TU WIEN Informatics

Efficient Reasoning with Quantifiers and Theories

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Jakob Rath, BSc
Registration Number 00825352

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof.in Dr.in techn. Laura Kovács, MSc
Co-advisor: Prof. Dr. Armin Biere
Co-advisor: Dr. Nikolaj Bjørner

The dissertation has been reviewed by:

Elvira Albert

Stephan Schulz

Vienna, October 21, 2024

Jakob Rath

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Jakob Rath, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 21. Oktober 2024

Jakob Rath

Acknowledgements

I am very grateful to my advisor Laura Kovács for the supervision of this work and for being available for numerous meetings over the course of my doctoral studies. I am also very grateful to my co-advisors Armin Biere and Nikolaj Bjørner, who were available for numerous technical discussions and provided many helpful suggestions. Many thanks go also to my co-authors Bernhard Gleiss, Clemens Eisenhofer, Daniela Kaufmann, Michael Rawson, Petra Hozzová, Robin Coutelier, who all were a joy to work with. I truly appreciate the support of Beatrix Buhl, which was invaluable to navigate the various bureaucratic requirements of TU Wien. I feel fortunate to have been part of the APRe group, and am grateful to all my colleagues at Forsyte for creating an amazing work environment. I also want to thank Elvira Albert and Stephan Schulz for taking their time to review this thesis and providing helpful comments and suggestions.

Finally, I want to thank my parents Josef and Elfriede and my sisters Maria and Judith. Without your support I could have never made it this far.

The research in this thesis was partially supported by the ERC Starting Grant 2014 SYMCAR 639270, the ERC Proof of Concept Grant 2018 SYMELS 842066, the Wallenberg Academy Fellowship 2014 TheProSE, the Austrian FWF research project W1255-N23, the OMAA grant 101öu8, the WWTF ICT15-193 grant, the ERC Consolidator Grant ARTIST 101002685, the TU Wien Doctoral College LogiCS, the TU Wien Doctoral College SecInt, the FWF SFB project SpyCoDe F8504, the FWF ESPRIT grant 10.55776/ESP666, and the Amazon Research Award 2023 QuAT.

Abstract

Automated reasoners, such as SAT and SMT solvers as well as first-order theorem provers, are becoming the backbones of applications of formal methods, for example in automating deductive verification, program synthesis, and security analysis. Automation in these formal methods domains crucially depends on the efficiency of the underlying reasoners towards finding proofs and/or counterexamples of the task to be enforced. The goal of this thesis is to improve efficiency of automated reasoning on different levels, inspired by automation of formal methods.

The first part of the thesis deals with improving efficiency of saturation-based first-order theorem proving. Such theorem provers use dedicated proof rules to keep proof search manageable.

Inspired by applications in program verification, we introduce a new inference rule, called subsumption demodulation, to improve support for reasoning with conditional equalities in superposition-based theorem proving. We show that subsumption demodulation is a simplification rule that does not require radical changes to the underlying superposition calculus and hence can be efficiently integrated in superposition provers. We implement subsumption demodulation in the theorem prover Vampire, by extending Vampire with a new clause index and adapting its multi-literal matching component. Our experiments, using the TPTP and SMT-LIB benchmark repositories, show that subsumption demodulation in Vampire can solve several new problems that previously could not be solved by state-of-the-art reasoners.

Next, we turn our attention to subsumption, which is one of the most important proof rules in practice. It is common that millions of subsumption checks are performed during proof search, necessitating efficient implementations. However, in contrast to propositional subsumption as used by SAT solvers and implemented using sophisticated polynomial algorithms, first-order subsumption in first-order theorem provers involves NP-complete search queries, turning the efficient use of first-order subsumption into a huge practical burden. In this thesis, we argue that integration of a dedicated SAT solver provides a remedy towards efficient implementation of first-order subsumption and related rules, and thus further increasing scalability of first-order theorem proving towards applications of formal methods. Our experimental results demonstrate that, by using a tailored SAT solver within first-order reasoning, we gain a large speed-up in state-of-the-art benchmarks.

In the second part of the thesis, we examine bit-vector reasoning within SMT solving. We introduce PolySAT, a word-level decision procedure supporting bit-precise SMT reasoning over polynomial arithmetic with large bit-vector operations. The PolySAT calculus extends conflict-driven clause learning modulo theories with two key components: (i) a bit-vector plugin to the equality graph, and (ii) a theory solver for bit-vector arithmetic with non-linear polynomials. PolySAT implements dedicated procedures to extract bit-vector intervals from polynomial inequalities. For the purpose of conflict analysis and resolution, PolySAT comes with on-demand lemma generation over non-linear bit-vector arithmetic. PolySAT is integrated into the SMT solver Z3 and has potential applications in model checking and smart contract verification where bit-blasting techniques on multipliers/divisions do not scale.

Kurzfassung

Automatische Beweiser wie SAT- und SMT-Solver, sowie Beweiser für Prädikatenlogik erster Stufe, werden immer wichtigere Komponenten in zahlreichen Anwendungen, vor allem im Bereich der formalen Methoden, wie beispielsweise in Software zur automatisierten deduktiven Verifikation, Programmsynthese und der Sicherheitsanalyse von Programmen. Insbesondere im Bereich der formalen Methoden hängt der Automatisierungsgrad entscheidend von der Effizienz der zugrunde liegenden Beweiser bei der Suche nach Beweisen und/oder Gegenbeispielen für die zu lösende Aufgabe ab. Das Ziel der vorliegenden Dissertation ist es, motiviert durch die Anwendungen im Bereich der automatischen Analyse von Programmen, die Effizienz der automatisierten Beweiser in verschiedenen Bereichen zu verbessern.

Der erste Teil der Arbeit beschäftigt sich mit der Optimierung von auf Saturierung basierenden Theorembeweisern für Prädikatenlogik erster Stufe. Beweiser dieser Art verwenden Inferenzregeln, um schrittweise Konsequenzen aus der Eingabe herzuleiten. Eine besondere Herausforderung ist dabei die enorme Menge an möglichen Konsequenzen, die die Suche nach Beweisen empfindlich verlangsamt. Einen gängigen Ansatz zur Verbesserung dieses Problems bieten sogenannte Vereinfachungsregeln: spezielle Inferenzregeln, deren Verwendung die Anzahl redundanter Klauseln reduziert oder zumindest nicht weiter erhöht.

Wir führen eine neue Inferenzregel namens *Subsumption Demodulation* ein, um das Erkennen von Redundanzen im Zusammenhang mit bedingten Gleichungen in Beweisern, die auf dem Superpositionskalkül basieren, zu verbessern. Wir zeigen, dass *Subsumption Demodulation* eine Vereinfachungsregel ist, die keine fundamentalen Änderungen am zugrunde liegenden Superpositionskalkül erfordert. Durch Erweiterung um einen neuen Klauselindex und Anpassung der Multiliteral-Matching-Komponente konnten wir *Subsumption Demodulation* effizient in dem Theorembeweiser Vampire implementieren. Unsere Experimente zeigen, dass Vampire mit Hilfe der in dieser Dissertation beschriebenen Änderungen mehrere neue Probleme aus den Benchmark-Sammlungen TPTP und SMT-LIB lösen kann, die bisher von modernen Beweisern nicht gelöst werden konnten.

Darüber hinaus wenden wir uns der *Subsumption*-Regel zu, die eine der wichtigsten Beweisregeln in der Praxis ist. Es kommt häufig vor, dass während der Beweissuche Millionen von potenziellen Anwendungen von *Subsumption* geprüft werden, weshalb eine effiziente Implementierung erforderlich ist. Im Gegensatz zu *Subsumption* in der

Aussagenlogik, die von SAT-Solvern verwendet wird und ausgefeilte Algorithmen mit polynomieller Laufzeit erlaubt, erfordert *Subsumption* in der Prädikatenlogik erster Stufe die Lösung eines NP-vollständigen Entscheidungsproblems. Dadurch ist die effiziente Nutzung von *Subsumption* in der Prädikatenlogik erster Stufe eine große praktische Herausforderung. In der vorliegenden Arbeit zeigen wir, dass die Integration eines dedizierten SAT-Solvers eine gute Möglichkeit zur effizienten Implementierung von *Subsumption* und verwandter Regeln bietet und damit die Skalierbarkeit des Theorembeweisens in Logik erster Stufe für Anwendungen formaler Methoden verbessert. Unsere experimentellen Ergebnisse zeigen, dass wir durch die Verwendung eines maßgeschneiderten SAT-Solvers innerhalb von Vampire einen großen Geschwindigkeitszuwachs in aktuellen Benchmarks erzielen.

Im zweiten Teil der Dissertation betrachten wir Bitvektoren im Kontext von SMT-Solvern. Wir stellen PolySAT vor, ein Entscheidungsverfahren auf Wortebene, das bit-präzises Schließen über polynomiale Arithmetik mit Operationen über Bitvektoren großer Länge erlaubt. Der PolySAT-Kalkül erweitert konfliktgesteuertes Klauselnernen modulo Theorien um zwei Schlüsselkomponenten: (i) ein Bitvektor-Plugin für den Gleichheitsgraphen, und (ii) einen Theorielöser für Bitvektor-Arithmetik mit nicht-linearen Polynomen. PolySAT implementiert spezielle Verfahren zur Extraktion von Bitvektor-Intervallen aus polynomialen Ungleichungen. Zur Konfliktanalyse und -resolution generiert PolySAT *on demand* Lemmas über nicht-linearer Bitvektor-Arithmetik. PolySAT wurde in den SMT-Solver Z3 integriert und hat potenzielle Anwendungen in der Modellprüfung und der Verifikation von Smart Contracts, wo die üblicherweise verwendete Methode des *bit-blasting* auf Multiplikationen/Divisionen an ihre Grenzen stößt.

Contents

CHAPTER 1

Introduction

Automated reasoning provides the foundation on which many formal verification applications in modern computer science are built. Such approaches use automated reasoners in their backend to, for example, discharge verification conditions [Lei17, CMP20, GGK20], produce/block counter-examples [KGC16, PMP+16, ABH+20], or enforce security and privacy properties [PFG20, MAD+19, BEG+19, SGSM20]. All these approaches crucially depend on the efficiency of the underlying reasoning procedures.

Depending on the concrete application, different levels of expressivity may be desired for the backend logic. As a rule of thumb, more expressivity in the language leads to higher complexity of the solving process and thus to more performance problems (i.e., timeouts). On the other hand, even in cases where it is possible, it may not be desirable to compile a problem down into a less expressive language, because a well-structured high-level representation in an expressive language may allow the solver to use shortcuts that are not easily discovered when working in the low-level representation. In practice, common target logics for automatic solving are propositional logic and first-order logic, with many different dialects of the latter that vary in which quantifiers and theories are allowed.

In this thesis, we are interested in the following three levels of automated reasoning, addressing increasingly complex logics:

L1. *SAT Solving* [Bie08] to solve propositional formulas (Section 1.2)

L2. *SMT Solving* [dMB08, BBB+22], which extends SAT Solving to solve formulas in certain fragments of first-order logic with theories (Section 1.3)

L3. *Saturation-based Theorem Proving* [WDF+09, KV13, Cru17, SCV19] for solving formulas in first-order logic with complete support for quantifiers (Section 1.1)

The thesis is split into two parts. Part I, consisting mainly of chapters 3 and 4, focuses on saturation-based theorem proving and combines levels L1 and L3. Chapter 3 introduces

the rule of *subsumption demodulation* with the aim of improving reasoning power for program analysis and verification [BEG+19]. In a nutshell, subsumption demodulation performs equational rewriting under side constraints, which allows the theorem prover to more easily handle constraints arising from statements within loops. Chapter 4 enhances the efficiency of superposition-based theorem provers as a whole by introducing a new method of performing *subsumption inferences*. Subsumption is a crucial inference rule in theorem proving, but is expensive to check and thus often accounts for a large portion of the prover's running time. To tackle this problem, we encode subsumption checks into propositional formulas, which are then checked by a specialized SAT solver.

Part II of the thesis focuses on the theory of bit-vectors within SMT solving and advances level L2. The state of the art in bit-vector solving is bit-blasting, i.e., translating the bit-vector formula into propositional logic and passing control to a SAT solver. As the size of bit-vectors grows, however, bit-blasting soon reaches its limits of scalability. To try and overcome this limitation, we present our new PolySAT framework, a word-level approach to bit-vector solving.

1.1 Saturation-Based Theorem Proving

The leading concept behind the proof-search algorithms used by state-of-the-art first-order theorem provers is *saturation* [SCV19, KV13]. While the concept of saturation is relatively unknown outside of the theorem proving community, similar algorithms that are used in other areas, such as Gröbner basis computation [Buc06], can be considered examples of saturation algorithms. The key idea behind saturation-based proof search is to reduce the problem of proving validity of a first-order formula F to the problem of establishing unsatisfiability of $\neg F$ by using a sound inference system, most commonly using the superposition inference system [NR01]. That is, instead of proving F, we refute $\neg F$, by selecting and applying inferences from the superposition calculus. In this work, we focus on saturation algorithms using the superposition calculus.

For the efficiency of saturation-based proof search, simplification rules are of critical importance. Simplification rules are inference rules that do not add new formulas to the clause database, but instead simplify formulas by deleting redundant clauses from the database. As such, simplification rules reduce the size of the proof search space and are crucial in making automated reasoning efficient.

When reasoning about properties of first-order logic with equality, one of the most common simplification rules is demodulation [KV13] for rewriting (and hence simplifying) formulas using unit equalities $l \simeq r$, where l, r are terms and $\simeq$ denotes equality. As a special case of superposition, demodulation is implemented in first-order provers such as E [SCV19], Spass [WDF+09] and Vampire [KV13]. Recent applications of superposition-based reasoning, for example to program analysis and verification [BEG+19], demand however new and efficient extensions of demodulation to reason about and simplify upon conditional equalities $C \rightarrow l \simeq r$, where C is a first-order formula. Such conditional equalities may, for example, encode software properties expressed in a guarded command

language, with C denoting a guard (such as a loop condition) and $l \simeq r$ encoding equational properties over program variables. We illustrate the need of considering generalized versions of demodulation in the following example.

Example 1. Consider the following formulas expressed in the first-order theory of linear integer arithmetic:

$$\begin{array}{l} f(i) \simeq g(i) \\ 0 \leq i < n \rightarrow p(f(i)) \end{array} \tag{1.1}$$

Here, i is an implicitly universally quantified logical variable of integer sort, and n is an integer-valued constant symbol. Saturation-based reasoners will first clausify formulas (1.1), deriving:

$$\begin{array}{l} f(i) \simeq g(i) \\ 0 \not\leq i \vee i \not< n \vee p(f(i)) \end{array} \tag{1.2}$$

By applying demodulation over (1.2), the formula $0 \not\leq i \vee i \not< n \vee p(f(i))$ is rewritten[1] using the unit equality $f(i) \simeq g(i)$, yielding the clause $0 \not\leq i \vee i \not< n \vee p(g(i))$. In terms of the original formula, this step corresponds to deriving $0 \leq i < n \rightarrow p(g(i))$ from (1.1) by one application of demodulation.

Let us now consider a slightly modified version of (1.1), as below:

$$\begin{array}{l} 0 \leq i < n \rightarrow f(i) \simeq g(i) \\ 0 \leq i < n \rightarrow p(f(i)) \end{array} \tag{1.3}$$

The clausal representation of (1.3) is given by:

$$\begin{array}{l} 0 \not\leq i \vee i \not< n \vee f(i) \simeq g(i) \\ 0 \not\leq i \vee i \not< n \vee p(f(i)) \end{array} \tag{1.4}$$

It is again obvious that from (1.3) one can derive the formula $0 \leq i < n \rightarrow p(g(i))$, or equivalently, the clause:

$$0 \not\leq i \vee i \not< n \vee p(g(i)) \tag{1.5}$$

Yet, one cannot anymore apply demodulation over (1.4) to derive such a clause, as (1.4) does not contain any unit equality. □

Contributions to saturation-based theorem proving. In Chapter 3, we propose a generalized version of demodulation, called *subsumption demodulation*, which allows to rewrite terms and simplify formulas using rewriting based on conditional equalities, such as in (1.3). To do so, we extend demodulation with subsumption, that is, with deciding whether (an instance of a) clause C is a submultiset of a clause D. In particular, the non-equality literals of the conditional equality (i.e., the condition) need to subsume the unchanged literals of the simplified clause. This way, subsumption demodulation can be applied to non-unit clauses and is not restricted to have at least one premise clause that

[1] assuming that g is simpler/smaller than f

is a unit equality. We show that subsumption demodulation is a simplification rule of the superposition framework (Section 3.1), allowing for example to derive the clause (1.5) from (1.4) in one inference step. By properly adjusting clause indexing and multi-literal matching in first-order theorem provers, we provide an efficient implementation of subsumption demodulation in Vampire (Section 3.3) and evaluate our work against state-of-the-art reasoners, including E [SCV19], Spass [WDF+09], CVC4 [BCD+11] and Z3 [dMB08] (Section 3.4).

In summary, Chapter 3 of the thesis brings the following results.

- To improve reasoning in the presence of conditional equalities, we introduce the new inference rule *subsumption demodulation*, which generalizes demodulation to non-unit equalities by combining demodulation and subsumption (Section 3.1).
- Subsumption demodulation does not require radical changes to the underlying superposition calculus. We implemented subsumption demodulation in the first-order theorem prover Vampire, by extending Vampire with a new clause index and adapting its multi-literal matching component (Section 3.3).
- We compared our work against state-of-the-art reasoners, using the TPTP and SMT-LIB benchmark repositories. Our experiments show that subsumption demodulation in Vampire can solve 11 first-order problems that could previously not be solved by any other state-of-the-art provers, including Vampire, E, Spass, CVC4 and Z3 (Section 3.4).

1.2 Saturation-Based Theorem Proving and SAT Solving

We remain in the setting of saturation-based theorem proving as in the previous Section 1.1, and expand on the importance of redundancy. During saturation, the prover keeps a set of *usable clauses* $C_1, \dots C_k$, with $k \geq 0$. This is the set of clauses that the prover considers as possible premises for inferences. After applying an inference with one or more usable clauses as premises, the consequence C_{k+1} is added to the set of usable clauses. The number of usable clauses is an important factor for the efficiency of proof search. A naive saturation algorithm that keeps all derived clauses in the usable set would not scale in practice. One reason is that first-order formulas in general yield infinitely many consequences.

Example 2. Consider the clause

$$\neg \mathit{positive}(x) \vee \mathit{positive}(\mathit{reverse}(x)), \tag{1.6}$$

where x is a universally quantified variable ranging over the algebraic data type `list`, where list elements are integers; *positive* is a unary predicate over `list` such that $\mathit{positive}(x)$ is valid iff all elements of x are positive integers; and *reverse* is a unary function symbol reversing a list. As such, clause (1.6) asserts that the reverse of a list x of positive integers is also a list of positive integers (which is clearly valid).

Note that, when having clause (1.6) as a usable clause during proof search, the clause $\neg positive(x) \vee positive(reverse^n(x))$ can be derived for any $n \geq 1$ from clause (1.6). Adding $\neg positive(x) \vee positive(reverse^n(x))$ to the set of usable clauses would however blow up the clause database unnecessarily. This is because $\neg positive(x) \vee positive(reverse^n(x))$ is a logical consequence of clause (1.6), and hence, if a formula F can be proved using $\neg positive(x) \vee positive(reverse^n(x))$, then F is also provable using clause (1.6). Yet, storing $\neg positive(x) \vee positive(reverse^n(x))$ as usable formulas is highly inefficient as n can be arbitrarily large.

Saturation with Redundancy. To avoid such and similar cases of unnecessarily increasing the set of usable formulas during proof search, first-order theorem provers implement the notion of *redundancy* [Rob65], by extending the standard superposition calculus with term/clause ordering and literal selection functions. These orderings and selection functions are used to eliminate so-called redundant clauses from the clause database, where redundant clauses are logical consequences of smaller clauses w.r.t. the considered ordering. In Example 2, the clause $\neg positive(x) \vee positive(reverse^n(x))$ would be a redundant clause as it is a logical consequence of clause (1.6), with clause (1.6) being smaller (i.e., using fewer symbols) than $\neg positive(x) \vee positive(reverse^n(x))$. As such, if clause (1.6) is already a usable clause, saturation algorithms implementing redundancy should ideally not store $\neg positive(x) \vee positive(reverse^n(x))$ as usable clauses. To detect and reason about redundant clauses, saturation algorithms with redundancy extend the superposition inference system with so-called *simplification rules* and *deletion rules*. Simplification and deletion rules do not add new formulas to the set of (usable) clauses, but instead simplify and delete redundant formulas from the clause database, respectively, without destroying the refutational completeness of superposition: if a formula F is valid, then $\neg F$ can be refuted using the superposition calculus extended with simplification rules. In Example 2, this means that if $\neg F$ can be refuted using $\neg positive(x) \vee positive(reverse^n(x))$, then $\neg F$ can be refuted in the superposition calculus extended with simplification rules, without using $\neg positive(x) \vee positive(reverse^n(x))$ but using clause (1.6) instead.

Ensuring that simplification rules are applied efficiently for eliminating redundant clauses is, however, not trivial. In Chapter 4, we show that SAT-based approaches can be used to identify the application of simplification rules during saturation, improving thus the efficiency of saturation algorithms implementing the superposition calculus extended with simplification rules, as discussed next.

Contributions to saturation-based theorem proving with SAT solving. While redundancy is a powerful criterion for keeping the set of clauses used in proof search as small as possible, establishing whether an arbitrary first-order formula is redundant is as hard as proving whether it is valid. For example, in order to derive that $\neg positive(x) \vee positive(reverse^n(x))$ is redundant in Example 2, the prover should establish (among other conditions) that it is a logical consequence of (1.6), which essentially requires proving based on superposition. To reduce the burden of establishing redundancy, first-

order provers implement sufficient conditions towards deriving redundancy, so that these conditions can be efficiently checked (ideally using only syntactic arguments, and no proofs). One such condition comes from the notion of *subsumption*, yielding one of the most impactful simplification rules in superposition-based theorem proving [BG94].

The intuition behind subsumption is that a (potentially large) instance of a clause C does not convey any additional information over C, and thus it should be avoided to have both C and its instance in the set of usable clauses; to this end, we say that the instance of C is subsumed by C. More formally, a clause C subsumes another clause D if there is a substitution σ such that $\sigma(C)$ is a submultiset[2] of D. In such a case, subsumption removes the subsumed clause D from the clause set. To continue Example 2, a unit clause $positive(reverse^m(x))$, with $m \geq 1$, would prevent us from deriving $\neg positive(x) \vee positive(reverse^n(x))$ for any $n \geq m$, and hence eliminate an infinite branch of clause derivations from the search space.

To detect possible inferences of subsumption and related rules, state-of-the-art provers use a two-step approach [SRV01]: (i) retrieve a small set of candidate clauses, using literal filtering methods, and then (ii) check whether any of the candidate clauses represents an actual instance of the rule. Step (i) has been well-researched over the years, leading to highly efficient indexing solutions [NHRV01, Sch13, SRV01]. Interestingly, step (ii) has not received much attention, even though it is known that checking subsumption relations between multi-literal clauses is an NP-complete problem [KN86]. Although indexing in step (i) allows the first-order prover to skip step (ii) in many cases, the application of (ii) in the remaining cases may remain problematic (due to NP-hardness). For example, while profiling subsumption in the world-leading theorem prover Vampire [KV13], we observed subsumption applications, and in particular calls to the literal-matching algorithm of step (ii), that consume more than 20 seconds of running time. Given that millions of such matchings are performed during a typical first-order proof attempt, we consider such cases highly inefficient, calling for improved solutions towards step (ii). In Chapter 4 we address this demand and show that a *tailored SAT-based encoding can significantly improve the literal matching, and thus subsumption, in first-order theorem proving.*

Chapter 4 brings the following main contributions.

- We propose a SAT-based encoding for capturing potential applications of subsumption in first-order theorem proving (Section 4.2). A solution to our SAT-based encoding gives a concrete application of subsumption, allowing the first-order prover to apply that instance of subsumption as a simplification rule during saturation. Our encoding uses so-called substitution constraints (Section 4.1) to formalize matching of literals within the premises (i.e., subset relation among literals of premises). Our encoding can be extended to other simplification rules, in particular when applying simplifications using the combination of subsumption with binary resolution (i.e., subsumption resolution).

[2] we consider a clause C as a multiset of its literals

- We introduce a lightweight SAT solving approach tailored to substitution constraints, by adjusting unit propagation and conflict resolution towards efficient handling of such constraints (Section 4.3). We introduce a tailored encoding of substitution constraints in SAT solving, advocating the direct use of our SAT solver for deciding application of subsumption within first-order proving.
- We implement our SAT-based subsumption approach as a new SAT solver in the theorem prover Vampire (Section 4.4). We empirically evaluate our approach on the standard benchmark library TPTP (Section 4.5). Our experiments demonstrate that using SAT solving for deciding and applying subsumption brings clear improvements in the saturation process of first-order proving, for example improving the (time) performance of the prover by a factor of 2.

1.3 SMT Solving for Bit-Vector Reasoning

Bit-vector reasoning plays a central role in applications of system verification, enabling for example efficient bounded model checking [CKL04], bit-precise memory handling [LLH+21], or proving safety of decentralized financial transactions [AGR+20]. Although one may argue that, because bit-vectors are bounded, bit-vector reasoning is simpler than proving arithmetic properties over the integers or reals, showing (un)satisfiability of bit-vector problems is inherently expensive due to complex arithmetic operations over large bit-widths [KFB16].

In Part II of this thesis, we propose PolySAT, a *word-level reasoning procedure* as a theory solver integrated into SMT solving. PolySAT is based on conflict-driven clause learning modulo theories (CDCL(T)), providing thus *an alternative to bit-blasting*. Our work builds on and extends previous research on bit-vector slicing [BS09], forbidden intervals [GJD20], and fixing bits [ZWR16]. From Part I, we apply subsumption resolution to simplify conflict clauses in some cases.

In our setting, we consider bit-vectors as elements of the ring $\mathbb{Z}/2^w\mathbb{Z}$. Informally, arithmetical operations on bit-vectors can be seen as the respective integer operations, where the result is evaluated "mod 2^w". Yet, due to modulo/bounded arithmetic, many properties of the integers (such as, there is no maximal element and no zero-divisors) do not hold over bit-vectors. Nevertheless, with PolySAT we support bit-vector arithmetic without bit-blasting.

Example 3. Let us illustrate the benefits of PolySAT using the following bit-vector constraints with large bit-width w:

$$xy + y >_\mathsf{u} y + 3 \qquad 1 = 3x + 6yz + 3z^2$$
$$6 = 2y + z \qquad 0 = (2y + 1)\ \&\ x$$

where "&" denotes the bit-wise *and* operation and $>_\mathsf{u}$ refers to unsigned comparison. PolySAT proves this set of bit-vector constraints to be Unsat, without using bit-blasting as follows.

We guess the assignment $x = 0$, simplifying the first constraint to $y >_u y + 3$. We pick the assignment $y = 2^w - 2$ which is feasible w.r.t. the inequality. Hence, the constraint $6 = 2y + z$ simplifies to $z = 10$, which conflicts with the constraint $1 = 3x + 6yz + 3z^2$. We backtrack, apply variable elimination upon y on the two equality constraints, and learn the equation $3x + 18z = 1$. From the bit-wise &-constraint, we derive that x is even, as $2y + 1$ is odd. This, however, conflicts with the learned clause, as it implies that x is odd. Hence, PolySAT concludes that the given constraints are Unsat.

Contributions to SMT Solving. With PolySAT (Chapters 7–12) we bring the following main improvements to word-level reasoning over bit-vectors.

- We adjust the concept of forbidden intervals [GJD20] to track viable values in PolySAT (Chapter 8);
- We extract bit-vectors intervals from polynomial (non-linear) inequalities (Chapter 9);
- We introduce lemmas on-demand for detecting and resolving non-linear conflicts in PolySAT (Chapter 10);
- We implement PolySAT directly in the SMT solver Z3 [dMB08] and evaluate our work on challenging examples (Chapter 11).

1.4 Publications

The thesis at hand is based on the following publications:

[GKR20] Bernhard Gleiss, Laura Kovács, and Jakob Rath. Subsumption Demodulation in First-Order Theorem Proving. In *Proceedings of IJCAR*, pages 297–315, 2020. doi:10.1007/978-3-030-51074-9_17

[RBK22] Jakob Rath, Armin Biere, and Laura Kovács. First-Order Subsumption via SAT Solving. In *Proceedings of FMCAD*, pages 160–169, 2022. doi:10.34727/2022/ISBN.978-3-85448-053-2_22

[REK+24] Jakob Rath, Clemens Eisenhofer, Daniela Kaufmann, Nikolaj Bjørner, and Laura Kovács. PolySAT: Word-level Bit-vector Reasoning in Z3, 2024. Accepted for VSTTE 2024. arXiv:2406.04696

I have been the main author of [RBK22] and [REK+24]. The work in [GKR20] has been led and implemented by me, including devising and integrating subsumption demodulation in saturation.

As part of my PhD research, the following additional works have also been published.

[HKR21] Petra Hozzová, Laura Kovács, and Jakob Rath. Automated Generation of Exam Sheets for Automated Deduction. In *Proceedings of CICM*, pages 185–196, 2021. doi:10.1007/978-3-030-81097-9_15

[CKRR23] Robin Coutelier, Laura Kovács, Michael Rawson, and Jakob Rath. SAT-Based Subsumption Resolution. In *Proceedings of CADE*, pages 190–206, 2023. doi:10.1007/978-3-031-38499-8_11

[CRR+24] Robin Coutelier, Jakob Rath, Michael Rawson, Armin Biere, and Laura Kovács. SAT Solving for Variants of First-Order Subsumption. *Formal Methods in System Design*, 2024. doi:10.1007/s10703-024-00454-1

Part I

On Saturation-Based Theorem Proving

CHAPTER 2

Background

We begin by defining the relevant vocabulary. In this chapter, we encounter both standard multi-sorted first-order logic with equality (for the target formula to be proved, and intermediate formulas), as well as standard propositional logic (as compilation target for sub-problems to be solved).

We first consider standard multi-sorted first-order logic with equality, where the equality predicate is denoted by $\simeq$. We support the standard Boolean connectives (negation $\neg$, conjunction $\wedge$, disjunction $\vee$, implication $\rightarrow$, equivalence $\leftrightarrow$) and quantifiers (universal $\forall$, existential $\exists$) in the language. We write $F \models G$ to indicate that formula G holds whenever formula F holds, or equivalently, that $F \rightarrow G$ is valid.

Let $\mathcal{V}$ be a countably infinite set of first-order *variables*. Throughout the chapter, we write x, y, z for first-order variables (i.e., elements of $\mathcal{V}$), c, d for constant symbols, f, g for function symbols, and p, q for predicate symbols. The set of first-order *terms* $\mathcal{T}$ is constructed inductively: variables and constant symbols are terms, and complex terms are built by applying function symbols to existing terms. We denote terms by l, r, s, t.

All notation throughout this text may use indices, and we may drop the qualifier "first-order" when it is clear from the context.

First-order *atoms* are predicates applied to terms. Atoms and negated atoms are also called first-order *literals*, and we denote them by L, M. We further call a literal for the form $s \simeq t$ an *equality literal*, and write $s \not\simeq t$ for the literal $\neg(s \simeq t)$.

First-order *clauses* are disjunctions of literals and denoted by C, D. A clause that consists of a single literal is called a *unit clause*, and a clause that consists of a single equality literal is called a *unit equality*. Clauses are often viewed as multisets of literals; that is, a clause $C = L_1 \vee L_2 \vee \cdots \vee L_n$ is considered to be the multiset $\{L_1, L_2, \ldots, L_n\}$. For example, the clause $p \vee \neg q \vee p$ is the multiset $\{p, p, \neg q\}$. The empty clause is denoted by $\square$.

An expression E is a term, literal, or clause. We denote the set of variables occurring in the expression E by $\mathcal{V}(E)$. A *substitution* is a function $\sigma\colon \mathcal{V} \to \mathcal{T}$ such that $\sigma(x) \neq x$ only for finitely many $x \in \mathcal{V}$. As a shorthand, we also write $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ for the substitution that maps x_i to t_i for $i \in \{1, \ldots, n\}$ and leaves y unchanged for $y \notin \{x_1, \ldots, x_n\}$. The function σ is extended to arbitrary expressions E by simultaneously replacing each variable x in E by $\sigma(x)$. Traditionally in logic, $\sigma(E)$ is often written in postfix notation as $E\sigma$.

An expression E is said to be *ground* if it does not contain any variables.

To formally define replacement of a distinguished occurrence of a subterm s within an expression E, consider an expression E' with a single occurrence of a fresh variable z such that $E'\{z \mapsto s\} = E$ (i.e., put a "hole" z in place of the distinguished occurrence of s). We can then obtain the version of E with the distinguished occurrence of s replaced by a term t as $E'\{z \mapsto t\}$. As a shorthand for such constructions, we write $E[s]$ (e.g., in the premise of an inference rule) to indicate a distinguished occurrence of the term s within the expression E. Subsequent notation $E[t]$ (e.g., in the conclusion of an inference rule) stands for the expression E with the distinguished occurrence of s replaced by the term t.

We say that $E\sigma$ is an *instance* of E. A *unifier* of two expressions E_1 and E_2 is a substitution σ such that $E_1\sigma = E_2\sigma$. If two expressions E_1 and E_2 have a unifier, they also have a *most general unifier (mgu)*; we write $mgu(E_1, E_2)$. A *match* of expression E_1 to expression E_2 is a substitution σ such that $E_1\sigma = E_2$; if such a σ exists we say E_1 can be matched to E_2. Note that, if the sets of variables in E_1 and E_2 are disjoint, any match is a unifier, but not every unifier is a match, as illustrated by the following example.

Example 4. Let E_1 and E_2 be the clauses $p(x, y) \vee q(x, y)$ and $p(c, d) \vee q(c, z)$, respectively. The only possible match of $p(x, y)$ to $p(c, d)$ is $\sigma_1 = \{x \mapsto c, y \mapsto d\}$. On the other hand, the only possible match of $q(x, y)$ to $q(c, z)$ is $\sigma_2 = \{x \mapsto c, y \mapsto z\}$. As σ_1 and σ_2 are not the same, there is no match of E_1 to E_2. Note however that E_1 and E_2 can be unified, for example, using $\sigma_3 = \{x \mapsto c, y \mapsto d, z \mapsto d\}$.

2.1 Saturation-Based Theorem Proving

In this section, we recall the basics of saturation-based theorem proving and superposition reasoning and refer to the literature for further details [BG94, BG01, NR01]. We adopt the notations and the inference system of superposition from [KV13].

The key idea behind saturation-based proof search is to reduce the problem of proving validity of a first-order formula F to the problem of establishing unsatisfiability of $\neg F$ by using a sound inference system. More precisely, the prover first obtains a clausification of $\neg F$ and tries to derive the empty clause $\square$ by applying sound inferences over its clause database. The details of how the clause database is organized depends on the concrete saturation algorithm, but usually, at each step of the saturation algorithm, not all clauses are eligible to trigger inferences.

An *inference* is usually written as

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

with $n \geq 0$ and clauses $C, C_1, \dots, C_n$. The clauses $C_1, \dots, C_n$ are called the *premises* and C is the *conclusion* of the inference above. An *inference rule* is a set of (concrete) inferences and an *inference system* is a set of inference rules. An inference is *sound* if its conclusion is a logical consequence of its premises. An inference rule is sound if all its inferences are sound, and an inference system is sound if all its inference rules are sound.

2.2 Superposition Inference System

Modern first-order theorem provers implement the *superposition inference system* for first-order logic with equality. This inference system is parametrized by a *simplification ordering* over terms and a *literal selection function* over clauses. In what follows, we denote by $\succ$ a simplification ordering over terms, that is $\succ$ is a well-founded partial ordering satisfying the following three conditions for all terms s and t:

- *stability under substitutions*: if $s \succ t$, then $s\sigma \succ t\sigma$ for any substitution σ;
- *monotonicity*: if $s \succ t$, then $l[s] \succ l[t]$ for any term l;
- *subterm property*: $s \succ t$ whenever t is a proper subterm of s.

The simplification ordering $\succ$ on terms can be extended to a simplification ordering on literals and clauses, using a multiset extension of orderings. For simplicity, the extension of $\succ$ to literals and clauses will also be denoted by $\succ$. Whenever $E_1 \succ E_2$, we say that E_1 is bigger than E_2 and E_2 is smaller than E_1 w.r.t. $\succ$. We say that an equality literal $s \simeq t$ is *oriented*, if $s \succ t$ or $t \succ s$. The literal extension of $\succ$ asserts that negative literals are always bigger than their positive counterparts. Moreover, if $L_1 \succ L_2$, where L_1 and L_2 are positive, then $\neg L_1 \succ L_1 \succ \neg L_2 \succ L_2$. Finally, equality literals are set to be smaller than any literal using a predicate different than $\simeq$.

A *selection function* selects at least one literal in every non-empty clause. In what follows, selected literals in clauses will be underlined: when writing $\underline{L} \vee C$, we mean that (at least) L is selected in $L \vee C$. In what follows, we assume that selection functions are *well-behaved* w.r.t. $\succ$: either a negative literal is selected or all maximal literals w.r.t. $\succ$ are selected.

In the sequel, we fix a simplification ordering $\succ$ and a well-behaved selection function and consider the superposition inference system, denoted by Sup, parametrized by these two ingredients. The inference system Sup for first-order logic with equality consists of the inference rules of Figure 2.1, and it is both sound and refutationally complete. That is, if a set S of clauses is unsatisfiable, then the empty clause (that is, the always false formula) is derivable from S in Sup.

- Resolution and Factoring

$$\frac{\underline{L} \vee C_1 \quad \underline{\neg L'} \vee C_2}{(C_1 \vee C_2)\sigma} \qquad \frac{\underline{L} \vee \underline{L'} \vee C}{(L \vee C)\sigma}$$

where L is not an equality literal and $\sigma = mgu(L, L')$.

- Superposition

$$\frac{\underline{s \simeq t} \vee C_1 \quad \underline{L[s']} \vee C_2}{(C_1 \vee L[t] \vee C_2)\sigma}$$

$$\frac{\underline{s \simeq t} \vee C_1 \quad \underline{l[s'] \simeq l'} \vee C_2}{(C_1 \vee l[t] \simeq l' \vee C_2)\sigma} \qquad \frac{\underline{s \simeq t} \vee C_1 \quad \underline{l[s'] \not\simeq l'} \vee C_2}{(C_1 \vee l[t] \not\simeq l' \vee C_2)\sigma}$$

where s' not a variable, L is not an equality, $\sigma = mgu(s, s')$, $t\sigma \not\succ s\sigma$ and $l'\sigma \not\succ l[s']\sigma$.

- Equality Resolution and Equality Factoring

$$\frac{\underline{s \not\simeq s'} \vee C}{C\sigma} \qquad \frac{s \simeq t \vee \underline{s' \simeq t'} \vee C}{(s \simeq t \vee t \not\simeq t' \vee C)\sigma}$$

where $\sigma = mgu(s, s')$, $t\sigma \not\succ s\sigma$ and $t'\sigma \not\succ t\sigma$.

Figure 2.1: The superposition calculus Sup.

2.3 Saturation and Redundancy

The basic idea behind saturation is simple: apply all possible inferences of Sup to the clauses in the clause database until (i) no more inferences can be applied or (ii) the empty clause $\square$ has been derived. However, a naïve implementation of saturation would not be very useful in practice, because simply applying all possible inferences will quickly blow up the search space.

The design of efficient saturation algorithms exploits the powerful concept of *redundancy*: deleting so-called redundant clauses from the clause database while preserving the completeness of Sup.

Redundancy builds on the literal ordering $\succ$ introduced in the previous section.

Definition 1 (Ground Redundancy)**.** A ground clause C is *ground-redundant* in a set of ground clauses S if there are $C_1, \ldots, C_n \in S$ such that

- $C_1, \ldots, C_n \models C$, and
- $C_i \prec C$ for all $i \in \{1, \ldots, n\}$.

Definition 2 (Redundancy)**.** A clause C is *redundant* in a set of clauses S if all ground instances of C are ground-redundant in a set of ground instances of clauses in S.

It is known that redundant clauses can be removed from the clause database without affecting the completeness of saturation-based proof search. However, determining exactly whether a clause C is redundant in the current database S is undecidable in general. Thus, in practice, the focus is on identifying sufficient conditions that guarantee redundancy.

Some redundancy is already baked into Sup in the form of ordering constraints.

The inferences of Sup add new clauses to the clause database and thus increase the size of the database; such inferences are called *generating inferences*. In addition to generating inferences, state-of-the-art theorem provers employ also *simplifying inferences* that simplify the clause database or reduce its size. Such inferences are unnecessary from a theoretical perspective, but they are responsible for major speed-ups in practice.

Simplification Rules. An inference with premises $C_1, \ldots, C_n$ and conclusion C is a *simplifying inference* if one of its premises C_i is redundant in the updated clause database $S \cup \{C\}$ after the addition of the conclusion C. In what follows, we will denote deleted clauses by drawing a line through them and refer to inference rules that only contain simplifying inferences as *simplification rules*. The premise C_i that becomes redundant is called the *main premise*, whereas any other premises are called *side premises* of the simplification rule. Intuitively, a simplification rule simplifies its main premise to its conclusion by using additional knowledge from its side premises.

In saturation-based proof search, we distinguish between *forward* and *backward* simplifications. During forward simplification, a newly derived clause is simplified using previously derived clauses as side clauses. Conversely, during backward simplification a newly derived clause is used as a side clause to simplify previously derived clauses.

One important example of a simplification rule is *demodulation*, also called *rewriting by unit equalities*.

Definition 3 (Demodulation)**.** *Demodulation* is the inference rule

$$\frac{l \simeq r \quad \cancel{L[t] \vee C}}{L[r\sigma] \vee C}$$

where $l\sigma = t$, $l\sigma \succ r\sigma$ and $L[t] \vee C \succ (l \simeq r)\sigma$, for some substitution σ.

It is easy to see that demodulation is a simplification rule. Moreover, demodulation is a special case of a superposition inference of Sup where one premise of the inference is deleted. However, unlike a superposition inference, demodulation is not restricted to selected literals.

Example 5. Consider the clauses $C_1 = f(f(x)) \simeq f(x)$ and $C_2 = p(f(f(c))) \vee q(d)$. Let σ be the substitution $\sigma = \{x \mapsto c\}$. By the subterm property of $\succ$, we have $f(f(c)) \succ f(c)$. Further, as equality literals are smaller than non-equality literals,

we have $p(f(f(c))) \vee q(d) \succ f(f(c)) \simeq f(c)$. We may thus apply demodulation and simplify C_2 into the clause $C_3 = p(f(c)) \vee q(d)$:

$$\frac{f(f(x)) \simeq f(x) \qquad \cancel{p(f(f(c))) \vee q(d)}}{p(f(c)) \vee q(d)}$$ □

Deletion Rules. Even with simplification rules, it is useful to identify other redundant clauses and delete them to keep the clause database small. For this reason, in addition to simplifying and generating rules, theorem provers also use deletion rules: a *deletion rule* checks whether clauses in the clause database are redundant due to the presence of other clauses, and removes redundant clauses from the database.

Definition 4 (Subsumption)**.** We say a clause C *subsumes* another clause D if there exists a substitution σ such that $C\sigma \sqsubseteq D$, where $\sqsubseteq$ denotes multiset inclusion. *Subsumption* is the deletion rule that removes subsumed clauses from the clause database.

Example 6. Let $C = p(x) \vee q(f(x))$ and $D = p(f(c)) \vee p(g(c)) \vee q(f(c)) \vee q(f(g(c))) \vee q(y)$ be clauses in the clause database. Using the substitution $\sigma = \{x \mapsto g(c)\}$, it is easy to see that C subsumes D, and hence we may delete D from the clause database. □

Example 7. On the other hand, the clause $C = p(x) \vee p(y)$ does not subsume $D = p(f(c))$, because it is not possible to satisfy the multiset inclusion constraint. A theorem prover based on Sup may first apply factoring to derive $C' = p(z)$ from C, and subsequently delete both C and D as they are now both subsumed by C'.

This example also gives some intuition why the definition of subsumption relies on multiset inclusion rather than ordinary set inclusion: with ordinary set inclusion, C would subsume C'. □

Subsumption gives a powerful basis for other simplification rules. For example, *subsumption resolution* [KV13, SCV19], also known as *contextual literal cutting* or *self-subsuming resolution*, is the combination of subsumption with binary resolution; and *subsumption demodulation* (see Chapter 3) results from combining subsumption with demodulation.

Definition 5 (Subsumption Resolution)**.** For the purpose of this thesis, let *subsumption resolution* denote the rules

$$\frac{L \vee C \quad \cancel{\neg M \vee D}}{D} \qquad \frac{\neg L \vee C \quad \cancel{M \vee D}}{D}$$

where $L\sigma = M$ and $C\sigma \subseteq D$ for some substitution σ.

Subsumption resolution is a well-known simplification rule. A notable difference to subsumption is the use of regular set inclusion rather than multi-set inclusion in the side conditions, which is justified because the conclusion is always strictly smaller than the main premise due to the removal of a literal. A more thorough definition and treatment of subsumption resolution may be found in [CRR+24].

2.4 Propositional Logic and SAT Solving

We introduce standard propositional logic, which will be used in Chapter 4 as compilation target for subsumption constraints.

Let $\mathcal{B}$ be a countably infinite set of *Boolean variables*. We denote Boolean variables by b, possibly with indices. We use the standard Boolean connectives $\wedge$, $\vee$, $\rightarrow$, $\neg$, and write $\top$ for the Boolean constant *true* as well as $\bot$ for the Boolean constant *false*. A Boolean *literal*, denoted l, is a variable b or its negation $\neg b$.[1] A Boolean *clause* is a disjunction of literals. As before, we drop the qualifier *Boolean* when it is clear from the context.

Modern SAT solvers are based on conflict-driven clause learning (CDCL) [BS97, MS99, MLM21], with the core procedures *decide*, *unit-propagate*, and *resolve-conflict*. The solver maintains a partial assignment of truth values to the Boolean variables. Unit propagation (also called Boolean constraint propagation), that is *unit-propagate* in a SAT solver, propagates clauses w.r.t. the partial assignment. If exactly one literal l in a clause remains unassigned in the current assignment while all other literals are false, the solver sets l to true to avoid a conflict. The two-watched-literals scheme [MMZ+01] is the standard approach for efficient implementation of unit propagation.

If no propagation is possible, the solver may choose a currently unassigned variable b and set it to true or false; hence, *decide* in SAT solving. The number of variables in the current assignment that have been assigned by decision is called the *decision level*.

If all literals in a clause are false in the current assignment, the solver enters conflict resolution, via the *resolve-conflict* procedure of SAT solving. If the current decision level is 0, the conflict follows unconditionally from the input clauses and the solver returns Unsat ("unsatisfiable"). Otherwise, by analyzing how the literals in the conflicting clause have been assigned, the solver may derive and learn a conflict lemma, undo some decisions, and continue solving.

If the solver succeeds in assigning all Boolean variables without falsifying any clause, it returns Sat ("satisfiable") and the current assignment as a model of the input formula.

[1] While we also use l for first-order terms in some places, the distinction will be clear from the context.

CHAPTER 3

Subsumption Demodulation

This chapter is based on the following publication:

[GKR20] Bernhard Gleiss, Laura Kovács, and Jakob Rath. Subsumption Demodulation in First-Order Theorem Proving. In *Proceedings of IJCAR*, pages 297–315, 2020. doi:10.1007/978-3-030-51074-9_17

In this chapter we introduce a new simplification rule, called subsumption demodulation, by extending demodulation as defined in Definition 3 to a simplification rule over conditional equalities. We do so by combining demodulation with subsumption checks to find simplifying applications of rewriting by non-unit (and hence conditional) equalities.

3.1 Definition and Soundness

Definition 6 (Subsumption Demodulation)**.** *Subsumption demodulation* is the inference rule:

$$\frac{l \simeq r \vee C \qquad L[t] \vee D}{L[r\sigma] \vee D} \tag{3.1}$$

where:

1. $l\sigma = t$,
2. $C\sigma \sqsubseteq D$,
3. $l\sigma \succ r\sigma$, and
4. $L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma$.

We call the equality $l \simeq r$ in the left premise of (3.1) the *rewriting equality* of subsumption demodulation.

Intuitively, the side conditions 1 and 2 of Definition 6 ensure the soundness of the rule: it is easy to see that if $l \simeq r \vee C$ and $L[t] \vee D$ are true, then $L[r\sigma] \vee D$ also holds. We thus conclude:

Theorem 1 (Soundness)**.** Subsumption demodulation is sound.

On the other hand, side conditions 3 and 4 of Definition 6 are vital to ensure that subsumption demodulation is a simplification rule (details follow in Section 3.2).

Detecting possible applications of subsumption demodulation involves (i) selecting one equality of the side clause as rewriting equality and (ii) matching each of the remaining literals, denoted C in (3.1), to some literal in the main clause. Step (i) is similar to finding unit equalities in demodulation, whereas step (ii) reduces to showing that C subsumes parts of the main premise. Informally speaking, subsumption demodulation combines demodulation and subsumption, as discussed in Section 3.3. Note that in step (ii), matching allows any instantiation of C to $C\sigma$ via substitution σ; yet, we we do *not* unify the side and main premises of subsumption demodulation, as illustrated later in Example 10. Furthermore, we need to find a term t in the unmatched part $D \setminus C\sigma$ of the main premise, such that t can be rewritten according to the rewriting equality into $r\sigma$.

As the ordering $\succ$ is partial, the conditions of Definition 6 must be checked a posteriori, that is after subsumption demodulation has been applied with a fixed substitution. Note however that if $l \succ r$ in the rewriting equality, then $l\sigma \succ r\sigma$ for any substitution, so checking the ordering a priori helps, as illustrated in the following example.

Example 8. Let us consider the following two clauses:

$$C_1 = f(g(x)) \simeq g(x) \vee q_1(x) \vee q_2(y)$$
$$C_2 = p(f(g(c))) \vee q_1(c) \vee q_1(d) \vee q_2(f(g(d)))$$

By the subterm property of $\succ$, we conclude that $f(g(x)) \succ g(x)$. Hence, the rewriting equality, as well as any instance of it, is oriented.

Let σ be the substitution $\sigma = \{x \mapsto c, y \mapsto f(g(d))\}$. Due to the previous paragraph, we know $f(g(c)) \succ g(c)$ As equality literals are smaller than non-equality ones, we also conclude $p(f(g(c))) \succ f(g(c)) \simeq g(c)$. Thus, we have

$$p(f(g(c))) \vee q_1(c) \vee q_1(d) \vee q_2(f(g(d))) \;\succ\; f(g(c)) \simeq g(c) \vee q_1(c) \vee q_2(f(g(d)))$$

and we can apply subsumption demodulation to C_1 and C_2, deriving the clause $C_3 = p(g(c)) \vee q_1(c) \vee q_1(d) \vee q_2(f(g(d)))$.

We note that demodulation cannot derive C_3 from C_1 and C_2, as there is no unit equality. □

Example 8 highlights limitations of demodulation when compared to subsumption demodulation. We next illustrate different possible applications of subsumption demodulation using a fixed side premise and different main premises.

Example 9. Consider the clause $C_1 = f(g(x)) \simeq g(y) \vee q_1(x) \vee q_2(y)$. Only the first literal $f(g(x)) \simeq g(y)$ is a positive equality and as such eligible as rewriting equality. Note that $f(g(x))$ and $g(y)$ are incomparable w.r.t. $\succ$ due to occurrences of different variables, and hence whether $f(g(x))\sigma \succ g(y)\sigma$ depends on the chosen substitution σ.

1. Consider the clause $C_2 = p(f(g(c))) \vee q_1(c) \vee q_2(c)$ as the main premise. With the substitution $\sigma_1 = \{x \mapsto c, y \mapsto c\}$, we have $f(g(x))\sigma_1 \succ g(x)\sigma_1$ as $f(g(c)) \succ g(c)$ due to the subterm property of $\succ$, enabling a possible application of subsumption demodulation over C_1 and C_2.

2. Consider now $C_3 = p(g(f(g(c)))) \vee q_1(c) \vee q_2(f(g(c)))$ as the main premise and the substitution $\sigma_2 = \{x \mapsto c, y \mapsto f(g(c))\}$. We have $g(y)\sigma_2 \succ f(g(x))\sigma_2$, as $g(f(g(c)) \succ f(g(c))$. The instance of the rewriting equality is oriented differently in this case than in the previous one, enabling a possible application of subsumption demodulation over C_1 and C_3.

3. On the other hand, using the clause $C_4 = p(f(g(c))) \vee q_1(c) \vee q_2(z)$ as the main premise, the only substitution we can use is $\sigma_3 = \{x \mapsto c, y \mapsto z\}$. The corresponding instance of the rewriting equality is then $f(g(c)) \simeq g(z)$, which cannot be oriented in general. Hence, subsumption demodulation cannot be applied in this case, even though we can find the matching term $f(g(c))$ in C_4. □

As mentioned before, the substitution σ appearing in subsumption demodulation can only be used to instantiate the side premise, but not for unifying side and main premises, as we would not obtain a simplification rule.

Example 10. Consider the clauses:

$$C_1 = f(c) \simeq c \vee q(d)$$
$$C_2 = p(f(c)) \vee q(x)$$

As we cannot match $q(d)$ to $q(x)$ (although we could match $q(x)$ to $q(d)$), subsumption demodulation is not applicable with premises C_1 and C_2. □

3.2 Simplification using Subsumption Demodulation

Note that in the special case where C is the empty clause in (3.1), subsumption demodulation reduces to demodulation and hence it is a simplification rule. We next show that this is the case in general.

Theorem 2 (Simplification Rule)**.** Subsumption demodulation is a simplification rule and we have

$$\frac{l \simeq r \vee C \qquad \cancel{L[t] \vee D}}{L[r\sigma] \vee D}$$

where

1. $l\sigma = t$,
2. $C\sigma \sqsubseteq D$,
3. $l\sigma \succ r\sigma$, and
4. $L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma$.

Proof. Because of the second condition of the definition of subsumption demodulation, $L[t] \vee D$ is clearly a logical consequence of $L[r\sigma] \vee D$ and $l \simeq r \vee C$. Moreover, from the fourth condition, we trivially have $L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma$. It thus remains to show that $L[r\sigma] \vee D$ is smaller than $L[t] \vee D$ w.r.t. $\succ$. As $t = l\sigma \succ r\sigma$, the monotonicity property of $\succ$ asserts that $L[t] \succ L[r\sigma]$, and hence $L[t] \vee D \succ L[r\sigma] \vee D$. This concludes that $L[t] \vee D$ is redundant w.r.t. the conclusion and left-most premise of subsumption demodulation. □ □

Example 11. By revisiting Example 8, Theorem 2 asserts that clause C_2 is simplified into C_3, and subsumption demodulation deletes C_2 from the search space. □

3.2.1 Refining Redundancy

The fourth condition defining subsumption demodulation in Definition 6 is required to ensure that the main premise of subsumption demodulation becomes redundant. However, comparing clauses w.r.t. the ordering $\succ$ is computationally expensive; yet, not necessary for subsumption demodulation. Following the notation of Definition 6, let D' such that $D = C\sigma \vee D'$. By properties of multiset orderings, the condition $L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma$ is equivalent to $L[t] \vee D' \succ (l \simeq r)\sigma$, as the literals in $C\sigma$ occur on both sides of $\succ$. This means, to ensure the redundancy of the main premise of subsumption demodulation, we only need to ensure that there is a literal from $L[t] \vee D$ such that this literal is bigger that the rewriting equality.

Theorem 3 (Refining Redundancy)**.** The following conditions are equivalent:

(R1) $L[t] \vee D \succ (l \simeq r)\sigma \vee C\sigma$

(R2) $L[t] \vee D' \succ (l \simeq r)\sigma$

As mentioned in Section 3.1, application of subsumption demodulation involves checking that an ordering condition between premises holds (side condition 4 in Definition 6). Theorem 3 asserts that we only need to find a literal in $L[t] \vee D'$ that is bigger than the rewriting equality in order to ensure that the ordering condition is fulfilled. In the next section we show that by re-using and properly changing the underlying machinery of first-order provers for demodulation and subsumption, subsumption demodulation can efficiently be implemented in superposition-based proof search.

3.3 Implementation in Vampire

We implemented subsumption demodulation in the first-order theorem prover Vampire. As for any simplification rule, we implemented both the forward and backward versions of subsumption demodulation, and their use can be controlled separately. Subsumption demodulation is available in the mainline version of Vampire[1] and can be enabled by setting the options `fsd` and `bsd`, both with possible values `on` and `off`, to enable forward and backward subsumption demodulation, respectively.

As discussed in Section 3.1, subsumption demodulation uses reasoning based on a combination of demodulation and subsumption. Algorithm 3.1 details our implementation for *forward subsumption demodulation*. In a nutshell, given a clause D as main premise, (forward) subsumption demodulation in Vampire consists of the following main steps:

1. *Retrieve candidate clauses* C as side premises of subsumption demodulation (line 1 of Algorithm 3.1). To this end, we design a new clause index with imperfect filtering, by modifying the subsumption index in Vampire, as discussed later in this section.

2. *Prune candidate clauses* by checking the conditions of subsumption demodulation (lines 3–7 of Algorithm 3.1), in particular selecting a rewriting equality and matching the remaining literals of the side premise to literals of the main premise. After this, prune further by performing a posteriori checks for orienting the rewriting equality E, and checking the redundancy of the given main premise D. To do so, we revised multi-literal matching and redundancy checking in Vampire (see later).

3. *Build simplified clause* by simplifying and deleting the (main) premise D of subsumption demodulation using (forward) simplification (line 8 of Algorithm 3.1).

Our implementation of *backward subsumption demodulation* requires only a few changes to Algorithm 3.1: (i) we use the input clause as side premise C of backward subsumption demodulation and (ii) we retrieve candidate clauses D as potential main premises of subsumption demodulation. Additionally, (iii) instead of returning a single simplified clause D', we record a replacement clause for each candidate clause D where a simplification was possible.

Clause Indexing for Subsumption Demodulation. We build upon the indexing approach [SRV01] used for subsumption in Vampire: the subsumption index in Vampire stores and retrieves candidate clauses for subsumption. Each clause is indexed by exactly one of its literals. In principle, any literal of the clause can be chosen. In order to reduce the number of retrieved candidates, the best literal is chosen in the sense that the chosen literal maximizes a certain heuristic (e.g., maximal weight). Since the subsumption index is not a perfect index (i.e., it may retrieve non-subsumed clauses), additional checks on the retrieved clauses are performed.

[1] Available at https://github.com/vprover/vampire

Algorithm 3.1: Forward Subsumption Demodulation – FSD

Input : Clause D, to be used as main premise
Output : Simplified clause D' if (forward) subsumption demodulation is possible

```
// Retrieve candidate side premises
candidates := FSDIndex.Retrieve(D)
for each C ∈ candidates do
    while m = FindNextMLMatch(C, D) do
        σ' := m.GetSubstitution()
        E := m.GetRewritingEquality()
        // E is of the form l ≃ r, for some terms l, r
        if exists term t in D \ Cσ' and substitution σ ⊇ σ' such that t = lσ then
            if CheckOrderingConditions(D, E, t, σ) then
                D' := BuildSimplifiedClause(D, E, t, σ)
                return D'
            end
        end
    end
end
```

Using the subsumption index of Vampire as the clause index for forward subsumption demodulation would however omit retrieving clauses (side premises) in which the rewriting equality is chosen as key for the index, omitting this way a possible application of subsumption demodulation. Hence, we need a new clause index in which the best literal can be adjusted to be the rewriting equality. To address this issue, we added a new clause index, called the *forward subsumption demodulation index (FSD index)*, to Vampire, as follows: we index potential side premises either by their best literal (according to the heuristic), the second best literal, or both. If the best literal in a clause C is a positive equality (i.e., a candidate rewriting equality) but the second best is not, C is indexed by the second best literal, and vice versa. If both the best and second best literal are positive equalities, C is indexed by both of them. Furthermore, because the FSD index is exclusively used by forward subsumption demodulation, this index only needs to keep track of clauses that contain at least one positive equality.

In the backward case, we can in fact reuse Vampire's index for backward subsumption. Instead we need to query the index by the best literal, the second best literal, or both (as described in the previous paragraph).

Multi-literal Matching. Similarly to the subsumption index, our new subsumption demodulation index is not a perfect index, that is it performs imperfect filtering for retrieving clauses. Therefore, additional post-checks are required on the retrieved clauses. In our work, we devised a multi-literal matching approach to:

- choose the rewriting equality among the literals of the side premise C, and

- check whether the remaining literals of C can be uniformly instantiated to the literals of the main premise D of subsumption demodulation.

There are multiple ways to organize this process. A simple approach is to (i) first pick any equality of a side premise C as the rewriting equality of subsumption demodulation, and then (ii) invoke the existing multi-literal matching machinery of Vampire to match the remaining literals of C with a subset of literals of D. For the latter step (ii), the task is to find a substitution σ such that $C\sigma$ becomes a submultiset of the given clause D. If the choice of the rewriting equality in step (i) turns out to be wrong, we backtrack. In our work, we revised the existing multi-literal matching machinery of Vampire to a new multi-literal matching approach for subsumption demodulation, by using the steps (i)-(ii) and interleaving equality selection with matching.

We note that the substitution σ in step (ii) above is built in two stages: first we get a partial substitution σ' from multi-literal matching and then (possibly) extend σ' to σ by matching term instances of the rewriting equality with terms of $D \setminus C\sigma$.

Example 12. Let D be the clause $p(f(c,d)) \vee q(c)$. Assume that our (FSD) clause index retrieves the clause $C = f(x,y) \simeq y \vee q(x)$ from the search space (line 1 of Algorithm 3.1). We then invoke our multi-literal matcher (line 3 of Algorithm 3.1), which matches the literal $q(x)$ of C to the literal $q(c)$ of D and selects the equality literal $f(x,y) \simeq y$ of C as the rewriting equality for subsumption demodulation over C and D. The matcher returns the choice of rewriting equality and the partial substitution $\sigma' = \{x \mapsto c\}$. We arrive at the final substitution $\sigma = \{x \mapsto c, y \mapsto d\}$ only when we match the instance $f(x,y)\sigma'$, that is $f(c,y)$, of the left-hand side of the rewriting equality to the literal $f(c,d)$ of D. Using σ, subsumption demodulation over C and D will derive $p(d) \vee q(c)$, after ensuring that D becomes redundant (line 8 of Algorithm 3.1). □

We further note that multi-literal matching is an NP-complete problem. Our multi-literal matching problems may have more than one solution, with possibly only some (or none) of them leading to successful applications of subsumption demodulation. In our implementation, we examine all solutions retrieved by multi-literal matching. We also experimented with limiting the number of matches examined after multi-literal matching but did not observe relevant improvements. Yet, our implementation in Vampire also supports an additional option allowing the user to specify an upper bound on how many solutions of multi-literal matching should be examined.

Redundancy Checking. To ensure redundancy of the main premise D after the subsumption demodulation inference, we need to check two properties. First, the instance $E\sigma$ of the rewriting equality E must be oriented. This is a simple ordering check. Second, the main premise D must be larger than the side premise C. Thanks to Theorem 3, this latter condition is reduced to finding a literal among the unmatched part of the main premise D that is bigger than the instance $E\sigma$ of the rewriting equality E.

Example 13. In case of Example 12, the rewriting equality E is oriented and hence $E\sigma$ is also oriented. Next, the literal $p(f(c,d))$ is bigger than $E\sigma$, and hence D is redundant w.r.t. C and D'. □

3.4 Experiments

We evaluated our implementation of subsumption demodulation in Vampire on the problems of the TPTP [Sut17, Sut24] (version 7.3.0) and SMT-LIB [BFT16] (release 2019-05-06) repositories. As described in Section 3.3, our implementation of subsumption demodulation is available in the mainline version of Vampire and may be controlled with command-line options. All our experiments were carried out on the StarExec cluster [SST14].

Benchmark Setup. From the 22 686 problems in the TPTP benchmark set, Vampire can parse 18 232 problems.[2] Out of these problems, we only used those problems that involve equalities as subsumption demodulation is only applicable in the presence of (at least one) equality. As such, we used 13 924 TPTP problems in our experiments.

On the other hand, when using the SMT-LIB repository, we chose the benchmarks from categories LIA, UF, UFDT, UFDTLIA, and UFLIA, as these benchmarks involve reasoning with both theories and quantifiers and the background theories are the theories that Vampire supports. These are 22 951 SMT-LIB problems in total, of which 22 833 problems remain after removing those where equality does not occur.

Comparative Experiments with Vampire. As a first experimental study, we compared the performance of subsumption demodulation in Vampire for different values of `fsd` and `bsd`, that is by using forward (FSD) and/or backward (BSD) subsumption demodulation. To this end, we evaluated subsumption demodulation using the CASC and SMTCOMP schedules of Vampire's portfolio mode. In order to test subsumption demodulation with the portfolio mode, we added the options `fsd` and/or `bsd` to *all* strategies of Vampire. While the resulting strategy schedules could potentially be further improved, it allowed us to test FSD/BSD with a variety of strategies.

Our results are summarized in Tables 3.1-3.2. The first column of these tables lists the Vampire version and configuration, where Vampire refers to Vampire in its portfolio mode (version 4.4). Lines 2-4 of these tables use our new Vampire, that is our implementation of subsumption demodulation in Vampire. The column "Solved" reports, respectively, the total number of TPTP and SMT-LIB problems solved by the considered Vampire configurations. Column "New" lists, respectively, the number of TPTP and SMT-LIB problems solved by the version with subsumption demodulation but not by the portfolio version of Vampire. This column also indicates in parentheses how many of the solved problems were satisfiable/unsatisfiable.

[2] The other problems contain features, such as higher-order logic, that have not been implemented in Vampire yet.

Table 3.1: Comparing Vampire with and without subsumption demodulation on TPTP, using Vampire in portfolio mode.

Configuration	Total	Solved	New (SAT+UNSAT)
Vampire	13 924	9 923	–
Vampire, with FSD	13 924	9 757	20 (3+17)
Vampire, with BSD	13 924	9 797	14 (2+12)
Vampire, with FSD and BSD	13 924	9 734	30 (6+24)

Table 3.2: Comparing Vampire with and without subsumption demodulation on SMT-LIB, using Vampire in portfolio mode.

Configuration	Total	Solved	New (SAT+UNSAT)
Vampire	22 833	13 705	–
Vampire, with FSD	22 833	13 620	55 (1+54)
Vampire, with BSD	22 833	13 632	48 (0+48)
Vampire, with FSD and BSD	22 833	13 607	76 (0+76)

While in total the portfolio mode of Vampire can solve more problems, we note that this comes at no surprise as the portfolio mode of Vampire is highly tuned using the existing Vampire options. In our experiments, we were interested to see whether subsumption demodulation in Vampire can solve problems that cannot be solved by the portfolio mode of Vampire. Such a result would justify the existence of the new rule because the set of problems that Vampire can solve in principle is increased. In future work, the portfolio mode should be tuned by also taking into account subsumption demodulation, which then ideally leads to an overall increase in performance. The columns "New" of Tables 3.1-3.2 give indeed practical evidence of the impact of subsumption demodulation: there are 30 new TPTP problems and 76 SMT-LIB problems[3] that the portfolio version of Vampire cannot solve, but forward and backward subsumption demodulation in Vampire can.

New Problems Solved Only by Subsumption Demodulation. Building upon our results from Tables 3.1-3.2, we analysed how many new problems subsumption demodulation in Vampire can solve when compared to other state-of-the-art reasoners. To this end, we evaluated our work against the superposition provers E (version 2.4) and Spass (version 3.9), as well as the SMT solvers CVC4 (version 1.7) and Z3 (version 4.8.7). We note however, that when using our 30 new problems from Table 3.1, we could not compare our results against Z3 as Z3 does not natively parse TPTP. On the other hand, when using our 76 new problems from Table 3.2, we only compared against CVC4 and Z3 as E and Spass do not support the SMT-LIB syntax.

Table 3.3 summarizes our findings. First, 11 of our 30 "new" TPTP problems can only

[3]The list of these new problems is available at
https://gist.github.com/JakobR/605a7b7db0101259052e137ade54b32c

Table 3.3: Comparing Vampire with subsumption demodulation against other solvers, using the "new" TPTP and SMT-LIB problems of Tables 3.1-3.2 and running Vampire in portfolio mode.

Solver/configuration	TPTP problems	SMT-LIB problems
Baseline: Vampire, with FSD and BSD	30	76
E with `--auto-schedule`	14	-
Spass (default)	4	-
Spass (local contextual rewriting)	6	-
Spass (subterm contextual rewriting)	5	-
CVC4 (default)	7	66
Z3 (default)	-	49
Only solved by Vampire (FSD+BSD)	11	0

be solved using forward and backward subsumption demodulation in Vampire; none of the other systems were able solve these problems.

Second, while all our 76 "new" SMT-LIB problems can also be solved by CVC4 and Z3 together, we note that out of these 76 problems there are 10 problems that CVC4 cannot solve, and similarly 27 problems that Z3 cannot solve.

Comparative Experiments without AVATAR. Finally, we investigated the effect of subsumption demodulation in Vampire without AVATAR [Vor14]. We used the default mode of Vampire (that is, without using a portfolio approach) and turned off the AVATAR setting. While this configuration solves less problems than the portfolio mode of Vampire, so far Vampire is the only superposition-based theorem prover implementing AVATAR. Hence, evaluating subsumption demodulation in Vampire without AVATAR is more relevant to other reasoners. Further, as AVATAR may often split non-unit clauses into unit clauses, it may potentially simulate applications of subsumption demodulation using demodulation. Table 3.4 shows that this is indeed the case: with both `fsd` and `bsd` enabled, subsumption demodulation in Vampire can prove 190 TPTP problems and 173 SMT-LIB examples that the default Vampire without AVATAR cannot solve. Again, the column "New" denotes the number of problems solved by the respective configuration but not by the default mode of Vampire without AVATAR.

Table 3.4: Comparing Vampire in default mode and without AVATAR, with and without subsumption demodulation.

	TPTP problems		
Configuration	Total	Solved	New (SAT+UNSAT)
Vampire	13 924	6 601	–
Vampire (FSD)	13 924	6 539	152 (13+139)
Vampire (BSD)	13 924	6 471	112 (12+100)
Vampire (FSD+BSD)	13 924	6 510	190 (15+175)
	SMT-LIB problems		
Configuration	Total	Solved	New (SAT+UNSAT)
Vampire	22 833	9 608	–
Vampire (FSD)	22 833	9 597	134 (1+133)
Vampire (BSD)	22 833	9 541	87 (0+87)
Vampire (FSD+BSD)	22 833	9 581	173 (1+172)

CHAPTER 4

Subsumption via SAT Solving

This chapter is based on the following publication:

[RBK22] Jakob Rath, Armin Biere, and Laura Kovács. First-Order Subsumption via SAT Solving. In *Proceedings of FMCAD*, pages 160–169, 2022. doi: 10.34727/2022/ISBN.978-3-85448-053-2_22

Recall that a first-order clause C subsumes a clause D iff there exists a substitution σ such that $\sigma(C) \sqsubseteq D$, where $\sqsubseteq$ denotes multiset inclusion (Definition 4). In what follows, we refer by *clausal subsumption* between C and D to the case when clause C subsumes clause D. Similarly, *literal subsumption* between L and M refers to the case when a first-order literal L subsumes a literal M.

We note that deciding literal subsumption amounts to checking whether the literal M is an instance of the literal L, which can be done in almost linear time by constructing a substitution (if it exists) σ such that $\sigma(L) = M$; in this case, the value of $\sigma(x)$ is uniquely determined by L and M for each variable x occurring in L.

However, when working with arbitrary, and not necessarily unit, clauses C and D, deciding clausal subsumption between C and D is NP-complete [KN86] for the following reason: for each literal L_i of C, one of the literals M_{j_i} of D needs to be chosen in such a way that a substitution σ *simultaneously* matches each L_i with its respective M_{j_i}; that is, $\sigma(L_i) = M_{j_i}$ for all i.

In this chapter, we first introduce *substitution constraints* (Section 4.1), allowing us to then formulate clausal subsumption as a SAT problem over substitution constraints (Section 4.2). Based on this SAT-encoding of subsumption, we present an effective approach towards using subsumption in saturation in Section 4.3.

4.1 Substitution Constraints

We first introduce *substitution constraints* as a key ingredient for our SAT encoding of clausal subsumption.

Definition 7 (Substitution Constraint)**.** A *substitution constraint* Γ is a partial function from $\mathcal{V}$ to $\mathcal{T}$, denoted as

$$(x_1, \ldots, x_k) \rhd (t_1, \ldots, t_k),$$

where $k \geq 0$, $x_i \in \mathcal{V}$ are pairwise different, and $t_i \in \mathcal{T}$. The set $\mathrm{dom}(\Gamma) \coloneqq \{x_1, \ldots, x_k\}$ is called the *domain* of Γ. We further write $\Gamma(x_i) = t_i$ for $i \in \{1, \ldots, k\}$.

A substitution $\sigma \colon \mathcal{V} \to \mathcal{T}$ *satisfies the substitution constraint* Γ, written $\sigma \models \Gamma$, iff $\sigma(x_i) = t_i$ for all $i \in \{1, \ldots, k\}$.

Two substitution constraints Γ_1, Γ_2 are *compatible* if there exists a substitution σ that satisfies both Γ_1 and Γ_2, that is, if $\Gamma_1(x) = \Gamma_2(x)$ for all variables $x \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2)$.

As already discussed, literal subsumption between two literals L and M can easily be determined (as there is only one literal L that needs to be matched to another single literal M). The substitution constraint corresponding to the literal subsumption between L and M is denoted by $\Gamma(L, M)$ and is defined below.[1]

Definition 8 (Substitution Constraint for Literals)**.** Let L and M be two literals. If there exists a substitution σ such that $\sigma(L) = M$, the *substitution constraint* $\Gamma(L, M)$ *for literals* L *and* M is

$$\Gamma(L, M) \coloneqq (x_1, \ldots, x_k) \rhd (t_1, \ldots, t_k),$$

where $\mathcal{V}(L) = \{x_1, \ldots, x_k\}$ and $\sigma(x_i) = t_i$ for all $i \in \{1, \ldots, k\}$. Otherwise, L cannot be matched to M and the *substitution constraint* $\Gamma(L, M)$ *for literals* L *and* M is

$$\Gamma(L, M) \coloneqq \bot.$$

Example 14. Consider the following first-order literals:

$$L_1 = p(x_1, x_2, x_3) \qquad L_2 = p(f(x_2), x_4, x_4)$$
$$M_1 = p(f(c), d, y_1) \qquad M_2 = p(f(d), c, c)$$

We obtain the following substitution constraints:

$$\Gamma(L_1, M_1) = (x_1, x_2, x_3) \rhd (f(c), d, y_1)$$
$$\Gamma(L_1, M_2) = (x_1, x_2, x_3) \rhd (f(d), c, c)$$
$$\Gamma(L_2, M_1) = \bot$$
$$\Gamma(L_2, M_2) = (x_2, x_4) \rhd (d, c)$$

[1] At this point, we ignore the symmetry of equality literals and match them purely syntactically, i.e., the left-hand side is matched with the left-hand side and the right-hand side is matched with the right-hand side (for a fixed but arbitrary ordering of equality literals). The treatment of equality literals during encoding is explained in Remark 4.

The constraints $\Gamma(L_1, M_1)$ and $\Gamma(L_1, M_2)$ are incompatible, as these constraints map, for example, x_1 to different values. The constraints $\Gamma(L_1, M_1)$ and $\Gamma(L_2, M_2)$ are compatible, as both constraints require their only shared variable x_2 to be mapped to d.

To encode clausal subsumption, we need to combine substitution constraints using Boolean connectives, and Boolean variables. For this reason, we now define the semantics of Boolean combinations of substitution constraints.

Definition 9 (Boolean Combination of Substitution Constraints)**.** Let F be a formula using standard Boolean connectives, whose atoms are Boolean variables and substitution constraints. An interpretation $I = (\alpha, \sigma)$ for such a formula is a pair of a standard Boolean assignment $\alpha\colon \mathcal{B} \to \{\top, \bot\}$ and a substitution $\sigma\colon \mathcal{V} \to \mathcal{T}$.

For a Boolean variable b, we define $I \models b$ iff $\alpha(b) = \top$. For a substitution constraint Γ, we define $I \models \Gamma$ iff $\sigma \models \Gamma$. For formulas F with a top-level connective of $\wedge$, $\vee$, $\rightarrow$, or $\neg$, we define $I \models F$ inductively in the standard way. For Boolean constants, $I \models \top$ and $I \not\models \bot$.

Remark 1. The formula F can also be translated into an SMT formula using the theory of equality and uninterpreted functions (EUF), where substitution constraints are replaced by conjunctions of equality literals. Let T denote the set of terms t appearing on the right-hand side of some substitution constraint in F. We then introduce fresh constant symbols $\{c_t \mid t \in T\}$, and replace each substitution constraint $\Gamma = (x_1, \ldots, x_k) \rhd (t_1, \ldots, t_k)$ in F by $x_1 = c_1 \wedge \cdots \wedge x_k = c_k$. To obtain correct semantics of substitution compatibility, we also need to add

$$\bigwedge \{c_t \neq c_u \mid t, u \in T, t \neq u\}, \tag{4.1}$$

asserting that constants representing different terms in F cannot be equal.

However, for clausal subsumption in a first-order theorem prover, it is vital that the process of encoding subsumption in SAT, as well as the setting up of our SAT solver for handling this encoding are as lightweight as possible (see Section 4.4). Hence, we did not employ a standard SMT solver with the EUF-based encoding discussed above, but instead opted to directly add support for substitution constraints to our SAT solver. The advantage of our SAT-based approach is that we use less Boolean literals, and we avoid using all-different constraints for terms, such as (4.1).

4.2 SAT-Encoding of Clausal Subsumption

We now present our formalization to express clausal subsumption between clauses C and D as a SAT problem over substitution constraints. To this end, assume that clause C is $L_1 \vee L_2 \vee \cdots \vee L_n$, whereas D is $M_1 \vee M_2 \vee \cdots \vee M_m$. Recall that deciding whether C subsumes D reduces to the problem of deciding whether there exists a substitution σ such that $\sigma(C) \sqsubseteq D$, where "$\sqsubseteq$" denotes multiset inclusion (over multisets of literals).

For arbitrary literals L_i and M_j, deciding the existence of a substitution σ with $\sigma(L_i) = M_j$ can easily be done. Yet, for clausal subsumption we are left with the challenge of finding a substitution σ such that, for each L_i, we have one of the M_j such that $\sigma(L_i) = M_j$. To address this challenge, we introduce new Boolean variables b_{ij} to encode possible matchings of L_i to M_j. Intuitively, b_{ij} should be true iff we want to match L_i to M_j and enforce $\sigma(L_i) = M_j$. Additionally, we use Definition 8 to derive the substitution constraints $\Gamma(L_i, M_j)$. Based on the Boolean variables b_{ij} and substitution constraints $\Gamma(L_i, M_j)$, we formalize clausal subsumption between C and D by ensuring its three properties: (i) each literal L_i in C is matched to a literal M_j in D, (ii) the same substitution σ is used for each of these matchings, and (iii) $C\sigma \sqsubseteq D$ is multiset inclusion. Our formalization of clausal subsumption between C and D is given as follows.

(i) We first define the following clauses, capturing that each literal L_i of C must be matched to (at least one) literal M_j of D:

$$\bigwedge_{i=1}^{n} b_{i1} \vee b_{i2} \vee \cdots \vee b_{im}. \qquad \text{(Completeness)}$$

(ii) We connect the Boolean variables b_{ij} to the substitution constraints $\Gamma(L_i, M_j)$ through the following clauses:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} b_{ij} \rightarrow \Gamma(L_i, M_j). \qquad \text{(Compatibility)}$$

These clauses employ the substitution constraints $\Gamma(L_i, M_j)$ to ensure that the same substitution σ is used for matching L_i and M_j simultaneously, for all i, j.

(iii) As clausal subsumption uses *multiset* inclusion over the respective multisets of literals of C and D, we encode the requirement that each literal of D may only be matched at most once:

$$\bigwedge_{j=1}^{m} \text{AtMostOne}(b_{1j}, \ldots, b_{nj}), \qquad \text{(Multiplicity Conservation)}$$

where $\text{AtMostOne}(b_{1j}, \ldots, b_{nj})$ is true iff zero or one of $b_{1j}, \ldots, b_{nj}$ are true.

Definition 10 (SAT-Encoding of Clausal Subsumption)**.** The SAT encoding $\mathcal{S}$ of clausal subsumption between clauses C and D is defined as the conjunction of (Completeness), (Compatibility), and (Multiplicity Conservation):

$$\mathcal{S} \coloneqq \text{(Completeness)} \wedge \text{(Compatibility)} \wedge \text{(Multiplicity Conservation)}.$$

Together, the constraints (Completeness), (Compatibility), (Multiplicity Conservation) fully capture clausal subsumption, yielding the following result.

Theorem 4 (Clausal Subsumption as SAT)**.** Clausal subsumption between clauses C and D is given by the formula $\mathcal{S}$. That is, C subsumes D iff $\mathcal{S}$ is satisfiable.

Note that for deciding clausal subsumption between C and D, we only need to establish satisfiability of $\mathcal{S}$ in Theorem 4: finding one substitution σ such that $C\sigma \sqsubseteq D$ is sufficient for deciding that C subsumes D, implying that D can be deleted from the clause database during saturation. Hence, while clausal subsumption $\mathcal{S}$ captures all substitutions σ for which $C\sigma \sqsubseteq D$, for deciding whether C subsumes D we are interested to find only one satisfying instance of $\mathcal{S}$. As a result, application of clausal subsumption in saturation can be decided by solving the satisfiability of $\mathcal{S}$.

Example 15. Consider the literals defined in Example 14 and clauses $C = L_1 \vee L_2$ and $D = M_1 \vee M_2$. The encoding of clausal subsumption between C and D resulting from Theorem 4 is the conjunction of the following clauses:

$$
\begin{aligned}
&b_{11} \vee b_{12}\\
&b_{21} \vee b_{22}\\
&b_{11} \rightarrow (x_1, x_2, x_3) \rhd (f(c), d, y_1)\\
&b_{12} \rightarrow (x_1, x_2, x_3) \rhd (f(d), c, c)\\
&b_{21} \rightarrow \bot\\
&b_{22} \rightarrow (x_2, x_4) \rhd (d, c)\\
&\neg b_{11} \vee \neg b_{21}\\
&\neg b_{12} \vee \neg b_{22}
\end{aligned}
$$

This set of clauses is satisfiable, as witnessed by the model that assigns b_{11} and b_{22} to true, b_{12} and b_{21} to false, and $\sigma(x_1) = f(c)$, $\sigma(x_2) = d$, $\sigma(x_3) = y_1$, $\sigma(x_4) = c$. We conclude that the first-order clause C subsumes D.

Remark 2 (Subsumption Resolution)**.** Our encoding of clausal subsumption can be adjusted to also decide the application of other simplification rules in saturation, when these rules implement variants of subsumption. To this end, we have extended the SAT encoding $\mathcal{S}$ of clausal subsumption to the inference rule *subsumption resolution* (Definition 5). In addition to clausal subsumption, subsumption resolution also uses instances of binary resolution. Hence, for finding substitutions σ such that subsumption resolution between clauses C and D can be applied (and D deleted from the clause database), we extended the clauses $\mathcal{S}$ with additional constraints capturing application of resolution, while also adjusting the encoding of $\mathcal{S}$ to set inclusion between literals of C and D (instead of multiset inclusion from subsumption). The details have been worked out in follow-up publications [CKRR23, CRR+24].

Remark 3 (At-Most-One Constraints)**.** We conclude this section by noting that a correct but naïve solution to encode AtMostOne$(b_{1j}, \ldots, b_{nj})$ in (Multiplicity Conservation) would

be the following:

$$\bigwedge_{1 \leq i_1 < i_2 \leq n} \neg b_{i_1 j} \vee \neg b_{i_2 j}. \tag{4.2}$$

More efficient encodings using at-most-one constraints (see, e.g., [FG10]) can be used instead of (4.2). In our work however, we opted to add direct support for at-most-one constraints when reasoning about (Multiplicity Conservation) (see Section 4.3).

Remark 4 (Equality Literals)**.** Special care needs to be taken for symmetric predicates such as equality, since pairs of such literals can be matched in two different ways. As an example, consider the equality literals $L \coloneqq x \simeq f(y)$ and $M \coloneqq f(c) \simeq f(d)$. Both $\{x \mapsto f(c), y \mapsto d\}$ and $\{x \mapsto f(d), y \mapsto c\}$ are possible matches of L to M, and both have to be considered for subsumption.

Assume now that L_i and M_j are equality literals that can be matched both ways. To handle this situation, we create *two* separate boolean variables b_{ij} and b'_{ij}, with the clauses $b_{ij} \to \Gamma(L_i, M_j)$ and $b'_{ij} \to \Gamma(L_i, M'_j)$, where M'_j is the symmetric variant of M_j (i.e., M'_j is obtained by swapping the left-hand side and right-hand side of M_j).

Note that $\Gamma(L_i, M_j) = \Gamma(L_i, M'_j)$ implies $M_j = M'_j$, which means M_j is trivial and would have been removed previously by other simplification rules (or alternatively, skip trivial literals when performing the subsumption test). Because of this, we can assume $\Gamma(L_i, M_j)$ and $\Gamma(L_i, M'_j)$ are incompatible, which means b_{ij} and b'_{ij} cannot both be true. Thus, no substantial changes are needed to adapt the SAT encoding to support equality predicates: in (Completeness), we use $b_{ij} \vee b'_{ij}$ instead of the single b_{ij}, and (Multiplicity Conservation) expands from $\mathrm{AtMostOne}(\dots, b_{ij}, \dots)$ to $\mathrm{AtMostOne}(\dots, b_{ij}, b'_{ij}, \dots)$.

4.3 Effective Subsumption via Lightweight SAT Solving

In Section 4.2 we showed that the application of subsumption, as an inference rule in saturation, can be reduced to the satisfiability problem of the formula $\mathcal{S}$ using substitution constraints (Theorem 4). In this section we describe our approach for solving $\mathcal{S}$.

A straightforward approach towards handling $\mathcal{S}$ could be obtained by simply translating $\mathcal{S}$ into purely propositional clauses. However, such a translation would either require additional propositional variables to encode at-most-one constraints or would come with a quadratic number of propositional clauses [FG10]; similarly for substitution constraints.

Due to the particular distribution of subsumption instances (see Section 4.4), the encoding must be lightweight to be practically feasible. To overcome the increase in propositional variables/clauses to be used for deciding clausal subsumption in an efficient manner, we support substitution constraints as in (Compatibility) and at-most-one constraints as in (Multiplicity Conservation) directly in SAT solving, and introduce a lightweight SAT solving approach tailored to subsumption properties. In particular, we adjust unit propagation and conflict resolution in CDCL-based SAT solving for handling propositional

formulas with substitution constraints. This way, we integrate our lightweight SAT solving methodology directly into the saturation process of first-order proving (Section 4.4), instead of interfacing first-order proving with an existing off-the-shelf SAT solver. Such a direct integration allows us to efficiently identify and apply subsumption during proof search (see Section 4.5).

Using Substitution Constraints in SAT Solving. For handling substitution constraints in clausal subsumption, we attach a substitution constraint $\Gamma(L_i, M_j)$ to each freshly introduced Boolean variable b_{ij} in (Completeness), which is equivalent to adding the constraint $b_{ij} \to \Gamma(L_i, M_j)$ of (Compatibility).

Unit Propagation with Substitution Constraints. Consider now the clauses $b_{ij} \to \Gamma(L_i, M_j)$ using substitution constraints, with $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$, from clausal subsumption $\mathcal{S}$. Semantically, these constraints are equivalent to the following set of binary clauses:

$$\Big\{\neg b_{ij} \vee \neg b_{i'j'} \;\Big|\; i, i' \in \{1 \ldots n\}, j, j' \in \{1 \ldots m\}, (i,j) \neq (i',j'), \exists x \in \mathrm{dom}(\Gamma(L_i, M_j)) \cap \mathrm{dom}(\Gamma(L_{i'}, M_{j'})) \text{ s.t. } \Gamma(L_i, M_j)(x) \neq \Gamma(L_{i'}, M_{j'})(x)\Big\}, \tag{4.3}$$

which intuitively encodes that no two incompatible substitution constraints may be true at the same time.

In our work, instead of creating the binary clauses of (4.3) explicitly, we introduce support for substitution constraints as an *additional propagator* in SAT solving: whenever a Boolean variable b_{ij} is assigned to true, our SAT solver processes the associated bindings for the first-order variables from $\mathrm{dom}(\Gamma(L_i, M_j))$, and propagates all Boolean variables $b_{i'j'}$ to false that are associated with conflicting bindings for variables $\mathrm{dom}(\Gamma(L_i, M_j)) \cap \mathrm{dom}(\Gamma(L_{i'}, M_{j'}))$; in other words, all $b_{i'j'}$ whose associated substitution constraints are incompatible with $\Gamma(L_i, M_j)$. This propagation is done exhaustively as soon as b_{ij} is assigned to true and before standard unit propagation is performed. Thus we ensure that no conflict can occur at this point: if there were a conflict, that would mean a $b_{i'j'}$ with conflicting bindings has already been assigned to true; in this case however, we would have already propagated b_{ij} to false when assigning $b_{i'j'}$. An exception in handling conflicts occurs with the initial propagation before starting the CDCL loop of SAT solving; in this case, we may get a conflict if two unit clauses with conflicting substitution constraints have been added, however, in that case the SAT solver is at decision level 0 and can terminate with reporting unsatisfiability (Unsat) of $\mathcal{S}$.

Conflict Resolution with Substitution Constraints. During conflict resolution in our SAT engine, we proceed as if the binary clauses (4.3) were part of the clause database, i.e., as if the binary clause $\neg b_{ij} \vee \neg b_{i'j'}$ were the reason for propagating $b_{i'j'}$. Therefore we only need to store the literal b_{ij} as the reason for unit propagation. Substitution

constraints during conflict resolution thus do not need specialized treatment in our SAT solving approach.

At-Most-One Constraints. During unit propagation and conflict resolution, our at-most-one constraints of (Multiplicity Conservation) are treated as if we had the corresponding binary clauses from (4.2), saving the overhead from creating additional clauses and variables.

Remark 5. While we presented our approach in the context of solving $\mathcal{S}$, our SAT solving approach naturally supports arbitrary Boolean clauses and at-most-one constraints, as well as substitution constraints in the form $b \to \Gamma$ (where b is a Boolean variable and Γ a substitution constraint).

4.4 SAT-Based Subsumption in Saturation-Based Theorem Proving

We implemented our lightweight SAT-based approach of Section 4.3 as a new extension to the theorem prover Vampire. While Vampire already had implemented highly optimized algorithms for checking subsumption previously, these algorithms are built on a standard, backtracking-based search procedure: using a static variable ordering and limited amount of unit propagation, without learning from conflicts. Hence, the full power of SAT-based reasoning with unit propagation and conflict resolution has not yet been supported for subsumption. We have overcome this limitation by integrating our SAT-based approach for clausal subsumption directly in Vampire. The SAT-based implementation of subsumption (and subsumption resolution [CKRR23, CRR+24]) is available in the mainline version of Vampire[2] since commit `ce340a081a5badfd8b04ba9fc64e38a6b5c192c3` and is now the default approach for subsumption and subsumption resolution[3].

Implementing Subsumption. When establishing satisfiability of $\mathcal{S}$, we can observe different types of subsumption instances:

(i) easy subsumption instances, where not much SAT-based search is required, i.e., very few or even no decisions/conflicts occur during the solving process. For such instances the overhead of setting up the encoding $\mathcal{S}$ largely determines the total running time of our SAT solver.

(ii) hard subsumption instances, whose application is determined by a significant number of conflict resolution steps in SAT solving.

(iii) other instances may fall in between these extremes, but in the following discussion we will focus on types (i) and (ii) to illustrate the design decisions of our implementation.

[2] Available at https://github.com/vprover/vampire

[3] The SAT-based implementation of subsumption has fully replaced the old backtracking-based implementation with the linked commit. The last commit using the old backtracking-based subsumption is `c54b8ef3167b9e8372c04a2c8e504182a86791fd`.

We recall that the overall goal of our work is to improve subsumption checking in first-order theorem proving. For this, we complemented Vampire with a SAT-based approach to decide application of subsumption. The majority of the subsumption instances encountered during a typical first-order proving attempt are of type (i), with instances of type (ii) appearing occasionally, depending on the input formula. Still, the total running time is often dominated by type (ii) instances, and these are the target of our SAT-based approach. We must however be careful to not become slower on type (i) instances, thus motivating our choice of a lightweight, dedicated SAT-solver embedded into Vampire.

In many of the trivial instances of $\mathcal{S}$, the unsatisfiabiliy of these instances can be discovered already while setting up the encoding of $\mathcal{S}$ (i.e., whenever an empty clause would be added). To save time on these instances, in our implementation we defer the construction of watch lists and other data structures until entering the solving loop of our SAT engine (which might not be entered at all).

We note that the number of subsumption instances, especially easy ones of type (i), during first-order proving can become quite large, often in the order of millions of instances in a 60 s run of a theorem prover. Allocating and deallocating a new SAT solver instance for each SAT-based subsumption query can thus become expensive (see Section 4.5); therefore, in our implementation we keep the same solver instance around, and re-use it for different queries. In particular, we keep the memory for data structures (such as clause storage, watch lists, trail, and others), instead of reallocating it for each query.

Unit Propagation. To achieve efficient unit propagation, our SAT solver for clausal subsumption watches two literals of each clause [MMZ+01], as is the state of the art in modern SAT solving. However, for at-most-one constraints the situation is different. Consider the constraint AtMostOne$(l_1, \ldots, l_k)$ for some $k \geq 3$ (note that for $k \leq 2$ we either drop the constraint or add a binary clause instead). As soon as any l_i is assigned true, all l_j with $j \neq i$ must be false to avoid violating the constraint, and are propagated thus. Hence, the solver watches *all* literals of at-most-one constraints.

4.5 Experiments

We evaluated our new SAT-based implementation for clausal subsumption in Vampire (see Section 4.4). In our experiments, we were interested (i) to measure the performance improvements we gain through our approach, as well as (ii) to assess the advantage of re-using our SAT solver objects, and thus having our SAT solver directly integrated the first-order proving process of Vampire.

Benchmarks. The basis for our benchmarks is formed by the TPTP library [Sut17, Sut24] (version 7.5.0), which is a standard benchmark library in the theorem proving community. The TPTP library contains altogether 24 098 problems in various languages, out of which 16 312 problems have been included in our evaluation of SAT-based subsumption in Vampire. The remaining TPTP problems that we did not use for our experiments

either use features that Vampire currently does not support (e.g., higher-order logic with theories), or did not involve subsumption checks.

Experimental Setup. All our experiments were carried out on a cluster at TU Wien, where the compute nodes contain two AMD Epyc 7502 processors, each of which has 32 CPU cores running at 2.5 GHz. Each compute node is equipped with 1008 GiB of physical memory that is split into eight memory nodes of 126 GiB each, with eight logical CPUs assigned to each node. We used the tool `runexec` from the benchmarking framework BenchExec [BLW17] to assign each benchmark process to a different CPU core and its corresponding memory node, while aiming to balance the load evenly across memory nodes. Further, we used GNU Parallel [Tan24] to schedule 32 benchmark processes in parallel.

Experimental Results on Measuring Speed Improvements. We emphasize that using a SAT-based approach for deciding clausal subsumption will, in theory, not prove problems that were not provable before. If a problem is provable while using saturation with redundancy, and hence with subsumption, then it is also provable using saturation without redundancy, and vice versa. However, in practice, saturation with redundancy (hence with subsumption) will improve the prover's performance in finding a proof. As such, the aim of our work is to speed up the application of subsumption in saturation. For this reason, we set up our first experiment to measure the cost of subsumption checks in isolation. A similar evaluation has previously been done for indexing techniques in first-order provers [NHRV01].

In preparation for this experiment, we ran Vampire, using the original backtracking-based subsumption implementation, with a timeout of 60 seconds on each TPTP problem while logging each subsumption check into a file. Each of these files contains a sequence of subsumption checks, which we call the *subsumption log* for a problem. This preparatory step yielded a large number of benchmarks that are representative for the checks appearing during actual proof search. These benchmarks occupy 1.75 TiB of disk space in compressed form, and contain approximately 114 billion subsumption checks in total. About 0.5 % of these subsumption checks are satisfiable (561 million), while the rest are unsatisfiable.

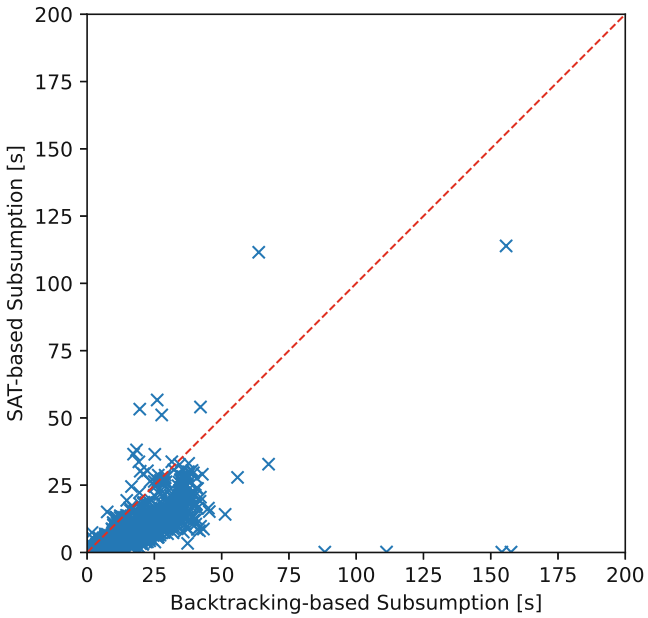
In addition to generating these benchmarks, we have profiled the portion of time spent by subsumption in Vampire. Over the TPTP problems used for our experiments and a time limit of 60 seconds, it ranges from 0 % (no subsumption checks) to more than 99 % (hard subsumption check), with a mean of 46 % and the median at 53 %.

Next, we executed the checks listed in each subsumption log and measured the total running times, once for the already existing subsumption algorithm of Vampire using backtracking, and once for our SAT-based subsumption approach in Vampire. The subsumption checks are benchmarked in a similar way as they would appear during a regular prover run, i.e., with the same caching of intermediate results. For increased reliability, we performed each measurement five times, and then took the arithmetic mean of the measured running times.

Table 4.1: Running time of subsumption checks

Subsumption Log for Problem	Backtracking-based Subsumption Setup	Solve	Total	SAT-based Subsumption Setup	Solve	Total	Δ_{abs}	Δ_{rel}
GRP134-1.005	42.87 s	2.21 s	45.08 s	13.87 s	2.61 s	16.48 s	28.60 s	2.74 x
GRP396+1	67.05 s	88.65 s	155.70 s	15.90 s	98.01 s	113.91 s	41.79 s	1.37 x
HAL007+1	33.25 s	30.54 s	63.79 s	17.05 s	94.51 s	111.56 s	-47.78 s	0.57 x
HWV056+1	26.72 s	1.01 s	27.73 s	48.73 s	2.37 s	51.10 s	-23.37 s	0.54 x
HWV058-1	17.32 s	1.05 s	18.37 s	37.57 s	0.53 s	38.10 s	-19.73 s	0.48 x
HWV059-1	24.21 s	0.95 s	25.16 s	35.79 s	0.68 s	36.48 s	-11.31 s	0.69 x
HWV060+1	16.61 s	0.66 s	17.26 s	35.82 s	0.73 s	36.55 s	-19.28 s	0.47 x
HWV086+1	17.76 s	1.80 s	19.57 s	50.12 s	3.15 s	53.27 s	-33.71 s	0.37 x
LCL662+1.020	43.78 s	1.64 s	45.42 s	14.33 s	0.86 s	15.19 s	30.23 s	2.99 x
MGT038-1	13.15 s	12.88 s	26.04 s	15.35 s	41.33 s	56.67 s	-30.64 s	0.46 x
MGT066+1	3.45 s	63.99 s	67.44 s	1.95 s	30.87 s	32.82 s	34.63 s	2.06 x
NLP023+1	0.08 s	154.05 s	154.13 s	0.04 s	0.10 s	0.14 s	153.99 s	1082.84 x
NLP023-1	0.09 s	157.46 s	157.55 s	0.05 s	0.10 s	0.14 s	157.40 s	1087.59 x
NLP024+1	0.08 s	88.26 s	88.34 s	0.04 s	0.09 s	0.14 s	88.20 s	642.68 x
NLP024-1	0.09 s	111.20 s	111.28 s	0.05 s	0.10 s	0.15 s	111.13 s	748.52 x
PUZ073+1	24.69 s	26.60 s	51.29 s	14.02 s	0.14 s	14.17 s	37.12 s	3.62 x
SYN307-1	2.09 s	53.81 s	55.90 s	1.17 s	26.73 s	27.90 s	28.01 s	2.00 x
TOP003-2	41.71 s	0.43 s	42.13 s	48.92 s	5.13 s	54.05 s	-11.92 s	0.78 x
... (+16 294)	...	...	...	...	...	...	...	...
Total	16.31 h	2.39 h	18.70 h	7.21 h	1.23 h	8.44 h	10.27 h	**2.22** x
Total (no reuse)	-	-	-	8.08 h	2.05 h	10.12 h	-	-
Total (VMTF)	-	-	-	7.62 h	1.40 h	9.02 h	-	-

Figure 4.1: Total running time (in seconds) of backtracking-based vs. SAT-based subsumption, with detailed information about outliers in Table 4.1. For marks below the dashed line, our SAT-based approach was faster.



The results of these experiments are given in Figure 4.1 and Table 4.1. Each mark in Figure 4.1 represents one subsumption log from a TPTP problem, and compares the total running times of executing all subsumption checks contained in the log with the old backtracking-based algorithm vs. the new SAT-based algorithm. The dashed line indicates equal runtime, hence, our SAT-based approach was faster for marks below the line. In Table 4.1, we give the cumulative times needed to set up the subsumption checks, to solve them, and the total time. Both the backtracking-based and our SAT-based subsumption algorithm can naturally be split up into a setup stage and a separate solving stage. The setup stage transforms the two input clauses into constraints while the solving stage searches for a solution to these constraints. Additionally the table gives detailed data for selected outliers (problems not in the bottom-left of Figure 4.1).

As shown in Figure 4.1 and Table 4.1, our SAT-based algorithm for clausal subsumption gives a clear overall improvement of the running time of subsumption checks in Vampire by a factor of 2.

Note that for some problems, the running time for the backtracking-based subsumption is higher than the original timeout of 60 s that has been used when collecting subsumption logs. The cause of this apparent discrepancy is that Vampire was working on a hard subsumption instance when hitting the timeout, with the subsequent measurements in Table 4.1 showing the true cost. Problems such as `NLP023+1` are getting stuck in the backtracking-based subsumption algorithm, while our SAT-based approach allows proof

search to continue much further within the same time limit.

We also evaluated the impact of our custom variable selection heuristic (see last paragraph of Chapter 5) compared to the variable-move-to-front (VMTF) heuristic of SAT solvers [BF15], as VMTF is conjectured to perform well for SAT problems that are unsatisfiable, being part of the "unstable phase" described in [Bie19]. Given that almost all subsumption instances are unsatisfiable, we were interested to see how our custom variable selection heuristic performs compared to a VMTF heuristic. Our results in this respect are listed in the last line of Table 4.1. While our custom heuristic shows slightly better solving times than VMTF, the difference is rather small.

Experimental Results on the Advantage of Re-Using SAT Solver Objects. We also assessed the importance of re-using the SAT solver object instead of re-allocating the solver for every subsumption query. The results for not re-using SAT solver objects are given in the second-to-last line of Table 4.1, confirming the significance of having SAT-based subsumption directly integrated in Vampire.

CHAPTER 5

Related Work

Subsumption Demodulation. While several approaches generalize demodulation in superposition-based theorem proving, we argue that subsumption demodulation improves existing methods either in terms of applicability and/or efficiency. The AVATAR architecture of first-order provers [Vor14] splits general clauses into components with disjoint sets of variables, potentially enabling demodulation inferences whenever some of these components become unit equalities. Example 1 demonstrates that subsumption demodulation applies in situations where AVATAR does not: in each clause of (1.4), all literals share the variable i and hence none of the clauses from (1.4) can be split using AVATAR. That is, AVATAR would not generate unit equalities from (1.4), and therefore cannot apply demodulation over (1.4) to derive (1.5).

The local rewriting approach of [Wei01] requires rewriting equality literals to be maximal in clauses w.r.t. the clause ordering. However, following [KV13], for efficiency reasons we consider equality literals to be "smaller" than non-equality literals. In particular, the equality literals of clauses (1.4) are "smaller" than the non-equality literals, preventing thus the application of local rewriting in Example 1.

To the extent of our knowledge, the ordering restrictions on non-unit rewriting [Wei01] do not ensure redundancy, and thus the rule is not a simplification inference rule. Subsumption demodulation includes all necessary conditions and we prove it to be a simplification rule. Furthermore, we show how the ordering restrictions can be simplified which enables an efficient implementation, and then explain how such an implementation can be realized.

We further note that the contextual rewriting rule of [BG94] is more general than our rule of subsumption demodulation, and has been first implemented in the Saturate system [NN93]. Yet, efficiently automating contextual rewriting is extremely challenging, while subsumption demodulation requires no radical changes in the existing machinery of superposition provers (see Section 3.3).

To the best of our knowledge, except Spass [WDF+09] and Saturate, no other state-of-the-art superposition provers implement variants of conditional rewriting. Subterm contextual rewriting [WW08] is a refined notion of contextual rewriting and is implemented in Spass. A major difference of subterm contextual rewriting when compared to subsumption demodulation is that in subsumption demodulation the discovery of the substitution is driven by the side conditions whereas in subterm contextual rewriting the side conditions are evaluated by checking the validity of certain implications by means of a reduction calculus. This reduction calculus recursively applies another restriction of contextual rewriting called recursive contextual ground rewriting, among other standard reduction rules. While subterm contextual rewriting is more general, we believe that the benefit of subsumption demodulation comes with its relatively easy and efficient integration within existing superposition reasoners, as evidenced also in Section 4.5.

Local contextual rewriting [HPWW13] is another refinement of contextual rewriting implemented in Spass. In our experiments it performed similarly to subterm contextual rewriting.

Finally, we note that SMT-based reasoners also implement various methods to efficiently handle conditional equalities [RWB+17, BGMR15]. Yet, the setting is very different as they rely on the DPLL(T) framework [GHN+04] rather than implementing superposition.

SAT-Based Subsumption in Saturation. Subsumption is one of the most important simplification rules in first-order theorem proving. While efficient literal- and clause-indexing techniques have been proposed [Tam98, Sch13], optimizing the matching step among multisets of literals, and hence clauses, has so far not been addressed. In our work, we show that SAT solving methods can provide efficient solutions in this respect, further improving first-order theorem proving.

A related approach that integrates multi-literal matching into indexing is given in [SRV01], using code trees. Code trees organize potentially subsuming clauses into a trie-like data structure with the aim of sharing some matching effort for similar clauses. However, the underlying matching algorithm uses a fixed branching order and does not learn from conflicts, and will thus run into the same issues on hard subsumption instances as the standard backtracking-based matching.

The specialized subsumption algorithm DC [GL85b] is based on the idea of separating the clause C into variable-disjoint components and testing subsumption for each component separately. However, the notion of subsumption considered in that work is defined using subset inclusion, rather than multiset inclusion. For subsumption based on multiset inclusion, the subsumption test for one variable-disjoint component is no longer independent of the other components.

An improved version of that algorithm, called IDC [GL85a], tests on each recursion level whether each literal of C by itself subsumes D under the current partial substitution, which is a necessary condition for subsumption. The backtracking-based subsumption

algorithm of Vampire uses this optimization as well, and our SAT-based approach also implements it as propagation over substitution constraints.

SAT- and SMT-based techniques have previously been applied to the setting of first-order saturation-based proof search, e.g., in form of the Avatar architecture [Vor14]. These techniques are however independent from our work, as they apply the SAT- or SMT-solver over an abstraction of the input problem, while in our work we use a SAT-solver to speed up certain inferences.

Some solvers, such as the pseudo-Boolean solver MiniCard [LM12] and the ASP solver Clasp [GKKS09], support cardinality constraints natively, in a similar way to our handling of AtMostOne constraints. Our encoding, however, only requires AtMostOne constraints instead of arbitrary cardinality constraints, thus simplifying the implementation.

We finally note that clausal subsumption can also be seen as a constraint satisfaction problem (CSP). In this view, the Boolean variables b_{ij} in our subsumption encoding $\mathcal{S}$ represent the different choices of a non-Boolean CSP variable, corresponding to the so-called *direct encoding* of a CSP variable [Wal00]. A well-known heuristic in CSP solving is the *minimum remaining values* heuristic: always assign the CSP variable that has the fewest possible choices remaining. We adapted this heuristic to our embedded SAT solver and use it to solve subsumption instances (see Section 4.4).

Part II

PolySAT – Word-Level Reasoning for Bit-Vectors

CHAPTER 6

Background

Part II of the thesis, i.e., Chapters 6–12, are based on the following publication:

[REK+24] Jakob Rath, Clemens Eisenhofer, Daniela Kaufmann, Nikolaj Bjørner, and Laura Kovács. PolySAT: Word-level Bit-vector Reasoning in Z3, 2024. Accepted for VSTTE 2024. arXiv:2406.04696

PolySAT integrates into the conflict-driven clause learning modulo theories (CDCL(T)) framework [BS97, MS99, MLM21], which is one of the main approaches to state-of-the-art SMT solving. The core of such an SMT solver uses a SAT solver to work on the boolean skeleton of the input formula. Furthermore, the core communicates with theory solvers, who answer conjunctive queries within the respective theory they can handle. In the remainder of the thesis, we introduce PolySAT, which is such a theory solver for the theory of bit-vectors.

6.1 Bit-Vector Language

For a given number of bits $w > 0$, we consider bit-vectors of size w as elements of the ring $\mathbb{Z}/2^w\mathbb{Z}$ (algebraic representation), or equivalently as strings of length w over $\{0, 1\}$ (binary representation). Throughout the remainder of the thesis, we write w for the size of related bit-vectors, when it is clear from the context. In other cases, we denote the size of a bit-vector x by $|x|$ explicitly.

Definition 11 (Unsigned/signed Interpretation)**.** For a bit-vector x, we write $\langle x \rangle_{\mathsf{u}}$ for the *unsigned interpretation* of x in $\mathbb{Z}$, i.e., we choose the representatives $\langle x \rangle_{\mathsf{u}} \in \{0, 1, \ldots, 2^w - 1\}$ for an element $x \in \mathbb{Z}/2^w\mathbb{Z}$. Similarly, let $\langle x \rangle_{\mathsf{s}}$ denote the *signed interpretation* in $\mathbb{Z}$, i.e., choose the representatives among $\{-2^{w-1}, \ldots, -1, 0, 1, \ldots, 2^{w-1} - 1\}$.

$\lvert x \rvert$	size (number of bits)
$\langle x \rangle_{\mathsf{u}}$	unsigned interpretation
$\langle x \rangle_{\mathsf{s}}$	signed interpretation
$x + y$, $x - y$	addition, subtraction
xy or $x \cdot y$	multiplication
$x \leq_{\mathsf{u}} y$, $x <_{\mathsf{u}} y$	unsigned inequality
$x \leq_{\mathsf{s}} y$, $x <_{\mathsf{s}} y$	signed inequality
$x[i]$	i-th bit of bit-vector x, $w > i \geq 0$
$x[i{:}j]$	extract sub-slice, $w > i \geq j \geq 0$
$x \mathbin{+\!\!+} y$	concatenation
$x \ll y$	left-shift
$x \gg y$	logical right-shift
$x \gg_{\mathsf{a}} y$	arithmetic right-shift
$x \mathbin{/} y$	unsigned division
$x \mathbin{\%} y$	unsigned remainder

Figure 6.1: Summary of notation

In the following text, we will default to the unsigned interpretation, and translate signed constraints into their unsigned counterparts where possible. In particular, note that to intuitively understand many constraints and lemmas it is often helpful to mentally replace negative bit-vector constants (such as -1) by their equivalent unsigned values (such as $2^w - 1$).

We write $x[i]$ for the i-th bit of the bit-vector x, where $x[0]$ denotes the least significant bit of x. Let $x \mathbin{+\!\!+} y$ denote the concatenation of x and y. We write $x[i{:}j]$ with $w > i \geq j \geq 0$ for the *sub-slice* ranging from bit i to bit j inclusively, i.e., $x[i{:}j] = x[i] \mathbin{+\!\!+} x[i-1] \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} x[j]$. We call the sub-slices $x[i{:}0]$ of x the *prefixes*[1] of x.

In general, for the language and semantics of bit-vector operations and predicates, we follow the definitions in the theory FixedSizeBitVectors and the logic `QF_BV` of the SMT-LIB library [BFT16]. The notations used in this thesis are summarized in Figure 6.1.

The basic building blocks of PolySAT constraints are *polynomials*, i.e., multiplications and additions of bit-vector variables and constants. We emphasize bit-vector multiplication by writing $\cdot$ explicitly.

We write $x \simeq y$ for equality constraints, $x \leq_{\mathsf{u}} y$ for unsigned inequality of bit-vectors, and $x \leq_{\mathsf{s}} y$ for signed inequality of bit-vectors. By slight abuse of notation, we will sometimes use constraints also as statements. Note that $x \leq_{\mathsf{u}} y$ iff $\langle x \rangle_{\mathsf{u}} \leq \langle y \rangle_{\mathsf{u}}$, where $\leq$ is the standard comparison over the integers. Similarly, $x \leq_{\mathsf{s}} y$ iff $\langle x \rangle_{\mathsf{s}} \leq \langle y \rangle_{\mathsf{s}}$.

[1] We choose the term *prefix* for consistency with other types of sequences, i.e., to denote a contiguous subsequence starting at index 0. While bit-vectors are typically written in "reverse" order (with the most significant bit at the front), the logical starting point of the sequence of bits is still the least significant bit at index 0.

6.2 Useful Bit-Vector Lemmas

We next list several useful lemmas about bit-vector inequalities that have been useful throughout this work. Some of these are inspired by related lemmas over the integers, however, the bit-vector versions usually require additional side conditions. As a simple sanity test of these lemmas, it is advisable to first check them with existing solvers based on bit-blasting over low bit-widths. Only after this check passes, start working on a general proof.

Lemma 1 (Inequality Equivalences)**.** The following inequalities are equivalent, for arbitrary bit-vectors p and q:

$$p \leq_u q \tag{6.1}$$
$$p \leq_u p - q - 1 \tag{6.2}$$
$$-q - 1 \leq_u p - q - 1 \tag{6.3}$$
$$-q - 1 \leq_u -p - 1 \tag{6.4}$$
$$q - p \leq_u -p - 1 \tag{6.5}$$
$$q - p \leq_u q \tag{6.6}$$

Note: for strict inequalities, use $p <_u q \iff \neg(q \leq_u p)$, i.e., the sides are swapped.

Proof. We only need to prove the two implications (6.1) $\implies$ (6.2) and (6.1) $\implies$ (6.6). It is easy to check that by alternating these rewrite rules we get the circular implication chain (6.1) $\implies$ (6.2) $\implies$ (6.3) $\implies$ (6.4) $\implies$ (6.5) $\implies$ (6.6) $\implies$ (6.1).

Let bit-vectors p, q such that $p \leq_u q$ holds, i.e., $\langle p \rangle_u \leq \langle q \rangle_u$.

We first prove (6.2). By definition, we have $0 \leq \langle p \rangle_u \leq 2^w - 1$ and $0 \leq \langle q \rangle_u \leq 2^w - 1$, from which we get $\langle p \rangle_u - \langle q \rangle_u - 1 \geq -2^w$. Since $\langle p \rangle_u \leq \langle q \rangle_u$, we also have $\langle p \rangle_u - \langle q \rangle_u - 1 < 0$. Thus

$$\begin{aligned}
\langle p - q - 1 \rangle_u &= (\langle p \rangle_u - \langle q \rangle_u - 1) \bmod 2^w && \text{by semantics of bit-vector operations} \\
&= \langle p \rangle_u - \langle q \rangle_u - 1 + 2^w && \text{by the above} \\
&\geq \langle p \rangle_u && \text{by } \langle q \rangle_u \leq 2^w - 1,
\end{aligned}$$

which implies $p \leq_u p - q - 1$.

Next, we prove (6.6). Since $\langle p \rangle_u \leq \langle q \rangle_u$, we have $0 \leq \langle q \rangle_u - \langle p \rangle_u \leq \langle q \rangle_u \leq 2^w - 1$, and from this range restriction we also get $\langle q - p \rangle_u = \langle q \rangle_u - \langle p \rangle_u$, thus $q - p \leq_u q$. □

Rewriting to one of these forms seems to be useful when $p - q$ is simpler than at least one of p or q.

Example 16. Consider the constraint $-x \leq_u -xy - x - 1$.

- It is equivalent to $x(y + 1) \leq_u xy$ by rewriting (6.1) $\implies$ (6.3).

- It is equivalent to $-x \leq_\mathsf{u} xy$ by rewriting (6.1) $\implies$ (6.2).
- It is equivalent to $x(y+1) \leq_\mathsf{u} x-1$ by rewriting (6.1) $\implies$ (6.4).

Lemma 2 (Strict-to-Nonstrict)**.** Since bit-vectors are discrete, we can represent strict inequalities as non-strict ones in different ways. However, side constraints are required due to overflow semantics.

$$p <_\mathsf{u} q \iff p \not\simeq -1 \wedge p+1 \leq_\mathsf{u} q \tag{6.7}$$
$$p <_\mathsf{u} q \iff q \not\simeq 0 \wedge p \leq_\mathsf{u} q-1 \tag{6.8}$$

Proof. We first show the direction from left to right of (6.7): we have $\langle p\rangle_\mathsf{u} < \langle q\rangle_\mathsf{u} \leq 2^w-1$, from which we get both $\langle p\rangle_\mathsf{u} \neq 2^w-1$ (i.e., $p \not\simeq -1$) as well as $\langle p+1\rangle_\mathsf{u} = \langle p\rangle_\mathsf{u}+1 \leq \langle q\rangle_\mathsf{u}$ (i.e., $p+1 \leq_\mathsf{u} q$).

In the other direction, we have $p \not\simeq -1$, i.e., $\langle p\rangle_\mathsf{u} < 2^w-1$, thus $\langle p\rangle_\mathsf{u}+1 \leq 2^w-1$, from this $\langle p+1\rangle_\mathsf{u} = \langle p\rangle_\mathsf{u}+1 \leq \langle q\rangle_\mathsf{u}$, and thus finally $\langle p\rangle_\mathsf{u} < \langle q\rangle_\mathsf{u}$.

The proof of (6.8) proceeds analogously. □

Lemma 3 (Inequality Flip)**.** The following implications hold:

$$p \not\simeq 0 \wedge p \leq_\mathsf{u} q \implies -q \leq_\mathsf{u} -p \tag{6.9}$$
$$p \not\simeq 0 \wedge p <_\mathsf{u} q \implies -q <_\mathsf{u} -p \tag{6.10}$$

Proof. To show (6.9), note that since $\langle p\rangle_\mathsf{u} \neq 0$ and $\langle p\rangle_\mathsf{u} \leq \langle q\rangle_\mathsf{u}$, we also have $\langle q\rangle_\mathsf{u} \neq 0$ and thus $\langle -p\rangle_\mathsf{u} = 2^w - \langle p\rangle_\mathsf{u} \geq 2^w - \langle q\rangle_\mathsf{u} = \langle -q\rangle_\mathsf{u}$.

(6.10) can be shown analogously. □

Lemma 4 (Inequality Across Slicing)**.** Consider concatenations of bit-vectors $x_1 +\!\!+ x_2$ and $y_1 +\!\!+ y_2$ with $|x_1| = |y_1|$ and $|x_2| = |y_2|$. The inequality can be split across the sub-slices:

$$x_1 +\!\!+ x_2 \leq_\mathsf{u} y_1 +\!\!+ y_2 \iff x_1 <_\mathsf{u} y_1 \vee (x_1 \simeq y_1 \wedge x_2 \leq_\mathsf{u} y_2)$$

Proof. The ordering $\leq_\mathsf{u}$ amounts to a lexicographic ordering over sequences of bits. Since x_1 and y_1 have the same size, we can obtain the result by grouping the bits according to the split into sub-slices. □

Lemma 5 (Inequality Constant Reduction)**.** Consider an equality of the form

$$2^k p + a \leq_\mathsf{u} 2^k q + b,$$

where $0 \leq k < w$ and a, b are constants with $0 \leq_\mathsf{u} a <_\mathsf{u} 2^k$ and $0 \leq_\mathsf{u} b <_\mathsf{u} 2^k$ (which is not a restriction, because the higher bits of the constant terms may be moved into the constant terms of p and q). We have

$$2^k p + a \leq_\mathsf{u} 2^k q + b \iff \begin{cases} 2^k p \leq_\mathsf{u} 2^k q & \text{if } a \leq_\mathsf{u} b, \\ 2^k p <_\mathsf{u} 2^k q & \text{otherwise.} \end{cases}$$

Proof. Corollary of Lemma 4. □

6.3 Intervals

Definition 12 (Wrapping Interval)**.** We use half-open *wrapping* intervals $[l; h[$ over bit-vectors, where the bit-vector l is the lower bound and the bit-vector h is the upper bound, defined as follows:

$$[l; h[\coloneqq \begin{cases} \{n \in \mathbb{Z}/2^w\mathbb{Z} \mid l \leq_\mathsf{u} n \text{ and } n <_\mathsf{u} h\} & \text{if } l \leq_\mathsf{u} h, \\ \{n \in \mathbb{Z}/2^w\mathbb{Z} \mid l \leq_\mathsf{u} n \text{ or } n <_\mathsf{u} h\} & \text{if } l >_\mathsf{u} h. \end{cases}$$

By this definition, if the endpoints are equal, the corresponding interval is empty: $[l; l[= \emptyset$. In general, the cardinality, or *length*, of an interval is

$$\text{len}([l; h[) = \langle h - l \rangle_\mathsf{u} = \begin{cases} \langle h \rangle_\mathsf{u} - \langle l \rangle_\mathsf{u} & \text{if } l \leq_\mathsf{u} h, \\ \langle h \rangle_\mathsf{u} - \langle l \rangle_\mathsf{u} + 2^w & \text{if } l >_\mathsf{u} h. \end{cases}$$

A constraint of the form $x \in [l; h[$ is equivalent to the bit-vector inequality $x - l <_\mathsf{u} h - l$, thus, no additional primitives are necessary to support such interval constraints in the bit-vector language. We list several additional equivalences that are used throughout the remainder of the thesis.

Lemma 6 (Wrapping Interval Equivalences)**.** The following constraints are equivalent for arbitrary bit-vectors x, l, h, a of the same size:

$$x - l <_\mathsf{u} h - l \tag{6.11}$$
$$x \in [l; h[\tag{6.12}$$
$$x + a \in [l + a; h + a[\tag{6.13}$$
$$-x \in [1 - h; 1 - l[\tag{6.14}$$

If $l \neq h$, the above constraints are also equivalent to

$$x \notin [h; l[\tag{6.15}$$

Proof. We first show (6.12) $\iff$ (6.11). Intuitively, the distance of elements x from the starting point l must be less than the length $h - l$ of the interval.

Formally, let us first consider the case $l \leq_\mathsf{u} h$. Under this condition, we have $\langle h - l \rangle_\mathsf{u} = \langle h \rangle_\mathsf{u} - \langle l \rangle_\mathsf{u}$, and (6.12) is equivalent to $l \leq_\mathsf{u} x \wedge x <_\mathsf{u} h$. From $l \leq_\mathsf{u} x$ and $x <_\mathsf{u} h$ we get $\langle x - l \rangle_\mathsf{u} = \langle x \rangle_\mathsf{u} - \langle l \rangle_\mathsf{u} < \langle h \rangle_\mathsf{u} - \langle l \rangle_\mathsf{u} = \langle h - l \rangle_\mathsf{u}$, thus (6.11) holds. In the other direction, (6.11) is given. Towards a contradiction, assume $l <_\mathsf{u} x$. This means $\langle x - l \rangle_\mathsf{u} = \langle x \rangle_\mathsf{u} - \langle l \rangle_\mathsf{u} + 2^w$, and thus (6.11) is equivalent to $\langle x \rangle_\mathsf{u} - \langle l \rangle_\mathsf{u} + 2^w < \langle h \rangle_\mathsf{u} - \langle l \rangle_\mathsf{u}$, which

simplifies to $\langle x\rangle_\mathsf{u}+2^w < \langle h\rangle_\mathsf{u}$, contradicting the fact that the range of $\langle\cdot\rangle_\mathsf{u}$ is $\{0,\dots,2^w-1\}$. We have shown $l \leq_\mathsf{u} x$. From this, (6.11) is equivalent to $\langle x\rangle_\mathsf{u} - \langle l\rangle_\mathsf{u} < \langle h\rangle_\mathsf{u} - \langle l\rangle_\mathsf{u}$, simplifying to $\langle x\rangle_\mathsf{u} < \langle h\rangle_\mathsf{u}$, from which we can conclude $x <_\mathsf{u} h$.

Now consider the second case $l <_\mathsf{u} h$. Here, we have $\langle h-l\rangle_\mathsf{u} = \langle h\rangle_\mathsf{u} - \langle l\rangle_\mathsf{u} + 2^w$, and (6.12) is equivalent to $l \leq_\mathsf{u} x \vee x <_\mathsf{u} h$. In the direction starting with (6.12), if we have $l \leq_\mathsf{u} x$, then $\langle x-l\rangle_\mathsf{u} = \langle x\rangle_\mathsf{u} - \langle l\rangle_\mathsf{u}$, and $\langle x\rangle_\mathsf{u} < \langle h\rangle_\mathsf{u} + 2^w$ due to the range of $\langle\cdot\rangle_\mathsf{u}$. If we instead have $x <_\mathsf{u} h$, then also $x <_\mathsf{u} l$ and $\langle x-l\rangle_\mathsf{u} = \langle x\rangle_\mathsf{u} - \langle l\rangle_\mathsf{u} + 2^w$. In both cases we can conclude $\langle x\rangle_\mathsf{u} - \langle l\rangle_\mathsf{u} < \langle h\rangle_\mathsf{u} - \langle l\rangle_\mathsf{u}$ and thus (6.11). In the other direction, the case $l \leq_\mathsf{u} x$ is trivial. Otherwise, we have $l >_\mathsf{u} x$, which means together with $l <_\mathsf{u} h$ that (6.11) is equivalent to $\langle x\rangle_\mathsf{u} - \langle l\rangle_\mathsf{u} + 2^w < \langle h\rangle_\mathsf{u} - \langle l\rangle_\mathsf{u} + 2^w$, which simplifies to $\langle x\rangle_\mathsf{u} < \langle h\rangle_\mathsf{u}$, allowing us to conclude (6.12).

For (6.13) $\iff$ (6.11), it is enough to apply some simplifications:

$$\begin{aligned}
&x + a \in [l + a; h + a[\\
&\iff (x + a) - (l + a) <_\mathsf{u} (h + a) - (l + a) && \text{by (6.12)} \iff \text{(6.11)} \\
&\iff x - l <_\mathsf{u} h - l && \text{by simplification}
\end{aligned}$$

For (6.14) $\iff$ (6.11), we apply Lemma 1 to obtain the result:

$$\begin{aligned}
&-x \in [1 - h; 1 - l[\\
&\iff h - x - 1 <_\mathsf{u} h - l && \text{by (6.12)} \iff \text{(6.11)} \\
&\iff x - l <_\mathsf{u} h - l && \text{by (6.1)} \iff \text{(6.2)}
\end{aligned}$$

For (6.12) $\iff$ (6.15), note that since $l \neq h$, the intervals $[l; h[$ and $[h; l[$ fall into different cases of Definition 12, i.e., we have

$$x \in [l; h[\iff l \leq_\mathsf{u} x \wedge x <_\mathsf{u} h \iff \neg(h \leq_\mathsf{u} x \vee x <_\mathsf{u} l) \iff x \notin [h; l[$$

for $l <_\mathsf{u} h$, and analogous for the other case. □

CHAPTER 7

PolySAT in a Nutshell

PolySAT serves as a decision procedure for bit-vector constraints and is developed as a theory solver within the SMT solver Z3 [dMB08]. An overview of the PolySAT architecture is given in Figure 7.1, with further details on key ingredients in Chapters 8–10.

In a nutshell, PolySAT consists of two inter-connected components that interact for theory solving in an SMT setting:

1. A *bit-vector plugin to the e-graph* (short for equality graph [DNS05, WNW+21]). This plugin handles structural constraints that involve multiple bit-vector sizes (concatenation, extraction) and determines canonical sub-slices of bit-vectors. The PolySAT e-graph plugin also propagates assigned values across bit-vector slices.

2. A *theory solver*, which handles the remaining constraints by translating them into *polynomial constraints* (Figure 7.2) and builds on information from the e-graph plugin to search for a satisfiable assignment (Chapters 8–10).

From its e-graph, PolySAT receives Boolean assignments to bit-vector constraints, and equality propagations between bit-vector terms. In return, the theory solver of PolySAT produces a satisfying assignment, or a conflicting subset of the received constraints. We next discuss these two components, and then focus on the theory solving aspects of PolySAT in Chapters 8–10.

7.1 E-graph Plugin

In SMT solving, an e-graph [DNS05, WNW+21] is typically shared between theory solvers. The primary purpose of the e-graph is to infer equalities that follow from congruence reasoning. For PolySAT, the e-graph is extended with theory reasoning for bit-vectors. Theory reasoning is dispatched when the e-graph merges two terms of bit-vector sort.

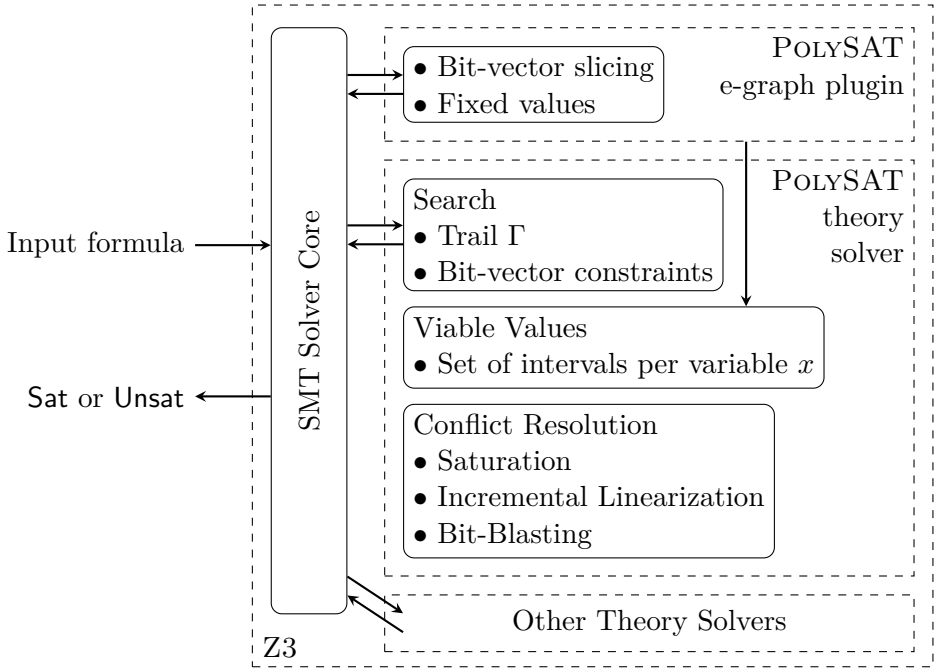


Figure 7.1: Integration of PolySAT into the SMT Solver Z3

Certain types of constraints fix some or all bits of a variable ("fixed bits"); the simplest example would be an equality such as $x \simeq 0$. The e-graph plugin of PolySAT determines such fixed bits and performs constant propagation over bit-vector extraction and concatenation. Furthermore, the PolySAT e-graph establishes equalities between bit-vector ranges. For example, it infers that $x[5{:}4] = x[1{:}0]$ from the equation $x[5{:}2] = x[3{:}0]$.

We note that congruence reasoning for bit-vectors was also considered in [MR98, BP98, BS09]. Moreover, e-graphs are also used for constant propagation in [GJD20]. The PolySAT integration of theory plugins to the e-graph structure is generic and not specific to bit-vectors.

7.2 Theory Solver

In this section, we describe the theory solver component of PolySAT. Note that all bit-vector extractions and concatenations have been handled by the e-graph before the corresponding constraints reach the theory solver. The theory solver queries the e-graph to find sub-slice relations when necessary.

7.2.1 Propositional Search

The propositional search is driven by the CDCL(T) core of the SMT solver [BS97, MS99, MLM21]. PolySAT receives Boolean assignments to bit-vector constraints and equality propagations between bit-vector terms. Both of them are internalized as PolySAT

$p \leq_{\mathsf{u}} q$	unsigned inequality
$\Omega^*(p, q)$	multiplicative overflow
$x \simeq p \,\&\, q$	bit-wise *and*
$x \simeq p \mid q$	bit-wise *or*
$x \simeq p \ll q$	left shift
$x \simeq p \gg q$	logical right shift
$x \simeq p \gg_{\mathsf{a}} q$	arithmetic right shift
$x \simeq p \,/\, q$	unsigned division
$x \simeq p \,\%\, q$	unsigned remainder

Figure 7.2: Primitive constraints

$p <_{\mathsf{u}} q$	$\rightsquigarrow$	$\neg(q \leq_{\mathsf{u}} p)$
$p \leq_{\mathsf{s}} q$	$\rightsquigarrow$	$p + 2^{w-1} \leq_{\mathsf{u}} q + 2^{w-1}$
$p \simeq q$	$\rightsquigarrow$	$p - q \leq_{\mathsf{u}} 0$
$\Omega^+(p, q)$	$\rightsquigarrow$	$p + q <_{\mathsf{u}} p$
$p[i]$	$\rightsquigarrow$	$2^{w-1} \leq_{\mathsf{u}} 2^{w-i-1} p$
$p - q$	$\rightsquigarrow$	$p + (2^w - 1)q$
$\sim p$	$\rightsquigarrow$	$-p - 1$

Figure 7.3: Some of the derived constraints/expressions ($w = |p| = |q|$)

constraints (cf. Figure 7.2) and tracked by the *trail* Γ, which is a list of the currently active PolySAT constraints. PolySAT maintains the invariant that each element of Γ is justified by previous elements, and that each constraint and variable is assigned at most once in Γ.

7.2.2 Constraints

As a whole, PolySAT fully supports the standardized bit-vector logic of SMT-LIB [BFT16]. Extraction and concatenation are handled by PolySAT's e-graph plugin, as discussed before (Section 7.1), while other bit-vector constraints are passed to PolySAT's theory solver. Figure 7.2 depicts the *primitive constraints*, where p, q are bit-vector polynomials and x is a bit-vector variable. Other constraints are either reduced to primitive constraints as shown in Figure 7.3, or axiomatized upfront.

Constraints of the form $x \simeq n$, where x is a variable and n is a bit-vector constant, are called *variable assignments*. Bit-vector terms p and constraints c can be evaluated w.r.t. the current trail Γ, that is, we substitute the variable assignments in Γ into p and c, respectively, and simplify. As a shorthand, we write $\widehat{p}$ for the evaluation of p under the current trail.

7.2.3 Axiomatized Operations

As mentioned before, some operations are axiomatized upfront. For example, to internalize the (unsigned) division $x \,/\, y$, PolySAT introduces fresh variables $u \coloneqq x \,/\, y$ and $v \coloneqq x \,\%\, y$ for the quotient and remainder, respectively. The main axiom is $x \simeq uy + v$, but for correctness in bit-vector logic, four more axioms are required:

$$\begin{aligned} &\neg\Omega^*(u, y) && y \not\simeq 0 \to v <_\mathsf{u} y \\ &\neg\Omega^+(uy, v) && y \simeq 0 \to u \simeq -1 \end{aligned}$$

where $\neg\Omega^+(uy, v)$ means that the addition $uy + v$ does not overflow, which can be implemented, e.g., as the constraint $uy \leq_\mathsf{u} -v - 1$.

7.2.4 Simplifying Rewrites

The canonical form of an equality constraint $p \simeq 0$ within the PolySAT solver is the unsigned inequality $p \leq_\mathsf{u} 0$. However, there are several other syntactic forms of inequalities that constrain a term to a single value and thus effectively represent equalities. Since equalities admit more reasoning than general inequalities, it is helpful to recognize such cases. For example, $1 >_\mathsf{u} p$, $-1 \leq_\mathsf{u} p - 1$, $p \leq_\mathsf{u} p - 1$, and $p - 1 >_\mathsf{u} -2$ are all equivalent to $p \simeq 0$.

PolySAT applies the rewrite rules listed in Figure 7.4 to newly created inequality constraints until fixpoint. The rewrite rules as listed in the table are applied from left to right; however logically each rule must be a valid equivalence. Some of the rewrite rules are somewhat cosmetic but still helpful to reduce different occurrences of equivalent constraints (e.g., "$-p$" indicates that the sign bit of the leading coefficient is negative, where the leading coefficient is arbitrarily but consistently chosen).

7.2.5 Constraint Solving

The PolySAT theory solver uses a waterfall model of refinements to generate lemmas on demand, using the following steps:

1. *Propagation*: Value propagation is triggered when a variable is assigned a value (Section 8.1).

2. *Viable Interval Conflict*: If propagation tightens the feasible intervals of a variable to the empty set, the solver yields an interval conflict (Section 8.3).

3. *Case Split on Viable Candidates*: If no further propagation is possible, and there are no interval conflicts, the solver picks a value for the next unassigned variable, if any. It produces a literal $x \simeq n$ for the CDCL solver to case split on, with a preference to the phase $x \simeq n$ over $x \not\simeq n$. The constant n is chosen to be outside the ranges of infeasible intervals stored for x so far (Section 8.2).

4. *Saturation Lemmas*: Saturation lemmas let us propagate consequences from non-linear constraints (Section 10.1).

5. *Incremental Linearization*: Our solver includes incremental linearization rules for the cases where variables are 0, 1, −1, or powers of two (Section 10.2).

6. *Bit-blasting*: As a final resort, PolySAT admits bit-blasting rules (Section 10.3).

The first three steps above (steps 1, 2, 3) operate on linear constraints, or rather, a *linear abstraction* of the original constraints, where non-linear monomials are treated as variables themselves. If no conflicts arise from the linear abstraction, then any conflicting non-linear constraints are handled by the latter stages (steps 4, 5, 6 above).

A conflict at any stage will cause PolySAT to return a conflict lemma to the SMT solver core, which will then backtrack and continue with search. When control is passed to PolySAT the next time, theory solving in PolySAT will begin again in the above step 1 of constraint solving. If, on the other hand, all stages pass without conflict, a model has been found and PolySAT returns Sat.

$0 \leq_u p$	$\rightsquigarrow$	$\top$	
$p \leq_u -1$	$\rightsquigarrow$	$\top$	
$p \leq_u p$	$\rightsquigarrow$	$\top$	
$n_1 \leq_u n_2$	$\rightsquigarrow$	$\top$	if $\langle n_1 \rangle_u \leq \langle n_2 \rangle_u$
$n_1 \leq_u n_2$	$\rightsquigarrow$	$\bot$	if $\langle n_1 \rangle_u > \langle n_2 \rangle_u$
$p \leq_u q$	$\rightsquigarrow$	$p \leq_u p - q - 1$	if $\lvert\mathcal{V}(p-q)\rvert < \lvert\mathcal{V}(p)\rvert \leq \lvert\mathcal{V}(q)\rvert$
$p \leq_u q$	$\rightsquigarrow$	$q - p \leq_u q$	if $\lvert\mathcal{V}(p-q)\rvert < \lvert\mathcal{V}(q)\rvert < \lvert\mathcal{V}(p)\rvert$
$-p + n \leq_u n$	$\rightsquigarrow$	$p \leq_u n$	
$n \leq_u p + n$	$\rightsquigarrow$	$p \leq_u -n - 1$	
$n \leq_u -p$	$\rightsquigarrow$	$p - 1 \leq_u -n - 1$	
$-p \leq_u n$	$\rightsquigarrow$	$-n - 1 \leq_u p - 1$	
$-p \leq_u 0$	$\rightsquigarrow$	$p \leq_u 0$	
$n \leq_u 2^{w-1}p + q + n - 1$	$\rightsquigarrow$	$n \leq_u 2^{w-1}p - q$	
$p - n \leq_u -n - 1$	$\rightsquigarrow$	$n \leq_u p$	
$n \leq_u 2^{w-1}p + q + n - 1$	$\rightsquigarrow$	$n \leq_u 2^{w-1}p - q$	
$-1 \leq_u p$	$\rightsquigarrow$	$p + 1 \simeq 0$	
$1 \leq_u p$	$\rightsquigarrow$	$p \not\simeq 0$	
$p \leq_u -2$	$\rightsquigarrow$	$p + 1 \not\simeq 0$	
$2p + 1 \leq_u 0$	$\rightsquigarrow$	$\bot$	
$np + q \leq_u 0$	$\rightsquigarrow$	$p + n^{-1}q \leq_u 0$	if n odd
$2^k p + n_1 \leq_u 2^k q + n_2$	$\rightsquigarrow$	$2^k p \leq_u 2^k q$	if $\langle n_1 \rangle_u \leq \langle n_2 \rangle_u \leq 2^k$
$2^k p + n_1 \leq_u 2^k q + n_2$	$\rightsquigarrow$	$2^k p <_u 2^k q$	if $\langle n_2 \rangle_u < \langle n_1 \rangle_u \leq 2^k$
$2^k p \leq_u n$	$\rightsquigarrow$	$\top$	if $\langle n \rangle_u + 2^k \geq 2^w$

Figure 7.4: Rewrite Rules for Inequalities, where p, q are bit-vector polynomials and n, n_1, n_2 are bit-vector values.

CHAPTER 8

Tracking Viable Values

A crucial part of the PolySAT theory solver tracks for each bit-vector variable x an over-approximation of the set of feasible values under the current trail Γ, which we call the viable values of x. More precisely, we say a value n is *viable* for a variable x if the assignment $x \simeq n$ is feasible for the linear abstraction of the current set of constraints. Specifically, the set of viable values is represented as a set of *forbidden intervals*, each of which excludes a certain range of values of x, and is justified by constraints in the current trail Γ.

In PolySAT, we adapt forbidden intervals from [GJD20] and use intervals for propagating and querying viable values of variables (Sections 8.1–8.2), and resolving respective conflicts (Section 8.3). Our approach extends [GJD20] by computing intervals when the coefficient of x is not a power of two (Section 9.2), or when the coefficients are different on both sides of an inequality (Section 9.3).

8.1 Value Propagation

PolySAT extracts forbidden intervals from inequalities and overflow constraints c that are linear in x under the current trail Γ. Along with each such interval a bit-width k is stored; $k \neq w$ means that the interval only constrains the lower k bits of x (e.g., the constraint $8x + p \leq_\mathsf{u} q$ only affects the lower $w - 3$ bits of x, if x appears neither in p nor q).

Definition 13 (Forbidden Interval)**.** Formally, a *forbidden interval* for a constraint c consists of an interval $[l; h[$, side conditions $c_1, \ldots, c_n$ that hold under Γ, and a bit-width k such that

$$c \wedge c_1 \wedge \cdots \wedge c_n \implies x[k-1{:}0] \notin [l; h[.$$

Algorithm 8.1: PolySAT Viable Value Query

Input : Set of forbidden intervals $\mathcal{I}$, set C of constraints
Output : Viable value x_0, or a conflict

1 $x_0 \leftarrow x_{prev}$ ▷ Start at previous viable value
2 $\mathcal{J} \leftarrow \langle\rangle$ ▷ Justification (sequence of visited intervals)
3 **loop**
4 **while** $\exists I \in \mathcal{I}$ *such that* $x_0 \in I$ **do**
5 Choose such an $I \in \mathcal{I}$ with smallest bit-width
6 $\mathcal{J} \leftarrow \langle \mathcal{J}; I \rangle$
7 $x_0 \leftarrow forward(x_0, I)$
8 **if** $isConflict(\mathcal{J})$ **then return** Conflict $\mathcal{J}$
9 **end**
10 **if** x_0 *does not violate any* $c \in C$ **then return** x_0
11 $\mathcal{I} \leftarrow \mathcal{I} \cup \{computeInterval(C, x_0)\}$
12 **end**

Intervals are ordered by their starting points, and we drop intervals that are fully contained in other intervals. Chapter 9 explains how intervals are obtained from constraints.

Value propagation in PolySAT is triggered when a variable is assigned a value, or in other words, the solver is presented with a literal $x \simeq n$, where x is a variable and n is a value. Propagation is limited to linear occurrences of variables. For example, if x is assigned 2, then from $x + y \geq_\mathsf{u} 10$, the forbidden intervals for y are updated to cover $y \notin [-2; 8[$. On the other hand, for $xz + y \geq_\mathsf{u} 10$, where x occurs in a non-linear term, there is no propagation. Non-linear propagation in PolySAT is currently side-stepped because we noticed that it produces very weak lemmas from viable interval conflicts. Non-linear conflicts are therefore handled separately, see Chapter 10.

8.2 Viable Value Query

To find a viable value for variable x, we collect the forbidden intervals $\mathcal{I}$ over the prefixes $x[k{:}0]$ of x for $0 \leq k < w$. In this context, if an interval $I \in \mathcal{I}$ is an interval for $x[k{:}0]$, we say I *has bit-width* $k + 1$. In addition, we consider intervals for variables that are equivalent to a prefix of x, as determined by the current state of PolySAT's e-graph.

In addition to forbidden intervals, we keep track of the set C of constraints that are linear in x. We then invoke Algorithm 8.1 to either find a value for x or detect a conflict. We adjust [GJD20] by using intervals to track viable values and detecting conflicts in Algorithm 8.1 as follows.

Algorithm 8.1 starts out with the previous viable value x_{prev} of x, initially set to 0. Then, in the loop of Algorithm 8.1, we check whether any of the known intervals $\mathcal{I}$ contain the current candidate value x_0 of x. If that is not the case, then the current value x_0 is compatible with the intervals in $\mathcal{I}$. We additionally test x_0 for admissibility against the set C of constraints (line 10 of Algorithm 8.1). If none of these constraints are

violated, the candidate value x_0 is returned as viable value for x. Otherwise (line 11 of Algorithm 8.1), *computeInterval*(C, x_0) extracts a new interval that covers x_0 (cf. Chapter 9) and the search for a viable value of x continues. If, on the other hand, the current value x_0 of x is contained in some forbidden interval, we choose an interval I of minimal bit-width among these (line 5 of Algorithm 8.1) and record it in the list $\mathcal{J}$ of justifications (line 6 of Algorithm 8.1).

The candidate value x_0 of x is updated to *forward*(x_0, I), the first value after x_0 that is not covered by I (line 7 of Algorithm 8.1). If a conflict is detected (line 8 of Algorithm 8.1), the justifications $\mathcal{J}$ are returned for further processing (see Section 8.3).

The following example illustrates Algorithm 8.1.

Example 17. Assume PolySAT needs to determine a viable value for the variable y where the set of initially known intervals $\mathcal{I}$ is empty, the set of constraints C consists of the three constraints c_1, c_2, c_3 listed below, and the trail contains the assignment $x \simeq 11$.

	Constraint	Equivalent Interval	Concrete Interval
c_1	$4 \leq_\mathsf{u} y$	$y \notin [0; 4[$	$y \notin [0; 4[$
c_2	$y \leq_\mathsf{u} 15$	$y \notin [16; 0[$	$y \notin [16; 0[$
c_3	$x + 3 \leq_\mathsf{u} y + 5$	$y \notin [-5; x - 2[$ when $x + 3 \not\simeq 0$	$y \notin [-5; 9[$

To find a viable value, PolySAT invokes Algorithm 8.1. As this is the first invocation, we begin with $y_0 = 0$. The condition of the while loop (line 4) is trivially false. Since constraint c_1 is false for $y = 0$, we extract the interval $[0; 4[$ as described in Chapter 9 and add it to $\mathcal{I}$ (line 11).

In the next iteration of the algorithm's outer loop (line 3), we enter the while loop body with the interval $I = [0; 4[$; advancing y_0 to the value 4 and recording I in the justifications $\mathcal{J}$. This time, we find y_0 violates constraint c_3 (line 10). As before, we extract the interval $[-5; 9[$ and store it in $\mathcal{I}$.

In the third iteration of the outer loop (line 3), we advance y_0 to 9 and record $[-5; 9[$ in the set of justifications $\mathcal{J}$. Further, we discover that $y \simeq 9$ does not contradict any of the relevant constraints C (line 10) and the algorithm terminates. We conclude that 9 is a viable value for y.

Consider now a subsequent invocation of Algorithm 8.1 for y with the additional constraint c_4 (listed below), i.e., $C = \{c_1, c_2, c_3, c_4\}$. This situation may happen in PolySAT if a conflict occurs in the branch with $y \simeq 9$ and c_4 is then derived by conflict resolution.

	Constraint	Equivalent Interval	Concrete Interval
c_4	$y - 12 \leq_\mathsf{u} y - x + 4$	$y \notin [x - 4; 12[$	$y \notin [7; 12[$

Note that the previously discovered viable value now serves as starting point for the search, i.e., the second invocation of Algorithm 8.1 begins with $y_0 = 9$. We notice that c_4

is violated for this value of y, compute the interval $[7; 12[$, and advance to $y_0 = 12$. Now, none of the constraints are violated and the algorithm returns the viable value 12.

Finally, consider a third invocation of Algorithm 8.1 for y with the additional constraint $c_5 \colon y + 1 \leq_u 11$, which is equivalent to the interval $y \notin [11; -1[$. This time, the search starts at $y_0 = 12$. We find, in the listed order, the intervals $[11; -1[$, $[-5; 9[$, $[7; 12[$ and finally, $[11; -1[$ again. At this point, a conflict is detected, justified by the listed intervals. The algorithm terminates and returns the justification for the conflict for further processing.

Note that, while constraint c_2 is relevant for y, it has been skipped in the algorithm because its interval has been covered by other constraints.

Remark 6. In the above Example 17, we obtained the intervals $[0; 4[$ and $[-5; 9[$. The former is fully contained in the latter and thus unnecessary to determine the viable value. In our implementation, we prune such subsumed intervals from the set $\mathcal{I}$ and from the justifications $\mathcal{J}$.

8.3 Interval Conflict

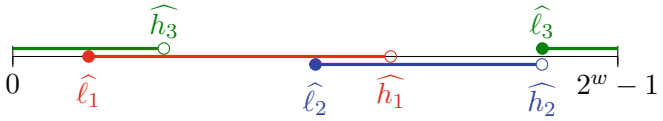
We detect conflicts by examining the list of justifications $\mathcal{J}$ after appending a new interval I to $\mathcal{J}$. The condition $\mathit{isConflict}(\mathcal{J})$ in Algorithm 8.1 is true iff the latest interval I has already been visited previously, and no interval of larger bit-width has occurred in between. Let $I_1, \ldots, I_{n+1}$ denote this subsequence of intervals, where $I_1 = I_{n+1} = I$, and let $I_i = [l_i; h_i[$. To block the current assignment to x, PolySAT creates a conflict lemma from $I_1, \ldots, I_{n+1}$ and reports it to its SMT core.

The PolySAT conflict lemma is designed to capture the following fact: the union of $I_1, \ldots, I_n$ covers the full domain $\mathbb{Z}/w\mathbb{Z}$, and the intervals have been chosen such that each upper bound h_i is contained in the next interval I_{i+1}. In other words, as long as $h_i \in I_{i+1}$ holds, for all i, and the intervals are valid for x, there can be no feasible value for x. The PolySAT conflict lemma is similar to the one of [GJD20], however, the representation of constraints $h_i \in I_{i+1}$ is different, because PolySAT currently tries to avoid introducing new extract-expressions.

Since the constraints $h_i \in I_{i+1}$ do not contain x itself, they are useful for formulating a conflict lemma. Let C_i denote the set consisting of the constraint and side conditions of I_i. Then, the PolySAT conflict lemma in its basic form is

$$\bigwedge_{i=1}^{n} C_i \wedge \bigwedge_{i=1}^{n} h_i \in I_{i+1} \implies \bot. \tag{8.1}$$

To illustrate the idea of conflict lemma generation in PolySAT, consider three intervals $[l_1; h_1[$, $[l_2; h_2[$, $[l_3; h_3[$ whose concrete evaluation under the current trail Γ covers the full domain by forming the following configuration:



Assuming the three intervals are justified by constraints C_1, C_2, C_3, respectively, the PolySAT conflict lemma is

$$\bigwedge C \wedge h_1 \in [l_2; h_2[\wedge h_2 \in [l_3; h_3[\wedge h_3 \in [l_1; h_1[\implies \bot,$$

where $C \coloneqq C_1 \cup C_2 \cup C_3$.

An alternative to the linking constraint $h_i \in I_{i+1}$ can be obtained from the dual version using lower bounds, i.e., $l_{i+1} - 1 \in I_i$.

Example 18. Consider a solver state where variables x, z are assigned and a conflict is caused by the literals $x + y - z \simeq 0$ and $x + y \leq_\mathsf{u} 1$. Since x and z have been assigned first, the two literals translate into the y-intervals $y \notin [z - x + 1; z - x[$ and $y \notin [2 - x; -x[$, respectively (details about the translation into intervals are given in Section 9.2).

By processing the conflict as explained in this section, we obtain the linking constraints $z - x \in [2 - x; -x[$ and $-x \in [z - x + 1; z - x[$. Translating into bit-vector language, we obtain $z - 2 <_\mathsf{u} -2$ and $-z - 1 <_\mathsf{u} -1$, respectively. The conflict lemma generated according to (8.1) is then

$$x + y - z \not\simeq 0 \vee x + y >_\mathsf{u} 1 \vee z - 2 \geq_\mathsf{u} -2 \vee -z - 1 \geq_\mathsf{u} -1. \tag{8.2}$$

8.4 Lemma Simplification by Subsumption

We say a conflict lemma is *asserting* if, after backtracking, exactly one literal in the lemma is unassigned. In general, we want conflict lemmas to be asserting because such lemmas clearly represent information learned from the conflict.

However, conflict resolution as discussed above may sometimes create non-asserting lemmas. In some cases, we can reduce such lemmas by applying subsumption resolution with a suitable theory clause, as illustrated in the following example.

Example 19. We continue Example 18. Note that the generated conflict lemma (8.2) is not asserting when the solver backtracks to a level where z is unassigned. In general, the SMT solver core (external to PolySAT) will choose one of the branches to proceed.

In this case, however, there is a simple way to reduce/strengthen the conflict lemma: note that $z - 2 \geq_\mathsf{u} -2$ iff $z \in \{0, 1\}$ and $-z - 1 \geq_\mathsf{u} -1$ iff $z \in \{0\}$, which means $-z - 1 \geq_\mathsf{u} -1$ implies $z - 2 \geq_\mathsf{u} -2$, thus the theory lemma

$$-z - 1 <_\mathsf{u} -1 \vee z - 2 \geq_\mathsf{u} -2 \tag{8.3}$$

is valid.

We can now apply subsumption resolution (see Definition 5) with the theory lemma as side premise to reduce the conflict clause (8.2) to

$$x + y - z \not\simeq 0 \vee x + y >_{\mathsf{u}} 1 \vee z - 2 \geq_{\mathsf{u}} -2,$$

which is asserting.

In the context of PolySAT, these theory lemmas are implicit. We say that literal $-z - 1 \geq_{\mathsf{u}} -1$ is subsumed by literal $z - 2 \geq_{\mathsf{u}} -2$, and reduce conflict clauses by removing subsumed literals. Simple theory implications of the same kind as (8.3) are easy to detect by comparing the equivalent intervals. Further simplifications may be obtained from lemmas such as $\Omega^*(x, y) \implies x >_{\mathsf{u}} 1$, or by combining adjacent intervals into a single constraint (e.g., as in $x = 3 \vee x >_{\mathsf{u}} 3 \iff x \geq_{\mathsf{u}} 3$).

CHAPTER 9

Computing Intervals

We now describe how forbidden intervals are extracted from a constraint $c \in C$ that is linear in the variable x under consideration. Intervals may be computed on demand, relative to a given candidate value (sample point) x_0 of x: the goal is then to find a maximal interval of x-values around x_0 that are excluded by the constraint c. In practice, we note the intervals are often not strictly maximal, but as large as reasonably possible to compute.

9.1 Fixed Bits

The e-graph plugin of PolySAT tracks fixed values for variables and their sub-slices. If the sample point x_0 contradicts the sub-slice assignment $x[h{:}l] = n$, the forbidden interval $x[h{:}0] \notin [2^l(n+1); 2^l n[$ is created. Note that fixed values for sub-slices may also be encoded as inequalities, for example, as follows:

Fixed slice	Equivalent Constraint	
$x[i]$	$2^{w-i-1}x \geq_\mathsf{u} 2^{w-1}$	
$x[h{:}0] = n$	$2^k x = 2^k n$	$k \coloneqq w - h - 1$
$x[h{:}l] = n$	$2^k x - 2^{k+l} n <_\mathsf{u} 2^{k+l}$	$k \coloneqq w - h - 1$

Such (inequality) constraints are turned into appropriate intervals as described in Section 9.2. We remark that it is not necessary to recover sub-slice assignments by recognizing certain patterns of constraints.

9.2 Linear Inequality with Equal Coefficients

Consider the inequality constraint $px + q \leq_\mathsf{u} rx + s$ that is linear in x. In the cases where either p or r evaluate to 0 or both to the same value a, the inequality constraint

is equivalent to an interval constraint [GJD20], according to the following table, and subject to side conditions $p = \widehat{p}$ and $r = \widehat{r}$:

Constraint under Γ	Forbidden Interval	Condition
$ax + \widehat{q} \leq_{\mathsf{u}} \widehat{s}$	$ax \notin [s - q + 1; -q[$	$s \neq -1$
$\widehat{q} \leq_{\mathsf{u}} ax + \widehat{s}$	$ax \notin [-s; q - s[$	$q \neq 0$
$ax + \widehat{q} \leq_{\mathsf{u}} ax + \widehat{s}$	$ax \notin [-s; -q[$	$q \neq s$

For the following, assume we have the interval constraint $ax \notin [l; h[$. Yet, we want to extract an interval on x, rather than on ax.

Case $\mathbf{a = \pm 1}$. The case $a = 1$ trivially gives an interval on x. In the case $a = -1$ (i.e., $2^w - 1$), the transformation $-x \in [l; h[\Leftrightarrow x \in [1 - h; 1 - l[$ from Lemma 6 is applied.

Case $\mathbf{a = 2^k a'}$ (reducing the bit-width). Consider the case where a is divisible by 2^k for some $k > 0$. Due to the factor 2^k, the upper k bits of x do not influence the value of the constraint. In this case, we consider an interval for the prefix $x[w - k - 1{:}0]$ of x:

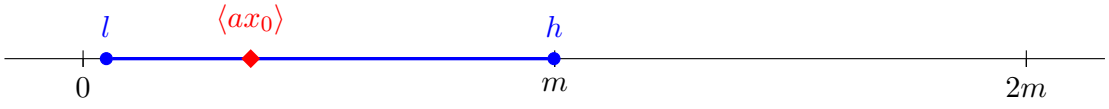
$$2^k a' x \notin [l; h[\iff \begin{cases} a' x[w - k - 1{:}0] \notin [l'; h'[& \text{if } l' \neq h' \\ 0 \notin [l; h[& \text{otherwise} \end{cases}$$

where $\beta' \coloneqq \lceil \frac{\beta}{2^k} \rceil \bmod 2^{w-k}$ for $\beta \in \{l, h\}$.

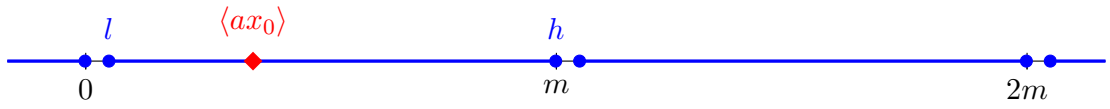
Other values of a. For other values of a, in general, multiple disjoint intervals exist. We extract intervals around a sample point x_0 on demand, i.e., given concrete values $a, x_0, l, h \in \mathbb{Z}/2^w\mathbb{Z}$ such that $ax_0 \in [l; h[$, the task is to compute the maximal x-interval $[x_l; x_h[$ such that $ax \in [l; h[$ for all $x \in [x_l; x_h[$. To compute x_l and x_h, we move the problem into the integers $\mathbb{Z}$ and work with non-wrapping intervals. Operations until the end of this section are therefore to be understood as operations in $\mathbb{Z}$.

Let w be a fixed bit-width and let $m \coloneqq 2^w$. Assume values $a, x_0, l, h \in \mathbb{Z}$ are given such that $1 \leq a < m$, $-m < l \leq h < m$, and $ax_0 \bmod m \in [l; h]$. Furthermore, the length of the interval should be less than m, i.e., $h - l + 1 < m$ (otherwise the computation is unnecessary because the corresponding modular interval covers the whole domain). The goal is to find the minimal x_l and the maximal x_h such that $ax \bmod m \in [l; h]$ for all $x \in [x_l; x_h]$.

Let $k_0 \in \mathbb{Z}$ such that $l \leq ax_0 + k_0 m \leq h$ (this value exists and is unique under the aforementioned conditions). To simplify notation, define $\langle x \rangle \coloneqq x + k_0 m$. The initial configuration is illustrated by the following diagram:

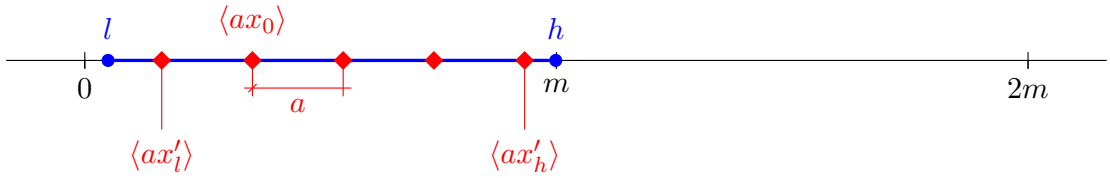


Since we are ultimately interested in the modular interval $[l; h] \bmod m$ over $\mathbb{Z}/m\mathbb{Z}$, we consider the set of all representatives of elements of that interval, i.e., the union of $[l; h] + im$ for all $i \in \mathbb{Z}$, as depicted in the following diagram.

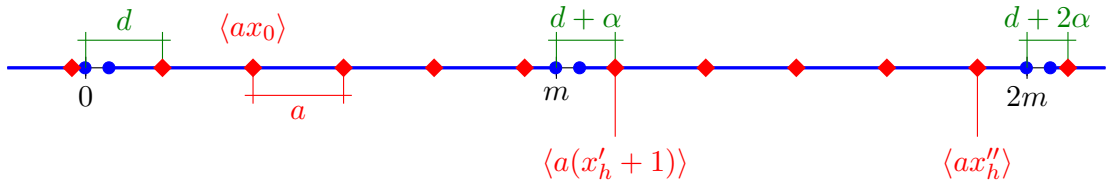


The underlying idea of our procedure is to look at each interval representative $[l; h] + im$ separately (intuitively, as a region where no overflow occurs) and take advantage of periodicity after each overflow.

In the first step, we compute the minimal x'_l and the maximal x'_h such that $l \leq \langle ax \rangle \leq h$ for all $x \in [x'_l; x'_h]$. Intuitively, $[x'_l; x'_h]$ is the maximal x-interval around x_0 such that no overflow occurs among the corresponding multiples of a.



However, the interval $[x'_l; x'_h]$ is often far from optimal, causing repeated queries over the same constraint in Algorithm 8.1. In case of the upper bound, this means that $\langle a(x'_h + 1) \rangle$ is contained in the next interval representative $[l; h] + m$. The following diagram illustrates the multiples of a across several interval representatives.



The situation in the second interval $[l; h] + m$ is very similar to the initial setting. However, the multiples of a (depicted by red diamonds) have shifted by some amount α relative to the interval.

In the example illustrated in the diagrams we have $\alpha < 0$, i.e., with each overflow, the multiples of a drift to the left (relative to the interval). With different parameters, $\alpha = 0$ (no drift) and $\alpha > 0$ (drift to the right) are also possible.

For $\alpha < 0$, we accumulate intervals until the leftmost multiple of a drifts outside of the interval. For $\alpha > 0$, similarly for the rightmost multiple of a (in this case, the final considered interval will be irregular in the sense that it contains one fewer multiple of a).

In case $\alpha = 0$, the situation for each interval representative is exactly the same, and we conclude no upper bound x_h exists (which means the final x-interval over $\mathbb{Z}/m\mathbb{Z}$ will be the full domain).

We have described our method to compute the upper bound x_h. The lower bound x_l can be computed analogously. In fact, PolySAT reduces the computation of x_l to the computation of x_h by mirroring the initial configuration and the result across 0. Let f denote the procedure for calculating x_h, i.e., $x_h = f(x_0, a, l, h, m)$. Then $x_l = -f(-x_0, a, -h, -l, m)$.

Even though this method works well in practice, some limitations remain. The interval extension ends as soon as one of the red diamonds is outside the blue interval. This is by specification, but it does mean that this method is only helpful when the gap between blue intervals (i.e., $m - (h - l)$) is less than the distance between red diamonds (i.e., a).

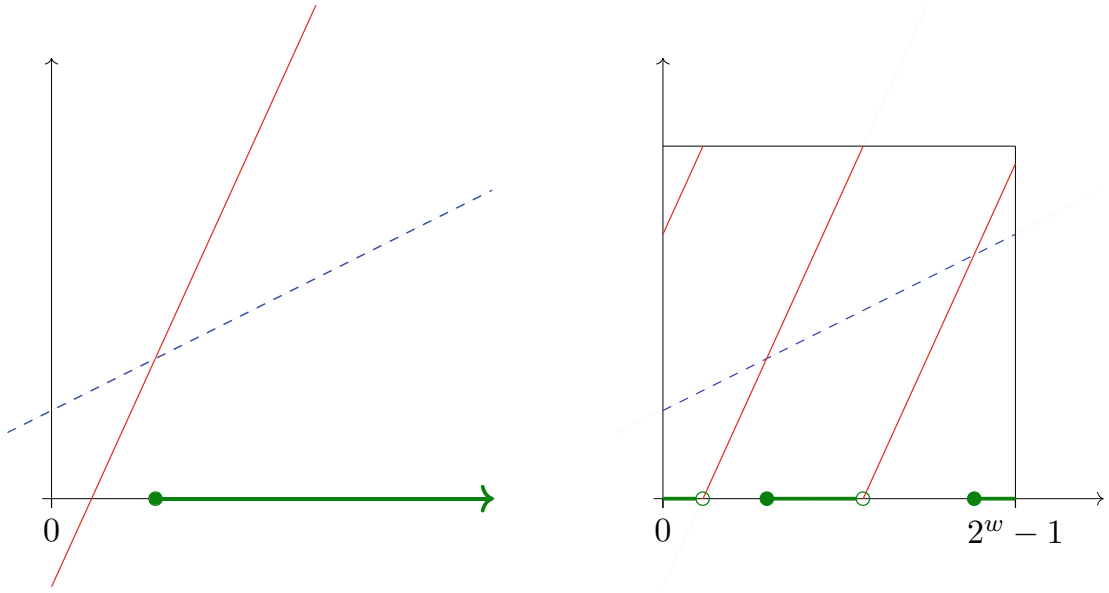
9.3 Linear Inequality with Different Coefficients

Let us now consider the remaining case of linear inequalities, i.e., an inequality c of the form $px + q \leq_\mathsf{u} rx + s$ with $\widehat{p} \neq \widehat{r}$. Again, the goal is to find the largest x-interval around a sample point x_0 where c is satisfied, however in this case, it is not possible to directly convert the inequality into an equivalent interval constraint as in the previous cases. As illustrated in Figure 9.1a with an example configuration, the corresponding problem is easily solved over infinite domains, such as the rational numbers, by computing the intersection point of the left- and right-hand side of the inequality. The interval (in that example configuration) then extends from the intersection point towards infinity.

However, in modular arithmetic, the left-hand side and the right-hand side of c do not represent continuous lines; instead, they wrap around at 2^w as seen in Figure 9.1b. The intervals extend from an intersection point to the next wraparound point. We compute and return the interval containing x_0.

We note that PolySAT computes only the intersection/wraparound points nearest to x_0. In some configurations, the gap between one interval to the next (i.e., between the bold green lines in Figure 9.1b) does not contain an integer, which means the obtained x-interval is not maximal. This method works best when the coefficients of x are near 0 or 2^w.

Computing intersection and wraparound points. In order to work out the above intuition more precisely, consider the inequality c of the form $px + q \leq_\mathsf{u} rx + s$ with $p, q, r, s \in \mathbb{Z}/2^w\mathbb{Z}$ such that $p \neq 0$, $r \neq 0$ and $p \neq r$. Let $x_0 \in \mathbb{Z}/2^w\mathbb{Z}$ be a sample value that violates c, i.e., such that $\langle px_0 + q\rangle_\mathsf{u} > \langle rx_0 + s\rangle_\mathsf{u}$.



(a) In $\mathbb{Q}$. (b) In $\mathbb{Z}/2^w\mathbb{Z}$.

Figure 9.1: Example for extracting intervals from an inequality constraint $px+q \leq_{\mathsf{u}} rx+s$ with different variable coefficients. The blue dashed line plots $\widehat{p}x + \widehat{q}$, and the red continuous line is $\widehat{r}x + \widehat{s}$. The bold green lines indicate the desired intervals.

The goal is to find a maximal x-interval around x_0 whose elements all violate the constraint c, i.e., we want to find the minimal x_l and the maximal x_h such that $x_l \leq x_0 \leq x_h$ and $\langle px + q\rangle_{\mathsf{u}} > \langle rx + s\rangle_{\mathsf{u}}$ for all $x \in [x_l; x_h]$.

In the following, we explain our method for extracting such intervals, however in general, we cannot yet guarantee to obtain a maximal interval in all cases. As illustrated in Figure 9.1, we extrapolate the left-hand side (LHS) and the right-hand side (RHS) of the constraint c using standard arithmetic until the next overflow point, and extract the maximal interval that can be obtained without overflow.

Let us define the abbreviations $a \coloneqq \langle px_0 + q\rangle_{\mathsf{u}}$ and $b \coloneqq \langle rx_0 + s\rangle_{\mathsf{u}}$. The following computations are carried out over the rational numbers $\mathbb{Q}$ by choosing the representative $\langle p\rangle_{\mathsf{u}}$ for p and analogously for the remaining parameters.

To compute a safe upper bound $x_h = x_0 + \delta_h$, we find the maximal $\delta_h \in \mathbb{Z}$ satisfying the following conditions:

- $\delta_h \geq 0$, i.e., it should be an *upper* bound,
- $\forall x \in \mathbb{Z}.(0 \leq x \leq \delta_h \rightarrow 2^w > a + \langle p\rangle_{\mathsf{u}} x > b + \langle r\rangle_{\mathsf{u}} x \geq 0)$, i.e., the LHS and RHS do not overflow within the interval and the constraint is violated for all values,
- $\langle x_0\rangle_{\mathsf{u}} + \delta_h < 2^w$, i.e., the upper bound does not overflow.

After several transformations, we finally obtain the formula

$$\delta_h = \min\left(\left\{2^w - \langle x_0\rangle_\mathsf{u}, \lceil\frac{2^w - a}{\langle p\rangle_\mathsf{u}}\rceil\right\} \cup \left\{\lceil\frac{a-b}{\langle r\rangle_\mathsf{u} - \langle p\rangle_\mathsf{u}}\rceil \;\middle|\; \langle r\rangle_\mathsf{u} > \langle p\rangle_\mathsf{u}\right\}\right) - 1.$$

Similarly, we obtain a safe lower bound $x_l = x_0 - \delta_l$, by finding the maximal $\delta_l \in \mathbb{Z}$ such that:

- $\delta_l \geq 0$ (it should be a *lower* bound),
- $\delta_l \leq \langle x_0\rangle_\mathsf{u}$ (lower bound does not overflow),
- $\forall x \in \mathbb{Z}.(0 \leq x \leq \delta_l \to 2^w > a - \langle p\rangle_\mathsf{u} x > b - \langle r\rangle_\mathsf{u} x \geq 0)$.

A sequence of similar transformations then leads us to the formula

$$\delta_l = \min\left(\left\{\langle x_0\rangle_\mathsf{u} + 1, \lceil\frac{b+1}{\langle r\rangle_\mathsf{u}}\rceil\right\} \cup \left\{\lceil\frac{a-b}{\langle p\rangle_\mathsf{u} - \langle r\rangle_\mathsf{u}}\rceil \;\middle|\; \langle p\rangle_\mathsf{u} > \langle r\rangle_\mathsf{u}\right\}\right) - 1.$$

Remark 7 (Strict Inequalities)**.** Finally, if we want to compute such bounds for a strict inequality $px + q <_\mathsf{u} rx + s$, we only have to change the strictness of one inequality in our initial conditions, i.e., replace $a \pm \langle p\rangle_\mathsf{u} x > b \pm \langle r\rangle_\mathsf{u} x$ by $a \pm \langle p\rangle_\mathsf{u} x \geq b \pm \langle r\rangle_\mathsf{u} x$. In the final formulas, this manifests as replacing $a - b$ in the numerator by $a - b + 1$; otherwise, the results are unchanged.

Remark 8 (Signed Interpretation of Coefficients)**.** For the above computations, we embedded the coefficients p, q from $\mathbb{Z}/2^w\mathbb{Z}$ into $\mathbb{Q}$ by choosing the representative $\langle p\rangle_\mathsf{u}$ in the interval $[0; 2^w[$. However, for large coefficients p or q near 2^w, we may obtain better bounds by interpreting them as negative numbers, i.e., by choosing the representative in the interval $[-2^w; 0[$ instead of the default unsigned value. To obtain the corresponding formulas for δ_l and δ_h, we can replace $a + \langle p\rangle_\mathsf{u} x$ by $a - (2^w - \langle p\rangle_\mathsf{u})x$ in the computation above. Similarly, $b + \langle r\rangle_\mathsf{u} x$ can be replaced by $b - (2^w - \langle r\rangle_\mathsf{u})x$. In total, this gives us four different ways to estimate each bound. Since each of these computations finds a safe bound, we choose the best among them.

The final formulas are listed below. The parts in red only apply if the inequality is strict and should be ignored otherwise. The four versions of δ_h are as follows:

$$\delta_{h,1} = \min\left(\left\{2^w - \langle x_0\rangle_\mathsf{u}, \lceil\frac{2^w - a}{\langle p\rangle_\mathsf{u}}\rceil\right\} \cup \left\{\lceil\frac{a-b+1}{\langle r\rangle_\mathsf{u} - \langle p\rangle_\mathsf{u}}\rceil \;\middle|\; \langle r\rangle_\mathsf{u} > \langle p\rangle_\mathsf{u}\right\}\right) - 1,$$

$$\delta_{h,2} = \min\left(\left\{2^w - \langle x_0\rangle_\mathsf{u}, \lceil\frac{2^w - a}{\langle p\rangle_\mathsf{u}}\rceil, \lceil\frac{b+1}{2^w - \langle r\rangle_\mathsf{u}}\rceil\right\}\right) - 1,$$

$$\delta_{h,3} = \min\left(\left\{2^w - \langle x_0\rangle_\mathsf{u}, \lceil\frac{a-b+1}{2^w - \langle p\rangle_\mathsf{u} + \langle r\rangle_\mathsf{u}}\rceil\right\}\right) - 1,$$

$$\delta_{h,4} = \min\left(\left\{2^w - \langle x_0\rangle_\mathsf{u}, \lceil\frac{b+1}{2^w - \langle r\rangle_\mathsf{u}}\rceil\right\} \cup \left\{\lceil\frac{a-b+1}{\langle r\rangle_\mathsf{u} - \langle p\rangle_\mathsf{u}}\rceil \;\middle|\; \langle r\rangle_\mathsf{u} > \langle p\rangle_\mathsf{u}\right\}\right) - 1.$$

The four corresponding versions of δ_l are:

$$\delta_{l,1} = \min\left(\left\{\langle x_0\rangle_\mathsf{u} + 1, \lceil\frac{b+1}{\langle r\rangle_\mathsf{u}}\rceil\right\} \cup \left\{\lceil\frac{a-b+1}{\langle p\rangle_\mathsf{u} - \langle r\rangle_\mathsf{u}}\rceil \;\middle|\; \langle p\rangle_\mathsf{u} > \langle r\rangle_\mathsf{u}\right\}\right) - 1,$$
$$\delta_{l,2} = \min\left(\left\{\langle x_0\rangle_\mathsf{u} + 1, \lceil\frac{a-b+1}{2^w - \langle r\rangle_\mathsf{u} + \langle p\rangle_\mathsf{u}}\rceil\right\}\right) - 1,$$
$$\delta_{l,3} = \min\left(\left\{\langle x_0\rangle_\mathsf{u} + 1, \lceil\frac{b+1}{\langle r\rangle_\mathsf{u}}\rceil, \lceil\frac{2^w - a}{2^w - \langle p\rangle_\mathsf{u}}\rceil\right\}\right) - 1,$$
$$\delta_{l,4} = \min\left(\left\{\langle x_0\rangle_\mathsf{u} + 1, \lceil\frac{2^w - a}{2^w - \langle p\rangle_\mathsf{u}}\rceil\right\} \cup \left\{\lceil\frac{a-b+1}{\langle p\rangle_\mathsf{u} - \langle r\rangle_\mathsf{u}}\rceil \;\middle|\; \langle p\rangle_\mathsf{u} > \langle r\rangle_\mathsf{u}\right\}\right) - 1.$$

Choosing the best of these bounds amounts to maximizing δ_h and δ_l, i.e.,

$$\delta_h = \max\{\delta_{h,1}, \delta_{h,2}, \delta_{h,3}, \delta_{h,4}\},$$
$$\delta_l = \max\{\delta_{l,1}, \delta_{l,2}, \delta_{l,3}, \delta_{l,4}\}.$$

9.4 Projecting Intervals to Sub-Slices

Since value assignments are propagated eagerly across bit-vector slices by the e-graph component of PolySAT, in some cases, a bit-vector variable is assigned to a value that contradicts an interval on a super-slice of the variable. Such contradictions may also be caused by the e-graph, because it does not take into account intervals when merging nodes.

Let $x = y +\!\!+ z$ and define $u \coloneqq |y|$ and $v \coloneqq |z|$. Given the forbidden interval $x \notin [l; h[$, then $2^v y + z \notin [l; h[$. We can learn intervals for y and z via the following PolySAT lemmas.

Lemma 7 (General Intervals)**.** In case no fixed value is known for the other sub-slice, it is possible to learn an interval as long as $[l; h[$ is big enough.

$$\text{len}([l; h[) \geq 2^u \implies y \notin [l_y; h_y[\tag{9.1}$$
$$\text{len}([l; h[) > 2^{u+v} - 2^v \implies z \notin [l_z; h_z[\tag{9.2}$$

where $l_y \coloneqq \lceil\frac{l}{2^v}\rceil \bmod 2^v$, $h_y \coloneqq \lfloor\frac{h}{2^v}\rfloor$, $l_z \coloneqq l \bmod 2^v$, and $h_z \coloneqq h \bmod 2^v$.

Lemma 8 (Specific Intervals)**.** If the other sub-slice has a fixed value, a larger interval can be projected [GJD20, Figure 1].

$$z = n \wedge l_y \neq h_y \implies y \notin [l_y, h_y[\tag{9.3}$$
$$z = n \wedge l_y = h_y \wedge h_y 2^v + n \in [l; h[\implies \bot \tag{9.4}$$
$$y = n \wedge l_z \neq h_z \implies z \notin [l_z; h_z[\tag{9.5}$$
$$y = n \wedge l_z = h_z \wedge n2^v \in [l; h[\implies \bot \tag{9.6}$$

where we define the following for $\beta \in \{l, h\}$:

$$\beta_y := \left\lceil \frac{(\beta - n) \bmod 2^{u+v}}{2^v} \right\rceil \bmod 2^u,$$
$$\beta_z := \begin{cases} \beta \bmod 2^v & \text{if } \lfloor \frac{\beta}{2^v} \rfloor = n, \\ 0 & \text{otherwise.} \end{cases}$$

These projections are applied iteratively in PolySAT to derive intervals for arbitrary sub-slices. At each step, a choice is made between Lemmas 7–8, depending on whether a fixed value is available at the required decision level.

Example 20. We can use the above to find an interval I such that

$$x \simeq 0 +\!\!+ y +\!\!+ z \wedge z[15{:}8] \simeq 123 \wedge x \notin [300007; 0[\implies y \notin I,$$

where $|x| = 64$ and $|y| = |z| = 16$.

- First, apply (9.5) to obtain $y +\!\!+ z \notin [300007; 0[$.
- Next, with (9.1) we obtain $y +\!\!+ z[15{:}8] \notin [1253; 0[$.
- Finally, with (9.3) we obtain $y \notin [5; 0[$.

CHAPTER 10

Non-Linear Conflicts

Non-linear conflicts are handled in PolySAT by saturation, incremental linearization, and bit-blasting. Saturation, incremental linearization and bit-blasting are postponed until all variables are assigned values and there are no conflicts detected by propagating bounds on linear constraints.

10.1 Saturation Lemmas

Saturation lemmas propagate consequences from non-linear constraints. The consequences are considered "simpler", when they are linear or if they contain fewer variables. Saturation lemmas, given in Lemmas 9–12, are added by PolySAT if their non-linear constraints are in the assertion trail Γ and they evaluate to false under the current assignment in Γ.

Lemma 9 (Saturation Modulo Multiplication Inequalities)**.** We list saturation rules over inequalities that are applied in PolySAT.

$$
\begin{array}{lcl}
px <_{\mathsf{u}} qx & \Longrightarrow & p \not\simeq q \\
px <_{\mathsf{u}} qx & \Longrightarrow & \Omega^*(p,x) \lor p <_{\mathsf{u}} q \\
px <_{\mathsf{u}} qx & \Longrightarrow & \Omega^*(-q,x) \lor p <_{\mathsf{u}} q \\
px <_{\mathsf{u}} qx & \Longrightarrow & \Omega^*(q,-x) \lor p >_{\mathsf{u}} q \lor p \simeq 0 \\
px <_{\mathsf{u}} qx & \Longrightarrow & \Omega^*(-p,-x) \lor p >_{\mathsf{u}} q \lor p \simeq 0 \\
px \leq_{\mathsf{u}} qx & \Longrightarrow & \Omega^*(p,x) \lor p \leq_{\mathsf{u}} q \lor x \simeq 0 \\
px \leq_{\mathsf{u}} qx & \Longrightarrow & \Omega^*(-q,x) \lor p \leq_{\mathsf{u}} q \lor x \simeq 0 \lor q \simeq 0 \\
px \leq_{\mathsf{u}} qx & \Longrightarrow & \Omega^*(q,-x) \lor p \geq_{\mathsf{u}} q \lor x \simeq 0 \lor p \simeq 0 \\
px \leq_{\mathsf{u}} qx & \Longrightarrow & \Omega^*(-p,-x) \lor p \geq_{\mathsf{u}} q \lor x \simeq 0 \lor p \simeq 0
\end{array}
$$

$$
\begin{array}{lcllllll}
px + s \leq_{\mathsf{u}} q & \Longrightarrow & \Omega^*(p,x) & \vee & \Omega^+(px,s) & \vee & pr \leq_{\mathsf{u}} q & \vee \quad x <_{\mathsf{u}} r \\
p \leq_{\mathsf{u}} x \wedge qx \leq_{\mathsf{u}} r & \Longrightarrow & \Omega^*(q,x) & \vee & pq \leq_{\mathsf{u}} r & & & \\
p \leq_{\mathsf{u}} x \wedge qx <_{\mathsf{u}} r & \Longrightarrow & \Omega^*(q,x) & \vee & pq <_{\mathsf{u}} r & & & \\
p <_{\mathsf{u}} x \wedge qx \leq_{\mathsf{u}} r & \Longrightarrow & \Omega^*(q,x) & \vee & pq <_{\mathsf{u}} r & \vee & q \simeq 0 & \\
p <_{\mathsf{u}} x \wedge qx \leq_{\mathsf{u}} r & \Longrightarrow & \Omega^*(q,x) & \vee & pq <_{\mathsf{u}} r & \vee & r \simeq 0 & \\
p \leq_{\mathsf{u}} qx \wedge x \leq_{\mathsf{u}} r & \Longrightarrow & \Omega^*(q,r) & \vee & p \leq_{\mathsf{u}} qr & & & \\
p <_{\mathsf{u}} qx \wedge x \leq_{\mathsf{u}} r & \Longrightarrow & \Omega^*(q,r) & \vee & p <_{\mathsf{u}} qr & & & \\
p \leq_{\mathsf{u}} qx \wedge x <_{\mathsf{u}} r & \Longrightarrow & \Omega^*(q,r) & \vee & p <_{\mathsf{u}} qr & \vee & p \simeq 0 & \\
p \leq_{\mathsf{u}} qx \wedge x <_{\mathsf{u}} r & \Longrightarrow & \Omega^*(q,r) & \vee & p <_{\mathsf{u}} qr & \vee & q \simeq 0 &
\end{array}
$$

Note that these rules do not require $x \notin p, q, r, s$, so they can be applied even when the degree of x is larger than 1.

Next, we can connect overflow constraints with multiplications or decompose them to linear inequalities.

Lemma 10 (Overflow Saturation)**.**

$$
\begin{array}{lcl}
\neg\Omega^*(p,q) \wedge q \not\simeq 0 & \Longrightarrow & p \leq_{\mathsf{u}} p \cdot q \\
\bar{0}p \cdot \bar{0}q \geq_{\mathsf{u}} 2^w & \Longrightarrow & \Omega^*(p,q) \\
\Omega^*(p,q) \wedge \neg\Omega^*(r,s) & \Longrightarrow & p >_{\mathsf{u}} r \vee q >_{\mathsf{u}} s \\
\Omega^*(p,q) \wedge p \geq_{\mathsf{u}} q & \Longrightarrow & p \geq_{\mathsf{u}} \lceil\sqrt{2^w}\rceil \\
\neg\Omega^*(p,q) \wedge p \geq_{\mathsf{u}} q & \Longrightarrow & q <_{\mathsf{u}} \lfloor\sqrt{2^w}\rfloor
\end{array}
$$

where $\bar{0}p$ and $\bar{0}q$ stands for a zero-extension with at least one bit of p and q, respectively. Note that here $w = |p| = |q| > 1$, since multiplication overflow is impossible for $w = 1$.

Variables can in some cases be resolved, producing constraints that are free of resolved variables.

Lemma 11 (Saturation Modulo Equalities)**.**

$$
\begin{array}{lcll}
ax + b \simeq 0 \wedge cx + d \simeq 0 & \Longrightarrow & ad - bc \simeq 0 & \\
ax + b \simeq 0 \wedge c[x] & \Longrightarrow & c[-b \cdot a^{-1}] & \text{if } a \text{ is odd}
\end{array}
$$

where $c[x]$ may be any constraint containing x. Note that the multiplicative inverse a^{-1} of a in $\mathbb{Z}/2^w\mathbb{Z}$ exists if and only if a is odd.

Definition 14 (Parity)**.** The *parity* of a bit-vector x is the largest integer $i \in \{0, \ldots, w\}$ such that 2^i divides x. We write $\text{parity}(x)$ for the parity of x.

A bit-vector has parity 0 if and only if it is odd. The parity of a bit-vector can be constrained by a linear inequality, where

$$\text{parity}(p) \geq i \iff p2^{w-i} \simeq 0$$

for $0 < i \leq w$.

Lemma 12 (Parity Saturation)**.** Parity inequalities can be used to constrain values of multipliers.

$$\begin{aligned} p \cdot q \simeq 0 &\implies \operatorname{parity}(p) + \operatorname{parity}(q) \geq w \\ p \cdot q \simeq 1 &\implies \operatorname{parity}(p) = 0 \\ p \cdot q \simeq q &\implies \operatorname{parity}(p-1) + \operatorname{parity}(q) \geq w \\ \operatorname{parity}(p \cdot q) &= \min(w, \operatorname{parity}(p) + \operatorname{parity}(q)) \end{aligned}$$

Obtaining Saturation Lemmas. Since bit-vector arithmetic does not match the intuition of standard arithmetic, it is tedious and error-prone to come up with saturation lemmas manually. We have therefore employed some automation to discover the rules given in Lemma 9. We start with the constraint on the left-hand side of the rule that triggers saturation (e.g., $px <_\mathsf{u} qx$) and generate a set of constraints that we want to allow in the right-hand side of the rule. We then add the constraints for a small fixed bit-width to Z3 and apply the MARCO algorithm [LPMM16] to find the minimal unsatisfiable subsets (MUS). Each MUS corresponds to a valid lemma; however, to be useful as saturation lemmas, we filter the candidates such that the right-hand side is simpler in some sense. Finally, we verify manually that the lemmas generalize to arbitrary bit-widths.

10.2 Incremental Linearization

PolySAT includes incremental linearization rules for the cases where variables are 0, 1, -1, or powers of two. Note that our vocabulary of incremental linearization lemmas is considerably smaller than what is used for non-linear integer arithmetic [CGI+18], but it is also materially different as it operates over modular semantics of bit-vector operations. Notably, we do not include here inferences for deriving ordering constraints, such as $a > b \wedge c > 0 \implies ac > bc$, which holds for integers, but not for bit-vectors. Note that Lemma 9 includes ordering constraints, but only for the cases where relevant uses of multiplication do not overflow.

Lemma 13 (Incremental Linearization)**.**

$$\begin{aligned} p \simeq 0 &\implies p \cdot q \simeq 0 \\ p \simeq 1 &\implies p \cdot q \simeq q \\ p \simeq -1 &\implies p \cdot q \simeq -q \\ p \simeq 2^k &\implies p \cdot q \simeq 2^k q \quad (k = 1, \ldots, w-1) \\ p \cdot q \simeq 1 &\implies p \simeq 1 \vee \Omega^*(p, q) \\ p \cdot q \simeq q &\implies p \simeq 1 \vee q \simeq 0 \vee \Omega^*(p, q) \end{aligned}$$

10.3 Bit-Blasting Rules

As a final resort, PolySAT admits bit-blasting. A product $x := p \cdot q$ can be equivalently represented as $\sum_i 2^i p[i]q$.

The other primitive operations (bit-wise *and*, bit-wise *or*, left shift, logical and arithmetic right shift) are unfolded using blasting as follows.

Lemma 14 ($x \coloneqq p \,\&\, q$)**.** Bit-wise *and* ("&") is handled using standard axioms that fall back to bit-blasting at each index i if the basic algebraic properties hold, but x still does not evaluate to the bit-wise *and* of p, q.

$$\begin{array}{lcll} \top & \Longrightarrow & x \leq_{\mathsf{u}} p & \\ p \simeq 0 & \Longrightarrow & x \simeq 0 & \\ p \simeq -1 & \Longrightarrow & x \simeq q & \\ p \simeq q & \Longrightarrow & x \simeq p & \\ p[i] \wedge q[i] & \Longrightarrow & x[i] & \text{for each } 0 \leq i < w \\ x[i] & \Longrightarrow & p[i] & \text{for each } 0 \leq i < w \end{array}$$

Note that we do not list symmetric rules such as $q \simeq 0 \implies x \simeq 0$.

Lemma 15 ($x \coloneqq p \mid q$)**.** Bit-wise *or* ("|") is handled similarly as bit-wise *and*.

$$\begin{array}{lcll} \top & \Longrightarrow & x \geq_{\mathsf{u}} p & \\ p \simeq 0 & \Longrightarrow & x \simeq q & \\ p \simeq -1 & \Longrightarrow & x \simeq -1 & \\ p \simeq q & \Longrightarrow & x \simeq p & \\ p[i] & \Longrightarrow & x[i] & \text{for each } 0 \leq i < w \\ x[i] & \Longrightarrow & p[i] \vee q[i] & \text{for each } 0 \leq i < w \end{array}$$

Lemma 16 ($x \coloneqq p \ll q$)**.** For shift operations, we split on the second argument.

$$\begin{array}{lcl} q \geq_{\mathsf{u}} w & \Longrightarrow & x \simeq 0 \\ q \simeq 0 & \Longrightarrow & x \simeq p \\ q \simeq i & \Longrightarrow & x \simeq 2^i p \end{array}$$

for all constants i such that $0 < i < w$.

Lemma 17 ($x \coloneqq p \gg q$)**.** Logical right-shift is analogous.

$$\begin{array}{lcl} q \geq_{\mathsf{u}} w & \Longrightarrow & x \simeq 0 \\ q \simeq 0 & \Longrightarrow & x \simeq p \\ q \simeq i & \Longrightarrow & 2^i x \leq_{\mathsf{u}} p \leq_{\mathsf{u}} 2^i x + 2^i - 1 \wedge x <_{\mathsf{u}} 2^{w-i} \end{array}$$

for all constants i such that $0 < i < w$.

Lemma 18 ($x \coloneqq p \gg_{\mathsf{a}} q$)**.** The arithmetic right-shift must take the sign bit $p[w-1]$ into account.

$$\begin{array}{lcl} p[w-1] \wedge q \geq_{\mathsf{u}} w & \Longrightarrow & x \simeq -1 \\ \neg p[w-1] \wedge q \geq_{\mathsf{u}} w & \Longrightarrow & x \simeq 0 \\ q \geq_{\mathsf{u}} w & \Longrightarrow & x + 1 \leq_{\mathsf{u}} 1 \\ q \simeq 0 & \Longrightarrow & x \simeq p \\ q \simeq i & \Longrightarrow & 2^i x \leq_{\mathsf{u}} p \leq_{\mathsf{u}} 2^i x + 2^i - 1 \\ p[w-1] \wedge q \simeq i & \Longrightarrow & x \geq_{\mathsf{u}} 2^w - 2^{w-i-1} \\ \neg p[w-1] \wedge q \simeq i & \Longrightarrow & x <_{\mathsf{u}} 2^{w-i-1} \end{array}$$

for all constants i such that $0 < i < w$.

PolySAT also performs partial bit-blasting for multiplication overflow predicates. It is based on partitioning the conditions for overflow by using the sum of most significant bits into three cases. To describe these, first let us define the shorthand $\mathrm{msb}(p)$ for the one-based index of the most significant bit of p. For example, $\mathrm{msb}(1) = 1, \mathrm{msb}(2) = 2$. It can be defined indirectly using the equivalence $\mathrm{msb}(p) \geq i \iff p \geq_\mathsf{u} 2^{i-1}$ for $1 \leq i \leq w$. The cases are

$$\begin{aligned} \mathrm{msb}(p) + \mathrm{msb}(q) \geq w + 2 &\implies \Omega^*(p,q) \\ \mathrm{msb}(p) + \mathrm{msb}(q) \leq w &\implies \neg\Omega^*(p,q) \\ \mathrm{msb}(p) + \mathrm{msb}(q) = w + 1 &\implies \\ (\Omega^*(p,q) &\iff (0p) \cdot (0q) \geq_\mathsf{u} 2^w), \end{aligned}$$

where $0p$ and $0q$ stand for the zero-extension by a single bit of p and q, respectively. In other words, when the most significant bits add up to w, multiplication overflow affects exactly one additional bit, so it suffices to extend p and q by a single bit to determine overflow.

CHAPTER 11

Experiments

We evaluated our PolySAT prototype[1] against recent versions of several state-of-the-art SMT solvers on the following four benchmark sets: the category `QF_BV` from SMT-LIB [BFT16] (release 2023, non-incremental); the BV2SMV benchmarks featuring large bit-widths [FKB13]; 14 benchmarks from smart contract verification related to the Certora prover [AGR+20]; and a set of benchmarks from the Alive2 compiler verification project [LLH+21]. Note that the solver STP [GD07] does not support the logic `QF_UFBV` used by some of the Certora benchmarks.

Our experiments were performed on a cluster at TU Wien, where each compute node contains two AMD Epyc 7502 processors, each of which has 32 CPU cores running at 2.5 GHz. Each compute node is equipped with 1008 GiB of physical memory that is split into eight memory nodes of 126 GiB each, with eight logical CPUs assigned to each node. We used `runexec` from the benchmarking framework BenchExec [BLW17] to assign each benchmark process to a different CPU core and its corresponding memory node. Further, we used GNU Parallel [Tan24] to schedule benchmark processes in parallel.

Our results are summarized in Table 11.1 and indicate that PolySAT is comparable to the other word-level approaches on the BV2SMV benchmark set, however in general, more work is needed. Concerning the Alive2 benchmarks that were solved by Yices2-mcsat but not by PolySAT, we found that in all but three cases Yices2-mcsat did not use any interval reasoning for conflicts/propagation; rather, Yices2-mcsat relied mostly on a fallback to bit-blasting. As PolySAT does not yet have such a fallback, this result suggests our bit-blasting rules (Section 10.3) alone are not enough.

Nevertheless, PolySAT complements Z3 with word-level bit-vector reasoning. Our experimental analysis found that PolySAT solved 135 problems that Z3 did not solve

[1] Available at https://github.com/Z3Prover/z3/tree/poly. This chapter refers to commit `16fb86b636047fd79ad5827f768b6f26d8812948`. To select PolySAT for bit-vector solving, add the following options: `sat.smt=true tactic.default_tactic=smt smt.bv.solver=1`.

		SMT-LIB		BV2SMV		Smart Contracts		Alive2	
		sat	unsat	sat	unsat	sat	unsat	sat	unsat
Bit-blasting	Bitwuzla [NP23]	17 745	27 203	32	115	1	3	39	3 954
	cvc5 [BBB+22]	16 417	25 922	31	114	0	4	39	2 722
	STP [GD07]	17 462	27 011	24	115	-	-	39	2 893
	Yices2 [Dut14]	17 589	26 600	24	107	0	3	39	1 519
	Z3 [dMB08]	16 112	25 597	29	94	0	3	39	1 514
Word-lvl	cvc5-IntBlast [ZIM+22]	11 251	24 376	32	64	1	9	5	1 047
	Yices2-mcsat [GJD20]	14 155	22 396	24	101	1	4	23	2 562
	Z3-IntBlast	10 912	24 371	28	56	1	5	30	921
	Z3-PolySAT	7 297	20 080	28	63	0	3	0	21
	Total	46 191		192		14		12 951	

Table 11.1: Number of problems solved within 60 s for several benchmark sets. The upper five solvers are based on bit-blasting, while the lower four solvers use word-level techniques.

and 404 problems that Z3-IntBlast did not solve (40 of which neither Z3 nor Z3-IntBlast solved).

CHAPTER 12

Related Work

Bit-vectors faithfully model the semantics of fixed-width integer types of common programming languages and computer architectures. As such, the bit-vector logic is a popular target for many tasks in formal verification applications, such as efficient bounded model checking [CKL04], bit-precise memory handling [LLH+21], or proving the safety of decentralized financial transactions [AGR+20].

The state of the art of solving bit-vector formulas in SMT solving is *bit-blasting* [KS08], i.e., translating bit-vector formulas into propositional formulas that can then be solved by ordinary SAT solvers. Practically all state-of-the-art SMT solvers supporting the bit-vector theory contain a bit-blasting solver [GD07, BKO+07, dMB08, CGSS13, Dut14, NPWB18, BBB+22, NP23]. While the core idea of translating bit-vector operations to SAT formulas is quite natural, the different solvers vary considerably in the specifics of this translation. and related techniques such as pre-processing and using over- and under-approximations to simplify solving.

While bit-blasting is effective for many practical problems, it generally scales poorly to large bit-widths, especially when multiplications are involved. In attempts to overcome this issue, several different strategies have been investigated previously.

Layered techniques [BCF+07, HBJ+14] first apply several layers of cheap but incomplete word-level procedures and then fall back to lazy bit-blasting; this way, only the relevant parts of the input where none of the incomplete procedures apply needs to be bit-blasted.

Recently, an abstraction-refinement loop on top of a bit-blasting solver [NPZ24] has been developed. The bit-vector multiplication, division, and remainder operations, often the cause of poor scaling of bit-blasting, are abstracted as uninterpreted functions. The abstracted operations are then refined incrementally by adding lemmas to the solver (somewhat similar as done in Chapter 10), falling back to standard bit-blasting when necessary.

In contrast, purely word-level techniques have been developed as well. In a method called *Int-Blasting* [ZIM+22], bit-vector constraints are translated into non-linear integer arithmetic, where the semantics of bit-vector operations are captured by range constraints ($0 \leq x < 2^w$) and inserting *modulo* operations where needed.

The model-constructing satisfiability calculus (MCSAT) [dMJ13] has been proposed as an alternative to CDCL(T). In MCSAT, Boolean and theory decisions are interleaved on a shared trail. Propagation and conflict explanation is handled by theory-specific plugins. Bit-vector plugins for MCSAT [ZWR16, GJD20] generate word-level explanations for supported fragments of the bit-vector language and fall back to bit-level explanations otherwise. Quantifier elimination algorithms [JC16] have been developed for certain fragments of bit-vector logic, which may serve as a basis of conflict explanation. An earlier version of PolySAT was internally based on MCSAT, while still being integrated as a theory solver into the CDCL(T)-based solver Z3. However, handling full Boolean clauses along with bit-vector slicing and translation of constraints turned out to be too cumbersome and error-prone to maintain during ongoing development.

The solver Wombit [WSS19] uses a hybrid approach: it employs word-level propagation during search, but then generates justifications based on individual bits during conflict resolution.

Stochastic local search Z3 [FBWH15, NPB17, NP20] has been developed to quickly find models for satisfiable instances, but in general, does not terminate for unsatisfiable problems.

CHAPTER 13

Summary and Outlook

In the first part of the thesis, we presented improvements to subsumption and related inferences within first-order theorem proving.

First, we introduced the simplifying inference rule subsumption demodulation to improve support for reasoning with conditional equalities in superposition-based first-order theorem proving. Subsumption demodulation builds on existing machineries of superposition provers and can therefore be efficiently integrated in superposition reasoning. Still, the rule remains expensive and does not pay off for all problems, leading to a decrease in total number of solved problems by our implementation in Vampire. However, this is justified because subsumption demodulation also solves many new examples that existing tools, including first-order provers and SMT solvers, cannot handle.

Future work includes the design of more sophisticated approaches for selecting rewriting equalities and improving the imperfect filtering of clauses indexes.

Next, we advocated the use of dedicated lightweight SAT solving to solve clauses subsumption checks in first-order theorem proving. We introduced substitution constraints to encode subsumption as a SAT instance. For solving such instances, we adjust unit propagation and conflict resolution in SAT solving towards a tailored treatment of substitution constraints. Crucially, our encoding together with our tailored solver enables efficient setup of subsumption instances. Our experimental results indicate that SAT-based subsumption significantly improves the performance of first-order proving.

The results on SAT-based subsumption have been extended successfully to subsumption resolution [CKRR23, CRR+24]. Extending this work towards equality reasoning, and hence also addressing subsumption demodulation is an interesting task for future work. For doing so, we believe our substitution constraints would need to encode matching also on the term level, and thus not only on the literal level, in order to find suitable terms to rewrite.

In the second part of the thesis, we attempted to develop a new approach to word-level bit-vector solving in the context of SMT solving.

We introduced PolySAT, a general purpose word-level bit-vector solver, to overcome the scalability issue of bit-blasting over large bit-vectors. PolySAT integrates into CDCL(T)-based SMT solving, generalizes interval-based reasoning, and performs saturation and incremental linearization of constraints. As a minor connection to the first part of the thesis, we apply subsumption resolution with certain theory lemmas to simplify conflict clauses that arise from interval conflicts.

PolySAT is implemented in the SMT solver Z3. While the experimental results thus far have been disappointing, PolySAT does complement bit-vector reasoning in Z3.

However, future work remains plentiful. First of all, the saturation lemmas to handle non-linear conflicts are incomplete. Further study is required to determine the class of problems covered by our saturation lemmas.

When discovering saturation rules automatically, we checked candidate rules for low bit-widths (usually all w such that $1 \leq w \leq 16$). Usually, one would expect lemmas that pass checks for all low bit-widths to generalize to arbitrary bit-widths, but this step involved a manual check. It would be an interesting direction to explore under what conditions this generalization is guaranteed, or alternatively, replace the fully manual check by setting up a suitable proof environment in an interactive theorem prover.

Further combinations of complementary approaches of word-level reasoning with bit-blasting is another promising direction to explore. For instance, Z3 supports stochastic local search for bit-vectors. This component could be valuable for getting candidate variable values when making guesses in PolySAT.

Finally, limitations remain for the extraction of intervals from inequalities (Section 9.2). We noticed periodic behaviour also in the cases that are not well-supported in our current approach, but have so far not succeeded to fully work out the necessary results. It is also unclear how to perform efficient intersections of periodically repeating intervals.

Übersicht verwendeter Hilfsmittel

The German version of the abstract is based on an automated translation provided by the free version of DeepL[1], which required heavy editing. Otherwise, no generative AI tools have been used in the preparation of this document.

[1] https://www.deepl.com

List of Figures

List of Tables

List of Algorithms

Bibliography

[ABH+20] Sepideh Asadi, Martin Blicha, Antti E. J. Hyvärinen, Grigory Fedyukovich, and Natasha Sharygina. Incremental Verification by SMT-based Summary Repair. In *Proceedings of FMCAD*, pages 77–82, 2020. doi:10.34727/2020/isbn.978-3-85448-042-6_14.

[AGR+20] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. Taming Callbacks for Smart Contract Modularity. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):209:1–209:30, 2020. doi:10.1145/3428277.

[BBB+22] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Proceedings of TACAS*, pages 415–442, 2022. doi:10.1007/978-3-030-99524-9_24.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of CAV*, pages 171–177, 2011. doi:10.1007/978-3-642-22110-1_14.

[BCF+07] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A Lazy and Layered SMT($\mathcal{BV}$) Solver for Hard Industrial Verification Problems. In *Proceedings of CAV*, pages 547–560, 2007. doi:10.1007/978-3-540-73368-3_54.

[BEG+19] Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovács, and Matteo Maffei. Verifying Relational Properties using Trace Logic. In *Proceedings of FMCAD*, pages 170–178, 2019. doi:10.23919/FMCAD.2019.8894277.

[BF15] Armin Biere and Andreas Fröhlich. Evaluating CDCL Variable Scoring Schemes. In *Proceedings of SAT*, pages 405–422, 2015. doi:10.1007/978-3-319-24318-4_29.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). https://www.smt-lib.org, 2016.

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994. doi:10.1093/logcom/4.3.217.

[BG01] Leo Bachmair and Harald Ganzinger. Resolution Theorem Proving. In *Handbook of Automated Reasoning*, pages 19–99. Elsevier and MIT Press, 2001. doi:10.1016/B978-044450813-3/50004-7.

[BGMR15] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, pages 24–51, 2015. doi:10.1007/978-3-319-23534-9_2.

[Bie08] Armin Biere. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97, 2008. doi:10.3233/SAT190039.

[Bie19] Armin Biere. CaDiCaL at the SAT Race 2019. In *Proceedings of SAT Race 2019: Solver and Benchmark Descriptions*, pages 8–9, 2019. URL: https://hdl.handle.net/10138/306988.

[BKO+07] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady. Deciding Bit-Vector Arithmetic with Abstraction. In *Proceedings of TACAS*, pages 358–372, 2007. doi:10.1007/978-3-540-71209-1_28.

[BLW17] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable Benchmarking: Requirements and Solutions. *Journal on Software Tools for Technology Transfer*, 21:1–29, 2017. doi:10.1007/s10009-017-0469-y.

[BP98] Nikolaj S. Bjørner and Mark C. Pichora. Deciding Fixed and Non-fixed Size Bit-vectors. In *Proceedings of TACAS*, pages 376–392, 1998. doi:10.1007/BFb0054184.

[BS97] Roberto J. Bayardo, Jr. and Robert Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proceedings of AAAI and IAAI*, pages 203–208, 1997. URL: https://www.aaai.org/Library/AAAI/1997/aaai97-032.php.

[BS09] Roberto Bruttomesso and Natasha Sharygina. A Scalable Decision Procedure for Fixed-Width Bit-Vectors. In *Proceedings of ICCAD*, pages 13–20, 2009. doi:10.1145/1687399.1687403.

[Buc06] Bruno Buchberger. Bruno Buchberger's PhD thesis 1965: An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal. *Journal of Symbolic Computation*, 41(3-4):475–511, 2006. doi:10.1016/j.jsc.2005.09.007.

[CGI+18] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Experimenting on Solving Nonlinear Integer Arithmetic with Incremental Linearization. In *Proceedings of SAT*, pages 383–398, 2018. doi:10.1007/978-3-319-94144-8_23.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *Proceedings of TACAS*, pages 93–107, 2013. doi:10.1007/978-3-642-36742-7_7.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *Proceedings of TACAS*, pages 168–176, 2004. doi:10.1007/978-3-540-24730-2_15.

[CKRR23] Robin Coutelier, Laura Kovács, Michael Rawson, and Jakob Rath. SAT-Based Subsumption Resolution. In *Proceedings of CADE*, pages 190–206, 2023. doi:10.1007/978-3-031-38499-8_11.

[CMP20] Martin Clochard, Claude Marché, and Andrei Paskevich. Deductive Verification with Ghost Monitors. *Proceedings of the ACM on Programming Languages*, 4(POPL):2:1–2:26, 2020. doi:10.1145/3371070.

[CRR+24] Robin Coutelier, Jakob Rath, Michael Rawson, Armin Biere, and Laura Kovács. SAT Solving for Variants of First-Order Subsumption. *Formal Methods in System Design*, 2024. doi:10.1007/s10703-024-00454-1.

[Cru17] Simon Cruanes. Superposition with Structural Induction. In *Proceedings of FroCoS*, pages 172–188, 2017. doi:10.1007/978-3-319-66167-4_10.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of TACAS*, pages 337–340, 2008. doi:10.1007/978-3-540-78800-3_24.

[dMJ13] Leonardo de Moura and Dejan Jovanovic. A Model-Constructing Satisfiability Calculus. In *Proceedings of VMCAI*, pages 1–12, 2013. doi:10.1007/978-3-642-35873-9_1.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a Theorem Prover for Program Checking. *Journal of the ACM*, 52(3):365–473, 2005. doi:10.1145/1066100.1066102.

[Dut14] Bruno Dutertre. Yices 2.2. In *Proceedings of CAV*, pages 737–744, 2014. doi:10.1007/978-3-319-08867-9_49.

[FBWH15] Andreas Fröhlich, Armin Biere, Christoph M. Wintersteiger, and Youssef Hamadi. Stochastic Local Search for Satisfiability Modulo Theories. In *Proceedings of AAAI*, pages 1136–1143, 2015. doi:10.1609/AAAI.V29I1.9372.

[FG10] Alan M. Frisch and Paul A. Giannaros. SAT Encodings of the At-Most-k Constraint. In *Workshop on Constraint Modelling and Reformulation*, 2010. URL: https://www2.it.uu.se/research/group/astra/ModRef10/papers/Alan%20M.%20Frisch%20and%20Paul%20A.%20Giannoros.%20SAT%20Encodings%20of%20the%20At-Most-k%20Constraint%20-%20ModRef%202010.pdf.

[FKB13] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. Efficiently Solving Bit-Vector Problems Using Model Checkers. In *Proceedings of Workshop on SMT*, pages 6–15, 2013. URL: https://fmv.jku.at/bv2smv/.

[GD07] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Proceedings of CAV*, pages 519–531, 2007. doi:10.1007/978-3-540-73368-3_52.

[GGK20] Pamina Georgiou, Bernhard Gleiss, and Laura Kovács. Trace Logic for Inductive Loop Reasoning. In *Proceedings of FMCAD*, pages 255–263, 2020. doi:10.34727/2020/isbn.978-3-85448-042-6_33.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast Decision Procedures. In *Proceedings of CAV*, pages 175–188, 2004. doi:10.1007/978-3-540-27813-9_14.

[GJD20] Stéphane Graham-Lengrand, Dejan Jovanovic, and Bruno Dutertre. Solving Bitvectors with MCSAT: Explanations from Bits and Pieces. In *Proceedings of IJCAR*, pages 103–121, 2020. doi:10.1007/978-3-030-51074-9_7.

[GKKS09] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. On the Implementation of Weight Constraint Rules in Conflict-Driven ASP Solvers. In *Proceedings of ICLP*, pages 250–264, 2009. doi:10.1007/978-3-642-02846-5_23.

[GKR20] Bernhard Gleiss, Laura Kovács, and Jakob Rath. Subsumption Demodulation in First-Order Theorem Proving. In *Proceedings of IJCAR*, pages 297–315, 2020. doi:10.1007/978-3-030-51074-9_17.

[GL85a] Georg Gottlob and Alexander Leitsch. Fast Subsumption Algorithms. In *Proceedings of EUROCAL*, pages 64–77, 1985. doi:10.1007/3-540-15984-3_239.

[GL85b] Georg Gottlob and Alexander Leitsch. On the Efficiency of Subsumption Algorithms. *Journal of the ACM*, 32(2):280–295, 1985. doi:10.1145/3149.214118.

[HBJ+14] Liana Hadarean, Kshitij Bansal, Dejan Jovanovic, Clark Barrett, and Cesare Tinelli. A Tale of Two Solvers: Eager and Lazy Approaches to Bit-Vectors. In *Proceedings of CAV*, pages 680–695, 2014. doi:10.1007/978-3-319-08867-9_45.

[HKR21] Petra Hozzová, Laura Kovács, and Jakob Rath. Automated Generation of Exam Sheets for Automated Deduction. In *Proceedings of CICM*, pages 185–196, 2021. doi:10.1007/978-3-030-81097-9_15.

[HPWW13] Thomas Hillenbrand, Ruzica Piskac, Uwe Waldmann, and Christoph Weidenbach. From Search to Computation: Redundancy Criteria and Simplification at Work. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 169–193, 2013. doi:10.1007/978-3-642-37651-1_7.

[JC16] Ajith K. John and Supratik Chakraborty. A layered algorithm for quantifier elimination from linear modular constraints. *Formal Methods in System Design*, 49(3):272–323, 2016. doi:10.1007/s10703-016-0260-9.

[KFB16] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of Fixed-Size Bit-Vector Logics. *Theory of Computing Systems*, 59(2):323–376, 2016. doi:10.1007/s00224-015-9653-1.

[KGC16] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-Based Model Checking for Recursive Programs. *Formal Methods in System Design*, 48(3):175–205, 2016. doi:10.1007/s10703-016-0249-4.

[KN86] Deepak Kapur and Paliath Narendran. NP-Completeness of the Set Unification and Matching Problems. In *Proceedings of CADE*, pages 489–495, 1986. doi:10.1007/3-540-16780-3_113.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Springer, 2008. doi:10.1007/978-3-540-74105-3.

[KV13] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *Proceedings of CAV*, pages 1–35, 2013. doi:10.1007/978-3-642-39799-8_1.

[Lei17] K. Rustan M. Leino. Accessible Software Verification with Dafny. *IEEE Software*, 34(6):94–97, 2017. doi:10.1109/MS.2017.4121212.

[LLH+21] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of PLDI*, pages 65–79, 2021. doi:10.1145/3453483.3454030.

[LM12] Mark H. Liffiton and Jordyn C. Maglalang. A Cardinality Solver: More Expressive Constraints for Free. In *Proceedings of SAT*, pages 485–486, 2012. doi:10.1007/978-3-642-31612-8_47.

[LPMM16] Mark H. Liffiton, Alessandro Previti, Ammar Malik, and Joao Marques-Silva. Fast, flexible MUS enumeration. *Constraints*, 21(2):223–250, 2016. doi:10.1007/s10601-015-9183-0.

[MAD+19] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *Proceedings of ESOP*, pages 30–59, 2019. doi:10.1007/978-3-030-17184-1_2.

[MLM21] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, chapter 4, pages 133–182. IOS Press, 2021. doi:10.3233/FAIA200987.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of DAC*, pages 530–535, 2001. doi:10.1145/378239.379017.

[MR98] M. Oliver Möller and Harald Rueß. Solving Bit-Vector Equations. In *Proceedings of FMCAD*, pages 36–48, 1998. doi:10.1007/3-540-49519-3_4.

[MS99] Joao Marques-Silva and Karem A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999. doi:10.1109/12.769433.

[NHRV01] Robert Nieuwenhuis, Thomas Hillenbrand, Alexandre Riazanov, and Andrei Voronkov. On the Evaluation of Indexing Techniques for Theorem Proving. In *Proceedings of IJCAR*, pages 257–271, 2001. doi:10.1007/3-540-45744-5_19.

[NN93] Pilar Nivela and Robert Nieuwenhuis. Saturation of First-Order (Constrained) Clauses with the *Saturate* System. In *Proceedings of RTA*, pages 436–440, 1993. doi:10.1007/978-3-662-21551-7_33.

[NP20] Aina Niemetz and Mathias Preiner. Ternary Propagation-Based Local Search for more Bit-Precise Reasoning. In *Proceedings of FMCAD*, pages 214–224, 2020. doi:10.34727/2020/isbn.978-3-85448-042-6_29.

[NP23] Aina Niemetz and Mathias Preiner. Bitwuzla. In *Proceedings of CAV*, pages 3–17, 2023. doi:10.1007/978-3-031-37703-7_1.

[NPB17] Aina Niemetz, Mathias Preiner, and Armin Biere. Propagation based local search for bit-precise reasoning. *Formal Methods in System Design*, 51(3):608–636, 2017. doi:10.1007/s10703-017-0295-6.

[NPWB18] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2, BtorMC and Boolector 3.0. In *Proceedings of CAV*, pages 587–595, 2018. doi:10.1007/978-3-319-96145-3_32.

[NPZ24] Aina Niemetz, Mathias Preiner, and Yoni Zohar. Scalable Bit-Blasting with Abstractions. In *Proceedings of CAV*, pages 178–200, 2024. doi:10.1007/978-3-031-65627-9_9.

[NR01] Robert Nieuwenhuis and Albert Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001. doi:10.1016/B978-044450813-3/50009-6.

[PFG20] Lauren Pick, Grigory Fedyukovich, and Aarti Gupta. Automating Modular Verification of Secure Information Flow. In *Proceedings of FMCAD*, pages 158–168, 2020. doi:10.34727/2020/isbn.978-3-85448-042-6_23.

[PMP+16] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of PLDI*, pages 614–630, 2016. doi:10.1145/2908080.2908118.

[RBK22] Jakob Rath, Armin Biere, and Laura Kovács. First-Order Subsumption via SAT Solving. In *Proceedings of FMCAD*, pages 160–169, 2022. doi:10.34727/2022/ISBN.978-3-85448-053-2_22.

[REK+24] Jakob Rath, Clemens Eisenhofer, Daniela Kaufmann, Nikolaj Bjørner, and Laura Kovács. PolySAT: Word-level Bit-vector Reasoning in Z3, 2024. Accepted for VSTTE 2024. arXiv:2406.04696.

[Rob65] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965. doi:10.1145/321250.321253.

[RWB+17] Andrew Reynolds, Maverick Woo, Clark W. Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. Scaling Up DPLL(T) String Solvers Using Context-Dependent Simplification. In *Proceedings of CAV*, pages 453–474, 2017. doi:10.1007/978-3-319-63390-9_24.

[Sch13] Stephan Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, pages 45–67, 2013. doi:10.1007/978-3-642-36675-8_3.

[SCV19] Stephan Schulz, Simon Cruanes, and Petar Vukmirovic. Faster, Higher, Stronger: E 2.3. In *Proceedings of CADE*, pages 495–507, 2019. doi:10.1007/978-3-030-29436-6_29.

[SGSM20] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *Proceedings of CCS*, pages 621–640, 2020. doi:10.1145/3372297.3417250.

[SRV01] R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov. Term Indexing. In *Handbook of Automated Reasoning*, pages 1853–1964. Elsevier and MIT Press, 2001. doi:10.1016/B978-044450813-3/50028-X.

[SST14] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A Cross-Community Infrastructure for Logic Solving. In *Proceedings of IJCAR*, pages 367–373, 2014. doi:10.1007/978-3-319-08587-6_28.

[Sut17] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017. https://www.tptp.org/. doi:10.1007/s10817-017-9407-7.

[Sut24] Geoff Sutcliffe. Stepping Stones in the TPTP World. In *Proceedings of IJCAR*, pages 30–50, 2024. https://www.tptp.org/. doi:10.1007/978-3-031-63498-7_3.

[Tam98] Tanel Tammet. Towards Efficient Subsumption. In *Proceedings of CADE*, pages 427–441, 1998. doi:10.1007/BFb0054276.

[Tan24] Ole Tange. GNU Parallel 20240122 ('Frederik X'), January 2024. GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them. doi:10.5281/zenodo.10558745.

[Vor14] Andrei Voronkov. AVATAR: The Architecture for First-Order Theorem Provers. In *Proceedings of CAV*, pages 696–710, 2014. doi:10.1007/978-3-319-08867-9_46.

[Wal00] Toby Walsh. SAT v CSP. In *Proceedings of CP*, pages 441–456, 2000. doi:10.1007/3-540-45349-0_32.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS Version 3.5. In *Proceedings of CADE*, pages 140–145, 2009. doi:10.1007/978-3-642-02959-2_10.

[Wei01] Christoph Weidenbach. Combining Superposition, Sorts and Splitting. In *Handbook of Automated Reasoning*, pages 1965–2013. Elsevier and MIT Press, 2001. doi:10.1016/B978-044450813-3/50029-1.

[WNW+21] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021. doi:10.1145/3434304.

[WSS19] Wenxi Wang, Harald Søndergaard, and Peter J. Stuckey. Wombit: A Portfolio Bit-Vector Solver Using Word-Level Propagation. *Journal of Automated Reasoning*, 63(3):723–762, 2019. doi:10.1007/s10817-018-9493-1.

[WW08] Christoph Weidenbach and Patrick Wischnewski. Contextual Rewriting in SPASS. In *Proceedings of PAAR*, 2008. URL: https://ceur-ws.org/Vol-373/paper-10.pdf.

[ZIM+22] Yoni Zohar, Ahmed Irfan, Makai Mann, Aina Niemetz, Andres Nötzli, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. Bit-Precise Reasoning via Int-Blasting. In *Proceedings of VMCAI*, pages 496–518, 2022. doi:10.1007/978-3-030-94583-1_24.

[ZWR16] Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. Deciding Bit-Vector Formulas with mcSAT. In *Proceedings of SAT*, pages 249–266, 2016. doi:10.1007/978-3-319-40970-2_16.