PRACTICAL ASPECTS OF AUTOMATED FIRST-ORDER REASONING

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2012

By
Kryštof Hoder
School of Computer Science

Contents

Word Count: 45,338

List of Tables

List of Figures

Abstract

PRACTICAL ASPECTS OF AUTOMATED
FIRST-ORDER REASONING
Kryštof Hoder
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2012

Our work focuses on bringing the first-order reasoning closer to practical applications, particularly in software and hardware verification. The aim is to develop techniques that make first-order reasoners more scalable for large problems and suitable for the applications.

In pursuit of this goal the work focuses in three main directions. First, we develop an algorithm for an efficient pre-selection of axioms. This algorithm is already being widely used by the community and enables off-the-shelf theorem provers to work with problems having millions of axioms that would otherwise be overwhelming for them. Secondly, we focus on the saturation algorithm itself, and develop a new calculus for separate handling of propositional predicates. We also do an extensive research on various ways of clause splitting within the saturation algorithm.

The third main block of our work is focused on the use of saturation based first-order theorem provers for software verification, particularly for generating invariants and computing interpolants. We base our work on theoretical results of Kovacs and Voronkov published in 2009 on the CADE and FASE conferences. We develop a practical implementation which embraces all the extensions of the basic resolution and superposition calculus that are contained in the theorem prover Vampire. We have also developed a unique proof transforming algorithm which optimizes the computed interpolants with respect to a user specified cost function.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

This thesis would not have been possible without my supervisor Andrei Voronkov, whom I owe my deepest gratitude for his guidance, support and for introducing me into the state-of-the-art in the field of automated reasoning by bringing me to the work on Vampire.

I would also like to give large thanks to Laura Kovacs with whom we worked on many of the papers included in this thesis and who was always enthusiastic about working with Vampire.

My thanks belong also to Nikolaj Bjorner with whom I spent altogether six months in Microsoft Research where I learned a lot (not only) about software verification and automated reasoning in practice.

Thanks are due also to the supervisor of my Master thesis, Josef Urban, who brought me to the area of automated reasoning and with whom I worked on the Sine algorithm which is also part of this thesis.

And last but not least, I would like to thank my parents, grandparents, my wife and my two siblings for their support and encouragement in all my work.

Chapter 1

Introduction

Our work focuses on bringing first-order reasoning closer to practical applications, particularly in software and hardware verification. The aim is to develop techniques that make first-order reasoners more scalable for large problems, and make them suitable for the applications.

The history of automated reasoning in first-order logic dates to the early 1950s. For a long time, almost until the year 2000, the problems given to first-order reasoners came mainly from the attempts to formalize mathematics. They would usually consist of a few quantified axioms (such as an axiomatization of groups) and of a conjecture to be proved. In the recent years, however, together with the developments of large ontologies such as [NP01, SKW07] and, more importantly, with the use of automated theorem proving in software and hardware verification, the problems passed to the first-order provers have become different. They grew in size significantly, even though large parts of the problems would not involve quantifiers and/or would be irrelevant to the conjecture that is to be proved.

These differences lead to several challenges which we are trying to address:

1. **Size** The usual size of first-order problems coming from the formalization of mathematics was in the order of tens or hundreds of axioms. The Problems coming from ontologies can have millions of axioms. In the area of verification it is not uncommon to have problems with thousands of axioms.

2. **Difficulty** There are problems which are difficult to be solved by the resolution calculus, for example, those that lead to the introduction of long clauses. These can be very expensive to handle; for example, to check whether one clause subsumes another can take time exponential in the length of the clauses.

3. **Interpolation** Obtaining a proof of unsatisfiability is often not sufficient for verification tools. Many verification techniques, for example the predicate abstraction [FQ02], require the output of an interpolant. Furthermore, for a single problem it may be possible to generate many different interpolants. This opens the question of generating the interpolants that are best suited for the use in verification.

These three challenges determine the three main directions in which our work focuses.

To tackle the first challenge of the problem size, we develop a new efficient algorithm for the pre-selection of axioms that are likely to be relevant to proving a given goal. This algorithm is already being widely used by the community and enables off-the-shelf theorem provers to work with problems having millions of axioms that would otherwise be too difficult for them.

In the second challenge, we focus on the saturation algorithm itself. As satisfiability checking for first-order logic is undecidable, there will always be problems that are too difficult for any solving algorithm. There are, however, problems that are difficult for the resolution calculus, but can be dealt with by other techniques.

Problems coming from software and hardware verification have often many ground, or even propositional atoms. Checking for propositional satisfiability, as well as for satisfiability of ground formulas with equality is a decidable problem and there are efficient decision procedures for it. We develop a calculus that separates reasoning on propositional formulas, so that it can be handed over to a dedicated tool such as a SAT solver. We also do an extensive investigation of splitting techniques which are beneficial particularly for reasoning with ground atoms.

For the third challenge of providing an interface useful for verification tools, we base our work on existing theoretical results given in [KV09a] and [KV09b]. We develop a practical implementation which embraces extensions of the resolution and superposition calculus that are contained in the theorem prover Vampire. We also develop a unique proof transforming algorithm which optimizes the computed interpolants with respect to a user specified cost function.

In the remaining sections of this introduction we give a brief overview of first-order theorem proving and techniques used for software and hardware verification. In the end of this chapter we present a more detailed discussion of the contributions of this work.

1.1 Automated First-Order Theorem Proving

The main aim of automated first-order theorem proving is to determine satisfiability of a set of formulas, and possibly provide a certificate of the result, either in the form of a derivation of a false formula, or some representation of a model satisfying the input formulas.

Many first-order theorem proving algorithms do not provide a general way of obtaining models when the set of formulas is shown to be satisfiable. An alternative to this (serving still as a verifiable certificate) can be a saturated set of formulas, where no new formula can be derived using some complete saturation algorithm.

A well known and generally best performing family of first-order proving algorithms are those based on various extensions of the calculus of ordered resolution and superposition described in [NR01].

The idea of resolution theorem proving comes from the seminal paper of Robinson [Rob65]. The basic resolution rule is

$$\frac{A \vee C \quad \neg B \vee D}{(C \vee D)\sigma}$$

where σ is a most general unifier of A and B. The resolution rule is accompanied by a factoring rule which can be expressed as

$$\frac{A \vee B \vee C}{(A \vee C)\sigma}$$

where σ is a most general unifier of A and B.

In a pure resolution calculus, the equality can be handled by axiomatizing it as a congruence relation. Reasoning with such axiomatization is however inefficient, and therefore the practical calculi use specialized rules for equality reasoning.

The first general method for equality reasoning based on the paramodulation was according to [DV01] proposed by Robinson and Wos in [RW69]. Their paramodulation rule can be written as

$$\frac{C[s] \quad l = r \vee D}{(C[r] \vee D)\sigma}$$

where σ is the most general unifier of s and l.

The proof search in a saturation theorem prover is conducted by repeated applications of inference rules such as the above. Each application of the above rules increases the size of the set of clauses, and therefore increases the search space in which the

prover tries to find the contradiction. A significant effort has been made to restrict the applicability of the rules while preserving completeness of the calculus. Such restrictions prune the search space without the danger of losing the chance to find a solution. The restrictions used in the saturation provers today are based on simplification orderings and were introduced in [Pet83] and [PP91]. To illustrate the restrictions, we show a possible formulation of a restricted paramodulation rule:

$$\frac{L[s] \vee C \quad l = r \vee D}{(L[r] \vee C \vee D)\sigma}$$

where

- σ is the most general unifier of l and s
- s is not a variable
- $r\sigma$ is not greater than $l\sigma$ in the used simplification ordering
- $(l = r)\sigma$ is maximal (wrt. the simplification ordering) in the clause $(l = r \vee D)\sigma$
- $L[s]\sigma$ is maximal (wrt. the simplification ordering) in the clause $(L[s] \vee C)\sigma$.

To form a complete calculus, the presented paramodulation rule would have to be accompanied by other rules such as the resolution, factoring or the equality resolution. Full set of the rules for such calculus is given in the Chapter 2 or in [NR01].

Apart from restrictions on the rules that generate new clauses, another important principle in saturation theorem proving is redundancy elimination. It allows us to remove clauses that are already derived, but not necessary for the completeness of the saturation algorithm. The general notion of redundancy is that a clause C is redundant in a set of clauses S if each of its ground instances C' is implied by the ground instances of the clauses in S that are smaller than C' with respect to a simplification ordering.

The redundancy as stated above is difficult to be tested in practice. However, there are several simpler tests which can identify some classes of redundant clauses. The most well known is probably the subsumption rule, which removes the clauses that are super-multisets of instances of other clauses. This rule was introduced by Robinson in [Rob65], even before the general notion of redundancy was introduced. The subsumption rule can be written as

$$\frac{C \quad \cancel{D_1 \vee D_2}}{}$$

if there exists a substitution σ such that $C\sigma$ is identical to D_1. For details on the concept of redundancy and simplification rules we refer to [NR01].

1.1.1 First-Order Theorem Provers

Along with the calculus used for reasoning, there are other aspects important when implementing a theorem prover. Davis and Putnam [DP60] proposed that the input formula passed to a first-order theorem prover should be preprocessed using skolemization to eliminate existential quantifiers and then converted to clausal normal form (CNF). This general preprocessing scheme has proven to be successful and is used by modern theorem provers, even though other preprocessing steps may be put into the clausification process to make the obtained set of clauses simpler.

The formulations of reasoning calculi are given in a declarative way, with statements such as that having clauses C_1 and C_2 of a certain form, we can infer a clause D, or perhaps show that one of the premise clauses is redundant. To have a complete reasoning system, one often needs to ensure that the inferences are applied in a fair way (that every possible non-redundant inference will eventually be performed). Saturation based theorem provers generally use the so called *given clause algorithm* which we describe in the Chapter 2.

Efficient retrieval of all possible inference candidates from a set of clauses is a problem that can be addressed by *term indexing*. Use of term indexing in theorem proving was mentioned in [OL80], and in today's first-order provers, some form of advanced indexing is almost a necessity to gain a reasonable performance. In this thesis, Chapter 10 focuses on a particular indexing algorithm called substitution trees and the use of unification algorithms within it.

Another important aspect of the area of automated first-order theorem proving is the evaluation of theorem proving systems. This is done on an empirical basis, by comparing the number of solutions a prover can find for problems from a certain set of benchmarks within a given time limit. A widely used collection of benchmarks is called TPTP (Thousands of Problems for Theorem Provers) [Sut09]. There is also an annual competition of theorem provers CASC [SS06] which uses problems from this library to compare the state-of-the-art theorem provers.

A more detailed overview of the history of reasoning tools is given in [Dav01] and [Bun99].

1.2 Software and Hardware Verification

Software and hardware verification are areas which offer many practical applications of automated reasoning. According to [Jan11], an estimated annual cost of software and hardware bugs to the US economy alone is 20 to 60 billion dollars. This makes the area of verification interesting not only to academia, but also to industrial companies. Companies such as Microsoft or Intel regularly use tools based on verification techniques as part of their product development.

Majority of verification tools use propositional reasoning (SAT) or ground reasoning modulo theories (SMT). There has been a significant amount of research in algorithms for SAT and SMT solving, and in many practical cases these offer good scalability and robustness.

By scalability we mean a rather vague notion that if a system is able to solve a small instance of a problem, it should not struggle too much to solve larger instances which are not qualitatively different. In the area of verification, one can see it as the transition from verifying toy examples to verification of practical code with thousands lines of code or more.

By robustness we mean that a small change in an irrelevant part of a problem should not have a large influence on the ability of the solver to find a solution. This is important for practical verification, as further modifications of the verified code are likely to happen. It is undesirable that after a minor change, large part of the verification work has to be re-done because the solver is no longer able to find the solution it found earlier.

In the area of SAT solving, there has been an intensive development since the 1990s. These developments enabled SAT solvers to solve significantly harder problems, and brought the scalability that allows one to verify problems of an industrial size. An example of this can be the use of SAT solvers for equivalence checking in [GPB01], or a propositional model checking tool IC3 [Bra11] which uses repeated calls to a SAT solver.

Together with the advances in SAT solving, the area of satisfiability modulo theories (SMT) started developing in the end of 1990s. The DPLL(T) technique [NOT04] suggests to look at a ground SMT problem as at a SAT problem with additional constraints imposed by its underlying theory. There are several theories, motivated mostly by the area of software verification. The most common are the theory of equality and uninterpreted functions [NO05], linear arithmetic [Dd06], arrays and bit-vectors [GD07]. Many software verification and testing tools (e.g. [BCD+05] or [TS06]) use

modern SMT solvers such as Z3 [dMB08] for reasoning about possible program executions.

Apart from checking the satisfiability of a formula, the users of propositional and SMT solvers often need a richer interface, for example output of a model, of an unsatisfiable subformula, of a proof or of an interpolant.

Our work focuses on the area of first-order reasoning. Compared to SAT and SMT solving, our disadvantage is that first-order logic is undecidable. On the other hand, the practice shows that there are many complex problems that can be solved by automated first-order reasoning, and the higher expressive power of the first-order logic allows for more straightforward problem encoding that better reflects the structure of the original problem. Our aim in the work is to bring the first-order theorem provers closer to the applications which currently use propositional or SMT reasoning, and to make them more appealing to those applications that will benefit from the higher expressive power.

1.3 Contributions

The contributions of our work are in four main areas:

- ▷ the Sine algorithm for selection of relevant axioms, addressing the first challenge of the growing problem size;
- ▷ techniques for splitting and reasoning with propositional predicates — these focus on the second challenge of problems difficult for resolution provers;
- ▷ the use of local proofs to generate invariants and optimized interpolants, which aims at the third challenge of interfacing with the verification tools;
- ▷ the implementation of the Vampire theorem prover itself to bring the presented techniques to an actual practical use.

The work in these areas lead to seven papers published in proceedings of international conferences including POPL, IJCAR or TACAS.

1.3.1 Selection of Relevant Axioms

To address the challenge of the growing problem size, we develop an axiom selection algorithm Sine. The algorithm has attracted a significant interest in the first-order theorem proving community, and is being used by many provers such as Vampire, E

[Sch02] and iProver [Kor08]. The algorithm is described in [HV11] which is included in this thesis as Chapter 3.

We also evaluate the benefits of the Sine axiom selection for proving problems coming from libraries formalizing mathematics. This work is presented in [UHV10] and is included as Chapter 4.

1.3.2 Splitting and Propositional Variables

Tackling the challenge of problems difficult for resolution theorem provers, we investigate techniques for splitting first-order clauses within a saturation algorithm, and develop a new calculus for separate handling of propositional predicates. The splitting work builds on [RV01a] and [Wei01]. It describes and evaluates decisions which need to be taken when combining splitting with saturation theorem proving (such as which clauses to split and when to do it). The newly developed calculus RePro allows to separate reasoning about propositional symbols from first-order reasoning, and to delegate it to a more suitable tool such as a SAT solver. This work has not yet been submitted for publication and is described in Chapter 9.

1.3.3 Local Proofs, Interpolation and Invariant Generation

For the problem of interaction between first-order provers and verification tools, we build a practical implementation of the interpolation and invariant generation algorithms based on local proofs described in [KV09b] and [KV09a]. The implementation is built inside Vampire and it is compatible with many of its advanced techniques such as splitting, which are not dealt with in the original papers. The implementation is described in [HKV10] which is included as Chapter 7.

We also extend [KV09b] with a technique for non-local proof transformations which optimizes the size of generated interpolants with regard to a specified cost function. We prove NP-hardness of this optimization problem and use a state-of-the-art SMT solver Yices [Dd06] to perform this optimization. The work is described in [HKV12] which is included in this thesis as Chapter 5.

Another part of [HKV12] and of Chapter 5 is our work on proof localization. We develop an algorithm which can transform an arbitrary proof into a local proof in the sense described in [KV09b], under the assumption that the only colored symbols are constants. This is a reasonable assumption which is common to hold in applications of interpolation.

And finally, we implement a program analysis tool in the Vampire prover that uses our results on invariant generation described in the Chapter 7. A description of this work was published in [HKV11b] and [HKV11a] which are included as Chapters 6 and 8.

1.3.4 Vampire Theorem Prover

Last but not least, we developed almost an entirely new version of the theorem prover Vampire. Within more than 160,000 lines of C++ code it contains implementation of several first-order reasoning algorithms. Along with the variations of saturation proving there is also an engine using the InstGen calculus [Kor09a] and a finite model finding loop based on [BFdNT09]. Together with the first-order reasoning procedures, it contains also propositional tools such as BDDs, an incremental SAT solver and various procedures using and-inverter graphs.

During the implementation, we performed an extensive benchmarking of the use of unification algorithms within the term and literal indexes inside a first-order prover. We created a set of benchmarks for unification algorithms and used it to compare several well known unification algorithms as well as our modification of the Robinson algorithm [Rob65] which avoids the exponential complexity of the original algorithm. The work is described in [HV09] which is included as Chapter 10.

Along with the publications, the contribution of the work carried out is also witnessed by the successes of the Vampire prover in the theorem prover competition CASC [SS06].

To date, the new version of the Vampire prover has participated in three of the annual CASC competitions, CASC–22 [Sut10], CASC–J5 [Sut11] and CASC–23 [Sut12]. At the CASC–22, the new version of Vampire entered the competition as a combination with an older version of the Vampire prover [RV02]. The combination has won the FOF division (general first-order formulas), while the old Vampire itself ranked as the third. It also won the LTB division which focuses on the problems with large axiomatizations.

For CASC–J5, the new version of Vampire entered on its own, and won both of the main divisions, FOF and CNF (problems presented in clausal normal form), together with the LTB division.

Unfortunately, the performance in the latest competition, CASC–23 was hindered by a bug introduced during the last-minute modifications. Despite this, the new version of Vampire won the LTB division, and the FOF division was won by the earlier version

of Vampire which entered the previous year's CASC-J5.

Chapter 2

Preliminaries

The chapters 3 and onward of this work are separate publications, each of them having their introduction where the necessary preliminaries are given. However, in this chapter we include some elementary definitions that are important for the resolution first-order theorem proving in general. We base our definitions on those given in [NR01] and [BG01].

2.1 Terms, Atoms, Formulas and Clauses

Definition 2.1.1 Signature is a triple $\Sigma = \langle \Sigma_f, \Sigma_p, arity \rangle$ where Σ_f is the set of function symbols, Σ_p set of predicate symbols and $arity : \Sigma_f \cup \Sigma_p \to \mathbb{N}$ a function assigning symbols their arity. We assume the sets Σ_f and Σ_p to be disjoint. Function symbols with zero arity we call constants and predicate symbols with zero arity are propositional constants. When there is no chance of ambiguity, we may refer to the set of all symbols $\Sigma_f \cup \Sigma_p$ as to Σ.

Having defined the signature, we define terms that use symbols from the signature together with the variables from a given set.

Definition 2.1.2 Let $\mathcal{X}$ be a set of variable symbols such that $\mathcal{X} \cap \Sigma$ is empty. The set of first-order terms over Σ and $\mathcal{X}$, denoted by $\mathcal{T}(\Sigma, \mathcal{X})$, is the smallest set containing $\mathcal{X}$ such that $f(t_1, \ldots, t_n)$ is in $\mathcal{T}(\Sigma, \mathcal{X})$ whenever $f \in \Sigma_f$, $arity(f) = n$ and $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{X})$. Syntactic equality of two terms will be written as $s \equiv t$. We denote $\mathcal{T}(\Sigma, \emptyset) = \mathcal{T}(\Sigma)$ to be a set of ground terms over Σ.

Note that if there is no constant symbol in Σ_f, $\mathcal{T}(\Sigma)$ is an empty set. In the further we will assume that there is at least one constant symbol in Σ_f.

Definition 2.1.3 Equality is an unordered pair of terms (s,t). It can be denoted as $s = t$ or equivalently as $t = s$. The set of atomic formulas over a signature Σ and variables $\mathcal{X}$ denoted as $\mathcal{A}(\Sigma,\mathcal{X})$ is the smallest set that contains equalities $s = t$ for all $s,t \in \mathcal{T}(\Sigma,\mathcal{X})$, and $p(t_1,\ldots,t_n)$ whenever $p \in \Sigma_p$, $arity(p) = n$ and $t_1,\ldots,t_n \in \mathcal{T}(\Sigma,\mathcal{X})$.

Definition 2.1.4 The set $\mathcal{A}(\Sigma,\emptyset)$ of the ground atoms over Σ can be referred to also as the *Herbrand universe*. A subset of Herbrand universe I is a *Herbrand interpretation* if the equality atoms form a congruence relation on the terms, and if for each $p \in \Sigma_p$ with $arity(p) = n$ it holds that for all $s_1,\ldots,s_n,t_1\ldots,t_n$ such that $\{s_1 = t_1,\ldots,s_n = t_n\} \subset I$ we have $p(s_1,\ldots,s_n) \in I$ if and only if $p(t_1,\ldots,t_n) \in I$. A Herbrand interpretation I assigns $\top$ to the all atoms A such that $A \in I$ and $\bot$ to all other atoms, i.e. to all A such that $A \in \mathcal{A}(\Sigma,\emptyset) \setminus I$.

Definition 2.1.5 A substitution σ is a total map from the variables $\mathcal{X}$ to the terms $\mathcal{T}(\Sigma,\mathcal{X})$. We denote applications of substitution with a post-fix notation, e.g. $x\sigma$ is σ applied on the variable x. The range of a substitution can be extended to terms the usual way: $t\sigma$ denotes the result of simultaneously replacing in t every $x \in Dom(\sigma)$ by $x\sigma$. A substitution can be written as a set of pairs $x \mapsto t$ where x is a variable and t is a term. For the variables that are not explicitely specified by this notation, the substitution is an identity. We say that a term t matches a term s if $s\sigma \equiv t$ for some σ. Then t is called *instance* of s. If t is ground, it is *ground instance* of s and the substitution σ is a *grounding substitution*. If a substitution is injective and its range is $\mathcal{X}$, we call it a *variable renaming*.

Definition 2.1.6 A term t is *unifiable* with a term s if $s\sigma \equiv t\sigma$ for some substitution σ. A substitution σ is called a *most general unifier* of s and t and denoted $mgu(s,t)$ if $s\sigma \equiv t\sigma$ and for every unifier θ of s and t it holds that $s\theta \equiv s\sigma\sigma' \equiv t\theta \equiv t\sigma\sigma'$ for some σ'. It is a well known result that the $mgu(s,t)$ is unique up to a variable renaming.

It is possible to view a term (or an atom) as a set of all its ground instances. The result of unifying terms s and t then represents the intersection of the sets of their ground instances. If the sets of ground instances for some terms do not have an intersection, these terms do not have a unifier.

We discuss the unification and unification algorithms in more detail in the Chapter 10 of this thesis.

Definition 2.1.7 The set of all formulas over a signature Σ and variables $\mathcal{X}$, denoted as $\mathcal{F}(\Sigma,\mathcal{X})$ is the smallest set that contains $\top$, $\bot$, $\mathcal{A}(\Sigma,\mathcal{X})$, and for all $\varphi,\rho \in \mathcal{F}(\Sigma,\mathcal{X})$ and $x \in \mathcal{X}$ it contains the negation $\neg\varphi$, conjunction $\varphi \wedge \rho$, disjunction $\varphi \vee \rho$, implication

$\varphi \to \rho$, equivalence $\varphi \leftrightarrow \rho$, universal quantification $\forall x : \varphi$ and existential quantification $\exists x : \varphi$. The connectives $\wedge$, $\vee$ and $\leftrightarrow$ are associative and commutative, and the connective $\to$ is associative.

There are two subclasses of formulas which are of a particular interest to the first-order theorem proving — the literals and the clauses. The usual architecture of theorem provers first performs the clausification, converting the general formulas to clauses, and only after that runs the main solving algorithm on the set of clauses.

Definition 2.1.8 We call a *literal* either an atomic formula or its negation, and we refer to the set of all literals over a signature Σ and variables $\mathcal{X}$ as to $\mathcal{L}(\Sigma, \mathcal{X})$. The atomic formula of a literal L can be denoted as $atom(L)$. A *clause* we define as a finite multiset of literals. The set of all clauses we denote as $\mathcal{C}(\Sigma, \mathcal{X})$. We may write a clause $\{L_1, \ldots, L_n\}$ as $L_1 \vee \ldots \vee L_n$ and view it as a formula which is a disjunction of the literals of the clause. We will denote the multiset of atoms present in the clause C as $atoms(C)$ and the multiset of literals as $literals(C)$. We define the result of applying a substitution σ to a clause C (denoted as $C\sigma$) to be a clause C' which for each literal $A \in C$ contains $A\sigma$ with the same multiplicity as the literal A in C.

We define the semantics of clauses using the Herbrand interpretations:

Definition 2.1.9 A ground clause $C \in \mathcal{C}(\Sigma, \emptyset)$ is true in a Herbrand interpretation I if either one of its positive literals appears in I, or one of its negative literals is not in I. A non-ground clause C is true in an interpretation I if all its ground instances (obtained by applying all the grounding substitutions to C) are true. Set of clauses S is *satisfiable* if there exists an interpretation I such that all the clauses in S are true in I. If this does not hold, we call the set of clauses *unsatisfiable*.

2.2 Orderings and Literal Selection

In the rest of this chapter we will focus on the notions useful particularly for the resolution and superposition theorem proving.

We will use the square bracket notation to denote subterms. Writing $u[s]$ means a term u with a subterm s (possibly also $u \equiv s$). If there is no danger of ambiguity, we may later write $u[t]$ which will mean the term $u[s]$ with one occurrence of the term s replaced by t. Similarly, one may write $A[s]$ for atoms, $L[s]$ for literals, $C[s]$ for clauses or $\varphi[s]$ for formulas.

Definition 2.2.1 A strict partial term ordering $\succ$ is a transitive and irreflexive binary

relation on the terms. An ordering is *stable under substitutions* if $s \succ t$ implies $s\sigma \succ t\sigma$ for any two terms s, t and a substitution σ. Ordering *fulfils the subterm property* if $u[s] \succ s$ for all terms u and their subterms s such that $u \not\equiv s$. The ordering is *total* on the set of ground terms $\mathcal{T}(\Sigma)$ if for all ground terms s, t it holds that either $s \equiv t$, $s \succ t$ or $t \succ s$. An ordering $\succ$ on terms is *monotonic* if for every function symbol f with the arity n it holds that $s_i \succ t$ implies $f(s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n) \succ f(s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_n)$. An ordering is *well-founded* if every set of terms has a minimal element in the ordering.

Definition 2.2.2 A *rewrite ordering* is a monotonic ordering stable under substitution. A *reduction ordering* is a well-founded rewrite ordering. A *simplification ordering* is a rewrite ordering with the subterm property.

There are several well know simplification orderings such as *lexicographic path ordering* [KL80], *recursive path ordering* or *Knuth-Bendix ordering* [KB70]. The last, often referred to as KBO, is the ordering most commonly used in the first-order provers, and our prover Vampire uses it as well.

Term orderings are often defined as total on the ground terms, and then lifted for the terms that contain variables, so that they remain stable under substitutions. This means that for the terms s, t that are not ground it holds that $s \succ t$ if and only if for all grounding substitutions σ it holds that $s\sigma \succ t\sigma$. Term orderings have usually an associated atom ordering. This can be extended to an ordering on literals and clauses in the following way:

For literals L_1 and L_2 it holds that $L_1 \succ L_2$ either if $atom(L_1) \not\equiv atom(L_2)$ and $atom(L_1) \succ atom(L_2)$, or if $L_1 \equiv \neg A$ and $L_2 \equiv A$ for some atom A.

For clauses C and D it holds that $C \succ D$ if and only if $C \neq D$ and for each literal $L_1 \in literals(D) \setminus literals(C)$ there is a literal $L_2 \in literals(C) \setminus literals(D)$ such that $L_2 \succ L_1$.

The literal selection is a concept closely related to simplification orderings. It allows us to restrict the proof search space by excluding some literals from the inferences which generate new clauses.

Definition 2.2.3 A *literal selection algorithm* is an algorithm that takes a clause C together with the state of the theorem prover as an input, and outputs its subset C'. A requirement on the selection algorithm is that once it gives a particular selection for a clause, any time in the future when it is given the same clause, the answer of the algorithm must be the same.

A literal selection algorithm is called *complete with respect to a simplification ordering* $\succ$ if whenever there is no negative literal in the selected subset C', C' contains

all the positive literals of C that are maximal in the ordering $\succ$.

We do not define literal selection algorithm to be a simple function $\mathcal{C} \mapsto \mathcal{C}$, as for the literal selection we may use also the state of the saturation algorithm. For example, we may attempt to select such literals that the number of inferences a clause may participate in is minimal, for which we need to know the other clauses that may participate in the inferences. However, to maintain completeness, it may be necessary that the algorithm always gives the same response when the same clause is given to it. This may be achieved by a simple caching of the results for clauses that the algorithm has already seen.

Also, the behavior of our literal selection is slightly different than the one of [BG01] — our selection function can select positive or negative literals, while in [BG01] only the negative literals are selected. Our definition can, however, capture the behavior of the selection in Vampire, as its selection functions can select positive literals even in the presence of negative ones.

2.3 Saturation

Definition 2.3.1 *Saturation calculus* is a set of generating, simplifying and deleting rules (altogether called *inference rules*).
A *generating rule* is a rule that takes as *premises* clauses $C_1, \ldots, C_n$, and if a specified formula $\varphi[C_1, \ldots, C_n, C']$ holds, it derives a new clause C'. An inference is sound if, whenever for some $C_1, \ldots, C_n, C'$ the formula $\varphi[C_1, \ldots, C_n, C']$ holds, the C' is a logical consequence of $C_1, \ldots, C_n$. A generating inference can be written as

$$\frac{C_1 \quad \ldots \quad C_n}{C'} \quad \varphi$$

A *simplifying rule* is a rule that has *premise* clauses $C_1, \ldots, C_n$, a reduced clause D and a consequence clause D'. Simplification is performed if a specified formula $\varphi[C_1, \ldots, C_n, D, D']$ holds. It makes the clause D *redundant* and adds a new conclusion clause D'. It can be written as

$$\frac{C_1 \quad \ldots \quad C_n \quad \not{D}}{D'} \quad \varphi$$

A *deleting rule* is a rule that has *premise* clauses $C_1, \ldots, C_n$ and a reduced clause D. Deletion is performed if a specified formula $\varphi[C_1, \ldots, C_n, D]$ holds. The reduction makes clause D *redundant*. It can be written as

$$\frac{C_1 \quad \ldots \quad C_n \quad \not\emptyset}{\varphi}$$

Definition 2.3.2 A generating (simplifying) inference is *sound* if whenever it is performed, the consequence logically follows from the premises (and from the reduced clause in case of a simplifying inference). A simplifying (deleting) rule *preserves completeness with respect to a simplification ordering* $\succ$ if the reduced clause logically follows from the premises (and from the consequence in the case of a simplifying inference), and if the reduced clause is larger than the premise clauses (and than the consequence in a simplifying inference).

Definition 2.3.3 *Saturation sequence* is a possibly infinite sequence of sets of clauses $S_0, \ldots$ such that S_0 is the set of input clauses, and for each $i > 0$, S_i is obtained from S_{i-1} by applying an inference rule of a calculus. If a saturation sequence is finite with the last set of clauses S_N, generating inferences applied to S_N can produce only clauses that are consequences of clauses in S_N that are smaller in the ordering $\succeq$.

The property of interest on a saturation sequence is whether it derives an empty clause at some point. If an empty clause is derived, from the soundness of the calculus rules we know that the original set of clauses is unsatisfiable.

To be able to say something on the sequences which do not derive empty clauses, we need the notion of *fairness* and *completeness*.

Definition 2.3.4 For a saturation sequence $S_0, \ldots$ we define the set of *infinitely occurring clauses* S_∞ to be set of clauses C such that for some $i \geq 0$ it holds that $\forall j \geq i : C \in S_i$. A saturation sequence is *fair* if it is finite, or if all non-redundant generating inferences in S_∞ are eventually performed.

Definition 2.3.5 A calculus is *complete* if for every unsatisfiable set of clauses S and every fair saturation sequence starting with S, it eventually derives an empty clause.

2.3.1 Resolution calculus

Below we present the base resolution and superposition calculus used in our theorem prover Vampire. We do not include the proof of completeness of this calculus; such proof can be found for example in [NR01]. Furthermore, the implementation in the Vampire prover has several extensions beyond what is described here. Probably the most important extension, splitting, is discussed in the Chapter 9.

Generating Rules

Binary Resolution is the following rule:

$$\frac{\underline{A} \vee C_1 \quad \underline{\neg B} \vee C_2}{(C_1 \vee C_2)\theta}$$

where θ is the most general unifier of A and B. The underlining of literals A and $\neg B$ means that they are selected in their clauses.

Factoring is the following rule:

$$\frac{\underline{A} \vee \underline{B} \vee C}{(A \vee C)\theta}$$

where θ is the most general unifier of A and B.

Superposition is one of the following rules:

$$\frac{\underline{l = r} \vee C_1 \quad \underline{L[s]} \vee C_2}{(L[r] \vee C_1 \vee C_2)\theta} \quad \text{or} \quad \frac{\underline{l = r} \vee C_1 \quad \underline{t[s] = t'} \vee C_2}{(t[r] = t' \vee C_1 \vee C_2)\theta}$$

where

- θ is the most general unifier of l and s
- s is not a variable
- $r\theta \not\succeq l\theta$
- (the first rule only) $L[s]$ is not an equality literal
- (the second rule only) $t'\theta \not\succeq t[s]\theta$

Equality resolution is the following rule:

$$\frac{\underline{s \neq t} \vee C}{C\theta}$$

where θ is the most general unifier of s and t.

Equality factoring is the following rule:

$$\frac{\underline{s = t} \vee s' = t' \vee C}{(s = t \vee t \neq t' \vee C)\theta}$$

where

▷ θ is the most general unifier of s and s'.

▷ $t\theta \not\succeq s\theta$

▷ $t'\theta \not\succ \theta$

Simplifying Rules

Duplicate literal deletion is the following rule:

$$\frac{\cancel{L} \vee \cancel{L} \vee C}{L \vee C}$$

Trivial non-equality removal is the following rule:

$$\frac{\cancel{t \neq t} \vee C}{C}$$

Demodulation is the following rule:

$$\frac{l = r \quad L[\cancel{l\theta}] \vee C}{L[r\theta] \vee C}$$

where $l\theta \succ r\theta$.

Subsumption resolution is one of the following rules:

$$\frac{A \vee C \quad \cancel{\neg B} \vee \cancel{D}}{D} \quad \text{or} \quad \frac{\neg A \vee C \quad \cancel{B} \vee \cancel{D}}{D}$$

such that for some substitution θ we have $A\theta \equiv B$ and $C\theta \subseteq D$.

Deleting Rules

Subsumption is the following rule:

$$\frac{C \quad \cancel{D}}{}$$

if there exists a substitution θ such that $C\theta$ is a submultiset of D.

Tautology deletion is one of the following rules:

$$\frac{A \vee \neg A \vee C}{} \quad \text{or} \quad \frac{t \simeq t \vee C}{}$$

2.4 Given Clause Algorithm

Having a complete calculus, we still need to perform its inferences in a fair manner. The generally used approach is called a "given clause algorithm."

This algorithm maintains clauses in two sets — a *passive* and an *active* set. In the beginning, all input clauses are in the passive set, and it is an invariant of the algorithm that all possible inferences between the clauses in the active set have been performed.

If there are no more passive clauses, we terminate having shown the input set of clauses is satisfiable. Otherwise, we select one clause from the passive set (we refer to the clause as *given*, hence the given clause algorithm), and perform all possible inferences between this clause and the clauses in the active set. First, we attempt to delete or simplify the clause using simplifying and deleting rules of the calculus. If we cannot do that, we try to use the given clause to simplify or delete the active clauses. Finally, we perform generating inferences between the given clause and all of the active clauses. Conclusions of these inferences are added into the passive set. Finally the given clause is added among active and we carry on selecting another passive clause to become given.

Chapter 3

Sine Qua Non for Large Theory Reasoning

Authors: Krystof Hoder, Andrei Voronkov

One possible way to deal with large theories is to have a good selection method for relevant axioms. This is confirmed by the fact that the largest available first-order knowledge base (the Open CYC) contains over 3 million axioms, while answering queries to it usually requires not more than a few dozen axioms. A method for axiom selection has been proposed by the first author in the Sumo INference Engine (SInE) system. SInE has won the large theory division of CASC in 2008. The method turned out to be so successful that the next two years it was used by the winner as well as by several other competing systems. This paper contains the presentation of the method and describes experiments with it in the theorem prover Vampire.

3.1 Introduction

First-order theorem provers traditionally were designed for working with relatively small collections of input formulas, for example, those based on a small axiomatisation of a class of algebras, or some axiomatisation of a set theory. Recently, several very large first-order axiomatisations and problems using these axiomatisations have become available. Problems of this kind usually come either from knowledge-base reasoning over large ontologies (such as SUMO [NP01] and CYC [Len95]) or from reasoning over large mathematical libraries (such as MIZAR [Rud92]). Solving these problems usually involves reasoning in theories that contain thousands to millions of axioms, of which only a few are going to be used in proofs we are looking for.

Reasoning with very large theories expressed in first-order logic requires radical re-design of theorem provers. For example, a quadratic time preprocessing algorithm (such algorithms were routinely used in the past) may become prohibitively expensive when the input contains a million formulas.

The first-order problems we will discuss in this paper consist of a very large (thousands to millions) set Ax of *axioms*, plus a small number of additional *assumptions* $A_1, \ldots, A_n$ and a *conjecture* G, sometimes also called a *goal*. We have to prove the conjecture from the axioms and assumptions. Since the set of additional assumptions is normally small (and often empty), it will be convenient for us to assume that we have a large set of axioms and a single goal $A_1 \wedge \ldots \wedge A_n \implies G$. When we discuss complexity of algorithms in this paper, we assume that all axioms and the goal are *small*, for example, have a size (number of symbols, connectives and quantifiers) bound by a constant.

If the conjecture is provable from the axioms, then it is normally provable from a very small subset of these axioms. For example, some of the CYC problems mentioned above contain over 3,000,000 axioms, and all of these problems have proofs involving only less than 20 axioms. If we only use the axioms occurring in such a proof instead of all axioms, a proof will be found by any modern theorem prover in essentially no time.

Provided that only a tiny subset T of axioms is sufficient for finding a proof, one can try to select a small subset $S \subseteq \mathsf{Ax}$ of axioms, which is likely to contain T, and search for a proof using a standard first-order theorem prover on the subset S instead of Ax. It is common that the subset S we are trying to select consists of the axioms *most relevant* to the goal. This paper describes an algorithm for axiom selection. The first version of the algorithm was originally introduced by the first author and implemented in the system SInE. The version and options described here are implemented in the theorem prover Vampire [HKV10].

This paper is structured as follows. In Section 3.2 we discuss the problem of selecting axioms relevant to a goal and the natural idea of *symbol-based selection*. Based on this discussion, in Subsection 3.2.2 we introduce a definition of *trigger-based selection*, which captures a special case of symbol-based selection. In Section 3.3 we present the Sine selection algorithm as a trigger-based selection algorithm. In Section 3.4 we discuss possible variations of this algorithm obtained by changing the trigger relation to overcome potential shortcomings of Sine selection. We also describe the Vampire parameters that can be used to invoke these variations.

Section 3.5 presents experimental results carried out over TPTP problems with large axiomatisations. It shows the effect of various parameter values on the size of the selected set of axioms, the number of iterations of the algorithm, and on the ability to solve hard TPTP problems. Section 3.6 describes the use of our selection method in the recent CASC competitions. In Section 3.7 we briefly overview other algorithms used for selection of relevant axioms and other related work.

3.2 Symbol-Based Selection

3.2.1 Idea: Relevance

When one thinks of selecting axioms relevant to a goal, perhaps the most natural idea is to use *symbol-based selection*. By a symbol we mean any predicate or function symbol (including constants) apart from equality $=$. Symbol-based selection means that axioms are selected based on symbols occurring in them. Let us call two symbols *neighbours* if either they occur in the same axiom. Let us also call two symbols s_1, s_2 *relevant* if (s_1, s_2) belongs to the reflexive and transitive closure of the neighbour relation. We will say that a symbol is *relevant* (to the goal) if it is relevant to a symbol occurring in the goal. Likewise, we say that an *axiom is relevant* if it contains at least one relevant symbol. Note that in an axiom containing at least one relevant symbol, all symbols will be relevant too.

One possible way of axiom selection is to use all relevant axioms. However, for all benchmark suites available to us, the set of relevant symbols is usually the set of all or nearly all axioms. This is mostly due to the use of very general relations having occurrences in many axioms: as soon as any such relation is relevant, all axioms containing this relation become relevant. We will refer to symbols having occurrences in many axioms as to *common symbols*. Typical examples of common symbols are "instance-of" and "subclass" relations in ontologies: many ontologies consists almost exclusively of axioms using these two relations (or similar relations, such as "subsumes"). Therefore, any selection procedure that tries to avoid selecting nearly all axioms should solve the problem of common symbols.

Another idea for selecting fewer axioms is not to use the full reflexive and transitive closure of the neighbour relation but only a subset thereof, for example by only allowing to make n steps of the computation of the transitive closure, for some (small) positive integer n. Let us formalise the relevancy relation, so that we can refine it later.

More precisely, we will deal with two relevancy relations, one for symbols and another for axioms.

1. If s is a symbol occurring in the goal, then s is 0-step relevant.

2. If s is k-step relevant and occurs in an axiom A, then A is $k+1$-step relevant.

3. A is k-step relevant and s occurs in A, then s is k-step relevant, too.

Clearly, a symbol or an axiom is relevant, if it is k-relevant for some $k \geq 0$. This definition implies also that a k-relevant symbol or axiom is m-relevant for every $m \geq k$.

One can use this inductive definition to select either the set of all k-relevant, for some fixed k, axioms, or the set of all relevant axioms. To compute the latter, we can mark all relevant axioms until the inductive step does not select any new axiom. Moreover, it is easy to implement this algorithm in the time linear in the size of the set of all axioms.

Even better, assuming that the set of all axioms is fixed (which is a natural assumption for applications), by preprocessing the set of axioms one can compute the set of relevant (or k-step relevant) axioms in the time *linear in the size of the computed set*. The key step is the second part of the inductive definition of relevance, as it refers to all axioms in which a symbol occurs. To answer queries for all such axioms, it is sufficient to index the set of all pairs (s,A) such that s is a symbol occurring in an axiom A (of course, A can be represented as a reference in the index). This index can be implemented by storing for every symbol the set of axioms in which it occurs. Let us show that such an algorithm is indeed linear in the size of the computed set. To this end, let us consider the set of pairs (s,A) inspected by this algorithm. Note that for every such pair, A will be included in the output, and an axiom A occurs in the number of pairs that is bounded by its length $len(A)$. This implies that, for a given axiom A, pairs of the form (s,A) can be inspected at most $len(A)$ times. This shows that if axioms $A_1,\ldots,A_n$ of the total size $m = len(A_1)+\ldots+len(A_n)$ were selected, the run of the algorithm inspected $O(m)$ pairs, therefore its time complexity is also $O(m)$.

Note that using k-step relevance instead of relevance does not solve the problem of common relations, since they can already become relevant for very small values of k.

3.2.2 General Scheme

We will introduce a class of symbol-based selection algorithms that generalise and refine the idea of symbol-based selection. The only deviation from the previous definition is to add extra conditions for an axiom to be relevant. We will achieve this by introducing a trigger relation as follows. Suppose that we have a relation $trigger(s,A)$ between symbols and axioms, such that this relation holds only if s occurs in A. If this relation holds, we will also say that *s triggers A*.

Definition 3.2.1 [trigger-based selection]

1. If s is a symbol occurring in the goal, then s is 0-step triggered.

2. If s is k-step triggered and s triggers A, then A is $k+1$-step triggered.

3. A is k-step triggered and s occurs in A, then s is k-step triggered, too.

An axiom or a symbol is called *triggered* if it is k-triggered for some $k \geq 0$.

It is easy to see that the introduction of the trigger relation can solve the problem of common symbols. One can simply postulate that s triggers A only if s is not a common symbol. Or, to be on a safe side, one can say that a common symbol s triggers A only if all symbols of A are common. This would avoid excluding axioms like

$$subclass(x,y) \wedge subclass(y,z) \implies subclass(x,z)$$

(transitivity of the subclass relation) or

$$instanceof(x,y) \wedge subclass(y,z) \implies instanceof(x,z).$$

It is not hard to argue that if the set of all symbols s such that s triggers axiom A can be computed in the time linear in the size of A, then the set T of all triggered (as well as all k-step triggered) axioms can be computed in time linear in the size of T. This is achieved by keeping a mapping from symbols to the sets of axioms which they trigger.

3.3 The Sine Selection

3.3.1 Idea

The Sine selection algorithm is a special case of trigger-based selection. It uses a trigger relation that tries to reflect, in a certain way, the hard-to-formalise notions *"s_2 is defined using s_1"* or *"s_1 is more general than s_2"* on symbols.

It is not unreasonable to assume that large knowledge bases contain large hierarchical collections of definitions, where more general terms are defined using less general terms. It is not easy to extract such definitions, since they can take various forms. It is also not easy to formalise the relation "more general". As a simple approximation to "more general" one can consider the relation "more common": a symbol s_2 is considered more common than s_1 if s_1 occurs in more axioms than s_2. Then, as a potential approximation to *"s_2 is defined in terms of s_1"* we can consider the relation *"s_1, s_2 occur in the same axiom A and s_2 is a least common symbol in A."* This is, essentially, the definition of the trigger function for the Sine selection.

Definition 3.3.1 [Trigger relation for the Sine selection] Let us denote by $occ(s)$ the number of axioms in which the symbol s appears. Then we define the relation *trigger* as follows: $trigger(s,A)$ iff for all symbols s' occurring in A we have $occ(s) \leq occ(s')$. In other words, an axiom is only triggered by the least common symbols occurring in it.

3.3.2 Examples

Example 3.3.2 [Sine selection] In this example we will denote variables by capital letters. The example illustrates the Sine selection and also a typical reason why it is, in general, incomplete. Consider the following set of axioms:

`subclass(X,Y)` $\wedge$ `subclass(Y,Z)` $\rightarrow$ `subclass(X,Z)`

`subclass(petrol,liquid)`

$\neg$`subclass(stone,liquid)`

`subclass(beverage,liquid)`

`subclass(beer,beverage)`

`subclass(guinness,beer)`

`subclass(pilsner,beer)`

The following table gives, for every symbol s, the number of axioms in which it occurs.

s	$occ(s)$
subclass	7
liquid	3
beer	3
beverage	2
petrol	1
stone	1
guinness	1
pilsner	1

Using the occurrence table, we can compute the trigger relation as follows:

axiom	symbols
subclass(X,Y) ∧ subclass(Y,Z) → subclass(X,Z)	subclass
subclass(petrol,liquid)	petrol
¬subclass(stone,liquid)	stone
subclass(beverage,liquid)	beverage
subclass(beer,beverage)	beverage
subclass(guinness,beer)	guinness
subclass(pilsner,beer)	pilsner

Consider the goal subclass(beer,liquid). This goal is a logical consequence of these axioms. However, the symbols from the goal subclass, beer, and liquid only trigger the first axiom. The selection will terminate with only the first axiom selected, which is insufficient to prove the goal. □

Consider another example. This example illustrates how (small) changes in the input set of axioms can influence selection.

Example 3.3.3 Let us remove the last axiom from the axioms of Example 3.3.2. This changes the function *occ* as follows:

s	$occ(s)$
subclass	6
liquid	3
beer	2
beverage	2
petrol	1
stone	1
guinness	1

This also changes the trigger relation as follows:

axiom	symbols
subclass(X,Y) ∧ subclass(Y,Z) → subclass(X,Z)	subclass
subclass(petrol,liquid)	petrol
¬subclass(stone,liquid)	stone
subclass(beverage,liquid)	beverage
subclass(beer,beverage)	beverage,beer
subclass(guinness,beer)	guinness

Consider the same goal subclass(beer,liquid) as in Example 3.3.2. Now the symbols from the goal subclass, beer, and liquid trigger the first axiom as before plus the axiom subclass(beer,beverage). This results in adding beverage to the list of triggered symbols. As a consequence, this addition triggers the axiom subclass(beverage,liquid). This triggers no new symbols (since beverage was already triggered before), and so selection terminates with the following subset of selected axioms

subclass(X,Y) ∧ subclass(Y,Z) → subclass(X,Z)
subclass(beverage,liquid)
subclass(beer,beverage)

This collection of axioms is sufficient to prove the goal, contrary to Example 3.3.2. Moreover, it is the minimal set of axioms sufficient to prove this goal. □

This example also illustrates that removing some axioms from the input set can result in selecting more axioms than before the removal.

3.3.3 The Selection Algorithm in More Detail

The algorithm runs in two phases. The first phase is goal-independent and only preprocesses the set of all axioms. In this phase we do the following:

1. count, for each symbol, the number of axioms in which it occurs;
2. store the set of pairs (s,A) such that s triggers A.

Note that this phase can be implemented in the time linear in the size of the set of axioms. It can be done by two traversals of the axioms (computing the trigger relation needs the number of occurrences).

In the second phase (which depends on the goal) we build the set of all triggered (or k-triggered, if k is given) axioms using the stored trigger relation. If the trigger relation is indexed on the first argument, the time complexity of the second phase is linear in the size of the resulting set of selected axioms and independent of the overall number

of theory axioms. After the selection, the goal and the selected axioms are passed to a first-order theorem prover.

Separating the two steps of the algorithm provides an efficient way to treat collections of problems that share a large number of theory axioms. After preprocessing the shared theory axiomatisation, only the second phase is run on each of the problems, which allows us to avoid repeated execution of the first phase.

3.4 Variations

To turn the Sine selection on, one uses Vampire with the option

```
--sine_selection on
```

(the default value of this option is `off`). In the previous versions of Vampire we had two values: `axioms` and `included` instead of `on`. The value `axioms` considered as axioms the formulas marked as such in the TPTP language. The value `included` considered as axioms formulas coming from included files.[1] However, the value `included` is fragile since simple preprocessing of the input can change which formulas are included. In addition it turned out not to be good in practice, so in the current version we replaced these two values by a single one.

In the rest of this section we consider several variations of Sine selection. Each of these variations is implemented in Vampire as a parameter whose value can be set by the user. We present experimental evaluation of these parameters in Section 3.5.

3.4.1 Tolerance

Since our selection algorithm is incomplete, we introduced parameters changing the trigger function in various ways. One obvious problem with the selection is that it is very fragile with the respect to the number of axioms in which a symbol occurs, as already illustrated in Examples 3.3.2 and 3.3.3. Indeed, suppose a symbol s_1 occurs in (say) 7 axioms while s_2 occurs in 8 axioms. One can argue that s_1 and s_2 are, essentially, equally common. However, s_1 can trigger an axiom in which both s_1 and s_2 occur, while s_2 cannot trigger it.

To cope with this problem, we introduced a parameter called *tolerance*. The value of this parameter is a real number $t \geq 1$. It changes the trigger relation as follows.

[1] The TPTP language classifies the input formulas into axioms, additional assumptions (hypotheses) and conjectures. Formulas can also be included from files using the TPTP directive include().

Definition 3.4.1 [Trigger relation with tolerance] Given the tolerance $t \geq 1$, define the relation *trigger* as follows: $trigger(s,A)$ iff for all symbols s' occurring in A we have $occ(s) \leq t \cdot occ(s')$.
Compare this definition with Definition 3.3.1.

Example 3.4.2 [Sine selection with tolerance] Consider the set of axioms and the goal of Example 3.3.2. Assume the tolerance value is 1.5. This changes the trigger relation for two of the axioms as follows:

axiom	symbols
`subclass(X,Y) ∧ subclass(Y,Z) → subclass(X,Z)`	`subclass`
`subclass(petrol,liquid)`	`petrol`
`¬subclass(stone,liquid)`	`stone`
`subclass(beverage,liquid)`	`beverage,liquid`
`subclass(beer,beverage)`	`beer,beverage`
`subclass(guinness,beer)`	`guinness`
`subclass(pilsner,beer)`	`pilsner`

For the same goal `subclass(beer,liquid)`, the set of selected axioms becomes

```
subclass(X,Y) ∧ subclass(Y,Z) → subclass(X,Z)
subclass(beverage,liquid)
subclass(beer,beverage)
```

This set is sufficient to prove the goal. □

Note that the set of selected axioms is monotonic with regard to the value of tolerance: if we increase the value, all previously selected axioms will also be selected. For large enough values of tolerance, the set of selected axioms is simply the set of all relevant axioms in the sense of Section 3.2.1, because axioms become triggered by all the symbols that occur in them. For example, in Example 3.3.2 all axioms become selected when $t \geq 3$ and each symbol triggers all axioms in which it occurs when $t \geq 7$.

Having a fixed value of the tolerance parameter, we may perform axiom selection using the two-phase algorithm described in section 3.3.3. However, a likely scenario is that we will want to run several proof attempts with different values of the tolerance parameter. Using the basic two-phase algorithm, we would have to run the first phase of the algorithm for each value of the tolerance parameter. At the cost of a slight increase in the complexity, the algorithm may be modified, so that the first phase is run only once, allowing selection with arbitrary tolerance values. During the first phase, instead of storing a mapping from every symbol s to the set of axioms triggered by s, we store a mapping from s to the list of all pairs $(A_1,t_1),\ldots,(A_m,t_m)$ such that s triggers A_i if $t \geq t_i$. This list is ordered by the values of the t_i's. Ordering such lists can take

$n \cdot log(n)$ time, so the complexity of the first phase slightly increases. However, the complexity of the second phase does not change: it is still linear in the size of selected axioms, since we only inspect the sublist of $(A_1,t_1),\ldots,(A_m,t_m)$ corresponding to the trigger relation.

To change the value of tolerance from the default value 1 to t, one uses Vampire with the option

`--sine_tolerance` t

3.4.2 Depth Limit

One can restrict the number of steps in computing the set of selected axioms so that it computes the set of all d-step triggered axioms. To this end, one can run Vampire with the option

`--sine_depth` d

The default value of `sine_depth` in Vampire is ∞. Evidently, the set of selected axioms is monotonic with regard to d: if we increase the value, all previously selected axioms will be selected, too.

3.4.3 Generality Threshold

The last modification of Vampire is based on the following idea: if a symbol s occurs in few axioms, then it triggers any axiom in which it occurs, even if the axiom contains symbols with fewer occurrences. To implement this, we fix some positive integer value $g \geq 1$ (called generality threshold) and modify the trigger relation as follows.

Definition 3.4.3 [Trigger relation with generality threshold] Given the generality threshold $g \geq 1$, define the relation *trigger* as follows: $trigger(s,A)$ iff either $occ(s) \leq g$ or for all symbols s' occurring in A we have $occ(s) \leq occ(s')$.
To turn the generality threshold in Vampire on, one can run Vampire with the option

`--sine_generality_threshold` g

The default value is 0 and the set of selected axioms is, evidently, monotonic with regard to g. If we set g to a large enough number, for example, the number of all axioms, then (similarly to setting a large value of the tolerance parameter) all the relevant axioms will be selected.

problems	axioms	atoms	predicates	functions
SUMO	298,420	323,170	20	24,430
CYC	3,341,990	5,328,216	204,678	1,050,014
Mizar	44,925	332,143	2,328	6,115

Table 3.1: Average problem size information

To use both a tolerance value t and a generality threshold value g, one should define the trigger relation as the union of the corresponding trigger relations. Namely, $trigger(s,A)$ iff either $occ(s) \leq g$ or for all symbols s' occurring in A we have $occ(s) \leq t \cdot occ(s')$.

3.5 Experiments

All experiments described in this section were carried out using a cluster of 64-bit quad core Dell servers having 12 GB RAM each.[2] Each of the runs used only one core and we never ran more than 3 tests in parallel on one computer to achieve the best performance.

The experiments were run on three benchmark suites taken from the TPTP library [Sut09]. The library contains three different classes of very large problems:[3]

1. problems from the SUMO ontology [NP01]: `CSR075` to `CSR109`.

2. problems from the CYC knowledge base [Len95]; `CSR025` to `CSR074`.

3. problems from the Mizar library [Rud92]: `ALG214` to `ALG234`, `CAT021` to `CAT037`, `GRP618` to `GRP653`, `LAT282` to `LAT380`, `SEU406` to `SEU451`, and `TOP023` to `TOP048`.

Each of these classes contains several different axiomatisations. To evaluate the size of the set of selected axioms and the number of iterations we only considered the largest axiomatisations in each class (that is, SUMO problems with the suffix `+3`, CYC problems with the suffix `+6`, and Mizar problems with the suffix `+4`. Table 3.1 contains information about sizes of these problems.

[2]The cluster was donated to our group by the Royal Society.

[3]These classes correspond to categories of the LTB division in the CASC competition [Sut10].

	$t = 1.0$		$t = 1.2$		$t = 1.5$		$t = 2.0$		$t = 3.0$		$t = 5.0$	
$d = 1$	29	1.17	35	1.09	41	1.05	47	1.02	60	1.02	72	1.01
$d = 2$	142	1.25	287	1.07	442	1.03	607	1.01	1027	1.00	1476	1.00
$d = 3$	505	1.32	937	1.13	1451	1.07	2484	1.02	5311	1.01	10482	1.01
$d = 4$	1784	1.41	3232	1.20	5716	1.10	11603	1.02	29963	1.01	69015	1.01
$d = 5$	4432	1.57	8870	1.27	16806	1.13	37599	1.03	110186	1.02	249192	1.04
$d = 7$	10698	2.16	25607	1.50	56337	1.21	150277	1.06	431875	1.09	832935	1.10
$d = \infty$	36356	28.37	495360	3.33	1310965	1.34	1562064	1.20	1822427	1.12	2057597	1.07

Table 3.2: Selected formulas of CYC problems depending on the depth, tolerance and generality threshold

3.5.1 Generality Threshold

Table 3.2 shows how the number of selected formulas depends on the generality threshold. We considered the smallest possible value $g = 0$ and a sufficiently large value $g = 16$. In every column of the table we show on the left the number of axioms selected when $g = 0$ and on the right the number of axioms selected when $g = 16$ divided by the value on the left. The numbers are average over all CYC problems. One can see from the table that the largest increase (by the factor of 28.37) was achieved when the depth was unlimited and tolerance equal to 1. Predictably, when the tolerance grows, the percentage of additional axioms selected by the generality threshold value becomes smaller, since some axioms selected due to the large value of generality threshold become selected due to the high tolerance.

Our results have also shown that all the problems Vampire could solve, could also be solved with the value $g = 0$, so for the rest of this section we will focus on the remaining two options (depth and tolerance) and only consider the results obtained when the generality threshold was not used, that is, when $g = 0$. Therefore, the conclusion we can draw is that, although the generality threshold parameter is intended to cope with the problem of common relations, in practice it can be replaced by other parameters.

3.5.2 Selected Formulas

Table 3.2 shows the average numbers of selected axioms for the CYC problems, and table 3.3 shows these numbers for the Mizar and SUMO problems. Note that the numbers for the Mizar problems are essentially different from the SUMO and CYC problems. The number of selected Mizar axioms is large (over 4,000) even when the depth limit is set to 1, while it is relatively small for the CYC and SUMO problems.

$d\backslash t$	1.0	1.2	1.5	2.0	3.0	5.0
1	4903	4911	4921	4936	4973	5038
2	5296	5395	5553	5823	6427	7743
3	6118	6451	7068	8280	10841	16337
4	6893	7556	9001	12176	18300	28878
5	7432	8517	11165	16945	26842	37284
7	7897	9991	15788	26203	36507	41443
∞	8047	15987	28353	35345	39389	41762

$d\backslash t$	1.0	1.2	1.5	2.0	3.0	5.0
1	12	13	14	16	21	28
2	70	82	115	158	272	654
3	188	230	372	762	1950	5980
4	316	470	942	3021	8720	23440
5	540	979	2417	8179	22644	52241
7	1027	2708	8517	24445	54958	97481
∞	1116	8361	26959	57322	82379	107926

Table 3.3: The number of selected formulas for the Mizar (above) and SUMO (below) problems

This is related to the fact that the Mizar problems usually have complex goals containing several assumptions and many symbols, while for the CYC and SUMO problems the goal is simple and usually is just an atomic formula with very few symbols. This is also one of the reasons why Mizar problems are much harder for all theorem provers using the Sine selection.

3.5.3 Number of Iterations

The next question we are interested in is the number of steps required by Sine selection to compute the set of all triggered relations. Table 3.4 contains statistics about the number of iterations. To our surprise, in some cases this number is very large (135 for CYC problems, 61 for Mizar problems, and 39 for SUMO problems). There is also no obvious pattern on how this parameter depends on the value of tolerance.

3.5.4 Essential Parameters

For experiments described in this subsection we considered all (not only largest) SUMO, CYC and Mizar problems.

suite		$t = 1.0$	$t = 1.2$	$t = 1.5$	$t = 2.0$	$t = 3.0$	$t = 5.0$
CYC	min	4	7	36	25	29	23
	avg	19.3	102.7	44.3	29.9	33.0	25.1
	max	47	135	60	41	37	27
Mizar	min	6	7	19	15	14	10
	avg	9.5	27.5	25.4	19.7	15.0	12.3
	max	15	61	33	23	18	14
SUMO	min	3	5	4	6	13	11
	avg	7.3	15.3	20.8	19.8	16.6	12.6
	max	17	35	39	25	23	15

Table 3.4: The minimal, average and maximal number of steps required to build the set of all triggered axioms as a function of tolerance

Since our main aim is to automatically prove (hard) theorems, the most important questions related to the use of Sine selection are the following:

1. How powerful is the selection method?
2. Which of the parameters (and ranges of values for these parameters) are essential in practice?

The first question cannot be answered in a simple way. On the one hand, we have strong evidence that the method is very powerful. To support this, consider Table 3.5. It shows the number of all the TPTP 4.0.1 CYC, Mizar, and SUMO problems solved with some Sine selection and without it, depending on the size of the problem measured as the number of atoms in it. The last row shows these numbers for the problems having TPTP rating 1. A problem has TPTP rating 1 if it was previously unsolved by all provers, including the previous versions of Vampire. For example, among the problems having 80,000 or more atoms, 373 problems were solved by Vampire all together. 187 problems could only be solved with the help of Sine selection, while only 3 problems out of 373 could not be solved with Sine selection.

When a problem is not solved, we do not know why Vampire (or any other prover) fails to prove it. This can be for at least the following three reasons, of which two are directly related to the power of our selection method:

1. the set of selected axioms can be insufficient to prove the goal;
2. the set of selected axioms is too large, which prevents theorem provers from success;

atoms	only with Sine	only without Sine	together
10,000	243	64	721
20,000	217	10	542
40,000	208	7	464
80,000	187	3	373
160,000	138	1	243
320,000	80	1	168
640,000	50	0	100
1,280,000	50	0	50
rating 1	232	25	402

Table 3.5: Problems solved with and without Sine selection

3. the problem is very hard even for small sets of axioms sufficient to prove the goal.

Let us now investigate which parameters and their values are essential for Vampire. As we pointed out, it turned out that the generality threshold parameter can be dropped without any effect on the set of problems solved by Vampire. It turned out that both the tolerance and depth limit are very essential. To show this, we used our database of proofs found by Vampire, which was generated using about 70 CPU years of run time and now contains about 575,000 results, of which over 43,000 are related to the mentioned benchmark suite.

We selected problems having less than 10 solutions in the database. The reason to use the number of solutions as a criterion was that problems with few solutions are believed to be harder. Also, such problems can be solved only with a small subset of possible values for the various Vampire parameters.

This selection resulted in 51 CYC problems, 231 Mizar problems and 12 SUMO problems. For each of these problems we checked which parameter values solve these problems. More precisely, we took their known solutions, changed the depth and tolerance parameters and checked which of the changes still solve the problem. The results are summarised in Tables 3.6 and 3.7. The table cells show the number of problems which can be solved by the given range of values. We do a projection on the possible values of the parameter that is not present in the table (tolerance in the case of Table 3.6 and depth limit in Table 3.7). For example, the third row in Table 3.7 means that there were 12 selected Mizar problems and 1 selected SUMO problem that could be solved with some values of the depth limit and only with the values of tolerance between 1.0 and 1.5.

range	CYC	Mizar	SUMO
$1-1$		16	
$1-2$		10	
$1-3$		5	
$1-4$		3	
$1-5$		2	
$1-10$		1	
$1-\infty$	15	107	6
$2-2$		21	
$2-3$		12	
$2-4$		3	
$2-5$		6	
$2-7$		1	
$2-10$		1	
$2-\infty$	21	39	4
$3-3$		1	
$3-4$		1	
$3-\infty$	6	1	1
$4-4$		1	
$4-\infty$	6		
$5-\infty$	3		
$10-\infty$	3		1
total	51	231	12

Table 3.6: The sine depth range for solved hard problems

range	CYC	Mizar	SUMO
$1.0-1.0$		3	
$1.0-1.2$		4	
$1.0-1.5$		12	1
$1.0-2.0$		17	
$1.0-3.0$		19	
$1.0-5.0$	49	155	11
$1.2-1.5$		2	
$1.2-2.0$		1	
$1.2-3.0$		1	
$1.2-5.0$	2	1	
$1.5-5.0$		1	
$2.0-2.0$		1	
$2.0-3.0$		7	
$2.0-5.0$		1	
$3.0-5.0$		3	
$5.0-5.0$		2	
total	51	231	12

Table 3.7: The sine tolerance range for solved hard problems

Let us first analyse the depth limit in Table 3.6. For the evaluation we used the following values of depth limit: 1, 2, 3, 4, 5, 7, 10 and ∞. The first observation is that this parameter is, indeed, very important. For example, there were 39 Mizar problems that could be solved with only one value of this parameter (1, 2, 3 or 4). For both CYC and SUMO collections setting the depth to ∞ is always a good strategy. On the contrary, only 147 out of 231 solved Mizar problems (and only 30 of 64 the largest Mizar problems) could be solved with this setting.

Next, let us analyse the tolerance in Table 3.7. For the evaluation we used the following values of tolerance: 1.0, 1.2, 1.5, 2.0, 3.0, 4.0, and 5.0. It turned out that this parameter is also very important. However, the behaviour of solutions depending on this parameter is more stable than for the depth parameter: only 6 Mizar problems could be solved with exactly one value of the tolerance and 155 Mizar problems out of 231 were solved with all the values we tried. Among the largest hard Mizar problems,

only 36 out of 64 were solved with all the tried values of tolerance. 11 out of 12 SUMO problems and 49 out of 51 CYC problems could be solved with any value of the tolerance and all CYC problems could be solved with the value 1.2 or higher.

3.6 Competition Performance

Our axiom selection algorithm was used by several systems participating in the Large Theory (LTB) division of recent CASC competitions.

The Sine selection was introduced in 2008, at the CASC-J4 [Sut08] competition. The only participant that used our algorithm was the SInE theorem prover. It has won the division by solving 88 out of 150 problems, which was 12 problems ahead of the second best participant.

In 2009, at the CASC-22 [Sut10] competition, four out of seven participants were using our selection algorithm, and these four participants ended up at the first four positions, solving 69 to 35 problems out of 100, while the best participant not using our algorithm solved only 18 problems.

In 2010, at the CASC-J5 competition, five out of seven systems were using our algorithm as the only axiom selection algorithm, including the winner (Vampire). The second best ranked system (Currahee) used our selection algorithm as one of possible selection algorithms.

3.7 Related work

In our algorithm we maintain the set of selected axioms (starting from goal), and select new axioms that are relevant to the goal step by step. The lightweight relevance filtering algorithm [MP09] shares this approach, but instead of a trigger relation, which is used by our algorithm, it selects an axiom if certain percentage of its symbols appears in the already selected axioms. This method also penalizes common symbols—the more often a symbol appears in the problem, the less impact its appearance in an axiom has.

Several other algorithms use some measure of distance in a graph of axioms, in order to determine which axioms are relevant to the goal. The contextual relevance filtering [ARS09] and the syntactic relevance measure [SP07] compute the weighted graph between axioms with weights based on the number of shared symbols (taking into account their commonness). The latter does not use the distance from conjecture

to select relevant axioms, but to order them for further (semantic) processing. The relevance restriction strategy described in [PY03] connects two clauses in the graph if they have unifiable literals. The paper also examines the suitability of different graph distance measures.

Semantic algorithms are another group of axiom selection algorithms that use a model of currently selected axioms to guide the selection of new ones. To this group belong the Semantic Relevance Axiom Selection System [SP07] and the algorithm for semantic selection of premises [Pud07].

Yet another approach is taken in the latent semantic analysis [ARS09], which uses a technique for analysing relationships between documents. Each formula is considered to be a document, and formulas with strong relationship toward the goal are selected. The MaLARea system [USPV08] uses machine learning on previous proofs in the theory to estimate which theory axioms are likely to contribute to proofs of new problems.

In [UHV10] the benefit of our axiom selection algorithm for reasoning on the Mizar Mathematical Library [Urb06] is examined.

3.8 Conclusion

We defined the Sine selection used in the theorem prover Vampire and several other theorem provers to select axioms potentially relevant to the goal. We formalised the Sine selection as a family of *trigger-based selection algorithms*. We showed that all the existing axiom selection parameters in Vampire can be formalised as a special case of such algorithms.

We also discussed, using extensive experiments over all TPTP problems with large axiomatisations, the effect of various parameter values on the size of the selected set of axioms, the number of iterations of the algorithm, and solutions of hard TPTP problems.

We also added a new mode to Vampire to make others able to experiment with our axiom selection. If Vampire is run in a new *axiom selection mode*, it does not try to prove the problem but only selects axioms according to the user-given options and outputs the selected axioms and the goal in the TPTP format. This mode can be invoked by using `vampire --mode axiom_selection`.

Acknowledgements

The authors would like to thank Josef Urban for helpful suggestions, and for his support and supervision of the first author during the first year of the work on the Sine algorithm.

Chapter 4

Evaluation of Automated Theorem Proving on the Mizar Mathematical Library

Authors: Josef Urban, Krystof Hoder, Andrei Voronkov

This paper investigates the strength of first-order automatic theorem provers (ATPs) in proving theorems and lemmas from the Mizar proof assistant's formal mathematical library. Several Mizar use-cases are described and evaluated, as well as various ATP systems and strategies. The new version of the leading Vampire ATP system is included in the evaluation, experiments with Mizar-specific strategy-selection are performed with E the prover, and the Sine axiom selection is evaluated on large Mizar problems with both E and Vampire. A rough mathematical division of the Mizar library is introduced, and the ATP performance is evaluated on it.

4.1 Introduction and Motivation

In the last five years there was a considerable increase of the use of fully automatic first-order theorem provers as assistants to interactive theorem provers (ITPs). A number of formal knowledge bases and core logics have been translated to first-order ATP formats such as TPTP.[1] For example, let us mention the related work on the Isabelle/Sledgehammer ATP link [MP09], and the export of the SUMO [PS07] and CYC [MJWD06] real-world formal knowledge bases.[2]

[1] Thousands of Problems for Theorem Provers, see `www.tptp.org`

[2] `www.ontologyportal.org` and `www.cyc.com`

One of the main goals of the MPTP[3] project is to make the large Mizar Mathematical Library[4] (MML) accessible to ATP and AI experiments and techniques. The particular value of MML/MPTP in comparison to the above mentioned related projects is that this is a comparatively large library focused primarily on standard mathematics as done by mainstream mathematicians (using first-order logic and set theory as the foundations). The first-order setting allows a practically complete and reasonably efficient translation for first-order ATPs, which is harder to do for higher-order systems. The size of the library and its consistency on the symbol-naming and theorem-naming level also allows experimenting with all kinds of "knowledge-based" ATP/AI techniques, which might be relevant for emulating the thinking of learned mathematicians, and bringing new insights to the fields of ATP and AI.

The first inclusion of the MML/MPTP problems in ATP benchmarks (the TPTP library) happened in 2006, when also the large-theory MPTP Challenge[5] was announced. Since then the CASC[6] Large Theory Batch (LTB) competition was introduced, and run already twice in 2008 and 2009. This influences the performance and tuning of existing ATP systems, and gives rise to new techniques and interesting metasystems.

The purpose of the current paper is to evaluate the progress made over the past five years in the area of reasoning in large formal mathematical theories, and particularly evaluate the strongest ATP systems and metasystems on sufficiently recent MML/MPTP and the mathematical subfields contained in it.

4.1.1 Recent Evolution of Mizar and MPTP

Despite its age, Mizar is a living and evolving system with a number of users around the world. Since the last published MPTP experiments done on MML version 938 (938 articles), a number of articles have been added to the library resulting in thousands of new "Mizar theorems"[7]. These range from a number of standard calculus results developing, e.g., the Riemann integral, to abstract algebra results like Sylow theorems [Ric07], to formalization of special fields like BCI/BCK algebras [Din07]

[3] Mizar Problems for Theorem Proving

[4] `www.mizar.org`

[5] `http://www.tptp.org/MPTPChallenge/`

[6] The CADE ATP System Competition, see `http://www.tptp.org/CASC/`

[7] We put Mizar theorems in quotes at least once to deliver the message that only very few of these "theorems" would be called a "theorem" by mathematicians. Large majority of these propositions are lemmas useful and re-usable for proving further results, and this property makes them "theorems" in the Mizar parlance.

to results from mathematical theory of social choice like Arrow's Impossibility Theorem [Wie07].

At least the following developments have been tried/done with the Mizar system between these two versions:

- ▷ The Mizar type system mechanisms (Horn-like mechanisms automatically inferring monadic adjectives about the objects of the set-theoretical universe) have been constantly strengthened, becoming one of the main automation tools in Mizar.
- ▷ Experiments have been done with strengthening the matching/unification mechanisms in the Mizar kernel module.
- ▷ Identifications (i.e., registered automated equalities applied implicitly by the system) have been introduced by Mizar and used in MML.
- ▷ Further elements of computer algebra have been introduced in the kernel module, to allow automated normalization and solving of systems of linear equations.

The development of MPTP has to reflect the Mizar/MML changes. Also, as a relatively young system, MPTP has a number of its own developments to do. Here is a short summary of the recent ones:

- ▷ Probably the largest change is that initial methods for ATP-export of Mizar internal arithmetics have been implemented. This is a constant cat-and-mouse pursuit with the experiments done with computer algebra in the Mizar kernel,[8] however it is now possible to do ATP experiments over Mizar problems containing arithmetics. The export is correct, but not always complete.[9] However, as can be seen in Section 4.3, counter-satisfiability is detected only reasonably rarely in practice by ATPs.
- ▷ MPTP changes in ATP problem creation, accommodating the new developments in the Mizar type automations, and introduction of identifications.
- ▷ Changes making MPTP faster and more real-time, including:

[8] This is the main reason why we choose a version of MML that is not completely recent at the time of writing this evaluation. The MML version 1011 that we choose for the evaluation here has now been sufficiently tested, and the ATP-export of Mizar internal arithmetics sufficiently debugged. It would be possible to experiment with a more recent MML version, however for a large-scale evaluation it is preferable to use a reasonable recent version for which MPTP is known to work well.

[9] This also really depends on the particular version of the Mizar kernel.

description	proved	countersatisfiable	timeout or memory out	total
E 0.9	4309	0	8220	12529
SPASS 2.1	3850	0	8679	12529
together	4854	0	7675	12529

Table 4.1: Reproving of the theorems from non-numerical articles by MPTP 0.2 in 2005

- ▷ More advanced (graph-like) datastructures to speed-up the process of selecting necessary parts of the library for generating the ATP problems.
- ▷ Larger use of available Prolog indexing and the asserted database for various critical parts of the code.
- ▷ Instead of working always with the whole loaded MML, MPTP was refactored to allow working only with the (usually much smaller) part of the MML needed for the newly processed article. This is specifically required for the new ATP-for-Mizar (MIZAR) service running now in real time at the RU Foundations' group server[10] [US10].

The summary of data from previous experiments with SPASS (version 2.1) and E (version 0.9) from 2005 on MML version 938 using MPTP 0.2 is given in Table 4.1 (see [Urb06] for details).

Note that these experiments have been done in 2005 only on "non-numerical" articles (containing 12529 theorems/problems), i.e., on Mizar articles guaranteed not to contain any arithmetical evaluations. The current experiments described in Section 4.3 are however performed on the whole MML, because a basic ATP-export of Mizar computer algebra is now available.

4.2 Mizar data, experimental setup

The experiments described in Section 4.3 are performed on three classes of data,[11] all coming from the proofs of all Mizar theorems from MML version 1011. There are 51424 theorems in this MML version. The classes differ by the average number of axioms (previous theorems and definitions from MML) included in the problems, coming from different Mizar use-cases. The classes and use-cases are as follows:

[10] http://mws.cs.ru.nl/~mptp/MizAR.html

[11] available at http://mws.cs.ru.nl/~mptp/mptp_1011/noint/

- ▷ *SMALL*: Problems with smallest number of included axioms. This use-case models a user who knows relatively well how a proof should proceed (what MML knowledge should roughly be used). In the HOL Light (established for its ITP/ATP inventions) terminology: MESON_TACTIC[12] . The average size of an MPTP problem in this class is 218 formulas. Many of these theorems have long Mizar proofs - tens to hundreds of lines - and can contain nontrivial mathematical ideas. See the listings at these web pages.[13]

- ▷ *ENVIRON*: Problems that include all axioms contained in article's environment (that is: articles imported by the current article). This use-case models Mizar authors who selected a particular combination of mathematical areas (previous articles) to base their articles on, and thus limited the Mizar knowledge to a smaller subset of MML. Inside this MML subset they however do not provide any additional guidance to the ATPs. Such problems can already be very large: their average size is 5830 formulas.

- ▷ *ALL*: Problems that include all of the Mizar knowledge available in the MML at the time of proving a particular theorem. This models users who do not want to limit their search to the articles imported in their environment, and provide no guidance to ATPs. The price for such intellectual laziness is obviously a large number of axioms in such ATP problems, the average size of a problem in this class is 40898 formulas.

All these three use-cases are interesting and relevant. As mentioned above, the *SMALL* case is used a lot in ITPs like HOL (Light) as a general method for solving a goal once the user feels that it is sufficiently simply derivable from other established premises. The Mizar system actually also works in a similar way (using a custom weak theorem prover for the "by" inference), however, the emphasis there is not on strength, but on capturing the notion of *obvious inference* [Dav81, Rud87]. Another advantage of the *SMALL* case is that the 218 average formulas (which means much less in a significant number of cases) can be reasonably attacked by existing standard resolution and tableaux techniques, and ATPs based on them, without introducing any novel techniques for dealing with a large number of axioms. Thus, for metasystems

[12] `http://www.cl.cam.ac.uk/~jrh13/hol-light/HTML/MESON_TAC.html`. Actually, MPTP does here more than MESON: it adds a lot of "background" formulas to the problem including knowledge used implicitly by Mizar (reflexivity of $\leq$, etc.).

[13] `http://mmlquery.mizar.org/mmlquery/fillin.php?filledfilename=mml-facts.mqt&argument=number+102`, and `http://www.cs.ru.nl/~freek/100/`

that combine custom axiom-selection methods with standard ATPs, the *SMALL* case can be thought of as a benchmark for the ATP component of such metasystems, i.e., telling how good the performance of the whole metasystem could be, if the axiom-selection component of the metasystem was perfect.

This is no longer true for the *ENVIRON* and *ALL* classes. As will be seen in Section 4.3, using standard ATP techniques on these problems is currently not productive, and axiom-preselection methods are necessary on the *ENVIRON* and *ALL* classes to make use of ATPs.

There are other possible classes of data and divisions of the *SMALL*, *ENVIRON* and *ALL* classes along various axes. A common objection to these three classes is that they are too hard: for example, the data for testing Isabelle/Sledgehammer come typically from goals that are easier than the "full Isabelle theorems". The answer is that using Mizar *simple justifications* ("by" steps – steps provable using the Mizar built-in limited checker [Wie00]) has with the development of MPTP and ATP methods over Mizar become too easy, and such data are no longer suitable as a Mizar/MPTP/ATP benchmark. The success rate of various combined ATP/AI methods on large pieces of "by" data is now around 99.9%, actually allowing for using such methods together with the GDV [Sut06] ATP-based verifier (enhanced to handle TPTP proofs with Jaskowski-like assumptions) to completely ATP-cross-verify large pieces of MML (see [US08] for details). Other classes of problems that could come to mind are:

- ▷ Internal Mizar sublemmas that serve to prove another theorem/lemma, but are not themselves "too easy" (i.e., are not proved by simple justification). Such sublemmas could be considered an easier dataset than theorems, but harder than the simple justifications.

- ▷ De-lemmatized theorems. This would be a dataset created from *SMALL*, where the references (other theorems) used to prove a theorem are (recursively, to some level of recursion) replaced by their own references, in the extreme case expanding them all the way to axioms and definitions. Such de-lemmatized theorems could be considered a harder dataset than the standard theorems.

The reason why it seems unnecessary to test also on such classes of data is that already the theorem dataset provides a variety of both easy and hard data. There are a number of Mizar theorems proved using a simple justification, and on the other hand, there are theorems (like ROLLE:1 in [KRS90] - Rolle's theorem) that take more than four hundred lines and a large number of references to prove, i.e., the amount of lemmatization

varies greatly across various Mizar articles and with various Mizar authors.

The ATP success rates reported in Section 4.3 on the *SMALL* theorem dataset indicate that also from the practical "benchmark" point of having data that are not too easy and not always very hard, this dataset seem to work well with current off-the-shelf ATPs. Again, this is not yet true with the large *ENVIRON* and *ALL* datasets which, on the other hand, can be considered to be hard benchmarks for ATP/AI metasystems that complement standard ATP with systems for axiom selection.

Divisions along various further axes of these datasets are certainly possible. In section 4.4 we attempt to define a "reasonable" crude mathematical categorization of 80% of MML articles, and provide an initial evaluation across this division.

4.3 Experiments

4.3.1 Overall Evaluation on *SMALL* problems

The large-scale experimental evaluation of the standard ATPs focuses on the *SMALL* class of problems (for the reasons mentioned above in Section 4.2). The three main evaluated ATPs are the latest versions of the SPASS [WDF+09] (version 3.7) system, the E prover [Sch02] (version 1.1-004 Balasun), and the Vampire [RV02] prover (version 0.6 - preliminary version for CASC-J5). SPASS and E are evaluated on the server of the Foundations group at Radboud University Nijmegen (RU), which is eight-core Intel Xeon E5520 2.27GHz with 8GB RAM and 8MB CPU cache. The time limit for the evaluations is 30s,[14] and the memory limit is 900MB for each problem. Vampire is evaluated on computers at the laboratory of the University of Manchester (UM), each of them being Intel Core2 Duo E7300 2.66GHz PC with 1G RAM and 3MB cache. The time limits used are again 30s. The relative performances of the two hardware platforms have been compared by evaluation on common ATP problems. The UM platform turns out to be approximately 10% faster.[15] No parallelization is used, each problem is always run serially. The Table 4.2 shows the results. Note that there are some counter-satisfiable problems (very likely arithmetical) however their number is insignificant. It turns out that E, SPASS, and Vampire can in 30s together (that is: if run in parallel) solve 44% of the MML *SMALL* problems.

[14] The experience from previous experiments with E and SPASS is that only a small fraction of problems is solved after 30s. This is obviously different with strategy-scheduling ATP systems like Vampire.

[15] This difference obviously does not translate to 10% more solved problems, however particularly with strategy-scheduling ATP like Vampire, it is quite significant.

description	proved	countersatisfiable	timeout or memory out	total
E 1.1-004	16191	4	35229	51424
SPASS 3.7	17550	12	33862	51424
Vampire 0.6	20109	0	31315	51424
together	22607	12	28817	51424

Table 4.2: Evaluation of E, SPASS, and Vampire on all *SMALL* problems in 30s

description	proved	countersatisfiable	timeout or memory out	total
SPASS 3.7-SOS	292	55	653	1000
SPASS 3.7	345	0	655	1000
together	377	0	623	1000

Table 4.3: Comparison of SPASS-SOS and SPASS on 1000 *SMALL* problems in 30s

In Table 4.3, the results of SPASS used in (the incomplete) SOS mode are shown, and compared to the results of standard SPASS. This is done on randomly selected 1000 *SMALL* problems. The number of countersatisfiable results is not relevant for SPASS-SOS however: it is incorrect when SPASS is used in SOS mode together with ordering-based ATP techniques. SPASS-SOS turns out to be significantly worse than SPASS on the same data (proving 345 of these 1000 problems), however the SOS strategy is reasonably complementary to the standard one: together, the both methods of running SPASS solve 377 problems from this dataset (Vampire solves 39% of these problems).

4.3.2 Overall Evaluation on *ENVIRON* and *ALL* Problems, Sine

To a smaller extent (the above mentioned dataset of 1000 problems) we also evaluate E and Vampire on the large *ENVIRON* and *ALL* problems, and focus on evaluation of a new heuristic axiom pre-selector Sine.[16]

The Sine selection algorithm uses a syntactic approach based on symbol presence in formulas of the problem. Sine builds a $trigger$ relation between symbols and axioms. Presence of a pair (s,A) in this relation represents the fact that the axiom A is (likely to be) needed for reasoning with symbol s. One may say that axiom A gives symbol s "its meaning." In order to construct the $trigger$ relation, for each symbol the number of axioms in which it appears is computed, this number is called the *commonness* of the symbol. Then each axiom is put into the $trigger$ relation with the least general symbol

[16]SUMO Inference Engine - originally developed by the second author for reasoning in the large SUMO knowledge base, described in more detail in the chapter 3 of this thesis.

problems	Vampire+Sine	Vampire+Sine(-d1)	E	E+Sine	E+Sine(-d1)
ENVIRON	181	205	65	135	161
ALL	84	141	21	64	153

Table 4.4: Evaluation of Vampire and E with Sine(-d1) on random 1000 *ENVIRON* and *ALL* MML.1011 problems in 30s

it contains.[17] After the relation is built, the actual axiom selection starts. All problem-specific formulas are selected, and in each iteration the selection is extended by all included formulas that are triggered by any of the symbols used in already selected formulas.[18] The iterating is done until the set of selected axioms becomes stable. The stable set of formulas is then passed to a theorem prover.

This standard fixpoint algorithm however tends to give too many axioms on MML problems.[19] To deal with this, we have introduced a depth limit parameter — a limit on the number of selection-extending iterations. With the depth limit equal to one ("-d1" parameter), for example, only the included axioms immediately related to symbols in the problem-specific axioms are included.

The Table 4.4 presents the results of evaluation of E and Vampire on 1000 *ENVIRON* and *ALL* problems run with 30s time limit on the UM PCs.[20] Note that the combination of Vampire and Sine run with -d1 solves 205 of the *ENVIRON* problems, and the combination of E and Sine with -d1 solves 153 of the *ALL* problems. This is a very good performance on problems with average size of 5830 formulas resp. 40898 formulas. E in this mode solves about half of the *SMALL* versions of the problems. This is very likely also due to improved E heuristics for dealing with large problems. The Sine preprocessing time needs to be added to the 30s given to E prover. For the *ENVIRON* problems this time is on average 1s, and for the *ALL* problems, this is on average 4s.

4.3.3 Evaluation of Strategy Selection and Combination

Design, selection and combination of sufficiently orthogonal useful ATP strategies has been for some time a well known technique significantly raising performance of ATPs.

[17] If there are more symbols with lowest generality index, axiom is put in relation with all of them.

[18] Note that the tested implementation relies on reasonable presentation of large-theory problems using TPTP-includes. This can be easily changed if needed.

[19] The structure of MML significantly differs from KBs like SUMO and CYC. There are many more nontrivial theorems in MML, while SUMO and CYC contain a lot of definitions.

[20] Vampire strategies use Sine automatically, thus we do not provide data for Vampire without Sine.

Both Vampire and E use strategy selection and machine learning on the TPTP library to select a collection of useful strategies. However, they use the found strategies in different ways. Vampire selects sequences of strategies, while E selects one "best" strategy when run in auto-mode. The effect of strategy combination in Vampire can be estimated by comparing Vampire's performance in shorter and longer times (here in 5s and 30s on a random set of 1000 *SMALL* problems). For E and SPASS (running a single strategy depending on the problem) this difference is relatively small. Only 41 problems out of 345 solved in 30s by SPASS are solved in time longer than 5s, which is approximately 13% increase. For Vampire, this improvement is much more significant: out of 384 problems solved in 30s, only 310 are solved in 5s, which gives a 24% increase. This significant difference in comparison to SPASS is due to Vampire running not only for a longer time, but also switching to a different strategy during proof-search.

This clue leads to a strategy-evaluation experiment done with E again on this random set of 1000 *SMALL* problems: Each of the 196 E strategies predefined by the E developer Stephan Schulz is tested on this set of problems with a 5s time limit. It turns out that the strategy selected by E as potentially the best solves 310 problems with the 30 seconds time limit, while the strategy that turns out to be the best in reality solves 317 problems with a 5s time limit.

This clearly demonstrates the potential of domain-based ATP strategy-tuning. All the 196 E strategies together solve 386 of the 1000 problems. This confirms a previous conjecture by the first author and the E developer that E with a suitably-tuned strategy combination mode will be considerably stronger. It also demonstrates that strategy combination is more robust on new problems. The strategies of E are defined using smaller building blocks and a special ("programming") language using a number of parameters. Given the performance potential gained by this strategy-tuning, it would be very interesting (and feasible) AI experiment to try to invent new E strategies by AI methods for (e.g., genetic) parameter-optimization/programming. Particularly on a large knowledge base like MML, this might lead to some surprising ATP-strategy inventions, in the same way as combining ATPs with learning on MML sometimes finds completely novel and significantly shorter proofs than those written by the Mizar authors.

4.4 Evaluation of ATPs on different mathematical domains in MML

Formal mathematics can be clustered according to many aspects, and just thinking about organizing mathematics can lead to all kinds of theoretical investigations in Foundations, Category Theory, but also into practical investigations with various classification schemes like Mathematics Subject Classification 2000[21], and also into very pragmatic classifications based on the shape of the formal theories, important for performing automated reasoning in various domains[22]

For the purpose of ATP evaluation in this paper we attempt a manual division of the MML articles into (currently) seventeen subdomains of main MML developments. This is motivated by the curiosity to verify experimentally various intuitions developed over the years in the ATP field, like "algebra is ATP-easier than calculus". As mentioned above, there can be many approaches to this, and for example a proper MathSC2000 classification would certainly serve even better than the coarse-grained division started by us, however detailed classification of more than 1000 formal articles requires a non-trivial amount of work, and making fine-grained decisions would be beyond our resources. The (evolving) division can be viewed at our web page[23]. The categories and the numbers of articles in each category are shown in Table 4.5. The categorization now includes 804 articles out of the total 1011.

To the extent to which MML is approximation of "real mathematics", and to the extent to which this rough categorization is valid, these seventeen large classes of problems (with the three different problem sizes coming from the different use-cases described in Section 4.2) express a large mathematically-oriented (and particularly Mizar/MPTP-oriented) ATP benchmark.

The table indeed seems to confirm quite convincingly the "algebra is ATP-easier than calculus" theory. The set-theoretical domain outperforms even the algebraic and is the most ATP-friendly. This is quite likely because of two related factors:

- ▷ Set theoretical articles belong to the more basic ones, not much previous implicit knowledge is included in the articles and their size is thus likely smaller, making them easier for ATPs.

[21] See `http://wiki.mizar.org/twiki/bin/view/Mizar/MathematicsSubjectClassification` for a so far 25%-successful attempt to classify MML according to MathSC2000.

[22] For example, the strategy-selection tuning typically works by defining suitable clustering based on the term and formula structure of the problems.

[23] `http://github.com/JUrban/MPTP2/raw/master/MMLdivision.1011`

description	articles	probs	S	E	V	All	S %	E %	V %	All %
Algebra	50	2798	1182	1086	1314	1481	42.24	38.81	46.96	52.93
Algebraic Topology	5	215	52	50	89	94	24.19	23.26	41.4	43.72
Arithmetic, Number theory	70	4095	1587	1515	1741	1943	38.75	37	42.52	47.45
Calculus (real, complex)	54	3255	585	538	651	783	17.97	16.53	20	24.06
Category theory	21	1023	298	305	406	455	29.13	29.81	39.69	44.48
Computers, Algorithms	81	3809	971	932	1120	1304	25.49	24.47	29.4	34.23
Functional analysis	30	1320	395	330	445	507	29.92	25	33.71	38.41
General Topology	65	3191	1199	1115	1441	1594	37.57	34.94	45.16	49.95
Geometry	36	1593	666	659	806	876	41.81	41.37	50.6	54.99
Graph theory,Finite structs	43	2756	1186	1094	1331	1455	43.03	39.7	48.29	52.79
Lattices	50	2434	707	570	764	917	29.05	23.42	31.39	37.67
Linear Algebra	32	1752	496	493	630	700	28.31	28.14	35.96	39.95
Logic, Model theory	52	2832	1042	1084	1196	1369	36.79	38.28	42.23	48.34
Probability and Measure	23	1123	348	274	448	489	30.99	24.4	39.89	43.54
Real plane,Euclidean spaces	84	4555	1018	897	1290	1439	22.35	19.69	28.32	31.59
Set Theory	74	4060	2412	2278	2570	2735	59.41	56.11	63.3	67.36
Universal Algebra	34	1093	391	372	434	502	35.77	34.03	39.71	45.93
together	804	41904	14535	13592	16676	18643	34.69	32.44	39.8	44.49

Table 4.5: Categorization of MML 1011, 804 articles covered, SPASS, E, Vampire, and overall success rates on the categories.

- ▷ As the needed implicit knowledge (encoding e.g. the type system) gets more involved in more advanced areas like calculus, the ATP-emulation of these Mizar type mechanisms becomes more costly, and the ATP performance suffers.

There are a number of ways that these data can be further analyzed, providing useful feedback to the MPTP algorithms and also to ATP (meta) systems.

4.5 Conclusions, Future Work

The most important message of this evaluation is that a combination of three recent ATP systems can solve 44% of the MML theorems coming from many different parts of mathematics, regardless of how much computer algebra is done in them. The leading Vampire ATP system alone can solve 39%. Another important message is that off-the-shelf ATPs are still weak in large theories, however fast axiom-selection heuristics like Sine can help them and improve this state very significantly. Other one-problem-at-atime large-theory heuristic methods similar to Sine are used for axiom pruning in Isabelle/Sledgehammer [MP09], and also for axiom ordering in the SRASS system [SP07]. If these other methods could be run easily on arbitrary and large TPTP problems, it would be interesting to evaluate and compare them on our benchmarks, or at least in the CASC LTB competition which includes a small selection of the older

MPTP problems in its MZR category. The problem of axiom selection in large theories and the possible gains from good solutions seem to be sufficiently important to warrant such benchmarking and further research in this field, possibly leading also to smarter clause-selection algorithms implemented directly inside ATPs.

In this evaluation, axiom-selection methods that use transfer of knowledge between problems (machine learning) like MaLARea [USPV08] are not considered. Methods using learning are very interesting and quite novel in the ATP context, and we are currently investigating various suitable methods for machine learning , characterizations of problems and proofs, and also suitable combinations with strategy-selection and with other axiom-selection methods working in the one-problem-at-a-time setting like Sine. We also plan to do a thorough strategy evaluation of a much larger set of strategies available with Vampire, and their Mizar/MPTP-oriented tuning similar to the current CASC-tuning of Vampire, possibly again with adding machine learning technology.

An important future work is translation of ATP proofs to a presentable ITP format. The (technically certainly admirable) solution used by Isabelle/Sledgehammer (and obviously also of HOL Light) is inclusion of a reasonably strong ATP system directly into their cores. This is however against the philosophy of readable proofs and "obvious inferences" of Mizar, and with strong external ATPs also potentially causing all kinds of other problems: not all proofs can be internalized, and the hard internalized proofs make the library refactoring slow and fragile.[24] With the growing strength of ATPs, a proper human-readable presentation of ATP proofs is a more and more pressing (and also very interesting) AI task. Some previous work in this direction has been done in the context of the Omega and ILF systems. A recent initial effort in this direction is described in [VSU10].

[24]Note that this philosophy is no longer just Mizar's: the Math Components project targeted at the large formalization of Feit-Thompson theorem in Coq is avoiding Coq mechanisms that keep "too much automation" inside proofs for very similar reasons as Mizar does.

Chapter 5

Playing in the Grey Area of Proofs

Authors: Krystof Hoder, Laura Kovacs, Andrei Voronkov

Interpolation is an important technique in verification and static analysis of programs. In particular, interpolants extracted from proofs of various properties are used in invariant generation and bounded model checking. A number of recent papers studies interpolation in various theories and also extraction of smaller interpolants from proofs. In particular, there are several algorithms for extracting of interpolants from so-called local proofs. The main contribution of this paper is a technique of minimising interpolants based on transformations of what we call the "grey area" of local proofs. Another contribution is a technique of transforming, under certain common conditions, arbitrary proofs into local ones.

Unlike many other interpolation techniques, our technique is very general and applies to *arbitrary* theories. Our approach is implemented in the theorem prover Vampire and evaluated on a large number of benchmarks coming from first-order theorem proving and bounded model checking using logic with equality, uninterpreted functions and linear integer arithmetic. Our experiments demonstrate the power of the new techniques: for example, it is not unusual that our proof transformation gives more than a tenfold reduction in the size of interpolants.

5.1 Introduction

Interpolants extracted from proofs have several applications in verification and static analysis, see e.g. [HJMM04, BHT06, JM07a, McM08, CGM+11]. Although interpolants are guaranteed to exist in some theories (for example, those having quantifier elimination), interpolants extracted from proofs turn out to be smaller and more useful

than those obtained by general interpolation algorithms, see, e.g. [JM07b]. For this reason, recent papers [JM06, McM08, KV09b, DKPW10, KLR10, BKRW11, GLS11] consider the problem of obtaining small interpolants for various theories.

In this paper we consider two related problems: extracting interpolants from proofs and minimising such interpolants. Papers [McM05, KV09b] define algorithms for extracting interpolants from so-called *local proofs*. Roughly, in local proofs some symbols are colored in the red or blue colors and others are uncolored. Uncolored symbols are said to be grey. A local proof cannot contain an inference that uses both red and blue symbols. In other words, colors cannot be mixed within the same inference.

However, building local proofs may require substantial changes to a first-order theorem prover or an SMT solver. In addition, local proofs do not necessarily exist. One of the contributions of this paper is a technique for changing proofs into local ones under some conditions. The ideas of this technique can be traced to an observation made in [KMZ06, KV09b] that existential quantification of constants results in an interpolant. We prove a simple result showing that this technique is correct and can be applied to translate non-local proofs with colored constants into local proofs.

When we already have a local proof, one can extract an interpolant from it. This interpolant is a boolean combination of (some) formulas occurring in the proof, if one uses the algorithm of [KV09b]. More exactly, the interpolant is obtained as a boolean combination of conclusions of some *symbol-eliminating inferences*: an inference having at least one colored premise and a grey conclusion. The interpolation extraction theorem of [KV09b] is not restricted to any particular theory. Essentially the only condition on proofs is *inference soundness*, that is, the conclusion of any inference is a logical consequence of its premises. This generality gives one a lot of freedom since one does not have to follow rules of any specific calculus (such as resolution and superposition) in building local proofs.

In this paper, we exploit the generality of [KV09b] by considering proof transformations that preserve both inference soundness and locality. It is interesting that such transformations can drastically change the shape and the size of the extracted interpolant. The transformations we consider are always applied to grey formulas in the proof, which inspired the title of this paper.

While the class transformations we consider (cutting off a grey formula) obviously preserve inference soundness, they can violate locality. To preserve locality, we create a SAT problem whose solutions encode all local proofs obtained by a sequence of cut-offs. Further, we create a linear expression over the variables of the SAT problem that

expresses some numeric characteristics of the interpolant, for example, the number of different atoms in it. Thus, we are interested in the solutions of the problem that minimise the linear expression: any such solution can be used to build a proof giving a smaller (in some sense) interpolant. These solutions can be found using an SMT solver or a pseudo-boolean optimisation tool.

The main contributions of our paper are summarised below.

- ▷ We present a new method of producing smaller interpolants from local proofs. The methods is based on transformation of the "grey area" of proofs. It uses the idea that proof locality can be expressed by a set of propositional formulas whose models represent all local proofs obtained by such transformations (Sections 5.5.1-5.5.2).

- ▷ We present a method for changing proofs into local ones. This method is applicable to all proofs in which all colored symbols are uninterpreted constants (Section 5.4).

- ▷ We define a transformation of interpolant minimisation problems into the problem of solving pseudo-boolean constraints (Section 5.5.4). Minimality is defined with respect to various measures of the size of interpolants.

- ▷ We implemented our minimisation algorithm in the Vampire theorem prover [RV02]. It uses the Yices SMT solver [Dd06] for solving pseudo-boolean constraints (Section 5.6.1). As Vampire cannot yet efficiently handle the combination of various theories, we generate proofs over SMT problems using Z3 [dMB08].

- ▷ We show experimentally that our method improves [KV09b] by generating considerably better/smaller interpolants in the size, the total weight and the number of quantifiers (Section 5.6).

The rest of this paper is structured as follows. Section 5.2 overviews relevant definitions and properties of first-order logic and interpolation. In Section 5.3 the notion of colored and local proofs are introduced. Our result on translating non-local proofs into local ones is formulated in Section 5.4. Section 5.5 details our approach to minimising interpolants. We present experimental results in Section 5.6 and overview related work in Section 5.7. Section 5.8 concludes the paper.

5.2 Interpolation

We consider the standard first-order predicate logic with equality. We allow all standard boolean connectives and quantifiers in the language. We assume that the language contains the logical constants $\top$ for always true and $\bot$ for always false formulas.

Throughout this paper, we denote formulas by A, B, C, D, G, R, terms by r, s, t, variables by x, y, z, constants by a, b, c and function symbols by f, g, possibly with indices. Let A be a formula with free variables $\bar{x}$, then $\forall A$ (respectively, $\exists A$) denotes the formula $(\forall \bar{x})A$ (respectively, $(\exists \bar{x})A$). A formula is called *closed*, or a *sentence*, if it has no free variables. We call a *symbol* a predicate symbol, a function symbol or a constant. Thus, variables are not symbols. We consider equality $=$ part of the language, that is, equality is not a symbol. A formula or a term is called *ground* if it has no occurrences of variables. A formula is called *universal* (respectively, *existential)* if it has the form $(\forall \bar{x})A$ (respectively, $(\exists \bar{x})A$), where A is quantifier-free. We write $C_1, \ldots, C_n \vdash C$ to denote that the formula $C_1 \wedge \ldots \wedge C_n \implies C$ is a tautology. Note that $C_1, \ldots, C_n, C$ may contain free variables.

A *signature* is any finite set of symbols. The *signature of a formula A* is the set of all symbols occurring in this formula. For example, the signature of $b = g(z)$ is $\{g, b\}$. The *language of a formula A*, denoted by $\mathcal{L}_A$, is the set of all formulas built from the symbols occurring in A, that is formulas whose signatures are subsets of the signature of A.

We recall the following theorem from [Cra57].

Theorem 5.2.1 [Craig's Interpolation Theorem]Let A, B be closed formulas and let $A \vdash B$. Then there exists a closed formula $I \in \mathcal{L}_A \cap \mathcal{L}_B$ such that $A \vdash I$ and $I \vdash B$.
In other words, every symbol occurring in I also occurs in both A and B. Every formula I satisfying this theorem will be called an *interpolant* of A and B.

We call a *theory* any set of closed formulas. If T is a theory, we write $C_1, \ldots, C_n \vdash_T C$ to denote that the formula $C_1 \wedge \ldots \wedge C_1 \implies C$ holds in all models of T. In fact, our notion of theory corresponds to the notion of *axiomatisable theory* in logic. When we work with a theory T, we call symbols occurring in T *interpreted* while all other symbols *uninterpreted*.

As proved in [KV09b], Craig's interpolation also holds for theories in the following sense:

Theorem 5.2.2 Let A, B be formulas and let $A \vdash_T B$. Then there exists a formula I such that

1. $A \vdash_T I$ and $I \vdash B$;
2. every uninterpreted symbol of I occurs both in A and B;
3. every interpreted symbol of I occurs in B.

Likewise, there exists a formula I such that

1. $A \vdash I$ and $I \vdash_T B$;
2. every uninterpreted symbol of I occurs both in A and B;
3. every interpreted symbol of I occurs in A.

The proof of this theorem in [KV09b] uses compactness, which is guaranteed when T is axiomatisable.

In the sequel we will sometimes be interested in the interpolation property with respect to a given theory T. We will use $\vdash_T$ instead of $\vdash$ and relativise all definitions to T. To be precise, we call an *interpolant* of A and B any formula I such that $A \vdash_T I$, $I \vdash_T B$, and every uninterpreted symbol of I occurs both in A and B.

If E is a set of expressions (for example, formulas) and constants $c_1, \ldots, c_n$ do not occur in E, then we say that $c_1, \ldots, c_n$ are *fresh* for E. We will less formally simply say *fresh constants* when E is the set of all expressions considered in the current context.

We call a *reverse interpolant* of A and B any formula I such that $A \vdash_T I, I, B \vdash_T \bot$, and every uninterpreted symbol of I occurs both in A and B.

Reverse interpolants for A and B are exactly interpolants of A and $\neg B$. Moreover, when B is closed, reverse interpolants are exactly interpolants in the sense of [McM05, McM08]. Reverse interpolants are convenient when we use a refutation-based inference system, such as resolution, for finding a proof of $A \implies B$ that can give us an interpolant: in this case one can search for a refutation from the set of formulas $A, \neg B$ instead.

5.3 Local Proofs

In this section we recall some terminology related to inference systems. Inference systems are commonly used in the theory of resolution and superposition [BG01, NR01]; however we do not restrict ourselves to the superposition calculus. The material of this section is based on [KV09b], adapting the terminology of [KV09b] to our setting.

We also introduce the notion of *local proofs* and recall results on extracting interpolants from local proofs as proved in [KV09b].

Definition 5.3.1 An *inference rule* is an n-ary relation on formulas, where $n \geq 0$. The elements of such a relation are called *inferences* and usually written as

$$\frac{A_1 \quad \dots \quad A_n}{A}.$$

The formulas $A_1, \dots, A_n$ are called the *premises* of this inference, whereas the formula A is the *conclusion* of the inference.

An *inference system* $\mathfrak{I}$ is a set of inference rules. An *axiom* of an inference system is any conclusion of an inference with 0 premises. Any inferences with 0 premises and a conclusion A will be written without the bar line, simply as A.

A *derivation* in an inference system $\mathfrak{I}$ is a tree built from inferences in $\mathfrak{I}$. If the root of this derivation is A, then we say it is a *derivation of* A. A derivation of A is called a *proof* of A if it is finite and all leaves in the derivation are axioms. A formula A is called *provable* in $\mathfrak{I}$ if it has a proof. We say that a derivation of A is *from assumptions* $A_1, \dots, A_m$ if the derivation is finite and every leaf in it is either an axiom or one of the formulas $A_1, \dots, A_m$. A formula A is said to be *derivable from* assumptions $A_1, \dots, A_m$ if there exists a derivation of A from $A_1, \dots, A_m$. A *refutation* is a derivation of $\bot$. □

Note that a proof is a derivation from the empty set of assumptions. Any derivation from a set of assumptions S can be considered as a derivation from any larger set of assumptions $S' \supseteq S$.

Let us now fix two sentences R (red) and B (blue). In the sequel we assume R and B to be fixed and give all definitions relative to R and B. Denote by $\mathcal{L}$ the intersection of the languages of R and B, that is the set $\mathcal{L}_R \cap \mathcal{L}_B$. We call signature symbols occurring both in R and B *grey*, symbols occurring only in R *red* and symbols occurring only in B *blue*. A symbol that is either red or blue is also called *colored*. For a formula C, we say that C is *grey* if $C \in \mathcal{L}$, otherwise we say that C is *colored*. In other words, grey formulas contain only grey symbols and every colored formula contains at least one red or blue symbol. A colored formula that only contains red and grey symbols, is called a *red* formula. Similarly, a *blue* formula is a colored formula containing only blue and grey symbols. In the rest of this paper, red formulas will be denoted by R, blue formulas by B, and grey formulas by G, possibly with indices.

Definition 5.3.2 [RB-derivation]Let us call an *RB-derivation* any derivation Π satisfying the following conditions.

(RB1) For every leaf C of Π one of the following conditions holds:

1. $R \vdash_T \forall C$ and $C \in \mathcal{L}_R$ or

2. $B \vdash_T \forall C$ and $C \in \mathcal{L}_B$.

(RB2) For every inference

$$\frac{C_1 \quad \dots \quad C_n}{C}$$

of Π we have $\forall C_1, \dots, \forall C_n \vdash_T \forall C$.

We will refer to property (RB2) as *soundness*. □

We will be interested in finding reverse interpolants of R and B. The case $\mathcal{L}_R \subseteq \mathcal{L}_B$ is obvious, since in this case R is a reverse interpolant of R and B. Likewise, if $\mathcal{L}_B \subseteq \mathcal{L}_R$, then $\neg B$ is a reverse interpolant of R and B. For this reason, in the sequel we assume that $\mathcal{L}_R \not\subseteq \mathcal{L}_B$ and $\mathcal{L}_B \not\subseteq \mathcal{L}_R$, that is, *both R and B contain at least one colored symbol*.

We are mostly interested in a special kind of derivation introduced in [JM06] and called *local* (or sometimes called *split-proofs*). The definition of a local derivation is relative to formulas R and B.

Definition 5.3.3 [Local RB-derivation]An inference

$$\frac{C_1 \quad \dots \quad C_n}{C}$$

in an RB-derivation is called *local* if the following two conditions hold.

(L1) Either $\{C_1, \dots, C_n, C\} \subseteq \mathcal{L}_R$ or $\{C_1, \dots, C_n, C\} \subseteq \mathcal{L}_B$.

(L2) If all of the formulas $C_1, \dots, C_n$ are grey, then C is grey, too.

A derivation is called *local* if so is every inference of this derivation. □

In other words, (L1) says that inferences cannot mix colors: no inference contains both red and blue symbols. Condition (L2) is natural (inferences should not introduce irrelevant symbols) but it is absent in other works. Condition (L2) is however essential for us since without it the proof of Theorem 5.3.4 does not go through [KV09b]. Note that standard derivations produced by theorem provers often contain inferences violating (L2), especially, in instantiation rules:

$$\frac{(\forall x)A(x)}{A(r)}\ ,$$

where r is a red term. However, such inferences can be removed from derivations without violating (L1).

We will now formulate one of the main theorems of [KV09b] on the extraction of interpolants from local proofs and explain the structure of interpolants obtained by the algorithm of [KV09b].

Consider any RB-derivation Π. Note that by the soundness condition (RB2) we can replace every formula C occurring in this derivation by its universal closure $\forall C$ and obtain an RB-derivation Π' where inferences are only performed on closed formulas. We will call such derivations Π' *closed* and assume, for simplicity, that we are dealing only with closed derivations.

We call a *symbol-eliminating inference* any inference that is

1. either a grey leaf G of Π such that $R \vdash_T G$.

2. or has the form

$$\frac{A_1 \quad \cdots \quad A_n}{G},$$

such that G is grey and and at least one of the formulas $A_1, \ldots, A_n$ is colored.

Any such inference "eliminates" at least one colored symbol. One could also call such inferences *color-eliminating*. The following theorem is proved in [KV09b]:

Theorem 5.3.4 Let Π be a closed local RB-refutation. Then one can extract from Π a reverse interpolant I of R and B. This reverse interpolant is a boolean combination of conclusions of symbol-eliminating inferences of Π. □

The proof of Theorem 5.3.4 in [KV09b] gives an algorithm for extracting an interpolant from a refutation.

By a close inspection of the algorithm of [KV09b], we noted that not all conclusions of symbol-eliminating inferences occur in the extracted interpolant. To characterise the set of all formulas occurring in the interpolant, in this paper we introduce a new notion, called the *digest* of a refutation, as given below.

Definition 5.3.5 [Digest]Consider a closed local RB-refutation Π with a formula G which is a conclusion of a symbol-eliminating inference.

If the inference eliminates a red symbol, then it has the form:

$$\frac{\cdots \quad R_0 \quad \cdots}{G}$$

Consider the path from G to the bottom formula of the refutation:

$$\dfrac{\cdots \quad R_0 \quad \cdots}{\begin{array}{c} G \\ \vdots \\ \bot \end{array}}$$

We say that G belongs to the *digest* of the refutation if either all formulas on the path are grey *or* the first (closest to G) colored formula on the path is blue.

Likewise, for a blue symbol eliminating inference:

$$\dfrac{\cdots \quad B_0 \quad \cdots}{\begin{array}{c} G \\ \vdots \\ \bot \end{array}}$$

G belongs to the *digest* of the refutation if at least one formula on the path is colored *and* the first (closest to G) colored formula on the path is red. □

Note the slight asymmetry in Definition 5.3.5 between red and blue symbol eliminating inferences, which is due to the interpolant generation algorithm of [KV09b]. Using the notion of digest, we can now refine Theorem 5.3.4 as follows:

Theorem 5.3.6 Let Π be a closed local RB-refutation. Then one can extract from Π a reverse interpolant I of R and B. This reverse interpolant is a boolean combination of the formulas in the digest of Π. □

In what follows we will refer to the reverse interpolant obtained from a refutation as described in Theorem 5.3.6 as the *interpolant extracted from* Π.

5.4 Proof Localisation

Extracting interpolants from proof requires a special interpolating prover, or a prover producing local proofs. While, as reported in [HKV10], the theorem prover Vampire can search for local proofs only and hence the algorithm of [KV09b] can be used in first-order resolution proofs, most provers and SMT solvers do not necessarily generate local proofs.

One of the main motivations of this paper was to check how our minimisation technique works on real-life examples taken from static analysis of software. Although such benchmarks exist, they can only be solved using an SMT solver, which in general produces non-local proofs.

In is interesting that in real-life examples, especially those taken from bounded model checking, all the colored symbols are normally uninterpreted constants representing state variables from intermediate states. In this section we show that for such examples one can transform arbitrary proofs into local ones, at the cost of quantifying some formulas in the proof. This idea has already appeared in [KMZ06, KV09b], see Lemma 5.4.1 below.

The downside of this approach is that a ground refutation can become a non-ground one, thus, the extracted interpolant may contain quantifiers. Once we have a local proof, the number of such quantifiers can be reduced using the technique of Section 5.5 (line 18 of Algorithm 5.5.7).

Lemma 5.4.1 [KMZ06, KV09b] Consider two formulas $A_1(a)$ and A_2 such that $A_1(a) \vdash_T A_2$ and a is an uninterpreted constant not occurring in A_2. Then, $A_1(a) \vdash (\exists x)A_1(x)$ and $(\exists x)A_1(x) \vdash A_2$.
This lemma can be used to localise non-local derivations by quantifying away colored constants that result in mixing colors.

Theorem 5.4.2 Given two formulas R and B such that $R \implies B$ and all the colored symbols of R and B are uninterpreted constant symbols. Then any proof Π of $R \implies B$ can be translated into a local proof Π_l.

PROOF. Let us take a non-local refutation Π of $R \implies B$. This means, that Π contains at least one inference that violates conditions (L1)-(L2) of Definition 5.3.3. The proof is by induction on Π. We will eliminate all color conflicts one by one, starting from the bottom of the proof. Thus, for every conflicting inference, we can assume that the derivation below it is already local. In particular, the conclusion of the violating inference cannot mix colors. Consider the case when the conclusion is blue (other cases are similar). Then the violating inference has the form

$$\frac{R_1 \quad \cdots \quad R_n \quad A_1 \quad \cdots \quad A_m}{A},$$

where $A, A_1, \ldots, A_m$ are either grey or blue and $R_1, \ldots, R_n$ are red. Let $r_1, \ldots, r_k$ be all the red constants occurring in this inference and formulas R'_i are obtained from R_i by replacing $r_1, \ldots, r_k$ by fresh variables $x_1, \ldots, x_k$. Note that all of the R'_i are either grey or blue. The above non-local inference can then be replaced by:

$$\frac{(\exists x_1 \ldots x_k)(R'_1 \wedge \ldots \wedge R'_n) \quad A_1 \quad \cdots \quad A_m}{A},$$

$$\dfrac{\dfrac{(\forall x_1)(p(x_1)\vee q(x_1,r_1))\ \neg p(b_1)}{q(b_1,r_1)}\quad\dfrac{(\forall x_2)(s(x_2)\vee\neg q(x_2,r_1))\ \neg s(b_1)}{\neg q(b_1,r_1)}}{\bot}$$

(a)

$$\dfrac{\dfrac{\dfrac{(\forall x_1)(p(x_1)\vee q(x_1,r_1))\ \ (\forall x_2)s(x_2)\vee\neg q(x_2,r_1)}{(\exists y)\big((\forall x_1)(p(x_1)\vee q(x_1,y))\wedge(\forall x_2)(s(x_2)\vee\neg q(x_2,y))\big)}\quad\neg p(b_1)}{(\exists y)\big(q(b_1,y)\wedge(\forall x_2)(s(x_2)\vee\neg q(x_2,y))\big)}\quad\neg s(b_1)}{(\exists y)\big(q(b_1,y)\wedge\neg q(b_1,y)\big)}$$

$$\bot$$

(b)

Figure 5.1: Proof localisation of proof (a) into proof (b).

This inference does not contain the red color, and we are done. Note that the premises of the formula $(\exists x_1\ldots x_k)(R'_1\wedge\ldots\wedge R'_n)$ are given by the union of the premises of $R_1,\ldots,R_n$. The correctness of the transformation is guaranteed by Lemma 5.4.1.

The above transformation can also be applied on inferences where a premise contains both a red and a blue symbol. The non-local inference is replaced by a local inference at the cost of using existential quantifiers over the premise with colored symbols.

□

This theorem gives us an algorithm for changing any non-local refutation to a local one, provided that the condition on colored symbols is satisfied.

Figure 5.1 illustrates how the non-local proof given in Figure 5.1(a) is translated into the local proof listed in Figure 5.1(b).

5.5 Playing in the Grey Area

This section presents the main idea of this paper. It is based on the following observation. One can change, sometimes considerably, the grey areas (that is, areas consisting of grey formulas) of the proof without violating locality. In addition, such proof transformations can change the extracted interpolant.

We will only consider one kind of proof transformations, called here *grey slicing*. Other proof transformations can be proposed as well, but are beyond the scope of this

paper.

Definition 5.5.1 [Grey slicing]Consider any derivation Π containing a subderivation Δ of the form

$$\dfrac{A_1 \quad \cdots \quad A_n \qquad \dfrac{A_{n+1} \quad \cdots \quad A_m}{A}}{A_0}\,, \tag{5.1}$$

where $n \geq 0$.

We say that a derivation Π' is obtained from Π by *slicing off A in Δ* (or simply, *slicing off A*) if Π' is obtained from Π by replacing the subderivation Δ by

$$\dfrac{A_1 \quad \cdots \quad A_n \quad A_{n+1} \quad \cdots \quad A_m}{A_0} \tag{5.2}$$

When A is a grey formula, we will refer to this transformation as *grey slicing*. □

Apparently, grey slicing preserves properties (RB1)-(RB2) of Definition 5.3.2, so it transforms an RB-derivation into an RB-derivation. It is also easy to see that grey slicing can violate the locality conditions (L1) of Definition 5.3.3. For example, slicing off G_1 in

$$\dfrac{B_0 \quad \dfrac{R_0}{G_1}}{G_0}$$

yields a non-local derivation

$$\dfrac{B_0 \quad R_0}{G_0}\,.$$

Consider now an example showing that grey slicing transformations can change the digest, and hence the extracted interpolant.

Example 5.5.2 Take the following refutation Π:

$$\dfrac{\dfrac{\dfrac{R_1 \quad G_1}{G_3} \qquad \dfrac{B_1 \quad G_2}{G_4}}{G_5}}{\dfrac{R_3 \qquad \dfrac{}{G_6}}{\dfrac{R_4}{G_7}}}$$
$$\bot$$

The digest of this refutation is $\{G_4, G_7\}$ and the extracted reverse interpolant is $G_4 \Longrightarrow G_7$. Slicing off G_4 in Π results in the refutation Π_1:

$$\dfrac{\dfrac{R_3 \quad \dfrac{\dfrac{R_1 \quad G_1}{G_3} \quad \dfrac{B_1 \quad G_2}{G_5}}{G_6}}{R_4}}{\dfrac{G_7}{\bot}}$$

with the digest $\{G_5, G_7\}$ and the extracted reverse interpolant $G_5 \Longrightarrow G_7$. Slicing off now G_5 in Π_1 results in Π_2:

$$\dfrac{\dfrac{R_3 \quad \dfrac{\dfrac{R_1 \quad G_1}{G_3} \quad \dfrac{B_1 \quad G_2}{G_6}}{R_4}}{\dfrac{G_7}{\bot}}$$

with the digest $\{G_6, G_7\}$ and the extracted reverse interpolant $G_6 \Longrightarrow G_7$. We can slice off G_7 in Π_2 and obtain the refutation:

$$\dfrac{\dfrac{R_3 \quad \dfrac{\dfrac{R_1 \quad G_1}{G_3} \quad \dfrac{B_1 \quad G_2}{G_6}}{R_4}}{\bot}$$

with the digest $\{G_6\}$, and the reverse interpolant $\neg G_6$.

However, if we slice off G_3 in the original derivation Π, we obtain the refutation:

$$\dfrac{\dfrac{R_3 \quad \dfrac{\dfrac{R_1 \quad G_1}{G_5} \quad \dfrac{B_1 \quad G_2}{G_4}}{G_6}}{R_4}}{\dfrac{G_7}{\bot}}$$

in which slicing off G_4 would violate the locality of the resulting refutation.

Example 5.5.2 gives us the following observations:

1. grey slicing can change the extracted interpolant, and sometimes considerably (compare $G_4 \implies G_7$ and $\neg G_6$).

2. a grey slicing step can prevent other grey slicing steps, thus preventing previously possible interpolants.

The main question we are going to answer in this section is how to use grey slicing to obtain smaller, and even minimal, in some sense, interpolants. To this end we will use the following ideas. First, we will introduce a set V_Π of propositional variables expressing some properties of refutations obtained by grey slicing from a given proof Π. Next, we will define propositional formulas P_Π of the variables V_Π that express locality. Thus, every refutation obtained from Π by grey slicing is local if and only if it satisfies P_Π. This means we can use a SAT solver to "compute" all local refutations that can be obtained from Π by grey slicing. Finally, we introduce propositional formulas expressing the digest of refutations. This set of propositional formulas allows us to use an SMT solver or a pseudo-boolean optimisation tool to find refutations minimising the digest in various ways.

Let us now formalise this idea. In the rest of this section, when we speak about a formula from a derivation, we will normally mean a concrete node in the derivation containing this formula (note that a tree-like derivation may contain more than one node with the same formula). Later we will also discuss derivations in the dag form. Nonetheless, for simplicity, for the moment we prefer to deal with trees instead of dags.

The first thing to note is that every derivation is also a set of nodes occurring in it and slicing off simply removes one node from this set. This means that a sequence of slicing off transformations removes a subset of nodes. Every removed node G, at the point of removal, is replaced by a set of other nodes occurring in the derivation (namely, the premises of G at that point). Each of the nodes in this set can in turn be removed (and replaced by other nodes) etc., so eventually the place of any removed node will be taken by a set of nodes occurring in the final derivation. We will call this set a *trace* of F and define it formally below.

Definition 5.5.3 [Trace]Let $S = \Pi_0, \ldots, \Pi_k$ be a sequence of derivations such that each member in the sequence except Π_0 is obtained by slicing off a single grey node from the previous one. For every grey node G in Π_0 we define a set of formulas, call *trace of* G (with respect to S), as follows:

1. If G was never sliced off, that is, it occurs in Π_k, then $trace(G) \stackrel{\text{def}}{=} \{G\}$.

2. Suppose G was sliced off at some point, that is, G is the formula A as in Definition 5.5.1. Then $trace(G) \stackrel{\text{def}}{=} trace(A_{n+1}) \cup \ldots \cup trace(A_m)$. □

Denote any sequence S of slicing off transformations with the initial derivation Π and final derivation Π' by $\Pi \dashrightarrow \Pi'$. It is not hard to argue that the following lemma holds.

Lemma 5.5.4 The trace of a node does not depend on the sequence of transformations S but only depends on the initial and the final derivation in S. That is, for every two derivations of the form $\Pi \dashrightarrow \Pi'$ with the same initial derivation Π and final derivation Π', and for every grey node G in Π, the trace of G is the same in both derivations. □

In the rest of this section we will normally assume a fixed initial derivation Π and various sequences $\Pi \dashrightarrow \Pi'$. In view of this lemma we will simply speak about the *trace of G in Π'*.

Suppose $\Pi \dashrightarrow \Pi'$ is a sequence of transformations. Let us introduce some propositions characterising the behaviour of grey nodes in Π on this sequence.

- ▷ $s(G)$: G was sliced off;
- ▷ $r(G)$: the trace of G contains a red formula;
- ▷ $b(G)$: the trace of G contains a blue formula;
- ▷ $g(G)$: the trace of G contains only grey formulas;
- ▷ $d(G)$: G belongs to the digest of Π'.

We define the set V_Π of propositional variables as consisting of all the variables $s(G)$, $r(G)$, $b(G)$, $g(G)$, $d(G)$ denoting these propositions. Later we will add to V_Π more variables.

Then, for every sequence of transformations $\Pi \dashrightarrow \Pi'$ and every grey node G in Π, each of the above propositions is either true or false on this sequence. Therefore, if we take any propositional formula built from these propositions, it is also either true or false on this sequence.

5.5.1 Expressing the Digest

Our next aims are to write down a propositional formula that expresses that Π' is local, and also represent the digest of any local refutation. To this end we will first introduce propositional variables and formulas over grey nodes, then write down further formulas of these propositions that are satisfied when Π' is local, and finally show that satisfiability of these propositions implies locality of Π'.

Propositions *rc* and *bc*. Take a local derivation Π with $\Pi \dashrightarrow \Pi'$. For each grey node G in Π we first introduce the *propositions* $rc(G)$ *and* $bc(G)$ expressing that G is not sliced off and is a conclusion of a symbol-eliminating inference in Π with at least one red (respectively, blue) premise. The propositional variables $rc(G)$ and $bc(G)$ are added to V_Π.

We will only define $rc(G)$, since the case of bc is symmetric.

Consider the following cases depending on the inference introducing G in Π.

1. G is introduced by an inference with only grey premises:

$$\frac{G_1 \quad \cdots \quad G_m}{G},$$

   We then write:

$$rc(G) \leftrightarrow (\neg s(G) \wedge (r(G_1) \vee \ldots \vee r(G_m))). \tag{5.3}$$

   The conditions on the traces of $G_1, \ldots, G_m$ ensure that G can be written as the conclusion of a symbol eliminating inference with at least one red premise. Namely, if $r(G_i)$ holds, then by slicing off G_i and some of the grey nodes from its derivation, G becomes the conclusion of a symbol eliminating inference with at least one red premise.

2. G is introduced by an inference with at least one red premise:

$$\frac{R_1 \quad \cdots \quad R_n \quad G_1 \quad \cdots \quad G_m}{G}.$$

   We then have:

$$rc(G) \leftrightarrow \neg s(G). \tag{5.4}$$

3. G is introduced by an inference with at least one blue premise

$$\frac{B_1 \quad \cdots \quad B_n \quad G_1 \quad \cdots \quad G_m}{G}\ .$$

Due to the locality of derivations, we write:

$$\neg rc(G). \tag{5.5}$$

Equations (5.3)-(5.5) are added to the set of propositional formulas P_Π over V_Π.

Propositions *rf* and *bf*. We introduce the propositions $rf(G)$ and $bf(G)$ for every grey node G, and add the corresponding variables to V_Π. These propositions are closely related to the definition of digest. The proposition $rf(G)$ holds iff on the path from G to the root of Π either (i) all nodes are grey, or (ii) the first colored node is a blue one. Likewise, the proposition $bf(G)$ expresses that on the path from G to the root of Π, there exists a colored node and the first colored node is a red one.

We will only write down properties of *rf*, the case of *bf* is similar. We define *rf* "inductively", starting from the root (the bottom formula) of the derivation Π.

1. If the successor of G in Π is a red formula, then we write

$$\neg rf(G). \tag{5.6}$$

2. If the successor of G in Π is a blue formula, then we write

$$rf(G). \tag{5.7}$$

3. Finally, if the successor of G in Π is a grey node G_1, then we write

$$rf(G) \leftrightarrow (rf(G_1) \vee bc(G_1)) \wedge \neg rc(G_1). \tag{5.8}$$

Equations (5.6)-(5.8) are added to P_Π.

Proposition *d*. By straightforward inspection of the definition of digest, it is not hard to argue that $d(G)$ can be expressed as follows:

$$d(G) \leftrightarrow (rc(G) \wedge rf(G)) \vee (bc(G) \wedge bf(G)). \tag{5.9}$$

We add (5.9) to P_Π.

5.5.2 Expressing Locality

Take a local derivation Π and a grey node G in it. Depending on the inference introducing G, there are four possible cases:

1. G is a leaf of Π;
2. G is introduced by an inference with grey premises;
3. G is introduced by an inference with at least one red premise;
4. G is introduced by an inference with at least one blue premise. In this case, due to the locality of Π, all premises in the derivation tree of G are either blue or grey.

For each of these cases, we will show how to write down formulas expressing that $\Pi \dashrightarrow \Pi'$ results in a local derivation, that is, Π' is local. Each below listed propositional formulas is added to P_Π.

General properties of grey nodes. Note that, if a node G is not sliced off, then its trace is $\{G\}$, so we have $g(G)$:

$$\neg s(G) \implies g(G). \tag{5.10}$$

We also know that a node which is sliced off cannot belong to the digest:

$$s(G) \implies \neg d(G). \tag{5.11}$$

Observe that equations (5.10)-(5.11) do not make use of the assumptions that Π is local. That is, (5.10)-(5.11) hold for *arbitrary* derivations.

Further, note that for local derivations the properties b, r and g are mutually exclusive. Therefore, for every grey node node G we add the following properties expressing mutual exclusion:

$$\begin{aligned} color(G) \stackrel{\text{def}}{=} \quad & (b(G) \vee r(G) \vee g(G)) \wedge \\ & (b(G) \implies \neg r(G) \wedge \neg g(G)) \wedge \\ & (r(G) \implies \neg b(G) \wedge \neg g(G)) \wedge \\ & (g(G) \implies \neg r(G) \wedge \neg b(G)). \end{aligned} \tag{5.12}$$

G is a leaf. In this case, G cannot be sliced off and we have:

$$leaf(G) \stackrel{\text{def}}{=} \neg s(G) \wedge g(G) \tag{5.13}$$

G is introduced by an inference with grey premises:

$$\frac{G_1 \quad \cdots \quad G_m}{G}\ .$$

The locality of $\Pi \dashrightarrow \Pi'$ implies that if the trace of any $G_1, \ldots, G_m$ contains a red (respectively, blue) formula, then the traces of $G_1, \ldots, G_m$ cannot contain a blue (respectively, red) formula. To further reason about the trace of G, consider the following cases.

(i) If G is never sliced off in $\Pi \dashrightarrow \Pi'$, then the trace of G is clearly grey. Whether G is a conclusion of a symbol eliminating inference only depends on whether the trace of some of the $G_1, \ldots, G_m$ contains either a blue or a red formula.

(ii) If G is sliced off, then the color of the formulas in the trace of G depend on the color of the formulas from the traces of $G_1, \ldots, G_m$.

Based on the above reasoning, we introduce the following formula capturing the properties of the trace of G:

$$\begin{aligned} grey(G) \stackrel{\text{def}}{=} \quad & (r(G_1) \vee \ldots \vee r(G_m) \implies \neg b(G_1) \wedge \ldots \wedge \neg b(G_m)) \wedge \\ & (b(G_1) \vee \ldots \vee b(G_m) \implies \neg r(G_1) \wedge \ldots \wedge \neg r(G_m)) \wedge \\ & (s(G) \wedge (r(G_1) \vee \ldots \vee r(G_m)) \implies r(G)) \wedge \\ & (s(G) \wedge (b(G_1) \vee \ldots \vee b(G_m)) \implies b(G)) \wedge \\ & (s(G) \wedge g(G_1) \wedge \ldots \wedge g(G_m) \implies g(G)) \wedge \\ & (\neg s(G) \implies g(G)). \end{aligned} \tag{5.14}$$

G is introduced by an inference with at least one red premise:

$$\frac{R_1 \quad \cdots \quad R_n \quad G_1 \quad \cdots \quad G_m}{G}\ .$$

In this case the locality of Π implies that the trace of G can contain only red and grey formulas. Moreover, the color of the formulas from the trace of G only depends on whether G is sliced off, as follows.

(i) If G is sliced off, then the trace of G depends on the traces of $R_1,\ldots,R_n$, $G_1,\ldots,G_m$, and hence the trace of G contains at least one red formula. Also note, that if G is sliced off, then G cannot belong to the digest of Π'.

(ii) If G is not sliced off, then $trace(G) = \{G\}$. Hence, the trace of G only contains grey formulas. Moreover, note that G is the conclusion of symbol eliminating inference. Thus, G also belongs to the digest of Π'.

We therefore introduce the below formula for G, capturing the properties of the trace of G:

$$\begin{array}{rcl} red(G) & \stackrel{\text{def}}{=} & \neg b(G_1)\wedge\ldots\wedge\neg b(G_m)\wedge \\ & & (s(G) \implies r(G))\wedge \\ & & (\neg s(G) \implies g(G)). \end{array} \tag{5.15}$$

G is introduced by an inference with at least one blue premise:

$$\frac{B_1 \quad \cdots \quad B_n \quad G_1 \quad \cdots \quad G_m}{G}\ .$$

Similarly to the previous case, we introduce the following formula:

$$\begin{array}{rcl} blue(G) & \stackrel{\text{def}}{=} & \neg r(G_1)\wedge\ldots\wedge\neg r(G_m)\wedge \\ & & (s(G) \implies b(G))\wedge \\ & & (\neg s(G) \implies g(G)). \end{array} \tag{5.16}$$

This completes our construction of the propositional variables and formulas explained in the beginning of this section. Namely, the set of variables V_Π consists of all variable $s(G)$, $r(G)$, $b(G)$, $g(G)$, $rc(G)$, $bc(G)$, $rf(G)$, $bf(G)$ and $d(G)$, and the set P_Π of formulas are all formulas (5.3)–(5.16).

Our construction clearly implies the following result.

Theorem 5.5.5 Let Π be a local derivation. Then a sequence $\Pi \dashrightarrow \Pi'$ satisfies all formulas (5.8)-(5.16) from P_Π if and only if Π' is local. Moreover, the propositions $r(G)$, $b(G)$, $g(G)$, $rc(G)$, $bc(G)$, $rf(G)$, $bf(G)$ and $d(G)$ have their intended meaning, in particular, in every model Π' of these formulas G belongs to the digest of Π' if and only if $d(G)$ holds on Π'.

5.5.3 Derivations as Dags

Refutations found by theorem provers are normally dags. Transforming a dag to a tree can result in an exponential growth in size. Therefore, it is desirable to change our technique to deal with dags. The modification is quite simple: we allow a formula in a dag to be sliced off only if *all* the tree derivations corresponding to the resulting dag are local. Note that this may result in a smaller choice of grey slicing transformations as compared to refutations as trees and hence larger interpolants. Nonetheless, expanding dags to trees may turn to be unfeasible. Therefore, our implementation uses dags.

To build propositional formulas expressing locality on dags, one should only modify the propositions $rf(G)$ and $bf(G)$.

Propositions *rf* and *bf* for dags. The proposition $rf(G)$ holds iff on *all paths* from G to the root of Π either (i) all nodes are grey, or (ii) the first colored node is a blue one. Likewise, the proposition $bf(G)$ expresses that on *all paths* from G to the root of Π, the first colored node is a red one. The axiomatisation of these propositions is given below, and (5.6)-(5.8) are replaced by the below formulas in P_Π.

We will only define *rf*, since the axiomatisation of *bf* is similar. It is defined "inductively" starting from the root (the bottom formula) of the derivation Π.

1. Suppose at least one of the successors of G is a red formula. In this case we write:

$$\neg rf(G). \tag{5.17}$$

2. Otherwise, all the successors of G are either grey or blue:

$$\frac{G}{G_1 \quad \cdots \quad G_m \quad B_1 \quad \cdots \quad B_k}\ .$$

In this case we write

$$\begin{aligned} rf(G) \quad \leftrightarrow & ((rf(G_1) \vee bc(G_1)) \wedge \ldots \wedge (rf(G_m) \vee bc(G_m)) \wedge \\ & \neg rc(G_1) \wedge \ldots \wedge \neg rc(G_m)). \end{aligned} \tag{5.18}$$

5.5.4 Minimising Interpolants in Local Proofs

Theorem 5.5.5 shows how one expresses locality and digest using the propositional formulas P_Π. This allows us to introduce various measures of "quality" of interpolants

and use these measures, together with an SMT solver, to find local proofs giving interpolants that are better in these measures.

As usual, we define a clause to be a disjunction, possibly empty, of literals, that is, atomic formulas and their negations. Since most theorem provers and SMT solvers present proofs as dags of clauses, apart from some preprocessing deriving a set of clauses from R, B and the theory, we assume that the digest of a proof is a set of clauses. If such a clause contains free variables, it is assumed to be implicitly universally quantified. We know that the interpolant extracted from a proof is a propositional combination of clauses occurring in this proof. If a particular clause is a propositional combination of smaller formulas, then the interpolant can be considered a propositional combination of these smaller formulas. The smallest formulas of this form are well-studied in the automated deduction community and called *components*.

Definition 5.5.6 [Component] A *component* of a clause C is its non-empty subclause such that it is the smallest possible, while for each variable that occurs in it, it contains all the literals that contain the variable.

We define *g-atom* to be either a non-ground component, or a ground literal with positive polarity.
For example, the clause $p(x) \vee a \neq 2 \vee q(x)$ has two components: $p(x) \vee q(x)$ and $a \neq 2$, and two g-atoms: $p(x) \vee q(x)$ and $a = 2$. Note that we have the following equivalence:

$$\forall x(p(x) \vee a \neq 2 \vee q(x)) \;\equiv\; \forall x(p(x) \vee q(x)) \vee \neg(a = 2).$$

In general, the universal closure of every clause is a boolean combination of the universal closures of its components. Therefore, the extracted interpolant is a boolean combinations of g-atoms, which are components of the formulas in the digest.

The problem of generating minimal reverse interpolants can be thus reduced to the problem of minimising, in some sense, the set of g-atoms used in the interpolants. As minimality of interpolants is not well-understood, we introduce various measures for minimising the size of interpolants. Namely, we are interested in minimising interpolants with respect to (i) the number of g-atoms and (ii) the total weight of g-atoms, counted as a number of symbols. One can also argue that ground interpolants are more useful than those containing quantifiers, so in addition, when the refutation is non-ground, we can also minimise (iii) the number of quantifiers in the g-atoms.

For doing so, we use the fact that the digest of a derivation can be expressed using propositional variables $d(G)$ over grey nodes G and transform the minimisation problem to solving a pseudo-boolean optimisation problem over V_Π as explained below.

We consider a local refutation Π. For every component g of a grey clause G of Π, we introduce a distinct propositional variable $v(g)$. Intuitively, this variable will denote that g occurs in the digest of the transformed proof Π'. For every grey node G in Π, let $g_1,\ldots,g_k$ be all g-atoms of G. We then introduce the following axiom:

$$d(G) \implies v(g_1) \wedge \ldots \wedge v(g_k). \tag{5.19}$$

In what follows, let $g_1,\ldots,g_m$ be all g-atoms occurring in all grey nodes of Π. Let $w_1,\ldots,w_m$ be the total weights of these atoms, respectively. We denote by $q_1,\ldots,q_m$ the number of quantifiers used, respectively, in $g_1,\ldots,g_m$.

The problem of minimising interpolants is then reduced to the problem of minimising (one of) the following sums:

$$\text{atom}_{\text{cost}} \stackrel{\text{def}}{=} v(g_1)+\ldots+v(g_m). \tag{5.20}$$

$$\text{weight}_{\text{cost}} \stackrel{\text{def}}{=} w_1 v(g_1)+\ldots+w_m v(g_m). \tag{5.21}$$

$$\text{quantifier}_{\text{cost}} \stackrel{\text{def}}{=} q_1 v(g_1)+\ldots+q_m v(g_m). \tag{5.22}$$

Each of these sums is expressed as a pseudo-boolean constraint over the g-atoms $g_1,\ldots.g_m$. A solution to the minimisation problem of the left-hand side of (5.20)-(5.22) gives us a subset of $\{g_1,\ldots,g_m\}$, such that the interpolant constructed from the boolean combinations of the formulas in this subset is a smallest interpolant among all interpolants that can be extracted from the various local Π' resulting from grey slicing.

Minimisation of $\text{atom}_{\text{cost}}$ gives the smallest interpolant in the number of distinct g-atoms. Likewise, the minimal values of $\text{weight}_{\text{cost}}$ and $\text{quantifier}_{\text{cost}}$ correspond to the interpolant with the smallest total weight and the smallest number of quantifiers, respectively. Algorithm 5.5.7 puts together the algorithm for minimising interpolants.

Algorithm 5.5.7 Minimising Reverse Interpolants
Input: Closed formulas R and B such that $R \implies \neg B$, and a refutation Π from R, B.
Output: Minimised reverse interpolants I_{atom}, I_{weight}, I_{quant} of R and $\neg B$
Assumption: All colored symbols of R and B are uninterpreted constants

1 **begin**

I. Proof Localisation.

2 Compute local proof Π_l from Π, using Theorem 5.4.2.

II. Expressing locality.

3 $\mathcal{G} := \{\}$, $P_\Pi := \{\}$

4 **for** each grey node G in Π_l **do**

5 Express $d(G)$. Let $P_\Pi := P_\Pi \cup \{(5.3),(5.4),(5.5),(5.17),(5.18),(5.9)\}$.

6 Express general properties. Let $P_\Pi := P_\Pi \cup \{(5.10),(5.11),(5.12)\}$.

7 If G is a leaf, $P_\Pi := P_\Pi \cup \{(5.13)\}$.

8 If G is introduced by an inference with only grey premises,

$$P_\Pi := P_\Pi \cup \{(5.14)\}.$$

9 If G is introduced by an inference with a red premise,

$$P_\Pi := P_\Pi \cup \{(5.15)\}.$$

10 If G is introduced by an inference with a blue premise,

$$P_\Pi := P_\Pi \cup \{(5.16)\}.$$

11 Compute $G = g_1 \vee \cdots \vee g_k$, where g_i are g-atoms.

12 $\mathcal{G} = \mathcal{G} \cup \{g_1, \ldots, g_k\}$

13 **endfor**

14 $P_\Pi := P_\Pi \cup \{(5.19)\}$

15 **endfor**

III. Minimising Interpolants.

16 $\text{min\_atom}_{\text{cost}} := \{g_{i_1}, \ldots, g_{i_n}\}$

$$:= \texttt{min}_{\{g_{i_1},\ldots,g_{i_n}\}} \left(\sum_{g_i \in \mathcal{G}} v(g_i) \ \wedge \ P_\Pi \right)$$

17 $\text{min\_weight}_{\text{cost}} := \{g_{i_1}, \ldots, g_{i_n}\}$

$$:= \texttt{min}_{\{g_{i_1},\ldots,g_{i_n}\}} \left(\sum_{g_i \in \mathcal{G}} w_i v(g_i) \ \wedge \ P_\Pi \right)$$

where w_i denotes the weight of g_i

18 $\text{min\_quant}_{\text{cost}} := \{g_{i_1}, \ldots, g_{i_n}\}$

$$:= \texttt{min}_{\{g_{i_1},\ldots,g_{i_n}\}} \left(\sum_{g_i \in \mathcal{G}} q_i v(g_i) \ \wedge \ P_\Pi \right)$$

where q_i denotes the number of quantifiers uses in g_i

19 $I_{\text{atom}} = \text{Interpolant}_{R,B}(\text{min\_atom}_{\text{cost}})$
20 $I_{\text{weight}} = \text{Interpolant}_{R,B}(\text{min\_weight}_{\text{cost}})$
21 $I_{\text{quant}} = \text{Interpolant}_{R,B}(\text{min\_quant}_{\text{cost}})$
22 **return** $\{I_{\text{atom}}, I_{\text{weight}}, I_{\text{quant}}\}$.
23 **end**

Algorithm 5.5.7 uses the result of Theorem 5.4.2 and starts with translating the input refutation Π of R, B into a local one Π_l (line 2). Note that this step is only applied when Π is non-local, more precisely, when the non-local steps of Π contain colored constants. Further, the set $\mathcal{G}$ of g-atoms from Π_l and the set P_Π of (pseudo-boolean) constraints expressing locality of Π_l are initialised (line 3). Next, for each grey node G in Π_l the constraints expressing locality conditions over the digest and inferences of Π_l are constructed, (lines 5-10). Note that the propositional formulas $rf(G)$ and $bf(G)$ are expressed based on the dag-representation of proofs. The set of g-atoms of G is extracted and added to $\mathcal{G}$ (lines 11-12). Then, the property whether G is in the digest of Π_l is expressed in terms of g-atoms and added to the constraint set P_Π (line 14). As a result of these steps, at the end of line 15 of Algorithm 5.5.7, the constraint set P_Π is expressed as a set of clauses ensuring the locality of Π_l (Theorem 5.5.5). Next, minimal interpolants wrt to the number of g-atoms (line 16), the total weight of g-atoms (line 17), and the number of quantifiers in the g-atoms (line 18) are derived by solving a pseudo-boolean optimisation problem over g-atoms. To this end, the interpolation procedure $\texttt{Interpolant}_{R,B}(\dots)$ of [KV09b] is called to generate interpolants as boolean combinations of the derived minimal set of g-atoms (lines 19-21).

Theorem 5.5.8 Algorithm 5.5.7 is correct. That is, given two formulas R and B and a refutation Π, Algorithm 5.5.7 returns the minimal interpolants of R and $\neg B$[1] wrt the imposed measures (5.20)-(5.22) among all interpolants extracted from proofs obtained by grey slicing of Π.

We next show that finding minimal interpolants by Algorithm 5.5.7 is NP-hard.

Theorem 5.5.9 Given two formulas R and B and a refutation Π of $R, B \implies \bot$. Extracting a minimal reversed interpolant from Π by grey slicing is an NP-hard optimisation problem.

PROOF. We use a reduction from finding a maximal independent set of a graph $G(V,E)$ with a set of vertices $V = \{v_1, \dots, v_m\}$ and a set of edges $E = \{(u_1, w_1), \dots, (u_n, w_n)\}$, which is known to be NP-hard.

[1] i.e. reversed interpolants of R and B

To fulfil conditions of Theorem 5.2.2, we first fix a background theory T. For each vertex $v \in V$ we introduce a propositional grey variable, also denoted by v, of weight 1. Further, for each edge $(u,v) \in E$ we add $u \implies v$ to the theory T.

Introduce also a blue propositional variable b and a red propositional variable r. Define B to be the blue formula $v_1 \wedge \cdots \wedge v_m \wedge b$ and R to be the red formula $\neg v_1 \wedge \cdots \wedge \neg v_m \wedge r$.

Further, for each edge $(u,w) \in E$ we introduce the following derivation $\Pi_{u,w}$:

$$\dfrac{\dfrac{B}{u}}{w}$$

Note that this derivation is sound in the underlying theory T. We next construct the proof tree Π to be:

$$\dfrac{\dfrac{B}{u_1}\quad}{w_1} \quad \ldots \quad \dfrac{\dfrac{B}{u_n}}{w_n} \quad R}{\bot}$$

where the weight of $\bot$ is considered to be zero. Observe that Π is a valid refutation of R, B. Also note that building a minimal interpolant from Π reduces to finding a derivation Π' obtained from Π by grey slicing with a minimal number of g-atoms.

Let Π' be any derivation obtained from Π by grey slicing. Denote by D' the digest of Π'. For every edge $(u,w) \in E$, the subderivation $\Pi_{u,w}$ either remains a subderivation of Π', or u gets sliced off. In the first case we have $u \in D'$, in the second case $w \in D'$. Therefore, either $u \notin V - D'$, or $w \notin V - D'$, which implies that $V - D'$ is an independent set.

Using similar arguments, one can prove that every independent set S of vertices is a subset of $V - D'$ for some digest D' of a derivation obtained from Π by grey slicing. As each set $V - D'$ is an independent set as well, for every maximal independent set S there exists a digest D' such that S is equal to $V - D'$. Therefore finding a digest of the minimal size is equivalent to finding a maximal independent set. □

Let us finally note that our method can be extended with other proof transformations and optimisation criteria (e.g., [DKPW10, JM07b]), to improve the quality of interpolants. For example, many approaches use templates to identify predicates desirable to be used in invariants or interpolants. Algorithm 5.5.7, thanks to its generality, can easily be modified to give preference to predicates matching predefined templates. We therefore believe that our method is of an independent value, since one can first

	Interpolant size decrease					
	some	$2-4$	$4-6$	$6-8$	>8	to 0
Symbols	1369	369	55	24	20	386
g-atoms	912	248	37	16	7	386

Table 5.1: Minimisation results on 6577 TPTP problems with non-trivial interpolants.

minimise the interpolant and then try to make it semantically better using other methods. Another important feature of our algorithm is that it optimises the proof globally: that is, the optimal solution is not necessarily a sum of optimal solutions given by local proof transformations. We believe this a very essential property of the algorithm not shared by other known approaches to minimising interpolants.

5.6 Experimental Results

5.6.1 Implementation

We implemented our interpolant minimisation method in C++ and integrated it in version 1.8 of the Vampire theorem prover [RV02]. For minimising the set of pseudo-boolean constraints we used version 1.0.29 of the SMT solver Yices [Dd06].

Due to the lack of realistic verification benchmarks, that is examples coming from some industrial project, we evaluated our method on the following two classes of problems. First, we took a collection of examples over first-order logic with equality from the TPTP library [Sut09]. We minimised interpolants in the first-order proofs generated by Vampire. Second, we considered SMT benchmarks from the SMT-Lib library [BST10] that come from bounded model checking. We analysed proofs generated by the Z3 SMT solver [dMB08]. We used version 2.19 of Z3 without any modification. However, for minimising interpolants from Z3 proofs, we implemented a parser for processing and translating Z3 proofs into local proofs. To this end, we used our algorithm for proof localisation (see proof of Theorem 5.4.2).

All experiments reported in this paper were carried out using a 64-bit 2.33 GHz quad core Dell server with 12 GB RAM.

	0	< 5	$5-9$	$10-19$	$20-49$	$50-99$	$100-199$	≥ 200
Before	3055	530	1099	1578	2035	991	260	84
After	3441	522	1225	1557	1882	766	166	73

Table 5.2: Number of symbols in TPTP benchmark interpolants, before and after minimisation.

5.6.2 First-Order Problems

For this part of the experiments, we took a collection of first-order problems from the TPTP library. We started with annotating these problems with coloring information, using the following coloring strategies.

(1) We order symbols by the number of their occurrences in the problem, and starting with the symbols occurring the least number of times, we attempt to assign colors to them. A color can be assigned to a symbol if the symbol does not occur in a formula with a symbol that was already assigned with the opposite color. The colors are being assigned in an alternating manner. If the last assigned color was red, we first attempt assigning blue to the next symbol, and try to assign red only if this the blue color ended in an unsuccessful assignment (i.e. an input formula with two different colored symbols is obtained). We stop when we have attempted to assign a color to all the symbols.

(2) The previous assignment strategy is more or less random. To use interpolants in a more logical way, we used the following idea. Typically, TPTP problems come with annotations classifying formulas from a problem into axioms, conjectures and hypotheses. We have to prove the conjecture from the axioms and hypotheses. It is commonly the case that axioms axiomatise some theory, so we have to prove that the hypotheses imply the conjecture in the theory given by the axioms. This gives us the following way of coloring the problem symbols. We assign blue color to symbols appearing only in the formulas of the conjecture (i.e. formula B), and red color to symbols occurring only in hypothesis (i.e. formula R). The symbols shared by the conjecture and the hypotheses are considered grey, as well as the symbols occurring only in the axioms.

Local proofs for the colored TPTP problems were generated using the interpolation mode of Vampire [HKV10]. Altogether, we evaluated our minimisation method on 9632 colored TPTP examples. Out of the 9632 problem instances, 3055 problems had trivial interpolants, that is the interpolant was either $\top$ or $\bot$. This left us with

	0	<5	5-9	10-19	20-49	50-99	100-199	≥200
$\text{Symbols}_{\text{pre}}$	112	3	149	150	82	90	321	1216
$\text{Symbols}_{\text{post}}$	112	5	168	158	56	87	323	1214
g-atoms	112	1558	303	114	9	0	0	0
Quantifiers	464	279	291	367	394	157	123	48

Table 5.3: Minimisation results on 2123 SMT benchmarks.

6577 problems with non-trivial interpolants. We were interested to see how our minimisation algorithm performs on these problems. To this end, for each of the 6577 problems, our minimisation method took the corresponding local proof generated by Vampire and derived the smallest interpolants by minimising (i) the number of symbols (i.e total weight) and (ii) the number of g-atoms in the interpolant. Table 5.1 gives a summary on how the size of the interpolant decreases after minimisation, as compared to the interpolant extracted from the original proof. Rows 1 and 2 of Table 5.1 show respectively the changes in the interpolant size after minimising the number of symbols, respectively the number of g-atoms in the interpolant. For each imposed measure, column 1 of Table 5.1 lists the number of examples where the size of the interpolants has decreased only by a small amount. The numbers shown in column 2 (resp. in column 3, column 4, and column 5) correspond to the number of those examples whose interpolants became 2-4 (resp. 4-6, 6-8, and more than 8) times smaller after minimisation. Column 6 gives the number of examples whose interpolants became trivial after minimisation, even though the non-minimised interpolants were non-trivial.

In Table 5.2 we report on the number of symbols in the interpolants before (row 1) and after (row 2) minimisation. Each column of Table 5.2 gives the number of interpolants whose number of symbols satisfy the numeric constraint given in the first cell of the column. That is, column 1 gives the number of trivial interpolants (the number of symbols is 0), column 2 shows the number of interpolants with less than 5 symbols, column 3 reports shows the number of interpolants that contain between 5 and 9 symbols etc.

By analysing the results of Table 5.1, we note that for 854 (respectively 694) examples the number of symbols (respectively, the number of g-atoms) of the interpolant decreased by at least a factor of 2. However, we also note that for 4354 (respectively 4971) problems out of the 6577 examples we tried minimisation did not improve the size: these examples are omitted in Table 5.1. As the qualitative studies of interpolants is a challenging topic, we believe that the experimental results reported in Table 5.1

show the potential of our method in generating better interpolants.

In Figures 5.2 and 5.3 we show a colored proof of a TPTP problem before and after minimization. Formulas appearing in the reverse interpolant are given in bold, while other consequences of symbol eliminating inferences in italic. Red symbols in the proof are underlined, whereas blue symbols are underbraced. Figures 5.4 and 5.5 show the proof before and after minimisation in a tree form. As mentioned, formulas denoted by R (resp. by B or G) refer to red (resp. blue or grey formulas). The formulas G_{814} and G_{41} appear in the original interpolant, but when G_{815}, G_{45} and G_{41} are sliced off by the minimisation algorithm, the new interpolant formulas are G_{853} and G_{86}. This is because the formula G_{853} is eliminating red symbols from the premises it received as a result of the slicing. The formula G_{86} now appears in the interpolant because it is an ancestor of a red symbol eliminating formula. Even though we still have two formulas in the interpolant, its size decreased because G_{853} is a trivial formula $\bot$. When compared to Figure 5.4, note that the number of grey formulas in Figure 5.5 has decreased due to grey slicing.

5.6.3 Experiments with SMT Problems

We used a set of SMT-Lib benchmarks coming from bounded model checking. Variables in these problems correspond to state variables representing various unrolling steps of loops. These variables are typically indexed by integer constants, where the integer index expresses the unrolling step to which the state variable belongs to.

Translating and localising Z3 proofs. We generated proofs of SMT problems by using Z3. Z3 proofs are expressed in the sequent calculus, while our proof localisation and minimisation algorithms work with resolution-style proofs. We therefore parsed and translated Z3 proofs into our framework by using a simple Lisp parser. To this end, we replaced conditionalised Z3 formulas of the form $A_1, \ldots, A_n \vdash F$ by implications $(A_1 \vee \ldots \vee A_n) \rightarrow F$.

The coloring strategy we used for the SMT benchmarks was as follows. Except the state variables, all other symbols were colored grey. We divided the set of state variables into three parts. State variables corresponding to the middle loop unrolling step were left grey, variables from earlier states were marked red and those from later states were colored blue. In our experiment this coloring strategy turned out to be successful, in 95% of all examples we tried input formulas have been translated into formulas colored by at most one color.

853. $\bot$ [815,86]
815. $\neg$ udl(sK_0,rl(sK_0)) [814,45]
814. $\neg$ udl(x_0,rl(x_1)) $\vee$ $\neg$ udl(x_0,x_1) [813,17]
813. $\neg$ udl(x_0,x_1) $\vee$ *lcl*(x_0,x_1) $\vee$ $\neg$ udl(x_0,rl(x_1)) [15,17]
86. udl(x_0,rl(x_0)) [79,49]
79. udl(x_9,x_7) $\vee$ $\neg$ udl(*pt p* (x_7,x_8),x_9) [61,42]
61. udl(x_7, *pt p* (x_6,x_8)) $\vee$ $\neg$ udl(x_7,x_5) $\vee$ udl(x_5,x_6) [57,33]
57. $\neg$ udl(x_5, *pt p* (x_6,x_7)) $\vee$ udl(x_5,x_6) [33,43]
49. udl(*pt p* (rl(x_3),x_4),x_3) [38,43]
45. udl(sK_0,rl(rl(sK_0))) [30,41]
43. $\neg$ udl(*pt p* (x_1,x_2),x_1) [25,24]
42. $\neg$ udl(x_0,x_0) [25,27]
41. udol(sK_0,rl(sK_0)) [6,7]
38. udl(x_0,rl(x_1)) $\vee$ udl(x_0,x_1) [input]
33. udl(x_1,x_2) $\vee$ $\neg$ udl(x_0,x_1) $\vee$ udl(x_0,x_2) [input]
30. $\neg$ udol(x_0,x_1) $\vee$ udl(x_0,rl(x_1)) [input]
27. eld(x_0,x_0) [input]
25. $\neg$ udl(x_0,x_1) $\vee$ $\neg$ eld(x_0,x_1) [input]
24. eld(*pt p* (x_1,x_0),x_1) [input]
17. $\neg$ *lcl*(x_0,x_1) [input]
15. $\neg$ udl(x_0,x_1) $\vee$ *lcl*(x_0,x_1) $\vee$ $\neg$ udl(x_0,rl(x_1)) $\vee$ *lcl*(x_0,rl(x_1)) [input]
7. $\neg$ *edol*(sK_0,rl(sK_0)) [input]
6. udol(x_0,x_1) $\vee$ *edol*(x_0,x_1) [input]

Figure 5.2: Proof of the GEO269+3 problem from the TPTP library.

However, the usage of colors yielded non-local Z3 proofs in more than 90% of all examples we tried. We translated non-local Z3 proofs into local ones by applying our proof localisation algorithm. To this end, we always used existential quantification to eliminate red symbols from non-local inferences. As the size of generated interpolants depends on the introduced quantified formulas, we believe that a dynamic analysis over the colored symbols to be eliminated, for example eliminate blue symbol instead of a red one, is an interesting topic for further examination.

The result of proof localisation was further used to minimise interpolants.

Minimising local SMT proofs. Altogether, we used 4347 SMT benchmarks. Out of these 4347 examples, we generated reverse interpolants for 2123 problems. We analyse these interpolants below and summarize our results in Table 5.3.

853. $\bot$ **[814,86,30,6,7]**
814. $\neg$ *udl*(x_0,*rl*(x_1)) $\vee$ $\neg$ *udl*(x_0,x_1) *[813,17]*
813. $\neg$ udl(x_0,x_1) $\vee$ *lcl*(x_0,x_1) $\vee$ $\neg$ udl(x_0,rl(x_1)) [15,17]
86. udl(x_0,**rl**(x_0)) **[79,49]**
79. udl(x_9,x_7) $\vee$ $\neg$ udl(*ptp*(x_7,x_8),x_9) [61,42]
61. udl(x_7,*ptp*(x_6,x_8)) $\vee$ $\neg$ udl(x_7,x_5) $\vee$ udl(x_5,x_6) [57,33]
57. $\neg$ udl(x_5,*ptp*(x_6,x_7)) $\vee$ udl(x_5,x_6) [33,43]
49. udl(*ptp*(rl(x_3),x_4),x_3) [38,43]
43. $\neg$ udl(*ptp*(x_1,x_2),x_1) [25,24]
42. $\neg$ udl(x_0,x_0) [25,27]
38. udl(x_0,rl(x_1)) $\vee$ udl(x_0,x_1) [input]
33. udl(x_1,x_2) $\vee$ $\neg$ udl(x_0,x_1) $\vee$ udl(x_0,x_2) [input]
30. $\neg$ udol(x_0,x_1) $\vee$ udl(x_0,rl(x_1)) [input]
27. eld(x_0,x_0) [input]
25. $\neg$ udl(x_0,x_1) $\vee$ $\neg$ eld(x_0,x_1) [input]
24. eld(*ptp*(x_1,x_0),x_1) [input]
17. $\neg$ *lcl*(x_0,x_1) [input]
15. $\neg$ udl(x_0,x_1) $\vee$ *lcl*(x_0,x_1) $\vee$ $\neg$ udl(x_0,rl(x_1)) $\vee$ *lcl*(x_0,rl(x_1)) [input]
7. $\neg$ *edol*(sK_0,rl(sK_0)) [input]
6. udol(x_0,x_1) $\vee$ *edol*(x_0,x_1) [input]

Figure 5.3: Transformed proof of Figure 5.2 by slicing off formulas using weight minimization.

The remaining 2224 SMT problems we could not fully process. This was due to a 60s time limit which we imposed as the processing time of one problem, including proof translation, coloring and localisation. In 102 cases the run was terminated by reaching the time limit in translating and coloring Z3 proofs, in 1580 cases the timeout was reached in the proof localisation phase, and for 542 benchmarks the time limit was reached during the minimization phase (in constructing and minimising the set of propositional formulas).

Among the 2123 interpolants, 112 interpolants were trivial, and 1659 contained existential quantifiers introduced by proof localisation. The number of symbols in the interpolants was decreased by our minimisation algorithm for 331 interpolants, out of which 14 interpolants had a decrease by more than two times. The number of g-atoms in the interpolant decreased for 83 interpolants, whereas the number of quantified variables was minimised for 7 interpolants. Table 5.3 shows the distribution of the number of symbol occurring in interpolants before (row 1) and after minimization (row 2). The

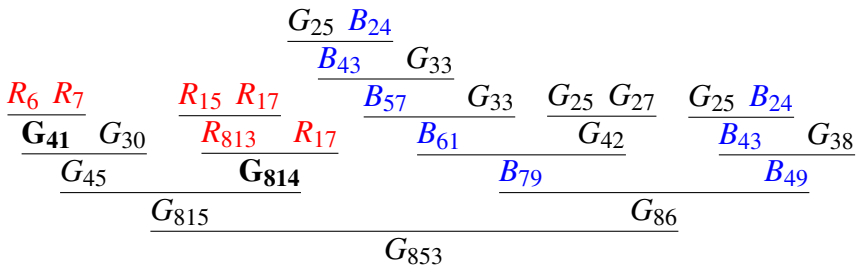

Figure 5.4: Proof tree for Figure 5.2.

distribution of the number of g-atoms (row 3) and quantifiers (row 4) in interpolants is shown only before minimization, because the effect of minimisation on these values was not significant. Each column of Table 5.3 gives the number of interpolants whose number of size/g-atoms/quantifiers are bounded by the numeric value given in the first cell of the column. That is, for examples, the number of symbols in 168 (row 2, column 3) interpolants is between 5-9 after minimisation. The numbers given in column 1 of Table 5.3 correspond to the number of trivial interpolants.

The experiments show that our minimisation algorithm is not very efficient on this benchmark suite compared to the first-order benchmarks. We believe that the problem is not in the method but in the way Z3 produces proofs (since the produced proofs were not intended for interpolation). It was often the case that the proofs contained very large formulas, sometimes mixing colors in these formulas. The formulas are then quantified by other algorithm and cannot further be removed from the proof, thus spoiling the minimisation statistics. These formulas are normally large conjunctions or if-then-else expressions, which can also be represented as conjunctions and could have been split into smaller ones. This would not only replace large formulas by smaller one, but also improve coloring of proofs and reduce (or eliminate) the necessity to quantify formulas in them. We believe that our technique will work very well if SMT solvers are modified to obtain proofs of a better quality. Moreover, once a proof is found, post processing can also be done and one may try to change non-local parts of the proof again by theorem proving.

5.7 Related Work

Interpolation has a number of application in formal verification, ranging from approximating the set of reachable sets in predicate abstraction [JM06, JM07b] to invariant

generation of loops [McM08]. Formal verification thus crucially depends to which extent "good" interpolants can be automatically generated.

General criteria for comparing interpolants can be defined by the logical strength of the interpolant, see e.g. [JM07b, DKPW10]. The approach described in [JM07b] reorders the sequence of resolution steps in a proof to strengthen the derived interpolants. The main heuristic used for proof transformation is to make resolution steps on red/blue variables before those on grey variables. The work of [DKPW10] extends [JM07b] and gives a theoretical investigation on the logical strength of propositional interpolants extracted from resolution proofs. The approach uses the notion of labeling functions, which essentially label literals by red, blue or grey labels. The differences among the labeling functions come from how grey literals are labeled (red, blue, or grey). The strength of the various labeling functions is compared, and weaker or stronger interpolants are derived by changing the deployed labeling functions and swapping some nodes in the derivation.

Examples of [DKPW10] emphasise that weaker interpolants might lead to better performance, whereas experimental results of [JM07b] show that stronger interpolants can speed up the convergence of a software model checker based on predicate abstraction. Optimising interpolants by only using the logical strength of the interpolant as a selection criteria is thus not always the best way to go in designing efficient interpolation algorithms.

The logical strength of the interpolant is also evaluated in [JM06, McM08], in the context of verification of programs with loops. Although one can derive various program properties by unwinding loop iteration, the resulting set of program properties is a diverging sequence of non-inductive formulas. In [JM06] interpolants are generated by searching the proof space and avoiding divergence by deeper unwindings of loop iterations. The method is further extended in [McM08] to infer quantified interpolants. It is shown that by bounding the behavior of the interpolating prover (e.g. delaying inferences over colored or grey symbols), divergence is prevented and an inductive invariant is eventually produced from quantified interpolants. A somehow related approach is presented in [KV09b, HKV10], where quantified interpolants are extracted from first-order local proof. These techniques generate interpolants by taking the boolean combinations of the grey conclusions of the largest colored subderivations.

The works of [BKRW10, KLR10, BKRW11, GLS11] evaluate the quality of interpolants by using, in some sense, a different selection criteria. These methods are motivated to generate interpolants that are small in the number of their components,

and describe interpolation procedures for the theory of linear integer arithmetic w/o uninterpreted function and predicate symbols. The approach of [BKRW10] computes ground interpolants that are exponential in the size of the proofs. The method is improved in [KLR10] by restricting the logical power of the interpolating prover, and is further extended in [BKRW11] by handling uninterpreted function and predicate symbols. To this end, [BKRW11] shows that quantified interpolants are needed. However, by using guarded quantifiers and divisibility predicates, the quantified interpolants can be translated into equivalent quantifier-free formulas. A similar problem is addressed and solved in [GLS11], where ceiling functions are used to avoid quantified interpolants and generate quantifier-free interpolants of quantifier-free formulas in linear integer arithmetic. Ceiling functions are handled in the interpolating prover by replacing every non-variable ceiling term by a fresh integer variable. Inequality constraints over the newly introduced integer variables are added to capture the semantics of ceiling terms. Whereas [BKRW10, KLR10, BKRW11, GLS11] show good performance on experiments, due to the lack of realistic benchmarks, it is hard to draw broad conclusions whether the interpolants generated by these works are the "best" in size and expressiveness.

Contrary to all aforementioned works, we define a set of pseudo-boolean constraints over the grey formulas of the proof. Any solution to this set of constraints gives a different interpolant, and any interpolant can be expressed as a solution of the constraint set. The proof transformations carried out in our approach use only slicing off formulas that are logical consequences of other formulas. Furthermore, we evaluate the logical strength of interpolants by minimising the size, the total weight and the number of quantifiers. Unlike [JM06, JM07b, DKPW10, BKRW10, KLR10, BKRW11, GLS11], our method can generate and minimise interpolants of quantified formulas. When compared to [HKV10], our experiments show that we get better interpolants then the ones of [HKV10] extracted from the largest colored subderivations. More generally spoken, our minimisation algorithm can be applied to any input proof, provided that the input proof can be translated into an equivalent local proof. A special case of such proofs are those whose only colored symbols are uninterpreted constants. Although such a condition might sound severe, it turns out that in practice a large class of examples satisfy this imposed restriction: interpolation benchmarks in the combined theory of uninterpreted functions, predicates and linear integer arithmetic coming from the SMT community satisfy this coloring constraint [BKRW10, KLR10, BKRW11, GLS11].

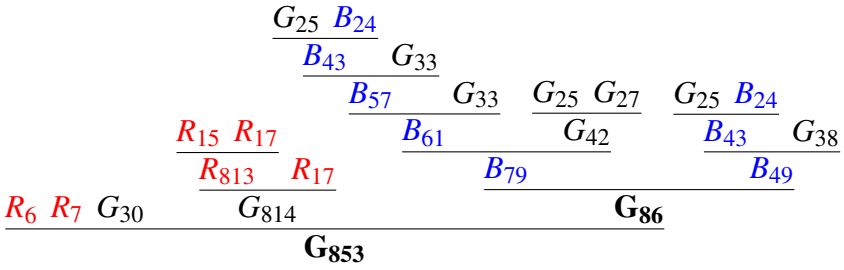

Figure 5.5: Proof tree for the minimized proof of Figure 5.3.

5.8 Conclusion

We described how interpolants extracted from arbitrary proofs can be obtained and minimised in various ways giving smaller interpolants. Our method (1) takes an arbitrary refutation proof, (2) translates it into a local one, provided that all colored symbols are uninterpreted constants, (3) applies minimisation based on analysis of grey areas in the refutation, and (4) computes a minimal interpolant by using pseudo-boolean optimisation.

Our method is very general and can be used with any theory and in conjunction with any theorem prover that outputs refutation proofs of interpolation problems. The evaluation of our method on first-order and SMT bounded model checking benchmarks shows that, in many cases, minimisation considerably decreases the interpolant size.

We intend to integrate our method into concrete verification tools and evaluate our approach on more realistic verification benchmarks. An interesting question we plan to address in the future is how the quality of minimised interpolants effects the efficiency of interpolation-based verification methods. Using a highly optimised pseudo-boolean solver instead an SMT solver is left for further experiments.

We believe that our method opens a new avenue on research in interpolation-based methods. Indeed, other proof transformation methods can be used as well. For example, we can quantify away not only red, but sometimes also blue symbols or slice off colored formulas. In addition, as we pointed out in Section 5.6 better proofs can considerably improve the quality of interpolants.

Chapter 6

Invariant Generation in Vampire

Authors: Krystof Hoder, Laura Kovacs, Andrei Voronkov

This paper describes a loop invariant generator implemented in the theorem prover Vampire. It is based on the symbol elimination method proposed by two authors of this paper. The generator accepts a program written in a subset of C, finds loops in it, analyses the loops, generates and outputs invariants. It also uses a special consequence removal mode added to Vampire to remove invariants implied by other invariants. The generator is implemented as a standalone tool, thus no knowledge of theorem proving is required from its users.

6.1 Introduction

In [KV09a] a new *symbol elimination* method of loop invariant generation was introduced. The method is based on the following ideas. Suppose we have a loop L with a set of (scalar and array) variables V. The set V defines the *language* of L. We extend the language L to a richer language L' by a number of functions and predicates. For every scalar variable v of the loop we add to L' a unary function $v(i)$ which denotes the value of v after i iterations of L, and similar for array variables. Thus, all loop variables are considered as functions of the loop counter. Further, we add to L' so-called *update predicates* expressing updates to arrays as formulas depending on the loop counter. After that we automatically generate a set P of first-order properties of the loop in the language L'. These properties are valid properties of the loop, yet they are not loop invariants since they use the extended language L'.

Note that any logical consequence of P that only contains variables in L is also a loop invariant. Thus, we are interested in finding logical consequences of formulas in

P expressed in L. To this end, we run a first-order theorem prover using a saturation algorithm on P in such a way that it tries to derive formulas in L. To obtain a saturation algorithm specialised to efficiently derive consequences in L, we enhanced the theorem prover Vampire [HKV10] by so-called *colored proofs* and a *symbol elimination mode*. In colored proofs, some (predicate and/or function) symbols are declared to have colors, and every proof inference can use symbols of at most one color.

As reported in [KV09a], we tested Vampire on several benchmarks for invariant generation. It was shown that symbol elimination can infer complex properties with quantifier alternations. Symbol elimination thus provides new perspectives in automating program verification, since such invariants could not be automatically derived by other methods.

As the method is new, its practical power and limitations are not well-understood. The main obstacle to its experimental evaluation lies in the fact that program analysis and generation of input for symbol elimination by a separate tool is error-prone and requires full knowledge of our invariant generation method. The tool described in this paper was designed with the purpose of creating a *standalone tool implementing invariant generation* by symbol elimination. Vampire can still be used for symbol elimination only, so that the program analysis is done by another tool (for example, for experiments with variations of the method).

The purpose of this paper is to describe the program analyser and loop invariant generator of Vampire, their implementation and use. We do not overview Vampire itself.

6.2 Related work

Reasoning about loop invariants is a challenging and widely studied research topic. We overview only some papers most closely related to our tool.

Automatic loop invariant generation is described in a number of papers, including [GRS05, SG09, HP08, McM08, GR09, HHKR10]. In [SG09] loop invariants are inferred by predicate abstraction over a set of a priori defined predicates, while [GR09] employs constraint solving over invariant templates. Input predicates in conjunction with interpolation are used to infer invariants in [McM08]. Unlike these works, we require no used guidance in providing input templates and/or predicates. User guidance is also not required in [GRS05, HP08], but invariants are derived using abstract interpretation [GRS05, HP08] or symbolic computation [HHKR10]. However, these

approaches can only infer universally quantified invariants, whereas we can also derive invariants with quantifier alternation.

Our work is also related to first-order theorem proving [Sch04, WSH+07, Kor09b]. These works implement superposition calculi, with a limited support for theories. However, only Vampire implements colored proofs and consequence removal essential for the symbol elimination method.

A more complex and general framework for program analysis is given in, e.g., [BLS04, CCPS10]. Whereas in [BLS04, CCPS10] theorem proving is integrated in a program analysis environment, we integrate program analysis in a theorem proving framework. Although our approach at the moment is limited to the analysis of a restricted class of loops, we are able to infer richer and more complex quantified invariants than [BLS04, CCPS10]. Combining our method with other techniques for verification and invariant generation is left for further work.

6.3 System Implementation

To create an integrated environment for invariant generation, we implemented a simple program analyser and several new features in Vampire. The workflow of the invariant generation process is given in Figure 6.1.

The analyser itself comprises about 4,000 lines of C++ code (all Vampire is written in C++). In addition to the analyser, we had to extend formulas and terms with if-then-else and let-in constructs, implement colored proofs, automatic theory loading, and the consequence removal mode. All together making Vampire into an invariant generator required about 12,000 lines of code. Currently, the analyser only generates loop properties for symbol elimination, but we plan to use it in the future for a more powerful integration of program analysis and theorem proving.

6.3.1 Program analysis

The program analysis part works as follows. First, it extracts all loops from the input program. It ignores nested loops and performs the following steps on every non-nested loop.

1. Find all loop variables.

2. Classify them into variables updated by the loop and constant variables.

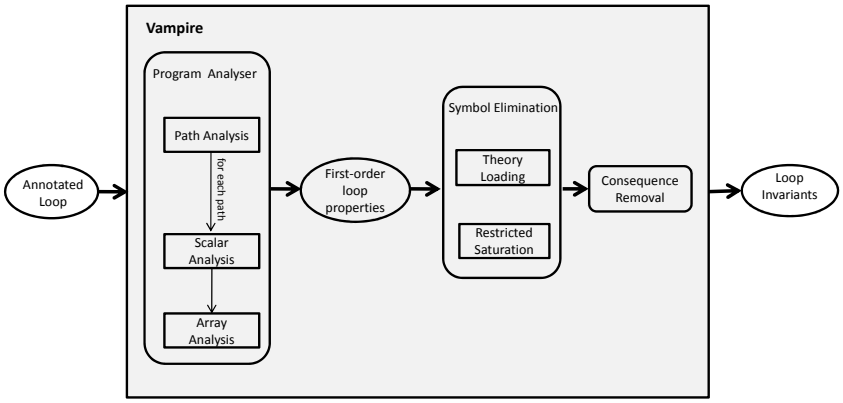

Figure 6.1: Program Analysis and Invariant Generation in Vampire

3. Find counters, that is, updated scalar variables that are only incremented or decremented by constant values. Note that expressions used as array indexes in loops are typically counters.

4. Save properties of counters.

5. Generate update predicates of updated array variables and save their properties.

6. Save the formulas corresponding to the transition relation of the loop.

7. Generate a symbol elimination task for Vampire.

The input to the analyser is a program written in a subset of C. The subset consists of scalar integer variables, array variables, arithmetical expressions, assignments, conditionals and while-do loops. Nested loops are not yet handled.

6.3.2 Symbol Elimination and Theory Loading

The program analyser generates a set of first-order loop properties and information about which symbols should be eliminated. A (predicate and/or function) symbol is to be eliminated in Vampire whenever it is specified to have some color. The next phase of invariant generation runs symbol elimination on the set of formulas generated by the analyser. Before doing symbol elimination, Vampire checks which theory symbols (such as integer addition) are used and loads axioms relevant to these theory symbols. Theory symbols have no color in Vampire. After theory loading, Vampire runs a saturation algorithm on the theory axioms and the formulas generated by its analyser. A special term ordering is used to ensure that symbol elimination is effective and efficient.

6.3.3 Consequence Removal

The result of the symbol elimination phase is a set of loop invariants. This set is sometimes too large. For example, it is not unusual that Vampire generates over a hundred invariants in less than a second.

An analysis of these invariants shows that some invariants are concise and natural for humans, while some other invariants look artificial (this does not mean they are not interesting and/or not useful). It is typically the case that the generated set of invariants contains many invariants implied by other invariants in the set.

The next phase of invariant generation prunes the generated set by removing the implied invariants. Checking whether each generated invariant is implied by all other invariants is too inefficient. To remove them efficiently, we implemented a special consequence removal mode. The output of the tool is the set of all non-removed invariants.

6.3.4 Availability

We implemented our approach to invariant generation in Vampire. The new version of Vampire is available from `http://www.vprover.org`. The current version of Vampire runs under Windows, Linux and MacOS.

6.4 Experiments

We evaluated invariant generation in Vampire using two benchmark suites: (1) challenging loops taken from [GRS05, SG09], and (2) a collection of 38 loops taken from programs provided by Dassault Aviation. We used a computer with a 2GHz processor and 2GB RAM and ran experiments using Vampire version 0.6. The symbol elimination phase was run with a 1 second time limit and the consequence removal phase with a 20 seconds time limit.

For all the examples the program analyser took essentially no time. It turned out that symbol elimination in one second can produce a large amount of invariants, ranging from one to hundreds. Consequence removal normally deletes about 80% of all invariants.

6.5 Conclusion

It is not unusual that program analysers call theorem provers or contain theorem provers as essential parts. Having a program analyser as part of a theorem prover is less common. We implemented an extension of Vampire by program analysis tools, which resulted in a standalone automatic loop invariant generator. Our tool derives logically complex invariants, strengthening the state-of-the-art in reasoning about loops.

Chapter 7

Interpolation and Symbol Elimination in Vampire

Authors: Krystof Hoder, Laura Kovacs, Andrei Voronkov

It has recently been shown that proofs in which some symbols are colored (e.g. local or split proofs and symbol-eliminating proofs) can be used for a number of applications, such as invariant generation and computing interpolants.

This tool paper describes how such proofs and interpolant generation are implemented in the first-order theorem prover Vampire.

7.1 Introduction

Interpolation offers a systematic way to generate auxiliary assertions needed for software verification techniques based on theorem proving [JM06, KV09b], predicate abstraction [HJMM04, JM06], constraint solving [RSS07], and model-checking [McM03, CGS08].

In [KV09a] it was shown that symbol-eliminating inferences extracted from proofs can be used for automatic invariant generation. Further, [KV09b] gives a new proof of a result from [JM06] on extracting interpolants from colored proofs:[1] this proof contains an algorithm for building (from colored proofs) interpolants that are boolean combinations of symbol-eliminating steps. Thus, [KV09b] brings interpolation and symbol elimination together.

Based on the results of [KV09a, KV09b] we implemented colored proof generation in the first-order theorem prover Vampire [RV02]. Colored proofs form the base for

[1] Such proofs are also called *local* and *split proofs*, in this paper we will call them colored.

our interpolation and symbol elimination algorithms.

The purpose of this paper is to describe how interpolation and symbol elimination are implemented and can be used in Vampire. We do not overview Vampire itself but only describe its new functionalities. The presented features have been explicitly designed for making Vampire appropriate for formal software verification: symbol elimination for automated assertion (invariant) synthesis and computation of Craig interpolants for abstraction refinement. Unlike its predecessors, the "new" Vampire thus provides functionalities which extend the applicability of state-of-the-art first theorem provers in verification. To the best of our knowledge, it is the first theorem prover that supports both invariant generation and interpolant computation.

The obtained symbol eliminating inferences and interpolants contain quantifiers, and can be further used as invariant assertions to verify properties of programs manipulating arrays and linked lists [McM08, KV09a]. We believe that software verification may benefit from the interpolant generation engine of Vampire.

Implementation. The new version of Vampire is available from `http://www.vprover.org` and runs under most recent versions of Linux (both 32 and 64 bits), MacOS and Windows. Vampire is implemented in C++ and has about 160,000 lines of code.

Experiments. We successfully applied Vampire on benchmarks taken from recent work on interpolants and invariants [JM05, YM05, GRS05, GT06, RSS07, JM07a] – see Section 7.4 and the mentioned URL. Our methods can discover required invariants and interpolants in all examples, suggesting its potential for automated software verification.

Related work. There are several interpolant generation algorithms for various theories. For example, [McM03, HJMM04, JM06, CGS08] derive interpolants from resolution proofs in the combined ground theory of linear arithmetic and uninterpreted functions. The approach described in [RSS07] generates interpolants in the combined theory of arithmetic and uninterpreted functions using constraint solving techniques over an a priori defined interpolants template. The method presented in [McM08] computes quantified interpolants from first-order resolution proofs over scalars, arrays and uninterpreted functions.

Our algorithm implemented in Vampire automatically extracts interpolants from colored first-order proofs in the superposition calculus. Theories, such as arithmetic

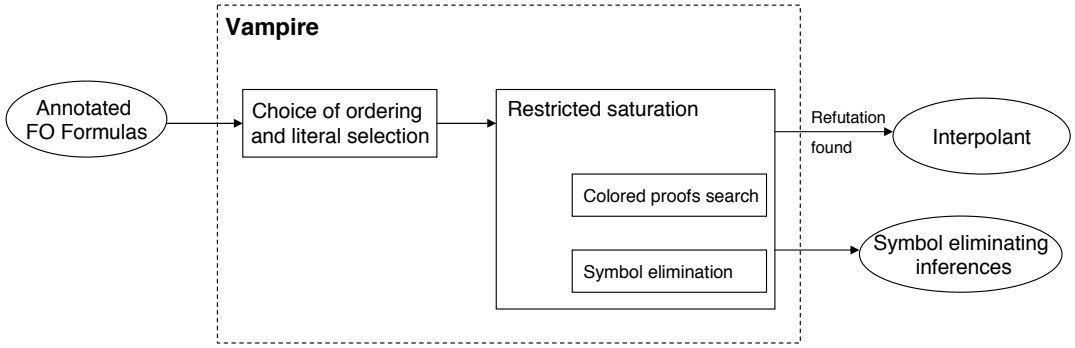

Figure 7.1: Interpolation and Symbol Elimination in Vampire.

or theories of arrays, can be handled by adding theory axioms to the first-order problem to be proved. Thus, interpolation in Vampire is not limited to decidable theories for which interpolation algorithms are known. One can use arbitrary first-order axioms. However, a consequence of this generality is that we do not guarantee finding interpolants even for decidable theories.

As far as we know, symbol elimination has not been implemented in any other system. A somehow related approach to symbol elimination is presented in [McM08, SG09] where theorem proving is used for inferring loop invariants. Contrarily to our approach, the cited works are adapted to prove given assertions as opposed to generating arbitrary invariants. Using the saturation-based theorem prover SPASS [WSH$^+$07], [McM08] generates interpolants as quantified invariants that are strong enough to prove given assertions. Paper [SG09] uses templates over predicate abstraction, and reduces the problem of invariant discovery to that of finding solutions, by the Z3 SMT solver [dMB08], for unknowns in an invariant template formula. Unlike [McM08, SG09], we automatically generate invariants as symbol eliminating inferences in full-first order logic, without using predefined predicate templates or assertions.

7.2 Colored Proofs, Symbol Elimination and Interpolation

Colored proofs are used in a context when some (predicate and/or function) symbols are declared to have colors. In colored proofs every inference can use symbols of at most one color, as a consequence, every term or atomic formula used in such proofs can use symbols of at most one color, too. We will call a symbol, term, clause etc.

colored if it uses a color, otherwise it is called *grey*.

In **symbol elimination** [KV09a] we are interested in inferences having at least one colored premise and a grey conclusion; such inferences are called *symbol-eliminating*. Conclusions of symbol-eliminating inferences can be used to find loop invariants. Symbol elimination can be reformulated as *consequence-finding*: we are trying to find grey consequences of a theory including both colored and grey formulas. Note that, unlike traditional applications of first-order theorem proving, we are not interested in finding a refutation: symbol-eliminating inferences can be obtained by running a theorem prover on a satisfiable formula, for which no refutation exists.

A formula I is called an **interpolant** of formulas L and R (with respect to a theory T) if the following conditions are satisfied:

(1) $T \vdash L \implies I$;

(2) $T \vdash I \implies R$;

(3) I uses only symbols occurring either in T or in both L and R.

Interpolation can be reformulated in terms of colors as follows: we assign one color to symbols occurring only in L and another color to symbols occurring only in R: then the last condition on interpolants can be reformulated as *I is grey*. For extracting interpolants from colored proofs we use the algorithm described in [KV09b].

The notion of interpolant has been changed in the model-checking community starting with [McM03]. Namely, the condition (2): $T \vdash I \implies R$ has been replaced by (2a): $T \vdash I \wedge R \implies \bot$. To avoid any confusion between the two notions of interpolant, in [KV09b] any formula I satisfying conditions (1), (2a) and (3) is called a *reverse interpolant of A and B*. Clearly, reverse interpolants for L and R are exactly interpolants of L and $\neg R$.

In the sequel, we reserve the notation L and R for the two formulas whose interpolant is to be computed.

7.3 Tool Overview

Vampire [RV02] is a general purpose first-order theorem prover based on the resolution and superposition calculus. To implement symbol elimination and interpolation in Vampire, we had to extend it by new functionalities, change the inference mechanism to be able to generate colored derivations, and implement an algorithm for extracting

```
vampire(symbol,function,a,0,left).
vampire(symbol,function,b,0,left).
vampire(symbol,predicate,q,1,left).
vampire(symbol,function,c,0,right).
vampire(option,show_interpolant,on).
```

```
vampire(left_formula).
  fof(a1,axiom, q(f(a))).
  fof(a1,axiom, ~q(f(b))).
vampire(end_formula).
vampire(right_formula).
  fof(a2,conjecture,
      ? [V] : (f(V)!=c)).
vampire(end_formula).
```

Figure 7.2: Specification of Interpolation.

interpolants. The workflow of interpolation and symbol elimination in Vampire is illustrated in Figure 7.1.

Annotated formulas. Vampire reads problems expressed in the TPTP syntax [Sut09]: a Prolog-like syntax allowing one to specify input axioms and conjecture for theorem provers. We had to extend the input syntax to make it rich enough to define colors and interpolation requests. In fact, we had to extend it even more since in the application of interpolation and symbol elimination the set of symbols that can occur in the interpolants is not necessarily the intersection of the languages of L and R with addition of theory symbols. We extended the TPTP syntax with Vampire-specific declarations. Their use is illustrated in Figure 7.2 and detailed in Example 7.3.1 taken from [McM08].

Example 7.3.1 [McM08] Consider the problem of computing an interpolant of $q(f(a))$ $\wedge\neg q(f(b))$ (i.e. L) and $\exists v(f(v) \neq c)$ (i.e. R).

The first three declarations shown in the left column of Figure 7.2 say that a, b are constants (function symbols of arity 0) and q is a unary predicate symbol colored in the "left" color (that is, in the language of L). Likewise, the fourth declaration in the left column says that c is a constant colored in the "right" color (that is, in the language of R). Finally, the left column contains an option that sets interpolant generation. This option can also be passed in the command line. The declarations `fof(..)` are TPTP declarations for introducing formulas. The vampire declarations `left_formula`, `right_formula` and `end_formula` are used to define L and R. If we have formulas not in the scope of the `left_formula` or `right_formula` declarations, they are considered as part of the theory T.

To use Vampire for symbol elimination, we can simply assign all symbols to be eliminated the left color and leave the right color unused. For concrete examples see `http:// www.vprover.org`.

```
3. ? [X0] : c != f(X0) [input]
4. ~? [X0] : c != f(X0) [negated conjecture 3]
6. ! [X0] : c = f(X0) [ennf transformation 4]
9. c = f(X0) (0:4) [cnf transformation 6]
10. f(X0) = f(X1) (1:5) [superposition 9,9]

1. q(f(a)) [input]
7. q(f(a)) (0:3) [cnf transformation 1]
2. ~q(f(b)) [input]
5. ~q(f(b)) [flattening 2]
8. ~q(f(b)) (0:3) [cnf transformation 5]
17. q(f(X0)) (2:3) [superposition 7,10]
18. ~q(f(X1)) (2:3) [superposition 8,10]
23. $false (2:0) [subsumption resolution 18,17]

Interpolant: ~! [X1,X0] : f(X0) = f(X1)
```

Figure 7.3: Proof and an interpolant output by Vampire.

Colored proof generation. In order to support the generation of colored proofs, the following had to be implemented.

1. We had to block inferences that have premises of two different colors.

2. We had to change the simplification ordering and literal selection, so that colored terms are larger than grey ones, and that (when possible) grey literals are selected only when there are no colored ones. To make colored terms bigger than grey, we had to implement the Knuth-Bendix ordering with ordinals as defined in [LW07] and make colored symbols to have the weight ω, while grey symbols to have finite weights.

To output the *conclusions of symbol eliminating inferences*, we check premises of each inference that produced a grey clause, and if one of the premises is colored, we output the resulting clause.

Interpolants are generated from refutations by the algorithm described in [KV09b]. For instance, given the input shown in Example 7.3.1, Vampire discovers an interpolant $\neg\forall x\forall y(f(x) = f(y))$. On Figure 7.3 we show the proof from which the interpolant was extracted. We reordered the proof steps to separate the reasoning done with different colors. The first block of steps uses the symbol c which belongs to the R language. Clause number 10 is a grey consequence of a symbol eliminating inference in this block, and is used in the second block as a premise of clauses 17 and 18. These in

Formulas	Coloring	Reverse Interpolant
L: $z<0\wedge x\leq z\wedge y\leq x$ R: $y\leq 0\wedge x+y\geq 0$	left: z right: -	$x<0$
L: $g(a)=c+5\wedge f(g(a))\geq c+1$ R: $h(b)=d+4\wedge d=c+1\wedge f(h(b))<c+1$	left: g,a right: h,b	$c+1\leq f(c+5)$
L: $p\leq c\wedge c\leq q\wedge f(c)=1$ R: $q\leq d\wedge d\leq p\wedge f(d)=0$	left: c right: d	$p\leq q\wedge(q>p\vee f(p)=1)$
L: $f(x_1)+x_2=x_3\wedge f(y_1)+y_2=y_3\wedge y_1\leq x_1$ R: $x_2=g(b)\wedge y_2=g(b)\wedge x_1\leq y_1\wedge x_3<y_3$	left: f right: g,b	$x_1>y_1\vee x_2\neq y_2\vee x_3=y_3$
L: $c_2=car(c_1)\wedge c_3=cdr(c_1)\wedge\neg(atom(c_1))$ R: $\neg(c_1)=cons(c_2,c_3)$	left: $car,cons$ right: -	$\neg atom(c_1)\wedge c_1=cons(c_2,c_3)$
L: $Q(f(a))\wedge\neq Q(f(b))$ R: $f(V)=c$	left: Q,a,b right: c	$\exists x,y: f(x)\neq f(y)$
L: $a=c\wedge f(c)=a$ R: $c=b\wedge b\neq f(c)$	left: a right: b	$c=f(c)$
L: $True\wedge a'[x']=y\wedge x'=x\wedge y'=y+1\wedge z'=x'$ R: $\neg(a'[z']=y'-1)$	left: x,y right: -	$1+a'[x']=y'\wedge x'=z'$

Table 7.1: Interpolation with Vampire.

the next step produce a contradiction. In accordance with the algorithm for extracting interpolants given in [KV09b], the output interpolant $\neg\forall x\forall y(f(x)=f(y))$ is the negation of the clause 10.

Symbol Elimination. To make Vampire output conclusions of symbol-eliminating inferences, one should set the option `show_symbol_elimination` to `on`. As Vampire is not supposed to terminate in the symbol-eliminating mode, it is wise to specify a time limit when it is run in this mode.

Figure 7.4 shows part of the symbol eliminating problem for the `Partition` program on of Table 7.2. We have replaced the TPTP arithmetic syntax by infix notation for better readability. Formula `iter_def` defines the `iter` predicate that holds for the loop counter values. Formulas `path1` and `path2` represent the transition in the two possible paths through the loop body and `ineq_i_j` expresses an invariant on the variables *i* and *j* that was discovered already by static analysis.[2]

The invariant generation outputs many formulas, among them are the following four:

```
fof(inv68, claim, ![X0,X1,X2]: (X0 != X1 | aa(sK2(X0)) != X2 | 0 > X0 |
                                 j <= X0 | bb(X1) = X2)).
fof(inv111, claim, sK2(0) > -1 | j = 0).
```

[2]The full examples in the TPTP format are available at `http://www.vprover.org/symbol_elimination_examples.zip`.

```
fof(inv128, claim, ![X19,X20]: (X19 <= X20 | X20+1 <= sK2(X19) |
                                j <= X19 | X19 <= 0)).
fof(inv192, claim, ![X21,X22]: (j <= X21 | i > X22 | X22 > sK2(X21) |
                                0 > X21)).
```

We will show that these formulas imply the desired invariant $\forall x : 0 \leq x < j \implies \exists y : 0 \leq y < i \ \wedge \ bb[x] = aa[y]$. These formulas can be simplified into

$$\forall x : 0 \leq x < j \implies bb[x] = aa[sK2(x)]$$

$$j \neq 0 \implies sK2(0) \geq 0$$

$$\forall x : 0 < x < j \implies sK2(x) \geq x$$

$$\forall x : 0 \leq x < j \implies i > sK2(x)$$

The last three formulas imply the following bounds on the $sK2$ function:

$$\forall x : 0 \leq x < j \implies 0 \leq sK2(x) < i$$

which in turn can be combined with the first formula into

$$\forall x : 0 \leq x < j \implies 0 \leq sK2(x) < i \ \wedge \ bb[x] = aa[sK2(x)]$$

$sK2$ is a Skolem function introduced in place of an existential quantifiers by Vampire during the problem clausification. We can replace the Skolem function back by an existential quantifier and obtain the desired invariant

$$\forall x : 0 \leq x < j \implies \exists y : 0 \leq y < i \ \wedge \ bb[x] = aa[y]$$

7.4 Experiments

We have successfully ran Vampire on benchmark examples taken from recent literature on interpolation and invariant generation. In this section we present two different sets of experimental results that underline the effectiveness of our implementation. The reported results were obtained on a machine with 2 GHz processor and 2GB of RAM.

```
fof(iter_def,hypothesis,
    ! [I] : (iter(I) <=> (I>=0 & n>I))).
fof(path1,hypothesis,
    ! [I] : ( iter(I) =>
               (
                (aa(i(I))>=0 & m>i(I) )
                =>
                ( bb(I+1,j(I)) = aa(i(I)) &
                  j(I+1) = j(I)+1 &
                  i(I+1)= i(I)+1 &
                  k(I+1) = k(I)
                ))))).
fof(path2,hypothesis,
    ! [I] : ( iter(I) =>
               (
                (aa(i(I))>=0 & m>i(I) )
                =>
                ( cc(I+1,k(I)) = aa(i(I)) &
                  k(I+1) = k(I)+1 &
                  i(I+1)= i(I)+1 &
                  j(I+1) = j(I)
                ))))).
fof(ineq_i_j,hypothesis,
  ![I]: ( iter(I) => i(I)>=j(I))).
```

Figure 7.4: Part of the first-order symbol elimination problem for the `Partition` program of Table 7.2.

Interpolation. Table 7.1 summarises some of our results for computing reverse interpolants on examples that have been used as motivating examples by previous techniques [JM05, YM05, RSS07, McM08]. The first column of Table 7.1 presents the input formulas L and R whose interpolants is going to be computed. The second column shows symbols declared colored, whereas the third column shows the reverse interpolant generated by Vampire.

All interpolants given in Table 7.1 were computed by Vampire in essentially no time (e.g. in less than 0.1 second). In the first four examples of Table 7.1 a simple axiomatisation of arithmetic with the greater-than relation and successor function was used. The fifth example of Table 7.1 uses the theory of lists, whereas the last example of Table 7.1 uses the combined theory of arrays and arithmetic.

The last example of Table 7.1 originates from an example taken from [JM05], and is a request to prove the infeasibility of the following one-path program annotated by a pre- and a post-condition:

$$\{\top\} \quad a[x] := y; y := y+1; z := x \quad \{a[z] \neq y-1\}.$$

Let x,y,z,a denote the initial and x',y',z',a' the final values of program variables. Based on the bounded-model checking approach [McM03], proving infeasibility of the above program path boils down to computing a reverse interpolant for the formulas $\top \wedge T(\{x,y,z,a\},\ \{x',y',z',a'\})$ and $a'[z'] \neq y'-1$, where $T(\{x,y,z,a\},\{x',y',z',a'\}) \equiv a'[x'] = y \wedge x' = x \wedge y' = y+1 \wedge z' = x'$ is the transition relation defined by the program. The reverse interpolant computed by Vampire proves that the program has no feasible path from the initial state to the final state.

Symbol Elimination. Experiments with symbol elimination on array programs taken from [GT06, JM07a] are summarised in Table 7.2. We ran Vampire in the symbol elimination mode with a time limit of 10 seconds. We recall that each conclusion of a symbol-eliminating inference is a loop invariant [KV09a].

For all examples of Table 7.2 we show in the rightmost column a desired invariant that could be computed using other techniques. We were interested in the following: (i) can Vampire generate the invariant itself and if not, (ii) can Vampire generate invariants that would imply the desired invariant?

After running Vampire in the symbol-eliminating mode we sometimes obtain a large set of invariants. The number of invariants (that is, the number of symbol-eliminating inferences) is shown in the second column of the Table 7.2. To make them

usable we did the following minimization: remove invariants that are implied by the theory axioms or by other invariants. For the task we used Vampire itself. Obviously, the problem whether an invariant is implied by other invariants, is undecidable, so we ran Vampire with a time limit of 0.3 seconds once for each invariant, trying to prove it from the remaining invariants and the theory axioms. The number of invariants that could not be proved redundant is shown in the third column of Table 7.2.

7.5 Conclusion

We described how interpolant generation and symbol elimination are implemented and can be used in the first-order theorem prover Vampire. Future work includes integrating Vampire into software verification tools for automatically generating interpolants and supporting the entire process of verification. Inferring a minimal set of invariants and improving these invariants can also be done using theorem proving and remains an interesting topic for further research.

Loop	♯ of SEI	♯ of Min SEI	SEI as Invariant

`Initialisation` [JM07a]
$i = 0;$
while $(i < m)$ do
$aa[i] = 0; i = i+1$
end do
	399	15	$\forall x : 0 \leq x < i \implies aa[x] = 0$

`Copy` [JM07a]
$i = 0;$
while $(i < m)$ do
$bb[i] = aa[i]; i = i+1$
end do
	379	14	$\forall x : \quad 0 \leq x < i \implies$
			$bb[x] = aa[x]$

`Vararg` [JM07a]
$i = 0;$
while $(aa[i] > 0)$ do
$i = i+1$
end do
	1	1	$\forall x : 0 \leq x < i \implies aa[x] > 0$

`Partition` [GT06]
$i = 0; j = 0; k = 0;$
while $(i < m)$ do
if $(aa[i] >= 0)$
then $bb[j] = aa[i]; j = j+1$
else $cc[k] = aa[i]; k = k+1$
end if;
$i = i+1$
end do
	150	61	$\forall x : \quad 0 \leq x < j \implies$
			$\exists y : 0 \leq y < i \ \wedge$
			$bb[x] = aa[y]$

`Partition_Init` [JM07a]
$i = 0; k = 0;$
while $(i < m)$ do
if $(aa[i] == bb[i])$
then $cc[k] = i; k = k+1$
end if;
$i = i+1$
end do
	18	13	$\forall x : \quad 0 \leq x < k \wedge 0 \leq x < i$
			$\implies$
			$aa(cc(x)) = bb(cc(x))$

Table 7.2: Symbol Elimination with Vampire on Array Programs.

Chapter 8

Case Studies on Invariant Generation Using a Saturation Theorem Prover

Authors: Krystof Hoder, Laura Kovacs, Andrei Voronkov

Automatic understanding of the intended meaning of computer programs is a very hard problem, requiring intelligence and reasoning. In this paper we evaluate a program analysis method, called symbol elimination, that uses first-order theorem proving techniques to automatically discover non-trivial program properties. We discuss implementation details of the method, present experimental results, and discuss the relation of the program properties obtained by our implementation and the intended meaning of the programs used in the experiments.

8.1 Introduction

The complexity of computer systems grows exponentially. Many companies and organisations are now routinely dealing with software comprising several millions lines of code, written by different people using different languages, tools and styles. This software is hard to understand and is integrated in an ever changing complex environment, using computers, networking, various physical devices, security protocols and many other components. Ensuring the reliability of such systems for safety-critical applications is extremely difficult. One way of solving the problem is to analyse or verify these systems using computer-aided tools based on computational logic.

In [KV09a] a new method, called *symbol elimination*, has been proposed to automatically generate statements expressing computer program properties. The approach requires no preliminary knowledge about program behavior, but uses the power of a

saturation theorem prover to derive and understand the *intended meaning* of the program. To undertake such a complex task, reasoning in the combination of first order logic and various theories is required as program components involve both bounded and unbounded data structures.

One can argue that automatic inference of program properties is a hard and creative problem whose solution improves our understanding of the relation between the computer reasoning and the human reasoning. Indeed, given a computer program, one can ask a computer programmer questions like "what are the essential properties of this program" or "what is the intended meaning of this program?" Answering such question requires intelligence. If the program is small and not highly sophisticated, one can expect that the programmer will be able to give some answers. For example, if the program copies one array into another, one can expect the programmer to say that the intended meaning of the program is to copy arrays and that among the most essential properties of this program are the facts that the two arrays will be equal and the first array will not be modified. These two properties are first-order properties, that is, they can be expressed in first-order logic.

The properties generated by the symbol elimination method are first-order properties, therefore one can ask a question of whether a computer program can generate such properties and whether it can generate "the intended" properties (whatever it means). This paper tries to answer this fundamental question by taking a program annotated by humans, removing these annotations, generating program properties by a computer program, and comparing the generated program properties with the intended properties.

The first implementation of symbol elimination was carried out in the first-order resolution theorem prover Vampire [HKV10]. This implementation could be used for symbol elimination, yet not for invariant generation. It was run on an array partitioning program to demonstrate that it can generate complex loop invariants with alternating quantifiers, which could not be generated by any other existing technique. Such complex loop properties precisely capture the intended first-order meaning of the program, as well as the programmer's intention while writing the program.

However, the practical power of symbol elimination was not clear since it required extensive experiments on programs containing loops. Such experiments were not easy to organise since they involved combining several tools for program analysis and theorem proving. Designing such a combination turned out to be non-trivial and error-prone. The first standalone implementation of program analysis in the theorem prover

Vampire is described in the system abstract [HKV11b]. Experimental evaluation of the method was however not yet carried out.

This paper undertakes the first extensive investigation into understanding the power and limitations of symbol elimination for invariant generation. In addition to the fundamental AI problems mentioned above, we were also interested in the power of the method for applications in program analysis, which can be measured by the following characteristics:

1. [Strength] Is the method powerful enough to infer automatically invariants that would imply intended loop properties?

2. [Time] Is invariant generation fast enough?

3. [Quantity] What is the number of generated invariants?

The method we use to answer these questions in this work is described below.

- ▷ We use the invariant generation framework of [HKV11b] implemented directly in Vampire. Vampire can now accept as an input a program written in a subset of C, find all loops in it, and generate and output invariants for each of the loops. To this end, to make [HKV11b] practically useful, we extended the program analyser of [HKV11b] with a C parser.

- ▷ We took a number of annotated C programs coming from an industrial software verification project, as well as several standard benchmarks circulated in the verification community (for example, [GRS05, SG09]), and removed annotations corresponding to loop invariants.

- ▷ We ran Vampire with various time limits and collected statistics relevant to the questions raised above.

The main contribution of this paper is an experimental evaluation of the symbol elimination approach, providing statistics and better understanding of the power of the method.

Our experiments show that, at least for small (but far from trivial) programs a computer program using the symbol elimination method turns out to generate in less than a second properties that imply the annotated properties, that is, the intended properties. Hence, symbol elimination confirms, in some way, the power of deductive methods in computer reasoning. However, even for relatively simple programs it also generates

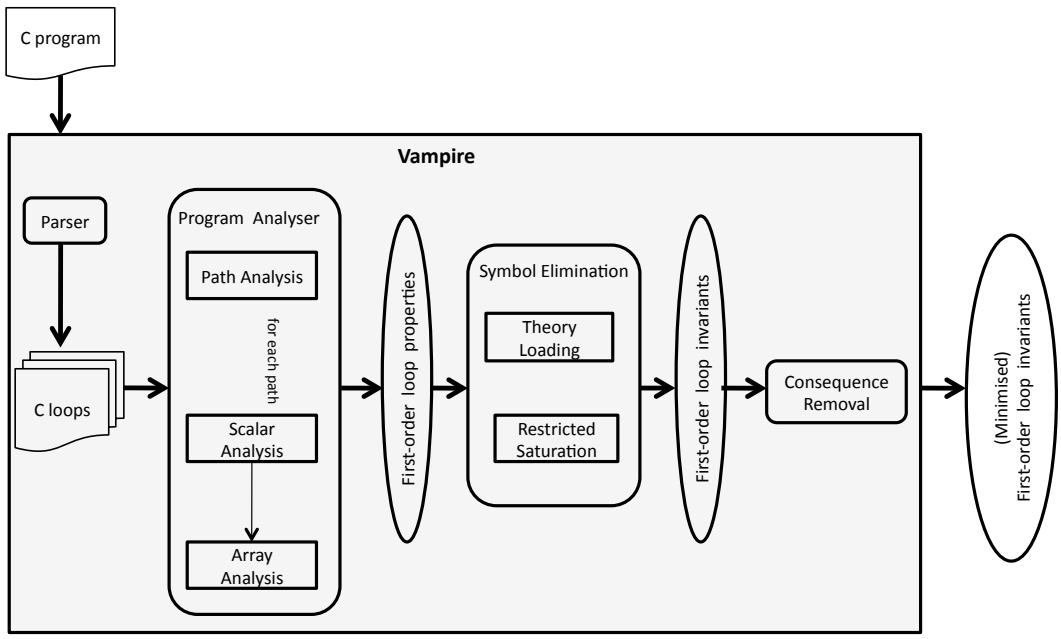

Figure 8.1: Invariant Generation in Vampire.

many other properties, which shows that further investigation may be required to bring the human and the computer understanding of the meaning of programs closer.

This paper is structured as follows. Section 8.3.1 overviews the program analysis framework of Vampire. Given a program in a subset of C as an input, Vampire now *automatically* generates a set of invariants for the loops occurring in the program.

Reasoning about programs requires reasoning in combined first-order theories, such as arithmetic, uninterpreted functions, arrays, etc. Since invariant generation by symbol elimination requires both theories and quantifiers, efficient handling of theories in Vampire was a major issue for us. We describe theory reasoning in Vampire in Section 8.3.2.

Since loop invariants are to be used in proving program properties, it is important that the generated set of invariants is not too large, yet powerful enough to imply important program properties. A framework for removing invariants implied by other invariants was introduced in [HKV11b], where invariants are removed in combination with invariant generation. In our work we use [HKV11b], as described in Sections 8.3.3 and 8.3.4. We further give an experimental evidence that such a removal is in practice "cheap" as compared to invariant generation (Section 8.4).

The "quality" of a set of automatically generated invariants refers to whether it can be used to easily derive program properties required for program verification or

static analysis. We discuss the quality of invariants generated by our technique in Section 8.3.5.

The key section of this paper is Section 8.4 on experimental evaluation of the symbol elimination method. The reported experimental results provide empirical evidence of the power of symbol elimination for generating complex invariant.

We briefly consider related work in Section 8.5 and draw conclusions in Section 8.6.

8.2 Preliminaries

Programs, Variables and Expressions. Figure 8.1 illustrates our approach to practical invariant generation in Vampire. Figure 8.1 extends [HKV11b] by the use of a program parser. Hence, Vampire now can handle C programs over integers and arrays using assignments, loops and if-then-else conditional statements with standard (C language) semantics. Non-integer variables can also be considered but regarded as uninterpreted.

We assume that the language expressions contain integer constants, variables, and some function and predicate symbols. The standard arithmetical functions, such as $+$, $-$, $\cdot$ are considered as interpreted, while all other function symbols are uninterpreted. Likewise, the arithmetical predicate symbols $=$, $\neq$, $\leq$, $\geq$, $<$ and $>$ are interpreted while all other predicate symbols are uninterpreted. As usual, the expression $A[e]$ is used to denote the element of an array A at the position given by the expression e. The current version of Vampire does not handle nested loops. If the input program contains such loops, only the innermost loops will be analysed.

The programming model for invariant generation in Vampire is summarized below.

$$\textsf{while } b \textsf{ do } \quad s_1; s_2; \ldots; s_m \quad \textsf{end do} \tag{8.1}$$

where b is a boolean expression, and statements s_i ($i = 1, \ldots, m$) are either assignments or (nested) if-then-else conditional statements. Some example loops for invariant generation in Vampire are given in the leftmost column of Table 8.3. Throughout this paper, integer program variables are denoted by lower-case letters $a, b, c, \ldots$, whereas array variables are denoted by upper-case letters $A, B, C, \ldots$.

Theorem proving and Vampire. The symbol elimination method described and investigated in this paper is essentially based on the use of a first-order theorem prover.

Moreover, one needs a theorem prover able to deal with theories (e.g. integer arithmetic), first-order logic, and generate consequences. Theorem provers using saturation algorithms are ideal for consequence generation. Saturation actually means that the theorem prover tries to generate all, in some sense, consequences of a given set of formulas in some inference system, for example, resolution and superposition [RV01b].

In reality, saturation theorem provers use a powerful concept of *redundancy elimination*. Redundancy elimination is not an obstacle to consequence generation, since redundant formulas are logical consequences of other formulas the theorem prover is dealing with.

All our experiments described in this paper are conducted using Vampire [RV02], which is a resolution and superposition theorem prover running saturation algorithms. Vampire is available from http://www.vprover.org.

8.3 Symbol Elimination and Invariant Generation in Vampire

In a nutshell, the symbol elimination method of [KV09a] works as follows. One is given a loop L, which may contain both scalar and array variables.

(1) In the *first phase*, symbol elimination tries to *generate as many loop properties as possible*. This may sound as solving a hard problem using an even harder problem, yet the method undertakes an easy way. First, it considers all (scalar and array) variables of L as functions of the *loop counter* n. This means that for every scalar variable a, a function $a(n)$ denoting the value of a at the loop iteration n is introduced. Likewise, for every array variable A a function $A(n, p)$ is introduced, denoting the value of A at the iteration n in the position (or index) p. Thus, the language of loop is extended by new function symbols, obtaining a new, extended language. Note that some loop properties in the new language are easy to extract from the loop body, for example, one can easily write down a formula describing the values of all loop variables at the iteration $n+1$ in terms of their values at an iteration n, by using the transition relation of the loop. In addition to the transition relation, some properties of *counters* (scalar variables that are only incremented or decremented by constant values in L) are also added. Further, the loop language is also extended by the so-called *update predicates for arrays* and their properties are added to the extended language. An update predicate for an array A essentially expresses updates made to A and their effect on the final value of A. After this step, a collection Π of valid loop properties expressed in the

extended language is derived.

(2) Formulas in Π cannot be used as loop invariants, since they use symbols not occurring in the loop, and even symbols whose semantics is described by the loop itself. These formulas, being valid properties of L, have a useful property: all their consequences are valid loop properties too. The *second* phase of symbol elimination tries to *generate logical consequences of* Π *in the original language of the loop*. Any such consequence is also a valid property of L, and hence an invariant of L. Logical consequences of Π are generated by running a saturation theorem prover on Π in a way that the theorem prover tries to eliminate the newly introduced symbols.

(3) The *third phase* of the method, added recently to symbol elimination [HKV11b], tries to remove invariants implied by other generated invariants.

It is important to note that (the first phase of) symbol elimination can be combined with other methods of program analysis. Indeed, any valid program property can be added to Π, resulting hopefully in a stronger set of invariants.

The rest of this section describes the details of how these three phases are implemented in Vampire.

8.3.1 Program Analysis in Vampire

In this section we briefly overview the program analysis phase of invariant generation in Vampire, introduced in [HKV11b].

The analyser works with simple programs of the form described in Section 8.2, and generates loop properties for the first phase of symbol elimination. The analyser works as follows. First, it extracts from the program all non-nested loops and performs the following steps on every such loop.

1. Find all loop variables and classify them into variables updated by the loop and constant variables.

2. Find counters, that is, updated scalar variables that are only incremented or decremented by constant values. Save properties of counters.

3. Generate update predicates of updated array variables and save their properties.

4. Save the formulas corresponding to the transition relation of the loop.

5. Create a symbol elimination task for Vampire by putting together all saved formulas and marking symbols that have to be eliminated.

```
Loops found: 1
Analyzing loop...
---------------------
while (a < m)
  {
    if (A[a] >= 0)
      {
        B[b] = A[a];
        b = b + 1;
      }
    else
      {
        C[c] = A[a];
        c = c + 1;
      }
    a = a + 1;
  }
---------------------
Variable: B: (updated)
Variable: a: (updated)
Variable: b: (updated)
Variable: m: constant
Variable: A: constant
Variable: C: (updated)
Variable: c: (updated)
```

```
Counter: a
Counter: b
Counter: c
Path:
  false: A[a] >= 0
  C[c] = A[a];
  c = c + 1;
  a = a + 1;
Path:
  true: A[a] >= 0
  B[b] = A[a];
  b = b + 1;
  a = a + 1;
Counter a: 1 min, 1 max, 1 gcd
Counter b: 0 min, 1 max, 1 gcd
Counter c: 0 min, 1 max, 1 gcd
8. ![X1,X0,X3]:(iter(X0) & iter(X1) & X1>X0 & c(X1)>X3 & X3>c(X0))
   => ?[X2]:(iter(X2) & c(X2)=X3 & X2>X0 & X1>X2) [program analysis]
7. ![X0]:(iter(X0) => c(X0)>=c0) [program analysis]
6. ![X0]:(iter(X0) => c(X0)<=c0+X0) [program analysis]
5. ![X1,X0,X3]:(iter(X0) & iter(X1) & X1>X0 & b(X1)>X3 & X3>b(X0))
   => ?[X2]:(iter(X2) & b(X2)=X3 & X2>X0 & X1>X2) [program analysis]
4. ![X0]:(iter(X0) => b(X0)>=b0) [program analysis]
3. ![X0]:(iter(X0) => b(X0)<=b0+X0) [program analysis]
2. ![X0]:(iter(X0) => a(X0)=a0+X0) [program analysis]
1. ![X0]:(iter(X0) <=> (0<=X0 & X0<n)) [program analysis]
```

Figure 8.2: Partial output of Vampire's program analysis on the `Partition` program of Table 8.3.

Example 8.3.1 Figure 8.2 shows the output of Vampire corresponding to the first four steps of the program analysing process for the `Partition` program of Table 8.3. In the output, `![X]` (respectively `?[X]`) denotes $\forall$`X` (respectively, $\exists$`X`). The full example contains, apart from the loop shown in Figure 8.2, initialisation of some loop variables.

As shown in Figure 8.2, the program analyser of Vampire detects that variables `B`, `A`, `a`, `b`, `c` are updated in the loop, variables `A`, `m` are constants, and the variables `a`, `b`, `c` are counters.

Next, the path analysis of the program analyser reports that there are two program paths (listed in two blocks starting with `Path`), depending on the value of the test `A[a]` $\geq$ `0`. The values `min` and `max` denote the maximal and the minimal increment of the counter over all paths in the program. The value `gcd` is the greatest common divisor of all such increments.

Formula on line denoted as 1 defines a predicate for loop iteration counter. Properties of counters are further generated on lines enumerated by 2-8 in the Figure 8.2. For example, Vampire generates axiom 2 expressing the following property: for *every loop iteration `X0`*, the value `a(X0)` of `a` at iteration `X0` is given by `a0+X0`, where `a0` denotes the initial value of `a`.

8.3.2 Theory Reasoning in Vampire

Standard resolution and superposition theorem provers are good in dealing with quantifiers but lack any support for theories, such as those of integers, reals, arrays, lists, etc. The standard way of adding a theory to such a theorem prover is by adding a first-order axiomatisation of the theory. There is no complete axiomatisation for all the above mentioned theories (assuming arbitrary quantifiers).

Adding incomplete axiomatisations is the approach used in [KV09a]. The new version of Vampire [HKV11b] came further than [KV09a] and also added integers as a data type in Vampire. The method of [HKV11b] is also the approach we follow in this paper. This means that integers can be used directly instead of representing them using, for example, zero and the successor function. Vampire "knows" several standard predicates and functions on integers: addition, subtraction, multiplication, successor, division, and standard inequality relations such as $\leq$. Since Vampire's users may not know much about combining arithmetic and first-order logic, automatic loading of relevant theories is taken care of by Vampire. For example, if the user uses the standard integer addition function symbol $+$, then Vampire will automatically add an axiomatisation of integer linear arithmetic including axioms for $+$.

Generally, for loading existing theory axiomatisations of Vampire, the user should add to the input (in the TPTP syntax) a Vampire-specific declaration binding an input symbol to an interpreted theory symbol. For example, one can write:

```
vampire(interpreted_symbol, geq, integer_greater_equal).
```

to declare that the input symbol `geq` denotes the inequality $\geq$ on integers. Given the above declaration, Vampire will add some theory axioms for this symbol to the input formulas. The user can also choose to use her own axiomatisation or to add more axioms to the axiomatisation loaded by Vampire.

The results reported in Section 8.4 show that Vampire's approach to reasoning with integers is good enough for proving properties of simple loops. However, the research into various approaches to reasoning with quantifiers and theories and their relative strength is still in its infancy and hindered by a lack of publicly available benchmarks.

8.3.3 Symbol Elimination in Vampire

If one just adds a collection of formulas obtained by program analysis to a theorem prover and expects the prover to generate consequences of these formulas using only a

given subset of functions and predicates from the input, the result will most likely be disappointing. For example, suppose that p is a symbol that cannot occur in invariants, while q, r are symbols that can occur in them. Suppose that we are also given two clauses $p \vee q$ and $\neg p \vee r$. A theorem prover may decide to derive the invariant $q \vee r$ from these two clauses but may also decide not to do anything with them, depending on the term ordering and literal selection it uses (e.g. q and r might be selected before p).

The main ingredient of the symbol elimination technique in Vampire is the concept of a *well-colored derivation* [McM08, HKV10]. To define well-colored derivations (also called local proofs or split proofs) some predicate and/or function symbols are declared to have colors, while other symbols are uncolored. A symbol, term, literal or formula using a color are called colored, otherwise they are called grey. A derivation is called *well-colored* if any inference can use symbols of at most one color. Any inference having at least one colored premise and a grey conclusion is called a *symbol eliminating inference*.

Following the symbol elimination approach of [KV09a], loop invariant generation can be thus be addressed using colors, as follows. One and the same color is assigned to all additional symbols introduced for formulating properties of loops (see Section 8.3), for example, the loop counter. All other symbols, that is the loop variables and the theory symbols, are grey. A loop invariant is then a grey formula describing a valid loop property. Since one is guaranteed that any grey consequence of the input set of formulas is a valid loop property, the problem of invariant generation reduces to the problem of generating grey consequences of this set. This means, in a way, that the colored symbols should be eliminated.

To make saturation more effective for deriving grey consequences, the Knuth-Bendix term ordering used in Vampire was modified in [HKV10], so that symbol weights are infinite ordinals and any colored ground term or atom is greater than any ground grey term or atom.

Example 8.3.2 Consider the `Partition` program from Table 8.3. As presented in Figure 8.2, the program analyser of Vampire generates the following valid loop property:

`a(X0)=a0+X0` for every loop iteration `X0`.

However, this property cannot be used as an invariant as it makes use of the additional unary function symbol `a(X0)` denoting the value of the loop variable `a` at an iteration `X0`. However, since we declare the unary symbol `a` colored, Vampire tries to eliminate `a(X0)` from the set of valid properties generated by its program analyser, and derives,

for example, the following grey formula by a symbol eliminating inference: [1]

$\texttt{a-a0} \geq \texttt{0}$, where the constant a denotes the value of a at the end of the loop.

This property is a loop invariant, as it uses only the grey symbols. Similarly,

$\texttt{B(X0,b(X0))} \geq \texttt{0}$ for every loop iteration X0

is a valid loop property, but not an invariant. However, by making the unary symbol b and the binary symbol B colored, Vampire generates the following invariant from this and other loop properties:

$$\texttt{0} \leq \texttt{X} < \texttt{b} \implies \texttt{B(X)} \geq \texttt{0},$$

where the unary symbol B and the constant b are the corresponding grey symbols denoting the final values of loop variables with the same names.

8.3.4 Pruning Generated Invariants

Symbol elimination can generate invariants implied by other generated invariants. For example, any inference applied to two invariants gives an invariant. For this reason, [HKV10] only stores invariants obtained by an inference having at least one colored premise, that is, a symbol-eliminating inference. However, even among conclusions of symbol-eliminating inferences there are typically many invariants implied by others.

To improve invariant generation, a new mode, called the *consequence-elimination mode*, was added to Vampire in [HKV11b]. In this mode, Vampire obtains a set S of clauses (i.e. invariants) as an input and tries to find its proper subset S' equivalent to S. In the process of computing S', Vampire is run with a small time limit. Naturally, one is interested in having S' as small as possible.

In the experiments reported in this paper, we made use of [HKV11b] and ran Vampire in the consequence elimination mode using four different strategies with a 20 seconds time limit. Our experimental results show that typically between 80% and 90% of all invariants obtained by symbol elimination are redundant, and hence discarded. It is usually the case that all or nearly all of the discarded invariants are discovered in a few milliseconds already by the first strategy.

[1] As described in [HKV10], for every program variable v two grey variables $v0$ and v are used, denoting the initial and the final values of this loop variable. Further, a colored unary function symbol v is introduced, such that $v(X)$ denotes the value of the loop variable v at iteration X.

Loop	Symbol Elimination within 1s					Symbol Elimination within 10s				
	♯ SEI	♯ Min SEI	% Redundancy	∀-Inv	∀∃-Inv	♯ SEI	♯ Min SEI	% Redundancy	∀-Inv	∀∃-Inv
`Initialisation`	15	5	67%	yes	no	40	5	88%	yes	no
`Copy`	24	5	79%	yes	no	25	5	80%	yes	no
`Find`	151	13	91%	yes	no	474	21	96%	yes	no
`Vararg`	1	1	0%	yes	no	1	1	0%	yes	no
`Partition`	166	38	77%	yes	yes	849	59	93%	yes	yes
`Partition_Init`	168	24	86%	yes	yes	692	127	82%	yes	yes
`Shift`	41	12	71%	yes	no	111	16	86%	yes	no

Table 8.1: Symbol elimination on programs from [GRS05, SG09], by running Vampire with 1 and 10 seconds of time limit.

8.3.5 Proving Invariants, Postconditions, and Assertions

The set of invariants and loop properties resulting from Vampire's program analysis and symbol elimination can be used to prove loop properties. Note that proving a loop property can be done in at least two different ways.

(a) First, we can add the negation of the property to the formulas obtained by program analysis and try to prove that the resulting set of formulas is unsatisfiable. It is easier than invariant generation since one does not have to take care of colors and can use arbitrary proofs, ordering and strategies, including goal-oriented ones.

(b) Second, one can prove that the property is an inductive invariant, which is a much simpler problem and can be reduced to proving a few formulas with respect to the theory.

However, both approaches assume that every loop is already annotated. Providing such annotations manually requires a considerable amount of work by highly qualified persons and thus often makes verification prohibitively expensive. Therefore, generation of invariants without using any annotations is invaluable in making verification and static analysis of programs economically feasible.

Evaluating the quality of an invariant generation technique is not easy since it cannot be measured using simple measures, such as the number of generated invariants or the speed of their generation. One can say that such a technique is powerful, if the set of invariants generated in small time implies the *intended* loop invariants, or invariants that humans would use to annotate this loop for verification purposes.

To evaluate our method, we used annotated code both from academic benchmarks and an industrial verification project. In this code every loop was *annotated by its intended property* in the form of loop invariants and/or postconditions. So we ran Vampire on the code as follows.

1. First, we generated a set of invariants using symbol elimination, as described in

Loop Shape	♯ of Loops	Average ♯ of SEI	Average ♯ of Non-Redundant SEI	% of SEI Redundancy	∀-Inv	∀∃-Inv
Simple	33	168	18	89.3%	yes	no
Multi-path	5	340	46	86.4%	yes	yes

Table 8.2: Symbol elimination on programs sent by Dassault Aviation.

Sections 8.3.1–8.3.4;

2. Then we checked, also using Vampire, whether the intended loop property is implied by this set of invariants and whether the intended property described a postcondition, if the latter was provided.

Example 8.3.3 In the fourth column of Table 8.3, we show one of the intended invariant of the `Partition` program. This invariant follows from the two invariants generated by Vampire, which are presented in the fifth column of the table.

8.4 Experimental Results

The experiments described in this section were carried out using two benchmark suites. One is a collection of 6 loops taken from various research papers (Tables 8.3 and 8.1). The other one is a collection of 38 loops taken from programs provided by Dassault Aviation. We used a computer with a 2 GHz Intel Core i7 CPU processor and 4GB RAM, and ran experiments using the Vampire version 0.6. The consequence elimination phase of Vampire was run with a 20 seconds time limit.

To analyse C programs, we had to extend Vampire by a C parser. Inputs to the parser are (large) C programs. After parsing, Vampire finds all loops in the program and checks, for each loop, if it is as given in (8.1) and thus can be analysed. Vampire outputs a set of loop invariants for each loop (8.1) under analysis. Figure 8.1 illustrates the invariant generation process within Vampire.

To use Vampire for generating invariants of arbitrary C code, one should use it in the program analysis mode as follows:

```
vampire --mode program_analysis < filename.c
```

8.4.1 Challenging Benchmarks

Table 8.3 describes the effect of symbol elimination on 6 programs. The names of the first 5 programs and their origins (that is, the papers where they were described) are given in column 1. The program given in the last row of Table 8.3 is taken from

our own case studies, and illustrates the possibility of generating invariants for loops using read-*and*-write arrays. Columns 2 contains the number of invariants generated in 1 second, while column 3 the number of invariants that remain after consequence elimination.

Column 4 contains the intended invariant (or the invariant of interest): for this invariant we checked whether it is implied by invariants generated by Vampire, and if yes, show (in column 5) the subset of the generated invariants that imply the intended one. Checking that the intended invariant follows from the generated invariants was done using Vampire, so clauses in column 5 are those extracted from the corresponding Vampire proof. Invariants are generated by Vampire in a certain order. We used this order to enumerate clauses in column 5, since it gives the reader an idea how fast the required invariants were found. For example, `inv81` for the `Partition` example means that this invariant was the 82nd generated invariant (counting invariants starts from 0). Note that some of the formulas in this column have skolem functions introduced by Vampire's clausifier. For example, sk_1 denotes a skolem function. They can be de-skolemised to give invariants with quantifier alternations.

For this benchmark suite, *all the intended invariants turned out to be logical consequences of the invariants generated by Vampire*. However, one of the intended invariants could not be proved by Vampire. Namely, for the `Shift` example, Vampire generated the invariant $\forall x(x \geq 0 \wedge x < a \implies A[x] = A[x+1])$, while the intended invariant was $\forall x(x \geq 0 \wedge x \leq a \implies A[x] = A[0])$. These two invariants are equivalent in arithmetic, however, to prove the intended one from the generated one in first-order logic one needs induction. By adding a simple induction axiom, specific to the `Shift` example, we could also prove the intended invariant.

We were also interested in checking the number of generated invariants depending on the time spent on invariant generation. Table 8.1 contains statistics about invariant generation with 1 and 10 seconds time limits on symbol elimination, respectively. It also shows the percentage of generated invariants shown to be redundant. As one can see, on the average over 80% of the generated invariants were proved to be redundant. Moreover, Table 8.1 reports whether quantified invariants with only universal quantifier ($\forall$-Inv) and with quantifier alternations ($\forall\exists$-Inv) have been generated. The relative size of the minimised set varies from example to example. Also, the intended invariant is always implied by the invariants generated in the first second (as reported in Table 8.3). For example, the intended (and rather complex) invariant for the `Partition` problem is implied by invariants 1 and 81 (see Table 8.3), while the first 166 invariants are

generated in 1 second (Table 8.1). This suggests that the symbol elimination method generates increasingly sophisticated invariants, while natural and simple invariants are generated quickly.

8.4.2 Industrial Examples

We also ran Vampire's symbol elimination on 48 annotated array examples provided by Dassault Aviation. Since Vampire does not deal with pointers, we safely replaced pointers by arrays in 5 examples, and structures by arrays in 3 example loops. The 48 annotated array examples involve array copying, initialisation and shifts, and used arithmetical operations (e.g. addition, minus, plus, multiplication) and comparisons (e.g. greater, not equal) over the array content.

Vampire failed to find sufficiently strong invariants for 10 of these loops, for the following reasons. 6 loops were nested (all related to sorting algorithms) and thus cannot be analysed by the current version at all. Of the remaining 4 loops, two traversed sorted arrays using a logarithm-time search, one accessed the array using logically complex manipulation with array indexes, and the last one computed the sum of all array elements.

The results for the remaining 38 loops are analysed in Table 8.2. The first row of Table 8.2 shows the performance of Vampire on loops having only a single path, whereas the second row gives the results for multi-path loops. The second column shows the number of such loops. The third column gives the average number of invariants generated by Vampire with a 1 second time limit. The fourth and the fifth columns show, respectively, the number of invariants in the minimised set and the percentage of invariants proved to be redundant. The last two columns show whether any quantified invariants with universal quantifiers ($\forall$-Inv) (respectively with quantifier alternations $\forall\exists$-Inv) have been generated. We note that *for all 38 examples the intended invariants have been implied by the ones generated by Vampire*.

8.4.3 Analysis of Experiments

By studying the minimised sets of generated invariants we discovered that it still contains many redundancies and that many generated clauses could have been further improved by a better theory reasoning or algebraic simplifications.

Example 8.4.1 For the `Partition_Init` program, 692 invariants were generated in 10 seconds (see Table 8.1), out of which 127 invariants were kept in the minimised set.

Loop	♯ SEI	♯ Min SEI	Inv of interest	Generated invariants implying Inv

`Initialisation` [SG09]
$i = 0;$
while $(i < m)$ do
$A[i] = 0; i = i+1$
end do
	15	5	$\forall x: \quad 0 \le x < i \implies A[x] = 0$	`inv7`:
				$\forall x_0, x_1, x_2: \quad 0 \neq x_0 \vee x_1 \neq x_2 \vee A(x_1) = x_0 \vee \neg i > x_2 \vee \neg x_2 \ge 0$

`Copy` [SG09]
$i = 0;$
while $(i < m)$ do
$B[i] = A[i]; i = i+1$
end do
	24	5	$\forall x: \quad 0 \le x < i \implies B[x] = A[x]$	`inv8`:
				$\forall x_0, x_1: \quad A[x_0] = B[x_1] \vee x0 \neq x_1 \vee \neg i > x_0 \vee \neg x_0 \ge 0$

`Vararg` [SG09]
$i = 0;$
while $(A[i] > 0)$ do
$i = i+1$
end do
	1	1	$\forall x: \quad 0 \le x < i \implies A[x] > 0$	`inv0`:
				$\forall x_0: \quad \neg i > x_0 \vee \neg x_0 \ge 0 \vee A(x_0) > 0$

`Partition` [SG09]
$i = 0; j = 0; k = 0;$
while $(i < m)$ do
if $(A[i] >= 0)$
then $B[j] = A[i]; j = j+1$
else $C[k] = A[i]; k = k+1$
end if;
$i = i+1$
end do
	166	38	$\forall x: \quad 0 \le x < j \implies B[x] \ge 0 \wedge$
			$\exists y: \quad B[x] = A[y]$
				`inv1`:
				$\forall x_0: \quad A(sk_2(x_0)) \ge 0 \vee \neg j > x_0 \vee \neg x_0 \ge 0$
				`inv81`:
				$\forall x_0: \quad \neg j > x_0 \vee \neg x_0 \ge 0 \vee A(sk_2(x_0)) = B(x_0)$

`Partition_Init` [SG09]
$i = 0; k = 0;$
while $(i < m)$ do
if $(A[i] == B[i])$
then $C[k] = i; k = k+1$
end if;
$i = i+1$
end do
	168	24	$\forall x: \quad 0 \le x < k \implies A[C[x]] = B[C[x]]$	`inv0`:
				$\forall x_0: A(sk_1(x_0)) = B(sk_1(x_0)) \vee \neg k > x_0 \vee \neg x_0 \ge 0$
				`inv30`:
				$\forall x_0, x_1, x_2: \quad sk_1(x_0) \neq x_1 \vee x_0 \neq x_2 \vee \neg k > x_0 \vee \neg x_0 \ge 0 \vee C(x_2) = x_1$

`Shift`
$i = 0;$
while $(i < m)$ do
$A[i+1] = A[i]; i = i+1$
end do
	24	5	$\forall x: \quad 0 \le x \le i \implies A[x] = A[0]$	`inv5`:
				$\forall x_0, x_1, x_2: \quad x_0 + 1 \neq x_1 \vee A[x_0] \neq x_2 \vee A[x_1] = x_2 \vee \neg i > x_0 \vee \neg x_0 \ge 0$
				`inv13`:
				$\forall x_0, x_1: \quad A[0] \neq x_0 \vee x_1 \neq 1 \vee A[x_1] = x_0$

Table 8.3: Invariant generation by symbol elimination with Vampire, within 1 second time limit.

By further inspection of these 127 invariants, we noticed the following 2 invariants:

`inv30`:
$\forall x_0, x_1, x_2: \quad sk_1(x_0) \neq x_1 \vee x_0 \neq x_2 \vee \neg c > x_0 \vee \neg x_0 \ge 0 \vee C(x_2) = x_1$

`inv677`:
$\forall x_0, x_1: \quad C(x_1) = sk_1(x_0 + 2) \vee 0 \ge x_0 \vee x_0 + 2 \neq x_1 \vee \neg c > x_0 + 2$

When running Vampire only on these two formulas (without symbol elimination and consequence generation), Vampire proves in essentially no time that `inv30` $\implies$ `inv677`. Hence `inv677` is redundant, but could not be proved to be redundant using

the consequence elimination mode with a 20 seconds time limit.

The above example suggests thus that further refinements of integer reasoning in conjunction with first-order theorem proving are crucial for generating a minimal set of interesting invariants. We leave this issue for further research.

Based on the experiments described here, we believe that we are now ready to answer the three questions raised in Section 8.1 about using symbol elimination for invariant generation.

1. [Strength] For each example we tried, (i) Vampire generated complex quantified invariants as conclusions of symbol eliminating inferences (some with quantifier alternations), (ii) using the invariants inferred by Vampire, the intended invariants and loop properties of the example could be automatically proved by Vampire in essentially no time. Hence, symbol elimination proves to be a powerful method for automated invariant generation.

2. [Time] Symbol elimination in Vampire is very fast. Within a 1 second time limit a large set of complex and useful quantified invariants have been generated for each example we tried.

3. [Quantity] Symbol elimination, even with very short time limits, can result in a large amount of invariants, ranging from one to several hundred. By interfacing symbol elimination with consequence elimination, one obtains a considerably smaller amount of non-redundant invariants: in practice, about 80% of invariants obtained by symbol elimination are normally proved to be redundant. We believe that the generated minimised set of invariants makes symbol elimination attractive for industrial software verification. It seems that the set of remaining invariants can be further reduced by better reasoning with quantifiers and theories.

8.5 Related Work

To the best of our knowledge, symbol elimination is a new approach that has not been previously evaluated. A related approach to symbol elimination is presented in [McM08] where theorem proving is used for generating interpolants as quantified invariants that imply given assertions. Predefined assertions and predicates are also the key ingredients in [GT07, BHMR07, GMT08, SG09] for quantified invariant inference.

For doing so, predicate abstraction is employed to derive the strongest boolean combination of a given set of predicates [GT07, GMT08], or invariant templates are used over predicate abstraction [SG09] or constraint solving [BHMR07]. Abstract interpretation is also used in [GRS05, HP08], where quantified invariants are automatically inferred by an interval-based analysis over array indexes, without requiring user-given assertions. Unlike the cited works, in our experiments with symbol elimination to invariant generation, we generated complex invariants with quantifier alternations without using predefined templates, predicates and assertions, and without using abstract interpretation techniques.

Quantified array invariants are also inferred in [HHKR10], by deploying symbolic computation based program analysis over loops. Although symbolic computation offers a more powerful framework than symbol elimination when it comes to arithmetical operations, all examples reported in [HHKR10] were successfully handled by using symbol elimination for invariant generation. However, [HHKR10] can only infer universally quantified invariants, whereas our experiments show that symbol elimination can be used to derive invariants with quantifier alternations.

8.6 Conclusions

We describe and evaluate the recent implementation of symbol elimination in the first-order theorem prover Vampire. This implementation includes a program analysis framework, theory reasoning, efficient consequence elimination, and invariant generation.

Our experimental results give practical evidence of the strength and time-efficiency of symbol elimination for invariant generation. Furthermore, we investigated quantitative aspects of symbol elimination.

We can answer affirmatively the question whether a computer program can automatically generate powerful program properties. Indeed, the properties generated by Vampire implied the intended properties of the programs we studied. However, our research also poses highly non-trivial problems. The main problem is the large number of generated properties. On the one hand, one can say that this is an accolade to the method that it can generate many more properties than a human would ever be able to discover. On the other hand, one can say that the majority of generated properties are uninteresting. This poses the following problems that can help us understand computer reasoning and intelligence better:

1. what makes some program properties more interesting than others from the viewpoint of programmers (or applications);

2. how can one automatically tell interesting program properties from other properties generated by a computer?

Answering these fundamental questions will also help us to improve program generation methods for the purpose of applications in program analysis and verification.

Chapter 9

The 481 Ways to Split a Clause and Deal with Propositional Variables

Authors: Krystof Hoder, Andrei Voronkov

This chapter is in the form of a paper, but not yet submitted for publication.

It is often the case that first-order problems contain propositional variables and that proof-search generates many clauses that can be split into components with disjoint sets of variables. This is especially true for problems coming from some applications, where many ground literals occur in the problems and even more are generated.

The problem of dealing with such clauses has so far been addressed using either splitting with backtracking (as in Spass [Wei01]) or splitting without backtracking (as in Vampire [RV02]). However, the only extensive experiments described in the literature [RV01a] show that on the average using splitting solves fewer problems, yet there are some problems that can be solved only using splitting.

In view of importance of dealing with propositional and ground literals we tried to identify essential issues contributing to efficiency in dealing with splitting in resolution theorem provers and enhanced the theorem prover Vampire with new options, algorithms and datastructures dealing with splitting. This paper describes these options, algorithms and datastructures and analyses their performance in extensive experiments carried out over the TPTP library [Sut09].

Another contribution of this paper is a calculus RePro separating propositional reasoning from first-order reasoning.

9.1 Introduction

In first-order theorem proving, theorem provers based on resolution and superposition caluli (in the sequel simply called *resolution theorem provers*) are predominant. This is confirmed by the results of the last CASC competitions [Sut08]. Among the top three theorem provers Vampire [RV02] and E [Sch02] are resolution-based, while iProver [Kor08] implements both an instance-based calculus and resolution with superposition.

Resolution theorem provers use *saturation algorithms*. They deal with a search space consisting of clauses. Inferences performed by saturation algoritms are of three different kinds:

1. *Simplifying inferences* replace a clause by another clause that is simpler in some strict sense.

2. *Deletion inferences* delete clauses from the search space.

3. *Generating inferences* derive a new clause from clauses in the search space. This new clause can then be immediately simplified and/or deleted by other kinds of inference.

On hard problems the search space is often growing fast and simplifications and deletions consume a lot of time. Performance of resolution theorem provers degrades especially fast if it generates many clauses having more than one literal (*multi-literal clauses* for short) and heavy clauses (clauses of large sizes). There are several reasons for this degradation of performance:

1. The complexity of algorithms implementing inference rules depends on the size of clauses. The extreme case are algorithms for subsumption and subsumption resolutions. These problems are known to be NP-complete and algorithms implementing them are exponential in the number of literals in clauses.

2. Heavy clauses require more memory to be stored. Moreover, every literal in a clause (and sometimes every term occurring in such a literal) are normally added to one or more indexes. Index maintenance may require a considerable space and time and operations on these indexes slow down significantly when the indexes become large.

3. Generating inferences applied to heavy clauses usually generate heavy clauses. Generating inferences applied to clauses with many literals usually generate

clauses with many literals. For example, resolution applied to two clauses containing n_1 and n_2 literals normally gives a clause with $n_1 + n_2 - 2$ literals.

There are two ways of dealing with multi-literal and heavy clauses. One is simply to start discarding them after some time, thus losing completeness as described in [RV03]. Another one is to use *splitting*. There are two kinds of splitting described in the literature: splitting with backtracking (as in Spass [Wei01]) or splitting without backtracking (as in Vampire [RV02]).

9.2 Propositional Variables in Resolution Provers

Both kinds of splitting are implemented using introduction of propositional variables denoting components of splitted clauses. When many such variables are introduced, they give rise to clauses with many propositional literals. Such clauses clobber up search space and slow down expensive operations, such as subsumption. Therefore, the problem of dealing with propositional literals is closely related to splitting. Apart from variables arising from splitting, propositional variables are common in many applications. They may also be introduced during preprocessing when naming is used to generate small clausal normal forms.

The resolution and superposition calculus is very efficient for proving theorems in first-order logic. In propositional logic, it is not competitive to DPLL. Suppose that we have a problem that uses both propostional and non-propositional atoms. Then treating propositional atoms in the same way as non-propositional one results in performance problems. For example, if we use the code trees technique for implementing subsumption [Vor95] and make no special treatment for propositional variables, it will work in the worst case in exponential time even for a pair of propositional clauses, while the best algorithms for propositional subsumption are linear.

9.2.1 The Calculus RePro

To address the problem of dealing with propositional variables, in this section we will introduce a *calculus RePro* for dealing with clauses having propositional literals and will illustrate some options of Vampire using this calculus. The calculus separates propositional reasoning from non-propositional.

Let us call a *pro-clause* any expression of the form $C \mid P$, where C is a clause containing no propositional variables and P is a propositional formula. Logically, this

pro-clause is equivalent to $C \vee P$, so the bar sign $\mid$ can be seen as simply separating the propositional and non-propositional parts of the pro-clause. We will consider a clause containing no propositional variables as a special kind of pro-clause, in which P is a false formula.

Note that a pro-clause $C \vee P$ is not necessarily a clause, since P can be an arbitrary formula. Also, any propositional formula P can be considered a special case of a pro-clause $\Box \mid P$, where $\Box$ denotes, as usual, the empty clause. We will call any pro-clause $\Box \mid P$ *propositional*.

The calculus RePro is parametrised by an *underlying resolution calculus*. That is, for every resolution calculus on clauses we will define an instance of the calculus RePro based on this resolution calculus. However, since we are not varying the underlying calculus in this paper, we will simply speak of RePro as a calculus.

Generating inferences. For every generating inference

$$\frac{C_1 \quad \cdots \quad C_n}{C}$$

of the resolution calculus the following is an inference rule of RePro:

$$\frac{C_1 \mid P_1 \quad \cdots \quad C_n \mid P_n}{C \mid (P_1 \vee \ldots \vee P_n)} \,.$$

Simplifying inferences. Let

$$\frac{C_1 \quad \cdots \quad C_n \quad \not\emptyset}{C} \tag{9.1}$$

be a simplifying inference of the resolution calculus. Speaking the theory of resolution, this means that C is implied by $C_1, \ldots, C_n, D$ and D is redundant with respect to $C_1, \ldots, C_n, C$. If $P_1 \vee \ldots \vee P_n \implies P$ is a tautology, then the following is a simplifying inference rule of RePro:

$$\frac{C_1 \mid P_1 \quad \cdots \quad C_n \mid P_n \quad D \mid \not P}{C \mid (P_1 \vee \ldots \vee P_n)} \,. \tag{9.2}$$

Deletion inferences. Let

$$C_1 \quad \cdots \quad C_n \quad \not\emptyset$$

be a deletion inference of the resolution calculus, that is, D is redundant with respect

to $C_1,\ldots,C_n$. If $P_1 \vee \ldots \vee P_n \implies P$ is a tautology, then the following is a deletion inference of RePro:

$$C_1 \mid P_1 \quad \cdots \quad C_n \mid P_n \quad \cancel{D \mid P}.$$

9.2.2 Completeness

It is not hard to derive soundness and completeness of RePro assuming the same properties of the underlying resolution calculus, however completeness here means something different from completeness in the theory of resolution. The reason for this difference is that RePro contains essentially no rules for dealing with the propositional part of clauses. In the completeness theorem below, we assume knowledge of the theory of resolution, as in [BG01, NR01]. Also, we do not specify the underlying calculus, for example, the calculus used in Vampire can be used.

Theorem 9.2.1 [Completeness] Let $S_0, S_1, S_2, \ldots$ be a fair sequence of sets of pro-clauses such that S_0 is unsatisfiable. Then there exists $i \geq 0$ such that the set of propositional pro-clauses in S_i is unsatisfiable too.
The proof is omitted here. Note that this theorem implies that the proof-search in RePro can be carried out by using any standard fair saturation algorithm to perform RePro inferences corresponding to the rules of the underlying calculus plus unsatisfiability checking for the propositional part. This is how it is implemented in Vampire.

To implement such an algorithm for RePro on top of a standard implementation of the resolution calculus one needs to address the following questions:

(q1) representation of the propositional part of pro-clauses;

(q2) representation of propositional pro-clauses (which can be different from the representation of the propositional part of pro-clauses);

(q3) unsatisfiability checking for sets of propositional pro-clauses;

(q4) efficient simplification rules for pro-clauses.

There are some other implementation details to be addressed. For example, the inference selection process in saturation algorithms usually depends on the weights of clauses (which is usually their size measured in the number of symbols). One can use different size measures for pro-clauses, especially when their propositional parts are not necessarily clauses. This adds one more question:

(q5) pro-clause selection.

However, before discussing these solutions we will introduce some other rules that can be used in RePro.
Propositional tautology deletion is a deletion rule of RePro formulated as follows:

$$\frac{}{D \mid P},$$

where P is a tautology.
The merge rule of RePro is formulated as follows:

$$\frac{C \mid P_1 \quad C \mid P_2}{C \mid (P_1 \wedge P_2)}$$

Note that so far this is the only rule that introduces propositional formulas other than clauses.
The merge subsumption rule of RePro is formulated as follows:

$$\frac{C \mid P_1 \quad D \mid P_2}{D \mid (P_1 \wedge P_2)},$$

where C subsumes D. This rule can also introduce propositional formulas that are not clauses.

9.2.3 The Calculus ReProR

One can also define simplifying rules on pro-clauses in an alternative way. Namely, the modification is as follows. Consider a simplifying rule (9.1) of the underlying resolution calculus. Then the following can be considered as a simplifying inference rule:

$$\frac{C_1 \mid P_1 \quad \cdots \quad C_n \mid P_n \quad D \mid P}{C \mid (P_1 \vee \ldots \vee P_n) \quad D \mid (P_1 \vee \ldots \vee P_n \implies P)}.$$

One can see that previously defined simplifying rule (9.2) is a special case of this one, since, if $P_1 \vee \ldots \vee P_n \implies P$ is a tautology, the second inferred clause can be removed. One can also reformulate the deletion rules in the same way. We will denote the resulting calculus ReProR (the refined RePro). Note that the simplification rules of the refined calculus introduce non-clauses in the propositional part.

The advantage of the alternative formulation of simplification and deletion rules is that one clause can be simplified away into a tautology using a sequence of simpifying or deletion rules impossible in the standard formulation of RePro. For example, a clause $A \vee B \mid (p \wedge q)$ is redundant in the presence if $A \mid p$ and $B \mid q$ using the following sequence of subsumption deletion rules:

$$\dfrac{\dfrac{A \mid p \quad \cancel{A \vee B \mid (p \wedge q)}}{\cancel{B \mid q \quad A \vee B \mid (p \Longrightarrow (p \wedge q))}}}{A \vee B \mid (q \Longrightarrow (p \Longrightarrow (p \wedge q)))}$$

whose conclusion is a tautology.

9.3 Splitting

In very simple terms, splitting is based on the following idea. Suppose that S is a set of clauses and $C_1 \vee C_2$ a clause such that the variables of C_1 and C_2 are disjoint. Then the set $S \cup \{C_1 \vee C_2\}$ is unsatisfiable if and only if both $S \cup \{C_1\}$ and $S \cup \{C_2\}$ are unsatisfiable. There is more than one way to implement splitting. Before discussing them let us introduce some definitions.

Recall that a clause is a disjunction $L_1 \vee \ldots \vee L_n$ of *literals*, where a literal is an atomic formula or a negation of an atomic formula. A literal or clause is *ground* if it contains no occurrences of variables. In the context of splitting we consider a clause as a set of its literals. In other words, we assume that clauses do not contain multiple occurrences of the same literal and clauses equal up to permutation of literals are considered equal. Let $C_1, \ldots, C_n$ be clauses such that $n \geq 2$ and all the C_i's have pairwise disjoint sets of variables. Then we say that the clause $C \stackrel{\text{def}}{=} C_1 \vee \ldots \vee C_n$ is *splittable* into *components* $C_1, \ldots, C_n$. We will also say that the set $C_1, \ldots, C_n$ is a *splitting* of C. For example, every ground multi-literal clause is splittable. There may be more than one way to split a clause, however there is always a unique splitting such that each component C_i is non-splittable: we call this splitting *maximal*. It is easy to see that a maximal splitting has the largest number of components and every splitting with the largest number of components is the maximal one. There is a simple algorithm for finding the maximal splitting of a clause [RV01a], which is, essentially, the union-find algorithm.

Splittable clauses appear especially often when theorem provers are used for software verification and static analysis. Problems used in these applications usually have

a large number of ground clauses (coming from describing properties of a program) and a small number of non-ground clauses.

There are essentially two ways of using splitting in a first-order resolution theorem prover. One is splitting with backtracking as implemented in Spass [Wei01] and another *splitting without backtracking* [RV01a]. Each of them is described in the next subsections, where we also point out potential efficiency problems associated with each kind of splitting.

When we discuss the use of splitting in resolution theorem provers, it is very important to understand how the use of splitting affects other components of such provers. The efficiency and power of modern resolution theorem provers comes from two techniques: *redundancy elimination* (see [BG01] for the theory and [RV03] for the implementation aspects) and *term indexing* (see [SRV01]). Another component important for understanding efficiency is the saturation algorithm and especially the clause selection algorithm used to implement this algorithm.

Redundancy elimination. Unlike backtracking algorithms used in DPLL, saturation algorithms are backtracking-free. When clauses are simplified or deleted, these simplifications and deletions do not have to be undone. Use of some forms of splitting can require backtracking.

Term indexing. Even when simplifications are used, the search space can quickly grow to hundreds of thousands of clauses. To perform inferences on such a large search space efficiently, theorem provers maintain several indexes storing information about terms and clauses. These indexes make it easier to find candidates for inferences. In some cases inferences can be performed only by using the relevant index, without retrieving clauses used for these inferences. The number of different indexes in theorem provers varies and can be as many as about 10. Frequent insertion and deletion in an index can affect performance of a theorem prover. A typical example is when a theorem prover generates an equailty $a = b$ between two constants and uses it to rewrite a into b. For nearly all indexing techniques used in the resolution theorem provers, every term and clause containing a must be removed from all indexes and a new term containing b inserted in them again. Doing this single simplification step on an indexes set with 100,000 clauses can take a very long time.

Clause selection. Selection of generating inferences in resolution theorem provers is implemented using *clause selection*, see [RV01a] for details. For selection, clauses are

put in one or more priority queues and selected based on their priorities. Normally, the majority of selected clauses are taken from the clauses of the smallest weight.

The use of splitting may heavily affect all these parts of the saturation algorithm implementation: redundancy elimination, term indexing and clause selection. Let us discuss this in more detail in the rest of this section.

9.3.1 Splitting without backtracking

Splitting without backtracking [RV01a] can be implemented using a naming technique. Suppose we have a splittable clause $C_1 \vee \ldots \vee C_n$ with components $C_1, \ldots, C_n$. We introduce new propositional variables $p_1, \ldots, p_{n-1}$ to "name" the first $n-1$ components. That is, we introduce them together with definitions $p_i \leftrightarrow C_i$. Then we use the rule

$$\frac{\cancel{C_1 \vee \ldots \vee C_n}}{C_1 \vee \neg p_1 \qquad \ldots \qquad C_{n-1} \vee \neg p_{n-1} \qquad C_n \vee p_1 \vee \ldots p_{n-1}}$$

If the same components appear more than once in a splittable clause, their names can be reused. In fact, they should be reused, as shown in [RV01a].

The advantage of this approach is that we do not need to perform any backtracking, which spares us the costs of inserting and deleting clauses from indexes. The only additional cost to the implementation of saturation is an index of components required to reuse names.

Splitting without backtracking is very efficient on some problems but may also be very inefficient, since it can introduce thousands of propositional variables and long clauses containing these variables.

Another drawback of this method is that simplifying inferences are not being performed "across branches." For example, when we split clause $f(a) = a \vee q(a)$, we obtain $f(a) = a \vee p$, so we cannot use the equality $f(a) = a$ to simplify expressions such as $q(f(a))$ by demodulation. In the case of splitting with backtracking, we would obtain the unit clause $f(a) = a$, and all the demodulation simplifications would be performed (at the cost of having to backtrack them later).

9.3.2 Splitting with Backtracking

Splitting with backtracking is based on the idea of the DPLL splitting. It uses the labelled clause calculus introduced in [FW09]. We will first describe the use of labels and then show how it can be captured by a variation of the RePro calculus.

When we have a splittable clause $C_1 \vee C$, where C_1 is a component, it is first replaced by its minimal component C_1, and when we derive a contradiction that follows from C_1, we (well, almost) backtrack to the point of the split and introduce the rest of the clause C. In the spirit of DPLL, we may also add $\neg C_1 \mid P'$ at this point. (Whether we do this is controlled by a Vampire option.)

To implement this technique, we assign a label to every split that we perform, and augment each clause C by a set of split labels on which it depends. Each newly derived clause depends on a set of labels that is the union of the sets belonging to its parents. When a clause is deleted, we need to examine the labels of the clauses which justified the deletion. If the deletion was justified by some labels on which the deleted clause itself does not depend, we do not delete the clause, but rather keep it aside to be restored if we backtrack the label that justified its deletion.

The backtracking splitting as we have implemented it can be captured by the RePro calculus if we restrict the inferences that can be performed at any given point, and introduce a different treatment for simplification inferences.

We do not use any of the RePro rules that would introduce non-clauses into the propositional part. The propositional parts of pro-clauses are therefore clauses, and their literals correspond to the labels that the clause has assigned in the labelled calculus.

We are maintaining a partial model M which is initially empty and to which we add propositional literals corresponding to active splits. At each point we rescrict inferences of the RePro calculus to the pro-clauses whose propositional part is not satisfied by the model M.

The split labels are seen as fresh propositional variables and the label introduction at splitting $C_1 \vee C$ can be viewed as naming C_1 with a propositional variable. More precisely, when we split $C_1 \vee C \mid P$ and use the name p_1 for C_1, we add pro-clauses $C_1 \mid (\neg p_1 \vee P)$ and $C \mid (p_1 \vee P)$. We also make a note that p_1 *depends* on every propositional variable in P and add p_1 into the partial model M.

At this point, the original clause $C_1 \vee C \mid P$ is subsumed by the newly introduced $C_1 \mid (\neg p_1 \vee P)$. Also, having added p_1 into M keeps the clause $C \mid (p_1 \vee P)$ from participating in any inferences for now.

Clause simplification and restoring upon backtracking is the part that does not fit well into the RePro calculus and for which we need to introduce a special treatment. When a clause $C \mid P$ is simplified with $C_1 \mid P_1, \ldots, C_n \mid P_n$ as premises, the restoration of the simplified clause in the labelled calculus corresponds with the point when the

formula $F \equiv (P_1 \vee \ldots \vee P_n) \rightarrow P$ becomes no longer satisfied by the partial model M. As a matter of fact, F is actually the propositional formula in one of the conclusions of simplifying inferences in the ReProR calculus. One would be therefore tempted to use the ReProR calculus to capture the splitting with backtracking. However, the problem is that the formula F is not a clause, and the labelled calculus we use to implement backtracking splitting does not easily capture general formulas using the clause labels.

We therefore rather check with each change of the model M whether the condition for restoring any of the simplified clauses does occur, and if so, we put the clause back into the saturation algorithm.

When we derive a propositional pro-clause $\Box \mid Q$, we select an atom p in the clause $Q \equiv Q' \vee \neg p$ so that no atom in Q' *depends* on it (there will always be at least one such; if there is several, we choose arbitrarily). Let us remind that there is only one pro-clause with a positive occurrence of p — the clause $C \mid (p \vee P)$ which we introduced after splitting $C_1 \vee C \mid P$. This clause became inactive as we added p into the partial model M, so it could not spread the literal p any further. Now we resolve the pro-clauses $C \mid (p \vee P)$ and $\Box \mid Q' \vee \neg p$ on the atom p to obtain $C \mid (Q' \vee P)$, delete the clause $C \mid (p \vee P)$ and replace p in M with $\neg p$.

Note that we have removed p from M which means that the originally splitted clause $C_1 \vee C \mid P$ is restored, even though just to become subsumed again by $C \mid (Q' \vee P)$. Now there are two possibilities. If Q' is an empty clause, $C_1 \vee C \mid P$ will never be restored as it is subsumed by $C \mid P$ which has the same propositional part.[1] This corresponds to the case when we have refuted the first branch of the split without any additional assumptions. If Q' is a non-empty clause, the original splitted clause may be restored if some of the assumptions on which we refuted the split is backtracked.

It should be noted that while we can change the polarity of a propositional variable in the model M from positive to negative, we never change it from negative to positive. Therefore, once we assign false to a propositional variable, all pro-clauses that contain it in a negative literal may be deleted.

Drawbacks. The disadvantage of this kind of splitting is that, upon backtracking, we sometimes have to delete and restore many clauses. This leads to costly index maintenance operations, and also a lot of work can be wasted.

For example, suppose we split a clause $a = b \vee C$ where the symbol b is the smallest in the simplification ordering and does not appear anywhere else in the problem, while

[1] The restoration trigger for the simplified clause is $P \rightarrow P$ which will never become unsatisfied.

a has many occurrences. Splitting this clause will introduce a unit clause $a = b$ and the backward demodulation will replace every occurence of a by b, resuting in massive updates in all indexes. Since b does not appear anywhere else, the equality will not be helpful in any way, but all the rewritten clauses will depend on this split. Once we reach a refutation using the rewritten clauses, we will have to restore all the clases containing a, once again resulting in massive updates in all indexes. Also note that we may end up doing a lot of repeated work as the proof search on the branch using $a \neq b$ will be likely similar to the one on the $a = b$ branch.

9.4 Implementation and Parameters

In this section we describe various parameters implemented in Vampire and related to splitting and/or use of propositional variables. We also discuss these parameters in the context of the questions (q1)–(q5). These parameters and their values are summarised in Table 9.1.

9.4.1 Splitting

The main parameter controlling splitting is `splitting`. It has three values: `backtracking`, `nobacktracking` and `off`. All other options have two values: `on` and `off`.

Clauses may be split either eagerly, before they enter the passive container, or the splitting can be postponed until a clause is selected for activation. This is controlled by the option `split_at_activation`.

The set of split clauses can be restricted in several ways. Option `split_goal_only` restricts splitting only to goal clauses and clauses that are derived from them. Enabling `split_input_only` excludes derived clauses from splitting, allowing splits only on the clauses which were initially passed to the saturation algorithm.

A different kind of restriction is to add a requirement that both split components contain less positive literals than the original clause. Such splitting will lead to clauses that are closer to the Horn form which allows only one positive literal per clause. This setting is enabled by the `split_positive` option.

Splitting with backtracking. The implementation is based on the description in [Wei01]. We extended it by use of time stamps and reference counters on clauses. This

option	short	values
general		
splitting	spl	backtracking, nobacktracking, off
split_add_ground_negation	sagn	on, off
what and when to split		
split_goal_only	sgo	on, off
split_input_only	sio	on, off
split_positive	spo	on, off
split_at_activation	sac	on, off
propositional pro-clauses		
propositional_to_bdd	ptb	on, off
sat_solver_for_empty_clause	ssec	on, off
sat_solver_with_naming	sswn	on, off
simplifications		
sat_solver_with_subsumption_resolution	sswsr	on, off
empty_clause_subsumption	ecs	on, off
bdd_marking_subsumption	bms	on, off
clause and literal selection		
nonliterals_in_clause_weight	nicw	on, off
splitting_with_blocking	swb	on, off

Table 9.1: Option names, short names and values

allows us to implement the structure for restoring clauses upon backtracking more efficiently — upon backtracking we only traverse the list of clauses that is to be restored, and let the time-stamping ensure that we never restore the same clause twice.

If the option split_add_ground_negation is enabled, upon backtracking caused by splitting a ground literal L, we add its negation $\neg L$ as a new clause. The parameter nonliterals_in_clause_weight means that the weight of each clause will be increased by the number of splits on which it depends.

9.4.2 Propositional Parts of Pro-Clauses

There are several possible implementations of pro-clauses with a clausal propositional part. However, variants of RePro using non-clausal propositional parts quickly lead to very complex formulas for which the only suitable data structure we could think of was (ordered) binary decision diagrams [Bry86], or simply BDDs. Thus, we extended the clause objects in Vampire by a reference to the BDD representing the propositional

part of a pro-clause.

Since the refined calculus ReProR requires the use of arbitrary formulas, we also used BDDs to implement this calculus. We hoped that it will be very efficient for some problems since many more clauses will be simplified away. In reality ReProR turned out to be almost disfunctional. The refined simplification rules created ever more complex propositional formulas with very large BDDs. In many cases these BDDs used all the available memory. It was also common that nearly all run-time of Vampire was consumed by BDD operations. Therefore, we decided not to use ReProR and report no results on it in this paper.

The option `propositional_to_bdd` (q1) chooses whether BDDs are used to store propisitional parts of pro-clauses. If BDDs are not in use, we treat propositional literals in the same way as all other literals. It is a separate issue how to deal with purely propositional clauses. One can also treat them as ordinary clauses. However, one can choose to pass them to a SAT solver instead. Since every propositional clause can be considered as a pro-clause $\Box \mid P$ with the empty non-propositional part, options for dealing with such clauses use `empty_clause` in their names. In the option `sat_solver_for_empty_clause` (q2,q3) is on, such clauses are passed to a SAT solver.

When we use BDDs for pro-clauses but not for propositional clauses, whenever we obtain a BDD for a propositional clause, we must convert this BDD to a set of clauses. The number of propositional clauses obtained from a BDD can be exponential. To cope with this problem, we added an option `sat_solver_with_naming` (q2) that would make conversion of BDDs to clauses almost linear time by introducing new propositional variables. An alternative to `sat_solver_with_naming` is the option `sat_solver_with_subsumption_resolution` which uses subsumption resolution to shorten the long clauses generated when converting BDDs to CNF without the introuction of new propositional variables.

If we decide to represent the propositional pro-clauses as BDDs, the transition from a first-order pro-clause into propositional is straightforward. We keep at most one propositional pro-clause by eagerly applying a propositional merging rule

$$\frac{\cancel{\Box \mid P} \quad \cancel{\Box \mid P'}}{\Box \mid (P \wedge P')}$$

whenever obtain a new propositional pro-clause $\Box \mid P'$. This way we know that if the set of propositional clauses becomes unsatisfiable, we will derive a propositional clause with associated $\bot$ BDD node.

For the first-order pro-clauses we may decide to reflect the complexity of the propositional part in the clause selection process. To this end, enabling the option `nonliterals_in_clause_weight` in presence of BDDs increases the size of clauses by the depths of the BDD graphs of their propositional parts.[2]

When we derive a propositional pro-clause $\Box \mid P$, clauses $C \mid P'$ such that $P \implies P'$ become redundant. This follows from the RePro version of subsumption rule as an empty clause subsumes any other clause:

$$\Box \mid P \qquad \cancel{C \mid P'} \qquad \text{if } P \implies P'$$

It would not be feasible to make an implication check between P and the propositional part of every pro-clause present in the saturation algorithm. We have implemeted two incomplete checks for subsumption by propositional pro-clauses.

First of the checks focuses on the premises of the derived propositional pro-clause, as there is a good chance that some of the ancestors will also have P as its propositional part. If we succeed with some of the premises, we carry on the check with their premises and further on in the derivation graph, as long as we are succeeding. This check is controlled by the option `empty_clause_subsumption`.

The second of the checks uses the shared structure of the BDDs. When we derive a propositional pro-clause $\Box \mid P$, we set a *subsumed* flag in the BDD node corresponding to P. Whenever we see a first-order pro-clause to have a BDD node with the *subsumed* flag, we know it is redundant and can be deleted. Moreover, our BDD implementation is aware of this flag and attempts to "spread" the mark while performing other BDD operations. For example, when performing a disjunction operation, if one of the operands has the flag set, it will be set also for the node representing the disjunction of the operands. This subsumption algorithm is controlled by the option `bdd_marking_subsumption`.

9.5 Evaluation

There are all together 481 different combinations of values for the Vampire parameters related to splitting and propositional variables, so analysing the results was far from

[2]The DAG size of BDD graphs would probably better reflect the complexity of propositional formulas, but computing this measure is not a "local" operation on BDDs — one would need to traverse the whole BDD subgraph to count the distinct nodes. The depth of a BDD graph can, however, be computed by using just the depths of immediate successors. The tree size of a BDD can be computed locally as well, however it can grow exponentially with the size of the BDD DAG.

trivial. For simplicity, we will call them *splitting parameters,* though this name is a bit misleading since some of them are actually related to dealing with propositional variables.

As the set of benchmarks we used unsatisfiable TPTP problems having non-unit clauses and rating greater than 0.2 and less than 1. Essentially, the rating is the percentage of existing provers that cannot solve a problem. For example, rating greater than 0.2 means that less than 80% of existing theorem provers can solve the problem. Likewise, rating 1 means that the problem cannot solved by the existing provers. However, the rating evaluation uses a single mode of every prover, so it is possible that the same prover can solve a problem of rating 1 using a different mode. For this reason, we also added problems of rating 1 and solvable by Vampire. We excluded very large problems since for them it was preprocessing, but not other options, that affect results the most. This resulted in 4,869 TPTP problems.

To conduct the experiments, we took a Vampire strategy that is believed to be nearly the best in the overall number of solved problems, and generated the 481 variations of this strategy obtained by setting the splitting parameters to all possible values. For each of this variations, we ran it on the selected problems with a 30 seconds time limit. This resulted in 2,341,989 runs, which roughly correspond to 1.5 years of run time on a single computer.

We evaluated the experiments in two different ways. First, we looked at the best overall strategies for the backtracking and non-backtracking splitting, and how may problems they solve. However, the number of solved problems for a single (even the best) setting of parameters is not the main criterion of importance for splitting parameters.

The reason for this is that it is known that problems are normally best solved by attempting them with a cocktail of strategies. The CASC [Sut08] version of Vampire uses a sequence of strategies to solve a problem, and using such a sequence is also a recommended mode for the users. Therefore in the second part of evaluation we looked at the numbers of problems solvable only by particular settings of the splitting parameters.

9.5.1 The Best and the Worst Strategies

Only 3,598 (about 74% of all problems) were solved by at least one splitting strategy. The top-level results are summarised in Table 9.2. The best and the worst strategies

splitting	strategies	worst	average	best
off	25	2708	2720	2737
backtracking	64	1825	2710	3143
non-backtracking	416	1756	2608	2929

Table 9.2: Problems solved by each setting of the splitting strategy.

	worst	best
splitting	nobacktracking	backtracking
propositional_to_bdd	on	
split_at_activation	off	on
split_goal_only	off	off
split_input_only	off	off
split_positive	off	off
nonliterals_in_clause_weight	off	off
bdd_marking_subsumption	off	
empty_clause_subsumption	on	
sat_solver_for_empty_clause	off	
split_add_ground_negation		on

Table 9.3: Best and worst strategies.

are shown in Table 9.3.[3] As one can see, without splitting all strategies behave very similar, which is expected, since problems normally contain few propositional symbols. However, the use of splitting makes a very substantial difference, especially for the best strategies. For example, the best strategy using splitting solved 3143 problems versus 2737 problems solved without splitting. Another interesting point is a huge gap between the performance of the worst and the best strategies using the same kind of splitting. However, the biggest surprise for us was the fact that the best strategies used splitting with backracking, as the anecdotal knowledge suggested that the splitting without backtracking on the average performs better.

9.5.2 Importance of Particular Parameters

To determine the importance of various splitting options, we put the numbers of problems that can be solved only with a particular setting of an option into Table 9.4.

[3]Some of the option values in the table are left out because they do not make sense in a particular configuration. E.g. for backtracking splitting we use labeled clauses, not BDDs, so the BDD related settings are left out.

Under (a) we show the number of problems that can be solved either only by backtracking or non-backtracking splitting. The number of problems solvable only without any splitting at all is zero. This perhaps surprising result is due to the fact that splitting can be restricted using the options `split_input_only`, `split_goal_only` and `split_positive` to the extent that almost no splits are actually performed.

The cases (b)–(m) show the numbers of problems requiring a particular setting of a parameter for some of the following cases: `off`, `backtracking`, `nobacktracking` or `all`. In the first three cases, the numbers for columns `off` and `on` stand for the number of problems which could be solved for the specified value of splitting only with the option enabled or disabled.[4] The row `all` gives numbers of problems where particular option setting was required across all relevant splitting modes.

From the Table 9.4 (j) it can be seen that the use of naming in clausification of BDDs is always a good thing to do, as none of the problems required to have this setting disabled. From case (n) it can be seen that it is very rarely the case that adding ground negations after refuting a splitting branch will harm, as only 6 problems are lost by enabling this setting, however 191 problems required to have this setting enabled. On the other hand, for many other options, having the possibility to enable or disable them is important, as either setting can solve problems which cannot be solved by the other.

9.6 Conclusion

We have implemented two variants of clause splitting in a first-order theorem prover, and through extensive experiments we have shown that the backtracking splitting in our setup gives the best performance. More importantly, we have also shown the importance of keeping a large portfolio of strategies, because large group of problems can be solved only by a variety of different approaches, not by having only one strategy, even though performing well on average.

Aside of the extensive experimental evaluation, we also presented new families of calculi RePro and ReProR which separate propositional from first-order reasoning.

All the described parameters are supported by the current version of the Vampire theorem prover which is available for download at `http://www.vprover.org`.

[4] More precisely, e.g. for the column `off` of option *A* we give the number of problems for which there existed setting of other options so that problem was solved with option *A* disabled, but for all the combinations of parameters the problem was not solved when the option *A* was enabled.

a) splitting

off	0
nobacktracking	128
backtracking	198

b) split_at_activation

	on	off
backtracking	147	73
nobacktracking	91	93
all	145	113

c) split_goal_only

	on	off
backtracking	31	155
nobacktracking	21	207
all	17	159

d) split_input_only

	on	off
backtracking	43	414
nobacktracking	67	302
all	33	384

e) split_positive

	on	off
backtracking	37	262
nobacktracking	28	146
all	35	181

f) propositional_to_bdd

	on	off
off	62	45
nobacktracking	227	107
all	226	106

g) nonliterals_in_clause_weight

	on	off
off	17	11
backtracking	55	45
nobacktracking	23	62
all	33	91

h) splitting_with_blocking

	on	off
nobacktracking	20	290

i) sat_solver_for_empty_clause

	on	off
off	8	5
nobacktracking	34	21
all	34	21

j) sat_solver_with_naming

	on	off
off	2	0
nobacktracking	22	0
all	22	0

k) sat_solver_with_ subsumption_resolution

	on	off
off	2	1
nobacktracking	1	2
all	2	2

l) bdd_marking_subsumption

	on	off
off	62	45
nobacktracking	227	107
all	226	106

m) empty_clause_subsumption

	on	off
off	5	7
nobacktracking	18	46
all	18	46

n) split_add_ground_negation

	on	off
backtracking	191	6

Table 9.4: Problems solved only by a single value of an option.

Chapter 10

Comparing Unification Algorithms in First-Order Theorem Proving

Authors: Krystof Hoder, Andrei Voronkov

Unification is one of the key procedures in first-order theorem provers. Most first-order theorem provers use the Robinson unification algorithm. Although its complexity is in the worst case exponential, the algorithm is easy to implement and examples on which it may show exponential behaviour are believed to be atypical. More sophisticated algorithms, such as the Martelli and Montanari algorithm, offer polynomial complexity but are harder to implement.

Very little is known about the practical performance of unification algorithms in theorem provers: previous case studies have been conducted on small numbers of artificially chosen problem and compared term-to-term unification while the best theorem provers perform set-of-terms-to-term unification using term indexing.

To evaluate the performance of unification in the context of term indexing, we made large-scale experiments over the TPTP library containing thousands of problems using the COMPIT methodology. Our results confirm that the Robinson algorithm is the most efficient one in practice. They also reveal main sources of inefficiency in other algorithms. We present these results and discuss various modifications of unification algorithms.

10.1 Introduction

Unification is one of the key algorithms used in implementing theorem provers. It is used on atoms in the resolution and factoring inference rules and on terms in the equality resolution, equality factoring and superposition inference rules. The performance of a theorem prover crucially depends on the efficient implementation of several key algorithms, including unification.

To achieve efficiency, theorem provers normally implement unification and other important operations using *term indexing*, see [RSV01, NHRV01]. Given a set L of *indexed terms*, and a term t (called the *query term*), we have to retrieve the subset M of L that consists of the terms l unifiable with t. The retrieval of terms is interleaved with insertion of terms to L and deletion of them from L. Indexes in theorem provers frequently store 10^5–10^6 complex terms and are highly dynamic since insertion and deletion of terms occur frequently. Our paper is the first ever study of unification algorithms in the context of term indexing.

The structure of this paper is the following. Section 10.2 introduces the unification problem, the notion of inline and post occurs checks and several unification algorithms. Section 10.3 presents implementation details of terms and relevant algorithms in the theorem prover Vampire [RV02], explains the methodology we used to measure the performance of the unification retrieval, and presents and analyses our results. To this end, we measure the performance of four unification algorithms on hundreds of millions of term pairs obtained by running Vampire on the TPTP problem library.

Section 10.4 discusses related work and contains the summary of this work.

Due to the page limit, we omit many technical details. They can be found in the full version of this paper available at `http://www.cs.man.ac.uk/\urltildahoderk/ubench/unification\_full.pdf`.

10.2 Unification Algorithms

A *unifier* of terms s and t is a substitution σ such that $s\sigma = t\sigma$. *A most general unifier* of two terms is their unifier σ such that for any other unifier τ of these two terms there exist a substitution ρ such that $\tau = \rho\sigma$. If two terms have a unifier, they also have a *most general unifier*, or simply *mgu*, which is unique modulo variable renaming. The *unification problem* is the task of finding a most general unifier of two terms.

For all existing unification algorithms, there are three possible outcomes of unification of terms s and t. It can either succeed, so that the terms are unifiable. It can fail due to a *symbol mismatch*, which means that at some point we have to unify two terms $s' = f(s_1,\ldots,s_m)$ and $t' = g(t_1,\ldots,t_n)$ such that f and g are two different function symbols. Lastly, it can fail on the *occurs check*, when we have to unify a variable x with a non-variable term containing x.

Unification algorithms can either perform occurs checks as soon as a variable has to be unified with a non-variable term, or postpone all occurs checks to the end. We call occurs checks of the first kind *inline* and of the second kind *post* occurs checks.

When we perform unification term-to-term, the post occurs check seems to perform well, also somehow confirmed by experimental results in [EIG88]. However, when we retrieve unifiers from an index, we do not build them at once. Instead, we build them incrementally as we descend down the tree performing *incremental unification*. In this case, we still have to ensure that there is no occurs check failure. It brings no additional cost to algorithms performing inline occurs check, but for post occurs check algorithms it means that the same occurs check labour may have to be performed repeatedly. On the other hand, postponing occurs check may result in a (cheap) failure on comparing function symbols. Our results in Section 10.3 confirm that algorithms using the inline occurs check outperform those based on the post occurs check.

In the rest of this paper, x,y will denote variables, f,g different function symbols, and s,t,u,v terms. We consider constants as function symbols of arity 0. All our algorithms will compute *triangle form* of a unifier. This means a substitution σ such that some power $\theta = \sigma^n$ of σ is a unifier and $\sigma^{n+1} = \sigma^n$. We will denote such θ as σ^*. When two terms have an exponential size mgu, it has a polynomial-size triangle form.

The Robinson Algorithm ROB. This is a simple unification algorithm [Rob65] with the worst-case exponential time complexity. It starts with an empty substitution σ and a stack of term pairs S that is initialized to contain a single pair (s_0,t_0) of terms s_0 and t_0 that are to be unified. At each step, we remove a term pair (s,t) from the stack and do the following.

1. If s is a variable and $s\sigma \neq s$, the pair $(s\sigma,t)$ is put on the stack S; and similar for t instead of s.

2. Otherwise, if s is a variable and $s = t$, do nothing.

3. Otherwise, if s is a variable, an occurs check is performed to see if s is a proper

subterm of $t\sigma^*$. If so, the unification fails, otherwise we extend σ by $\sigma(s) = t$.

4. Otherwise, if $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ for some function symbol f, the pairs $(s_1, t_1), \ldots, (s_n, t_n)$ are put on the stack S.

5. Otherwise, the unification fails.

When there is no pair left on the stack, σ^* is an mgu of s_0 and t_0. The occurs check is performed before each variable binding, which makes ROB an inline occurs check algorithm.

The Martelli-Montanari Algorithm MM. We call a set of terms a *multi-equation*. We say that two multi-equations M_1 and M_2 can be *merged*, if there is a variable $x \in M_1 \cap M_2$. The merge operation then replaces M_1 and M_2 by $M = M_1 \cup M_2$. A set of multi-equations is said to be in *solved form*, if every multi-equation in this set contains at most one non-variable term and no multi-equations in the set can be merged.

Let us inductively define the notion of *weakly unifiable terms s, t with the disagreement set E*, where E is a set of multi-equations.

1. If $s = t$ then s and t are weakly unifiable with the empty disagreement set.

2. Otherwise, if either s or t is a variable, then s and t are weakly unifiable with the disagreement set $\{\{s, t\}\}$.

3. Otherwise, if $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$ and for all $i = 1, \ldots, n$ the terms s_i and t_i are weakly unifiable with the disagreement set E_i, then s and t are weakly unifiable with the disagreement set $E_1 \cup \ldots \cup E_n$.

4. In all other cases s and t are not weakly unifiable.

It is not hard to argue that weak unifiability is a necessary condition for unifiability.

The Martelli-Montanari algorithm unifying terms s_0 and t_0 maintains a set of multi-equations $\mathcal{M}$, initially equal to $\{\{s_0, t_0\}\}$. Until $\mathcal{M}$ is in solved form, it merges all multi-equations in $\mathcal{M}$ that can be merged, and for each multi-equation $M \in \mathcal{M}$ containing two non-variable terms s and t, if s and t are weakly unifiable with the disagreement set E, we set $\mathcal{M} := (\mathcal{M} \setminus M) \cup \{M \setminus \{t\}\} \cup E$. If they are not weakly unifiable, the unification fails.

When $\mathcal{M}$ is in solved form, an occurs check is performed. A directed graph G is built, so that its vertices correspond to multi-equations in $\mathcal{M}$ containing non-variable

terms, and an edge is going from vertex M_1 to M_2 iff the non-variable term in M_1 contains a variable in M_2. The occurs check is successful if the graph G is acyclic. When the occurs check succeeds, one can extract the triangle form of an mgu from $\mathcal{M}$. For a proof of correctness and termination in almost linear time, see [MM82].

In our implementation of the algorithm, we maintain and merge the variable parts of multi-equations using the union-find algorithm [Tar75], and check that the graph G is acyclic using the topological sort algorithm [Kah62]. The occurs check is performed as the last step of the algorithm, which makes MM a post occurs check algorithm.

The Escalada-Ghallab Algorithm EG. In order to examine a post occurs check algorithm that aims to be practically efficient, we implemented this algorithm presented in [EIG88]. To make the algorithm competitive with inline occurs check algorithms on the incremental unification problem, we made the EG occurs check incremental. The details can be found in the full version of this paper.

PROB. Inspired by our experiments described below we implemented a modification PROB of the Robinson algorithm having polynomial worst-case time complexity. It provides an easy-to-implement polynomial-time alternative to ROB. In PROB, we keep track of term pairs that previously occurred in the stack S. When we encounter such a pair again, we simply skip it. In the occurs-check routine, we similarly maintain a set of bound variables that have already been checked or are scheduled for checking. Such variables are then skipped. In the implementation, we do not keep track of pairs that contain an unbound variable at the top. Practical results have shown that this happens frequently and that the cost of keeping track of such pairs does not pay off.

We modified all the above-mentioned algorithms to work on the *substitution tree index*, we refer the reader to [RSV01] for its description. We also had to modify algorithms to make them *incremental* so that the computed unifier can be refined to unify new pairs of terms. These incremental algorithms can be implemented to retrieve unifiable terms from a substitution tree as follows. We traverse the tree depth-first, left-to-right. When we move down the tree to a node containing a substitution $x = t$, we extend the currently computed substitution to be also a unifier of (x,t). When we return to a previously visited node, we restore the previous substitution and, in the case of MM, the previous value of $\mathcal{M}$.

10.3 Implementation and Experiments

We implemented four algorithms for retrieval of unifiers, corresponding to the unification algorithms of Section 10.2. In this section, we describe the data structures and algorithms used in the new version of Vampire [RV02].

We use shared Prolog terms to implement terms and literals. In Prolog, non-variable terms are normally implemented as a contiguous piece of memory consisting of some representation of the top symbol followed by a sequence of pointers to its subterms (actually, in the reverse order). We add to this representation *sharing* so that the same term is never stored twice. Besides conserving memory, this representation allows for constant-time equality checking. Another difference with Prolog terms is that, when an argument is a variable, Prolog stores a pointer pointing to itself, while we store the variable number.

When performing an inference on two different clauses (and in some cases even on two copies of the same clause), we must consider their variables as disjoint, although some variables may be the same, that is, have the same number. To deal with this, we use the idea of *variable banks* used in several theorem provers, including Waldmeister [HBVL97], E [Sch02] and Vampire [RV02].

Terms whose variables should be disjoint are assigned different *bank indexes*. One can view it as adding a subscript to all variables in a term — instead of terms $f(x,y)$ and $f(y,a)$, we will work with terms $f(x_0,y_0)$ and $f(y_1,a)$. In practice it means that when it is unclear from which clause a term origins, we store a pair of the term and a bank index instead of just the term.

Substitutions that store unifiers are stored as maps from pairs (variable number, bank index) to pairs (term pointer, bank index). Those maps are implemented as double hash tables[GS78] with fill-up coefficient 0.7 using two hash functions. The first one is a trivial function that just returns the variable number increased by a multiple of the bank index. This function does not give randomly distributed results (which is usually a requirement for a hash function), but is very cheap to evaluate. The second hash function is a variant of FNV. It gives uniformly distributed outputs, but is also more expensive to evaluate. It is, however, evaluated only if there is a collision of results of the first function.

The union-find data structures of EG and MM are implemented on top of these maps. In EG, we use path compression as described in [EIG88]. In MM, it turned out that the path compression led to lower performance, so it was omitted.

Benchmarking Methodology. Our benchmarking method is COMPIT [NHRV01]. First, we log all index-related operations (insertion, deletion and retrieval) performed by a first-order theorem prover. This way we obtain a description of all interactions of the prover with the index and it is possible to reproduce the indexing process without having to run the prover itself. Moreover, benchmarks generated this way can be used by other implementations, including those not based on substitution trees, and we welcome comparing our implementation of unification with other implementations.

The main difference of our benchmarking from the one presented in [NHRV01] is that instead of just success/failure, we record the number of terms unifiable with the query term. This reflects the use of unification in theorem provers, since it is used for generating inferences, so that *all* such inferences with a given clause have to be performed.

We created two different instrumentations of the development version of the Vampire prover, which used the DISCOUNT [ADF95] saturation algorithm. The first instrumentation recorded operations on the unification index of selected literals of active clauses (the *resolution index*). The second one recorded operations on the unification index of all non-variable subterms of selected literals of active clauses (the *superposition index*).

Both of these instrumentations were run on several hundred randomly selected TPTP problems with the time limit of 300s to produce benchmark data.[1] In the end we evaluated indexing algorithms on all of these benchmarks and then removed those that ran in less than 50ms, as such data can be overly affected by noise and are hardly interesting in general. This left us with about 40 percent of the original number of benchmarks,[2] namely 377 resolution and 388 superposition index benchmarks.

Results and Analysis. We compared the algorithms described above. Our original conjecture was that MM would perform comparably to ROB on most problems and be significantly better on some problems, due to its linear complexity. When this conjecture showed to be false, we added the PROB and EG algorithms, in order to find a well-performing polynomial algorithm.

On a small number of problems (about 15% of the superposition benchmarks and

[1] Recording could terminate earlier in the case the problem was proved. We did not make any distinction between benchmarks from successful and unsuccessful runs.

[2] This number does not seem to be that small, when we realise that many problems are proved in no more than a few seconds. Also note that in problems without equality there are no queries to the superposition index at all.

none of the resolution ones), the performance of ROB and MM was approximately the same ($\pm 10\%$), but on most of the problems MM was significantly slower. On the average, it was almost 6 times slower on the superposition benchmarks and about 7 times slower on the resolution benchmarks. On 3% of the superposition benchmarks and 5% of the resolution benchmarks, MM was more than 20 times slower.

The only case where MM was superior was in a handcrafted problem designed to make ROB behave exponentially containing the following two clauses:

$$p(x_0, f(x_1, x_1), x_1, f(x_2, x_2), x_2, \ldots, x_9, f(x_{10}, x_{10}));$$
$$\neg p(f(y_0, y_0), y_0, f(y_1, y_1), y_1, \ldots, y_9, f(y_{10}, y_{10}), y_{11}).$$

This problem was solved in no time by MM and PROB and in about 15 seconds by ROB.

In general, PROB has shown about the same performance as ROB. It was only about 1% slower, so it can provide a good alternative to ROB if we want to avoid the exponential worst-case complexity of the ROB. EG did not perform as bad as MM, but it was still on the average over 30% slower than ROB was.

Table 10.1 summarises the performance of the algorithms on resolution and superposition benchmarks. The first two benchmarks in each group are those on which MM performed best and worst relatively to ROB, others are benchmarks from randomly selected problems. In the table, *term size* means the number of symbols in the term; *average result count* is the average number of results retrieved by a query, and *query fail rate* is the ratio of queries that retrieved no results. The last three numbers show the use of substitutions in the index—the number of successful unification attempts, failures due to symbol mismatch, and failures due to an inline occurs check.

To determine the reason for the poor performance of MM, we used a code profiler on benchmarks displaying its worst performance. It turned out that over 90% of the measured time was being spent on performing the occurs checks, most of it actually on checking graph acyclicity. It also showed that the vast majority of unification requests were just unifying an unbound variable with a term. Based on this, we tested an algorithm performing the PROB occurs checks instead of the MM ones after such unifications. This caused the worst-case complexity to be $O(n^2)$ but improved the average performance of MM from about 600% worse than ROB did to just about 30% worse.

Our results also show empirically that the source of terms matters. For example, the MM algorithm gives relatively better performance on the superposition index, than it does on the resolution index. Other papers do not make such a distinction, for example

Problem	Time [ms] MM	EG	ROB	PROB	Relative MM	EG	All ops	Ins	Dels	Maximal index size	Avg. term size indexed	query	Avg res cnt	Query fail rate	Unification outcomes success	mism.	o.c. fail
Resolution index benchmarks																	
AGT022+2	2921	2831	2285	2303	1.3	1.2	175346	87673	0	87673	3.2	3.2	134.7	0.2	1275420	16392	0
SET317-6	51997	2600	1958	1915	26.6	1.3	68338	33440	1458	31982	10.6	10.6	52.2	0.0	2025401	5079	63
ALG229+1	1853	720	474	497	3.9	1.5	23861	11447	967	10480	7.8	7.8	54.8	0.5	420047	461	15128
ALG230+3	1490	1046	752	711	2.0	1.4	48025	23912	201	23711	2.9	2.9	40.3	0.4	768620	380	1569
CAT028+2	675	399	295	306	2.3	1.4	17989	8752	485	8267	3.6	3.6	47.5	0.2	302026	185	899
CAT029+1	3065	520	400	417	7.7	1.3	12897	6426	45	6381	11.6	11.6	114.1	0.3	498626	3	155
FLD003-1	6058	1210	941	949	6.4	1.3	14384	7187	9	7178	7.2	7.2	312.2	0.0	1247011	187	0
FLD091-3	4626	890	690	736	6.7	1.3	23037	11505	26	11479	7.7	7.7	239.4	0.0	798127	97	0
LAT289+2	1331	850	625	629	2.1	1.4	32447	16076	295	15781	3.2	3.2	36.2	0.3	608904	2678	1585
LAT335+3	1482	1002	730	742	2.0	1.4	42330	21044	242	20802	3.0	3.0	44.1	0.3	756292	252	1930
LCL563+1	4972	564	445	431	11.2	1.3	5899	2868	163	2705	14.3	14.3	135.2	0.2	441681	470	7
NUM060-1	9658	1480	1098	1104	8.8	1.3	101608	49138	3331	45807	9.8	9.8	38.4	0.0	1001317	16940	242
SET170-6	48152	2495	1864	1830	25.8	1.3	71396	35332	731	34601	10.6	10.6	49.8	0.0	1915322	5694	63
SET254-6	13807	1759	1259	1260	11.0	1.4	78914	38729	1455	37274	10.6	10.6	44.0	0.0	1244861	9979	63
SET273-6	13833	1752	1261	1268	11.0	1.4	78680	38643	1393	37250	10.7	10.7	44.0	0.0	1243272	9605	63
SET288-6	51151	2606	1946	1924	26.3	1.4	68348	33445	1458	31987	10.6	10.6	52.2	0.0	2025514	5079	63
SEU388+1	3641	821	610	603	6.0	1.3	27895	13911	73	13838	6.4	6.4	103.9	0.1	683318	10	2792
TOP031+3	1664	1089	821	831	2.0	1.3	42771	21273	225	21048	3.0	3.0	43.2	0.3	823743	3809	1808
Superposition index benchmarks																	
SEU388+1	55	54	57	53	0.96	0.9	63410	63194	200	62994	2.7	4.2	2.9	0.4	38	0	0
SET288-6	48717	2484	1808	1824	26.95	1.4	71228	35279	669	34610	10.6	10.6	49.8	0.0	1913644	5336	63
ALG229+1	80	75	71	72	1.13	1.1	63466	56466	6673	49819	4.2	10.0	4.1	0.7	1916	127	0
ALG230+3	1489	1058	764	765	1.95	1.4	49787	24780	227	24553	2.9	2.9	37.0	0.4	744009	391	1639
CAT028+2	719	432	314	321	2.29	1.4	18885	9368	149	9219	3.6	3.6	47.2	0.2	322684	238	872
CAT029+1	3073	523	387	419	7.94	1.4	12917	6436	45	6391	11.6	11.6	114.3	0.3	498651	3	155
FLD003-1	6157	1181	1010	983	6.10	1.2	14384	7187	9	7178	7.2	7.2	312.2	0.0	1246881	187	0
FLD091-3	4655	890	733	718	6.35	1.2	23003	11488	26	11462	7.7	7.7	239.7	0.0	798205	105	0
LAT289+2	1334	851	619	642	2.16	1.4	32352	16106	140	15966	3.2	3.2	36.2	0.3	605700	2753	1583
LAT335+3	1728	1150	855	862	2.02	1.3	43904	21829	246	21583	3.0	3.0	42.3	0.3	842139	4522	1797
LCL563+1	5711	636	489	503	11.68	1.3	6158	3005	148	2857	14.5	14.5	142.5	0.2	496902	497	7
NUM060-1	8820	1457	1098	1077	8.03	1.3	118475	57574	3326	54248	9.6	9.6	31.1	0.0	950386	23750	244
SET170-6	13700	1709	1255	1256	10.92	1.4	78669	38642	1385	37257	10.7	10.7	44.0	0.0	1243409	9604	63
SET254-6	13809	1725	1279	1262	10.80	1.3	78904	38724	1455	37269	10.6	10.6	44.0	0.0	1245069	9980	63
SET273-6	51081	2569	1909	1932	26.76	1.3	68156	33380	1396	31984	10.6	10.6	52.3	0.0	2023900	4718	63

Table 10.1: ROB and MM comparison on selected benchmarks

[EIG88] just states "theorem proving" as the source of the benchmarked term pairs.

10.4 Related Work and Summary

There is another comparison of ROB and MM in [ACF+91], which presents a proof that on a certain random distribution of terms the expected average (according to some measure) number of steps of ROB is constant, while the expected number of MM steps is linear in the size of terms. However, the use of random terms is essential for their results. A practical comparison of ROB, MM and EG is undertaken in [EIG88], but this comparison is not of much use for us since it is only done on a small number of examples (less than 100 term pairs alltogether), many of them handcrafted, and uses no term indexing.

There are several unification algorithms not considered here, which we did not evaluate for the reasons explained below. The Paterson algorithm [PW76] has linear asymptotic time complexity, but according to [EIG88], this benefit is redeemed by the use of complex data structures to the extent that it is mainly of theoretical interest. The Corbin-Bidoit algorithm [CB83] might look promising, as it uses an inline occurs check, but it requires input terms to be dags modified during the run of the algorithm which we cannot do because of term sharing. The Ruzicka-Privara algorithm [RP88], which is an improvement of the Corbin-Bidoit one, suffers from the same problem, and moreover uses a post occurs check.

Summary. We studied the behaviour, in the framework of term indexing, of four different unification algorithms: the exponential time Robinson algorithm, the almost linear time Martelli-Montanari and Escalada-Ghallab algorithms, and a polynomial-time modification of the Robinson algorithm. To this end, we used the appropriately modified COMPIT method [NHRV01] on a substitution tree index. The evaluation has shown that the Martelli-Montanari and Escalada-Ghallab algorithms, although asymptotically superior in the worst case, in practice behave significantly worse than the other two. The main cause of this behaviour was the occurs-check that verified acyclicity of the substitution. On the other hand, the PROB algorithm turned out to perform comparably to the Robinson one, while having the advantage of being polynomial in the worst case.

The benchmarks are available at `http://www.cs.man.ac.uk/~hoderk/`.

Bibliography

[ABD08] A. Armando, P. Baumgartner, and G. Dowek, editors. *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*. Springer, 2008.

[ACF+91] L. Albert, R. Casas, F. Fages, A. Torrecillas, and P. Zimmermann. Average case analysis of unification algorithms. In C. Choffrut and M. Jantzen, editors, *STACS*, volume 480 of *LNCS*, pages 196–213. Springer, 1991.

[ADF95] J. Avenhaus, J. Denzinger, and M. Fuchs. Discount: A system for distributed equational deduction. In *RTA'95*, pages 397–402, London, UK, 1995. Springer-Verlag.

[ARS09] Y. Puzis A. Roederer and G. Sutcliffe. Divvy: An ATP meta-system based on axiom relevance ordering. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 157–162. Springer, 2009.

[BCD+05] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[BFdNT09] P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing finite models by reduction to function-free clause logic. *J. Applied Logic*, 7(1):58–74, 2009.

[BG01] L. Bachmair and H. Ganzinger. Resolution theorem proving. In Robinson and Voronkov [RV01b], chapter 2, pages 19–99.

[BHMR07] D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant Synthesis for Combined Theories. In *Proc. of VMCAI*, volume 4349 of *LNCS*, 2007.

[BHT06] D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy Shape Analysis. In *Proc. of CAV*, pages 532–546, 2006.

[BKRW10] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic. In *Proc. of IJCAR*, pages 384–399, 2010.

[BKRW11] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. Beyond Quantifier-Free Interpolation in Extensions of Presburger Arithmetic. In *Proc. of VMCAI*, pages 88–102, 2011.

[BLS04] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec♯ Programming System: An Overview. In *Proc. of CASSIS*, volume 3362 of *LNCS*, 2004.

[Bra11] Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, 35(8):677–691, 1986.

[BST10] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[Bun99] A. Bundy. A survey of automated deduction. In *Artificial Intelligence Today*, pages 153–174. Springer-Verlag, 1999.

[CB83] J. Corbin and M. Bidoit. A rehabilitation of Robinson's unification algorithm. In *IFIP Congress*, pages 909–914, 1983.

[CCPS10] L. Correnson, P. Cuoq, A. Puccetti, and J. Signoles. *Frama-C User Manual*. CEA LIST, 2010.

[CGM+11] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. Kratos - A Software Model Checker for SystemC. In *Proc. of CAV*, pages 310–316, 2011.

[CGS08] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Theories. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 397–412, 2008.

[Cra57] W. Craig. Three uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.

[Dav81] Martin Davis. Obvious logical inferences. In Patrick J. Hayes, editor, *IJCAI*, pages 530–531. William Kaufmann, 1981.

[Dav01] M. Davis. The early history of automated deduction. In Robinson and Voronkov [RV01b], pages 3–15.

[Dd06] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[Din07] Yuzhong Ding. Several classes of BCI-algebras and their properties. *Formalized Mathematics*, 15(**1**):1–9, 2007.

[DKPW10] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In *Proc. of VMCAI*, pages 129–145, 2010.

[dMB08] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[DV01] A. Degtyarev and A. Voronkov. Equality reasoning in sequent-based calculi. In Robinson and Voronkov [RV01b], pages 611–706.

[EIG88] G. Escalada-Imaz and M. Ghallab. A practically efficient and almost linear unification algorithm. *Artif. Intell.*, 36(2):249–263, 1988.

[FQ02] C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *Proc. of POPL*, pages 191–202, 2002.

[FW09] A. Fietzke and C. Weidenbach. Labelled splitting. *Ann. Math. Artif. Intell.*, 55(1-2):3–34, 2009.

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[GLS11] A. Griggio, T. T. Hoa Le, and R. Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Linear Integer Arithmetic. In *Proc. of TACAS*, pages 143–157, 2011.

[GMT08] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting Abstract Interpreters to Quantified Logical Domains. In *Proc. of POPL*, pages 235–246, 2008.

[GPB01] Evguenii I. Goldberg, Mukul R. Prasad, and Robert K. Brayton. Using sat for combinational equivalence checking. In *DATE*, pages 114–121, 2001.

[GR09] A. Gupta and A. Rybalchenko. InvGen: An Efficient Invariant Generator. In *Proc. of CAV*, pages 634–640, 2009.

[GRS05] D. Gopan, T. W. Reps, and S. Sagiv. A Framework for Numeric Analysis of Array Operations. In *POPL*, pages 338–350, 2005.

[GS78] L. J. Guibas and E. Szemerédi. The analysis of double hashing. *J. Comput. Syst. Sci.*, 16(2):226–274, 1978.

[GT06] S. Gulwani and A. Tiwari. Combining Abstract Interpreters. In *Proc. of PLDI*, pages 376–386, 2006.

[GT07] S. Gulwani and A. Tiwari. An Abstract Domain for Analyzing Heap-Manipulating Low-Level Software. In *Proc. of CAV*, volume 4590 of *LNCS*, pages 379–392, 2007.

[HBVL97] T. Hillenbrand, A. Buch, R. Vogt, and B. Löchner. Waldmeister: High-performance equational deduction. *Journal of Automated Reasoning*, 18(2):265–270, 1997.

[HHKR10] T. Henzinger, T. Hottelier, L. Kovacs, and A. Rybalchenko. Aligators for Arrays. In *Proc. of LPAR-17*, volume 6355 of *LNAI*, pages 103–118, 2010.

[HJMM04] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from Proofs. In *Proc. of POPL*, pages 232–244, 2004.

[HKV10] K. Hoder, L. Kovacs, and A. Voronkov. Interpolation and symbol elimination in Vampire. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 188–195. Springer, 2010.

[HKV11a] K. Hoder, L. Kovács, and A. Voronkov. Case studies on invariant generation using a saturation theorem prover. In I. Z. Batyrshin and G. Sidorov, editors, *MICAI (1)*, volume 7094 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2011.

[HKV11b] K. Hoder, L. Kovacs, and A. Voronkov. Invariant Generation in Vampire. In *Proc. of TACAS*, pages 60–64, 2011.

[HKV12] K. Hoder, L. Kovács, and A. Voronkov. Playing in the grey area of proofs. In J. Field and M. Hicks, editors, *POPL*, pages 259–272. ACM, 2012.

[HP08] N. Halbwachs and M. Peron. Discovering Properties about Arrays in Simple Programs. In *Proc. of PLDI*, pages 339–348, 2008.

[HV09] K. Hoder and A. Voronkov. Comparing unification algorithms in first-order theorem proving. In M. Hund B. Mertsching and Z. Aziz, editors, *KI 2009: Advances in Artificial Intelligence, Proceedings of the 32nd German Conference on Artificial Intelligence*, LNAI 5803. Springer, 2009.

[HV11] K. Hoder and A. Voronkov. Sine qua non for large theory reasoning. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2011.

[Jan11] P. Janicic. Automated reasoning: Some successes and new challenges. In *Central European Conference on Information and Intelligent Systems*, 2011.

[JM05] R. Jhala and K. L. McMillan. Interpolant-Based Transition Relation Approximation. In *Proc. of CAV*, pages 39–51, 2005.

[JM06] R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *Prc. of TACAS*, pages 459–473, 2006.

[JM07a] R. Jhala and K. L. McMillan. Array Abstractions from Proofs. In *Proc. of CAV*, volume 4590 of *LNCS*, pages 193–206, 2007.

[JM07b] R. Jhala and K. L. McMillan. Interpolant-Based Transition Relation Approximation. *Logical Methods in Computer Science*, 3(4), 2007.

[Kah62] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962.

[KB70] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–267. Pergamon, New York, 1970.

[KL80] S. Kamin and J. J. Levy. Attempts for generalizing the recursive path ordering. *Unpublished notes*, 1980.

[KLR10] D. Kroening, J. Leroux, and P. Rümmer. Interpolating Quantifier-Free Presburger Arithmetic. In *Proc. of LPAR-17*, pages 489–503, 2010.

[KMZ06] D. Kapur, R. Majumdar, and C. G. Zarba. Interpolation for Data Structures. In M. Young and P. T. Devanbu, editors, *SIGSOFT FSE*, pages 105–116. ACM, 2006.

[Kor08] K. Korovin. iProver—an instantiation-based theorem prover for first-order logic (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR 2008*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer Verlag, 2008.

[Kor09a] K. Korovin. Instantiation-based automated reasoning: From theory to practice. In Schmidt [Sch09], pages 163–166.

[Kor09b] K. Korovin. iProver - An Instantiation-based Theorem Prover for First-order Logic (System Description). In *Proc. of IJCAR*, volume 5195 of *LNAI*, pages 292–298, 2009.

[KRS90] Jarosław Kotowicz, Konrad Raczkowski, and Paweł Sadowski. Average value theorems for real functions of one variable. *Formalized Mathematics*, 1(**4**):803–805, 1990.

[KV09a] L. Kovacs and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proc. of FASE*, pages 470–485, 2009.

[KV09b] L. Kovács and A. Voronkov. Interpolation and symbol elimination. In Schmidt [Sch09], pages 199–213.

[Len95] D. B. Lenat. CYC: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.

[LW07] M. Ludwig and U. Waldmann. An Extension of the Knuth-Bendix Ordering with LPO-Like Properties. In *Proc. of LPAR*, pages 348–362, 2007.

[McM03] K. L. McMillan. Interpolation and SAT-Based Model Checking. In *Proc. of CAV*, pages 1–13, 2003.

[McM05] K. L. McMillan. An Interpolating Theorem Prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[McM08] K. L. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 413–427, 2008.

[MJWD06] C. Matuszek, Cabral J., M. Witbrock, and J. DeOliveira. An Introduction to the Syntax and Content of Cyc. In Baral C., editor, *Proceedings of the 2006 AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, pages 44–49, 2006.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[MP09] J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic*, 7(1):41–57, 2009.

[NHRV01] R. Nieuwenhuis, T. Hillenbrand, A. Riazanov, and A. Voronkov. On the evaluation of indexing techniques for theorem proving. In *IJCAR 2001*, volume 2083 of *LNCS*, pages 257–271, 2001.

[NO05] R. Nieuwenhuis and A. Oliveras. Proof-producing congruence closure. In Jürgen Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005.

[NOT04] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and Abstract DPLL Modulo Theories. In Franz Baader and Andrei Voronkov, editors, *LPAR*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2004.

[NP01] I. Niles and A. Pease. Towards a standard upper ontology. In *FOIS*, pages 2–9, 2001.

[NR01] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In Robinson and Voronkov [RV01b], pages 371–443.

[OL80] R. A. Overbeek and E. L. Lusk. Data structures and control architectures for implementation of theorem-proving programs. In W. Bibel and R. A. Kowalski, editors, *CADE*, volume 87 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 1980.

[Pet83] G. E. Peterson. A technique for establishing completeness results in theorem proving with equality. *SIAM J. Comput.*, 12(1):82–100, 1983.

[PP91] J. Pais and G. E. Peterson. Using forcing to prove completeness of resolution and paramodulation. *J. Symb. Comput.*, 11(1/2):3–19, 1991.

[PS07] A. Pease and G. Sutcliffe. First Order Reasoning on a Large Ontology. In J. Urban, G. Sutcliffe, and S. Schulz, editors, *Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories*, 2007.

[Pud07] P. Pudlak. Semantic selection of premisses for automated theorem proving. In *ESARLT*, pages 27–44, 2007.

[PW76] M. S. Paterson and M. N. Wegman. Linear unification. In *STOC'76*, pages 181–186, New York, NY, USA, 1976. ACM.

[PY03] D. A. Plaisted and A. H. Yahya. A relevance restriction strategy for automated deduction. *Artif. Intell.*, 144(1-2):59–93, 2003.

[Ric07] Marco Riccardi. The Sylow theorems. *Formalized Mathematics*, 15(**3**):159–165, 2007.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[RP88] P. Ruzicka and I. Privara. An almost linear robinson unification algorithm. In *MFCS'88*, volume 324 of *LNCS*, pages 501–511, 1988.

[RSS07] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. In *Proc. of VMCAI*, pages 346–362, 2007.

[RSV01] I. V. Ramakrishnan, R. C. Sekar, and A. Voronkov. Term indexing. In Robinson and Voronkov [RV01b], pages 1853–1964.

[Rud87] Piotr Rudnicki. Obvious inferences. *J. Autom. Reasoning*, 3(4):383–393, 1987.

[Rud92] P. Rudnicki. An overview of the mizar project. In *University of Technology, Bastad*, pages 311–332, 1992.

[RV01a] A. Riazanov and A. Voronkov. Splitting without Backtracking. In *Proc. of IJCAI*, pages 611–617, 2001.

[RV01b] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[RV02] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Commun.*, 15(2,3):91–110, 2002.

[RV03] A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computation*, 36(1-2):101–115, 2003.

[RW69] G. Robinson and L. Wos. Paramodulation and theorem-proving in first order theories with equality. *Machine Intelligence*, 4:135–150, 1969.

[Sch02] S. Schulz. E – a brainiac theorem prover. *Journal of AI Communications*, 15(2-3):111–126, 2002.

[Sch04] S. Schulz. System Description: E 0.81. In *Proc. of IJCAR*, volume 3097 of *LNAI*, pages 223–228, 2004.

[Sch09] R. A. Schmidt, editor. *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*. Springer, 2009.

[SG09] S. Srivastava and S. Gulwani. Program Verification using Templates over Predicate Abstraction. In *PLDI*, pages 223–234, 2009.

[SKW07] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A Core of Semantic Knowledge. In *16th international World Wide Web conference (WWW 2007)*, New York, NY, USA, 2007. ACM Press.

[SP07] G. Sutcliffe and Y. Puzis. SRASS - a semantic relevance axiom selection system. In *CADE*, pages 295–310, 2007.

[SRV01] R. Sekar, I.V. Ramakrishnan, and A. Voronkov. Term indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1964. Elsevier Science, 2001.

[SS06] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[Sut06] G. Sutcliffe. Semantic Derivation Verification. *International Journal on Artificial Intelligence Tools*, 15(6):1053–1070, 2006.

[Sut08] G. Sutcliffe. CASC-J4 the 4th IJCAR ATP system competition. In Armando et al. [ABD08], pages 457–458.

[Sut09] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.

[Sut10] G. Sutcliffe. The cade-22 automated theorem proving system competition - casc-22. *AI Commun.*, 23(1):47–59, 2010.

[Sut11] G. Sutcliffe. The 5th ijcar automated theorem proving system competition - casc-j5. *AI Commun.*, 24(1):75–89, 2011.

[Sut12] G. Sutcliffe. The cade-23 automated theorem proving system competition - casc-23. *AI Commun.*, page to appear, 2012.

[Tar75] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[TS06] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38–47, 2006.

[UHV10] J. Urban, K. Hoder, and A. Voronkov. Evaluation of automated theorem proving on the Mizar Mathematical Library. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *ICMS*, volume 6327 of *Lecture Notes in Computer Science*, pages 155–166. Springer, 2010.

[Urb06] J. Urban. Mptp 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning*, 37(1-2):21–43, 2006.

[US08] Josef Urban and Geoff Sutcliffe. ATP-based cross-verification of Mizar proofs: Method, systems, and first experiments. *Mathematics in Computer Science*, 2(2):231–251, 2008.

[US10] Josef Urban and Geoff Sutcliffe. Automated reasoning and presentation support for formalizing mathematics in Mizar. In Serge Autexier, editor, *AISC 2010*, LNAI. Springer, 2010. To appear.

[USPV08] J. Urban, G. Sutcliffe, P. Pudlák, and J. Vyskocil. MaLARea SG1 - machine learner for automated reasoning with semantic guidance. In Armando et al. [ABD08], pages 441–456.

[Vor95] A. Voronkov. The anatomy of Vampire: Implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2):237–265, 1995.

[VSU10] Jiri Vyskocil, David Stanovsky, and Josef Urban. Automated proof shortening by invention of new definitions. In *LPAR 2010*, Lecture Notes in Artificial Intelligence. Springer, 2010. to appear.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In Schmidt [Sch09], pages 140–145.

[Wei01] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.

[Wie00] Freek Wiedijk. CHECKER - notes on the basic inference step in Mizar. available at http://www.cs.kun.nl/∼freek/mizar/by.dvi, 2000.

[Wie07] Freek Wiedijk. Arrow's impossibility theorem. *Formalized Mathematics*, 15(**4**):171–174, 2007.

[WSH+07] C. Weidenbach, R. A. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic. System Description: SpassVersion 3.0. In *Proc. of CADE*, volume 4603 of *LNAI*, pages 514–520, 2007.

[YM05] G. Yorsh and M. Musuvathi. A Combination Method for Generating Interpolants. In *Proc. of CADE*, pages 353–368, 2005.