} \in \mathtt{A})$. The use of "=" in the narrative here is derived from [18]. See the documentation of `ProofBuilder` (referenced in Section 4 below) for further details about using Equivalence Substitution (and safely unifying formulas).

Another deductive step illustrated in the sample proof above is nonclausal Resolution, which is applied to formulas (2) and (6), yielding formula (7). [19,20] describe nonclausal resolution. Whereas clausal resolution involves joining two clauses containing unifiable terms of the form "$\mathtt{P}(\ldots)$" for which one of the terms is negated and the other is not, nonclausal resolution here joins two formulas containing safely unifiable subformulas φ_1 and φ_2 for which their "polarities" are opposite. Polarity generalizes negation: goal formulas have positive polarity, supposition formulas have negative polarity, and negation that is explicit or implicit (as with "α" in an implication formula "$\alpha \Rightarrow \kappa$") yields reverse polarity. For example, in the sample proof above the supposition formula (2), $(\forall \mathtt{x})[\neg(\underline{\mathtt{x} \in \varnothing})]$, has negative polarity, and the underlined subformula $\underline{\mathtt{x} \in \varnothing}$ inside it has positive polarity; the goal formula (6), $\underline{\mathtt{a} \in \varnothing} \Rightarrow \mathtt{a} \in \mathtt{A}$, has positive polarity, and the underlined subformula $\underline{\mathtt{a} \in \varnothing}$ inside it has negative polarity; (See the documentation of `ProofBuilder` (referenced in Section 4 below) for further details about polarities). And these two underlined subformulas which have opposite polarities are safely unifiable via the substitution $[\mathtt{x} := \mathtt{a}]$. (Again, `ProofBuilder` implicitly/automatically removes supposition formula (2)'s universal quantifier, $(\forall \mathtt{x})$.) To do nonclausal resolution here, the user selects the desired formulas and subformulas. When a supposition formula ς contains a subformula φ_1 with positive polarity and a goal formula γ contains a subformula φ_2 with negative polarity and φ_1 and φ_2 are safely unifiable, then in this case nonclausally resolving these formulas yields a new goal formula $\neg\hat{\varsigma} \wedge \hat{\gamma}$, where $\hat{\varsigma}$ is derived from ς by replacing φ_1 with `true` and $\hat{\gamma}$ is derived from γ by replacing φ_2 with `false` — plus the safely unifying substitutions are applied, and `ProofBuilder` also simplifies the result. For example, in the sample proof above, the result of nonclausally resolving formula (2) and formula (6) — with the subformulas in them selected as indicated above — is the formula $\neg[\neg(\mathtt{true})] \wedge (\mathtt{false} \Rightarrow \mathtt{a} \in \mathtt{A})$, which `ProofBuilder` then simplifies as follows: the subformula $\neg[\neg(\mathtt{true})]$ simplifies to `true`, the subformula $(\mathtt{false} \Rightarrow \mathtt{a} \in \mathtt{A})$ simplifies to `true`, and then the intermediate result at this point, $\mathtt{true} \wedge \mathtt{true}$, simplifies to `true`, which `ProofBuilder` adds to the deductive tableau as formula (7) — and that completes this proof. See Section 4 below for further details about simplifying, and see the documentation of `ProofBuilder` (referenced in Section 4 below) for further details about using nonclausal Resolution.

Here is a second sample proof, of the theorem $\sum\limits_{1 \le \mathtt{i} \le \mathtt{n}} \mathtt{i} \;=\; \dfrac{\mathtt{n}(\mathtt{n}+1)}{2}$, i.e. $\sum\limits_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{i} \;=\; \dfrac{\mathtt{n}(\mathtt{n}+1)}{2}$:



Proof of the following formula:

```
(∀n)[(∑ i for 1 ≤ i ≤ n) = n*(n+1)/2]
```

(#)	Suppositions	Use	Theorem/Subgoals
	We presuppose the following:		

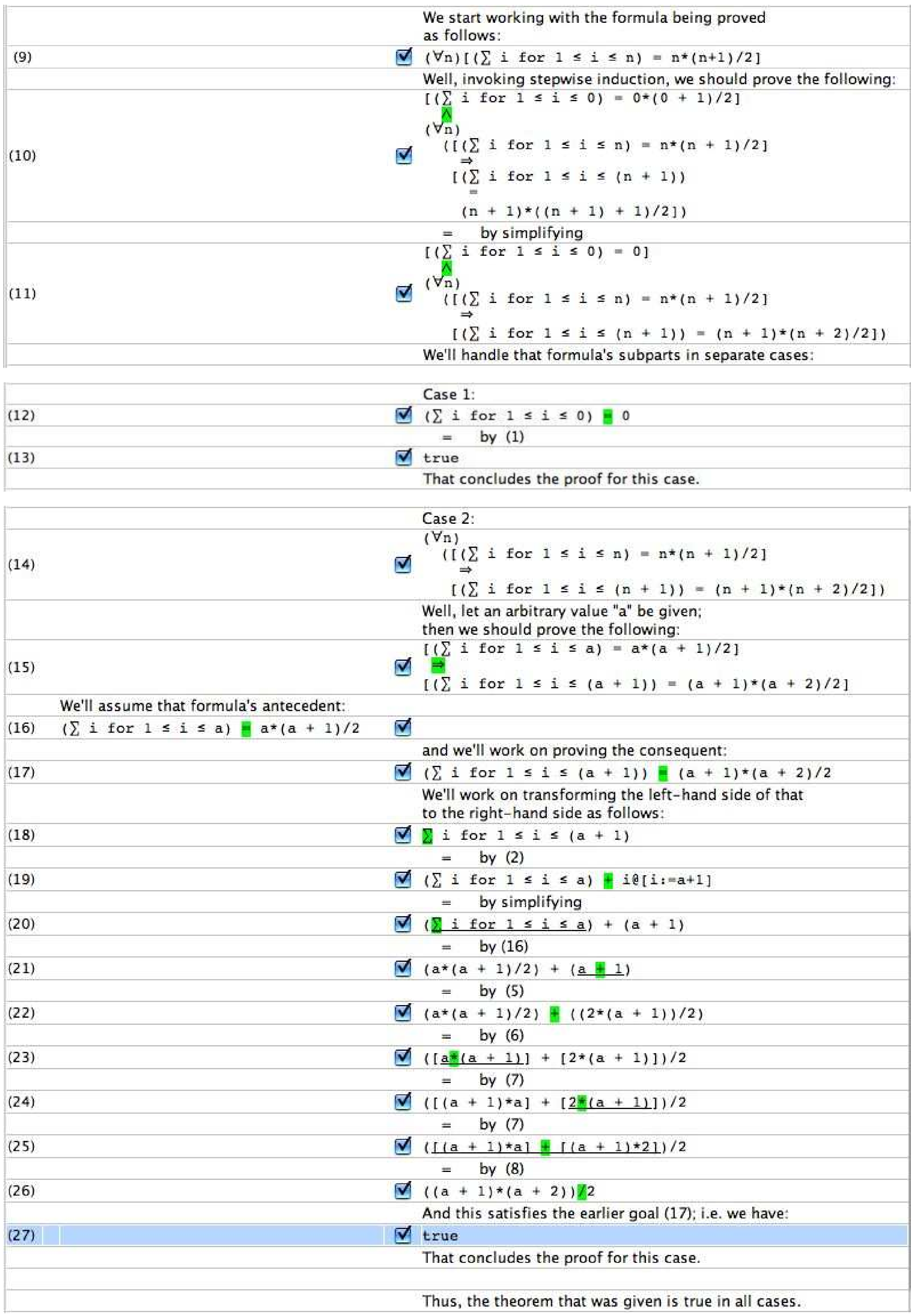
(1)	`(∀x,y,e,v)`
	`[x > y`
	`⇒`
	`(∑ e for x ≤ v ≤ y) = 0]`
		✓	

(2)	`(∀x,y,e,v)`
	`[(∑ e for x ≤ v ≤ y + 1)`
	`=`
	`(∑ e for x ≤ v ≤ y) + e@[v:`
		✓	

(3)	`(∀x,y)[y ≠ 0 ⇒ (y*x)/y = x]`	✓	
	Instantiating y with "2" in formula (3) yields:		
(4)	`(∀x)[2 ≠ 0 ⇒ (2*x)/2 = x]`	✓	
	= by simplifying		
(5)	`(∀x)[(2*x)/2 = x]`	✓	

(6)	`(∀x,y,z)`
	`[z ≠ 0 ⇒ x/z + y/z = (x + y)/z]`
		✓	

(7)	`(∀x,y)[x*y = y*x]`	✓	
(8)	`(∀x,y,z)[x*(y+z) = x*y + x*z]`	✓	



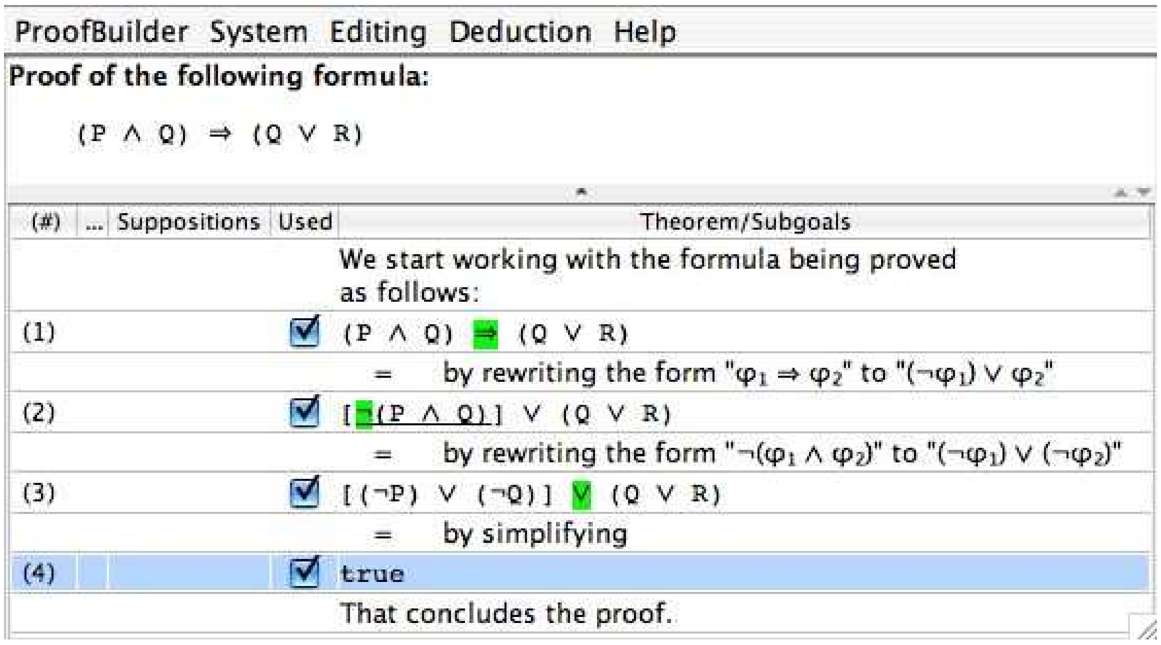
```
                                   We start working with the formula being proved
                                   as follows:
(9)                            ✓  (∀n)[(∑ i for 1 ≤ i ≤ n) = n*(n+1)/2]
                                   Well, invoking stepwise induction, we should prove the following:
                                   [(∑ i for 1 ≤ i ≤ 0) = 0*(0 + 1)/2]
                                    ∧
                                   (∀n)
(10)                           ✓    ([(∑ i for 1 ≤ i ≤ n) = n*(n + 1)/2]
                                      ⇒
                                     [(∑ i for 1 ≤ i ≤ (n + 1))
                                       =
                                      (n + 1)*((n + 1) + 1)/2])
                                     =    by simplifying
                                   [(∑ i for 1 ≤ i ≤ 0) = 0]
                                    ∧
(11)                           ✓   (∀n)
                                    ([(∑ i for 1 ≤ i ≤ n) = n*(n + 1)/2]
                                      ⇒
                                     [(∑ i for 1 ≤ i ≤ (n + 1)) = (n + 1)*(n + 2)/2])
                                   We'll handle that formula's subparts in separate cases:

                                   Case 1:
(12)                           ✓  (∑ i for 1 ≤ i ≤ 0) = 0
                                     =    by (1)
(13)                           ✓  true
                                   That concludes the proof for this case.

                                   Case 2:
                                   (∀n)
(14)                           ✓    ([(∑ i for 1 ≤ i ≤ n) = n*(n + 1)/2]
                                      ⇒
                                     [(∑ i for 1 ≤ i ≤ (n + 1)) = (n + 1)*(n + 2)/2])
                                   Well, let an arbitrary value "a" be given;
                                   then we should prove the following:
                                   [(∑ i for 1 ≤ i ≤ a) = a*(a + 1)/2]
(15)                           ✓     ⇒
                                   [(∑ i for 1 ≤ i ≤ (a + 1)) = (a + 1)*(a + 2)/2]
     We'll assume that formula's antecedent:
(16) (∑ i for 1 ≤ i ≤ a) = a*(a + 1)/2   ✓
                                   and we'll work on proving the consequent:
(17)                           ✓  (∑ i for 1 ≤ i ≤ (a + 1)) = (a + 1)*(a + 2)/2
                                   We'll work on transforming the left-hand side of that
                                   to the right-hand side as follows:
(18)                           ✓  ∑ i for 1 ≤ i ≤ (a + 1)
                                     =    by (2)
(19)                           ✓  (∑ i for 1 ≤ i ≤ a) + i@[i:=a+1]
                                     =    by simplifying
(20)                           ✓  (∑ i for 1 ≤ i ≤ a) + (a + 1)
                                     =    by (16)
(21)                           ✓  (a*(a + 1)/2) + (a + 1)
                                     =    by (5)
(22)                           ✓  (a*(a + 1)/2) + ((2*(a + 1))/2)
                                     =    by (6)
(23)                           ✓  ([a*(a + 1)] + [2*(a + 1)])/2
                                     =    by (7)
(24)                           ✓  ([(a + 1)*a] + [2*(a + 1)])/2
                                     =    by (7)
(25)                           ✓  ([(a + 1)*a] + [(a + 1)*2])/2
                                     =    by (8)
(26)                           ✓  ((a + 1)*(a + 2))/2
                                   And this satisfies the earlier goal (17); i.e. we have:
(27)                           ✓  true
                                   That concludes the proof for this case.

                                   Thus, the theorem that was given is true in all cases.
```

Features of ProofBuilder highlighted with this proof are as follows. For details beyond what's indicated here (e.g., for strong induction), see the documentation of ProofBuilder (referenced in Section 4 below).

– In addition to the notation shown above for summations, derived from [21] (and to a lesser extent from [18]), ProofBuilder also accepts notation like what is more common: "∑ i for i = 1 to n".

– ProofBuilder uses notation shown in formula (2), "e@[v := y + 1]", to represent substitution. This notation is derived from [11,18].
– ProofBuilder performs induction as shown with formulas (9)-(10).
– As shown above with formula (11), ProofBuilder enables the user to handle parts of a goal formula in separate cases. Parts of a supposition formula can be handled similarly.
– As shown with formulas (15)-(17), ProofBuilder enables the user to do direct proof.
– As shown with formulas (17)ff., ProofBuilder enables the user to prove an equation conveniently by transforming one side to the other.
– As shown with formulas (18)-(26), an additional deductive step is Equality Substitution. This is similar to Equivalence Substitution.
– As shown with formulas (3)-(5) (the latter of which is used with formula (21)), ProofBuilder enables the user to instantiate a universally quantified supposition with a value.

Finally, here is a third sample proof produced in ProofBuilder:



Proof of the following formula:

$(P \wedge Q) \Rightarrow (Q \vee R)$

(#)	...	Suppositions	Used	Theorem/Subgoals
				We start working with the formula being proved as follows:
(1)			✓	$(P \wedge Q) \Rightarrow (Q \vee R)$
				= by rewriting the form "$\varphi_1 \Rightarrow \varphi_2$" to "$(\neg\varphi_1) \vee \varphi_2$"
(2)			✓	$[\neg(P \wedge Q)] \vee (Q \vee R)$
				= by rewriting the form "$\neg(\varphi_1 \wedge \varphi_2)$" to "$(\neg\varphi_1) \vee (\neg\varphi_2)$"
(3)			✓	$[(\neg P) \vee (\neg Q)] \vee (Q \vee R)$
				= by simplifying
(4)			✓	true
				That concludes the proof.

The theorem here, $(P \wedge Q) \Rightarrow (Q \vee R)$, could be proved several other ways in ProofBuilder; but the point here is that this is one way that people like to do proofs, and ProofBuilder is capable of this method as well as others. Rewritings like the ones shown here work for predicate as well as propositional formulas. See Section 4 below for details about Rewriting.

4 Details of Operating ProofBuilder

Running ProofBuilder

ProofBuilder is written in Java (version 5.0), so it runs on any platform where that (or a later version of Java) is installed. The software is contained in the file ProofBuilder.jar. Anyone can obtain the software and documentation (etc.) at the following URL: http://www.cis.gvsu.edu/~mcguire/ProofBuilder/ . In a GUI operating system such as Microsoft Windows, one can start running ProofBuilder by double-clicking on ProofBuilder.jar (if Java ≥5.0 is installed properly). Or in a command-line environment such as LINUX, one can start running ProofBuilder by typing the following command:

```
java -ea -jar ProofBuilder.jar
```

ProofBuilder starts by presenting a window in which the user can enter the theorem being proved; as indicated above, ProofBuilder inserts the theorem in the first row in the Theorem/Subgoals column. An item in the Editing menu enables the user to enter premises, i.e. suppositions other than than ones arising inside the proof (an example of a supposition arising inside a proof is when a goal formula is an implication $\alpha \Rightarrow \kappa$ and we suppose that α is true, and we try to prove κ).

After entering the theorem and premises, the user uses the mouse to select expressions and uses the Deduction menu to choose deductions for `ProofBuilder` to apply to the selected expressions, and `ProofBuilder` adds the results of the deductions to the proof until it achieves a goal of `true` (or a supposition of `false`), which concludes the proof. The deductions that `ProofBuilder` provides are as follows:

Rewrite selection	Use additional row	Transform one side to other
Split formula into components	Invoke bivalance	Instantiate universal supposition
Suppose negation	Substitute using equivalence	Try Simplification
Separate into cases	Substitute using equation	Conjoin two suppositions
Remove quantifier	Invoke induction	

Notation

When typing a theorem or presupposition formula, the user can copy and paste formulas from other convenient sources such as Web pages — including symbols such as "$\leq$", "$\forall$", and "$\subseteq$". Alternatively, the user can type such symbols by pressing the `ALT` key together with specified keys; for example, pressing `ALT-<` yields "$\leq$, pressing `ALT-A` yields "$\forall$, and pressing `ALT-{` yields "$\subseteq$. (On an Apple Macintosh, it may be necessary to press the `CTRL` key also.) Here are such keymappings that `ProofBuilder` provides:

`A` $\longrightarrow \forall$	`E` $\longrightarrow \exists$	`s` $\longrightarrow \sum$	`p` $\longrightarrow \prod$	`=` $\longrightarrow \equiv$	`B` $\longrightarrow \Leftrightarrow$	`b` $\longrightarrow \leftrightarrow$	`I` $\longrightarrow \Rightarrow$	`i` $\longrightarrow \rightarrow$
`o` $\longrightarrow \vee$	`a` $\longrightarrow \wedge$	`:` $\longrightarrow \leftarrow$	`#` $\longrightarrow \neq$	$<\ \longrightarrow \leq$	$>\ \longrightarrow \geq$	`e` $\longrightarrow \in$	`[` $\longrightarrow \subset$	`{` $\longrightarrow \subseteq$
`u` $\longrightarrow \cup$	`t` $\longrightarrow \cap$	`+` $\longrightarrow \Theta$	`W` $\longrightarrow \Omega$	`w` $\longrightarrow \omega$	`n` $\longrightarrow \neg$	`y` $\longrightarrow \infty$	`R` $\longrightarrow \mathbb{R}$	`Q` $\longrightarrow \mathbb{Q}$
`Z` $\longrightarrow \mathbb{Z}$	`N` $\longrightarrow \mathbb{N}$	`/` $\longrightarrow \varnothing$						
superscript:	`0` $\longrightarrow {}^{0}$	`1` $\longrightarrow {}^{1}$	`2` $\longrightarrow {}^{2}$	`3` $\longrightarrow {}^{3}$	`-` $\longrightarrow {}^{-}$			

And actually, the user can use different symbols for things as desired; for example, the user can use any of the symbols "$\Rightarrow$", "$\rightarrow$", "`implies`", and "`IMPLIES`" as an implication symbol. Thus, `ProofBuilder` can be used by people who have different preferences for notation.

Incidentally, note the 'naïveté' of the logic, e.g. for set theory as demonstrated in the first illustration in Section 3 above. As is standard with textbooks for introductory courses on Discrete Mathematics, `ProofBuilder` makes no restrictions on types of variables or other terms (`Deductive Tableau` and `Deduct` provided such restrictions, but they were annoying), nor are axioms necessarily restricted to non-naïve ones for set theory such as Zermelo-Fraenkel. It might be considered bad that `ProofBuilder` thus allows one to enter a Russell's-paradox formula such as $(\exists \mathtt{r})(\forall \mathtt{x})[\mathtt{x} \in \mathtt{r} \Leftrightarrow \neg(\mathtt{x} \in \mathtt{x})]$ and then instantiate `x` with the the value obtained for `r`, obtaining a contradiction. But enabling students to play with this paradox may actually facilitate leading into discussion of non-naïve set theories.[4]

Simplifying

Here are simplifications that `ProofBuilder` performs, automatically. In this list, the abbreviations "`f`" and "`t`" are used for "`false`", and "`true`", respectively.

$$\begin{array}{llll}
\mathtt{f} \wedge \varphi \longrightarrow \mathtt{f} & \mathtt{t} \wedge \varphi \longrightarrow \varphi & \varphi \wedge \varphi \longrightarrow \varphi & \varphi \wedge \neg\varphi \longrightarrow \mathtt{f} \\
\mathtt{f} \vee \varphi \longrightarrow \varphi & \mathtt{t} \vee \varphi \longrightarrow \mathtt{t} & \varphi \vee \varphi \longrightarrow \varphi & \varphi \vee \neg\varphi \longrightarrow \mathtt{t} \\
\mathtt{f} \Rightarrow \varphi \longrightarrow \mathtt{t} & \varphi \Rightarrow \mathtt{f} \longrightarrow \neg\varphi & \mathtt{t} \Rightarrow \varphi \longrightarrow \varphi & \varphi \Rightarrow \mathtt{t} \longrightarrow \mathtt{t} \\
\varphi \Rightarrow \varphi \longrightarrow \mathtt{t} & \varphi \Rightarrow \neg\varphi \longrightarrow \varphi & \neg\varphi \Rightarrow \varphi \longrightarrow \neg\varphi & \\
\neg\varphi_1 \Rightarrow \neg\varphi_2 \longrightarrow \varphi_2 \Rightarrow \varphi_1 & & \neg(\varphi_1 \Rightarrow \neg\varphi_2) \longrightarrow \varphi_1 \wedge \varphi_2 & \neg\neg\varphi \longrightarrow \varphi \\
\xi = \xi \longrightarrow \mathtt{t} & & \xi \neq \xi \longrightarrow \mathtt{f} &
\end{array}$$

Additionally, `ProofBuilder` simplifies evaluable arithmetic expressions to their values. For example, $1+2 \longrightarrow 3$, $4<5 \longrightarrow$ `true`, and $48 \bmod 7 \longrightarrow 6$.

Rewriting

Here are Rewritings that `ProofBuilder` can perform:

$$\begin{array}{lcl}
\neg(\varphi_1 \wedge \varphi_2) & \longleftrightarrow & \neg\varphi_1 \vee \neg\varphi_2 \\
\neg(\varphi_1 \vee \varphi_2) & \longleftrightarrow & \neg\varphi_1 \wedge \neg\varphi_2 \\
\varphi_1 \Rightarrow \varphi_2 & \longleftrightarrow & \neg\varphi_1 \vee \varphi_2 \\
\varphi_1 \wedge (\varphi_2 \vee \varphi_3) & \longleftrightarrow & (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3) \\
\varphi_1 \vee (\varphi_2 \wedge \varphi_3) & \longleftrightarrow & (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)
\end{array}$$

$$\begin{array}{lcl}
\varphi_1 \Leftrightarrow \varphi_2 & \longleftrightarrow & (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1) \\
\varphi_1 \Leftrightarrow \varphi_2 & \longleftrightarrow & (\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \neg\varphi_2) \\
\neg(\forall \nu)\varphi & \longleftrightarrow & (\exists \nu)[\neg\varphi] \\
\neg(\exists \nu)\varphi & \longleftrightarrow & (\forall \nu)[\neg\varphi]
\end{array}$$

[4] In the future, `ProofBuilder` may provide types as in [18].

Further Rewritings are too trivial to list here, e.g. commutativity for $\wedge$, $\vee$, and $\Rightarrow$.

As shown in Section 3, `ProofBuilder` applies a Rewriting chosen by the user to an expression selected by the user (using the mouse).

Safely Unifying

Some of `ProofBuilder`'s deductive steps, notably nonclausal resolution and Equivalence Substitution, require safely unifying subformulas of two formulas. Details of safe unification are as follows:

1. The subformulas are not allowed to contain any quantified variables because it is unsound to allow arbitrary substitution for quantified variables.
2. Renaming is done as necessary to ensure that the two formulas have distinct variables.
3. Then, unifying substitutions of terms for free variables are accumulated, subject to the restriction that a variable must not occur in the term being substituted for it.

Further Details

For further details of operating `ProofBuilder`, see the full documentation of it at the following URL: `http://www.cis.gvsu.edu/~mcguire/ProofBuilder/` .

5 Conclusion

To summarize, `ProofBuilder` is an interactive system designed to help students learn to construct proofs, by allowing use of symbols and steps as in different textbooks, by structuring proofs and their construction clearly, and by performing menial or error-prone symbolic manipulations so students can concentrate on the higher-level process of deduction. It provides powerful capabilities such as nonclausal resolution, forward and backward reasoning, proving by contradiction, transforming one side of an equation or equivalence to the other, and proving by cases; but it does not automatically do such deductions: `ProofBuilder` leaves the activity of directing the deductive process to the student users, so they learn it.

Future Work

`ProofBuilder` is still a work in progress. Future work planned for it includes the following:

– Enhance the user interface. For example, enable the user to change formulas' formats — amounts of spaces, line breaks, indentation, and delimiters.
– Employing graphics, provide more canonical mathematical notations for things such as summations and sequences (e.g., "$\mathtt{a}_\mathtt{n}$").
– Cover more topics such as graphs, boolean algebra, and formal languages.
– Extend `ProofBuilder` to more advanced logics such as modal temporal logic.

References

1. Wiedijk, F.: The Seventeen Provers of the World. Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence. Springer-Verlag, New York (2006)
2. Benzmueller, C., Fiedler, A., Gabsdil, M., Horacek, H., Kruijff-Korbayova, I., Pinkal, M., Siekmann, J., Tsovaltzi, D., Vo, Q.B., Wolska, M.: Tutorial dialogs on mathematical proofs. Proceedings of IJCAI-03 Workshop on Knowledge Representation and Automated Reasoning for E-Learning Systems (2003) 12–22
3. Moszner, P.: Mizar as a tool of computer aided teaching. Computerised Logic Teaching Bulletin **2**(2) (1989) See also `http://mizar.org/project/bibliography.html` , not to mention the journal *Formalized Mathematics*.
4. Autexier, S., Fiedler, A.: Textbook proofs meet formal logic - the problem of underspecification and granularity. Proceedings of MKM'05 **3863** (2005)
5. Wenzel, M., Wiedijk, F.: A comparison of the mathematical proof languages mizar and isar. Journal of Automated Reasoning **29** (2002)
6. McMath, D., Rozenfeld, M., Sommer, R.: A computer environment for writing ordinary mathematical proofs. In: Procs. 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR). Volume 2250 of LNCS (LNAI)., London, UK, Springer-Verlag (2001) 507–516 Havana, Cuba.

7. Billingsley, W., Robinson, P.: Student proof exercises using MathsTiles and Isabelle/HOL in an intelligent book. Journal of Automated Reasoning **39** (2007)
8. McCune, W.: Otter 3.3 Reference Manual, Argonne, IL. (Aug 2003)
9. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
10. Burback, R., Fraser, S., McGuire, H., Smith, J., Winstandley, M.: Using the Deductive Tableau System. Chariot Software Group (1990)
11. Manna, Z., Waldinger, R.: The Deductive Foundations of Computer Programming. Addison-Wesley (1993)
12. Abell, M.L., Braselton, J.P.: Maple by Example. 3rd edn. Academic Press (2005)
13. Malik, D., Sen, M.: Discrete Mathematical Structures: Theory and Applications. Thomson (2004)
14. Rosen, K.: Discrete Mathematics and Its Applications. McGraw-Hill (2006)
15. Wolfram, S.: The Mathematica Book. 5th edn. Wolfram Media (2003)
16. Epp, S.S.: Discrete mathematics applets etc. (2006) Available [May 9, 2007] WWW: `http://condor.depaul.edu/~sepp/DMappletsEtc.htm` .
17. Ensley, D., Crawley, J.: Discrete Mathematics: Mathematical Reasoning and Proof with Puzzles, Patterns, and Games. Wiley (2006)
18. Gries, D., Schneider, F.: A Logical Approach to Discrete Math. Springer-Verlag (1994)
19. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Transactions on Programming Languages and Systems **2** (1980) 90–121
20. Murray, N.: Completely nonclausal theorem proving. Artificial Intelligence **18**(1) (1982) 67–85
21. Graham, R., Knuth, D., Patashnik, O.: Concrete Mathematics. 2nd edn. Addison-Wesley (1994)

Appendix: Soundness and Completeness of ProofBuilder

This explanation is derived from [11].

We associate with each deductive tableau a formula of the form "$\varsigma \Rightarrow \gamma$", where ς here is obtained as the conjunction of the universal closures of the deductive tableau's supposition formulas, and γ is obtained as the disjunction of the existential closures of the deductive tableau's goal formulas. I.e., if a deductive tableau's supposition formulas are $\varsigma_1, \varsigma_2, \varsigma_3, \ldots, \varsigma_{\mathtt{m}}$ and its goal formulas (i.e., the theorem and subgoals) are $\gamma_1, \gamma_2, \gamma_3, \ldots, \gamma_{\mathtt{n}}$, then the associated formula is $\bigwedge_{1\le \mathtt{i} \le \mathtt{m}} (\forall *)\varsigma_{\mathtt{i}} \;\Rightarrow\; \bigvee_{1\le \mathtt{j} \le \mathtt{n}} (\exists *)\gamma_{\mathtt{j}}$, where "$(\forall *)$" and "$(\exists *)$" stand for universal and existential quantifications of all appropriate free variables.

For example, consider the following 'artificial' deductive tableau:

Suppositions	**Theorem/Subgoals**
$\mathtt{IsZero(0)}$	
	$(\forall \mathtt{z})(\mathtt{y} < \mathtt{z}) \wedge \mathtt{IsZero(x)}$
$(\exists \mathtt{y})(\mathtt{x} < \mathtt{y})$	
	$\mathtt{z} < 0$

If the symbols "x", "y", and "z" are variables, then the associated formula for this deductive tableau is as follows:

$$[\mathtt{IsZero(0)}] \wedge [(\forall \mathtt{x})(\exists \mathtt{y})(\mathtt{x} < \mathtt{y})] \;\Rightarrow\; \Big[(\exists \mathtt{x},\mathtt{y})\Big((\forall \mathtt{z})(\mathtt{y} < \mathtt{z}) \wedge \mathtt{IsZero(x)}\Big)\Big] \vee [(\exists \mathtt{z})(\mathtt{z} < 0)]$$

Or when one starts constructing a proof of a theorem τ with presuppositions ς_1, ς_2, and ς_3, then the deductive tableau appears as follows:

Suppositions	**Theorem/Subgoals**
ς_1	
ς_2	
ς_3	
	τ

If ς_1, ς_2, ς_3, and τ are closed formulas (i.e., if all of the variables in these formulas appear within the scope of quantifiers in these formulas), then the associated formula for this deductive tableau is $(\varsigma_1 \wedge \varsigma_2 \wedge \varsigma_3) \Rightarrow \tau$. Clearly, at this starting point, the deductive tableau's associated formula may be valid (i.e., true under every model or interpretation) if and only if every model that satisfies the presuppositions ς_1, ς_2, and ς_3 also satisfies τ. That is to say, the validity of the associated formula of the initial deductive tableau corresponds to the validity of the theorem being proved (relative to presuppositions such as axioms or lemmas that are added to the proof).

The criterion for soundness here is that each deductive step is supposed to preserve the validity or nonvalidity of deductive tableaux' associated formulas. That is to say, the associated formula of a deductive tableau after a deductive step may be valid or nonvalid if and only if the associated formula of the deductive tableau before the deductive step was also equally valid or nonvalid. (In fact, almost all of the deductive steps here preserve equivalence of deductive tableaux' associated formulas, not just validity.) For example, suppose a supposition formula ς contains a subformula φ_1 with positive polarity and a goal formula γ contains a subformula φ_2 with negative polarity and φ_1 and φ_2 are safely unifiable, in which case nonclausally resolving these formulas yields a new goal formula $\gamma_{\mathtt{new}}$ which is $\neg\hat{\varsigma} \land \hat{\gamma}$, where $\hat{\varsigma}$ is derived from ς by replacing φ_1 with `true` and $\hat{\gamma}$ is derived from γ by replacing φ_2 with `false` (plus the safely unifying substitutions are applied). Recall that the associated formula α of the deductive tableau is $\bigwedge\limits_{\mathtt{i}}(\forall*)\varsigma_{\mathtt{i}} \Rightarrow \bigvee\limits_{\mathtt{j}}(\exists*)\gamma_{\mathtt{j}}$, with $\gamma_{\mathtt{new}}$ included among the $\gamma_{\mathtt{j}}$s after the deductive step but not before it. We show that nonvalidity (or validity) is preserved as follows:

- If the associated formula after the deductive step is not valid, i.e. if in some model the associated formula after the deductive step has the value false, then clearly in that model each formula $(\forall*)\varsigma_{\mathtt{i}}$ has the value true and each formula $(\exists*)\gamma_{\mathtt{j}}$ has the value false after the deductive step — which is when $\gamma_{\mathtt{new}}$ is included among the $\gamma_{\mathtt{j}}$s. Then clearly in that model each formula $(\forall*)\varsigma_{\mathtt{i}}$ had the value true and each formula $(\exists*)\gamma_{\mathtt{j}}$ had the value false also before the deductive step when $\gamma_{\mathtt{new}}$ was not included among the $\gamma_{\mathtt{j}}$s, hence in that model the associated formula before the deductive step had the value false, hence the associated formula before the deductive step was not valid,
- If the associated formula before the deductive step was not valid, i.e. if in some model the associated formula before the deductive step had the value false, then (like above) in that model each formula $(\forall*)\varsigma_{\mathtt{i}}$ had the value true and each formula $(\exists*)\gamma_{\mathtt{j}}$ had the value false before the deductive step when $\gamma_{\mathtt{new}}$ was not included among the $\gamma_{\mathtt{j}}$s. Recall that formula $\gamma_{\mathtt{new}}$ is $\neg\hat{\varsigma} \land \hat{\gamma}$, where $\hat{\varsigma}$ is derived from ς by replacing φ_1 with `true` and $\hat{\gamma}$ is derived from γ by replacing φ_2 with `false` (plus the safely unifying substitutions are applied). Also note that in the model here, the formula $(\forall*)\varsigma$ has the value true and the formula $(\exists*)\gamma$ has the value false. Also, $(\exists*)\gamma_{\mathtt{new}}$ i.e. $(\exists*)[\neg\hat{\varsigma} \land \hat{\gamma}]$ is equivalent to $\neg(\forall*)\hat{\varsigma} \land (\exists*)\hat{\gamma}$ because $\hat{\varsigma}$ and $\hat{\gamma}$ have undergone safe unification, which ensures that all their respective free variables are actually distinct. We will see that in the model here, the value of $(\exists*)\gamma_{\mathtt{new}}$ is false. Consider the value assigned by the model here to the formula φ obtained when φ_1 and φ_2 are unified; this value is either true or false.
  - First, suppose this value is true. Well, recall that the value of $(\forall*)\varsigma$ is true in this model. Safely unifying substitutions are applied to ς, changing some of its free variables to other terms; but since $(\forall*)\varsigma$ is true in this model, ς is true for all values at the places of its free variables, so $(\forall*)\varsigma$ with the safely unifying substitutions applied to ς is true in this model. Now, the unifying substitutions change φ_1 inside ς to φ, and we are assuming in this case that the value of φ is true in this model. Then replacing φ inside there with `true` should still yield the same value. Thus, $(\forall*)\hat{\varsigma}$ must be true in this model. Then, in this case, the value of $\neg(\forall*)\hat{\varsigma} \land (\exists*)\hat{\gamma}]$ is false, i.e. the value of $(\exists*)\gamma_{\mathtt{new}}$ is false.
  - Otherwise, suppose the value of φ is false in the model here. Well, recall that the value of $(\exists*)\gamma$ is false in this model. Safely unifying substitutions are applied to γ, changing some of its free variables to other terms; but since $(\exists*)\gamma$ is false in this model, γ is false for all values at the places of its free variables, so $(\exists*)\gamma$ with the safely unifying substitutions applied to γ is false in this model. Now, the unifying substitutions change φ_2 inside γ to φ, and we are assuming in this case that the value of φ is false in this model. Then replacing φ inside there with `false` should still yield the same value. Thus, $(\exists*)\hat{\gamma}$ must be false in this model. Then, in this case also, the value of $\neg(\forall*)\hat{\varsigma} \land (\exists*)\hat{\gamma}]$ is false, i.e. the value of $(\exists*)\gamma_{\mathtt{new}}$ is false.

  Thus, even when $\gamma_{\mathtt{new}}$ is included among the $\gamma_{\mathtt{j}}$s after the deductive step here, each formula $(\forall*)\varsigma_{\mathtt{i}}$ has the value true and each formula $(\exists*)\gamma_{\mathtt{j}}$ has the value false, so the associated formula after the deductive step has the value false in the model here. Consequently, the associated formula after the deductive step is not valid.

Thus, the associated formula of a deductive tableau after this deductive step may be nonvalid if and only if the associated formula of the deductive tableau before the deductive step was also equally nonvalid. This implies that the associated formula of a deductive tableau after this deductive step may be valid if and only if the associated formula of the deductive tableau before the deductive step was also equally valid.

Now, consider a final deductive tableau containing either a proof-terminating 'supposition' $\varsigma_{\mathtt{m}}$ which is `false` or a 'subgoal' $\gamma_{\mathtt{n}}$ which is `true`. If either $\varsigma_{\mathtt{m}}$ is `false` or $\gamma_{\mathtt{n}}$ is `true`, then clearly the associated formula at this point, $\bigwedge\limits_{1\leq\mathtt{i}\leq\mathtt{m}}(\forall*)\varsigma_{\mathtt{i}} \Rightarrow \bigvee\limits_{1\leq\mathtt{j}\leq\mathtt{n}}(\exists*)\gamma_{\mathtt{j}}$, is valid. But every deductive step here preserves validity of deductive tableaux' associated formulas, i.e. the associated formula of a deductive tableau after any deductive step is valid if and only if the associated formula of the deductive tableau before the deductive step was also valid. Well then, with the associated formula of the final deductive tableau being valid, then the associated formula of the initial deductive tableau must also be valid. But also, the validity of the associated formula of the initial deductive tableau corresponds to the validity of the theorem being proved (relative to presuppositions such as axioms or lemmas that are used in the proof). Therefore, achieving a 'subgoal' of `true` (or a 'supposition' of `false`) establishes the validity of the theorem being proved (relative to presuppositions such as axioms or lemmas that are used in the proof).

Regarding completeness, `ProofBuilder` can do all the steps of classical clausal resolution — including the process of reducing a given theorem to clausal form; thus, as is known for classical clausal resolution, `ProofBuilder` is refutation complete.

Labeled tableaux for distributed temporal logic*

David Basin[1], Carlos Caleiro[2] and Jaime Ramos[2], and Luca Viganò[3]

[1] Department of Computer Science, ETH Zurich, Switzerland
[2] SQIG-IT and CLC, Department of Mathematics, IST, TU Lisbon, Portugal
[3] Department of Computer Science, University of Verona, Italy

Abstract

The distributed temporal logic DTL [3] is a logic for reasoning about temporal properties of distributed systems from the local point of view of the system's agents, which are assumed to execute sequentially and to interact by means of synchronous event sharing.

DTL was first proposed in [3] as a logic for specifying and reasoning about distributed information systems. The logic has also been used in the context of security protocol analysis for reasoning about the interplay between protocol models and security properties [1,2]. However, all of the previous results have been obtained directly by semantic arguments. It would be reassuring, and useful in general, to have a usable deductive system for DTL. An attractive possibility in this regard is a labeled tableaux system as deductions will then closely follow semantic arguments.

We present a sound and complete labeled tableaux system for DTL. To achieve this, we formalize a labeled tableaux system for reasoning locally at each agent and afterwards we combine the local systems into a global one by adding rules that capture the distributed nature of DTL.

References

1. C. Caleiro, L. Viganò, and D. Basin. Metareasoning about Security Protocols using Distributed Temporal Logic. In *Proc. ARSPA04*, pp. 6789. ENTCS 125(1), 2005.
2. C. Caleiro, L. Viganò, and D. Basin. Relating strand spaces and distributed temporal logic for security protocol analysis. *Logic Journal of the IGPL*, 13(6):637664, 2005.
3. H.-D. Ehrich and C. Caleiro. Specifying communication in distributed information systems. *Acta Informatica*, 36:591–616, 2000.

* The work of the second and third authors was partially supported by FCT and EU FEDER, namely via the KLog project PTDC/MAT/68723/2006 of SQIG-IT, and the QuantLog project POCI/MAT/55796/2004 of CLC.

What First Order Theorem Provers Do For Monodic Temporal Reasoning

Ullrich Hustadt
Based on joint work with Clare Dixon, Michael Fisher, Boris Konev, and Alexei Lisitsa

Department of Computer Science, University of Liverpool, UK
U.Hustadt@csc.liv.ac.uk

Abstract

Temporal logics have long been recognised as introducing appropriate languages for specifying a wide range of important computational properties in computer science and artificial intelligence. However, the use of first-order temporal logic has been hampered by its lack of a complete proof system. Hodkinson, Wolter, and Zakharyaschev [1] were the first to show that a non-trivial fragment of first-order temporal logic, called the *monodic* fragment, or *monodic* first-order temporal logic, has the completeness property. This initial result was followed by an examination of the monodic fragment in terms of decidable subclasses, automated deduction, and applications.

In particular, Konev, Degtyarev, Dixon, Fisher and Hustadt investigated monodic first-order temporal logic in the context of resolution. In [4,5] they devise the *fine-grained resolution calculus* for monodic first-order temporal logic. This calculus forms the basis of the prover **TeMP** [2]. A refinement of this calculus, ordered fine-grained resolution with selection, is presented by Hustadt, Konev, and Schmidt [3], and shown to decide the guarded fragment and the dual Maslov class $\overline{\mathrm{K}}$ fragment of monodic first-order temporal logic.

In this paper we first recall the definition of the ordered fine-grained resolution with selection calculus. We then discuss the contribution that classical first-order resolution can make to the implementation of that calculus, using the architecture of **TeMP** and its connection with Vampire to illustrate this particular approach. Finally, we discuss a problem with this particular architecture, namely that derivations in general cannot be guaranteed to be fair and present a revised architecture which solves this problem.

References

1. I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable fragments of first-order temporal logics. *Annals of Pure and Applied Logic*, 106:85–134, 2000.
2. U. Hustadt, B. Konev, A. Riazanov, and A. Voronkov. **TeMP**: A temporal monodic prover. In *Proc. IJCAR 2004*, vol. 3097 of *LNAI*, pp. 326–330. Springer, 2004.
3. U. Hustadt, B. Konev, and R. A. Schmidt. Deciding monodic fragments by temporal resolution. In *Proc. CADE-20*, vol. 3632 of *LNAI*, pp. 204–218. Springer, 2005.
4. B. Konev, A. Degtyarev, C. Dixon, M. Fisher, and U. Hustadt. Towards the implementation of first-order temporal resolution: the expanding domain case. In *Proc. TIME-ICTL 2003*, pp. 72–82. IEEE, 2003.
5. B. Konev, A. Degtyarev, C. Dixon, M. Fisher, and U. Hustadt. Mechanising first-order temporal resolution. *Information and Computation*, 199(1–2):55–86, 2005.

A Graph-based Strategy for the Selection of Hypotheses [⋆]

Jean-François Couchot[1,2], Thierry Hubert[1,2,3]

[1] INRIA Futurs, ProVal, Parc Orsay Université, F-91893
[2] LRI, Univ Paris-Sud, CNRS, Orsay, F-91405
[3] Dassault Aviation, Saint-Cloud, F-92214
{couchot,hubert}@lri.fr

Abstract. In previous works on verifying C programs by deductive approaches based on SMT provers, we proposed the heuristic of separation analysis to handle the most difficult problems. Nevertheless, this heuristic is not sufficient when applied on industrial C programs: it remains some Verification Conditions (VCs) that cannot be decided by any SMT prover, mainly due to their size.
This work presents a strategy to select relevant hypotheses in each VC. The relevance of an hypothesis is the combination of two separated dependency analysis obtained by some graph traversals. The approach is applied on a benchmark issued from an industrial program verification.

1 Introduction

Using formal methods for verifying properties of programs at their source code level has gained more interest with the increased use of embedded programs, as for instance, plane command control, cars, smart cards... Such embedded programs, which require a high-level of confidence, are often written in C. In such a case, it is widely known that verifying safe pointers manipulation is one of the most critical tasks: aliasing, that is referencing a memory location by several pointers and out-of-bounds array access must be taken into account for instance.

Among the verification methods, the deductive one consists in transforming logical annotations of the program (pre- and post-conditions of a function, invariants) into formulae whose validity implies the correctness of these annotations. In practice, the technique that has shown itself the most effective is the Weakest Precondition (wp) calculus of Dijkstra [12]. It is the basis of effective tools such as ESC/Java [11], several tools for Java programs annotated using the Java Modeling Language [4], Boogie [1] for the C# programming language, and a tool of our own for C programs called Caduceus/Why [14].

However all these methods suffer from generating large Verification Conditions (VCs) when applied on industrial programs where scalability and efficiency

[⋆] This work is partially funded by the French Ministry of Research, thanks to the CAT (C Analysis Toolbox) RNTL (Reseau National des Technologies Logicielles).

are of paramount importance. Possible solutions are to optimize the memory model (e.g. by introducing separations of zones of pointers [16]), to improve the calculus of weakest precondition [17] and to apply strategies for simplifying VCs [15,9,18].

This work focuses on the latter. A VC is expressed as a goal and a context. The goal encodes the execution of the program, which can be seen as hypotheses, and the property that the program should satisfy, namely the conclusion. The context is an extension of a base theory (usually a combination of equality with uninterpreted function symbols and linear arithmetic) with a large set of axioms. context describes many features of the program such as the memory model.

Verification Conditions may contain useless axioms (e.g., when the program does not manipulate pointers, all the axioms about pointers could be dropped) and a huge number of useless hypotheses (e.g., when the property is initially established and does not concern subsequent instructions) introduced by wp. Moreover, a large number of useless axioms/hypotheses unnecessarily enlarge the search space of SMT solvers (i.e., Satisfiability Modulo Theories), and degrade unacceptably their performances.

Instead of invoking the SMT solvers blindly on the whole set of hypotheses in the whole context, we present a method to remove as many hypotheses as possible by a suitable selection strategy, which allows us to significantly prune the search space of SMT solvers.

The idea of the strategy developed here is quite natural: an hypothesis is relevant if it contains both the predicates and the variables needed to establish the conclusion. To compute this relevance result, we analyse dependencies between variables of the conclusions and variables of each hypothesis on one hand and predicates of the conclusion, predicates of the hypotheses and predicates used in the theory on the other hand.

Section 2 presents how the goal is preprocessed as the first step of the method. Section 3 presents a running example. Section 4 shows how we store dependencies in graphs. The selection of hypotheses is then presented in Sec. 5. These last two sections are the first contribution. The second contribution is the implementation of this strategy as a module of Caduceus/Why [14] and its application on an industrial C example that is about 4000 lines of annotated C code (Sec. 6). Section 7 discusses related work, concludes and presents future work.

2 Verification Conditions Normal Form

This section presents the normal form for VCs that we consider in the rest of this paper. A first point to be noted is that due to the application of a weakest precondition calculus on imperative programs, the usual form of Verification Condition is

$$\forall X(H_1 \Rightarrow (H_2 \Rightarrow \ldots (H_n \Rightarrow C)))$$

where X is the set of variables in the formula collected during a prenexing step, H_i, $1 \leqslant i \leqslant n$ and C are first-order logic formulas.

In what follows we consider C and H_i, $1 \leqslant i \leqslant n$ to be quantifier free. Such a restriction is not a problem since each remaining quantified subformula φ may be replaced by a predicate $p(Y)$, where p is a fresh symbol and Y is the set of free variables in φ with $Y \subseteq X$. To be correct and complete, we add the axiom $\forall Y \,.\, p(Y) \Leftrightarrow \varphi$ into the theory (see [7] for more details and proof of equisatisfiability).

$$\begin{array}{ll}
\mathsf{split}(H_1 \vee H_2 \Rightarrow C) & = \mathsf{split}(H_1 \Rightarrow C) \cup \mathsf{split}(H_2 \Rightarrow C) \\
\mathsf{split}(H \Rightarrow C) & = \bigcup_{c \in \mathsf{split}(C)} \{H \Rightarrow c\} \\
\mathsf{split}(C_1 \wedge C_2) & = \mathsf{split}(C_1) \cup \mathsf{split}(C_2) \\
\mathsf{split}(\varphi) & = \{\varphi\}
\end{array}$$

Fig. 1. Generating small VCs with split function

For simplification purpose, we consider that each hypothesis is written in DNF and the conclusion is written in CNF. We use then a function that reduces the size but raise the number of the VCs by splitting conjunctions in positive occurrences and equivalently disjunction in negative occurrence. It is formalized with the function split given in Fig. 1, where rules are applied from the left to the right and where the last rule is considered if upper ones can not be applied. It results in a set of valid VCs if and only if the given larger VC is valid. Notice that resulting VCs are free of negative occurrences of disjunction and positive occurrence of conjunction.

We are left with VCs of the form

$$l_1^1 \wedge \ldots \wedge l_1^{m_1} \Rightarrow (l_2^1 \wedge \ldots \wedge l_2^{m_2} \Rightarrow \ldots \Rightarrow (l_n^1 \wedge \ldots \wedge l_n^{m_n} \Rightarrow (l_{n+1}^1 \vee \ldots \vee l_{n+1}^{m_{n+1}})))$$

where each l_i,$1 \leqslant i \leqslant n+1$ is a literal. Obviously, each VC of this form is valid if and only if one of the m_{n+1} VCs given by

$$\big(l_1^1 \wedge \ldots \wedge l_1^{m_1} \bigwedge \ldots \bigwedge l_n^1 \wedge \ldots \wedge l_n^{m_n}\big) \Rightarrow l_{n+1}^i,$$

$1 \leqslant i \leqslant m_{n+1}$, is valid. The *normal form* of our VC is

$$\big(l_1 \wedge \ldots \wedge l_n\big) \Rightarrow l_{n+1}, \tag{1}$$

where each l_i is a literal which is a Horn Clause.

3 Running Example

Figure 2 is the starting point of the running example that illustrates the approach throughout the following sections. The left side column of this C program introduces matryoshka structures, whereas the right side column presents

```
struct p{
int x;
} p;

struct s{
struct p v[2];
} s;

struct t{
 struct s *y;
}t;
```

```
/*@ requires \valid(c)
    assigns c->v[0].x*/
void g(struct s *c);

/*@ requires \valid(a) && \valid(b)
           && \valid(a->y)
    assigns a->y->v[0..1].x*/
void f(struct t *a, struct p *b){
int i = b->x;
g(a->y);
a->y->v[1].x=i;
}
```

Fig. 2. C running example

the interface `g` and the function `f`. This later function calls `g` and explicitly modifies the value stored in one of the innermost fields of the structure pointed by `a` (namely `a->y->v[1].x`).

Functions are annotated in the language of the Caduceus/Why [14], which is composed of

- pre-conditions (defined by `requires` keyword); such pre-conditions ensure that each pointer given in parameter is `\valid` (i.e. is correctly allocated) when entering the function `f` or the interface `g`;
- a list of data modified by side effects (defined by `assigns` keyword); for instance `assigns a->y->v[0..1].x` means that the function `f` does not modify locations outside the set {`a->y->v[0].x`, `a->y->v[1].x`}; such property can be established by considering the side effects of `g` and the body of `f`.

Caduceus VCs generator yields predicates valid and diff which have the semantic of `\valid` and `assigns` respectively.

This example is quite representative of the class for which it is hard to show the absence of threats (null pointer dereferencing, out-of-bounds array access, or more generally dereferencing a pointer that does not point to a regularly allocated memory block).

For such a program, Caduceus/Why yields two kinds of VCs: some that establish validity of pointers for each memory access, and some that establish which the list of side effects given in annotations is a superset of the function's side effects. For instance, the instruction `g(a->y)` constraints `a` to be valid and `a->y` to be also valid due to the pre-condition of `g`.

We apply a Burstall-Bornat method [5,3] which consists in having one 'array' variable (later called a memory) for each structure field. This modeling syntactically encodes the fact that two structure fields cannot be aliased. The important consequence is that whenever one field is updated, the corresponding array is the only one which is modified. Hence, we have for free that any other field is left unchanged. In practice the fields `x`, `v` and `y` yield respectively the memories m_x, m_v and m_y.

Memories can be accessed only by function acc; $\mathsf{acc}(m,p)$ returns the value stored in the memory m at index p. A fresh memory can be generated by function upd; $\mathsf{upd}(m,p,v)$ duplicates m except at pointer p where it sets the value v.

Let us define the predicate $\mathsf{diff}(m_1,m_2,l)$ where m_1 and m_2 are two memories, and let l be a set of pointers. Intuitively, $\mathsf{diff}(m_1,m_2,l)$ means that differences between m_1 and m_2 only concern the set l. It is formalized with

$$\mathsf{diff}(m_1,m_2,l) \Leftrightarrow \big(\forall p\,.\,\mathsf{valid}(p) \wedge \neg\mathsf{mem}(p,l) \Rightarrow \mathsf{acc}(m_1,p) = \mathsf{acc}(m_2,p)\big) \quad (2)$$

where p is a pointer and $\mathsf{mem}(p,l)$ means that p is a member of l. Note that the formula (2) is one of the 80 axioms that compose the memory model of Caduceus/Why.

The VC generated according to the `assigns` annotation of function f is

$$\begin{array}{ll}
H_1 \left\{ \begin{array}{l} \left(\begin{array}{l}\mathsf{valid}(a) \wedge \mathsf{valid}(b) \wedge \mathsf{valid}(\mathsf{acc}(m_y,a)) \wedge \\ \quad \mathsf{valid\_acc\_range}(m_v,2) \wedge \mathsf{separation1\_range}(m_v,2)\end{array}\right) \Rightarrow \end{array}\right. & \\
H_2 \left\{ \Big(\mathsf{diff}(m_x, m_x\_0, \mathsf{singleton}(\mathsf{shift}(\mathsf{acc}(m_v,\mathsf{acc}(m_y,a)),0))) \Rightarrow \right. & \quad (3) \\
C \left\{ \begin{array}{l} \quad \mathsf{diff}(m_x, \mathsf{upd}(m_x\_0, \mathsf{shift}(\mathsf{acc}(m_v,\mathsf{acc}(m_y,a),1)), \mathsf{acc}(m_x,b)), \\ \qquad \mathsf{range}(\mathsf{singleton}(\mathsf{acc}(m_v,\mathsf{acc}(m_y,a))),0,1)) \Big) \end{array}\right. &
\end{array}$$

where all variables are universally quantified. We have two hypotheses:

H_1. The first line corresponds to the pre-condition of function `f`. The second line is issued from the definition of structure `s`: predicate $\mathsf{valid\_acc\_range}(m_v,2)$ means that accessing to the memory m_v, for each index p that is a valid pointer, returns an array t s.t. pointers $t[0]$ and $t[1]$ are valid; with the same notations $\mathsf{separation1\_range}(m_v,2)$ means that $t[0] \neq t[1]$.

H_2. It is issued from the instruction `g(a−>y)`: function $\mathsf{singleton}$ has the usual meaning and $\mathsf{shift}(t,i)$ allows to access in the array t to the index i. This hypothesis defines the access of variable m_x_0 which is equal to the access in m_x except for the index $\mathsf{shift}(\mathsf{acc}(m_v,\mathsf{acc}(m_y,a)),0)$ corresponding to `assigns c−>v[0].x` where `c` is replaced by `a−>y`.

The conclusion C is a diff predicate applied to two memories. The first memory is the memory before execution of `f` and the second memory is the memory after execution of `f`. The third parameter $\mathsf{range}(\mathsf{singleton}(\mathsf{acc}(m_v,\mathsf{acc}(m_y,a))),0,1)$ defines the set of pointers located at the indices 0 and 1 in the array $\mathsf{acc}(m_v,\mathsf{acc}(m_y,a))$ in fact this is the representation of `a->y->v[0..1]`

Normal form of the VC (3) is

$$\begin{array}{l}\left(\begin{array}{l}\mathsf{valid}(a)\wedge\\ \mathsf{valid}(b)\wedge\\ \mathsf{valid}(\mathsf{acc}(m_y,a))\wedge\\ \mathsf{valid\_acc\_range}(m_v,2)\wedge\\ \mathsf{separation1\_range}(m_v,2)\wedge\\ \mathsf{diff}(m_x,m_x\_0,\mathsf{singleton}(\mathsf{shift}(\mathsf{acc}(m_v,\mathsf{acc}(m_y,a)),0)))\end{array}\right)\Rightarrow\\ \mathsf{diff}\left(\begin{array}{c}m_x,\mathsf{upd}(m_x\_0,\mathsf{shift}(\mathsf{acc}(m_v,\mathsf{acc}(m_y,a),1)),\mathsf{acc}(m_x,b)),\\ \mathsf{range}(\mathsf{singleton}(\mathsf{acc}(m_v,\mathsf{acc}(m_y,a))),0,1))\end{array}\right)\end{array} \tag{4}$$

Even if this example is quite short, among the SMT-provers Simplify [10], Yices [13], Ergo [6], haRVey [21] and CVC-lite [2], Simplify and haRVey are the only ones which succeed in establishing the validity of this VC in a few seconds.

However, an engineer would have deduced from the background theory that diff and valid_acc are not directly linked in the present case. Removing the hypothesis concerning $\mathsf{valid\_acc\_range}$ permits him to obtain the desired result.

The next section shows how dependencies are memorized in the problem of proving a goal in a SMT solver. This is the starting point of the approach of removing useless hypotheses.

4 Memorizing Dependency

Dependencies between hypotheses or between an hypothesis and the conclusion of the goal can be of two levels: predicative level and variable level. In the former case, two predicates are dependent if there exists a (deductive) path leading from the first one to the second one. Such a path may be deduced from axioms of the theory and does not depend on the program. Such a dependency, later denoted as *static*, is presented in Sec. 4.1. In the later case, it is obvious that two formulae are dependent if they have a common set of variables. These variables may either be program variables or may result from a weakest precondition calculus applied on the program and its assertions. For that reason such dependency is later denoted *dynamic* and is presented in Sec. 4.2.

4.1 Static Dependency

Our goal is to compute an directed graph with weights representing the implicative relation between predicates. Intuitively, in this graph, each vertex represents a predicate name and an arc from the vertex p to the vertex q means that p may imply q. What follows details how to compute such a graph of predicates named G_P.

Practically, each vertex of the graph is labelled with a predicate symbol that appears in one literal except equality or inequality predicates of the theory. Notice that if a predicate r appears in a negative occurrence (e.g. in a axiom of the form $\neg r$), it is nevertheless represented as an vertex labeled with r. Let us show how the dependency between these predicates are represented as edges.

First, each axiom is written as a CNF, but in a straightforward way (by opposition to elaborate CNF [19]): axioms are of short size and their transformation into CNF do not yield combinatorial explosion. Then, each resulting clause C (viewed as a set of literals) can be seen as the union of C_- and C_+ containing negative literals of C and positive literals of C respectively. Intuitively each predicate symbol of C_- is a premise and each predicate symbol of C_+ is a consequence. An edge is then added for each pair in $C_- \times C_+$ that does not contain an equality or an inequality.

Let p be a predicate symbol in C_- and q be a predicate symbol in C_+. We propose to label each edge from p to q with a weight k such that the lowest the edge weight k is, the highest is the probability of p to establish q. Such an edge is labeled with the number $card(C)-1$. A large clause with many premises, among of them p, and with many consequents, among of them q, has less chance to be used in a deduction step leading to q than the clause $\{\neg p, q\}$. If there exist $p \xrightarrow{w_1} q$ and $p \xrightarrow{w_2} q$, we leave only the edge $p \overset{min(w_1,w_2)}{\longrightarrow} q$. Notice that even if equality and inequality predicates are not represented as edges they are however considered in the graph of dependency since they are involved in the calculus of weights.

Running example. As a short example, we apply the method on axiom (2). Its CNF is given by

$$\begin{aligned}
&\{\{\neg\mathsf{diff}(M_1, M_2, L), \neg\mathsf{valid}(P), \mathsf{mem}(P, L), \mathsf{acc}(M_1, P) = \mathsf{acc}(M_2, P)\},\\
&\quad\{\mathsf{valid}(p_0), \mathsf{diff}(M_1, M_2, L)\},\\
&\quad\{\neg\mathsf{mem}(p_0, L), \mathsf{diff}(M_1, M_2, L)\},\\
&\quad\{\mathsf{acc}(M_1, p_0) \neq \mathsf{acc}(M_2, p_0), \mathsf{diff}(M_1, M_2, L)\}\}
\end{aligned} \tag{5}$$

where capitalized variables are universally quantified and p_0 is a fresh constant resulting from the skolemization of p. Figure 3 represents the dependency graph corresponding to this axiom. It is then an excerpt of the graph representing the memory model of Caduceus/Why.

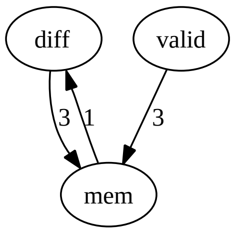


Fig. 3. Dependency graph of axiom (2)

4.2 Dynamic Dependency

Our aim is to compute an undirected graph which represents the relation between hypotheses as relations between the variables they contain. Let us detail how to build such a graph of variables named G_V.

Vertices are labeled with the variables of the goal and variables resulting from a flattening process on hypotheses: in some hypothesis H, a functional term $f(t_1, \ldots, t_n)$ that is a parameter of a predicate, or a function, should be replaced by a fresh variable x. The hypothesis $x = f(t_1, \ldots, t_n)$ should be added just before H. Obviously, the flattening task is not applied when the functional symbol is the parameter of the equality predicate.

Each predicate is then represented by the complete graph composed by all the vertices corresponding to the predicate variables.

Notice that both flattening and edge computing do not concern the conclusion of the goal: adding intermediate variables and allowing to incrementally select deeper and deeper variables is only meaningful for hypotheses. All the variables of the conclusion have to be selected without any graph traversal.

Running example. The graph representing verification condition (4) is given in Fig. 4. In such a graph, $singleton\_2$,$shift\_3$, acc_4 and acc_5 are fresh variables introduced by the flattening-like step applied on the hypothesis

$$\mathsf{diff}(m_x, m_x\_0, \mathsf{singleton}(\mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a)), 0))).$$

It leads to five complete graphs corresponding to sets $\{m_x, m_x\_0, singleton\_2\}$, $\{singleton\_2, shift\_3\}$, $\{shift\_3, acc\_4\}$, $\{acc\_4, acc\_5, m_v\}$ and $\{acc_5, m_y, a\}$.

The approach is similar for acc_1 which is issued from $\mathsf{valid}(\mathsf{acc}(m_y, a))$.

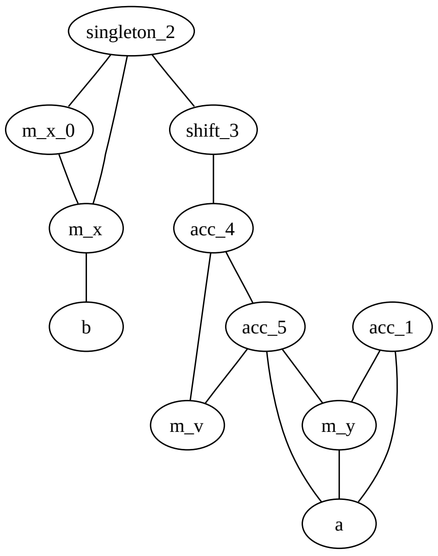


Fig. 4. Dependency graph of verification condition (4)

5 Selection of Relevant Hypotheses

We are left to select *relevant* hypotheses. Intuitively, a sub-formula is relevant with respect to the conclusion of the formula if this formula cannot be established without the former. More formally, let $\varphi =_{def} h_1 \wedge \ldots \wedge h_n \Rightarrow C$ be a formula. The hypothesis h_1 is relevant w.r.t. C if $h_1 \wedge \ldots \wedge h_n \Rightarrow C$ is valid whereas $h_2 \wedge \ldots \wedge h_n \Rightarrow C$ is not, and similarly for any h_i, $1 \leqslant i \leqslant n$. This section shows how to select relevant predicates (Sec. 5.1), relevant variables (Sec. 5.2) and explains how to combine these results to select relevant hypotheses (Sec. 5.3).

5.1 Relevant Predicates

As shown in Sec. 2, the conclusion can be reduced to one literal c without loss of generality. In what follows, we do not distinguish a predicate symbol from its corresponding node in the graph of predicates G_P.

A predicate symbol p is relevant w.r.t. a predicate symbol q if there is a path from p to q. Intuitively, the weakest the weight of the path is, the highest the probability of p to establish q is.

A search computes the set of predecessors of positive occurrence of any predicate appearing in the conclusion. Similarly, successors of any predicate wich appears in a negative occurrence in the conclusion are computed (e.g. if the conclusion is $\neg q$). All these predecessors and successors are stored into the list $\mathcal{L}$ ordered by the path weight. Finally, completeness of the selection is obtained by adding unreachable predicates into the list tail.

In the following, if i is an index that is positive or null, $\mathcal{L}[0 \ldots i]$ denotes the set of predicates symbols of $\mathcal{L}$ stored in the $i+1$ first places of the list. Particularly, $\mathcal{L}[0 \ldots 0]$ is the set of conclusion predicates.

Running example. According to the graph given in Fig. 3, we have :

$$\begin{array}{l}\mathcal{L}[0] = \{\mathsf{diff}\}\\ \mathcal{L}[1] = \mathcal{L}[0] \cup \{\mathsf{mem}\}\\ \ldots\end{array}$$

Notice that $\mathsf{valid\_acc\_range}$ appears neither in $\mathcal{L}[0]$ nor in $\mathcal{L}[1]$.

5.2 Relevant Variables

Starting with the variables of the conclusion $\mathcal{V}^0$, a breadth-first search algorithm computes the fix-point $\mathcal{V}^*$ variables that are reachable from $\mathcal{V}^0$ in the variable graph G_V.

The sequence $(\mathcal{V}^n)_{n\in\mathbb{N}}$ of reachable variables sets is defined for $n \in \mathbb{N}$ with

$$\left\{\begin{array}{l}\mathcal{V}^{2n+1} = \mathcal{V}^{2n} \cup \{v \mid \exists v_1, v_2 \,.\, v_1 \neq v_2 \wedge v_1 \in \mathcal{V}^{2n} \wedge v_2 \in \mathcal{V}^{2n} \wedge v \notin \mathcal{V}^{2n} \wedge \\ \qquad\qquad\qquad (v \leftrightarrow v_1, v \leftrightarrow v_2 \text{ are edges of } G_V)\}\\ \mathcal{V}^{2n+2} = \mathcal{V}^{2n+1} \cup \{v \mid \exists v' \,.\, v' \in \mathcal{V}^{2n} \wedge v \notin \mathcal{V}^{2n+1} \wedge (v \leftrightarrow v' \text{ is an edge of } G_V)\}\end{array}\right.$$

Practically, the heuristic which consists in first computing the set of new nodes that are doubly linked before nodes that are simply linked introduces more granularity in the calculus of reachable variables. Semantically it privileges variables that are strongly connected with already selected variables, *i.e.* those which are closer to the conclusion. Finally, unreachable variables are added to the fix-point $\mathcal{V}^*$ for completeness reason and let $\mathcal{V}^\infty$ be the set so obtained.

Running example. The sequence of reachable variable sets of verification condition (4) is

$$\begin{array}{ll}
\mathcal{V}^0 & = \{m_x, m_x\_0, m_v, m_y, a, b\}, \\
\mathcal{V}^1 & = \mathcal{V}^0 \cup \{acc\_1, acc\_5, singleton\_2\}, \\
\mathcal{V}^2 & = \mathcal{V}^1 \cup \{acc\_4\}, \\
\mathcal{V}^3 & = \mathcal{V}^2 \cup \{shift\_3\}, \\
\mathcal{V}^* & = \mathcal{V}^3 \text{ and} \\
\mathcal{V}^\infty & = \mathcal{V}^*.
\end{array}$$

5.3 Hypothesis Selection

Suppose given the ordered list of predicates $\mathfrak{L}$, the sequence $\left(\mathcal{V}^n\right)_{n\in\mathbb{N}}$ of reachable variables sets and an hypothesis H. Let i be the counter which represents the level of predicate selection. Similarly, j is the counter corresponding to the level of variables selection. Let V be the set of variables of H augmented with variables resulting from flattening (see Sec. 4.2). Let P be the set of predicates of H.

Different criteria can be used to select an hypothesis H according to its sets V and P. Possible choices are, in increasing order of selectiveness

1. when $V \cap \mathcal{V}^j \neq \emptyset$ or $P \cap \mathfrak{L}[0 \ldots i] \neq \emptyset$: in the hypothesis, there is at least one relevant variable or one relevant predicate;
2. when $card(V \cap \mathcal{V}^j)/card(\mathcal{V}^j)$ and $card(P \cap \mathfrak{L}[0 \ldots i])/card(\mathfrak{L}[0 \ldots i])$ are greater than a threshold;
3. when both $V \subseteq \mathcal{V}^j$ and $P \subseteq \mathfrak{L}[0 \ldots i]$ (i.e. all the hypothesis variables and hypothesis predicates are relevant).

Our experiments on these criteria have shown that a too weak a criterion does not accomplish what it is designed for: too many hypotheses are selected for few iterations, making the prover quickly diverge. In what follows, we only consider the strongest criterion.

Consider a formula resulting from the selection of hypotheses according to the strongest criterion. Then, three issues can arise when discharging it into a prover:

1. The formula is declared to be valid and the procedure ends.
2. The formula is declared to be invalid, maybe because we have omited some hypotheses; we are then left to increment either i or j, i.e. to enlarge either the set of selected predicates or the set of selected variables.
   However, divergence appears when the generation of new literals by a set of axioms is a process that falls in a bottomless pit. Such a generation is

controlled by the presence in the formula of predicates of incriminated axioms. Given a set of predicates and a set of variables, allowing the use of new predicates has a more critical impact than allowing the use of new variables. To conclude, we first increment j, eventually until we reach $\mathcal{V}^{\infty}$ before considering incrementing i. In this later case, j reset to 0.
3. The formula is not decided in less than a given time. If this case occurs after having iteratively incremented i and j, the approach halts. The user is left with an unsatisfactory answer. Otherwise, we propose first to reduce i in order to be in a state where the prover can conclude and to restart the procedure.

Running example. For $\mathcal{L}[0]$, no hypothesis is selected with $\mathcal{V}^0$, $\mathcal{V}^1$, $\mathcal{V}^2$. However with $\mathcal{V}^3$ it yields the VC

$$\begin{array}{l}\mathsf{diff}(m_x, m_x\_0, \mathsf{singleton}(\mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a)), 0))) \Rightarrow \\ \mathsf{diff}(m_x, \mathsf{upd}(m_x\_0, \mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a), 1)), \mathsf{acc}(m_x, b)), \\ \qquad \mathsf{range}(\mathsf{singleton}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a))), 0, 1))\end{array} \tag{6}$$

which does not contain the hypothesis with valid_acc_range which is the expected result. In addition to Simplify and haRVey that already discharged the original VC, Ergo and Yices run successfully on VC (6).

6 Experimentations

In previous work, we presented our context of certification of anotated C programs: we proposed in [16] a separation analysis that allows to greatly simplify the verification conditions generated by a weakest precondition calculus, and thus greatly helps proving such pointer programs. We illustrate the improvements both in term of scaling for codes of large size, and in term of simplification of the reasoning for establishing advanced behaviors.

We have applied this separation analysis on an avionic program that is about 4000 lines of annotated C code, and which aims at analyzing information returned by sensors. This code is critical since it is embedded into a Dassault Aviation airplane. It is a simple program without explicit memory allocation but with many structures, as sketched in the running example. Caduceus/Why yielded about 184000 VCs.

Among of them, 65 were not discharged by any automatic prover with a timeout of 240 seconds for each VC, on an Intel Xeon 3.20GHz with 2Gb of Memory. All the VCs not discharged concern the predicate `not_assigns` and are not so short since they contain an average of 200 literals, corresponding to the same number of hypotheses.

Thanks to the method developed along these lines, all the remaining VCs are automatically discharged in less that 240s, by Simplify notably. In such experiment, i and j have reached the maximal number 1 and 3 respectively.

7 Related Work and Conclusion

In this work, we have presented a new strategy to select relevant hypotheses in formulas issued from program verification. To do so we have combined two separated dependency analyses based on graph computation and graph traversal. Moreover, we have given some heuristics to compute the graph with sufficient granularity. Finally we have shown the pertinence of this approach with a benchmark issued from a real industrial code.

Strategies to simplify the prover task have been widely studied since automated provers exist [25], mainly to propose more efficient deductive systems [25,24,23].

The work presented here can be compared with the set of support (sos) selection strategy [25,20]. Such an approach starts with asking the user to provide an initial sos: it is classically the denial of the conclusion and a subset of hypotheses. It is then restricted to only apply inferences with at least one clause in the sos, consequences being added next into the sos.

Our work can also be viewed as an automatic guess of the initial sos guided by the formula to prove. In this sense, it is close to [18] where initial relevant clauses are selected according to syntactical criteria, i.e. counting matching rates between symbols of any clause and symbols of clauses issued from the conclusion. By considering syntactical filtering on clauses issued from axioms and hypotheses, this later work does not consider the relation between hypotheses, formalized by axioms of the theory: it provides a reduced forward proof. In the opposite, by emphasizing sharing static dependency and dynamic dependency, we are not so far from backward proof search.

By focusing on predicative part of the proof obligation, our objectives are dual than those developed in [15]: this later work concerns boolean verification conditions of any boolean structure whereas we treat predicative formula whose symbols are axiomatized in a quantified theory.

Work presented here does apply a strategy to select relevant axioms of the theory, even if, most of the time, each proof obligation only requires a tiny portion of such a big theory. In [22,8], an instance of such a strategy is presented but it needs a preliminary manual task of classifying axioms. We plan to extend these works in the direction of automation.

References

1. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.
2. Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Computer Aided Verification, 16th International Conference, CAV 2004*, Lecture Notes in Computer Science. Springer, 2004. http://verify.stanford.edu/CVCL/.

3. Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.
4. Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. Technical Report NIII-R0309, Dept. of Computer Science, University of Nijmegen, 2003.
5. Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
6. Sylvain Conchon and Evelyne Contejean. The Ergo automatic theorem prover. http://ergo.lri.fr/.
7. J.-F. Couchot, D. Déharbe, A. Giorgetti, and S. Ranise. Scalable automated proving and debugging of set-based specifications. *Journal of the Brazilian Computer Society*, 9(2):17–36, November 2003. ISSN 0104-6500.
8. David Deharbe and Silvio Ranise. Satisfiability Solving for Software Verification. available at http://www.loria.fr/~ranise/pubs/sttt-submitted.pdf, 2006.
9. Ewen Denney, Bernd Fischer, and Johann Schumann. An empirical evaluation of automated theorem provers in software certification. *International Journal on Artificial Intelligence Tools*, 15(1):81–108, 2006.
10. David Detlefs, Greg Nelson, and James B. Saxe. The Simplify decision procedure (part of ESC/Java). http://research.compaq.com/SRC/esc/simplify/.
11. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, December 1998. See also http://research.compaq.com/SRC/esc/.
12. Edsger W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.
13. Bruno Dutertre and Leonardo de Moura. The YICES SMT Solver. available at http://yices.csl.sri.com/tool-paper.pdf, 2006.
14. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification*, LNCS. Springer-Verlag, 2007.
15. E. Pascal Gribomont. Simplification of boolean verification conditions. *Theoretical Computer Science*, 239(1):165–185, 2000.
16. Thierry Hubert and Claude Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, Braga, Portugal, March 2007. http://www.lri.fr/~marche/hubert07hav.pdf.
17. K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.
18. Jia Meng and L.C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. In *ESCoR: Empirically Successful Computerized Reasoning*, 2006.
19. A. Nonnengart and C. Weidenbach. Computing Small Clause Normal Forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 6, pages 335–367. Elsevier Science, 2001.
20. David A. Plaisted and Adnan H. Yahya. A relevance restriction strategy for automated deduction. *Artificial Intelligence*, 144(1-2):59–93, 2003.
21. Silvio Ranise and David Deharbe. Light-weight theorem proving for debugging and verifying units of code. In *Proc. SEFM'03*, Canberra, Australia, September 2003. IEEE Computer Society Press. http://www.loria.fr/equipes/cassis/softwares/haRVey/.

22. Wolfgang Reif and Gerhard Schellhorn. Theorem proving in large theories. In Maria Paola Bonacina and Ulrich Furbach, editors, *Int. Workshop on First-Order Theorem Proving, FTP'97*, pages 119–124. Johannes Kepler Universität, Linz (Austria), 1997.
23. Larry Wos. Conquering the meredith single axiom. *Journal of Automated Reasoning*, 27(2):175–199, 2001.
24. Larry Wos and Gail W. Pieper. The hot list strategy. *Journal of Automated Reasoning*, 22(1):1–44, 1999.
25. Larry Wos, George A. Robinson, and Daniel F. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM*, 12(4):536–541, 1965.

Redundancy for Geometric Resolution

Hans de Nivelle

Institute of Computer Science,
University of Wroclaw, Poland,
http://www.ii.uni.wroc.pl/∼nivelle

Abstract. We study lemma simplification for geometric resolution, mainly from a theoretical viewpoint. For this purpose we develop a framework of proof permutations, which is somewhat similar to the permutions used in proofs of cut elimination. A side effect of this framework is, that one of the rules of the original geometric resolution calculus, can be simplified into simpler rules, which may have an advantage for proof presentation. Using the framework of proof permutations, we are able to prove theoretical results on proof length for three simplification rules that have been empirically successful in our implementation **geo**. These rules are subsumption, functional reduction, and equality splitting.
This work is work-in-progress, because there exist more simplification principles, for which at this moment we have neither theoretical results, nor practical experience.

1 Introduction

Geometric resolution is a proof search strategy, which was initiated in [3]. It works on a normal form called geometric formula, which it tries to refute by enumerating candidate models.

The variant of geometric resolution studied in this paper was initiated in [6], and differs from the one in [3] in the following ways:

1. The structure of geometric formulas is more restricted, but it is allowed to contain equality.
2. Witnesses for quantifiers are enumerated in the same way as in [4], (and different from [3]), which makes it possible to obtain completeness for first-order logic with equality.
3. From every failed attempt to construct a model, a *lemma* is learnt, which ensures that no similar models will be explored later during proof search.

We now define (our variant) of a geometric formula, then we outline the proof search algorithm for geometric resolution. It makes use of a resolution-like calculus (the geometric resolution calculus) with which it derives a closing lemma from every failed attempt to find a model. (Very similar to the way lemmas are learnt in modern approaches to DPLL, see [7]). The fact that this is always possible, was proven in [6].

After that we discuss what effects one can expect from simplification in geometric resolution, and we compare with simplification for saturation-based calculi ([1]). The main difference is that in geometric resolution, inferences are controlled by the model search algorithm, where in saturation-based calculi, they are made blindly. Because of this, we expect the effect of simplification in geometric resolution to be more predictable.

Definition 1. *We assume an infinite set of* variables $\mathcal{V}$. *A* variable atom *is defined by one of the following two forms:*

- $x_1 \not\approx x_2$, *with* $x_1, x_2 \in \mathcal{V}$ *and* $x_1 \neq x_2$.
- $p(x_1, \dots, x_n)$ *with* $n \geq 0$ *and the* $x_i \in \mathcal{V}$.

There are no constants and no function symbols in variable atoms. There are also no positive equalities. Negative of equalities of the form $v \not\approx v$ are disallowed, because they are trivially false. Geometric formulas are built from variable atoms as follows:

Definition 2. *A* geometric formula *has form*

$$\forall \overline{x}\ A_1(\overline{x}) \wedge \cdots \wedge A_p(\overline{x}) \wedge x_1 \not\approx x'_1 \wedge \cdots \wedge x_q \not\approx x'_q \rightarrow Z(\overline{x}),$$

where $p \geq 0,\ q \geq 0$, *and the* $x_1, x'_1, \dots, x_q, x'_q \in \overline{x} \subseteq \mathcal{V}$.
The right hand side $Z(\overline{x})$ *must have one of the following three forms:*

1. *The false constant* $\bot$.
2. *A non-empty disjunction of non-disequality atoms* $B_1(\overline{x}) \vee \cdots \vee B_r(\overline{x})$ *with* $r > 0$.
3. *An existential formula of form* $\exists y\ B(\overline{x}, y)$ *with* $y \in \mathcal{V}$ *but* $y \notin \overline{x}$. *The variable* y *must occur in* $B(\overline{x}, y)$.

A formula of the first type is called lemma. *A formula of the second type is called* disjunctive. *A formula of the third type is called* existential.

The notations can be clarified as follows:

- In $\forall \overline{x}$, $\overline{x}$ denotes an enumeration of $\overline{x}$, in arbitrary order, mentioning each variable of $\overline{x}$ exactly once. The scope of $\forall \overline{x}$ is the whole geometric formula.
- In $A(\overline{x})$, $\overline{x}$ denotes a sequence of variables from $\overline{x}$. Variables may be repeated, and not all variables need to occur.
- Later in this paper, an expression of form $\Phi(\overline{x})$, $\Psi(\overline{x})$ or $X(\overline{x})$ will denote a conjunction of variable atoms, possibly containing disequality and non-disequality atoms.

It was shown in [6] that every first-order formula can be translated into an equisatisfiable set of geometric formulas. The translation is related to the translation in of [2], and also somewhat related to the translation in [5]. The main difference with [5] is that we do not introduce functionality axioms. (although we will see them back as simplification rules later)

An *interpretation* can be viewed as a set of ground atoms I. An interpretation does not contain disequality atoms. Let $\rho = \forall \overline{x}\ \Phi(\overline{x}) \rightarrow Z(\overline{x})$ be a geometric rule. Let Θ be a substitution that assigns constants occurring in I to the variables in $\overline{x}$. We call the rule ρ *applicable* in I *with substitution* Θ if

1. for each disequality atom $x_1 \not\approx x_2 \in \Phi(\overline{x})$, we have $x_1\Theta \neq x_2\Theta$,
2. for each usual atom $A(\overline{x}) \in \Phi(\overline{x})$, the atom $A(\overline{x})\Theta$ occurs in I, and
3. $Z(\overline{x})\Theta$ is false in I under substitution Θ.
   (The constant $\perp$ is always false. The disjunction $B_1(\overline{x}) \vee \cdots \vee B_r(\overline{x})$ is false if none of the $B_j(\overline{x})\Theta$ occurs in I. A formula of the form $\exists y\ B(\overline{x}, y)$ is false if I contains no constant c, s.t. $B(\overline{x}\Theta, c) \in I$)

As an example, $\forall x\ A(x) \rightarrow B(x)$ is not applicable in $\{A(0), B(0), A(1), B(1)\}$.
The rule $\forall x\ A(x) \rightarrow B(x) \vee C(x)$ is not applicable in $\{A(0), B(0), A(1), C(1)\}$.
It is applicable in $\{A(0), C(0), A(1)\}$ with substitution $\{x := 1\}$.
The rule $\forall xy\ A(x) \wedge B(y) \wedge x \not\approx y \rightarrow \exists z\ C(x, y, z)$ is applicable in
$\{A(0), A(1), B(0), B(1), C(0, 1, 0)\}$ with substitution $\{x := 1, y := 0\}$. It is not applicable with any other substitution.
In geometric resolution, proof search proceeds by a combination of model search and lemma generation. The algorithm recursively tries to extend an interpretation I into a model. (i.e. an interpretation in which no rule is applicable) At each recursive level, the input consists of an interpretation I, and a set of geometric formulas G. When I cannot be extended into a model, the algorithm returns a pair (ρ, Θ) s.t. ρ is a lemma which is applicable on I with substitution Θ. In case the lemma ρ is not already present in G, either I is a model, or there is an applicable rule ρ' which is not a lemma. In that case, the algorithm uses ρ' to extend I, by which it possibly has to backtrack. When backtracking is complete, it uses the geometric resolution rules to derive a ρ. It was proven in [6] that this is always possible, using geometric resolution. We describe the algorithm:

1. Select a rule ρ and a substitution Θ, s.t. ρ is applicable in I with Θ.
2. If no (ρ, Θ) was found, then I is a model. Report I.
3. If ρ is of type 1, then return (ρ, Θ).
4. If ρ is of type 2, then write $\rho = \forall \overline{x}\ \Phi(\overline{x}) \rightarrow B_1(\overline{x}) \vee \cdots \vee B_q(\overline{x})$. Recursively call the algorithm on each $I \cup \{B_j(\overline{x}\Theta)\}$. If one of the recursive calls results in a model, then report this model. Otherwise, the recursive calls will collect a sequence of pairs $(\rho_1, \Theta_1), \ldots, (\rho_q, \Theta_q)$, s.t. each ρ_j is a lemma applicable in the interpretation $I \cup \{B_j(\overline{x}\Theta)\}$ with Θ_j. Using disjunction resolution, it is possible to derive a pair (ρ', Θ'), s.t. ρ' is a lemma applicable in I with substitution Θ'.
5. If ρ is of type 3, then write $\rho = \forall \overline{x}\ \Phi(\overline{x}) \rightarrow \exists y\ B(\overline{x}, y)$. Assume that the constants occurring in I are called $c_0, \ldots, c_{n-1}$. Let c_n be the next constant which is not in I. For each i with $1 \leq i < n$, recursively call the model search algorithm on the interpretation $I \cup \{B(\overline{x}\Theta, c_i)\}$. Also call the model search algorithm on $I \cup \{B(\overline{x}\Theta, c_n)\}$. If one of the recursive calls constructs a model, then report this model. Otherwise, the recursive calls will collect a sequence of pairs $(\rho_1, \Theta_1), \ldots, (\rho_n, \Theta_n)$, s.t. each ρ_i is a lemma which is applicable in

$I \cup \{B(\overline{x}\Theta, c_i)\}$ with substitution Θ_i. Using existential resolution, one can derive a pair (ρ', Θ'), s.t. ρ' is a lemma applicable in I with substitution Θ'.

The most important heuristic of the algorithm is the choice which application (ρ, Θ) should be expanded. In general, it seems sensible to prefer lemmas over rules of other types. Between lemmas, **geo** currently decides by selecting the smaller lemma. Between rules $\forall \overline{x}\ \Phi(\overline{x}) \rightarrow Z(\overline{x})$ of type 2 or type 3, it decides by selecting the application for which $\Phi(\overline{x})\Theta$ has the smallest set of premises, viewing this set as a multiset, and considering older atoms smaller than new atoms. In this way, fairness is guaranteed. However, there is a much variation possible and the effect of the heuristic on the performance of **geo** is largely unexplored.
The main distinction between geometric resolution and saturation-based theorem proving, (e.g. superposition) apart from the different normal form, is the fact that in geometric resolution, proof search is controlled by the model search algorithm. The model search algorithm decides which resolution inferences are made. When it needs a closing lemma, it calls the resolution module with detailed instructions about which inferences should be made. In saturation-based theorem proving, inferences are made essentially 'in a blind way'. Clauses are selected, and all possible inferences are made. This difference has some important consequences for the use of redundancy.
First recall that in saturation-based theorem proving a clause d is called *redundant* when it is implied by a set of clauses $c_1, \ldots, c_n$, such that (somewhat informally) $c_1, \ldots, c_n$ come before d in the multiset order. This notion was introduced in [1], and it is able to prove the completeness of most of the existing simplification rules for superposition theorem proving.
In this paper, we study only a relatively weak version of redundancy in the geometric setting: A lemma λ is redundant when it is implied by a set of lemmas $\lambda_1, \ldots, \lambda_n$, s.t. each of the lemmas $\lambda_1, \ldots, \lambda_n$ would be preferred over λ by the heuristic. This notion would cover λ-subsumption, but also the following example:
$\forall xyz\ S(x,y) \wedge S(x,z) \wedge y \not\approx z \rightarrow \bot$ and $\forall xy\ A(x) \wedge B(y) \wedge x \not\approx y \rightarrow \bot$ make $\forall xyzt\ A(x) \wedge S(x,y) \wedge B(y) \wedge S(y,z) \wedge y \not\approx z \rightarrow \bot$ redundant. If one wants to obtain stronger notions of redundancy, where the implying clauses do not need to be lemmas, then one very probably needs to modify the heuristic. This will be a subject for future study.
Now that we have defined redundancy in our setting, we can discuss the differences with redundancy in saturation-based theorem proving.

1. Redundancy is much more important for saturation-based theorem proving than it is for geometric resolution. In saturation-based theorem proving, redundant clauses can become selected, they will create consequences, which again may be selected, etc. Therefore, high priority should be given to the deletion of redundant clauses.
   In geometric resolution, redundant lemmas will not be selected by the heuristic. Therefore, they will not be used in the derivation of new lemmas.

2. For saturation-based theorem proving, forward redundancy checking is more important than backward redundancy checking. For geometric resolution, forward redundancy checking is wasted effort: The algorithm will never create a redundant lemma. If $\lambda_1, \ldots, \lambda_n$ make λ redundant, then one of $\lambda_1, \ldots, \lambda_n$ would have been a closing lemma, and the algorithm would have reused it.

We have now seen, that whereas the price for tolerating redundant clauses in saturation-based theorem proving can be exponential, it causes only a small overhead in geometric resolution. So we could stop here, and make this is pleasant, short paper.
Unfortunately there is something more to tell, namely about simplification. Simplification is when one derives a consequence λ' of a lemma λ, s.t. λ' (possibly with some other lemmas) makes λ redundant. Although it is not possible that the model search algorithm derives a redundant lemma, it is possible that it derives a lemma that can be simplified. If one does not simplify a lemma that could have been simplified, it will possibly resolve with other lemmas that could have been simplified, and the effect will add up. For this reason, simplification is important for geometric resolution. As an example, consider the following simplification rule, which is an instance of functional reduction. Suppose that the rule system contains only one positive occurrence of A, which has the form $\exists y \; A(y)$. Then in every interpretation constructed by the model search algorithm, there will be at most one constant c such that $A(c)$ occurs in the model. As a consequence, in any lemma containing more than one occurrence of A, all these occurrences can be unified (Because they will be unified anyway in every application of the lemma) If one does not unify all occurrences of A in a lemma, it may resolve with another lemma which also contains multiple occurrences of A. In that case, the resulting lemma will inherit the repeated occurrences of A from both its parents.
In the rest of this paper, we analyze redundancy-based simplification refinements of geometric resolution using proof theoretic methods. The reason for this is that we want to obtain results about proof length.
In saturation-based theorem proving, in general one cannot prove anything about proof length when redundant clauses are removed. The completeness proof implies that the new proof is smaller under the multiset order, but the length of the new proof can actually be bigger. One notable exception to this situation is subsumption. For subsumption, it can be proven that the new proof using the subsuming clause, is not longer than the proof using the subsumed clause.
Our intuition is that the chances of obtaining results about proof length with geometric resolution are better. The reason for this is the fact that the derivations are in some sense more deterministic, because they are governed by the model search algorithm. At this moment, we only have results for a few equality-based refinements, but we think that more results are possible. In order to prepare for proving the results about proof length, we first introduce a modification of the calculus of [6]. The reason for this is that the calculus of [6] is too much tuned towards the model search algorithm. (In particular the $\exists$-resolution rule of [6] is

too complicated to analyze) We show that the new calculus is as strong as the old calculus, and that proofs constructed by the model search algorithm are in a certain normal form, which we call $\approx\exists$-normal form. Note that this change of the calculus has no influence at all on the model search algorithm, because the lemmas derived remain the same.
We then study the effect of proof replacements. Suppose that one has a proof π obtained by a run of the model search algorithm. Let λ be a lemma occurring in π that was redundant at the moment it was used. If λ is made redundant by a set of lemmas $\lambda_1, \dots, \lambda_n$, then it is possible to construct a proof π' which proves λ from $\lambda_1, \dots, \lambda_n$. We can remove λ from π and replace it by the new proof π'. In all probability this new proof will not correspond to a run of the model search algorithm anymore, because of two possible reasons:

1. The new proof is not in $\approx\exists$-normal form.
2. The new proof is in $\approx\exists$-normal form, but is not consistent with the application selection heuristic.

We will introduce a set of proof permutations, with which every proof can be permuted back into $\approx\exists$-normal form. In case a clause was deleted due to redundancy, the proof $\pi[\pi']$ is almost in $\approx\exists$-normal form, except for the path leading to π' and π' itself. We will give examples (functional reduction, nested subsumption) where it can be shown that the permutation back to $\approx\exists$-normal form does not increase the size of the proof. This means that these refinements can be applied without restriction.

2 A Modified Calculus for Geometric Resolution

We present the modified calculus that we used for analyzing proof lengths. The main difference with the calculus of [6] is that we simplified the existential resolution rule and introduced a new rule called equality resolution. Apart from that, the only difference is that we made instantiation explicit. In practice the instantiations are determined by unification, but for analysis it is more convenient to have a calculus with explicit instantiation.

Definition 3. *The new calculus consists of the following rules:*

instantiation: *Let*

$$\rho = \forall \overline{x} \quad \Phi(\overline{x}) \rightarrow Z(\overline{x})$$

be a geometric rule. Let Σ be a substitution of form $x_1 := x_2$, where $x_1 \in \overline{x}$. Then

$$\forall (\overline{x}\Sigma) \quad \Phi(\overline{x}\Sigma) \rightarrow Z(\overline{x}\Sigma)$$

is an instance *of ρ. In case both x_1 and x_2 occur in $\Phi(\overline{x})$ or $Z(\overline{x})$, and $x_1 \neq x_2$, we call the instantiation a* proper *instantiation. In case x_2 does not occur in $\Phi(\overline{x})$ or $Z(\overline{x})$, we call the result a* renaming *of ρ.*

merging: *Let λ be a lemma of form*

$$\forall \overline{x}\ \Phi(\overline{x}) \wedge A(\overline{x}) \wedge A(\overline{x}) \rightarrow \bot.$$

Then the lemma

$$\forall \overline{x}\ \Phi(\overline{x}) \wedge A(\overline{x}) \rightarrow \bot$$

is a merging *of λ. $A(\overline{x})$ can be either a disequality atom, or a usual atom.*

disjunction resolution: *Let*

$$\rho = \ \forall \overline{x}\ \Phi(\overline{x}) \rightarrow B_1(\overline{x}) \vee \cdots \vee B_q(\overline{x})$$

be a disjunctive formula. For $1 \leq j \leq q$, let each

$$\lambda_j = \forall \overline{x}\ \Psi_j(\overline{x}) \wedge B_j(\overline{x}) \rightarrow \bot$$

be a lemma. Then

$$\forall \overline{x}\ \Phi(\overline{x}) \wedge \Psi_1(\overline{x}) \wedge \cdots \wedge \Psi_q(\overline{x}) \rightarrow \bot$$

is a disjunction resolvent of ρ with $\lambda_1, \ldots, \lambda_q$.

existential resolution: *Let $\rho = \forall \overline{x}\ \Phi(\overline{x}) \rightarrow \exists y\ B(\overline{x}, y)$ be an existential formula. Let λ have form*

$$\forall \overline{x}\ \forall y\ \Psi(\overline{x}) \wedge B(\overline{x}, y) \rightarrow \bot.$$

We have $y \notin \overline{x}$. Then

$$\forall \overline{x}\ \Phi(\overline{x}) \wedge \Psi(\overline{x}) \rightarrow \bot$$

is an existential resolvent of ρ with λ.

(degenerated) existential resolution: *Let $\rho = \forall \overline{x}\ \Phi(\overline{x}) \rightarrow \exists y\ B(\overline{x}, y)$ be an existential formula. Let λ have form*

$$\forall \overline{x}\ \forall y\ \Psi(\overline{x}) \rightarrow \bot.$$

We have $y \notin \overline{x}$. Then

$$\forall \overline{x}\ \Phi(\overline{x}) \wedge \Psi(\overline{x}) \rightarrow \bot$$

is a (degenerated) existential resolvent of ρ with λ.

equality resolution: *Let $\rho = \forall \overline{x}\ \Phi(\overline{x}) \wedge x_1 \not\approx x_2 \rightarrow \bot$ be a lemma. Let Σ be the substitution $x_1 := x_2$. Let λ be a lemma that can be written in the form*

$$\lambda = \forall (\overline{x}\Sigma)\ \Psi(\overline{x}\Sigma) \rightarrow \bot.$$

Then the lemma

$$\forall \overline{x}\ \Phi(\overline{x}) \wedge \Psi(\overline{x}) \rightarrow \bot$$

is an equality resolvent of ρ with λ.

Disjunction resolution is the same as hyperresolution. Equality resolution can be explained as follows: If Σ is the substitution $x_1 := x_2$, then $\forall(\overline{x}\Sigma)\Psi(\overline{x}\Sigma) \rightarrow \bot$is equivalent to $\forall\overline{x}\ x_1 \approx x_2 \wedge \Psi(\overline{x}) \rightarrow \bot$. In this formula, the equality can resolve with the disequality in ρ.
Most cases of degenerated existential resolution can be simulated by instantiating y to one of the variables of $\overline{x}$. In that case, one would obtain $\forall\overline{x}\ \Psi(\overline{x}) \rightarrow \bot$ which subsumes the result. We keep the degenerated existential resolution rule because it is still needed for the case where $\overline{x}$ is empty, and in the future we may want to add types to geometric resolution. In that case it may happen that the type of y is not among the types of $\overline{x}$.

Theorem 1. *The calculus of Definition 3 is complete. If $\rho_1, \ldots, \rho_n$ are geometric rules, and λ is a lemma, then if $\rho_1, \ldots, \rho_n \models \lambda$, then λ is provable from $\rho_1, \ldots, \rho_n$.*

For $\lambda = \bot$, completeness follows from the fact that the new calculus can simulate the old calculus. (This will be proven in the next section) For $\lambda \neq \bot$, completeness can be proven in a fairly standard way, by enumerating the models of $\rho_1, \ldots, \rho_n$. Full completeness, (for $\lambda \neq \bot$) is not used in this paper, but it may be important in future studies of other redundancy rules.

3 $\approx\exists$-Normal Derivations

In this section we show that the calculus of Definition 3 can be used in the same way as the calculus of [6] for the generation of closing lemmas during model search. This change of calculus will have no impact on the model search algorithm and on the lemmas that it generates. [1] The reason for introducing the new calculus is that its permutations can be understood more easily. There could also exist an advantage in proof output for external verification, because the new calculus is more standard.
The difference between the old calculus and the calculus of Definition 3, is the replacement of the stronger existential resolution rule of [6] by the combination of a weaker existential resolution rule and equality resolution.
It is not possible to derive the stronger existential resolution rule in the new calculus, but it can be shown that every lemma obtained by an application of strong existential resolution can be obtained by a combination of weak existential resolution and equality resolution.
In [6], existential resolution is always used in the way shown in Figure 1. Figure 2 shows a proof of the same result using (non-generalized) existential resolution and equality resolution. First, the disequalities in $\forall\overline{x}\ \Phi(\overline{x}) \wedge B(\overline{x}, y) \wedge y \not\approx x_1 \wedge \cdots \wedge y \not\approx x_n \rightarrow \bot$ are resolved away one-by-one using equality resolution. On the result, (non-generalized) existential resolution is applied, and the same result is obtained.

[1] actually, they can sometimes be slightly stronger

It can be seen from Figure 2 that equality resolution is never applied 'stand alone' in proofs that are constructed by the model search procedure. Equality resolution is only used for resolving away the disequalities in a lemma of form $\forall \overline{x} y \ \Psi(\overline{x}) \wedge B(\overline{x}, y) \wedge y \not\approx x_1 \wedge \cdots \wedge y \not\approx x_n \rightarrow \bot$ in order 'to prepare it' for an existential resolution step in which $B(\overline{x}, y)$ is resolved away. We call proofs satisfying this condition $\approx\exists$-*normal*. Proofs constructed by the model search algorithm will be always $\approx\exists$-normal.

Fig. 1. Application of General $\exists$-Resolution

Let π be an existential resolution step

$$\frac{\forall \overline{x} \ \Phi(\overline{x}) \rightarrow \exists y \ B(\overline{x}, y) \qquad \forall \overline{x} \ \Psi(\overline{x}) \wedge B(\overline{x}, y) \wedge y \not\approx x_1 \wedge \cdots \wedge y \not\approx x_n \rightarrow \bot}{\forall \overline{x} \ \Phi(\overline{x}) \wedge \Psi(\overline{x}) \rightarrow B_1(\overline{x}, x_1) \vee \cdots \vee B_q(\overline{x}, x_n),}$$

It is used in a proof of form

$$\frac{\pi \qquad \forall \overline{x} \ X_1(\overline{x}) \wedge B_1(\overline{x}, x_1) \rightarrow \bot \qquad \cdots \qquad \forall \overline{x} \ X_n(\overline{x}) \wedge B_1(\overline{x}, x_n) \rightarrow \bot}{\forall \overline{x} \ \Phi(\overline{x}) \wedge \Psi(\overline{x}) \wedge X_1(\overline{x}) \wedge \cdots \wedge X_n(\overline{x}) \rightarrow \bot} \ (\vee\text{-res})$$

4 Redundancy through Proof Permutations

The final goal of the research reported in this paper is to study the effect of redundancy on proof length, using proof transformations. At present, we have only hard results for a restricted form of simplifications but we expect that more results are possible.

We outline the general technique: In case a lemma λ is made redundant by formulas $\rho_1, \ldots, \rho_n$, we take out every application of λ from the proof, and replace it by a proof of $\rho_1, \ldots, \rho_n \models \lambda$. The resulting proof is still a valid proof, but very probably it is not in $\approx\exists$-normal form anymore. The proof can be permuted back into $\approx \exists$-normal form, using proof permutations. If the new proof is not too long, in comparison to the old proof, then efficiency improves when λ is replaced by $\rho_1, \ldots, \rho_n$.
Since our calculus is similar to resolution, all transformations have essentially one of the forms that follow below. (Equality resolution is similar to standard resolution, if one keeps in mind that $\forall (\overline{x}\Sigma) \ \Phi(\overline{x}\Sigma) \rightarrow \bot$ with $\Sigma = \{x_1 := x_2\}$ is equivalent to $\forall \overline{x} \ x_1 \approx x_2 \wedge \Phi(\overline{x}) \rightarrow \bot$)
In both of the permutations, application of the first rule is postponed until after the second rule. The two possibilities depend on where the premise of the second rule originates from. If it originates from only one of the parents of the first rule,

Fig. 2. Reconstruction of General $\exists$-Resolution

For each i with $1 \leq i \leq n$, let Σ_i be the substitution $\{y := x_i\}$.
Then each of the lemmas $\forall\overline{x}\ X_i(\overline{x}) \wedge B(\overline{x}, x_i) \rightarrow \bot$ can be written in the form

$$\forall((\overline{x}y)\Sigma_i)\ X_i((\overline{x}y)\Sigma_i) \wedge B((\overline{x}y)\Sigma_i, y\Sigma_i) \rightarrow \bot,$$

because no variable in $\overline{x}$ is modified by Σ_i, and $y\Sigma_i = x_i$. Let π_1 be the proof

$$\dfrac{\forall((\overline{x}y)\Sigma_1)\ X_1((\overline{x}y)\Sigma_1)\wedge B((\overline{x}y)\Sigma_1, y\Sigma_1) \rightarrow \bot \qquad \forall\overline{x}y\ \Psi(\overline{x})\wedge B(\overline{x}, y)\wedge y \not\approx x_1 \wedge\cdots\wedge y \not\approx x_n \rightarrow \bot}{\forall\overline{x}y\ X_1(\overline{x}) \wedge \Psi(\overline{x}) \wedge B(\overline{x}, y) \wedge B(\overline{x}, y) \wedge y \not\approx x_2 \wedge \cdots \wedge y \not\approx x_n \rightarrow \bot}\ (\approx\text{-res})$$

Similarly, let π_2 be the proof

$$\dfrac{\forall((\overline{x}y)\Sigma_2)\ X_2((\overline{x}y)\Sigma_2) \wedge B((\overline{x}y)\Sigma_2, y\Sigma_2) \rightarrow \bot \qquad \pi_1}{\forall\overline{x}y\ X_1(\overline{x}) \wedge X_2(\overline{x}) \wedge \Psi(\overline{x}) \wedge B(\overline{x}, y) \wedge B(\overline{x}, y) \wedge B(\overline{x}, y) \wedge y \not\approx x_3 \wedge \cdots \wedge y \not\approx x_n \rightarrow \bot}\ (\approx\text{-res})$$

Continuing, one eventually reaches π_n, which has form

$$\dfrac{\forall((\overline{x}y)\Sigma_n)\ X_n((\overline{x}y)\Sigma_n) \wedge B((\overline{x}y)\Sigma_n, y\Sigma_n) \rightarrow \bot \qquad \pi_{n-1}}{\forall\overline{x}y\ X_1(\overline{x}) \wedge X_2(\overline{x}) \wedge \cdots \wedge X_n(\overline{x}) \wedge \Psi(\overline{x}) \wedge B(\overline{x}, y) \wedge \cdots \wedge B(\overline{x}, y) \rightarrow \bot}\ (\approx\text{-res})$$

At this point, one can apply $\exists$-resolution:

$$\dfrac{\forall\overline{x}\ \Phi(\overline{x}) \rightarrow \exists y\ B(\overline{x}, y) \qquad \dfrac{\pi_n}{\forall\overline{x}y\ X_1(\overline{x}) \wedge \cdots \wedge X_n(\overline{x}) \wedge \Psi(\overline{x}) \wedge B(\overline{x}, y) \rightarrow \bot}\ (\text{merging})}{\forall\overline{x}\ \Phi(\overline{x}) \wedge X_1(\overline{x}) \wedge \cdots \wedge X_n(\overline{x}) \wedge \Psi(\overline{x}) \rightarrow \bot}\ (\exists\text{-res})$$

then the transformation is unproblematic, because the size of the proof does not increase. If the premise of the second rule originates from both parents of the first rule, then the size of the proof does increase. We give examples of both situations:
unproblematic:

$$\dfrac{\dfrac{A \lor B \lor R_1 \qquad \neg A \lor R_2}{B \lor R_1 \lor R_2}\,\text{(res)} \qquad \neg B \lor R_3}{R_1 \lor R_2 \lor R_3}\,\text{(res)}$$

permutes into

$$\dfrac{\dfrac{A \lor B \lor R_1 \qquad \neg B \lor R_3}{A \lor R_1 \lor R_3}\,\text{(res)} \qquad \neg A \lor R_2}{R_1 \lor R_2 \lor R_3}\,\text{(res)}$$

problematic:

$$\dfrac{\dfrac{A \lor B \lor R_1 \qquad \neg A \lor B \lor R_2}{B \lor R_1 \lor R_2}\,\text{(res+merging)} \qquad \neg B \lor R_3}{R_1 \lor R_2 \lor R_3}\,\text{(res)}$$

permutes into

$$\dfrac{\dfrac{A \lor B \lor R_1 \qquad \neg B \lor R_3}{A \lor R_1 \lor R_3}\,\text{(res)} \qquad \dfrac{\neg A \lor B \lor R_2 \qquad \neg B \lor R_3}{\neg A \lor R_2 \lor R_3}\,\text{(res)}}{R_1 \lor R_2 \lor R_3}\,\text{(res+merging)}$$

As mentioned above, proof permutations can be used to bring a proof back into $\approx\exists$-normal form, after some lemma λ has been replaced by some formulas $\rho_1, \ldots, \rho_n$ that make it redundant. They also can be used to make a proof consistent with the selection heuristic of the search algorithm.
Unfortunately, each time a rule application from the redundancy proof is permuted down, it may double the part of π that it permutes through, due to the problematic permutations.
We conclude that, when designing redundancy strategies, one should look for strategies that do not cause too much doubling. One of the possible ways to

do this, is by showing that there exists a low upperbound on one of the copies. Since the other copy is not bigger than the original proof, the increase in proof size can be kept small in this way. We end the paper with a few examples, and explain for each of the examples how this can be done.

– We first study functional reduction. Functional reduction exploits the fact that some predicate can be shown to be functional in one or more of its arguments during proof search. Let F be a predicate whose only positive occurrences are in formulas of form $\forall \overline{x}\ \Phi(\overline{x}) \rightarrow \exists y\ F(\overline{x}, y)$. The model search algorithm will create an atom $F(\overline{c}, d_1)$ only in case there exists no other atom of form $F(\overline{c}, d_2)$ in the interpretation. Therefore, in every interpretation I the last argument of F is a function of the other arguments. This fact can be used in simplifications. Whenever a lemma contains two atoms of form $F(\overline{x}, y_1)$ and $F(\overline{x}, y_2)$, the variables y_1 and y_2 can be unified.
In order to ensure that the simplified clause implies the original clause, we add so called *inductive axioms*. The axiom for F is
$$\forall \overline{x} y_1 y_2\ F(\overline{x}, y_1) \wedge F(\overline{x}, y_2) \wedge y_1 \not\approx y_2 \rightarrow \bot.$$
Since F is functional, the inductive axiom will never be applicable. However, it triggers functional reduction. Consider the formula
$$\forall \overline{x} yz\ F(\overline{x}, y) \wedge F(\overline{x}, z) \wedge B(y, z) \rightarrow \bot,$$
which can be functionally reduced into
$$\forall \overline{x} y\ F(\overline{x}, y) \wedge B(y, y) \rightarrow \bot.$$
Using the inductive axiom for F, one can can construct the following proof:
$$\frac{\forall \overline{x} yz\ F(\overline{x}, y) \wedge F(\overline{x}, z) \wedge y \not\approx z \rightarrow \bot \qquad \forall \overline{x} z\ F(\overline{x}, z) \wedge B(z, z) \rightarrow \bot}{\forall \overline{x} yz\ F(\overline{x}, y) \wedge F(\overline{x}, z) \wedge B(y, z) \rightarrow \bot}\ (\approx\text{-res})$$
The $\approx$-resolution has to be permuted down in order to restore $\approx\exists$-normality. We show that during these permutations, the $\approx$-resolution disappears. First the $\approx$-resolution permutes down to the point where either $F(\overline{x}, y)$ or $F(\overline{x}, z)$ is used in disjunction resolution or exists resolution. At this point, since functionality holds for F, $F(\overline{x}, y)$ and $F(\overline{x}, z)$ have to be merged before they are resolved away. But then the $\approx$-resolution will become an instantiation when it permutes with the merging.
Using the same argument, it can be shown that the $\approx$-resolution also disappears in case the $\approx$-resolution is already in $\approx\exists$-normal form.
Note the peculiar way in which the inductive axiom $\forall \overline{x} yz\ F(\overline{x}, y) \wedge F(\overline{x}, z) \wedge y \not\approx z \rightarrow \bot$ was used in the simplification. If F is indeed functional, the axiom will never be applicable, and therefore never occur in a proof. The axiom caused the functional reduction step, but the correctness of the step does not rely on it. Soundness follows from the fact that functional reduction is a form of instantiation.

– Next we consider nested subsumption. Suppose we want to use $a \approx b$ to delete $s(a) \approx s(b)$:

$$\frac{\forall xyt\ S(x,y) \wedge S(x,t) \wedge y \not\approx t \rightarrow \bot \qquad \forall xz\ A(x) \wedge B(z) \wedge x \not\approx z \rightarrow \bot}{\forall xyzt\ A(x) \wedge S(x,y) \wedge B(z) \wedge S(z,t) \wedge y \not\approx t \rightarrow \bot}\ (\approx\text{-res})$$

  The equality resolution step uses the substitution $\{x := z\}$. By the same argument as in the previous case, it can be seen that the $\approx$-resolution will disappear when it is permuted downward.

– The following simplification has no counterpart in resolution, because it can be expressed only with relations. In the presence of

$$\lambda_1 = \forall xyzt\ A(x) \wedge S(x,y) \wedge B(z) \wedge S(z,t) \wedge y \not\approx t \rightarrow \bot,$$

  the formula

$$\lambda_2 = \forall xyzt\ \alpha\beta\gamma\delta\ A(x) \wedge S(x,y) \wedge C(\alpha) \wedge F(y,\alpha,\beta) \wedge$$
$$B(z) \wedge S(z,t) \wedge D(\gamma) \wedge F(t,\gamma,\delta) \wedge \beta \not\approx \delta \rightarrow \bot$$

  can be simplified into

$$\lambda_3 = \forall xzt\alpha\beta\gamma\delta A(x) \wedge S(x,t) \wedge C(\alpha) \wedge F(t,\alpha,\beta) \wedge$$
$$B(z) \wedge S(z,t) \wedge D(\gamma) \wedge F(t,\gamma,\delta) \wedge \beta \not\approx \delta \rightarrow \bot.$$

  λ_2 is an equality resolvent of λ_1 and λ_3. In case all positive occurrences of the predicates S, A, B are in existential formulas, the only way in which formula λ_1 can be used is in an equality resolution, followed by an $\exists$-resolution. It follows that the path from λ_1 towards the $\approx$-resolution must have length 1. Therefore the increase in proof length is at most 1.

5 Conclusions and Future Work

First, we have modified the calculus of [6], in such a way that that the resulting calculus is close to standard resolution. The most notable difference, which is the equality resolution rule, can be explained from the equivalence

$$\forall(\overline{x}\{x_1 := x_2\})\ \Phi(\overline{x}\{x_1 := x_2\}) \rightarrow \bot \quad \Leftrightarrow \forall\overline{x}\ x_1 \approx x_2 \wedge \Phi(\overline{x}) \rightarrow \bot.$$

We intend to change **geo** to use the new calculus, because we expect that it will make proof verification easier.

We have introduced a proof theoretical method with which it is possible to justify some of the successful forms of redundancy in geometric resolution. With this method, we can rigorously prove that functional reduction and nested subsumption (the first two cases in the previous section) do not increase proof length.

At this moment, we do not have sufficient empirical evidence to be able to tell whether a more sophisticated form of analysis will be necessary. In particular, it may be necessary to take reuse of proofs into account. A concrete example where this could be the case is the last case of previous section, (the one with $\lambda_1, \lambda_2, \lambda_3$) It seems likely that also in the case when S occurs positively in disjunctive formulas, the replacement $\lambda_2 \Rightarrow \lambda_3$ would be an improvement. In order to justify such replacements, one could argue that it is very likely (perhaps provable) that the system will encounter situations in which λ_1 alone is applicable, as well as situations where λ_2 is applicable. The effect of the simplification can be viewed as replacing λ_2 by $\lambda_2 \backslash \lambda_1$. If, whenever $\lambda_2 \backslash \lambda_1$ is applied, λ_1 has already been applied before, then the simplification has caused no loss in logical strength. At this moment, we first need to collect some more experience with ad hoc implemented redundancy criteria.

References

1. Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.
2. Peter Baumgartner, Alexander Fuchs, Hans de Nivelle, and Cesare Tinelli. Computing finite models by reduction to function-free clause logic. *Journal of Applied Logic*, 2007.
3. Marc Bezem and Thierry Coquand. Automating coherent logic. In Geoff Sutcliffe and Andrei Voronkov, editors, *LPAR*, volume 3835 of *LNCS*, pages 246–260. Springer, 2005.
4. François Bry and Sunna Torge. A deduction method complete for refutation and finite satisfiability. In Jürgen Dix, Luis Fariñas del Cerro, and Ulrich Furbach, editors, *JELIA*, volume 1489 of *LNCS*, pages 122–138. Springer Verlag, 1998.
5. K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *CADE-19, Workshop W4. Model Computation — Principles, Algorithms, Applications*, 2003.
6. Hans de Nivelle and Jia Meng. Geometric resolution: A proof procedure based on finite model search. In John Harrison, Ulrich Furbach, and Natarajan Shankar, editors, *International Joint Conference on Automated Reasoning 2006*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 303–317, Seattle, USA, August 2006. Springer.
7. Niklas Eén and Niklas Sörensson. An extensible sat solver. `http://www.cs.chalmers.se/~nik/` , 2003.

We give a list of the possible rule permutations. They will not be part of the final version of the paper.

A Pushing Equality Resolution Towards the Root

An equality resolution step on disequality $x_1 \not\approx x_2$ can be pushed towards the root, until it reaches a sequence of disequality steps after which one of x_1, x_2 is resolved away in an existential resolution step. When this point is reached, the result is $\approx\exists$-normal. We list for each of the possible other rules, how the equality resolution permutes through it.
$\vee$-resolution

$$\dfrac{\dfrac{\forall(\overline{x}\Sigma)\ \Psi(\overline{x}\Sigma) \wedge [A_1(\overline{x}\Sigma)] \to \bot \qquad \forall\overline{x}\ X(\overline{x}) \wedge x_1 \not\approx x_2 \wedge [A_1(\overline{x})] \to \bot}{\forall x\ \Psi(\overline{x}) \wedge X(\overline{x}) \wedge A_1(\overline{x}) \wedge A_1(\overline{x}) \to \bot}(\approx\text{-res})}{\forall x\ \Psi(\overline{x}) \wedge X(\overline{x}) \wedge A_1(\overline{x}) \to \bot}(\text{merging})$$

$$\dfrac{\forall\overline{x}\ \Phi(\overline{x}) \to A_1(\overline{x}) \vee \cdots \vee A_p(\overline{x}) \qquad \vdots \qquad \forall\overline{x}\ A_2(\overline{x}) \wedge Y_2(\overline{x}) \to \bot\ \cdots\ \forall\overline{x}\ A_p(\overline{x}) \wedge Y_p(\overline{x}) \to \bot}{\forall\overline{x}\ \Phi(\overline{x}) \wedge \Psi(\overline{x}) \wedge X(\overline{x}) \wedge Y_2(\overline{x}) \wedge \cdots \wedge Y_p(\overline{x}) \to \bot}(\vee\text{-res})$$

If $\forall(\overline{x}\Sigma)\Psi(\overline{x}\Sigma) \wedge [A_1(\overline{x})] \to \bot$ contains A_1, then let π_1 the following proof, which is defined through $\rho_0, \rho_2, \ldots, \rho_p$: Define

$$\rho_0 = \dfrac{\forall\overline{x}\ \Phi(\overline{x}) \to A_1(\overline{x}) \vee \cdots \vee A_p(\overline{x})}{\forall(\overline{x}\Sigma)\ \Phi(\overline{x}\Sigma) \to A_1(\overline{x}\Sigma) \vee \cdots \vee A_p(\overline{x}\Sigma)}(\text{int})$$

$$\rho_2 = \dfrac{\forall\overline{x}\ A_2(\overline{x}) \wedge Y_2(\overline{x}) \to \bot}{\forall(\overline{x}\Sigma)\ A_2(\overline{x}\Sigma) \wedge Y_2(\overline{x}\Sigma) \to \bot}(\text{inst}) \qquad \cdots \qquad \dfrac{\forall\overline{x}\ A_p(\overline{x}) \wedge Y_p(\overline{x}) \to \bot}{\forall(\overline{x}\Sigma)\ A_p(\overline{x}\Sigma) \wedge Y_p(\overline{x}\Sigma) \to \bot}(\text{inst})$$

$$\pi = \dfrac{\rho_0 \qquad \forall(\overline{x}\Sigma)\ \Psi(\overline{x}\Sigma) \wedge A_1(\overline{x}\Sigma) \to \bot \qquad \rho_2 \qquad \cdots \qquad \rho_p}{\Phi(\overline{x}\Sigma) \wedge \Psi(\overline{x}\Sigma) \wedge Y_2(\overline{x}\Sigma) \wedge \cdots \wedge Y_p(\overline{x}\Sigma) \to \bot}(\vee\text{-res})$$

Otherwise, let π_1 be just $\forall(\overline{x}\Sigma)\ \Psi(\overline{x}\Sigma) \to \bot$.

Similarly, if $\forall\overline{x}\ X(\overline{x}) \wedge x_1 \not\approx x_2 \wedge [A_1(\overline{x})] \to \bot$ does contain $A_1(\overline{x})$, then let π_2 be the following proof:

$$\dfrac{\begin{array}{c}\forall\overline{x}\ \Phi(\overline{x}) \to A_1(\overline{x}) \vee \cdots \vee A_p(\overline{x}) \qquad \forall\overline{x}\ X(\overline{x}) \wedge x_1 \not\approx x_2 \wedge A_1(\overline{x}) \to \bot \\ \forall\overline{x}\ A_2(\overline{x}) \wedge X_2(\overline{x}) \to \bot\ \cdots\ \forall\overline{x}\ A_p(\overline{x}) \wedge X_p(\overline{x}) \to \bot\end{array}}{\forall\overline{x}\ \Phi(\overline{x}) \wedge X(\overline{x}) \wedge x_1 \not\approx x_2 \wedge Y_2(\overline{x}) \wedge \cdots \wedge Y_p(\overline{x}) \to \bot}(\vee\text{-res})$$

Otherwise let π_2 be just $\forall\overline{x}\ X(\overline{x}) \wedge x_1 \not\approx x_2 \to \bot$.
Finally, we can construct

$$\dfrac{\pi_1 \qquad\qquad \pi_2}{}(\approx\text{-res})$$

$\forall \overline{x}\ \Phi(\overline{x}) \wedge \Psi(\overline{x}) \wedge X(\overline{x}) \wedge Y_2(\overline{x}) \wedge \cdots \wedge Y_p(\overline{x}) \rightarrow \bot$

∃-resolution
Consider the following proof, in which existential resolution is applied on the result of an equality resolution step.

$$\dfrac{\forall \overline{x}\ \Phi(\overline{x}) \rightarrow \exists y\ B(\overline{x},y) \qquad \dfrac{\forall(\overline{x}\Sigma)y\ \Psi(\overline{x}\Sigma)\wedge[B(\overline{x}\Sigma,y)]\rightarrow\bot \qquad \forall\overline{x}y\ X(\overline{x})\wedge x_1\not\approx x_2\wedge[B(\overline{x},y)]\rightarrow\bot}{\forall\overline{x}y\ \Psi(\overline{x})\wedge X(\overline{x})\wedge B(\overline{x},y)\rightarrow\bot}\,(\approx\text{-res})}{\forall\overline{x}\ \Phi(\overline{x})\wedge\Psi(\overline{x})\wedge X(\overline{x})\rightarrow\bot}\,(\exists\text{-res})$$

The notation $[B(\overline{x},y)]$ means that $B(\overline{x},y)$ is optional. One of the premises of the $\approx$-resolution step must contain $B(\overline{x},y)$, because otherwise the next ∃-resolution step would be not possible. Σ denotes the substitution $x_1 := x_2$. We have $y \neq x_1$, for the reasons mentioned before. For this reason, there is no need to apply Σ on y.
Write λ_1 for the lemma $\forall(\overline{x}\Sigma)y\ \Psi(\overline{x}\Sigma) \wedge [B(\overline{x}\Sigma,y)] \rightarrow \bot$. If λ_1 does contain $B(\overline{x}\Sigma,y)$, then let π_1 denote the following proof:

$$\dfrac{\dfrac{\forall \overline{x}\ \Phi(\overline{x}) \rightarrow \exists y\ B(\overline{x},y)}{\forall(\overline{x}\Sigma)\ \Phi(\overline{x}\Sigma) \rightarrow \exists y\ B(\overline{x}\Sigma,y)}\,(\text{inst}) \qquad \forall(\overline{x}\Sigma)y\ \Psi(\overline{x}\Sigma)\wedge B(\overline{x}\Sigma,y)\rightarrow\bot}{\forall(\overline{x}\Sigma)\ \Phi(\overline{x}\Sigma)\wedge\Psi(\overline{x}\Sigma)\rightarrow\bot}\,(\exists\text{-res})$$

Otherwise, let π_1 be just λ_1. Write λ_2 for the other premise $\forall\overline{x}y\ X(\overline{x}) \wedge x_1 \not\approx x_2 \wedge [B(\overline{x},y)] \rightarrow \bot$. If λ_2 does contain $B(\overline{x},y)$, then let π_2 be the following proof:

$$\dfrac{\forall \overline{x}\ \Phi(\overline{x}) \rightarrow \exists y\ B(\overline{x},y) \qquad \forall\overline{x}y\ X(\overline{x})\wedge x_1\not\approx x_2\wedge B(\overline{x},y)\rightarrow\bot}{\forall\overline{x}\ \Phi(\overline{x})\wedge X(\overline{x})\wedge x_1\not\approx x_2\rightarrow\bot}\,(\exists\text{-res})$$

Otherwise, let π_2 be just λ_2. Finally, we can construct

$$\dfrac{\pi_1 \qquad \pi_2}{\forall\overline{x}\ \Phi(\overline{x})\wedge X(\overline{x})\wedge\Psi(\overline{x})\rightarrow\bot}\,(\approx\text{-res+merging})$$

For $i = 1$ or $i = 2$, if λ_i contains $B(\overline{x},y)$, (or $B(\overline{x}\Sigma,y)$), then the conclusion of π_i contains $\Phi(\overline{x})$. (or $\Phi(\overline{x}\Sigma)$). In both cases, the final result will contain $\Phi(\overline{x})$. If both of λ_1, λ_2 contain $B(\overline{x},y)$, then the result will receive two copies of $\Phi(\overline{x})$, which have to be merged.
≈-resolution

Our concern is to arrange applications of $\approx$-resolution in such a way, that the resulting proof is $\approx \exists$-normal.

$$\dfrac{\dfrac{\dfrac{\forall \overline{x}\ \Phi(\overline{x}) \wedge x_1 \not\approx x_2 \wedge x_1' \not\approx x_2' \to \bot \qquad \forall (\overline{x}\Theta)\ \Psi(\overline{x}\Theta) \wedge x_1\Theta \not\approx x_2\Theta \to \bot}{\forall \overline{x}\ \Phi(\overline{x}) \wedge \Psi(\overline{x}) \wedge x_1 \not\approx x_2 \wedge x_1 \not\approx x_2 \to \bot}\,(\approx\text{-res})}{\forall \overline{x}\ \Phi(\overline{x}) \wedge \Psi(\overline{x}) \wedge x_1 \not\approx x_2 \to \bot}\,(\text{merging}) \qquad \forall (\overline{x}\Sigma)\ X(\overline{x}\Sigma) \to \bot}{\forall \overline{x}\ \Phi(\overline{x}) \wedge \Psi(\overline{x}) \wedge X(\overline{x}) \to \bot}\,(\approx\text{-res})$$

Θ denotes the substitution $x_1' := x_2'$, and Σ denotes the substitution $x_1 := x_2$. We may assume that the second equality resolution step is already part of a normal sequence. (which will end with poor x_1 being resolved away)
If $x_1 = x_1'$, nothing needs to be changed, because the proof is already normal.
If $x_1 = x_2'$, then define $\Theta' = (x_2' := x_1')$. The first equality resolution can be replaced by
First define π_1 as the following proof:

$$\dfrac{\forall \overline{x}\ \Phi(\overline{x}) \wedge x_1 \not\approx x_2 \wedge x_1' \not\approx x_2' \to \bot \qquad \forall (\overline{x}\Sigma)\ X(\overline{x}\Sigma) \to \bot}{\forall \overline{x}\ \Phi(\overline{x}) \wedge x_1' \approx x_2' \wedge X(\overline{x}) \to \bot}\,(\approx\text{-res})$$

Before we can define π_2, we need a property of substitution. Obviously, Σ is the mgu of x_1 and x_2. Let Σ' be the substitution $x_1\Theta := x_2\Theta$. Then clearly for the substitution $\Theta \cdot \Sigma'$ holds that $x_1\Theta \cdot \Sigma' = x_2\Theta \cdot \Sigma'$. Therefore, there exists a substitution Θ', s.t. $\Theta \cdot \Sigma' = \Sigma \cdot \Theta'$.
We define π_2 as the proof:

$$\dfrac{\forall (\overline{x}\Theta)\ \Psi(\overline{x}\Theta) \wedge x_1\Theta \not\approx x_2\Theta \to \bot \qquad \begin{array}{c} \dfrac{\forall (\overline{x}\Sigma)\ X(\overline{x}\Sigma) \to \bot}{\forall (\overline{x}\Sigma\Theta')\ X(\overline{x}\Sigma\Theta') \to \bot}\,(\text{inst}) \\ = \\ \forall (\overline{x}\Theta\Sigma')\ X(\overline{x}\Theta\Sigma') \to \bot \end{array}}{\forall (\overline{x}\Theta)\ \Psi(\overline{x}\Theta) \wedge X(\overline{x}\Theta) \to \bot}\,(\approx\text{-res})$$

It remains to apply $\approx$-resolution on the results of π_1 and π_2.
We know come to the case where the instantiated premise of an equality resolution step is derived by another equality resolution step:

$$\dfrac{\dfrac{\forall (\overline{x}\Sigma\Theta)\ \Phi(\overline{x}\Sigma\Theta) \to \bot \qquad \forall (\overline{x}\Sigma)\ x_1' \not\approx x_2' \wedge \Psi(\overline{x}\Sigma) \to \bot}{\forall (\overline{x}\Sigma)\ \Phi(\overline{x}\Sigma) \wedge \Psi(\overline{x}\Sigma) \to \bot}\,(\approx\text{-res}) \qquad \forall \overline{x}\ X(\overline{x}) \wedge x_1 \not\approx x_2 \to \bot}{\forall \overline{x}\ \Phi(\overline{x}) \wedge \Psi(\overline{x}) \wedge X(\overline{x}) \to \bot}\,(\approx\text{-res})$$

In this proof, Σ denotes the substitution $x_1 := x_2$, and Θ denotes the substitution $x_1' := x_2'$. It can be assumed that $x_1 \neq x_1'$, because otherwise Θ would have no effect, and the first equality resolution step could be trivially removed.

Similarly, if $x_1 = x_2'$, we would have either $x_1 = x_2$, in which case Σ would be the empty substitution and it would be possible to remove the last equality resolution step, or x_1 does not occur in $\overline{x}\Sigma$. In that case, Θ would be renaming on $\overline{x}\Sigma$, and the upper equality resolution step could be replaced by an instantiation using Θ^{-1}. We define π_1 as the following proof:

$$\dfrac{\forall(\overline{x}\Sigma\Theta)\ \Phi(\overline{x}\Sigma\Theta) \to \bot \qquad \dfrac{\forall\overline{x}\ X(\overline{x}) \wedge x_1 \not\approx x_2 \to \bot}{\forall(\overline{x}\Theta)\ X(\overline{x}\Theta) \wedge x_1\Theta \not\approx x_2\Theta \to \bot}\ (\text{inst})}{\forall(\overline{x}\Theta)\ \Phi(\overline{x}\Theta) \wedge X(\overline{x}\Theta) \to \bot}\ (\approx\text{-res})$$

In order to show that this proof is correct, we show that

$$\Sigma \cdot \Theta = \Theta \cdot \{x_1\Theta := x_2\Theta\}.$$

Since $x_1 \not\approx x_1'$, we have $x_1\Theta = x_1$. As a consequence, the domains of the substitutions Σ, Θ, $\{x_1\Theta := x_2\Theta\}$ consist only of the variables x_1 and of x_1'. It is therefore sufficient to compare the behaviour of the substitutions on x_1 and x_1'.

$$x_1\Sigma \cdot \Theta = x_2\Theta, \qquad x_1\Theta \cdot \{x_1\Theta := x_2\Theta\} = x_2\Theta.$$

$$x_1'\Sigma \cdot \Theta = x_1'\Theta = x_2', \qquad x_1'\Theta \cdot \{x_1\Theta := x_2\Theta\} = x_2'\{x_1\Theta := x_2\Theta\} = x_2'.$$

The last step is correct because $x_1\Theta = x_1$, and $x_1 \not\approx x_2'$.
Next, let π_2 be the proof:

$$\dfrac{\forall(\overline{x}\Sigma)\ x_1' \not\approx x_2' \wedge \Psi(\overline{x}\Sigma) \to \bot \qquad \forall\overline{x}\ X(\overline{x}) \wedge x_1 \not\approx x_2 \to \bot}{\forall\overline{x}\ x_1' \not\approx x_2' \wedge \Psi(\overline{x}) \wedge X(\overline{x}) \to \bot}\ (\approx\text{-res})$$

It remains to apply equality resolution on the results of π_1 and π_2.
instantiation
Consider the following proof

$$\dfrac{\dfrac{\forall(\overline{x}\Sigma)\ \Phi(\overline{x}\Sigma) \to \bot \qquad \forall\overline{x}\ \Psi(\overline{x}) \wedge x_1 \not\approx x_2 \to \bot}{\forall\overline{x}\ \Phi(\overline{x}) \wedge \Psi(\overline{x}) \to \bot}\ (\approx\text{-res})}{\forall(\overline{x}\Theta)\ \Phi(\overline{x}\Theta) \wedge \Psi(\overline{x}\Theta) \to \bot}\ (\text{inst})$$

Σ denotes the substitution $\{x_1 := x_2\}$. Θ is the substitution used in the instantiation. Define $\Sigma' = \{x_1\Theta := x_2\Theta\}$. For the substitution $\Theta \cdot \Sigma'$ holds that $x_1\Theta \cdot \Sigma' = x_2\Theta \cdot \Sigma'$.

The substitution Σ is clearly the mgu of x_1 and x_2. Hence there exists a substitution Θ', s.t. $\Sigma \cdot \Theta' = \Theta \cdot \Sigma'$, and we can construct the following proof:

$$\dfrac{\dfrac{\forall(\overline{x}\Sigma)\ \Psi(\overline{x}\Sigma) \to \bot}{\begin{array}{c}\forall(\overline{x}\Sigma\Theta')\ \Psi(\overline{x}\Sigma\Theta') \to \bot\\ =\\ \forall(\overline{x}\Theta\Sigma')\ \Psi(\overline{x}\Theta\Sigma') \to \bot\end{array}}\text{(inst)} \qquad \dfrac{\forall\overline{x}\ \Phi(\overline{x}) \wedge x_1 \not\approx x_2 \to \bot}{\forall(\overline{x}\Theta)\ \Phi(\overline{x}\Theta) \wedge x_1\Theta \not\approx x_2\Theta \to \bot}\text{(inst)}}{\forall(\overline{x}\Theta)\ \Phi(\overline{x}\Theta) \wedge \Psi(\overline{x}\Theta) \to \bot}(\approx\text{-res})$$

In case that $x_1\Theta = x_2\Theta$, Σ' will be the empty substitution, and the equality resolution step will transform itself into an instantiation step.

SOMETHING about preservance $\approx \exists$-normality should be said at the end. The essential thing is that both π_1 and π_2 consists of an equality resolution on $x_1 = x_1\Theta$.

B Pushing $\vee$-Resolution Towards the Root

We list the possible other rules, and show how the $\vee$-resolution step permutes through it.
$\vee$-resolution

Let π be the following disjunction resolution proof:

$$\dfrac{\dfrac{\begin{array}{l}\forall\overline{x}\ \Phi(\overline{x}) \to A_1(\overline{x}) \vee \cdots \vee A_p(\overline{x})\\ \qquad\qquad \forall\overline{x}\ A_1(\overline{x}) \vee [B_1(\overline{x})]\ \wedge X_1(\overline{x}) \to \bot\ \cdots\ \forall\overline{x}\ A_p(\overline{x}) \vee [B_1(\overline{x})]\ \wedge X_p(\overline{x}) \to \bot\end{array}}{\forall\overline{x}\ \Phi(\overline{x}) \wedge B_1(\overline{x}) \wedge \cdots \wedge B_1(\overline{x}) \wedge X_1(\overline{x}) \wedge \cdots \wedge X_p(\overline{x}) \to \bot}(\vee\text{-res})}{\forall\overline{x}\ \Phi(\overline{x}) \wedge B_1(\overline{x}) \wedge X_1(\overline{x}) \wedge \cdots \wedge X_p(\overline{x}) \to \bot}\text{(merging)}$$

The square brackets are used for indicating the fact that $[B_1(\overline{x})]$ is optional. However, at least one of the premises of π must contain $B_1(\overline{x})$. In the result of π, the atom $B_1(\overline{x})$ is resolved away in another disjunction resolution step:

$$\dfrac{\begin{array}{l}\qquad\qquad\qquad\qquad\quad \pi \qquad\qquad \forall\overline{x}\ B_2(\overline{x}) \wedge Y_2(\overline{x}) \to \bot\ \cdots\ \forall\overline{x}\ B_q(\overline{x}) \wedge Y_q(\overline{x}) \to \bot\\ \forall\overline{x}\ \Psi(\overline{x}) \to B_1(\overline{x}) \vee \cdots \vee B_q(\overline{x})\end{array}}{\forall\overline{x}\ \Psi(\overline{x}) \wedge \Phi(\overline{x}) \wedge X_1(\overline{x}) \wedge \cdots \wedge X_p(\overline{x}) \wedge Y_2(\overline{x}) \wedge \cdots \wedge Y_q(\overline{x}) \to \bot.}\vee\text{-res}$$

For i with $1 \leq i \leq p$, define $\lambda_i = \forall\overline{x}\ A_i(\overline{x}) \vee [B_1(\overline{x}] \wedge X_i(\overline{x}) \to \bot$. If λ_i contains $B_1(\overline{x})$, then let π_i be the following proof:

$$\forall\overline{x}\ \Psi(\overline{x}) \to B_1(\overline{x}) \vee \cdots \vee B_q(\overline{x}) \qquad \lambda_i \qquad \forall\overline{x}\ B_2(\overline{x}) \wedge Y_2(\overline{x}) \to \bot\ \cdots\ \forall\overline{x}\ B_q(\overline{x}) \wedge Y_q(\overline{x}) \to \bot$$

$$\frac{\qquad}{\forall\overline{x}\;\Psi(\overline{x})\land A_i(\overline{x})\land X_i(\overline{x})\land Y_2(\overline{x})\land\cdots\land Y_q(\overline{x})\to\bot}\;(\lor\text{-res})$$

In case λ_i does not contain $B_1(\overline{x})$, π_i is just the proof of λ_i. On the results of the π_i, the first disjunctive rule can be applied:

$$\frac{\forall\overline{x}\;\Phi(\overline{x})\to A_1(\overline{x})\lor\cdots\lor A_p(\overline{x})\qquad \pi_1\qquad\cdots\qquad\pi_p}{\forall\overline{x}\;\Phi(\overline{x})\land\Psi(\overline{x})\land X_1(\overline{x})\land\cdots\land X_p(\overline{x})\land Y_2(\overline{x})\land\cdots\land Y_q(\overline{x})\to\bot}\;(\lor\text{-res+merging})$$

∃-resolution:
Consider a proof, where first a disjunction resolution is applied, and after that existential resolution:

$$\frac{\forall\overline{x}\;\Psi(\overline{x})\to\exists y\;B(\overline{x},y)\qquad \dfrac{\dfrac{\begin{array}{l}\forall\overline{x}\;\Phi(\overline{x})\to A_1(\overline{x})\lor\cdots\lor A_p(\overline{x})\\ \qquad\forall\overline{x}y\;A_1(\overline{x})\land X_1(\overline{x})[\land B(\overline{x},y)]\to\bot\qquad\cdots\qquad\forall\overline{x}y\;A_p(\overline{x})\land X_p(\overline{x})[\land B(\overline{x},y)]\to\bot\end{array}}{\forall\overline{x}y\;\Phi(\overline{x})\land X_1(\overline{x})\land\cdots\land X_p(\overline{x})\land B(\overline{x},y)\land\cdots\land B(\overline{x},y)\to\bot}\;(\lor\text{-res})}{\forall\overline{x}y\;\Phi(\overline{x})\land X_1(\overline{x})\land\cdots\land X_p(\overline{x})\land B(\overline{x},y)\to\bot}\;(\text{merging})}{\forall\overline{x}\;\Psi(\overline{x})\land\Phi(\overline{x})\land X_1(\overline{x})\land\cdots\land X_p(\overline{x})\to\bot.}\;(\exists\text{-res})$$

Again, the notation $[B(\overline{x},y)]$ means that $B(\overline{x},y)$ is optional. For each i with $1\le i\le p$, write λ_i for the lemma $\forall\overline{x}y\;A_i(\overline{x})\land X_i(\overline{x})\land[B(\overline{x},y)]\to\bot$. For those λ_i that do contain $B(\overline{x},y)$ let π_i be the following proof:

$$\frac{\forall\overline{x}\;\Psi(\overline{x})\to\exists y\;B(\overline{x},y)\qquad\forall\overline{x}y\;A_i(\overline{x})\land X_i(\overline{x})\land B(\overline{x},y)\to\bot}{\forall\overline{x}\;\Psi(\overline{x})\land A_i(\overline{x})\land X_i(\overline{x})\to\bot}\;(\exists\text{-res})$$

For the other λ_i, let π_i be just λ_i. Using $\pi_1,\ldots,\pi_p$, we can construct the permuted proof:

$$\frac{\dfrac{\forall\overline{x}\;\Phi(\overline{x})\to A_1(\overline{x})\lor\cdots\lor A_p(\overline{x})\qquad\pi_1\qquad\cdots\qquad\pi_p}{\forall\overline{x}\;\Phi(\overline{x})\land\Psi(\overline{x})\land\cdots\land\Psi(\overline{x})\land X_1(\overline{x})\land\cdots\land X_p(\overline{x})\to\bot}\;(\lor\text{-res})}{\forall\overline{x}\;\Phi(\overline{x})\land\Psi(\overline{x})\land\cdots\land X_1(\overline{x})\land\cdots\land X_p(\overline{x})\to\bot}\;(\text{merging})$$

≈-resolution:
We first study the case where the disjunction resolution is used for deriving the instantiated premise of the equality resolution. Consider a proof of the following form, where the result of a disjunction resolution application is the instianted premise of an equality resolution application:

$$\dfrac{\dfrac{\begin{array}{l}\forall \overline{x}\ \Phi(\overline{x}) \rightarrow A_1(\overline{x}) \vee \cdots \vee A_p(\overline{x}) \\ \qquad\qquad \forall \overline{x}\ A_1(\overline{x}) \wedge \Psi_1(\overline{x}, y\Sigma) \rightarrow \bot \quad \cdots \quad \forall \overline{x}\ A_p(\overline{x}) \wedge \Psi_p(\overline{x}, y\Sigma) \rightarrow \bot\end{array}}{\forall \overline{x}\ \Phi(\overline{x}) \wedge \Psi_1(\overline{x}, y\Sigma) \wedge \cdots \wedge \Psi_p(\overline{x}, y\Sigma) \rightarrow \bot}\ (\vee\text{-res}) \qquad \begin{array}{c}\vdots\end{array} \qquad \forall \overline{x}y\ X(\overline{x}, y) \wedge y \not\approx x \rightarrow \bot}{\forall \overline{x}y\ \Phi(\overline{x}) \wedge \Psi_1(\overline{x}, y) \wedge \cdots \wedge \Psi_p(\overline{x}, y) \wedge X(\overline{x}, y) \rightarrow \bot}\ (\approx\text{-res})$$

In this proof, Σ denotes the substitution $y := x$. We have $x \in \overline{x}$, and $y \notin \overline{x}$. Because of this, we have $(\overline{x}y)\Sigma = \overline{x}$, and also $\overline{x}\Sigma = \overline{x}$.

¿From Figure 2, we know that every equality resolution step instantiating y belongs to a sequence of equality resolution steps in which y is instantiated, and that ands in an existential resolution step in which y is resolved away.

We are interested in proofs that have been constructed by the model generation algorithm, and in which the model generation had the choice between applying the disjunctive rule, or an existential rule. From this, it follows that y does not occur in the disjunctive rule, and that therefore the proof can be written in the form above.

For each i with $1 \leq i \leq p$, let π_i be the following proof:

$$\dfrac{\forall \overline{x}\ A_i(\overline{x}) \wedge \Psi_i(\overline{x}, y\Sigma) \rightarrow \bot \qquad \forall \overline{x}y\ X(\overline{x}, y) \wedge y \not\approx x \rightarrow \bot}{\forall \overline{x}y\ A_i(\overline{x}) \wedge \Psi_i(\overline{x}, y) \wedge X(\overline{x}, y) \rightarrow \bot}\ (\approx\text{-res})$$

And the complete proof is:

$$\dfrac{\dfrac{\forall \overline{x}\ \Phi(\overline{x}) \rightarrow A_1(\overline{x}) \vee \cdots \vee A_p(\overline{x}) \qquad \pi_1 \qquad \cdots \qquad \pi_p}{\forall \overline{x}y\ \Phi(\overline{x}) \wedge \Psi_1(\overline{x}, y) \vee \cdots \vee \Psi_p(\overline{x}, y) \wedge X(\overline{x}, y) \vee \cdots \vee X(\overline{x}, y) \rightarrow \bot}\ \vee\text{-res}}{\forall \overline{x}y\ \Phi(\overline{x}) \wedge \Psi_1(\overline{x}, y) \wedge \cdots \wedge \Psi_p(\overline{x}, y) \wedge X(\overline{x}, y) \rightarrow \bot}\ (\text{merging})$$

Next we come to the situation where the disjunction resolution derives the premise with the disequality atom.

$$\dfrac{\dfrac{\begin{array}{l}\forall \overline{x}\ \Phi(\overline{x}) \rightarrow A_1(\overline{x}) \vee \cdots \vee A_p(\overline{x}) \\ \qquad \forall \overline{x}y\ A_1(\overline{x}) \wedge \Psi_1(\overline{x}, y) \wedge y \not\approx x \rightarrow \bot \quad \cdots \quad \forall \overline{x}y\ A_p(\overline{x}) \wedge \Psi_p(\overline{x}, y) \wedge y \not\approx x \rightarrow \bot\end{array}}{\forall \overline{x}y\ \Phi(\overline{x}) \wedge \Psi_1(\overline{x}, y) \wedge \cdots \wedge \Psi_p(\overline{x}, y) \wedge y \not\approx x \rightarrow \bot}\ (\vee\text{-res} + \text{merging}) \qquad \begin{array}{c}\vdots\end{array} \qquad \forall \overline{x}\ X(\overline{x}, y\Sigma) \rightarrow \bot}{\forall \overline{x}y\ \Phi(\overline{x}) \wedge \Psi_1(\overline{x}, y) \wedge \cdots \wedge \Psi_p(\overline{x}, y) \wedge X(\overline{x}, y) \rightarrow \bot}\ (\approx\text{-res})$$

As above, Σ denotes the substitution $y := x$. Also here, the variable y does not occur in the disjunctive rule, although this fact is not needed for the proof permutation in the present case. For each i with $1 \le i \le p$, let π_i be the following proof:

$$\dfrac{\forall \overline{x} y \; A_i(\overline{x}) \wedge \Psi_i(\overline{x}, y) \wedge y \not\approx x \rightarrow \bot \qquad \forall \overline{x} \; X(\overline{x}, y\Sigma) \rightarrow \bot}{\forall \overline{x} y \; A_i(\overline{x}) \wedge \Psi_i(\overline{x}, y) \wedge X(\overline{x}, y) \rightarrow \bot} \; (\approx\text{-res})$$

The complete, permuted proof is:

$$\dfrac{\dfrac{\forall \overline{x} \; \Phi(\overline{x}) \rightarrow A_1(\overline{x}) \vee \cdots \vee A_p(\overline{x}) \qquad \pi_1 \qquad \cdots \qquad \pi_p}{\forall \overline{x} y \; \Phi(\overline{x}) \wedge \Psi_1(\overline{x}, y) \vee \cdots \vee \Psi_p(\overline{x}, y) \wedge X(\overline{x}, y) \vee \cdots \vee X(\overline{x}, y) \rightarrow \bot} \; (\vee\text{-res})}{\forall \overline{x} y \; \Phi(\overline{x}) \wedge \Psi_1(\overline{x}, y) \wedge \cdots \wedge \Psi_p(\overline{x}, y) \wedge X(\overline{x}, y) \rightarrow \bot} \; (\text{merging})$$

instantiation
Permuting an instance of disjunction resolution through an instantiation is straightforward:

$$\dfrac{\dfrac{\forall \overline{x} \; \Phi(\overline{x}) \rightarrow A_1(\overline{x}) \vee \cdots \vee A_p(\overline{x}) \qquad \forall \overline{x} \; A_1(\overline{x}) \wedge \Psi_1(\overline{x}) \rightarrow \bot \qquad \cdots \qquad \forall \overline{x} \; A_p(\overline{x}) \wedge \Psi_p(\overline{x}) \rightarrow \bot}{\forall \overline{x} \; \Phi(\overline{x}) \wedge \Psi_1(\overline{x}) \wedge \cdots \wedge \Psi_p(\overline{x}) \rightarrow \bot} \; (\vee\text{-res})}{\forall (\overline{x}\Sigma) \; \Phi(\overline{x}\Sigma) \wedge \Psi_1(\overline{x}\Sigma) \wedge \cdots \wedge \Psi_p(\overline{x}\Sigma) \rightarrow \bot} \; (\text{inst})$$

For i with $1 \le i \le p$, let π_i be the following instantiation:

$$\dfrac{\forall \overline{x} \; A_i(\overline{x}) \wedge \Psi_i(\overline{x}) \rightarrow \bot}{\forall (\overline{x}\Sigma) \; A_i(\overline{x}\Sigma) \wedge \Psi_i(\overline{x}\Sigma) \rightarrow \bot} \; (\text{inst})$$

Then the complete, permuted proof is:

$$\dfrac{\dfrac{\forall \overline{x} \; \Phi(\overline{x}) \rightarrow A_1(\overline{x}) \vee \cdots \vee A_p(\overline{x})}{\forall (\overline{x}\Sigma) \; \Phi(\overline{x}\Sigma) \rightarrow A_1(\overline{x}\Sigma) \vee \cdots \vee A_p(\overline{x}\Sigma)} \; (\text{inst}) \qquad \pi_1 \qquad \cdots \qquad \pi_p}{\forall (\overline{x}\Sigma) \; \Phi(\overline{x}\Sigma) \wedge \Psi_1(\overline{x}\Sigma) \wedge \cdots \wedge \Psi_p(\overline{x}\Sigma) \rightarrow \bot} \; (\vee\text{-res})$$

C Pushing ∃-Resolution Towards the Root

We show how an ∃-resolution step can be moved closer towards the root. In case the ∃-resolution step has associated ≈-resolution steps, these steps can be permuted together with the ∃-resolution step. It can be shown that the property of being an ≈∃-normal sequence is preserved when an ∃-resolution step and its associated ≈-resolution steps are permuted together.

∨-resolution

$$\dfrac{\forall\overline{x}\;\Psi(\overline{x})\to B_1(\overline{x})\vee\cdots\vee B_q(\overline{x}) \qquad \begin{array}{c}\dfrac{\forall\overline{x}\;\Phi(\overline{x})\to\exists y\;A(\overline{x},y)\qquad \forall\overline{x}y\;X_1(\overline{x})\wedge B_1(\overline{x})\wedge A(\overline{x},y)\to\bot}{\forall\overline{x}\;\Phi(\overline{x})\wedge X_1(\overline{x})\wedge B_1(\overline{x})\to\bot}\,(\exists\text{-res})\\ \vdots\end{array} \qquad \forall\overline{x}\;B_2(\overline{x})\wedge X_2(\overline{x})\to\bot\;\cdots\;\forall\overline{x}\;B_q(\overline{x})\wedge X_q(\overline{x})\to\bot}{\forall\overline{x}\;\Psi(\overline{x})\wedge\Phi(\overline{x})\wedge X_1(\overline{x})\wedge\cdots\wedge X_q(\overline{x})\to\bot}\,(\vee\text{-res})$$

can be permuted into

$$\dfrac{\forall\overline{x}\;\Phi(\overline{x})\to\exists y\;A(\overline{x},y) \qquad \begin{array}{c}\dfrac{\begin{array}{c}\forall\overline{x}\;\Psi(\overline{x})\to B_1(\overline{x})\vee\cdots\vee B_q(\overline{x})\qquad \forall\overline{x}y\;X_1(\overline{x})\wedge B_1(\overline{x})\wedge A(\overline{x},y)\to\bot\\ \forall\overline{x}\;B_2(\overline{x})\wedge X_2(\overline{x})\to\bot\qquad\cdots\qquad\forall\overline{x}\;B_q(\overline{x})\wedge X_q(\overline{x})\end{array}}{\forall\overline{x}y\;\Psi(\overline{x})\wedge X_1(\overline{x})\wedge\cdots\wedge X_q(\overline{x})\wedge A(\overline{x},y)\to\bot}\,(\vee\text{-res})\\ \vdots\end{array}}{\forall\overline{x}\;\Phi(\overline{x})\wedge\Psi(\overline{x})\wedge X_1(\overline{x})\wedge\cdots\wedge X_q(\overline{x})\to\bot}\,(\exists\text{-res})$$

∃-resolution

$$\dfrac{\forall\overline{x}\;\Psi(\overline{x})\to\exists z\;B(\overline{x},z)\qquad \dfrac{\forall\overline{x}\;\Phi(\overline{x})\to\exists y\;A(\overline{x},y)\qquad\forall\overline{x}yz\;X(\overline{x})\wedge A(\overline{x},y)\wedge B(\overline{x},z)\to\bot}{\forall\overline{x}z\;\Phi(\overline{x})\wedge X(\overline{x})\wedge B(\overline{x},z)\to\bot}\,(\exists\text{-res})}{\forall\overline{x}\;\Psi(\overline{x})\wedge\Phi(\overline{x})\wedge X(\overline{x})\to\bot}\,(\exists\text{-res})$$

can be replaced by:

$$\dfrac{\forall\overline{x}\;\Phi(\overline{x})\to\exists y\;A(\overline{x},y)\qquad \dfrac{\forall\overline{x}\;\Psi(\overline{x})\to\exists z\;B(\overline{x},z)\qquad\forall\overline{x}yz\;X(\overline{x})\wedge A(\overline{x},y)\wedge B(\overline{x},z)\to\bot}{\forall\overline{x}y\;\Psi(\overline{x})\wedge X(\overline{x})\wedge A(\overline{x},y)\to\bot}\,(\exists\text{-res})}{\forall\overline{x}\;\Phi(\overline{x})\wedge\Psi(\overline{x})\wedge X(\overline{x})\to\bot}\,(\exists\text{-res})$$

≈-resolution

As with ∨-resolution, we first study the case where the result of the ∃-resolution is the instantiated premise of the equality resolution step. The general form of this situation is as follows:

$$\dfrac{\dfrac{\forall\overline{x}\ \Phi(\overline{x}) \to \exists y\ A(\overline{x},y) \qquad \forall\overline{x}y\ \Psi(\overline{x},z\Sigma)\wedge A(\overline{x},y)\to\bot}{\forall\overline{x}\ \Phi(\overline{x})\wedge\Psi(\overline{x},z\Sigma)\to\bot}\,(\exists\text{-res}) \qquad \forall\overline{x}z\ z\not\approx x\wedge X(\overline{x},z)\to\bot}{\forall\overline{x}z\ \Phi(\overline{x})\wedge\Psi(\overline{x},z)\wedge X(\overline{x},z)\to\bot}\,(\approx\text{-res})$$

Here Σ denotes the substitution $z := x$.

$$\dfrac{\forall\overline{x}\ \Phi(\overline{x}) \to \exists y\ A(\overline{x},y) \qquad \dfrac{\forall\overline{x}y\ \Psi(\overline{x},z\Sigma)\wedge A(\overline{x},y)\to\bot \qquad \forall\overline{x}z\ z\not\approx x\wedge X(\overline{x},z)\to\bot}{\forall\overline{x}yz\ \Psi(\overline{x},z)\wedge A(\overline{x},y)\wedge X(\overline{x},z)\to\bot}\,(\approx\text{-res})}{\forall\overline{x}z\ \Phi(\overline{x})\wedge\Psi(\overline{x},z)\wedge X(\overline{x},z)\to\bot}\,(\exists\text{-res})$$

Next comes the situation where $\exists$-resolution is used to derive the premise containing the disequality atom:

$$\dfrac{\dfrac{\forall\overline{x}\ \Phi(\overline{x}) \to \exists y\ A(\overline{x},y) \qquad \forall\overline{x}yz\ A(\overline{x},y)\wedge\Psi(\overline{x},z)\wedge z\not\approx x\to\bot}{\forall\overline{x}z\ \Phi(\overline{x})\wedge\Psi(\overline{x},z)\wedge z\not\approx x\to\bot}\,(\exists\text{-res}) \qquad \forall\overline{x}\ X(\overline{x},z\Sigma)\to\bot}{\forall\overline{x}z\ \Phi(\overline{x})\wedge\Psi(\overline{x},z)\wedge X(\overline{x},z)\to\bot}\,(\approx\text{-res})$$

$$\dfrac{\forall\overline{x}\ \Phi(\overline{x}) \to \exists y\ A(\overline{x},y) \qquad \dfrac{\forall\overline{x}yz\ A(\overline{x},y)\wedge\Psi(\overline{x},z)\wedge z\not\approx x\to\bot \qquad \forall\overline{x}\ X(\overline{x},z\Sigma)\to\bot}{\forall\overline{x}yz\ A(\overline{x},y)\wedge\Psi(\overline{x},z)\wedge X(\overline{x},z)\to\bot}\,(\approx\text{-res})}{\forall\overline{x}z\ \Phi(\overline{x})\wedge\Psi(\overline{x},z)\wedge X(\overline{x},z)\to\bot}\,(\exists\text{-res})$$

instantiation

$$\dfrac{\dfrac{\forall\overline{x}\ \Phi(\overline{x}) \to \exists y\ A(\overline{x},y) \qquad \forall\overline{x}y\ \Psi(\overline{x})\wedge A(\overline{x},y)\to\bot}{\forall\overline{x}\ \Phi(\overline{x})\wedge\Psi(\overline{x})\to\bot}\,(\exists\text{-res})}{\forall(\overline{x}\Sigma)\ \Phi(\overline{x}\Sigma)\wedge\Psi(\overline{x}\Sigma)\to\bot}\,(\text{merging})$$

becomes

$$\dfrac{\dfrac{\forall\overline{x}\ \Phi(\overline{x}) \to \exists y\ A(\overline{x},y)}{\forall(\overline{x}\Sigma)\ \Phi(\overline{x}\Sigma)\to\exists y\ A(\overline{x}\Sigma,y)}\,(\text{merging}) \qquad \dfrac{\forall\overline{x}y\ \Psi(\overline{x})\wedge A(\overline{x},y)\to\bot}{\forall((\overline{x}\Sigma)y)\ \Psi(\overline{x}\Sigma)\wedge A(\overline{x}\Sigma,y)\to\bot}\,(\text{merging})}{\forall(\overline{x}\Sigma)\ \Phi(\overline{x}\Sigma)\wedge\Psi(\overline{x}\Sigma)\to\bot}\,(\exists\text{-res})$$

Edit and Verify

Radu Grigore[1] and Michał Moskal[2]

[1] UCD CASL, University College Dublin, Belfield, Dublin 4, Ireland
[2] Institute of Computer Science, University of Wrocław, ul. Joliot-Curie 15, 50-383 Wrocław, Poland, `mjm@ii.uni.wroc.pl`

Abstract. Automated theorem provers are used in extended static checking, where they are the performance bottleneck. Extended static checkers are run typically after incremental changes to the code. We propose to exploit this usage pattern to improve performance. We present two approaches of how to do so and a full solution.

1 Introduction

Extended static checking [1] is a technology that makes automated theorem proving relevant to a wide group of programmers. The architecture of an Extended Static Checker (ESC) is similar to that of a compiler (see Fig. 1). It has a front-end that translates high-level code and specifications into a simpler intermediate representation, and a back-end that formulates first order logic formulas as queries for a theorem prover. The queries are called *verification conditions* (VCs). If the ESC is sound then the VC is UNSAT only if the code meets its specifications; if the ESC is complete then the program meets its specification only if the VC is UNSAT. ESC/Java2 [1] is an ESC that was designed to be unsound and incomplete (as a tradeoff to make it more usable in practice); Spec$^{\#}$ [2] is an ESC that was designed to be sound.

In this article we shall assume an ideal ESC that is both sound and complete. Automated first order theorem provers used in extended static checking are incomplete: They either find a proof that a formula is UNSAT or they give an assignment that *probably* satisfies the formula. As a result, even if the ESC is sound and complete, spurious warnings are possible.

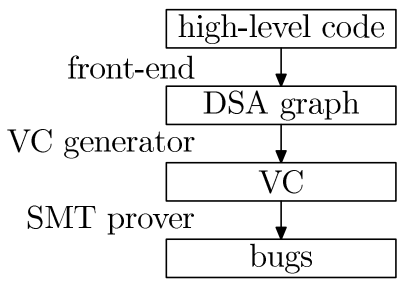


Fig. 1. The architecture of an ESC

The purpose of an ESC is to provide warnings that help programmers to write high-quality code. In practice it is used much like a compiler. Either the programmer runs it periodically or the Integrated Development Environment (IDE) runs it in the background. Because of these usage patterns, performance is quite important. The bottleneck is the prover. Luckily, the fact that the ESC is run often can be exploited since it means that the program does not change much between two runs. Compilers already exploit this by doing incremental compilation [3]. ESCs do checking in a modular way, method by method. Nevertheless, once the contract of a method is altered all its clients must be rechecked. In such a scenario the VCs of the clients do not change much.

```
// blank line                          // (1)
class Day {
  //@ ensures 1 <= \result && \result <= 12;
  public abstract int getMonth();

  //@ ensures 1970 <= \result;
  //@ ensures \result <= 2038;             // (2)
  public abstract int getYear();

  //@ ensures 1 <= \result && \result <= 31;
  public abstract int getDay();

  //@ ensures 1 <= \result;
  //@ ensures \result <= 366;              // (3)
  public int dayOfYear() {
    int offset = 0;
    if (getMonth() > 1) offset += 31;
    if (getMonth() > 2) offset += 28;
    if (getMonth() > 3) offset += 31;
    if (getMonth() > 4) offset += 30;
    if (getMonth() > 5) offset += 31;
    if (getMonth() > 6) offset += 30;
    if (getMonth() > 7) offset += 31;
    if (getMonth() > 8) offset += 31;
    if (getMonth() > 9) offset += 30;
    if (getMonth() > 10) offset += 31;
    if (getMonth() > 11) offset += 30;
    boolean isLeap = getYear() % 4 == 0 &&
                   (getYear() % 100 != 0 || getYear() % 400 == 0);
    //@ assert offset <= 335;              // (4)
    if (isLeap && getMonth() > 2) offset++;
    return offset + getDay();
  }
}
```

Fig. 2. Typical evolution of annotated Java code

This paper (1) argues for the importance of using techniques analogous to incremental compilation in software verification, (2) formalizes the problem and explores possible solutions (Sect. 2), (3) presents a specific solution that works exclusively inside an automated theorem prover (Sect. 3), in the process (4) presents a technique to heuristically determine similarities between formulas, and (5) gives a mechanically verified proof for the correctness of a part of the specific solution presented.

2 Discussion and Definitions

The problem in a nutshell is how to do incremental extended static checking. We shall explore the solution space and then we will see in detail a particular solution, including some experimental data.

Consider the JML-annotated Java code from Fig. 2 When checking the method dayOfYear the ESC will assume the implicit empty precondition holds and will try to prove the postcondition. It will also try to prove all the explicit and implicit assertions in the body. When the method getMonth is called the ESC inserts (implicit) assertions for its preconditions followed by assumptions for its postconditions. Moreover, the ESC will introduce assertions that ensure the absence of runtime exceptions. For example, the receiver object of a method call is asserted to be nonnull.

Notice the lines marked by (1), (2), (3), and (4). Adding these lines represents typical edits that can be done on annotated source code. For example, line (3) is a newly added postcondition. An incremental VC would only check if this new assertion holds, provided that the last VC was UNSAT. It is somehow cumbersome to formulate the problem precisely at the source code level. We can be more precise by descending at the level of an idealized intermediate representation, a *Dynamic Single Assignment (DSA) graph.*

Definition 1 (DSA graph) *The* DSA graph *of a method is a directed acyclic (control flow) graph. Its vertices are* $1, 2, \ldots$ *and they are labeled respectively by the first order logic formulas* $\phi_1, \phi_2, \ldots$. *A vertex represents either an* assertion *(in which case we say it is* black*) or an* assumption *(in which case we say it is* white*). We denote the set of vertices that are predecessors of* v *by* $\mathrm{in}(v)$ *and the set of successors of* v *by* $\mathrm{out}(v)$. *The* in-degree *of* v *is* $|\,\mathrm{in}(v)|$ *and the* out-degree *is* $|\,\mathrm{out}(v)|$. *The nodes with in-degree zero are called* initial nodes*; the nodes with out-degree zero are called* final nodes.

The assertions model the postconditions of the verified method and the checks inside its body (such as the check that an index in an array access is in-bounds, a receiver of a method call is nonnull, the preconditions of a called method hold, explicit JML assertions, and so on). The assumptions model postconditions of the called methods and semantics of the Java language (including properties ensured by the type system).

For this presentation we simply assume that the intermediate representation is obtained from the source code by some technique, without committing to any one in particular. The curious reader can start exploring the subject from other papers [2,4,5,6].

The VC is generated from the intermediate representation. The particular algorithm used has a big impact on performance [7,5]. Here we only present a conceptually simple technique that illustrates well the general form VCs have in practice.

Definition 2 (behaviors) *Vertices have associated* preconditions *denoted by* $\alpha_1, \alpha_2, \ldots$, postconditions *denoted by* $\beta_1, \beta_2, \ldots$, *and* wrong behaviors *denoted by* $\gamma_1, \gamma_2, \ldots$ *For all* i *we have*

$$\alpha_i = \begin{cases} \top & \textit{for initial nodes} \\ \bigvee_{v_j \in \mathrm{in}(v_i)} \beta_j & \textit{for non-initial nodes} \end{cases} \tag{1}$$

$$\beta_i = \alpha_i \wedge \phi_i \tag{2}$$

$$\gamma_i = \begin{cases} \alpha_i \wedge \neg\phi_i & \textit{for assertions} \\ \bot & \textit{for assumptions} \end{cases} \tag{3}$$

Definition 3 (verification condition) *The* verification condition *is*

$$\psi = \bigvee_i \gamma_i \tag{4}$$

The wrong behaviors are something we want to avoid, therefore we ask the prover if all the wrong behaviors are impossible which is the same as asking if the VC is Unsat. If it is, then the ESC concludes that all the assertions are valid and the method is correct. The basic idea behind the more efficient techniques of generating VCs is to generate factored form.

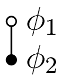
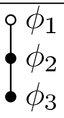
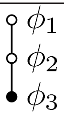
Old	New	Simplified
		
$\psi_1 = \phi_1 \wedge \neg\phi_2$	$\psi_2 = (\phi_1 \wedge \neg\phi_2) \vee (\phi_1 \wedge \phi_2 \wedge \neg\phi_3)$	$\psi_2' = \phi_1 \wedge \phi_2 \wedge \neg\phi_3$

Table 1. Simplification example

The problem can now be stated as follows: Given two similar formulas ψ_1 and ψ_2, find a formula ψ_2' that is Unsat if and only if ψ_2 is Unsat, provided that ψ_1 is Unsat. An example is given in Table 1. The following equations show step by step how to compute ψ_2 from its corresponding DSA graph.

$$\alpha_1 = \top \qquad \beta_1 = \phi_1 \qquad \gamma_1 = \bot \tag{5}$$
$$\alpha_2 = \phi_1 \qquad \beta_2 = \phi_1 \wedge \phi_2 \qquad \gamma_2 = \phi_1 \wedge \neg\phi_2 \tag{6}$$
$$\alpha_3 = \phi_1 \wedge \phi_2 \qquad \beta_3 = \phi_1 \wedge \phi_2 \wedge \phi_3 \qquad \gamma_3 = \phi_1 \wedge \phi_2 \wedge \neg\phi_3 \tag{7}$$

To make the example concrete the reader might wish to plug in $\phi_1 = x > 2$ and $\phi_2 = x > 1$ and $\phi_3 = x > 0$.

Note that $\psi_2' = \phi_1 \wedge \neg\phi_3$ is sound too, but we do not want to drop parts of the formula that are assumptions because they can make the proof easier. The simplified formula can be obtained in two ways. One is to replace the assertions that appear in both DSA graphs by assumptions and generate the VC for the modified DSA graph; the other is to work directly on the formulas ψ_1 and ψ_2. In this paper we will explore in greater detail the latter.

In both approaches, a solution has to solve two subproblems. First, we must find a correspondence between parts of the two DSA graphs (or formulas). Second, we must simplify one of the DSA graphs (or formulas). The methods we present in the next section for finding a correspondence between parts of the formulas can be partially reused for finding a correspondence between parts of the DSA graphs. Simplifying a formula is harder than changing assertions into assumptions, but on the other hand it is independent of the particular intermediate representation used.

3 Pruning First Order Formulas

One subproblem is to find a correspondence between parts of ψ_1 and parts of ψ_2. We substitute (some) uninterpreted constants in ψ_1 by uninterpreted constants that appear in ψ_2. We also normalize the formulas with respect to commutative operators (Fig. 3). We also use hash-consing [8,9] so later terms are simply compared by reference equality.

Note that if ψ_1 is Unsat, then any substitution that renames uninterpreted constants leaves it Unsat. The only assumption we make in solving the second subproblem is that ψ_1 is Unsat, so there is no 'right' or 'wrong' correspondence between old and new constants. It is true, however, that for different substitutions of constants we will end up with different results ψ_2', some bigger and some smaller. Also we need to remember not to rename interpreted constants (such as 1 and 42).

Assuming that all constants that are 'the same' have the same name in ψ_1 as in ψ_2 would not allow us to prune the VC (to $\bot$) when the programmer only renamed a variable. (Variables in the program appear as uninterpreted constants in the VC.) Even worse, the ESC encodes extra information in identifiers [10] that changes, for example, when a new line is added to the source Java file. Despite these variations, a human that sees both ψ_1 and ψ_2 is generally able to say which sub-term corresponds to which sub-term. So there are good chances to find a heuristic that works well!

```
class Term
  public Name : string
  public Children : list [Term]
def SortTerm(t)
  def CompareTerms(a, b)
    def nc = a.Name.CompareTo(b.Name)
    if (nc != 0) nc
    else LexicographicCompare(a.Children, b.Children, CompareTerms)
  def children = t.Children.Map(SortTerm)
  if (IsCommutative(t)) Term(t.Name, t.Children.Sort(CompareTerms))
  else                  Term(t.Name, children)
def oldVC = SortTerm(oldVC)
def newVC = SortTerm(newVC)
```

Fig. 3. Normalizing queries

We only consider renaming of uninterpreted constants because of the particular algorithm used to build VCs. If some of the function symbols would also need to be renamed, the algorithm can be easily extended by the standard technique of introducing a special function symbol *apply*, and replacing $f(t_1, \ldots, t_n)$ with $apply(f, t_1, \ldots, t_n)$.

The heuristic we use to find a good substitution assigns a *similarity* value to each pair of (old, new) constants and then finds a maximum bipartite matching (using the Hungarian method [11]) between the old and the new constants. A complete bipartite graph is constructed from the set V_1 of uninterpreted constants that appear in ψ_1 and the set V_2 of uninterpreted constants that appear in ψ_2. Each pair $(i, j) \in V_1 \times V_2$ has an associated weight, which in this case is the similarity of the two constants. A matching is a subset $M \subset V_1 \times V_2$ such that for all pairs $(i, j) \in M$ and $(i', j') \in M$ we have $i = i'$ if and only if $j = j'$. The weight of the matching is the sum of the weights of all its elements. The similarity has two components: One is the length of the longest common subsequence [12] of the two identifiers; the other, more important, is how many times the constants appear in similar positions in the two VCs.

To measure similarity of position we use path strings [13]. A *path string* is a sequence of function symbols interleaved with the positions, on a path from the root of the term to a particular occurrence of a sub-term. For example $f.2.g.1$ is a path string for the occurrence of b in $f(a, g(b, c))$, and $f.2.g.2$ is a path string for c. We construct a *stripped path string* by treating logical connectives as function symbols, the entire formula as a term, and skipping positions for commutative symbols. For example $\wedge.\vee.f.2.g.1$ is the stripped path string for b in $(f(a, g(b)) \vee g(c)) \wedge g(d)$. The *environment* of a constant c in a formula ψ is the multiset of the stripped path strings for all occurrences of c in ψ. Let E_1 be the environment of x in ψ_1 and E_2 be the environment of y in ψ_2. The similarity of x and y is $2|E_1 \sqcap E_2| - |(|E_1| - |E_2|)|$, where $\sqcap$ is multiset intersection. Other measures, that take environments into account, are also possible.

```
def Prune(p1 : list [ list [Term]], p2 : Term)
  def p1 = Flatten(p1)
  // |p1| is a DNF form, assumed to be UNSAT
  match (p2.Name)
    | "and" =>
        mutable common = []
        foreach (x in p1) foreach (y in x) common = y :: common
        def p1 = p1.Map(x => x.Filter(y => !common.Contains(y)))
        def p2 = p2.Children. Filter (y => !common.Contains(y))
        if (p1.Contains ([])) Term("false", [])
        else                  Term("and", common + p2.Map(x => Prune(p1, x)))
    | "or" =>
        Term("or", p2.Children.Map(x => Prune(p1, x)))
    | _ =>
        if (p1. Exists (x => Implies(p2, Term("and", x)))) Term ("false", [])
        else                                               p2
def prunedVC = Prune([[oldVC]], newVC)
```

Fig. 4. Pruning the VC

The algorithms are presented as Nemerle-like pseudocode [14]. Some obvious optimizations are omitted[3] to improve readability. We also omit textbook algorithms. The algorithm for normalizing queries with respect to commutative operators is given in Fig. 3. It recursively sorts arguments of commutative operators using lexicographic ordering.

The second subproblem, simplification of formulas, is solved by the pruning algorithm in Fig. 4. The function Prune returns a formula equisatisfiable to p2 under the assumption that all elements of p1 are Unsat. Elements of p1 are conjunctions represented as lists.

The function Implies explores the structure of two formulas and returns **true** only if the first is stronger than the second. The last branch is clearly correct: If p2 is stronger than a conjunct known to be Unsat then it is also Unsat. In the case that p2 is a disjunction we can treat its children independently. The case when p2 is a conjunction is more interesting. To understand why it works consider a small example.

$$\psi_1 = (\phi_1 \wedge \phi_2) \vee (\phi_3 \wedge \phi_4) \tag{8}$$
$$\psi_2 = \phi_2 \wedge \phi_4 \wedge (\phi_1 \vee \phi_3) \tag{9}$$
$$\psi_2' = \phi_2 \wedge \phi_4 \wedge \bot = \bot \tag{10}$$

We write $\mathrm{P}(\psi_1, \psi_2) = \psi_2'$ for the result of pruning ψ_2 under the assumption that ψ_1 in Unsat. The common part of ψ_1 and ψ_2, as computed in the variable common in Fig. 4, is $\phi_2 \wedge \phi_4$. Pruning $\phi_1 \vee \phi_3$ knowing that $\phi_1 \vee \phi_3$ is Unsat results in $\bot$. The formulas that appear in both ψ_1 and ψ_2 can always be factored.

[3] See http://nemerle.org/svn.fx7/branches/fx8/Pruner.n for all details.

$$(\phi_1 \wedge \phi_2) \vee (\phi_3 \wedge \phi_4) \tag{11}$$
$$\Leftarrow (\phi_1 \wedge \phi_2 \wedge \phi_4) \vee (\phi_3 \wedge \phi_2 \wedge \phi_4) \tag{12}$$
$$\Leftrightarrow \phi_2 \wedge \phi_4 \wedge (\phi_1 \vee \phi_3) \tag{13}$$

Hence, we can always reduce the problem to the form

$$\psi_1 = \phi'_1 \wedge \phi'_2 \tag{14}$$
$$\psi_2 = \phi'_1 \wedge \phi'_3 \tag{15}$$
$$\psi'_2 = \phi'_1 \wedge \mathrm{P}(\phi'_2, \phi'_3) \tag{16}$$

where ϕ'_1 is the common part and ϕ'_2 is what we assume to be UNSAT while pruning ϕ'_3 (see also Fig. 4). In this example $\phi'_1 = \phi_2 \wedge \phi_4$ and $\phi'_2 = \phi'_3 = \phi_1 \vee \phi_3$. It is easy to see that the above is correct, by doing a case analysis on whether $\phi'_1(\mathbf{x})$ holds for some vector $\mathbf{x}$. The formalization[4] in Coq [15] of a simplified version of the pruning function emphasizes the main points of the proof. The formulas abstract theories by arbitrary predicates over the domain of uninterpreted constants.

```
Inductive Formula : Type :=
  | FPred : (Dom -> Prop) -> Formula
  | FAnd : Formula -> Formula -> Formula
  | FOr: Formula -> Formula -> Formula.
Fixpoint Eval (f : Formula) (x : Dom) {struct f} : Prop :=
  match f with
    | FPred p => p x
    | FAnd fa fb => Eval fa x /\ Eval fb x
    | FOr fa fb => Eval fa x \/ Eval fb x
  end.
```

The simplified version of the algorithm whose proof we check mechanically is

```
Fixpoint Prune (p1 p2 : Formula) {struct p2} : Formula :=
  match p1, p2 with
    | FAnd a b, FAnd aa c => if eq a aa then FAnd a (Prune b c) else p2
    | _, FOr a b => FOr (Prune p1 a) (Prune p1 b)
    | _, _ => if eq p1 p2 then FPred PFalse else p2
  end.
```

This function has two important invariants.

```
Lemma PruneInvA : forall p1 p2 : Formula, forall x : Dom,
  (~ Eval p1 x -> Eval p2 x -> Eval (Prune p1 p2) x).
Lemma PruneInvB : forall p1 p2 : Formula, forall x : Dom,
  (~ Eval p1 x -> Eval (Prune p1 p2) x -> Eval p2 x).
```

[4] Available at http://radu.ucd.ie/hp/papers/ev.html

These are proved by double induction on the structure of p1 and p2. We use one extra fact.

```
Lemma UnsatImp : forall a b : Formula,
  ( forall x : Dom, Eval a x −> Eval b x) −> Unsat b −> Unsat a.
```

At this point we can prove that the algorithm is sound and complete.

```
Lemma PruneSound : forall p1 p2 : Formula,
  Unsat p1 −> Unsat (Prune p1 p2) −> Unsat p2.
Lemma PruneComplete : forall p1 p2 : Formula,
  Unsat p1 −> Unsat p2 −> Unsat (Prune p1 p2).
Theorem PruneCorrect : forall p1 p2 : Formula,
  Unsat p1 −> (Unsat p2 <−> Unsat (Prune p1 p2)).
```

The algorithm in Fig. 4 is more efficient since it exploits the associativity and commutativity of the $\wedge$ and $\vee$ operators. The worst case time complexity is $O(mn)$, and arises when the formula known to be UNSAT and the formula to be simplified have, respectively, the form

$$\psi_1 = \bigvee(\phi_1, \ldots, \phi_{m-1}) \tag{17}$$

$$\psi_2 = \underbrace{\wedge \ldots \wedge}_{n \text{ times}} \phi_m \tag{18}$$

where $\wedge$ and $\vee$ are written as $n$ary operators. Unfortunately, the average case that appears in practice is hard to describe. Experimental data from 20 cases suggests that the running time grows linearly with the size of the formulas. But we need more data before we can make a definite statement (see Sect. 5 for details).

4 Case Study

In this section, we explain how the common way of editing programs affects the DSA and therefore also the VC and how pruning exploits the changes.

Let us again consider the program from Fig. 2. We used ESC/Java2 to generate VCs for a version without any of the lines marked (1), (2), (3), and (4). This was the base case. Next we ran it on a method with only line (1) added, only line (2) added and so forth. Finally we ran the pruning algorithm with the old formula being the base case and the new formula being being VC for a method with an added line. Table 2 lists three times for each such formula. The first is the time it takes to prove the formula using Simplify [16]; the second is the time it takes to prune the formula; the third is the time it takes to prove the pruned formula. The reader can note that the running times of Simplify on the original formulas vary rather nondeterministically. In particular, one would expect the base case and the one with an added empty line to have the same running time, but they do not. The reason for this is a "butterfly effect" in the prover, where for example a slight change in the selection of a literal for a case split can cause large changes in the final shape of the proof search tree.

Marker	Description	Original	Pruning	Pruned
	base case	$20.91s$		
(1)	empty line	$17.59s$	$2.23s$	$0.01s$
(2)	irrelevant postcondition	$16.91s$	$2.31s$	$0.06s$
(3)	additional postcondition	$21.65s$	$2.19s$	$19.34s$
(4)	assertion in the middle	$22.81s$	$2.16s$	$7.67s$

Table 2. Case study results

The first edit operation (marked by (1)) is adding an empty line somewhere, or in general changing the locations of symbols. As ESCs often use location information for encoding symbol names, the uninterpreted constants in the second VC are different than in the first one. Our algorithm generates a query that is just $\perp$.

The second edit strengthens the postcondition of a method getYear used in the verified dayOfYear method. Here, we are able to prune almost everything, i.e. the resulting query is propositionally UNSAT.

The third edit adds a postcondition to the verified method. We can imagine that the DSA graph gets one more black node at the end, so this is the only thing that should be verified now. In this case we do prune parts of the formula, it however fails to speed up checking.

Finally the last edit adds an assertion near the end of the method. Here the heuristics work well and the time is reduced considerably.

The dayOfYear method (Fig. 2) is an example of a case where the VC is relatively small (around 60 kilobytes), but hard to prove. This is due to the large number of possible paths in the method. There are other reasons methods can be hard to prove: methods can be more complicated, the specifications can be complicated, the modelling of the language can be more accurate (for example in multi-threading programs). All those scenarios are good for our pruning algorithm as it runs in polynomial time and can potentially save a lot of proving time. The bad case is when the formula is large, but not that hard to prove. In particular it sometimes happen that most of the time is spent just reading/writing the formula and doing basic preprocessing, like skolemization.

5 Related and Future Work

The work presented here parallels the work done in the compiler community under the name *incremental compilation*. In the context of software verification by theorem proving the term *incremental verification* is taken—it refers to the process of proving stronger assertions using weaker ones as lemmas [17]. Hence, we use the distinct term *edit and verify* for the related idea of proving only what has not been proven before, and doing so automatically. In the context of interactive theorem proving the term *proof reuse* is used for a similar technique [18].

A Program Verification Environment (PVE) is the same for an ESC, as an Integrated Development Environment (IDE) is for a compiler. It provides an easy to use interface to the tool. As incremental compilation is very useful in IDEs, we expect Edit and Verify to be even more useful in PVEs. This is because static verification consumes much more resources than compilation. There is much research on software verification using PVEs, there is also vast amount of interest from the industry in PVEs.

One of the goals of the Mobius research project [19] is to produce a PVE for Java. Penelope [20] is an early PVE that processes a subset of Ada. Its designers chose to rely on interactive theorem proving. The KeY Tool [21] is a modern PVE for Java that uses the same approach but differs in the mechanisms and theory of verification condition generation. Spec$^{\#}$ [2] is a modern PVE for C$^{\#}$ that uses automated theorem proving. ESC/Java2 [1,22] is an ESC for JML-annotated [23] Java code. It produces VCs in the Simplify [16] format and in the SMT format [24] for other automated theorem provers. It also generates VCs for the Coq interactive theorem prover [15].

Whether an ESC is considered a PVE or not depends chiefly on how well integrated it is with the editor. ESC/Java2 is integrated into Eclipse using a plugin. Spec$^{\#}$ is more tightly integrated into Visual Studio using a plugin. Work on incremental compilation [3] suggests that an even tighter integration leads to important performance benefits.

There are two improvements that we will try in the near future. One is to prune the DSA graph. The other is to modify Fx7 [25] to produce a formula weaker than the query but still Unsat, and use that to prune subsequent queries. Another idea that is worth exploring is to integrate pruning more tightly not with the ESC but instead with the proving process. For example, we could save the relevance of specific axioms in the old proof, so they can be prioritized while searching for a proof of the new query.

To assess the effectiveness of these improvements we need a better benchmark. The amount of JML-annotated Java is still modest. Moreover, code from the version control history is not appropriate because the commit cycle is typically much longer that the duration between two invocations of ESC/Java2. Therefore we need to collect such data ourselves and this is a time consuming effort. Such a benchmark would hopefully nicely complement the existing (very useful) Boogie benchmarks and SMT-COMP benchmarks [24]. A theoretical analysis seems to require a good model for the type of queries that are produced as verification conditions.

An idea very similar to the one explored in this paper did lead to interesting results in model checking [26], the so called *extreme model checking*. Model checking is sometimes used together with unit testing and therefore it is run often on code with minor modifications. Therefore, it is natural to take advantage of the results of previous runs.

6 Conclusion

We described the typical usage pattern of automated theorem proving in extended static checking and two approaches that exploit it to improve performance. We gave a detailed solution that processes first order formulas. The implementation is a part of the Fx7 theorem prover [25]. It was tested on queries generated by ESC/Java2, without requiring any modifications to the latter. The other approach, working on the intermediate representation of the extended static checker, promises to be more efficient but requires a tighter integration of the prover with the checker.

The first part of the solution is a heuristic that, given two formulas, finds which sub-terms of one formula correspond to which sub-terms of the other. This heuristic may prove to be a useful technique in solving related problems since it performs well and there is ample room for tuning. The second part of the solution is a formula pruning algorithm. This algorithm is proven correct, and part of the proof is mechanically verified. Its efficiency is reasonable because of the use of hash-consing and because formulas are normalized with respect to commutative operators. The pruned formulas are clearly easier to prove.

Acknowledgements. This work is partly funded by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. The article contains only the authors' views and the Community is not liable for any use that may be made of the information therein. The second author is partially supported by Polish Ministry of Science and Education grant 3 T11C 042 30.

The authors would like to thank Joseph Kiniry, Mikoláš Janota, and Fintan Fairmichael for their detailed feedback on a draft of this article. The authors would also like to thank the anonymous reviewers who pointed out that a formal analysis of the performance gains is needed. We will try to include such an analysis once the work progresses.

References

1. Flanagan, C., Leino, K., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002). (2002) 234–245
2. Barnett, M., Leino, K., Schulte, W.: The Spec$^{\#}$ programming system: An overview. In: Proceeding of CASSIS. Volume 3362 of Lecture Notes in Computer Science., Springer–Verlag (2004)
3. Schwartz, M.D., Delisle, N.M., Begwani, V.S.: Incremental compilation in Magpie. Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction (1984) 122–131
4. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology **3**(6) (2004) 27–56

5. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In Ernst, M.D., Jensen, T.P., eds.: Workshop on Program Analysis For Software Tools and Engineering, ACM Press (September 2005) 82–87
6. Darvas, A., Müller, P.: Reasoning about method calls in JML specifications. Formal Techniques for Java-like Programs (2005)
7. Flanagan, C., Saxe., J.B.: Avoiding exponential explosion: generating compact verification conditions. Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (2001) 193–205
8. Ershov, A.P.: On programming of arithmetic operations. Communications of the ACM **1**(8) (1958) 3–6
9. Filliâtre, J.C., Conchon, S.: Type-safe modular hash-consing. In: Proceedings of the 2006 workshop on ML, New York, NY, USA, ACM Press (2006) 12–19
10. Leino, K.R.M., Millstein, T., Saxe, J.B.: Generating error traces from verification-condition counterexamples. Science of Computer Programming **55**(1–3) (2005) 209–226
11. Knuth, D.E.: The Stanford GraphBase: A platform for combinatorial computing. ACM Press (1993) See the program `assign_lisa`.
12. Hirschberg, D.S.: A linear space algorithm for computing maximal common subsequences. Communications of the ACM **18**(6) (1975) 341–343
13. Ramakrishnan, I.V., Sekar, R.C., Voronkov, A.: Term indexing. In Robinson, J.A., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier and MIT Press (2001) 1853–1964
14. The Nemerle programming language website. `http://nemerle.org/`.
15. Casteran, P., Bertot, Y.: Interactive Theorem Proving And Program Development: Coq'Art—the Calculus of Inductive Constructions. Springer (2004)
16. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM **52**(3) (2005) 365–473
17. Uribe, T.E.: Combinations of model checking and theorem proving. Proceedings of the Third International Workshop on Frontiers of Combining Systems **1794** (2000) 151–170
18. Beckert, B., Klebanov, V.: Proof reuse for deductive program verification. Software Engineering and Formal Methods (2004) 77–86
19. The Mobius project website. `http://mobius.inria.fr/`.
20. Guaspari, D., Marceau, C., Polak, W.: Formal verification of Ada programs. IEEE Transactions on Software Engineering **16**(9) (1990) 1058–1075
21. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. LNCS 4334. Springer-Verlag (2007)
22. Cok, D., Kiniry, J.: ESC/Java2: Uniting ESC/Java and JML. Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices **3362** (2005) 108–128
23. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. Behavioral Specifications of Businesses and Systems (1999) 175–188
24. SMT-LIB: The satisfiability modulo theories library. `http://www.smt-lib.org/`.
25. Moskal, M.: Fx7 or it is all about quantifiers (SMTCOMP, 2007) Also, `http://nemerle.org/fx7/`.
26. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme model checking. In: Verification: Theory and Practice. Springer (2004)

Author Index