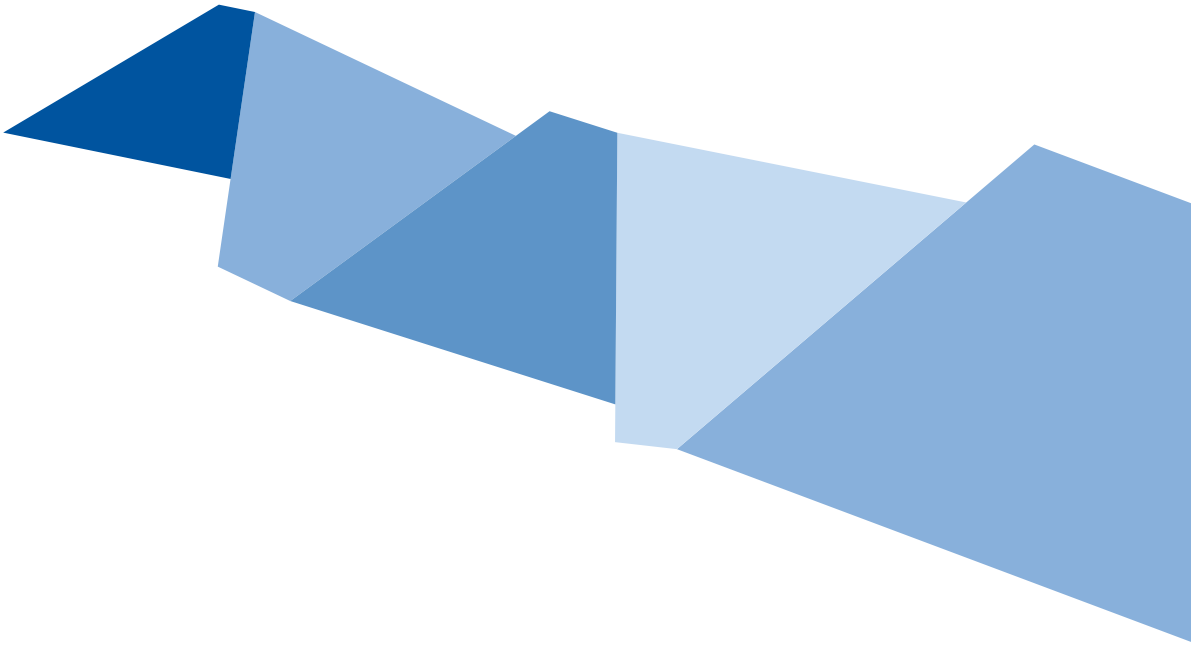


Cylindrical Algebraic Decomposition for Nonlinear Arithmetic Problems

Gereon Kremer

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2020-04
RWTH Aachen University | Department of Computer Science | May 2020

Cylindrical Algebraic Decomposition for Nonlinear Arithmetic Problems

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Gereon Kremer, Master of Science

aus Bergisch Gladbach

Berichter: Universitätsprofessorin Dr. Erika Ábrahám
Universitätsprofessor Dr. James H. Davenport
Privatdozent Dr. Viktor Levandovskyy

Tag der mündlichen Prüfung: 12. März 2020

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

Abstract

Satisfiability modulo theories solving is a technology to solve logically encoded problems for many applications like verification, testing, or planning. Among the many theories that are considered within this logical framework, nonlinear real arithmetic stands out as particularly challenging, yet decidable and sufficiently well understood from a mathematical perspective. The most prominent approach that can decide upon nonlinear real questions in a complete way is the cylindrical algebraic decomposition method.

We explore the usage of the cylindrical algebraic decomposition method for satisfiability modulo theories solving, both theoretically and experimentally. This method is commonly understood as an almost atomic procedure that gathers information about an algebraic problem and then allows to answer all kinds of questions about this algebraic problem afterward. We essentially break up this method into smaller components that we can then process in varying order to derive the particular piece of information – whether the problem is satisfiable or unsatisfiable – allowing to avoid some amount of computations. As this method often exhibits doubly exponential running time, these savings can be very significant in practice.

We furthermore embed this method in the regular satisfiability modulo theories framework where the cylindrical algebraic method is faced with a sequence of problems that are "related" in the sense that they usually share large parts of their problem statements. We devise different approaches to retain information from a previous run so that it can be reused when the problem is only "extended" as well as purging now obsolete information if the problem is "reduced". These variants change in how much information can be reused, the granularity of the information that is removed, and how much bookkeeping needs to be done.

This integration is then enhanced with techniques that are more or less well-known in the computer algebra community, for example, different projection operators, equational constraints, or employing the so-called resultant rule. Furthermore, we present novel features necessary for an efficient embedding in the satisfiability modulo theories framework like infeasible subset computations and early termination as well as extensions to integer problems and optimization problems.

We then turn to an alternative approach to satisfiability modulo theories solving that is commonly called model-constructing satisfiability calculus. The core idea of this framework is to integrate the theory reasoning, in particular the construction of a theory model, tightly with the Boolean reasoning. The most popular theory reasoning engine is again based on the cylindrical algebraic decomposition method, though we focus on the overall framework here.

We start with our own variant of the model-constructing satisfiability calculus and discuss some general insights and changes compared to current implementations. We then proceed to present a whole series of reasoning engines for arithmetic problems and show how a proper (though still naive) combination of those serves to significantly improve a practical solver. We also show how the tight integration into the Boolean reasoning allows for novel strategies for notoriously hard problems like the theory variable ordering or expedient cooperation between the Boolean and the theory reasoning.

Finally, we consider the theoretical relation of the model-constructing satisfiability calculus to other proof systems, in particular, the aforementioned regular satisfiability modulo theories solving. Under certain assumptions – that turn out to be instructive in and of themselves – we show that they are equivalent with respect to their proof complexity and even establish what we call "algorithmic equivalency" afterward.

Zusammenfassung

Satisfiability modulo Theories Solving ist eine Technologie, um logisch kodierte Probleme für Anwendungen wie Verifikation, Testen oder Planungsprobleme zu lösen. Unter den in diesem Framework untersuchten Theorien sticht die nicht-lineare reelle Arithmetik heraus: Sie ist anspruchsvoll, aber noch entscheidbar, und aus mathematischer Sicht hinreichend gut verstanden. Die bekannteste Möglichkeit, solche Probleme vollständig zu behandeln, ist die Methode der Zylindrisch Algebraischen Zerlegung.

Wir untersuchen die Verwendung dieser Methode für Satisfiability modulo Theories sowohl theoretisch als auch experimentell. Die Methode wird typischerweise als atomarer Algorithmus verstanden, der zunächst Informationen über das Eingabeproblem sammelt und anschließend darauf basierend eine Vielzahl von Fragen über die Eingabe beantworten kann. Wir zerlegen die Methode in kleinere Teile, die dann in beliebiger Reihenfolge ausgeführt werden können, um die entscheidende Information – ob das Problem erfüllbar oder unerfüllbar ist – ableiten zu können, ohne tatsächlich alle Berechnungen vollständig durchführen zu müssen. Da die Methode häufig eine doppelt exponentielle Laufzeit aufweist, können diese Einsparungen in der Praxis erheblich sein.

Wir betten diese Methode anschließend in das typische Framework für Satisfiability modulo Theories ein, bei dem die Methode der Zylindrisch Algebraischen Zerlegung eine Folge von Eingabeproblemen beantworten muss. Diese sind „ähnlich“ in dem Sinne, dass üblicherweise weite Teile der Problemstellung übereinstimmen. Wir zeigen verschiedene Ansätze, um Berechnungen von vorherigen Läufen wiederzuverwenden (falls das Problem nur „erweitert“ wurde) oder einzelne Berechnungen zu entfernen (falls das Problem „reduziert“ wurde). Diese Varianten unterscheiden sich in Bezug auf die Menge der wiederverwendbaren Berechnungen, der Granularität der zu entfernenden Berechnungen und dem Aufwand für die Buchhaltung.

Diese Einbettung wird schließlich durch mehr oder weniger bekannte Techniken aus der Computeralgebra erweitert, beispielsweise verschiedene Projektionsoperatoren, Equational Constraints oder die sogenannte Resultantenregel. Zusätzlich entwickeln wir Funktionen, die für eine effiziente Behandlung im Kontext von Satisfiability modulo Theories notwendig sind, wie Gründe für Unerfüllbarkeit, vorzeitige Terminierung oder die Erweiterung auf ganzzahlige Probleme oder Optimierungsprobleme.

Anschließend wenden wir uns einem alternativen Ansatz für Satisfiability modulo Theories zu, dem Model-Constructing Satisfiability Calculus. Die Kernidee ist hierbei, die Berechnungen in der Theorie enger mit denen auf boolescher Ebene zu verzahnen, insbesondere die Konstruktion einer Belegung für Theorievariablen. Die Hauptmethode für die Theorie basiert auch hier auf der Zylindrisch Algebraischen Zerlegung, wobei wir uns in diesem Teil mehr auf das Framework konzentrieren.

Wir stellen zunächst unsere Variante des Model-Constructing Satisfiability Calculus vor und diskutieren das generelle Verständnis und Unterschiede zu anderen Implementierungen. Anschließend präsentieren wir eine Reihe von Methoden für Berechnungen in der Theorie und zeigen, wie selbst eine recht naive Kombination dieser Methoden eine praktische Implementierung wesentlich verbessern kann. Zudem stellen wir fest, dass die enge Verzahnung zwischen booleschen und Theorieberechnungen neue Ansätze für notorisch schwere Probleme eröffnet, beispielsweise die Variablenordnung für Theorievariablen oder eine zielführende Kooperation zwischen booleschen und Theorieberechnungen.

Zuletzt schauen wir auf den theoretischen Zusammenhang des Model-Constructing Satisfiability Calculus und anderen Beweissystemen, insbesondere dem typischen Framework für Satisfiability modulo Theories. Unter gewissen Annahmen – die bereits für sich genommen interessant sind – sind diese bezüglich ihrer Beweiskomplexität äquivalent und wir zeigen sogar einen stärkeren Zusammenhang, den wir als „algorithmische Äquivalenz“ bezeichnen.

Declaration of Authorship

I, Gereon Lukas Kremer declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research. I do solemnly swear that:

1. This work was done wholly or mainly while in candidature for the doctoral degree at this faculty and university;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others or myself, this is always clearly attributed;
4. Where I have quoted from the work of others or myself, the source is always given. This thesis is entirely my own work, with the exception of such quotations;
5. I have acknowledged all major sources of assistance;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published before. A detailed list can be found in Section 1.2.

Aachen, June 12, 2020

Acknowledgements

First and foremost, I want to thank Erika. She not only supervised this thesis and all the work that went into it but also provided for a wonderful environment to work in. This of course also includes my current and former colleagues, in particular Florian, Francesco, Rebecca, and Stefan, but also all other co-workers from the MOVES group.

My first few years were characterized by the liberty to explore a rather wide range of topics, also thanks to the funding provided by AlgoSyn. Later on, the SC^2 project provided for extremely valuable interaction with people that I learned a lot from, in particular my other two supervisors, James and Viktor. James proved time and again to be an irreplaceable source of knowledge and never tired to explain in his inimitable way whatever I asked about CAD. Viktor on the other hand not only provided advice on a number of student projects but also helped repeatedly with the algebraic foundations of this work.

Our cooperations, in particular within the SC^2 project, sparked many more enlightening and productive discussions and cooperations with, among others, John Abbott, Christopher W. Brown, Matthew England, Pascal Fontaine, Vijay Ganesh, Einar Broch Johnsen, Jacopo Mauro, and Thomas Sturm.

Neither the work presented in this thesis nor the thesis itself would have been possible in this form without the help of other people. While quite a few students contributed to `CArL` and `SMT-RAT` over these years, I want to thank Jasper and Sebastian in particular. Furthermore, I want to thank Jens and Marcel for their patience in a series of discussions about algebraic details in both theory and implementation.

Last but not least, I wish to thank my family and friends for their support throughout these years, in particular Beate, Thomas, Elisa, Fiona, and Noam.

Contents

Part I

Introduction

Introduction

Digital systems are becoming ubiquitous in all aspects of our lives, from obvious examples like smartphones or social media to less evident developments like autonomous aircraft and cars or smart home devices. While software used to be confined to the particular machine it was running on, more and more systems have a direct physical impact on their environment and preventing malfunction is becoming ever more important: a "Blue Screen of Death" or a "Kernel panic" on a desktop computer are way less threatening than having an autonomous car shut down on the highway.

This (perceived) threat is reinforced by an inevitable collection of well-documented software failures with fatal consequences: the Ariane 5 rocket exploded due to an incorrect conversion from floating-point to integers; the Therac-25 radiation therapy machine caused at least three deaths from overdoses of radiation, attributed to an overall poor software design prone to concurrency issues; several airplane incidents have been traced to erroneous behavior of automatic flight control systems, for example, Qantas Flight 72 in 2008 or Lion Air Flight 610 in 2018.

While solid software designs and extensive testing can do their part in preventing such events, *formal verification* presents a more rigorous approach that aims to *prove a component safe* under a certain specification. While the better part of formal verification is concerned with software, the above examples highlight that the interaction between software and hardware is particularly important and challenging.

One possible approach that has gained significant traction is to model a hardware component in combination with a software controller as a *hybrid system* that allows for *continuous or dynamic behavior*, as well as *discrete control*. Safety can then be shown by *reachability* (or rather non-reachability) of bad states. One popular approach to do this eventually encodes this reachability problem as a satisfiability problem and uses *satisfiability modulo theories* (SMT) solvers that can decide upon the satisfiability of *nonlinear real arithmetic*. While most solvers aimed at this application of SMT employ techniques like *interval constraint propagation* we propose the use of *exact algebraic* procedures like the *cylindrical algebraic decomposition* (CAD) method. We focus on *first-order logic with nonlinear real arithmetic* as a rigorous language that is independent of the actual application.

Within this work we start with an introduction to algebraic procedures and SMT solving and present two main techniques: firstly using CAD as a theory solver within a regular SMT solver and secondly using an alternative framework for SMT solving called `MCSAT` whose theory reasoning is also based on CAD. Note that we do not consider any verification techniques that build on SMT solvers, but only discuss SMT solving itself.

1.1 Related work

We give a short overview of various works that are in some way or another related to what we discuss in this thesis. We start with some historical notes on decision procedures for nonlinear real arithmetic – or the theory of the reals – and give a few more details on important milestones in the development and improvement of CAD. We then name some alternative approaches that are incomplete in that they can not deal with the whole set of nonlinear problems and finally survey the solving techniques used in the SMT community for nonlinear problems.

1.1.1 Theory of the reals

Soon after the formalization of mathematical logic and theories was (mostly) agreed on early in the twentieth century, the desire to have methods to determine the truth of statements within certain logics arose. For this quest for *decision procedures*, Hilbert coined the term *metamathematics*, aiming for mechanical algorithms that could answer whether a given statement (formally *sentence*) is valid (or is a consequence) within a given theory (or an axiomatic system that defines it).

A closely connected (though not identical) question is whether a theory admits *quantifier elimination*, that is whether we can construct a quantifier-free formula that is equivalent to a given quantified formula. A quantifier elimination method usually constitutes a complete *decision procedure* by eliminating all quantifiers from a sentence (a formula without free variables) and evaluating the resulting sentence (that contains no variables) to either *true* or *false*. Based on this, the question for the satisfiability of a formula φ with free variables $\overline{x}$ can be rephrased equivalently to whether $\exists \overline{x}.\ \varphi$ is valid.

Arguably one of the easiest quantifier elimination techniques is the recursive variant of enumeration for propositional logic. It works by instantiating the variable with both *true* and *false* and constructing their disjunction or conjunction, for existential or universal quantification, respectively, where $\varphi[x/c]$ denotes the syntactic substitution of c for a variable x in φ.

$$\exists x.\ \varphi \Leftrightarrow (\varphi[x/\mathit{true}] \vee \varphi[x/\mathit{false}])$$
$$\forall x.\ \varphi \Leftrightarrow (\varphi[x/\mathit{true}] \wedge \varphi[x/\mathit{false}])$$

We can easily generalize this idea for variables of any finite domain D_x and already note that this technique fails for infinite domains:

$$\exists x.\ \varphi \Leftrightarrow \bigvee_{d \in D_x} \varphi[x/d] \qquad\qquad \forall x.\ \varphi \Leftrightarrow \bigwedge_{d \in D_x} \varphi[x/d]$$

Another possibility for quantifier elimination for propositional logic is using the resolution rule to successively eliminate Boolean variables from a formula in conjunctive normal form as we later show in Algorithm 3.3. It essentially combines two *clauses* to produce a new clause without the variable that is to be eliminated and gives certain guarantees when original clauses can safely be discarded. Very similarly, Dines [Din19] and Motzkin [Mot36] reframed a method originally due to Fourier [Fou25; Fou26] to provide quantifier elimination for sets of linear inequalities over real variables – that is without multiplication among variables – to what we now call the *Fourier–Motzkin variable elimination*.

The overarching question for decision procedures promptly incited an active field of research that not only rediscovered existing methods but also devised new decision procedures and provided fundamental new insights into the nature of logic itself.

One of the earliest (published) results in this direction is due to Presburger [Pre30] which gives a decision procedure – again by quantifier elimination – for formulae with linear constraints over integer variables. (We strongly recommend [Sta84] and [Cro75] not only for an enjoyable read but also for some historical perspective and hints to further work in this area.)

Shortly after these, some fundamental results emerged that restrict the range of logics for which we can hope to find a decision procedure, most prominently the undecidability results due to Gödel [Göd31] and shortly after due to Church [Chu36] and his student Rosser [Ros36]. Most importantly (for us) they show that formulae with nonlinear constraints over integer variables are undecidable in general, that is no complete decision procedure for nonlinear integer problems exists.

Whether nonlinear real arithmetic is decidable remained unsolved for quite some time, though it seems now that Tarski essentially had the result since about 1930 [Chu69]. It was only (publicly) resolved in [Tar51] and [Sei54] which (once again) employed a constructive quantifier elimination method. We refer to [Dri88] for a historical and thematical assessment with many references to related work.

All methods above have been superseded by more efficient techniques in most practical applications. Propositional logic, which only consists of Boolean variables and connectives, is commonly dealt with using `CDCL`-style SAT solving (see Section 3.6), the simplex method is used for linear real arithmetic and branch-and-bound-based methods are used for the linear integer case. However, all of the original quantifier elimination methods are still useful for more specialized cases as we will see for Fourier–Motzkin elimination in Section 8.3.3, for example.

Tarski's method is different in this respect in that it exhibits an asymptotic complexity that is *not elementary* – it can not be bounded by an exponential tower of finite size – and hence was never seriously used in practice. Nevertheless, it remains one of the most important results in this area, at least sparking the hope for a solution efficient enough for practical applications.

Encouraged and inspired by Tarski's result, a number of alternative (and hopefully more efficient) decision procedures emerged including the cylindrical algebraic decomposition method due to Collins [Col74], but also other methods due to Grigor'ev and Vorobjov [GV88], Renegar [Ren88], or Basu, Pollack, and Roy [BPR96] which all improved significantly upon the asymptotic complexity of Collins' cylindrical algebraic decomposition (which is doubly exponential in the number of variables).

One might ask why this thesis is concerned with CAD then, and consequently why Collins' work sparked a whole field of sustained and active research while the other methods stayed theoretical side notes. The answer may be given by Hong [Hon91] in that these advances on the asymptotic side are outweighed by huge constants in practice. The approaches due to Grigor'ev and Renegar are impractical even for trivial problems and the break-even-point is moved so far away that there is essentially no way to reach it. The third approach due to Basu, Pollack, and Roy – not included in Hong's comparison as it is more recent – also did not result in an implementation to the best of our knowledge.

Consequently, the cylindrical algebraic decomposition method is the predominant decision procedure (or quantifier elimination method) for nonlinear real problems nowadays, if one wishes to have a complete method.

1.1.2 Origins of CAD

We now give a summary of the genesis of the cylindrical algebraic decomposition method and refer to [CJ98] for more information. Shortly after Collins received his Ph.D. degree under Rosser – about twenty years after [Ros36] – Collins was already concerned with Tarski's work and actually aimed at implementing it at IBM, noting that the "amount of labor involved in even very simple applications has prohibited any progress in this aspiration to this date" [Col56].

Though we assume that this project did not yield a practical implementation – witnessed by the lack of ensuing publications – Collins stuck to the question of quantifier elimination and made significant advances in the field we now call computer algebra and beyond, including reference counting [Col60], the implementation of the early computer algebra systems PM [Col66] and SAC-1 [Col71b], and efficient ways to compute subresultants [Col67] and resultants [Col71a].

In 1973, Collins gave a talk [Col73] presenting a novel method for quantifier elimination that he called "cylindrical algebraic decomposition", followed by two papers [Col74; Col75]. They not only contain a detailed description of the algorithms but also give bounds on the asymptotic complexity which is doubly exponential – way better than the non-elementary complexity of Tarski's method. Furthermore, they also hint to an ongoing effort to implement it within the SAC-1 computer algebra system.

The implementation apparently proved to be more difficult than expected: a group around Collins with experience from SAC-1 and SAC-2 started in 1974 but still had major sub-algorithms unimplemented in 1978. Meanwhile, parts of the method were implemented by Müller [Mül78] to solve nonlinear optimization problems. In 1979, finally, Arnon [Arn81] took up the task to produce a complete CAD implementation that was eventually integrated into SAC-2 [Col85].

This first implementation was then promptly followed by various improvements to CAD – mostly within the projection operator as we discuss in Section 5.2 – by several of Collins' students, most prominently McCallum [McC84; McC85; McC88], Hong [Hon90], and Brown [Bro01].

At the latest since the second half of the 1980s, the topic was picked up by a larger community as witnessed by being mentioned in [Wei88], some theoretical contributions based on CAD by Davenport and Heintz [DH88] and another projection operator for CAD due to Lazard [Laz94].

1.1.3 CAD today

During the first attempts for a full CAD implementation, it became clear that it requires a large amount of nontrivial functionality to work and thus the number of implementations of CAD is still somewhat limited today. Most notably, we have `QEPCAD` [Hon91] due to Hong and its successor `QEPCAD B` [Bro03] (mostly due to Brown), as well as implementations within the computer algebra systems `Redlog` [DS97], Mathematica [Str00] and Maple [CMX+09; IYA+09].

Only a few special-purpose versions of CAD exist beyond the above, as far as we know. The `NLSAT`-style explanations that we also discuss in Section 8.3 are implemented in both `Z3` [JM12] and `Yices` [Dut14] based on libpoly [JD17]. Also, the theorem prover Coq features a custom CAD implementation [Mah07] for proofs over nonlinear arithmetic that is tailored to proof generation.

To the best of our knowledge, our implementation in `SMT-RAT` (and its related projects) is the only other general-purpose implementation of CAD. Earlier versions also exist within our libraries GiNaCRA [LÁ11] and CArL [KÁ18]. The current implementation powers not only a theory solver for regular SMT solving as described in [CKJ+15], but also a quantifier elimination method [Neu18a] and our version of `MCSAT` [NKÁ19].

1.1.4 Variants of CAD

The CAD implemented within Maple – or rather the RegularChains library [CMX+09] – deviates from the traditional CAD framework presented in this work. While what we call CAD is only concerned with real numbers, it can also be used in the *complex space* as presented in [CMX+09]. Intuitively, we can understand it as using a larger projection to obtain a CAD-like object in the complex space, using a somewhat easier lifting procedure to obtain sample points, and projecting them into the real space.

Another recent variant of CAD stays in the real space but (partly) departs from the fundamental concept of cylindricity. In what is called *non-uniform cylindrical algebraic decomposition* or NuCAD [Bro15], the individual regions may have overlapping projections onto lower-dimensional space. In this sense, every cell itself is still cylindrical, but the entirety of cells is not *arranged cylindrically* relative to each other. While this approach makes it harder to perform classical quantifier elimination, it is suitable for answering many other questions about a nonlinear problem, promising a significantly reduced number of cells in practice.

We presented a novel interpretation of CAD in [ÁDE+20]. We can understand it as bringing the conflict-driven reasoning style of `MCSAT` into a regular CAD-based `CDCL(T)`-style theory solver. Similar to `MCSAT`, we construct a theory model dimension-wise and exclude intervals on every dimension. Once a dimension is fully covered, we combine these intervals into an interval on a lower dimension using a characterization step that closely resembles a CAD projection. Preliminary experiments suggest, that this approach is competitive to the CAD-based theory solver we present in Chapter 6.

1.1.5 Other complete methods

As we already mentioned, the CAD method is not the only complete method that deals with nonlinear real arithmetic. The first method presented by Tarski [Tar51] had non-elementary complexity and thus it was essentially clear since its discovery that it would not be suitable for implementation – though the method is constructive.

Later approaches – we already mentioned Grigor'ev and Vorobjov [GV88], Renegar [Ren88], and Basu, Pollack, and Roy [BPR96] – all share that they significantly improve upon the theoretical complexity of the cylindrical algebraic decomposition method. Still, to the best of our knowledge, no practical implementation of either of the three methods exists. For the former two, Hong gave a strong argument in [Hon91] in that – though the asymptotic complexity improved – the methods are incredibly inefficient in practice.

For the approach due to Basu, Pollack, and Roy [BPR96] we could only conjecture why it has not been implemented or at least did not result in any implementation we are aware of. Our group – once upon a time – tried to implement this approach as well but abandoned it in favor of cylindrical algebraic decomposition due to the complexity of the mathematical background and the complexity of the upcoming implementation.

1.1.6 Incomplete methods

Trying to deal with nonlinear problems has a long history – just think of the Pythagorean theorem – and has thus spawned a wealth of diverse methods with very different targets, characteristics, and trade-offs. We have previously focused on methods that give us precise (or symbolic) information on nonlinear (polynomial) equalities and inequalities. In the following, we give an overview on different approaches that for some reason or another do not satisfy those conditions, for example, because they are approximate (what we call numerical algorithms) or are restricted in their input in that they can deal only with equalities (or inequalities) or only polynomial up to a certain degree.

Note that these other methods are by no means less relevant or *worse*. Quite the contrary, some of the following approaches are well established and routinely used in practice and are in this respect way ahead of CAD.

1.1.6.1 Linearization

Possibly the most obvious approach is called *linearization* which tries to describe a nonlinear problem in terms of linear constraints. Note that this method is inherently incomplete as nonlinearity is a qualitative trait that we can not eliminate without loss of information: nonlinear constraints allow to describe solution spaces that are beyond the linear formalism and thus the linearization of something nonlinear can only ever be an approximation.

We, however, observe in practice that a linear approximation can very well be enough to answer different kinds of interesting questions about a nonlinear problem. Most applications have uncertainties, require tolerances, or are only concerned with a small area around some target point and we are looking for an open solution space that allows for safe over-approximations that can be linear. Even if we encounter equalities – and thus the solution is restricted to a lower-dimensional space – linearization techniques may be able to make a linear solver provide a solution that satisfies the nonlinear problem. Some examples can be found in [Isi95; ADG07].

A novel linearization approach for nonlinear SMT problems that proved to be very powerful in practice was presented in [CGI+18; Irf18] and was adapted within our solver `SMT-RAT` in [Zam19]. Though not finished yet, we hope that further improvements to this approach and proper integration with a CAD-based theory solver – or even within the `MCSAT` framework – will be fruitful.

1.1.6.2 Numerical algorithms

A wealth of fundamentally different methods to deal with nonlinear problems can be found in the field we call *numerical analysis*, employing numerical approximation in one form or another. Prominent examples include the *Newton method*, various methods for *linear equation systems* using matrix decompositions, the area commonly called *convex optimization*, eigenvalue problems or a plethora of methods for differential equations.

Some of them can be adapted to be useful in our field, and we have reported on some work in [Kre13; Kre18]. Unfortunately, such attempts oftentimes fail due to a very fundamental problem: numerical analysis is concerned with finding a *good approximation* and gives guarantees on the *convergence* – that is how fast the approximation improves – but usually gives no *absolute error bounds* on the numeric results.

Simply put: if you figure out your approximation is not good enough, you only need to let it run some more time. Figuring out whether your approximation is good enough is usually hard to determine, though. Furthermore, this research is usually very focused on what one could describe as "realistic" problems, which allows these methods to essentially fail for "obscure" corner cases, or at least require manual intervention to reformulation of the problem at hand.

This is in stark contrast to our expectations: we want to obtain "push-button solutions" that deal with any input problem from some formal language – first-order logics in our case – *without* human interaction. Furthermore, we consider our tools to be *sound and complete* in the sense that we can guarantee that our results are correct – and not *approximate* or *probably correct* – and that they *always* work, even for the more obscure cases. While we endure degraded performance for such corner cases, incorrectness is usually unacceptable.

Thus, we sometimes try to adapt numerical methods, but can usually only do so as fast preprocessing techniques to solve easy cases and need to provide complete methods as fall-backs. One such case is described in [Kre13] where eigenvalue computations serve as *preconditioning* for the actual root finding method. The usage of interval arithmetic in our implementation of real algebraic numbers that we describe in Section 2.5 also partly fits this description (though it is also strictly required for some operations).

1.1.6.3 Interval constraint propagation

Instead of abandoning numerical algorithms altogether, we might want to equip such methods with some technique to provide absolute guarantees on their results. The most prominent approach is called *interval arithmetic*, essentially defining arithmetic operations on intervals that capture the "real results" of some operation in an over-approximating way. To give a rough idea, let us consider multiplication:

Let $a, b \in \mathbb{R}$ be represented by two intervals $a \in (\underset{\sim}{a}, \tilde{a})$, $b \in (\underset{\sim}{b}, \tilde{b})$. Let us assume for now that the intervals are reasonably small as our intuition is that they approximate individual numbers a and b. An approximation of the product $a \cdot b$ can then be obtained from the intervals as follows:

$$a \cdot b \in (\underset{\sim}{a}, \tilde{a}) \cdot (\underset{\sim}{b}, \tilde{b}) = (\min\{\underset{\sim}{a} \cdot \underset{\sim}{b}, \underset{\sim}{a} \cdot \tilde{b}, \tilde{a} \cdot \underset{\sim}{b}, \tilde{a} \cdot \tilde{b}\}, \max\{\underset{\sim}{a} \cdot \underset{\sim}{b}, \underset{\sim}{a} \cdot \tilde{b}, \tilde{a} \cdot \underset{\sim}{b}, \tilde{a} \cdot \tilde{b}\})$$

This construction not only accumulates the inaccuracy of the two intervals for a and b but possibly introduces more inaccuracy. Firstly, one oftentimes uses floating-point representations for the interval endpoints which provide great performance, but also introduce additional rounding errors. Note that this is not only an issue for the precision of the result, but we also need to take care that this rounding happens safely – that is over-approximating – as we otherwise risk getting incorrect results by losing possible solutions.

Secondly, we may be unable to consider *algebraic* relations between variables. If for example, we know that $a = b$ it is clear that $a \cdot b = a^2 \geq 0$. If we perform direct interval

arithmetic we lose this knowledge and the resulting interval may very well contain negative numbers. We can sometimes treat such cases specifically, but these relations are oftentimes not that obvious and once this calculation has been performed, it is usually very difficult to exploit them after the over-approximation has been introduced.

The most popular – and most widely adopted – method to exploit interval arithmetic for SMT solving is called *interval constraint propagation* [BG06], being implemented in solvers like dReal [GKC13], iSAT [LSC+13; SKB13], or raSAT [TKO17], as well as our own solver `SMT-RAT` [Sch13]. Its fundamental idea is to use individual constraints to maintain an interval that contains the possible range of values for every variable and shrink these intervals using relatively simple propagations like the following:

$$(y, z \in (0,1) \land x = y \cdot z) \implies x \in (0,1)$$

This technique is surprisingly effective in excluding large parts of the search space quickly, and oftentimes even proving unsatisfiability. It is however notoriously difficult to construct satisfying assignments, in particular, if equalities are present. We mainly propose to use it to quickly shrink the space of solution candidates and then exploit these additional *bounds* in the algebraic methods as described in [LSC+13] or [KÁG19].

1.1.6.4 Virtual substitution

In [Wei88] Weispfenning studies the complexity of quantifier elimination of linear problems over different fields, which also turns out to be doubly exponential – though only in the number of quantifier alternations instead of the number of variables. He also gives a method that consequently runs in (singly) exponential time if no quantifier alternations are present.

This method was subsequently implemented within REDUCE [LW93] and extended to quadratic constraints [Wei97] to what we call *virtual (term) substitution*. The classical notion of virtual substitution essentially uses (parametric) solution formulae for the roots of polynomials and substitutes their results, thereby eliminating the respective variable. This, however, yields a restriction on the degree as solution formulae only exist for polynomials up to degree four. Even worse, the substitution rules for degree three are sufficiently difficult that they were only implemented recently in [Koš16].

The restriction on the degree of the polynomial can be lifted by a generalization of virtual substitution [KS15] which essentially substitutes a characterization of the polynomial root – as an expression in first-order logic – instead of an explicit representation. This allows for arbitrary degrees in principle, however, the only automated way to obtain these first-order logic expressions is by means of quantifier elimination. In this sense, one could argue that this method merely reproduces the results of parametric quantifier eliminations obtained from other procedures – for example CAD.

Despite these restrictions, virtual substitution is impressively effective in practice and certainly is a valuable tool for quantifier elimination and related problems. We thus also implement virtual substitution in our solver and use it in combination with cylindrical algebraic decomposition [CKJ+15].

1.1.6.5 Gröbner bases

Since the formalization of polynomial ideals, computer algebra was looking for a canonical way to represent such ideals and – if possible – construct them effectively. Buchberger

answered both questions with the introduction of *Gröbner bases* in 1965 [Buc65], giving conditions that characterize a canonical representation of a set of generators for a polynomial ideal and providing an algorithm to construct this set. To this day, Gröbner bases are one of the most important tools in computer algebra to deal with a set of polynomials – or implicitly with a set of equalities.

Essentially, Gröbner bases provide a canonical way to simplify a set of polynomials, retaining the set of common roots and thereby the solutions to the set of polynomial equalities. Though Gröbner bases have some caveats in practice – it may exhibit doubly exponential run time that is very sensitive to the variable ordering we need to choose – it is arguably the most successful tool to deal with such problems.

As for our problem of satisfiability of real arithmetic formulae, Gröbner bases have several restrictions that need to be overcome. Firstly, Gröbner bases only deal with polynomials – implicitly equalities – while we usually want to consider inequalities as well. We can convert inequalities to equalities while introducing new variables as described for example in [Jun12] like this.

$$p \geq 0 \Leftrightarrow \exists y.\ p - y^2 = 0$$

Note, however, that practical experience shows that this technique works well in some examples, but does not seem to help a lot in many other cases.

Secondly, Gröbner bases can determine whether some polynomials have a common complex root but usually can not argue about real roots. If we certify that no complex root exists we of course also have unsatisfiability in the reals, but a gap remains where a complex root exists but we do not find a real one.

Given that we are only interested in real solutions we have certain opportunities to apply additional simplifications to our polynomials – actually computing or approximating what is sometimes called the *real radical* instead of the Gröbner basis. Some ways to do exactly that are explored in [Jun12] and more recently as well in [ABP18; SYZ18].

Even if a complex root exists it is not trivial to actually construct one, even more so if one aims to find a real one. Some attempts are made in [Jun12], but again their applicability highly depends on the individual input.

1.1.7 Boolean reasoning and SMT solving

The formalization of mathematical logic we discussed at the beginning of Section 1.1.1 did not only launch a plethora of developments on arithmetic theories – though it seems to have received more attention – but also sparked the interest in decision procedures for Boolean logics.

Oftentimes – Fourier–Motzkin being a notable exception – methods for arithmetic theories naturally incorporated dealing with a Boolean structure. Thus the question of satisfiability – or validity – for purely Boolean formulae (propositional logic) was well-established but was usually treated very naively. It took until [DP60] for a method that outperformed simple enumeration in practice. We give an overview of the methods for the satisfiability of purely Boolean formulae in Chapter 3.

From the very beginning, the question of satisfiability for purely Boolean formulae was usually motivated by dealing with more complex logics. Nowadays, we usually consider the satisfiability of first-order logic formulae and call the corresponding satisfiability

problem, as well as methods to solve it, *satisfiability modulo theories* (SMT). That being said, propositional satisfiability is an established field of research on its own which has arguably outgrown its origins due to the impressive practical results.

While the origins of practical approaches for SMT solving are usually dated to the late 1970s and early 1980s, we tend to understand [DP60] as a form of SMT solving already, given that it employs a method for propositional satisfiability to solve the validity of a richer logic. Early works that approach what we understand as SMT solving today include [NO79; Sho79; NO80; BM90] while the approach that we understand as *(lazy) SMT solving* – a combination of a regular *SAT solver* and what we call *theory solvers* as described in Chapter 4 – emerged in the late 1990s, apparently at multiple places around the same time, for example [GS96; ACG99; BGV99; PRS+99].

1.1.8 SMT solving today

Shortly after the *lazy SMT solving* framework emerged, an effort was made to draw the whole "SMT community" together, or rather form such a community, by establishing both SMT-LIB [BFT16] in 2002 and the SMT-COMP [BMS05]. These initiatives and their close cooperation, in particular, made it possible to gather the better part of a comparably diverse community – that works on very different theories – using a common input language and a common set of benchmarks that both live in the SMT-LIB initiative.

Within the last five years, more than twenty different solvers have participated within about fifty different logics (about twenty of them involving quantifiers) at the annual SMT-COMP, witnessing both the wide-spread interest in SMT solving and the diversity of tackled logics. Note that some of the competing solvers are actually targeted at other problems and rather solve SMT problems as a side issue, for example, `AProVE`, `Alt-Ergo`, `Redlog`, and `Vampire`.

Given the scope of this thesis, we want to focus on solvers for nonlinear real arithmetic which is called `QF_NRA` in SMT-LIB. Within the last five years, the following seven solvers competed in this logic: `CVC4` [BCD+11], `MathSAT` [CGS+13], `raSAT` [TKO17], `veriT+Redlog` [FOS+18], `Yices` [Dut14], `Z3` [MB08c] and our own `SMT-RAT` [CKJ+15]. While `CVC4` and `MathSAT` employ linearization techniques as described in Section 1.1.6.1, `raSAT` is solely based on interval constraint propagation similar to Section 1.1.6.3, `veriT+Redlog` integrates the full-fledged computer algebra system `Redlog` into the SMT solver `veriT` and finally `Yices` and `Z3` employ an alternative framework for SMT solving called `MCSAT` that we discuss in more detail in Chapters 7 and 8.

Our own solver `SMT-RAT` implements all of these techniques with a particular focus on complete methods. As we have already seen, the only complete method for nonlinear arithmetic – that has ever been seriously implemented – is CAD. As far as we know, there are only two integrations of CAD into SMT solvers: 1. using CAD as a theory solver in regular SMT solving as implemented in `veriT+Redlog` and `SMT-RAT` and 2. adapting CAD as an explanation backend for `MCSAT` as implemented in `Yices`, `Z3`, and `SMT-RAT`.

1.2 Contributions

In this thesis, we present novel approaches that roughly fall into the following categories:

1. embedding and adapting the CAD method into an SMT compliant theory solver for nonlinear real arithmetic,
2. extending and using this theory solver for other problems like nonlinear integer arithmetic or quantifier elimination,
3. implementing the `MCSAT` proof system with a special focus on modularity, and
4. theoretical study of the `MCSAT` proof system.

Note that by "novel approaches" we do not mean to talk about "as of yet unpublished" results, but also about results presented in publications that have been (co-) authored by this author. Publications that are relevant for this are listed and discussed in the following Section 1.2.1. We now give a very brief overview of the contributions of this thesis, deferring a more detailed discussion to Section 10.1.

We start with a general introduction into CAD that shifts the understanding into a point of view that is more suitable for our application, followed by a novel proof system that allows decomposing the oftentimes "monolithic" CAD easily. We then discuss a number of design choices and heuristics and show how to apply this variant of CAD to integer problems, quantifier elimination, and optimization problems.

The second part is concerned with `MCSAT`, starting with a general definition and two extensions for *model refinement* (instead of *model construction*) and optimization. We then describe a concrete implementation based on a novel embedding into an existing `CDCL`-style SAT solver and various possibilities for its main theory reasoning methods – the *assignment finder* and the *explanation function* – and evaluate this implementation using several different strategies. Finally, we discuss some theoretical observations that relate `CDCL(T)`-style SMT solving with `MCSAT`-style SMT solving in the hope to underpin some practical observations with theoretical results.

1.2.1 Relevant publications

We present the list of our publications that are relevant to this thesis and detail the author's contributions to them below. For full bibliographic references, we refer to the bibliography at the end of this work.

[ÁNK17] Erika Ábrahám, Jasper Nalbach, and Gereon Kremer. "Embedding the Virtual Substitution Method in the Model Constructing Satisfiability Calculus Framework". SC^2 2017.

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. "SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving". SAT 2015.

[HKÁ18] Rebecca Haehn, Gereon Kremer, and Erika Ábrahám. "Evaluation of Equational Constraints for CAD in SMT Solving". SC^2 2018.

[KÁ18] Gereon Kremer and Erika Ábrahám. "Modular strategic SMT solving with SMT-RAT". *Acta Universitatis Sapientiae, Informatica*, 2018.

[KÁ20] Gereon Kremer and Erika Ábrahám. "Fully Incremental Cylindrical Algebraic Decomposition". *Journal of Symbolic Computation*, 2020.

[KÁG19] Gereon Kremer, Erika Ábrahám, and Vijay Ganesh. "On the Proof Complexity of MCSAT". SC^2 2019.

[KCÁ16] Gereon Kremer, Florian Corzilius, and Erika Ábrahám. "A Generalised Branch-and-Bound Approach and Its Application in SAT Modulo Nonlinear Integer Arithmetic". CASC 2016.

[NKÁ19] Jasper Nalbach, Gereon Kremer, and Erika Ábrahám. "On Variable Orderings in MCSAT for Non-linear Real Arithmetic (extended abstract)". SC^2 2019.

[VKÁ17] Tarik Viehmann, Gereon Kremer, and Erika Ábrahám. "Comparing Different Projection Operators in the Cylindrical Algebraic Decomposition for SMT Solving". SC^2 2017.

The contributions published in these papers mainly fall into three categories: adaptations and extensions of CAD for SMT solving, research on `MCSAT`, and implementation within `SMT-RAT`. For all the above papers, the author took part in the discussions and writing of the actual publication. Most publications were attained in cooperation with other researchers and we discuss the individual contributions in more detail now.

The overall architecture of our integration of CAD into SMT is described in [KÁ20] which contains most of what is described in Chapter 6, though presented as actual algorithms while we formulate it as a proof system here. The fundamental ideas to perform CAD incrementally predate the author's work and are mostly due to Ulrich Loup (see for example [LÁ11; CLJ$^+$12; LSC$^+$13; Lou18]). The presented approach is however significantly different from previous publications and is based on a completely new implementation conducted by the author. A preliminary presentation of this approach was given by the author at [ÁK18].

While working on the overall framework for incremental CAD, a whole range of individual issues were investigated in the context of CAD. First, we extended `SMT-RAT` to allow for efficient solving of nonlinear integer formulae using a branch-and-bound approach, resulting in [KCÁ16]. While Florian Corzilius worked on the adaption of virtual substitution, the author was responsible for the required changes in CAD.

Together with Tarik Viehmann, we analyzed how different projection operators (that we describe in Section 5.2) perform in the context of SMT solving. The work started as his thesis project in [Vie16] and was then extended to [VKÁ17]. The author mainly provided the underlying implementation for the individual projection operators, consulted on ideas for further experiments, and finally took part in writing the publication.

In her thesis [Hae18], Rebecca Haehn integrated the usage of equational constraint (as presented in Section 5.2.8) into our incremental CAD and how they improve the applicability of CAD in practice. Given the deep level of integration into the existing implementation, the author not only contributed essential ideas but also parts of the implementation in the preparation of the subsequent paper [HKÁ18].

This work was accompanied by improvements to the general framework in `SMT-RAT` and the underlying library `CArL`, as well as various additional solving techniques not directly related to this thesis. Some of them were presented in a series of papers, others remain unpublished, but are freely available as implemented in `CArL` and `SMT-RAT`. While [CLJ$^+$12] predates the author's activities, the author contributed major parts of the implementation and writing to both [CKJ$^+$15] and [KÁ18].

Another, though related, line of research was the exploration of the MCSAT solving framework (originally presented in [JM12; MJ13]) which we still consider work in progress. In [ÁNK17] (based on [Nal17]) we present Jasper Nalbach's work to employ virtual substitution as a novel explanation backend in MCSAT in addition to CAD and Fourier–Motzkin (that were previously described in [JM12; JBM13]). The MCSAT framework gives a more flexible integration of Boolean and theory reasoning and we explored possible heuristics in [NKÁ19]. The author contributed the majority of our implementation of MCSAT, initially in cooperation with Florian Corzilius and eventually supported by Jasper Nalbach.

In [KÁG19], we had a more theoretical look at MCSAT by relating it to CDCL*(T) in terms of its proof complexity. We have proven MCSAT to be equivalent to Res*(T) which in turn is equivalent to CDCL*(T), as was previously shown in [RKG18]. This whole topic was almost exclusively one of the author's side projects.

1.2.2 Further publications

During the time the author worked on the presented topics, the following other publications and talks where attained. Again we refer to the bibliography at the end of this work for full references.

[ÁCJ+16] Erika Ábrahám, Florian Corzilius, Einar Broch Johnsen, Gereon Kremer, and Jacopo Mauro. "Zephyrus2: On the Fly Deployment Optimization Using SMT and CP Technologies". SETTA 2016.

[ÁDE+20] Erika Ábrahám, James H. Davenport, Matthew England, and Gereon Kremer. "Deciding the Consistency of Non-Linear Real Arithmetic Constraints with a Conflict Driven Search Using Cylindrical Algebraic Coverings". *arXiv e-prints*, 2020.

[ÁK16] Erika Ábrahám and Gereon Kremer. "Satisfiability Checking: Theory and Applications". SEFM 2016.

[ÁK17] Erika Ábrahám and Gereon Kremer. "SMT Solving for Arithmetic Theories: Theory and Tool Support". SYNASC 2017.

[Kre18] Gereon Kremer. "Computer Algebra and Computer Science". ACA 2018.

A case study on the applicability of SMT solving – in particular in comparison to *constraint programming* – was conducted in [ÁCJ+16], aiming at automated deployment of cloud-based web applications. The author contributed to the problem formulation and encoding as an SMT formula, performed the benchmarking presented in the paper, and took part in the writing process.

The remaining papers [ÁK16; ÁK17] and the talk [Kre18] all aim at presenting the SMT approach with its unique advantages and challenges to different audiences. While [ÁK16] was targeted at software engineering researchers interested in formal verification, we describe the current state-of-the-art (including our own solver SMT-RAT) to the computer algebra community in [ÁK17]. The talk [Kre18] was a result of the SC^2 project and presented concrete challenges we encountered when integrating established computer algebra software (like Maple or CoCoALib) into an SMT framework. Finally, we proposed a novel CAD-based decision procedure in [ÁDE+20] that employs a conflict-driven strategy within a theory solver with surprisingly good results.

1.2.3 Supervised thesis projects

The author oversaw many thesis projects that mostly explored topics more or less closely related to this work. While some of them resulted in subsequent publications that are mentioned later, the others might be interesting to read as well, covering topics that just did not make it into the present work.

[Bar18] Lorena Calvo Bartolomé. “Using Fourier-Motzkin Variable Elimination for MCSAT Explanations in SMT-RAT”.

[Fra20] Hanna Franzen. “Conflict Driven Cylindrical Algebraic Coverings for Nonlinear Arithmetic in SMT Solving”.

[Gro17] Marta Grobelna. “Solving Pseudo-Boolean Constraints”.

[Hae18] Rebecca Haehn. “Using Equational Constraints in an Incremental CAD Projection”.

[Hen17] Wanja Hentze. “Computing minimal infeasible subsets for the Cylindrical Algebraic Decomposition”.

[Kor17] Leonard Korp. “SMT-based Planning for Autonomous Robot Fleets”.

[Krü15] Andreas Krüger. “Bitvectors in SMT-RAT and Their Application To Integer Arithmetics”.

[Kuk19] Denis Kuksaus. “SMT-basierte Lösung reell-algebraischer Probleme mittels Linearisierung”.

[Lös18] Christopher Lösbrock. “Implementing an Incremental Solver for Difference Logic”.

[Lot18] Henri Lotze. “Automated Optimization in Production Planning”.

[Nal17] Jasper Nalbach. “Embedding the Virtual Substitution in the MCSAT Framework”.

[Nal20] Jasper Nalbach. “A novel adaption of the Simplex algorithm for linear real arithetic”.

[Neu15] Lukas Neuberger. “Generation of Infeasible Subsets in Less-Lazy SMT-Solving for the Theory of Uninterpreted Functions”.

[Neu18a] Tom Neuhäuser. “Quantifier Elimination by Cylindrical Algebraic Decomposition”.

[Neu18b] Malte Neuß. “Using Single CAD Cells as Explanations in MCSAT-style SMT Solving”.

[Sal18] Ömer Sali. “Linearization Techniques for Nonlinear Arithmetic Problems in SMT”.

[Vie16] Tarik Viehmann. “Comparing different projection operators in the Cylindrical Algebraic Decomposition for SMT solving”.

[Vol15] Matthias Volk. “Using SAT Solvers for Industrial Combinatorial Problems”.

[Win16] Tobias Winkler. “Using Thom Encodings for Real Algebraic Numbers in the Cylindrical Algebraic Decomposition”.

[Zam19] Aklima Zaman. “Incremental Linearization for SAT Modulo Real Arithmetic Solving”.

Most of these thesis projects also deal with some aspects or special approaches to nonlinear problems, mostly centering around the application of cylindrical algebraic decomposition or the MCSAT approach. While [Hae18] enhanced our CAD implementation with equational constraints, variants for minimal infeasible subsets for CAD were explored in [Hen17], properties of different projection operators were compared in [Vie16], a novel method based on CAD from [ÁDE+20] was implemented in [Fra20], and a different representation for real algebraic numbers was implemented in [Win16]. Meanwhile, [Neu18a] employed this CAD implementation for quantifier elimination as described in Section 6.9.

Some were also targeted at the MCSAT approach where [Bar18] made a first attempt at implementing a Fourier–Motzkin-based explanation backend, the theory for an explanation backend based on virtual substitution was laid out in [Nal17], and an implementation of the OneCell CAD approach (see Section 8.3.2) was carried out in [Neu18b].

Several thesis projects were concerned with solving techniques unrelated to CAD or even not meant for nonlinear real arithmetic: pseudo-Boolean constraints in [Gro17], bit-vectors and their usage for nonlinear integer arithmetic in [Krü15], a solver for difference logic in [Lös18], a novel interpretation of the simplex method in [Nal20], uninterpreted functions and different variants for infeasible subsets in [Neu15], two linearization techniques based on subtropical satisfiability and reduction to linear integer arithmetic in [Sal18], and finally incremental linearization based on axiom instantiation in [Zam19; Kuk19].

Lastly, some projects aimed at applying SMT techniques to (more or less) practical applications: combinatorial system design problems in cooperation with Siemens Belgium in [Gro17], planning for robot fleets for the RoboCup Logistics Team "Carologistics" in [Kor17], production planning for Robert Bosch GmbH in [Lot18], and industrial product configuration problems for DAF Trucks N.V. in [Vol15].

1.3 Implementation

As we have already mentioned at several places, the author is currently the main author of our own software SMT-RAT [Kre+20b; CKJ+15] which is both a toolbox for and around SMT technology and an SMT solver itself. This project is closely accompanied by our library CArL [Kre+20a; KÁ18] that implements many fundamental data structures like polynomials, real algebraic numbers, and formulae as well as algorithms like resultant computation, real root isolation, interval arithmetic, Gröbner bases, or computing set coverings. Of course, CArL again relies on other libraries like Boost [Boo19], CoCoALib [AB19], Eigen3 [GJ+10] or GMP [Gt19].

Both CArL and SMT-RAT are freely available as open-source C++ software. As the border between SMT-RAT and CArL is both pretty technical and in some parts historical, but not relevant for this work, we say "SMT-RAT" to talk about the whole implementation comprised of both SMT-RAT and CArL from now on. Almost everything of what we discuss in this work has been implemented in SMT-RAT and we encourage everyone interested in implementing any of this to study our implementation.

Preliminaries

To ease the subsequent presentation and allow for a consistent notation, we give some introduction and definitions for numbers, polynomials, formulae, proof systems and show how real algebraic numbers and experimental evaluation work.

2.1 Fundamentals

We start with some fundamental definitions to set a common notation. Of course, we already build on other elementary concepts here and refer, for example, to [Art91] for definitions of algebraic concepts.

Definition 2.1: Total order

Let Ω be some set of elements. A *total order* $\leq$ on Ω is a relation that adheres to the following properties:

$$\begin{aligned} \forall a, b \in \Omega.\ (a \leq b \wedge b \leq a) &\implies a = b && \text{(antisymmetry)} \\ \forall a, b, c \in \Omega.\ (a \leq b \wedge b \leq c) &\implies a \leq c && \text{(transitivity)} \\ \forall a, b \in \Omega.\ (a \leq b \vee b \leq a) & && \text{(connexity)} \end{aligned}$$

Definition 2.2: Ring

Let R be a set of elements with binary operations $+$ and $\cdot$ such that $(R, +)$ is a commutative group, $(R, \cdot)$ is a semigroup, and the operation $\cdot$ is distributive with respect to $+$. We call $(R, +, \cdot)$ a *ring* and usually only denote it by R.
If $(R, \cdot)$ is a monoid, that is $1 \in R$, where 1 is the multiplicative identity, we call R a *ring with* 1. If $\cdot$ is commutative over R, we call R *commutative*. If $\forall a, b, c \in R.\ (a \cdot b = a \cdot c \wedge a \neq 0) \rightarrow b = c$ we call R an *integral domain*.

Recognizing that rings without 1 and non-commutative rings are interesting concepts in their own rights, all rings within this work are both *with* 1 and *commutative* and we simply call such objects *rings*.

Definition 2.3: Sets of Numbers

Let $\mathbb{B} = \{true, false\}$ be the set of Booleans, $\mathbb{N} = \mathbb{N}_0 = \{0, 1, \dots\}$ the set of *natural numbers*, $\mathbb{Z}$ the set of *integers*, $\mathbb{Q}$ the set of *rational numbers*, and $\mathbb{R}$ the set of *real numbers*. Furthermore, we denote the set of *real algebraic numbers* by $\mathcal{R}$ and refer to Section 2.5 for a definition.

Definition 2.4: Indexed sets

We write $\overline{\omega}$ for a set of elements $\{\omega_1, \ldots, \omega_n\}$ and specify n if relevant and not clear from the context.

Definition 2.5: Power sets

Let Ω be some set of elements. We define the *power set* of Ω as

$$\mathscr{P}(\Omega) = \{\omega \mid \omega \subseteq \Omega\}$$

We sometimes talk about *sequences* of certain objects, for example the *trails* of `DPLL`, `CDCL`, `CDCL(T)`, or `MCSAT`. We fix a common notation in the following Definition 2.6.

Definition 2.6: Sequences

Let Ω be some set of elements. A (finite) *sequence* M over Ω is an ordered (finite) list of elements from Ω and we write

$$M = [\![\omega_1, \ldots \omega_k]\!]$$

By abuse of notation, we may also use M as a set and write $M \subseteq \Omega$, $M_1 \subseteq M_2$, $\omega \in M$, or $|M|$. We call $M' = [\![\omega_{i_1}, \ldots, \omega_{i_l}]\!]$ a *subsequence of* $M = [\![\omega_1, \ldots \omega_k]\!]$ if $M' \subseteq M$ and $i_1 < \cdots < i_l$. Furthermore, we write $[\![M, \omega]\!]$ to *append* ω *to* M or in general $[\![M_1, \omega, M_2]\!]$ for the *concatenation* of M_1, $[\![\omega]\!]$, and M_2. We call M_1 (M_2) a *prefix (suffix) of* $M = [\![M_1, \omega, M_2]\!]$.

2.2 Polynomials

Most of this thesis centers around techniques for nonlinear real arithmetic and thus needs to work on polynomials. In the following, we give a brief introduction to polynomials and some non-trivial methods and properties.

Definition 2.7: Variables

We denote *variables* by $x_1, \ldots, x_n$ and associate every variable x_i with a *domain* D_{x_i}. If no domain is given or clear from the context, we assume the domain $\mathbb{R}$.

In many of the following ideas and algorithms, we impose a total order on variables. We usually assume the canonical order as defined below for all examples, but could plug in any arbitrary order. The importance of selecting a *good order* is discussed later, for example, in Section 6.5.1.

Definition 2.8: Variable order

A *variable order* is a *total order* on a set of variables $\overline{x} = \{x_1, \ldots, x_n\}$. We use $x_1 < \cdots < x_n$ as the *canonical order*.

We usually aim to associate every variable to a value from its domain so that this *variable assignment* satisfies our formula. We call such a mapping a *model*, following common naming in mathematical logic.

Definition 2.9: Variable assignment

Let $\overline{x} = \{x_1, \ldots, x_n\}$ be variables with domains $D_1, \ldots, D_n$. We call

$$\mathcal{A}_{\overline{x}} = \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\} \text{ with } d_i \in D_i \text{ for } i \in \{1, \ldots, n\}$$

a *variable assignment for* $\overline{x}$, denote the *(set of all) variable assignments of* $\overline{x}$ by

$$\boldsymbol{\mathcal{A}}_{\overline{x}} = \{\mathcal{A}_{\overline{x}} \mid d_1 \in D_1, \ldots, d_n \in D_n\}$$

and write $\mathcal{A}$ and $\boldsymbol{\mathcal{A}}$ if the set of variables is clear from the context. To retrieve a variable assignment for an individual variable from $\mathcal{A}$, we commonly write $(x_i \mapsto d_i) \in \mathcal{A}$ or $\mathcal{A}(x_i) = d_i$.

As the domains of our variables are usually the real numbers – and all other domains from Definition 2.3 can be embedded canonically into the real numbers – we oftentimes interpret a variable assignment for variables $\overline{x}$ as an element of $\mathbb{R}^n$, assuming some variable ordering. Thus, we can *identify* a variable assignment with a point in the n-dimensional real space.

The fundamental operational objects for most of the methods we present here are polynomials. We define polynomials over arbitrary rings, but already note that their coefficients come from $\mathbb{Q}$ for most parts of this work. We observe that most of these methods are almost exclusively interested in the roots of polynomials, and we can thus go from coefficients from $\mathbb{Q}$ to coefficients from $\mathbb{Z}$ without loss of generality (though we acknowledge that the *polynomial rings* over $\mathbb{Q}$ and $\mathbb{Z}$ are very different in nature). An example for this is given in Example 2.1.

Definition 2.10: Polynomials

Let R be some ring and x a variable. A *polynomial* with x over R has the form

$$p = \sum_{i=0}^{n} c_i \cdot x^i$$

where $\overline{c} \in R^n$ are *coefficients* from R. We define the *degree* $\deg(p) = n$ for $p \neq 0$ and $\deg(0) = -\infty$ and the *coefficients* $\operatorname{coeffs}(p) = \{c_i \mid 0 \leq i \leq n\}$ as usual. We call p *(affine) linear* if $\deg(p) = 1$ and *constant* if $\deg(p) \leq 0$. We write $R[x]$ for the *set of all polynomials* with variable x over R and call it *the polynomial ring with* x *over* R.
We oftentimes identify a polynomial p with its polynomial function $p : D_x \to D_x, x \mapsto p(x)$ (assuming $R \subseteq D_x$) to *evaluate* p at some point $x \mapsto \alpha$ with $\alpha \in D_x$ and write $p(\alpha)$.

If R is a set of numbers – usually $\mathbb{Q}$ or $\mathbb{Z}$ – we call polynomials from $R[x]$ *univariate*. However, R can itself be a polynomial ring as well, allowing for polynomials over multiple variables that we call *multivariate*. We usually denote such a polynomial ring by $R[\overline{x}]$ where R is a set of numbers and $\overline{x}$ is some set of variables. Note that the order of the variables is not relevant here, and we can thus give a standard form for multivariate polynomials that is not based on a recursive application of Definition 2.10, but uses all variables from $\overline{x}$ directly as given in the following Definition 2.11.

Definition 2.11: Multivariate polynomials

Let R be some ring and $\overline{x} = \{x_1, \ldots, x_n\}$ variables. A *multivariate polynomial* with $\overline{x}$ over R has the form

$$p = \sum_i c_i \cdot \prod_j x_j^{e_j^i}$$

where $e_j^i \in \mathbb{N}$ are *exponents* and $c_i \in R$ *coefficients*. We write $R[\overline{x}]$ for the *set of all polynomials* with variables $\overline{x}$ over R and call it *the polynomial ring* $R[\overline{x}]$. We call the summands the *terms of* p, $\operatorname{coeffs}(p) = \overline{c}$ the *coefficients of* p, and $\overline{e^i}$ the *exponent vectors* (for every term). We assume that the terms are ordered according to a variable ordering on $\overline{x}$, or rather a derived term ordering, and usually use the *degree reverse lexicographical* ordering. Under such an ordering we define the *leading term* $\operatorname{lterm}(p)$, *leading coefficient* $\operatorname{lcoeff}(p)$, and *trailing coefficient* $\operatorname{tcoeff}(p)$, as well as the *total degree* $\operatorname{tdeg}(p)$.

We note that we can directly convert polynomials between the representations given in Definition 2.10 and Definition 2.11. We consider these the *canonical representations* for univariate and multivariate polynomials, respectively. It is sometimes convenient – in particular in the context of CAD – to *univariately represent* a multivariate polynomial.

Definition 2.12: Univariately represented polynomial

Let $p \in R[\overline{x}]$ be a multivariate polynomial and $x_i \in \overline{x}$. We say that p is *univariately represented*, denoting p as specified in Definition 2.10 and its coefficients from $R[\overline{x} \setminus \{x_i\}]$ in any representation, usually as in Definition 2.11. We call x_i *the main variable of* p. We usually assume that x_i is the *largest* variable among $\overline{x}$ (under the variable ordering) and say that the *level of* p *is* $|\overline{x}|$ and write $\mathit{level}(p) = |\overline{x}|$. If $p \in R$ (that is $\overline{x} = \emptyset$) we define $\mathit{level}(p) = 0$.

Note that we have defined multiple "canonical" representations for polynomials, and there are even more ways to write polynomials that are all meaningful in their own way – all of which induce the same polynomial function and allow to convert arbitrarily from one to another. For example, $x_1^2 - 2x_1x_2 - 3x_2^2$ (as a *multivariate polynomial* in $\mathbb{Z}[x_1, x_2]$, or *distributive representation*) can also be written as $(1) \cdot x_1^2 + (-2x_2) \cdot x_1^1 + (-3x_2^2) \cdot x_1^0$ (as a *univariate polynomial* in $\mathbb{Z}[x_2][x_1]$, or *recursive representation*) or even as $(x_1 - 3x_2) \cdot (x_1 + x_2)$. These different notations allow to easily obtain different information, for example, the (total) degree or the constant part of the polynomial, the structure with respect to one "main" variable or polynomial factors.

We recognize that different representations have different trade-offs in practice and choosing a suitable representation can have a huge impact on real-world performance. Though the choice of representation is sometimes crucial for performance and the conversion between different representations is not always computationally easy, we ignore this issue for the most part of this work.

We are oftentimes interested in a *factorization* of a polynomial p, usually over $R[\overline{x}]$. As a thorough introduction into this topic is beyond the scope of this work, we use notions like the *factorization* p, *irreducible normalized factors of* p ($\operatorname{factors}(p)$), p being *square-free* and the *square-free part of* p, the *content of* p ($\operatorname{cont}(p)$) and the *primitive*

part of p ($\text{prim}(p)$), the *(finest) square-free basis* and the *(finest) irreducible basis* of sets of polynomials without definition and refer to any of [Art91; GCL92]. Another important property of polynomials (at least in the context of this work) are their *real roots*, intuitively the variable assignments that make a polynomial evaluate to zero.

Definition 2.13: Real roots

Let $p \in R[\overline{x}]$ be a polynomial. We call $\text{roots}(p) \coloneqq \{\alpha \in \mathbb{R}^n \mid p(\alpha) = 0\}$ the *real roots of* p. If $p \neq 0$ is univariate we observe that $|\text{roots}(p)| \leq \deg(p)$.

In fact, we are solely interested in the *real roots* of polynomials in most parts of this work. This allows us to not only rewrite polynomials but actually simplify them by normalizing the coefficients or removing multiple factors and thereby multiple roots as we show in the following Example 2.1.

Example 2.1: Simplifying polynomials

Let $p = -4x^3 + 12x - 8$ be a polynomial and we assume that we are only interested in the real roots of p. The following derivations show how we can use coefficient normalization and the removal of multiple factors to safely replace this polynomial by $x^2 + x - 2$:

$$\begin{aligned} & -4x^3 + 12x - 8 = -4 \cdot (x^3 - 3x + 2) \\ \implies \quad & \text{roots}(-4x^3 + 12x - 8) = \text{roots}(x^3 - 3x + 2) \\ & x^3 - 3x + 2 = (x-1)^2 \cdot (x+2) \\ \implies \quad & \text{roots}(x^3 - 3x + 2) = \text{roots}((x-1) \cdot (x+2)) \\ & (x-1) \cdot (x+2) = x^2 + x - 2 \end{aligned}$$

We finally define some operations on polynomials that are used, in particular, for the projection operators in Section 5.2: *reducta*, *resultants*, and *discriminants*.

Definition 2.14: Reducta and reducta sets

Let $P \subseteq R[x]$ be a set of polynomials and $p \in P$. We define

$$\begin{aligned} red(p) &\coloneqq p - \text{lterm}(p) \\ red^0(p) &\coloneqq p \\ red^k(p) &\coloneqq red(red^{k-1}(p)) \\ RED(p) &\coloneqq \{red^k(p) \mid 0 \leq k \leq \deg(p)\} \\ RED(P) &\coloneqq \bigcup_{p \in P} RED(p) \end{aligned}$$

We call $red(p)$ the *reductum of* p, $red^k(p)$ the *k'th reductum of* p, $RED(p)$ the *reducta set of* p, and $RED(P)$ the *reducta set of* P.

All further components of the projection operators that we define later rest on *resultants* and various related (or rather derived) concepts: *subresultants*, *principal (sub-)resultant coefficients*, and *discriminants*. We only give brief definitions and refer to relevant literature on this topic like [BPR10] or [GCL92].

Definition 2.15: Resultant

Let $p, q \in R[x]$ be univariately represented polynomials with $\deg(p) = n$ and $\deg(q) = m$. The *resultant* of p and q is the determinant of the Sylvester matrix:

$$\operatorname{res}_x(p,q) = Det \begin{pmatrix} p_n & \cdots & p_0 & & & \\ & \ddots & & \ddots & & \\ & & \ddots & & \ddots & \\ & & & p_n & \cdots & p_0 \\ q_m & \cdots & q_0 & & & \\ & \ddots & & \ddots & & \\ & & \ddots & & \ddots & \\ & & & \ddots & & \ddots \\ & & & & q_m & \cdots & q_0 \end{pmatrix}.$$

If p and q are univariate, or the main variable x is clear from the context, we usually write $\operatorname{res}(p,q)$ instead of $\operatorname{res}_x(p,q)$.

Note that other variants of the Sylvester matrix exist, mostly obtained by permutations of rows and columns, which sometimes have certain computational benefits. Another type of matrix is the *Bézout matrix*, whose determinant also turns out to be the resultant of two polynomials. The Sylvester matrix also allows to compute *subresultants* as the determinants of certain submatrixes.

In practice, however, resultants and subresultants are usually not computed using determinants, but using *subresultant pseudo-remainder sequences*. These sequences allow to iteratively compute all subresultants and ultimately the resultant, which is commonly considered the 0th element of this sequence. Our implementation, for example, uses a variant of what is presented in [Duc00].

In contrast to the definition of resultants, we usually compute a simplified version of the resultant. As we are only interested in the roots of the resultant, we can safely make it square-free and remove any constant (or non-vanishing) factors. We also use these simplifications whenever we use resultants throughout this work, hence the polynomials that result from resultant computations are not necessarily identical to the resultants one computes with other systems like Maple.

As the definition of subresultants and principal subresultant coefficients is very technical, we refer to [GCL92; Duc00] for exact characterizations and denote the set of all *principal subresultant coefficients* of two polynomials p, q by $PSC(p,q)$ without definition.

Apart from this rather technical and unintuitive definition, a very instructive characterization for resultants is as follows. Let p, q be polynomials of degree d and e with roots α_i and β_j respectively, then we have

$$\operatorname{res}(p,q) = \operatorname{lcoeff}(p)^e \cdot \operatorname{lcoeff}(q)^d \cdot \prod_{\substack{1 \le i \le d \\ 1 \le j \le e}} (\alpha_i - \beta_j)$$

in an algebraic closure of the according polynomial ring, as the resulting polynomial might not factor into linear factors in the polynomial ring itself.

If we assume that p and q are univariate polynomials, then α_i and β_j are algebraic numbers and the resultant is zero if and only if p and q have a common root. If we instead let p and q be multivariate polynomials in some main variable, α_i and β_j themselves contain the remaining variables, that one could consider them *parameters*. Now, the resultant has a root – in the remaining variables – wherever any α_i and β_j coincide (or *meet*). Hence the resultant can be seen as an indicator for intersections of the root surfaces of multivariate polynomials.

While resultants provide indicators for common roots of two polynomials, we are also sometimes interested in multiple roots of a single polynomial. Multiple roots can be recovered using the (normalized) resultant of a polynomial and its derivative which is commonly called the *discriminant*.

Definition 2.16: Discriminant

Let $p \in R[x]$ be a univariately represented polynomial with $\deg(p) = n$. The *discriminant* of p is defined as follows in terms of the resultant of p and its partial derivative with respect to x:

$$\operatorname{disc}_x(p) = (-1)^{\frac{n\cdot(n-1)}{2}} \cdot \frac{\operatorname{res}_x(p, p')}{\operatorname{lcoeff}(p)}$$

2.3 First-order logic

We now depart from the algebraic world of polynomials into the realm of logic. Within this work, the basic logical objects – except for Boolean variables and constants – are *(arithmetic) constraints*, consisting of a polynomial and a *sign condition*.

Definition 2.17: Sign condition

We define a *sign condition* to be a function from a *sign* (negative, zero or positive) to a Boolean value, formally $\sigma : \{-, 0, +\} \to \mathbb{B}$. Except for the trivial sign conditions $\cdot \mapsto \mathit{true}$ and $\cdot \mapsto \mathit{false}$, sign conditions are what we commonly know as *relations* $\{<, \leq, =, \neq, \geq, >\}$.
We usually write $p\,\sigma\,0$ instead of $\sigma(\operatorname{sgn}(p))$ – where $\operatorname{sgn} : \mathbb{R} \to \{-, 0, +\}$ denotes the *sign function* – to allude to the intuition of σ being a relation.

We observe that $p\,\sigma\,p'$ can equivalently be rewritten to $p - p'\,\sigma\,0$ and thus this formalism of sign conditions can be used to define what we commonly know as *polynomial (in-)equalities*. For what we call *constraints*, we use a slightly more generic definition, allowing for other "arithmetic expressions" than polynomials.

Definition 2.18: Constraint

Let p be an arithmetic expression and σ a sign condition. We call the pair (p, σ) an *(arithmetic) constraint* with the semantics of $p\,\sigma\,0$. We call p its *left-hand side* and σ its *relation*.

Note that we can determine the truth value of a constraint in multiple ways. If its left-hand side is a numeric constant – or we can evaluate it to one using a variable assignment – we can simply compare it to zero. If it is not, though, we can sometimes

evaluate it anyway using further knowledge about the left-hand side: for example, we know that $x^2 + 1$ is strictly positive and thus $x^2 + 1 > 0$ is true.

We have only seen polynomials as arithmetic expressions that could be used as left-hand sides of constraints, however, we trust the reader to know further arithmetic functions like square roots, cosines, or logarithms. While polynomials are sufficient for most of this work, we eventually introduce another variant in Section 8.3.1 that compares variables against the *kth root of some polynomial*.

Based on constraints, we can finally construct more complex logical objects that we call *formulae*. Formulae are in some sense *the language of mathematics*, allowing to concisely formulate statements and problems in a very general way. We classify formulae into *logics* with well-defined rules for syntax and semantics. A large variety of logics exists for different (practical or theoretical) purposes. While some logics consider only Boolean constructs – Boolean variables and Boolean connectives like $\vee$ or $\wedge$ – most involve some kind of *theory* $\mathcal{T}$. The theory adds *predicates* or *theory atoms* – in our case constraints – that in some way allow to map *theory expressions* to the Boolean level. Some logics add *quantifiers* like $\exists$ or $\forall$ while sometimes more specialized operators are used (for example, in LTL).

This work mostly considers *first-order logics*, which are the focus of *satisfiability modulo theories* solving. Apart from the standard Boolean structure, it allows for theory atoms as specified before and quantification over the theory variables from these theory atoms.

Definition 2.19: First-order logic

Let $\mathcal{T}$ be some theory with variables $\overline{x}$ and *theory atoms* $\mathcal{A}$, $\mathcal{B}$ the set of Boolean variables, and $\mathbb{B} = \{\mathit{false}, \mathit{true}\}$ the Boolean constants. A *first-order logic formula* φ is one of the following:

$$\begin{aligned} &\varphi ::= b \mid c \mid p && b \in \mathcal{B}, c \in \mathbb{B}, p \in \mathcal{A} \\ &\varphi ::= \varphi \vee \varphi \mid \neg\varphi \\ &\varphi ::= \exists x.\ \varphi \end{aligned}$$

We call Boolean constants, Boolean variables, and theory atoms *first-order logic atoms* and write $\Phi_{\mathcal{T}}$ for the set of all such first-order logic formulae.

We note that $\vee$ and $\neg$ are what we call *functionally complete* and can be used to construct all other Boolean connectives like $\wedge$ (and) and $\oplus$ (xor) as syntactic sugar. Similarly, $\forall x$ (universal quantification) can be constructed from $\exists x$ and $\neg$.

Throughout this work, we oftentimes manipulate or relate formulae and sometimes need to distinguish explicitly between *syntactic* operations and *semantic reasoning*. We usually employ semantic reasoning to argue about the actual meaning of a formula instead of how we write it down as this allows us to pass over certain technical details.

Definition 2.20: Semantic deduction

Let $\overline{x}$ be variables, φ and φ' formulae, and $\mathcal{A}$ an assignment over $\overline{x}$. If φ evaluates to *true* when $\overline{x}$ is substituted by $\mathcal{A}$, we say that $\mathcal{A}$ *satisfies* φ and write $\mathcal{A} \models \varphi$. We say that φ' *follows from* φ, denoted by $\varphi \models \varphi'$, if

$$\forall \mathcal{A} \in \boldsymbol{\mathcal{A}}.\ \mathcal{A} \models \varphi \implies \mathcal{A} \models \varphi'.$$

It is oftentimes convenient to assume some normal form for first-order logic formulae. Throughout this thesis – unless stated otherwise – we assume all first-order logic formulae to be in *prenex normal form* and the quantifier-free part to be in *conjunctive normal form* (which implies *negation normal form*) and simply call them "formulae".

Definition 2.21: Prenex normal form

A formula φ is in *prenex normal form* if it has the form

$$\varphi ::= Q^1_{x_1} \dots Q^n_{x_n} \varphi'$$

with quantifiers $Q^i \in \{\exists, \forall\}$ and φ' containing no quantifiers. We call φ' the *quantifier-free part of* φ.

If φ has multiple *identical subsequent* quantifiers, we allow to combine them into a single *quantifier block* and write $Q^1_{x_1,\dots,x_k} \varphi'$. We observe that we can arbitrarily reorder variables within such a quantifier block without changing the semantics of φ.

A given formula can be transformed to prenex normal form by pushing all quantifiers towards the front of the formula, taking care that 1. the relative order of quantifiers does not change, 2. variables are renamed if they are referenced by multiple quantifiers, and 3. quantifiers are properly changed if pushed over a negation.

Definition 2.22: Negation normal form

A (quantifier-free) formula is in *negation normal form* if negations only occur immediately before *atoms*.

Similar to how we converted formulae to prenex normal form, we can transform any (quantifier-free) formula to negation normal form by pushing all negations inside towards the atoms using De Morgan's laws.

Definition 2.23: Conjunctive normal form

A (quantifier-free) formula is in *conjunctive normal form* if it is a conjunction of disjunctions of (possibly negated) atoms. We write

$$\varphi ::= \bigwedge_i \bigvee_j a_{ij}$$

where a_{ij} are atoms or negated atoms. We call a_{ij} *literals* and $\bigvee_j a_{ij}$ *clauses*, identify a clause with its corresponding set of literals, and a formula with the corresponding sets of its clauses, allowing for notations like $a_{ij} \in c \in \varphi$.

It is rather easy to see that any (quantifier-free) formula can be converted to conjunctive normal form. One possibility is to 1. negate it, 2. convert to *disjunctive normal form* with a Quine-McCluskey method as in [McC56], 3. negate it again, and 4. bring it to negation normal form. Unfortunately, this method can produce exponentially large formulae – just as for *disjunctive normal form*.

Tseitin found a way around this in [Tse68] by introducing fresh Boolean variables, devising a possibility to construct *equisatisfiable* formulae that are only linearly larger. The fundamental idea is to replace every *subformula* by a fresh Boolean variable and

require equivalence of the subformula and the Boolean variable. Instead of a deeply nested formula, we obtain a conjunction of very shallow subformulae of constant size which can then be transformed to conjunctive normal form individually.

Definition 2.24: Tseitin's transformation

Let φ be a (quantifier-free) formula and $t(\phi) = x_\phi$ for every subformula ϕ of φ where x_ϕ are fresh (Boolean) variables. The result of *Tseitin's transformation* is $\varphi' \wedge t(\varphi)$ where φ' is constructed inductively as follows:

$$\begin{aligned}
&\varphi ::= b \mid c \mid p && \varphi' := t(\varphi) \leftrightarrow \varphi && \\
&\varphi ::= \varphi_1 \vee \varphi_2 && \varphi' := t(\varphi) \leftrightarrow (t(\varphi_1) \vee t(\varphi_2)) && \wedge \varphi_1' \wedge \varphi_2' \\
&\varphi ::= \neg\varphi_1 && \varphi' := t(\varphi) \leftrightarrow \neg t(\varphi_1) && \wedge \varphi_1'
\end{aligned}$$

We note that the resulting formula is technically not yet in conjunctive normal form, but a conjunction of subformulae of constant size. Transforming every subformula individually with a generic method as described above yields a formula in conjunctive normal form with only linear overhead.

We have defined first-order logic for arbitrary theories and shown how to convert arbitrary formulae to conjunctive normal form. Throughout this work, we focus on first-order logic with *nonlinear real arithmetic*, commonly defined as real variables with addition, multiplication, and comparisons as defined in Definition 2.25.

Definition 2.25: Nonlinear real arithmetic

Let $\overline{x}$ be a set of real-valued variables and $=, <$ the relations with the usual semantics. We define terms and predicates of *nonlinear real arithmetic* as follows:

$$\begin{aligned}
t &::= 0 \mid 1 \mid x \in \overline{x} \mid t + t \mid t \cdot t \\
p &::= t = t \mid t < t
\end{aligned}$$

The predicates from Definition 2.25 nicely coincide with the constraints from Definition 2.18 where arithmetic expressions are restricted to polynomials.

Note that we use the term *nonlinear real arithmetic* for both the arithmetic theory and the first-order logic over said theory. We use the following other theories analogously without explicit definition as well: *nonlinear integer arithmetic* ($\overline{x}$ ranges over $\mathbb{Z}$ instead of $\mathbb{R}$), *linear real arithmetic* (without $t \cdot t$ or, equivalently, restricted to linear polynomials), and *linear integer arithmetic* (both over $\mathbb{Z}$ and only linear).

While a formula has many interesting properties, we mainly focus on the question of *satisfiability*. In this work, we consider the satisfiability of *quantifier-free* formulae where all variables are implicitly quantified existentially. We observe that checking for satisfiability can be used to derive *validity* as well by checking whether the negation of a formula is unsatisfiable.

Definition 2.26: Satisfiability

We call a quantifier-free formula φ *satisfiable* if it has a variable assignment $\mathcal{A}$ such that $\mathcal{A} \models \varphi$. If no such $\mathcal{A}$ exists then φ is *unsatisfiable* and thus $\varphi \equiv \mathit{false}$. We call $\mathcal{A}$ *a model for* φ.

2.3.1 Extended polynomial constraints

While we generally operate in the realm of first-order logic as defined before, we sometimes need to extend it a bit to allow for more expressive statements. Note that the constraints we are about to present are not *more expressive* in the strict logical sense – they can be formulated equivalently within first-order logic – but allow for a much shorter and more concise formulation.

We want to state that some variable is smaller than some root of a polynomial at several places in this work, for example, in Section 6.9 and Section 8.3. In particular, we want these polynomials to be multivariate and evaluate these expressions with respect to some theory model. We call such constraints *extended polynomial constraints* and define them as follows, mostly following the corresponding definition in [JM12] or what is called an *indexed root expression* in [Bro99].

Definition 2.27: Extended polynomial constraints

Let $p \in \mathbb{Q}[z, \overline{x}]$ be a polynomial, v some variable, and σ a sign condition. We call $v\,\sigma\,\mathrm{root}(p,k)$ an *extended polynomial constraint* where $\mathrm{root}(p,k)$ refers to the *kth root of* p *in* z (for fixed values of $\overline{x}$), v *the left-hand side* and $\mathrm{root}(p,k)$ *the root expression* of this extended polynomial constraint.
To evaluate $v\,\sigma\,\mathrm{root}(p,k)$ we first compute the kth root of p with respect to a model $\mathcal{A}$. If the number of roots is smaller than k, the result is *false*, otherwise the result is $v\,\sigma\,\alpha_k$ where α_k is the kth root of p.

Note that this definition fails to give proper semantics for the evaluation, in particular, if p does not become univariate in z over $\mathcal{A}$. Our intuition is, that such a constraint is a *constraint on* v and that it *does not apply yet*, much like a regular constraint does not give a Boolean result if it does not fully evaluate.

One special case that exists, however, is when v is already assigned in $\mathcal{A}$ but some $x_i \in \overline{x}$ is not. The constraint then intuitively becomes a *constraint on* x_i, though it is not clear at all how to deal with this case. Whether this particular situation actually occurs and how we can deal with it depends on how we construct and use such constraints, and we thus defer this discussion to Section 8.4.1.

2.4 Deductive proof systems

A popular framework to perform derivations in a systematic way are (deductive) *proof systems*. A proof system is a set of *proof rules* that allow deriving new statements from a set of assumptions to eventually obtain a (logically sound) proof in the form of a derivation sequence from the *initial assumptions* Φ to the *target statement* Ψ. We later consider proof systems that argue about formulae and are at least *sound*: the syntactic transformation is logically sound.

The proof rules are specified so that they can be applied *syntactically* and thus allow to construct a *semantic* proof using only *syntactic* operations. As we are mostly interested in the satisfiability of formulae, we usually have $\Psi = \mathit{false}$. We start with the definition of a single proof rule.

Definition 2.28: Deductive proof rule

Let $\overline{\varphi} = \{\varphi_1, \ldots, \varphi_n\}$ and $\overline{\psi} = \{\psi_1, \ldots, \psi_m\}$ be sets of formulae. We call $PR(\overline{\varphi}, \overline{\psi})$ a *proof rule*, commonly write

PR:

$$\frac{\varphi_1, \ldots, \varphi_k}{\psi_1, \ldots, \psi_m} \qquad \textbf{if} \quad \varphi_{k+1}, \ldots, \varphi_n$$

for the proof rule itself, and call $\overline{\varphi}$ the *premises* and $\overline{\psi}$ the *conclusions*. We sometimes split the premises into two parts to simplify the presentation and call $\{\varphi_{k+1}, \ldots, \varphi_n\}$ the *side conditions*.
The proof rule PR is *applicable to a set of formulae* Φ if $f(\overline{\varphi}) \subseteq \Phi$ where f may rename variables from $\overline{\varphi}$ appropriately. The result of the *application of* PR to a set of formula Φ is $\Phi \wedge f(\bigvee \overline{\psi})$ and we write $\Phi \vdash_{PR} \Phi \wedge f(\bigvee \overline{\psi})$, though we usually omit f to simplify notation. We call $\Phi \vdash_{PR} \Psi$ a *derivation* and oftentimes write $\Phi \vdash \Psi$ if PR is either irrelevant or clear from the context.

The separation of the premises into two sets is purely to simplify the notation and we can move individual formulae into or out of the side conditions without changing the meaning. We use this distinction (intuitively) to describe *what the rule does* in $\{\varphi_1, \ldots, \varphi_k\}$ and *when it is applicable* in $\{\varphi_{k+1}, \ldots, \varphi_n\}$.

Though we defined applicability as a purely syntactic property – $\overline{\varphi} \subseteq \Phi$ up to renaming – we usually allow f to perform some easy logical reasoning, like conversion to a normal form or deriving immediate corollaries from Φ, as well. In particular, side conditions are usually not part of the formula but can be easily inferred from it. We understand proof rules as (almost) syntactic rewriting operators that hide a (more or less) complex reasoning such that it can be automated.

Applying a proof rule requires a *conjunction of premises* but results in a *disjunction of conclusions*, severely limiting the applicability of proof rules to the result of a previous application. Thus, proof rules usually have only a single conclusion – all proof rules in this work have this form – and thereby keep the formula in a conjunctive form.

The above definition of a proof rule poses no restriction on the *logical soundness* of the derivation. However, one can very well argue that a proof rule whose conclusion *does not logically follow* from the assumptions is not particularly useful. We thus assume all proof rules in this work to be sound in the sense of the following Definition 2.29.

Definition 2.29: Soundness of a proof rule

We say a proof rule $PR(\overline{\varphi}, \overline{\psi})$ is *sound* if it only allows for *valid derivations*:

$$\forall \overline{\varphi}. \left(\bigwedge \overline{\varphi} \vdash \bigvee \overline{\psi}\right) \implies \left(\bigwedge \overline{\varphi} \models \bigvee \overline{\psi}\right)$$

We now combine one or more proof rules into a *proof system*, which can roughly be seen as the equivalent of an algorithm to manipulate formulae. Note, though, that a proof system only provides certain operations – the proof rules – but leaves it to the "user" when to *apply* which proof rule to which formulae. To obtain an actual *proof*, one thus has to provide what some would call a *scheduler* for a given proof system.

Definition 2.30: Deductive proof system

We call a set $\boldsymbol{\mathcal{P}}$ of proof rules a *(deductive) proof system*.

Throughout this work, we use two arguably different interpretations of proof systems, though they both match the general definition from above. The first one operates on a *set of formulae* and progresses by selecting a subset of them and applying a proof rule to this subset to derive a new formula that is added to the (global) set of formulae. The most prominent proof system within this work – the resolution proof system as shown in Definition 3.3 – is of this type.

The second variant more closely resembles an algorithm in that it merely manipulates a single formula, usually discarding previous "versions" of this formula. Intuitively, we think of this formula as our *state* and for a proof rule the premises $\overline{\varphi}$ describe (or rather unpack) the state while the side conditions $\overline{\varphi'}$ contain conditions on the state, restricting the applicability of the proof rule. Formally, we can enhance the underlying theory with a new predicate for the state and thereby adhere to the above definition of a proof rule. This is only a technicality, though, and we, therefore, avoid it here.

It is sometimes convenient, in particular when we extend an existing proof system, to *compose* multiple proof rules into a single one, thereby restricting how the proof system is allowed to operate. This is no fundamental extension to the definition of proof systems: we can again enhance our underlying theory with predicates that represent *intermediate states*, thereby enforcing this composition completely within the previously defined framework.

Definition 2.31: Composition of deductive proof rules

Let $PR_1(\overline{\varphi_1}, \overline{\psi_1})$ and $PR_2(\overline{\varphi_2}, \overline{\psi_2})$ be two proof rules that are "chainable":

$$\forall \Phi.\ (PR_1 \text{ is applicable to } \Phi \implies PR_2 \text{ is applicable to } \Psi \text{ where } \Phi \vdash_{PR_1} \Psi)$$

We call $PR(\overline{\varphi_1}, \overline{\psi_2})$ the *composition of* PR_1 *and* PR_2 and write $PR = PR_1 \circ PR_2$.

For a proof system as defined here, the initial formula essentially already contains the "question" the proof is supposed to answer. As every application of a proof rule yields a logically valid deduction, we can only *steer* a proof system – by applying different proof rules – but not change the *destination* – assuming that the logic is consistent and all proof rules are sound. In this sense, applying proof rules is only a way to *discover* the right pieces of information that allow deriving the final result, one reasonably easy step at a time.

However, we sometimes want to logically change the formula the proof system is working on. Our main intention is to work on a sequence of formulae without completely *restarting* the proof system, for example, because we can reuse some partial computations. In order to do this, we allow for proof rules to take *external inputs*.

Definition 2.32: Deductive proof rule with external input

Let $PR(\overline{\varphi} \cup \{i\}, \overline{\psi})$ be a proof rule with $\overline{\varphi} = \{\varphi_1, \ldots, \varphi_n\}$, $\overline{\psi} = \{\psi_1, \ldots, \psi_m\}$ and an additional input i. We write

PR i**:**

$$\frac{\varphi_1, \ldots, \varphi_k}{\psi_1, \ldots, \psi_m} \qquad \textbf{if} \quad \varphi_{k+1}, \ldots, \varphi_n$$

where both $\overline{\varphi}$ and $\overline{\psi}$ can make use of the input i.

Note that we did not restrict the input to be "logical" but instead intend it to be rather "operational". If i would always be a "logical" information (a formula) the only reasonably intuitive thing to do would be to *replace* $\overline{\varphi}$ *by* $\overline{\varphi} \wedge i$ and continuing from there. This however only allows to "add to the formula", but never "remove from it".

By "operational" we instead mean that i can be an *instruction* like "assume that φ holds" or analogously "remove the assumption that φ holds" – which is fundamentally different from "assume that $\neg\varphi$ holds". Again we can formally add such operational statements as new predicates to our theory, bringing this extension in line with the previous definitions. It may not be immediately clear how this can be useful yet, and we put the reader off until Section 6.2 on this issue.

Definition 2.33: Deductive proof

Let $\boldsymbol{\mathcal{P}} = \{PR_1, \ldots, PR_n\}$ be a proof system, Φ an *initial formula*, and Ψ a *target formula*. A *proof* $\mathcal{P}$ for Φ and Ψ consists of a sequence of proof rule applications that derives Ψ from Φ:

$$\mathcal{P} := \Phi \vdash_{PR_{i_1}} \cdots \vdash_{PR_{i_k}} \Psi$$

with $i_1, \ldots, i_k \in \{1, \ldots, n\}$, allowing to repeatedly apply proof rules in arbitrary order. We denote the *set of all proofs* for a proof system by $\boldsymbol{\mathcal{P}}^{\vdash}$ and call the number of proof rule applications k the *length of* $\mathcal{P}$ and write $|\mathcal{P}| = k$.

Finally, we should discuss two fundamental properties of (many) proof systems: *soundness* and *completeness*. As for individual proof rules, soundness ensures that the derivations within a proof system are logically sound and thereby all (syntactic) proofs from $\boldsymbol{\mathcal{P}}^{\vdash}$ are logically valid. Again, we argue that a proof system that is not sound is not particularly useful and thus all proof systems in this work are sound (which technically already follows from the analogous assumption for proof rules).

Completeness, on the other hand, indicates whether a proof system can produce a proof *for every* $\Phi \models \Psi$ that is valid within our logic. As a proof merely combines syntactic rewriting steps based on the proof rules provided by the proof system, it becomes clear that the proof rules must be carefully crafted such that these *syntactic* operations can be combined to prove all *logical* statements. As we usually only consider the question of satisfiability, it is mostly sufficient for a proof system to be *refutationally complete* where we can produce a proof *for every* $\Phi \models \mathit{false}$.

Definition 2.34: Soundness & Completeness

We say that a proof system $\boldsymbol{\mathcal{P}}$ is *sound* if all of its proofs are *logically valid*:

$$\forall \mathcal{P} \in \boldsymbol{\mathcal{P}}^{\vdash}.\ (\mathcal{P} = \Phi \vdash \cdots \vdash \Psi) \implies (\Phi \models \Psi)$$

If all proof rules of $\boldsymbol{\mathcal{P}}$ are sound, we immediately get soundness of $\boldsymbol{\mathcal{P}}$ as every

step of every proof $\mathcal{P}$ is sound. We say that a proof system $\boldsymbol{\mathcal{P}}$ is *complete* if it allows to construct a proof for every valid $\Phi \models \Psi$:

$$(\Phi \models \Psi) \implies \exists \mathcal{P} \in \boldsymbol{\mathcal{P}}^{\vdash}.\ \mathcal{P} = \Phi \vdash \cdots \vdash \Psi$$

Moreover, we say that a proof system $\boldsymbol{\mathcal{P}}$ is *refutationally complete* if it allows to construct a proof for every valid $\Phi \models \mathit{false}$:

$$(\Phi \models \mathit{false}) \implies \exists \mathcal{P} \in \boldsymbol{\mathcal{P}}^{\vdash}.\ \mathcal{P} = \Phi \vdash \cdots \vdash \mathit{false}$$

2.5 Real algebraic numbers

When computing with numbers in an exact way, we usually think of integers, rationals, or reals. While there are decent libraries for arbitrarily large integers and rationals, for example [Gt19], we should not expect to deal with the real numbers in a similarly constructive way.

One common approach is to trade in provable correctness (by computing with an exact representation) for efficiency by using floating-point numbers that usually come with hardware support following [IEE08]. Throughout this thesis, and when provable correctness is strictly required, this is oftentimes not sufficient as this floating-point representation comes with a limited precision that may introduce rounding errors. Furthermore, practical experience shows that these rounding errors accumulate to be significant during the computation and we thus need some exact way to represent and compute with real numbers.

It is possible to exploit the performance of floating-point numbers by representing a real number by an interval that contains it and ensure that all operations on this interval are *over-approximating*. This is common for certain applications, for example in *interval constraint propagation* (ICP), and efficient implementations of such intervals exist that take care of appropriate rounding, for example, Boost interval [Boo19]. They work well if the intervals can be refined regularly – either because this is how the method works anyway, like for ICP, or because we have additional *symbolic* information – but otherwise tend to accumulate rounding errors.

Fortunately, we do not require the full set of real numbers in this work because the ways how numbers "emerge" in our scenario are limited. As our input is always finite, numbers can (exactly) be represented as rationals. We then compute with these numbers using what ultimately are only basic arithmetic operations most of the time. The only exception is when we compute the real roots of polynomials (over the rationals) by real root isolation as presented in the previous section. The set of such roots (of polynomials over the rationals) is called the *real algebraic numbers* and covers all reals except for the *transcendental* numbers – like π or Euler's number e.

Definition 2.35: Real algebraic numbers

Let $\alpha \in \mathbb{R}$ be a real number. We call α a *real algebraic number* if there is a polynomial $0 \neq p \in \mathbb{Q}[\xi]$ with $p(\alpha) = 0$. We denote the set of all *real algebraic numbers* by $\mathcal{R}$ and note that $\mathbb{Q} \subsetneq \mathcal{R} \subsetneq \mathbb{R}$.

We are commonly used to *explicit numeric* representations for integers, typically in binary, and rationals – given by explicit numerators and denominators. Therefore, we usually make no conceptual difference between the meaning of a number and its representation. Though representations like $\sqrt{2}$ might suggest that it is just as easy for real algebraic numbers, there oftentimes is no (finite) *explicit numeric* representation, but only a *symbolic declarative* one. We thus recognize that we can not *just compute* with real algebraic numbers. Though they are (almost) seamlessly integrated within software packages like Maple or Mathematica, dealing with real algebraic numbers requires a significant amount of algebraic machinery in the background.

At this point, we have two possibilities: we can consider real algebraic numbers an implementation detail and assume our environment to "just support" them, for example, if we use a full-fledged computer algebra system like Maple, Mathematica, or alike. If this is the case, the reader may want to skip to the end of this section. The presented implementation, however, did not take place within such a system – for various reasons – and the proper implementation of real algebraic numbers was a major concern.

Consider for example $\sqrt{2}$ where it is well-known that the sequence of decimal places is infinite and non-repeating. We have a concise symbolic representation – commonly written $\sqrt{2}$ – which only bears little information on how to work with this number, though. Can we easily determine whether we have some $q \in \mathbb{Q}$ such that $q \cdot \sqrt{2} = \sqrt{8}$? Can we still do the same for $\sqrt{3+\sqrt{3}}$ and $\sqrt{3-\sqrt{3}}$?

To resolve these issues, we introduce a representation for real algebraic numbers that is both *symbolic* and *numeric*, allowing us to work both symbolically and numerically, whatever is suitable or more efficient for the desired operation. The symbolic part is a polynomial with one root being the represented number while the numeric part is an interval containing exactly one of the roots of the polynomial. Though alternative representations exist as well, and we mention them briefly in Section 2.5.4, we consider this "the" representation of a real algebraic number for the purpose of this thesis.

Definition 2.36: Representation of real algebraic numbers

Let $\alpha \in \mathcal{R}$, $0 \neq p \in \mathbb{Q}[\xi]$, and I be a real interval that is either open ($I = (\underline{\alpha}, \tilde{\alpha}) \subseteq \mathbb{R}$) or a point interval ($I = [\alpha, \alpha]$). We say that (p, I) *represents* the real algebraic number α if $\text{roots}(p) \cap I = \{\alpha\}$. In this case, we call p a *polynomial of* α and I an *isolating interval of* α.

Note that we oftentimes *identify* α with (p, I) and thus say that (p, I) *is* a real algebraic number, though neither the polynomial nor the interval is unique for a real algebraic number α. Although this is not strictly necessary, we usually make sure that p is *irreducible* and *normalized*, making it the *minimal polynomial* of α (and thereby unique). A proper definition of these terms follows shortly in Insertion 2.5.2.

We have different options for how to store the interval endpoints, for example, floating-point numbers or arbitrary rational numbers. While the former may be much faster, it ultimately suffers from its limited precision. We assume the interval endpoints to be rationals with arbitrary precision (which works in general as $\mathbb{Q}$ is dense in $\mathbb{R}$) and thus, the isolating interval may only be a point interval if $\alpha \in \mathbb{Q}$, but is necessarily an open interval if $\alpha \in \mathcal{R} \setminus \mathbb{Q}$. For a point interval, we can furthermore extract the (precise) rational assignment and eventually replace the above representation by a rational to easily exploit this information in all subsequent operations.

Before we discuss the operations we need to perform on real algebraic numbers and how to implement them, we look at how we obtain them in the first place. As already mentioned before they arise when we compute the real roots of polynomials.

2.5.1 Real root isolation

Many methods that argue about properties of polynomials – in particular those that this thesis is concerned with – employ one fundamental building block: they need to distinguish the different real roots of a polynomial. While some purely symbolic methods exist that we discuss later as well, it has proven to be very beneficial in practice to work with some explicit approximation of a real root, which is usually maintained as an *isolating interval* that contains the real root.

Definition 2.37: Real root isolation

Let $p \in \mathbb{Q}[x]$ be a univariate polynomial. A *real root isolation of* p is a set that contains a single isolating interval for every real root of p.

As an isolating interval needs to contain the real root – and not only be a *good* approximation – standard numerical techniques for real root finding are not immediately applicable with similar arguments as discussed in Section 1.1.6.2. Attempts to apply these techniques anyway can be found in [Kre13], though with arguably modest success.

As numerical methods are difficult to apply – and this problem in some form even predates modern numerical analysis methods – it has sparked various specialized methods. Research on this specific problem includes Descartes' rule of signs [Des37] and Sturm's theorem [Stu29], but also more recent ones like [Hei70; CA76; Sag12].

They all have in common that some algebraic technique is used to count (in the case of Sturm's theorem) or at least provide an upper bound (in the case of Descartes' rule of signs) on the number of real roots within a given interval. Given an initial interval, we can subdivide the interval until every interval contains at most a single real root.

As for the initial interval, we have methods to compute an upper bound on the (absolute) value of any real root – for example [Cau28; Fuj16; HM97] or [Lag08; MS02]. If any of those yields an upper bound a, then we can perform such a bisection approach with the initial interval $[-a, a]$ and obtain isolating intervals for all real roots. An abstract algorithm that employs these building blocks – and may very well be extended with various optimizations – is sketched in Algorithm 2.1.

While the correctness immediately follows from the ability to correctly compute an initial interval and safely answer the question whether the interval has no or only a single real root, termination can be a bit more involved if we can not exactly count the number of real roots – as is the case for Descartes' rule of signs that only yields an upper bound. We refer to the literature already cited above for details on this topic. Furthermore, we need to employ a sensible splitting heuristic, for example, we should only split into finitely many intervals and not allow for infinitesimally small intervals.

Note that we oftentimes need to take special care of the interval endpoints, as the methods to *count* the real roots within an interval usually only work in *open intervals*. Therefore, we split intervals into open intervals and point intervals, where point intervals can be processed by simply evaluating the polynomial on this (rational) point.

Algorithm 2.1: Abstract real root isolation algorithm

```
1  Function RealRootIsolation(p)
2      R := ∅
3      Q := { initial interval based on some root bound }
4      while Q ≠ ∅ do
5          I := remove some element from Q
6          if I contains no root of p then
7              Drop I
8          else if I contains a single root of p then
9              R := R ∪ {I}
10         else
11             Split I into disjoint intervals I_1, I_2, ... with I = ∪_i I_i
12             Q := Q ∪ {I_1, I_2, ...}
13     return R
```

2.5.2 Algebraic foundations

Before we go into the algorithmic details on how to work with real algebraic numbers, let us make a turn and lay some algebraic groundwork here, in particular minimal polynomials and the correspondence of field extensions and quotient rings. The main question that we target here is how we can (partially) evaluate a multivariate polynomial over real algebraic numbers. The following discussion is mostly taken from [Art91] and we refer the reader to [Art91], or any other textbook on algebra, for proper definitions. We summarize the key insights below and expect these to be sufficient to understand the subsequent discussion.

Insertion: Algebraic foundations [Art91]

Let K be a field and α an algebraic number over K. We define $K(\alpha)$ as the smallest field that contains both K and α and call it an *extension field of* K. Let $\varphi_\alpha : K[\xi] \to K(\alpha)$ be the K-linear function that maps ξ to α.
The fundamental theorem on homomorphisms provides that the quotient ring $K[\xi]/\ker(\varphi_\alpha)$ is isomorphic to $\text{image}(\varphi_\alpha) = K(\alpha)$ as a ring, and as $K(\alpha)$ is a field, $K[\xi]/\ker(\varphi_\alpha)$ is a field as well. As $K[\xi]$ is a polynomial ring over a field, and thus a principal ideal domain, the kernel of φ_α is generated by a single element and we write $K[\xi]/\ker(\varphi_\alpha) = K[\xi]/\langle p_\alpha \rangle$. The generating element of $\ker(\varphi_\alpha) = \langle p_\alpha \rangle$ is unique (up to normalization with respect to K) and we call it the *minimal polynomial of* α *over* K.
Let α and β be two real algebraic numbers with $p_\alpha = p_\beta$. Then we have $K(\alpha) \cong K[\xi]/\langle p_\alpha \rangle = K[\xi]/\langle p_\beta \rangle \cong K(\beta)$, however we may very well have $K(\alpha) \neq K(\beta)$, for example, $\beta \notin K(\alpha)$ and $\alpha \notin K(\beta)$.
We can iterate this construction of extension fields and the corresponding quotient rings by using $K(\alpha_1)$ and $K[\xi_1]/\langle p_{\alpha_1} \rangle$ as base fields to obtain $K(\alpha_1)(\alpha_2)$ and $(K[\xi_1]/\langle p_{\alpha_1} \rangle)[\xi_2]/\langle p_{\alpha_2} \rangle$. However, p_{α_2} needs to be a minimal polynomial *over* $K(\alpha_1)$. If p_{α_2} is reducible over $K(\alpha_1)$, $(K[\xi_1]/\langle p_{\alpha_1} \rangle)[\xi_2]/\langle p_{\alpha_2} \rangle$ is no longer an integral domain (and thus not a field) and not isomorphic to $K(\alpha_1)(\alpha_2)$.

The extension fields we construct are (in general) neither *splitting fields*, *normal*, nor *Galois extensions* as the minimal polynomial p_α may very well have another root $\beta \notin K(\alpha)$ and thus may not decompose into linear factors over $K(\alpha)$. This is (in some way) already implied by the fact that we only care about *real* algebraic numbers and never consider complex roots.
For $K' = K[x]/\langle p \rangle$ we call $id_{K'} : K[x] \to K', q \mapsto q + \langle p \rangle$ the *canonical embedding homomorphism*. We also write $id_{K'}$ for $id_{K'[\overline{y}]} : K[x][\overline{y}] \to K'[\overline{y}]$. Furthermore, we identify any $p + \langle p_\alpha \rangle \in K[\xi]/\langle p_\alpha \rangle$ with its canonical representative p as given by its preimage under the following bijection:

$$\varphi : \{p \in K[\xi] \mid \deg(p) < \deg(p_\alpha)\} \to K[\xi]/\langle p_\alpha \rangle, p \mapsto p + \langle p_\alpha \rangle$$

Let us briefly reiterate the key insights in an intuitive way. The *substitution ring homomorphism* φ_α substitutes a real algebraic number – represented by its minimal polynomial – for a variable into a polynomial. The result is a polynomial over an *extension field* of $\mathbb{Q}$ with the real algebraic numbers that we need to represent the result of the substitution. Such an extension field is *isomorphic* ("essentially identical") to a *quotient ring* and we can use the canonical representative of an element of this quotient ring instead of an element of the extension field.

Substituting a real algebraic number for a variable into a polynomial can now be done with the *(canonical) embedding homomorphism* with respect to the minimal polynomials of a *tower of field extensions*.

We feel that it is important to point out the intricacies of the "identity" of an extension field. While an (extension) field is usually seen as the set of its elements (numbers) it contains, we oftentimes distinguish between different *representations* of the same set. Though $\mathbb{Q}(\sqrt{2})$ and $\mathbb{Q}(-\sqrt{2})$ contain the same numbers – and are thus the same – their representations may differ, affecting practical computations with these objects.

Our common representation for an extension field $K(\alpha)$ is a quotient ring of the form $K[\xi]/\langle p_\alpha \rangle$ where p_α is the minimal polynomial of α over K. Such a quotient ring is isomorphic to all extension fields $K(\alpha_i)$ where α_i are the roots of p_α, α being one of them. Note that all these extension fields are isomorphic, but not (necessarily) the same: they may very well contain different elements as we show in Example 2.4.

In the following we oftentimes *identify* an extension field $K(\alpha)$ with the corresponding quotient ring $K[\xi]/\langle p_\alpha \rangle$, acknowledging the aforementioned ambiguities. We always use this formalism with respect to a model, and thus we can resolve these ambiguities and obtain the "correct" extension field by numeric evaluation with the model for the variables we use to construct the quotient ring. For example, we do this when selecting the appropriate factor f from the factorization of p_x in Algorithm 2.2. Being fully aware of this issue, we nevertheless call such a quotient ring an *extension field* as well, provided that we have the model at hand.

It is important to realize that the *substitution homomorphism* does not actually substitute a real algebraic number – in the numeric sense – but only the *algebraic information*. In particular, it does not (and can not) distinguish between the different roots of the minimal polynomial. Furthermore, it does not really "get rid of the variable" but "only" exploits the algebraic information from the minimal polynomial. We thus split the task of "evaluating a polynomial" into three separate steps that

can roughly be described as "algebraic cancellation", "algebraic substitution" and "numeric evaluation" where the substitution homomorphism is responsible for most of the algebraic cancellation. Before discussing these individual steps, we briefly summarize our task more formally.

Definition 2.38: (Partial) evaluation of a polynomial

Let $p \in \mathbb{Q}[x_1, \ldots, x_n]$ and $\mathcal{A} = \{x_1 \mapsto \alpha_1, \ldots, x_{n-1} \mapsto \alpha_{n-1}\}$. The result of *(partially) evaluating* p *over* $\mathcal{A}$ shall be a polynomial $q \in \mathbb{Q}[x_n]$ such that $\text{roots}(p) \subseteq \text{roots}(q) \times \mathbb{R}^{n-1}$.
We perform this evaluation in two steps: 1. *algebraic cancellation* makes sure that all cancellation over $\mathcal{A}$ is done and thus no terms of p vanish anymore and 2. *algebraic substitution* reduces the resulting polynomial from $\mathbb{Q}[\overline{x}]$ to an appropriate polynomial in the remaining variables.

The resulting polynomial is then used to either evaluate a constraint or isolate its real roots. In both cases, we employ *numeric evaluation*, either to perform the evaluation with the sign condition or to identify the roots of q that actually correspond to roots of p and discard all others. We now present what we call *algebraic cancellation* and *algebraic substitution* and leave the *numeric evaluation* for later as it depends on what we need to do with the resulting polynomial q.

2.5.2.1 Algebraic cancellation

We can understand "algebraic cancellation" as turning *numeric cancellation* – terms are zero under numeric evaluation with some assignment – into *symbolic cancellation* – terms are zero under syntactic rewriting. Indeed, the latter steps require that *any possible cancellation* is done symbolically. Let us assume we want to substitute $\sqrt{2}$ into $x^3 - x^2 + x - 1$. If computing manually, we would proceed roughly as follows:

$$\underbrace{\sqrt{2}^3}_{=2\sqrt{2}} - \underbrace{\sqrt{2}^2}_{=2} + \sqrt{2} - 1 = 2\sqrt{2} - 2 + \sqrt{2} - 1 = 3\sqrt{2} - 3$$

We can not further process this expression symbolically and would start some numeric approximation now. Instead we can identify $x^2 - 2$ as the minimal polynomial of $\sqrt{2}$ over $\mathbb{Q}$ and *embed* the polynomial into $\mathbb{Q}(\sqrt{2})$, or rather $\mathbb{Q}[x]/\langle x^2 - 2\rangle$, obtaining

$$(x^3 - x^2 + x - 1) + \langle x^2 - 2\rangle = (3x - 3) + \langle x^2 - 2\rangle$$

in $\mathbb{Q}[x]/\langle x^2 - 2\rangle$. Note that the result in both variants still contains some expression that we can not immediately deal with (either $\sqrt{2}$ or x), however, we stay within the realm of polynomials over rational coefficients for the latter.

In case we need to substitute multiple real algebraic numbers, we can iterate this process, but need to honor a small detail here: the minimal polynomial p_α is a minimal polynomial over $\mathbb{Q}$, and to properly construct the quotient ring we need to obtain an appropriate minimal polynomial over the quotient ring $\mathbb{Q}^*$ constructed from the preceding real algebraic numbers. We thus need to take the minimal polynomial of the next real algebraic number over $\mathbb{Q}$ and *reduce* it with respect to $\mathbb{Q}^*$ as described in Lemma 2.1 and exploited in the subsequent Algorithm 2.2.

Lemma 2.1: Minimal polynomial over an extension field

Let $\mathbb{Q}^*$ be an extension field of $\mathbb{Q}$ and p be the minimal polynomial of α over $\mathbb{Q}$. The minimal polynomial p^* of α over $\mathbb{Q}^*$ is a unique element from the irreducible normalized factors of p over $\mathbb{Q}^*$ and we have $\mathbb{Q}^*[\xi]/\langle p^* \rangle \cong \mathbb{Q}^*(\alpha)$.

Proof. As α is an algebraic number, a *minimal polynomial* for α exists over $\mathbb{Q}$ and every extension field of $\mathbb{Q}$. Let φ and φ^* be the $\mathbb{Q}$-linear ring homomorphisms for $\mathbb{Q}$ and $\mathbb{Q}^*$ for α as follows:

$$\begin{aligned} \varphi: \quad \mathbb{Q}[\xi] &\to \mathbb{Q}(\alpha), \quad \xi \mapsto \alpha_x \\ \varphi^*: \quad \mathbb{Q}^*[\xi] &\to \mathbb{Q}^*(\alpha), \quad \xi \mapsto \alpha_x \end{aligned}$$

Let p^* be the minimal polynomial of α over $\mathbb{Q}^*$. As $\varphi = \varphi^*|_{\mathbb{Q}[\xi]}$, we have

$$p \in \operatorname{kern}(\varphi) = \operatorname{kern}(\varphi^*|_{\mathbb{Q}[\xi]}) \subseteq \operatorname{kern}(\varphi^*) = \langle p^* \rangle$$

and thus p^* divides p. Hence, we find the minimal polynomial of α over $\mathbb{Q}^*$ by considering all the irreducible normalized factors of p over $\mathbb{Q}^*$. Finally, we construct $\mathbb{Q}^*[\xi]/\langle p^* \rangle$, which is isomorphic to $\mathbb{Q}^*(\alpha)$, using the embedding homomorphism as p^* is the minimal polynomial of α over the base field $\mathbb{Q}^*$. □

In Algorithm 2.2, we need to construct a minimal polynomial for α_x over $\mathbb{Q}^*$ from the minimal polynomial p_x over $\mathbb{Q}$. Following Lemma 2.1, we consider all irreducible and normalized factors of p_x over $\mathbb{Q}^*$ and select the one that vanishes over α_x. Due to Lemma 2.1, this factor f exists and is unique. Finally, we use this minimal polynomial f over $\mathbb{Q}^*$ to construct the new extension field.

Algorithm 2.2: Perform algebraic cancellation

Input: polynomial p, model $\mathcal{A}$

Function `AlgebraicCancellation`(p, $\mathcal{A}$)
- $\mathbb{Q}^* := \mathbb{Q}$, $\mathcal{A}^* := \emptyset$
- **for** $x \mapsto \alpha_x \in \mathcal{A}$ **do**
  - Let p_x the minimal polynomial of α_x ▷ p_x `in variable` ξ_x
  - Let $f \in \text{factors}(p_x, \mathbb{Q}^*[\xi_x])$ such that $f(\mathcal{A}^*, \xi_x \mapsto \alpha_x) = 0$
  - **if** f *is linear in* ξ_x **then**
    - Substitute $\xi_x - f$ for x in p
  - **else**
    - Let $\mathbb{Q}^* := \mathbb{Q}^*[\xi_x]/\langle f \rangle$
    - Let $\mathcal{A}^* := \mathcal{A}^* \cup \{\xi_x \mapsto \alpha_x\}$
- **return** $id_{\mathbb{Q}^*}(p)$

If the minimal polynomial over $\mathbb{Q}^*$ is linear in ξ_x, then already $\alpha \in \mathbb{Q}^*$ and we can simplify this procedure by directly replacing x in p instead of making the effort to construct another quotient ring and keeping an additional variable (though this would still be correct). We finally embed p into $\mathbb{Q}^*$ to obtain the properly reduced polynomial and rest the correctness of Algorithm 2.2 on Lemma 2.1.

We concede that it may not be immediately clear intuitively why the reduction from Algorithm 2.2 is necessary. Let us thus work through a few examples that illustrate how the resulting polynomial q is not "fully canceled" if we do not reduce the minimal polynomials. The first Example 2.2 shows the most intuitive case, where two real algebraic numbers come from the same extension field.

Example 2.2: Same extension field

Let $\mathcal{A} = \{x_1 \mapsto \sqrt{2}, x_2 \mapsto \sqrt{2}\}$ and $q = x_1 + x_2$. Naturally, we want to realize that indeed $x_1 = x_2$ and only perform the algebraic operations once. Starting with $\mathbb{Q}^* = \mathbb{Q}$ and $x_1 \mapsto \sqrt{2}$, $\xi_1^2 - 2$ is the minimal polynomial of $\sqrt{2}$ over $\mathbb{Q}^*$ and we construct $\mathbb{Q}^* = \mathbb{Q}[\xi_1]/\langle \xi_1^2 - 2 \rangle$.
For x_2 we have the minimal polynomial $\xi_2^2 - 2$ for $\sqrt{2}$ over $\mathbb{Q}$ which factors into $(\xi_2 + \xi_1) \cdot (\xi_2 - \xi_1)$ over $\mathbb{Q}^*$ and we select the factor $\xi_2 - \xi_1$ as the minimal polynomial of $\sqrt{2}$ over $\mathbb{Q}^*$ (as the corresponding polynomial factor $x_2 - x_1$ vanishes over $\mathcal{A}$). We can recognize that this is *linear in* ξ_2 and instead of constructing another quotient ring simply eliminate x_2 using $x_2 \coloneqq x_1$.
This is not unique to identical real algebraic numbers, but for all that can directly be represented within the current extension field $\mathbb{Q}^*$. Consider for example $x_2 \mapsto 5$ or $x_2 \mapsto 1 + \sqrt{2}$ which can also be dealt with in this way. We can understand this as *finding a linear parametrization in the previous assignments.*

Let us consider a slightly more complex and arguably less intuitive case. Assume the extension field of a real algebraic number to be *related* to the current $\mathbb{Q}^*$ in the sense that it has a nontrivial intersection (more than $\mathbb{Q}$) but is not included in $\mathbb{Q}^*$ as depicted on the right. In this case, the minimal polynomial factors into multiple factors, but the vanishing factor f is not linear.

$$\begin{array}{c} \mathbb{Q}\left(\sqrt[4]{2}\right) \\ \cup\!\!\!\!\!\!\backslash \\ \mathbb{Q}\left(\sqrt{2}\right) \\ \cup\!\!\!\!\!\!\backslash \\ \mathbb{Q} \end{array}$$

Example 2.3: Related extension field

Let $\mathcal{A} = \{x_1 \mapsto \sqrt{2}, x_2 \mapsto \sqrt[4]{2}\}$ with $p_1 = \xi_1^2 - 2$ and $p_2 = \xi_2^4 - 2$ the minimal polynomials of $\sqrt{2}$ and $\sqrt[4]{2}$ over $\mathbb{Q}$. We observe, that p_2 factors into $(\xi_2^2 - \xi_1) \cdot (\xi_2^2 + \xi_1)$ over $\mathbb{Q}[\xi_1]/\langle p_1 \rangle$, but ignore this knowledge for the moment and attempt to extend $\mathbb{Q}^*$ with the original p_2.
Assume $a = \xi_2^2 - \xi_1$, $b = \xi_2^2 + \xi_1$ and $c = 0$, then we have $a \cdot b = 0 = a \cdot c$ and $a \neq 0$ in $(\mathbb{Q}[\xi_1]/\langle p_1 \rangle)[\xi_2]/\langle p_2 \rangle$, but $b \neq c$. This violates the definition for *integral domains* and thus $(\mathbb{Q}[\xi_1]/\langle p_1 \rangle)[\xi_2]/\langle p_2 \rangle$ is not a field.

It might not be immediately clear why working on a $\mathbb{Q}^*$ that is not a field like shown in Example 2.3 should be a problem. Of course, fields are a "more convenient" structure in general, but we show now that the algebraic cancellation may indeed fail.

Example 2.3 continued

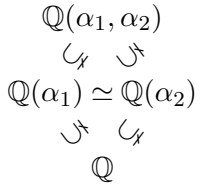
Let $p = (x_2^2 - x_1) \cdot x_3$. Embedding p into $(\mathbb{Q}[\xi_1]/\langle p_1 \rangle)[\xi_2]/\langle p_2 \rangle$ — which is not even an integral domain as shown before – yields $(\xi_2^2 - \xi_1) \cdot x_3$, though p vanishes identically at $\mathcal{A}$. The factorization of p_2 over $\mathbb{Q}[\xi_1]/\langle p_1 \rangle$ instead gives us $(\xi_2^2 - \xi_1) \cdot (\xi_2^2 + \xi_1)$ and we construct $\mathbb{Q}^* = (\mathbb{Q}[\xi_1]/\langle p_1 \rangle)[\xi_2]/\langle \xi_2^2 - \xi_1 \rangle$. Embedding p into $\mathbb{Q}^*$ now correctly yields $p = 0$.

We used two real algebraic numbers that have an algebraic relation (namely $\alpha_1 = \alpha_2^2$) and we see that cancellation is only performed incompletely if the information about this relation is not explicitly stated. Also note that this process heavily depends on the order in which the real algebraic numbers are considered as we see next.

Example 2.3 continued

Let instead $\mathcal{A} = \{x_1 \mapsto \sqrt[4]{2}, x_2 \mapsto \sqrt{2}\}$. Then p_2 factors into $(\xi_1^2 - \xi_2) \cdot (\xi_1^2 + \xi_2)$ over $\mathbb{Q}[\xi_1]/\langle p_1 \rangle$ and we can derive that either $\xi_2 = \xi_1^2$ or $\xi_2 = -\xi_1^2$. This allows us to avoid constructing another quotient ring altogether and simply substitute ξ_2 (by either ξ_1^2 or $-\xi_1^2$) in p.

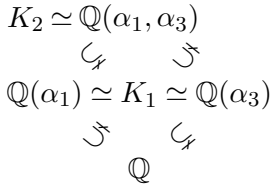
In the previous example we considered *related* real algebraic numbers – in the sense that $\alpha_1 \in \mathbb{Q}(\alpha_2)$ – with different minimal polynomials. It is also possible that two *unrelated* real algebraic numbers have the same minimal polynomial as we see in Example 2.4. Note that, as depicted on the right, the extensions field for these *unrelated* real algebraic numbers are still *isomorphic*.



Example 2.4: Unrelated extension field

Let $p = \xi^4 - 6\xi^2 + 6$ be *irreducible over* $\mathbb{Q}[\xi]$ with four real roots $\alpha_{1,2} = \pm\sqrt{3 + \sqrt{3}}$ and $\alpha_{3,4} = \pm\sqrt{3 - \sqrt{3}}$. We can show that $\alpha_3 \notin \mathbb{Q}(\alpha_1)$ and $\alpha_1 \notin \mathbb{Q}(\alpha_3)$ and thus $\mathbb{Q}[\xi]/\langle p \rangle$ represents two different – though isomorphic – extension fields.

If we indeed want to work with all of these roots, for example α_1 and α_3, we have to construct further extension fields – one for each group of what one could call "adjoint" roots – until all of the roots are contained in $\mathbb{Q}^*$. For this to work, we compute the minimal polynomial of α_3 in $\mathbb{Q}(\alpha_1)$ as shown in the continued example. Consider the visualization on the right on how these fields relate to each other.



Example 2.4 continued

Assume $K_1 = \mathbb{Q}[\xi_1]/\langle \xi_1^4 - 6\xi_1^2 + 6 \rangle$. p is not irreducible over K_1 but factors into $(\xi + \xi_1) \cdot (\xi - \xi_1) \cdot (\xi^2 + \xi_1 - 6)$ and we can construct a second extension field $K_2 = (\mathbb{Q}[\xi_1]/\langle \xi_1^4 - 6\xi_1^2 + 6 \rangle)[\xi_2]/\langle \xi_2^2 + \xi_1 - 6 \rangle$.

2.5.2.2 Algebraic substitution

Now that we have taken care of all cancellations, we need to eliminate the respective variables from the polynomial. Let us first discuss possible implementations and then consider an example. We propose two versions, either based on Gröbner bases or resultants, which both allow projecting algebraic information about the roots of polynomials onto lower dimensions. Note that we sometimes speak about "eliminating variables" here, fully aware that this is, in fact, different from *quantifier elimination* and more similar to how a projection operator in the cylindrical algebraic decomposition "eliminates variables".

The fundamental insight for both variants is to understand the problem of substitution as a system of equalities, consisting of the input polynomial p (embedded into the extension field) and the minimal polynomials m_i (of the extension fields). To obtain the polynomial q we need to eliminate $x_1, \dots x_k$ from this system of equalities.

One of the most popular methods in computer algebra in general are Gröbner bases. We refrain from a definition or an in-depth discussion here and only refer to appropriate literature like the one referenced in Section 1.1.6.5. In particular, however, Gröbner bases can be instrumented to perform exactly this kind of "variable elimination" by employing an appropriate *elimination order*. In our case we use the canonical variable order $x_1 < \dots < x_n$ – or rather the induced lexicographical term order.

Under this lexicographical order (which is an *elimination order*) the resulting Gröbner basis always contains a polynomial q from $\mathbb{Q}[x_n]$ and the roots of this polynomial cover the roots of the input polynomial p over the given assignment. We formalize this statement in the following Theorem 2.1 and note that we only consider the special case of $k = n - 1$ here. For arbitrary k this works essentially the same, however, we only require $k = n - 1$ for the subsequent operations.

Theorem 2.1: Gröbner basis for algebraic substitution

Let $p \in \mathbb{Q}[x_1, \dots, x_n]$ be a polynomial, $\mathcal{A} = \{x_1 \mapsto \alpha_1, \dots, x_{n-1} \mapsto \alpha_{n-1}\}$ an assignment, and p_i minimal polynomials for α_i over $\mathbb{Q}[x_1, \dots, x_i]/\langle p_1, \dots, p_{i-1}\rangle$ as constructed in Algorithm 2.2.
Let $I \coloneqq \{p_1, \dots, p_{n-1}, p\}$ and let G be the Gröbner basis of I with an elimination block ordering $\{x_1, \dots, x_{n-1}\} > \{x_n\}$. Then G contains at least one non-zero polynomial $q_n \in \mathbb{Q}[x_n]$. Thus, the real roots of q_n (in x_n) cover all solutions of G and as $\langle G\rangle = \langle I\rangle$ also *cover all real roots of p evaluated at $\mathcal{A}$*:

$$\operatorname{roots}(p(\mathcal{A})) \subseteq \operatorname{roots}(q_n)$$

Proof. We have seen that $K \coloneqq \mathbb{Q}[x_1, \dots, x_{n-1}]/\langle p_1, \dots, p_{n-1}\rangle$ is a field and its dimension over $\mathbb{Q}$ is finite. Furthermore, the leading coefficient of p as polynomial from $\mathbb{Q}[x_1, \dots, x_n]/\langle p_1, \dots, p_{n-1}\rangle$ (or rather $K[x_n]$) is invertible and we obtain $p' = p/\operatorname{lcoeff}(p) = x_n^d - \sum_{i=1}^{d-1} a_i x_n^i$ for some coefficients $a_i \in K$. Thus, we have $\langle I'\rangle \coloneqq \langle p_1, \dots, p_{n-1}, p'\rangle = \langle p_1, \dots, p_{n-1}, p\rangle = \langle I\rangle$.
Now consider $J \coloneqq \mathbb{Q}[x_1, \dots, x_n]/\langle I'\rangle$ as vector space over $\mathbb{Q}$. From $p' \in \langle I'\rangle$ we get $p' = 0$ in J, or equivalently $x_n^d = \sum_{i=1}^{d-1} a_i x_n^i$. Hence, J is finitely generated as vector space over K by $\{1, x_n, x_n^2, \dots, x_n^{d-1}\}$ and, thus, has finite dimension

as vector space over $\mathbb{Q}$. We claim that every (non-zero) element from this vector space J is algebraic and we can use its minimal polynomial over $\mathbb{Q}$ to construct a polynomial in $\langle I' \rangle$.
Let $0 \neq p \in J$ be such a non-zero element and consider the vector space over $\mathbb{Q}$ given by $J' \coloneqq \langle 1, p, p^2, \dots \rangle_{\mathbb{Q}}$. As J is a $\mathbb{Q}$-algebra and thus closed under multiplication, J' is a sub-vector space of J. J has finite dimension, thereby J' has finite dimension as well and thus $J' = \langle 1, p, p^1, \dots, p^{d-1} \rangle_{\mathbb{Q}}$ for some d. Hence, there exist coefficients $a_i \in \mathbb{Q}$ such that $p^d = \sum_{i=0}^{d-1} a_i \cdot p^i$. If d is minimal, $\sum_{i=0}^{d-1} a_i \cdot p^i$ is the minimal polynomial for p and in particular $p^d - \sum_{i=0}^{d-1} a_i \cdot p^i = 0$. This implies that $p^d - \sum_{i=0}^{d-1} a_i \cdot p^i \in \langle I' \rangle$ and for $p = x_n$ we have a polynomial $q_n \in \mathbb{Q}[x_n]$ such that $q_n \in \langle I' \rangle = \langle I \rangle$.
Under an elimination block ordering of the form $\{x_1, \dots, x_{n-1}\} > \{x_n\}$, such a $q_n \in \mathbb{Q}[x_n]$ is one of the generators of the reduced Gröbner basis of $\langle I \rangle$. □

We can also slightly reframe the system of equalities as the search for common roots of multiple polynomials. Resultants are commonly used to generate a polynomial with one variable less whose roots *witness* the common roots of the input polynomials, which is essentially what we need here. Note that the resultant itself argues about *complex roots*, and thus not every *real root* of the resultant witnesses a *common real root* of the input polynomials. Instead, the real roots of the resultants *cover* all common real roots of the input polynomials.

Given that we have one polynomial over $\overline{x}$ and one polynomial over every x_i that shall be eliminated, we can give an easy scheme that applies resultants to compute q:

$$q_k \coloneqq p \qquad q_i \coloneqq \operatorname{res}_{\xi_{i+1}}(q_{i+1}, p_i) \qquad q \coloneqq q_0$$

Both methods – either using a Gröbner basis with an appropriate elimination order or using iterated resultants – yield a polynomial (equation) that is *univariate in* x_n and *logically follows* from the input equations. Thus, the real roots for the input polynomial p are a subset of the assignment $\mathcal{A}$ extended with the real roots of the polynomial q. We show how both approaches can be used in the following Example 2.5.

Example 2.5: Algebraic substitution

Let $\mathcal{A} = \{x_1 \mapsto \sqrt{2}, x_2 \mapsto \sqrt[4]{2}\}$ and $p = (x_2^2 + x_1) \cdot x_3$. We have seen in Example 2.3 that we construct $\mathbb{Q}^* = (\mathbb{Q}[\xi_1]/\langle \xi_1^2 - 2 \rangle)[\xi_2]/\langle \xi_2^2 - \xi_1 \rangle$. Embedding p into $\mathbb{Q}^*$ yields $(\xi_2^2 + \xi_1) \cdot x_3$.
The Gröbner basis of $\{\xi_1^2 - 2, \xi_2^2 - \xi_1, (\xi_2^2 + \xi_1) \cdot x_3\}$ under the elimination order $\xi_1 > \xi_2 > x_3$ is $\{\xi_1 - \xi_2^2, \xi_2^4 - 2, x_3\}$ and we thus extract $q \coloneqq x_3$.
Using resultants we can compute $q_2 = p$, $q_1 = \xi_1 \cdot x_3$ and finally $q = q_0 = x_3$ to obtain the same result.

Note that this algebraic substitution is not necessarily *exact* (or *precise*) in the sense that the resulting polynomial q has *exactly* the roots we are looking for. As the algebraic substitution only works on the minimal polynomials – that can not distinguish between any of its real roots – we essentially get all possible solutions for all the roots. The resulting polynomial q might thus have additional roots where p does not vanish and we call such roots *spurious*.

As q only has finitely many roots, we can check them individually (whether p vanishes over this root) and thereby identify the "actual" real roots of p and discard the spurious roots as shown in Example 2.6.

Example 2.6: Algebraic substitution with spurious roots

Let $\mathcal{A} = \{x_1 \mapsto \sqrt{2}\}$ and $p = x_2^2 + (2 - x_1) \cdot x_2 + 1 - x_1$. We obtain $x_2^3 + 3x_2^2 + x_2 - 1$ – either from a Gröbner basis or the (reduced) resultant – which again factors into $(x_2 + 1) \cdot (x_2^2 + 2x_2 - 1)$ with real roots -1 and $-1 \pm \sqrt{2}$. We check that $\{x_1 \mapsto \sqrt{2}, x_2 \mapsto -1\}$ is indeed a root, as is $\{x_1 \mapsto \sqrt{2}, x_2 \mapsto -1 + \sqrt{2}\}$. However, $\{x_1 \mapsto \sqrt{2}, x_2 \mapsto -1 - \sqrt{2}\}$ is spurious as p does not vanish at this point.

Note that evaluating a polynomial p over some variable assignment is usually formulated as isolating the (unique) real root of $v - p$, where v is a fresh variable. However, this requires real root isolation over a (partial) variable assignment again and leaves us with a cyclic dependency of these two methods. We thus show how to evaluate the constraint $p = 0$ without relying on real root isolation in the following Section 2.5.3.

2.5.3 Computing with real algebraic numbers

Though we introduced real algebraic numbers as a general set of numbers – just like the rationals – we do not need to perform arbitrary arithmetic operations on them. For example, we do not strictly require addition or multiplication for our purpose. We thus implement only the bare minimum of functionality here, mostly consisting of the following operations: ordering two real algebraic numbers, evaluating constraints over a model with real algebraic numbers, and isolating the real roots of a multivariate polynomial over a (partial) model. In addition, we present a few more elementary operations, required for the above.

Our implementation follows a few guiding principles that are by no means unique or particularly ingenious, but we feel are worth noting. If we can infer the result from the numerical information alone – for example for ordering two real algebraic numbers – doing so is usually way more efficient than considering the algebraic information. Furthermore, we oftentimes *refine* a real algebraic number – usually by making the isolating interval smaller, but possibly also reducing the polynomial if it is not a minimal polynomial already – and we "propagate" this refinement to all instances of this particular real algebraic number by maintaining a *common global state*.

Refinement. The refinement of a real algebraic number α aims at making the isolating interval smaller but still having it isolate the same real root. The easiest strategy is to split the interval in half, check whether the root is in the left part or the right part of the interval, and replace the interval by its left or right half accordingly. To check whether a particular interval contains a root we can employ methods like Sturm's theorem [Stu29] or Descartes' rule of sign [Des37; CA76] that we also use for real root isolation as discussed in Section 2.5.1.

If we can ensure that the polynomial is square-free – for example because it is a minimal polynomial – we can improve upon this by only considering the sign of the polynomial at the interval endpoints. We can then establish the invariant that the signs of the endpoints of any isolating interval are different and use this criterion to select the new isolating interval. We formalize this observation in the following Theorem 2.2.

Theorem 2.2: Refinement with irreducible polynomials

Let $p \neq 0$ be an *irreducible* polynomial (for example a minimal polynomial) and $(\underset{\sim}{\alpha}, \tilde{\alpha})$ an isolating interval for some real root of p. Then $\operatorname{sgn}(p(\underset{\sim}{\alpha})) \neq \operatorname{sgn}(p(\tilde{\alpha}))$. Let now $c \in (\underset{\sim}{\alpha}, \tilde{\alpha})$ be a *pivot point*. Then a new isolating interval is

$$\begin{aligned} (\underset{\sim}{\alpha}, c) &\text{ if } \operatorname{sgn}(p(c)) = \operatorname{sgn}(p(\tilde{\alpha})) \\ [c, c] &\text{ if } p(c) = 0 \\ (c, \tilde{\alpha}) &\text{ if } \operatorname{sgn}(p(c)) = \operatorname{sgn}(p(\underset{\sim}{\alpha})) \end{aligned}$$

Proof. Let p' be the derivative of p. As $p \neq 0$ is irreducible and $p \neq p'$ due to $\deg(p') < \deg(p)$, we know that p and p' have no common roots and thus $p'(\alpha) \neq 0$ for every real root α of p. Therefore, the signs of $\lim_{x \uparrow \alpha} p(x)$ and $\lim_{x \downarrow \alpha} p(x)$ are different and thus an infinitesimally small interval around α is isolating with the above property.

Now assume that $(\underset{\sim}{\alpha}, \tilde{\alpha})$ is an isolating interval. Due to the observation that the upper and lower limits have different signs, we also know that $p(\underset{\sim}{\alpha})$ and $p(\tilde{\alpha})$ have different signs. Otherwise, p would need to have another root in the interval – to change from one sign at $\underset{\sim}{\alpha}$ ($\tilde{\alpha}$) to the other sign at the lower (upper) limit – according to the intermediate value theorem. The existence of this other root, however, contradicts the assumption that the interval was isolating.

Given that the endpoints of any isolating interval have different signs (and the interval contains only a single root) the only sign change occurs exactly at the real root. We can thus refine the interval by selecting an arbitrary new interval contained in the isolating interval which still contains this single sign change. □

It can be useful to not select the *pivot point* arbitrarily "at random". If we refine the representations of α_1 and α_2 to establish that $\alpha_1 \neq \alpha_2$ by showing that I_1 and I_2 are disjoint, we can use $\underset{\sim}{\alpha}_2$ (or $\tilde{\alpha}_2$) as pivot points. If I_1 is refined to be below $\underset{\sim}{\alpha}_2$ (or above $\tilde{\alpha}_2$) we obtain the result after a single refinement step.

Comparison. Comparing two real algebraic numbers is the evaluation of the common relational operators – $<, \leq, =, \neq, \geq, >$ – in line with the standard relation on real numbers. This allows us to recognize that two real algebraic numbers are, in fact, equal – even if their representation is not identical – or sort a list of real algebraic numbers. In many cases, a reasonably good numeric approximation (from the interval I) is sufficient for comparison. If we assume the real algebraic numbers to be *different*, we can refine the numeric approximations until the intervals are disjoint.

Let us thus consider the question whether two real algebraic numbers $\alpha_1 = (p_1, I_1)$ and $\alpha_2 = (p_2, I_2)$ are equal. Recall that we did not assume the polynomial to be the minimal polynomial, hence α_1 and α_2 might be equal although $p_1 \neq p_2$. Still, $\alpha_1 = \alpha_2$ implies that p_1 and p_2 have a common root, or equivalently that they are a root of $\gcd(p_1, p_2)$. We can thus compute the greatest common divisor and use the result to refine α_1 and α_2 as shown in Algorithm 2.3. If we always use minimal polynomials, $p_1 \neq p_2$ directly implies $\alpha_1 \neq \alpha_2$ and this refinement is not only unnecessary but also completely pointless: if p_1 and p_2 are both minimal polynomials, they are either identical or their greatest common divisor is one and no refinement can be done.

Algorithm 2.3: Refine polynomial of two real algebraic numbers

```
1 Function RefinePolynomial(α1, α2)
2     g := gcd(p1, p2)
3     if α1 is a root of g then
4         p1 := g
5     else
6         p1 := p1/g
```

This algorithm relies on being able to decide whether the real algebraic number α_1 is a root of the polynomial g. Though this is a more difficult question in general, we have that g is a factor of p_1 here which makes it significantly easier to answer: we have that $\text{roots}(g) \subseteq \text{roots}(p_1)$ and hence the isolating interval of α_1 either isolates a single or no root of g. Thus we only need to check whether g has a root in I_1 which can be done by Descartes' rule of signs, or any other method mentioned in Section 2.5.1.

After this refinement – or without it, if we use minimal polynomials anyway – we check whether $p_1 \neq p_2$. If so, we immediately have $\alpha_1 \neq \alpha_2$, otherwise we need to refine the intervals until they are either disjoint (certifying $\alpha_1 \neq \alpha_2$) or one of the intervals contains the other (implying $\alpha_1 = \alpha_2$).

Sampling. An important part of many procedures that are presented later is to generate a sample point s relative to one or two real algebraic numbers. We generally distinguish the following three cases:

$$s < \alpha \qquad\qquad s \in (\alpha_1, \alpha_2) \qquad\qquad \alpha < s$$

Assuming that $\alpha_1 < \alpha_2$, a sufficiently good numeric approximation (such that $\tilde{\alpha}_1 < \underset{\sim}{\alpha}_2$) is enough to generate a rational sample point s in all cases. We can then implement sampling as follows:

$$\begin{aligned} \text{for } s < \alpha: &\quad s = \underset{\sim}{\alpha} - 1 \\ \text{for } s \in (\alpha_1, \alpha_2): &\quad s = \mathit{midpoint}(\tilde{\alpha}_1, \underset{\sim}{\alpha}_2) \\ \text{for } \alpha < s: &\quad s = \tilde{\alpha} + 1 \end{aligned}$$

Evaluation of constraints. Given that we want to find a solution to systems of constraints, we need to be able to evaluate constraints – or, in other words, evaluate the sign of a polynomial for some model. To simplify the later presentation, we define the exact problem in Definition 2.39.

Definition 2.39: Evaluation over real algebraic numbers

Let $p \sim 0$ be a constraint and a model $\mathcal{A}$ that may contain real algebraic numbers. We denote the question whether $p \sim 0$ is true after substituting $\mathcal{A}$ into p by *evaluating* $p \sim 0$ or equivalently *determining the sign of* p.

As before, a good numeric approximation can be very beneficial in practice. Let us assume that the polynomial p evaluates to some value that is different from zero. In this case, we can employ interval arithmetic to evaluate p over the isolating intervals

from $\mathcal{A}$ and refine these isolating intervals until the result of the evaluation (by interval arithmetic) no longer contains zero. We can then safely derive the sign of p under $\mathcal{A}$. However, we need to stop this method eventually in case p evaluates to zero to ensure termination.

One possible approach – which is the one we implement – is to construct the polynomial $q := \xi - p$ and isolate the real roots in ξ. For several reasons – in particular because q is linear in ξ – this is much easier than the general *multivariate real root isolation* we discuss in the subsequent section: 1. we know that we are only looking for a single real root α, 2. we know that we can obtain an interval I that contains α by evaluating p over the isolating intervals from $\mathcal{A}$, 3. we only need to determine the sign of α.

Our method furthermore relies on the fact that we can use any of the bounds on real roots from Section 2.5.1 to obtain a *lower bound lb* on positive real roots (and accordingly *upper bound ub* on negative real roots). From these roots, we can derive 1. if $I \subseteq (-\infty, 0)$ then $\alpha < 0$, 2. if $I \subseteq (ub, lb)$ then $\alpha = 0$, 3. if $I \subseteq (0, \infty)$ then $\alpha > 0$. Given these observations, we only need to refine the real algebraic numbers in $\mathcal{A}$ until I satisfies one of these criteria.

The careful reader might notice that the bounds on real roots from Section 2.5.1 are defined for univariate polynomial over number coefficients while q is a multivariate polynomial. We thus need to "eliminate" all variables (except ξ) from q and do so as described in Definition 2.38 to compute bounds on the real roots. Furthermore, note that although we initially described our approach as isolating the real roots of $\xi - p$ in ξ, we have not used this more general approach here but rather use a more specialized method that exploits the additional knowledge mentioned above.

We have seen that the partial evaluation of q that we use to compute the bounds on real roots might introduce what we called *spurious roots*. They do not pose a serious problem here as they might only make the bounds worse and we thus simply ignore this issue here.

Multivariate real root isolation

Finally, we need to compute the real roots of a polynomial over a given model $\mathcal{A}$. More specifically, we have a multivariate polynomial $p \in \mathbb{Q}[x_1, \ldots, x_k]$ and $\mathcal{A} = \{x_1 \mapsto \alpha_1, \ldots x_{k-1} \mapsto \alpha_{k-1}\}$ and want to compute all real roots of the univariate polynomial that is created by substituting the model into p. As noted above, we can not simply substitute the model into p and invoke a method for real root isolation – at least not in every case – but need to employ some other method.

As before, we employ the (partial) evaluation of p over $\mathcal{A}$ to obtain a polynomial q that covers the roots of p. Multivariate root isolation is thus implemented as 1. partially evaluating p in the sense of Definition 2.38, obtaining q, 2. isolating the (univariate) real roots α_i of q, and 3. removing spurious roots where p does not vanish at $\mathcal{A} \cup \{x_k \mapsto \alpha_i\}$ by evaluating the constraint $p = 0$.

As already explained and shown in Example 2.6, the result of the partial evaluation can have spurious roots where p in fact does not vanish. Note that we thus require evaluating the constraint $p = 0$ in the third step. Therefore, it is crucial that we implement this constraint evaluation from the previous section without actually relying on multivariate root isolation to avoid cyclic dependencies.

2.5.4 Other representations

As already mentioned, other representations for real algebraic numbers exist. We now briefly describe two alternatives based on root indexing and employing Thom's lemma to get a unique characterization of roots. Be aware that using different representations within one implementation usually has the significant caveat that they do not interact nicely. We assume that our whole software uses either of them and all real algebraic numbers that we have – in one execution of our program – have the same representation.

Of course, a rational root can be represented as a rational number and we convert all of the representations to a simple rational representation if possible. All described algorithms gracefully integrate rational representations, thus being an exception to the above restriction.

Indexed representation. Instead of storing an isolating interval, we can also identify a root by its index among all the (ordered) real roots of the given polynomial. This makes for a very concise representation and also allows for some computations. However, we eventually need some numeric representation – usually intervals – for many operations.

Note that this indexed representation can nicely be generalized to higher dimensions to what we call *multivariate roots* in Section 8.3 where we identify a root of a multivariate polynomial over a partial model (that assigns all but one particular variable).

Thom representation. Another representation based on Thom's lemma consists of *sign conditions on the derivatives* of a polynomial and is sometimes called *Thom encoding*. They offer a very different approach to represent real algebraic numbers and a thorough definition can be found in [BPR10] which also proposes a variant of CAD based on Thom encodings.

We have experimented with Thom encodings and found them to be surprisingly efficient but still not on par with the interval-based representation shown before. Details on our implementation, as well as some experiments, can be found in [Win16]. Note, however, that some issues with our implementation are listed at the end of [Win16], thus its performance relative to the interval-based representation could probably be improved.

2.6 Benchmarks and methodology

When trying to evaluate any software, it is important to have a somewhat reasonable set of test cases that represent the important use cases for the software at hand and allow to check for the interesting qualitative and quantitative properties of the runs. In our case, the software being tested are different solvers that we check for correctness, memory usage, and run time. The set of test cases (or *benchmarks*) is taken from the SMT-LIB benchmark set [BFT16], arguably the standard set for SMT solving. Unless explicitly stated otherwise, we always refer to the quantifier-free nonlinear real arithmetic benchmarks (called `QF_NRA`).

The SMT-LIB benchmark sets are usually organized into "families", where each family contains a number of benchmarks that usually come from a particular application and thus also share common characteristics. Given that the number of benchmarks per family varies wildly, it is an ongoing discussion in the SMT community how to compare

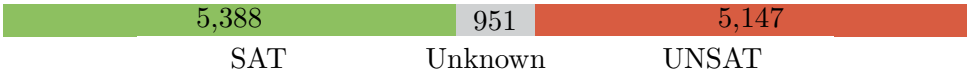


Figure 2.1: Known satisfiability

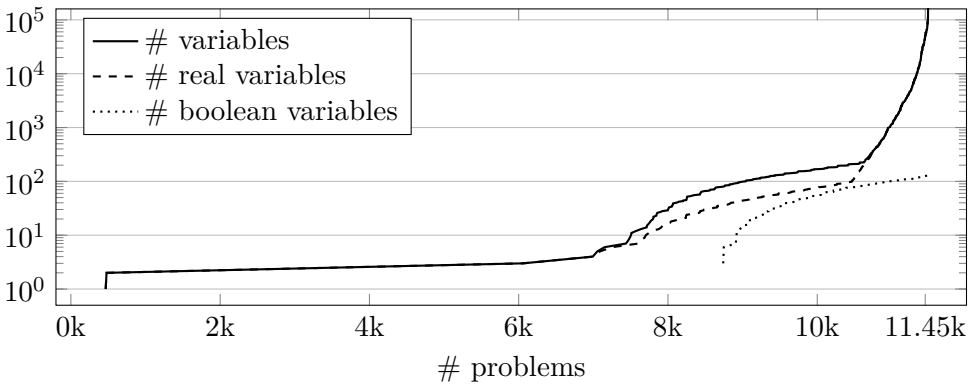


Figure 2.2: Number of variables

different solvers across such a benchmark set, for example, witnessed by regularly changing scoring rules for the SMT competition.

We sometimes observe that the performance of different heuristics significantly varies across these families, giving an apparent advantage to heuristics that happen to work better on larger families. We do not engage in this kind of analysis throughout this thesis and only consider the overall number of solved instances, as 1. there is no other metric that enjoys a similarly widespread acceptance throughout the community, 2. there is no point in analyzing the specific characteristics of the individual families for this thesis and 3. it would multiply the space and effort needed for our evaluation without a clear benefit. Instead, we encourage the reader to evaluate multiple variants of the proposed techniques on the particular problems of interests, even if they performed rather badly for us.

Separate from the question of how we use this benchmark, we need to understand that this benchmark set is not the absolute truth. Being only a collection of benchmarks – the best we have, though – it merely contains whatever someone submitted at some point. Some benchmarks are clearly artificial while others stem from actual applications that use SMT queries in their backends. However, rumor has it that some of these were generated by feeding artificial problems to these applications, making the resulting SMT queries somewhat artificial as well. Altogether, we can use this benchmark set as an indicator for performance but need to be cautious to extrapolate to new applications.

To get a feeling for this set of benchmarks, we give a rough analysis of its characteristics. For this, we parsed the problem instances with `SMT-RAT` and extracted some statistical data from the resulting formulae. Note that some amount of simplification and normalization is performed while constructing the formulae.

Firstly, Figure 2.1 shows the known satisfiability of all problem instances where *unknown* represents files without annotation or unknown status. Having checked that no solvers produce conflicting results, we assume a problem to have SAT or UNSAT status if they are annotated by SMT-LIB or any solver can solve them.

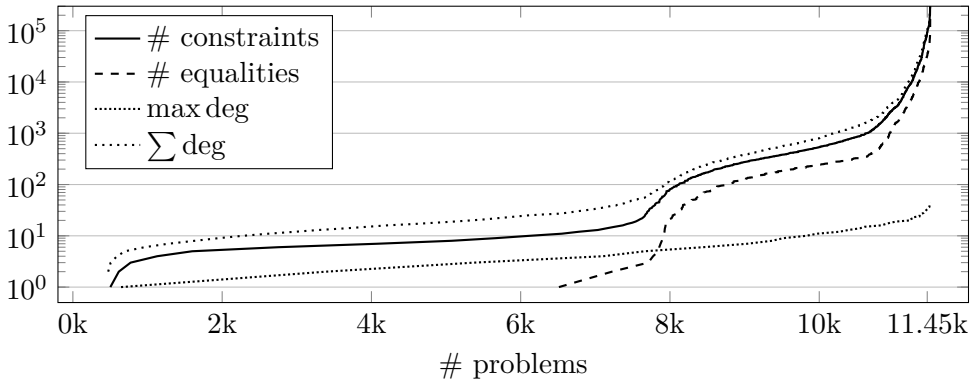


Figure 2.3: Number of constraints and equalities

The number of variables is one of the main drivers of asymptotic complexity for essentially all methods that are discussed in this thesis. We see in Figure 2.2 that almost two-thirds of the benchmarks only contain a low number of variables (below 10), but also a few hundred with a thousand or more variables are present.

More than three-quarters of the benchmarks contain no Boolean variables at all, which already indicates a low Boolean complexity in most cases – though this might change after Tseitin's transformation. Interestingly, we observe that 461 examples have no variables at all, mostly because normalization and naive simplifications were sufficient to reduce the formula to *true* or *false*.

The number of constraints also plays a considerable role and we see some statistics in Figure 2.3. Once again, we observe that 461 benchmarks were simplified to either *true* or *false* and thus contain no constraints. About half of the benchmarks have at most ten constraints, though the maximum is at almost $250k$.

More interesting observations can be drawn from the degrees of the constraints. Comparing the sum of all degrees with the number of constraints indicates that the average degree is rather small and usually less than two. The maximum degree, however, is significantly larger, growing up to 44. We infer that many examples consider only a few (or even only a single one) high-degree constraint in conjunction with a larger set of low-degree constraints, apparently linear in many cases. This observation motivates a number of techniques that aim to exploit linear constraints in particular, for example [LSC+13] or variable elimination from Section 6.6.1.

We also observe that more than a third of the examples contain a significant number of equalities, which also allows for effective simplifications in many cases as we discuss for example in Section 5.2.8 or Section 6.6.1.

2.6.1 Preprocessing and tuned heuristics

We now try to give some intuition on how different components of a solver might contribute to the overall solving performance and how we are going to analyze the practical performance of a solver. To do this we exemplarily look at the performance of the SMT solver `Z3` [MB08c] that is shown in Figure 2.4 on this benchmark set. `Z3` implements an `MCSAT` – or more specifically `NLSAT` – approach which is explained in more detail in Chapter 7.

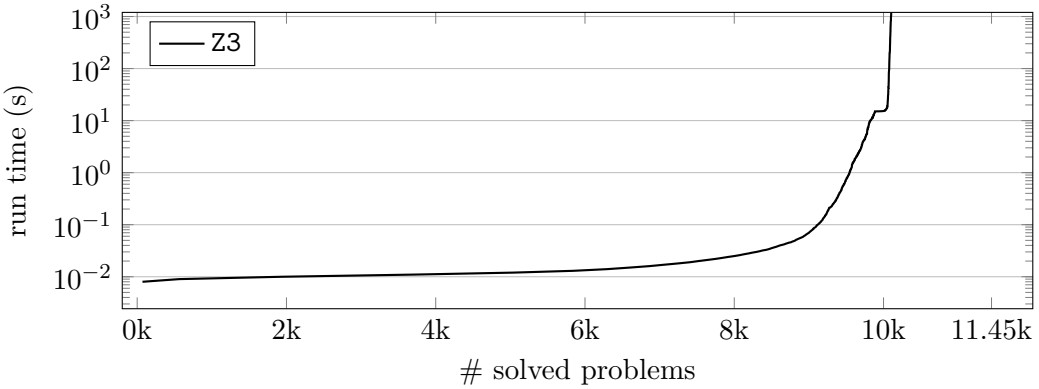


Figure 2.4: Run times of Z3 (seconds)

timeout	solved
0.1 s	9115
1 s	9543
5 s	9761
10 s	9824
20 s	10 054
60 s	10 070
120 s	10 076
300 s	10 087
600 s	10 093
1200 s	10 103

property	solved
Overall	10 105
In NLSAT	8457
At least one propagation	8457
At least one stage	7749
At least five propagations	7664
At least one conflict	5323
At least five stages	3090
At least one Boolean decision	3016
At least five conflicts	2255
At least five Boolean decision	1870

Figure 2.5: Statistics of Z3 on QF_NRA

First, we observe that a longer timeout is unlikely to yield more results as the required run time tends to grow exponentially in practice. Consider, for example, the results shown in Figure 2.4. We see that more than 90% of the instances – that can be solved within 20 minutes – are already solved within 0.1 seconds and less than 0.5% of the solved ones required more than 20 seconds.

Altogether, we can say that we either solve a benchmark instantly – within a few seconds – or there is a good chance that we cannot solve it in a practically acceptable time at all. It seems tempting to immediately argue with the doubly-exponential run-time behavior of CAD which is the underlying technique of the NLSAT solver that Z3 ultimately uses for this logic.

We should, however, put this claim into perspective and give some selected views on the Z3 solving statistics. We see in Figure 2.5 that about 1650 problems do not enter the NLSAT solver at all – note that the 8457 problems that have at least one propagation cover all the ones below.

Moreover, only 7749 instances create at least a single stage – selecting a theory variable to be assigned – indicating that the difference of about 700 problems never even considers performing a theory decision. Only 5323 ever finish conflict resolution at least once – thus 3134 problems either detect unsatisfiability upon the first conflict or construct a satisfying assignment on the first try for theory and Boolean variables.

Solver	SAT		UNSAT		overall	
Z3	5004	2.14 s	5099	1.63 s	10 103	87.9 %

Figure 2.6: Experimental results for Z3

The situation gets even worse if we look for problems where *actual solving happens* in the sense that we need all the (theoretic) power that the solver has. One could very well argue that benchmarks where solving hits (almost) no conflicts are essentially trivial – and we have only 2255 problem instances that show at least five conflicts.

2.6.2 Presentation of experimental results

Throughout the thesis, we show experimental results that analyze the impact of individual features or different configurations. We thus use a "minimal solver"– for example just a SAT solver and a CAD-based theory solver – without applying more involved preprocessing techniques.

All benchmarks presented throughout this thesis were run on Intel Xeon Platinum 8160 processors and were allowed to use up to four gigabytes of memory and two minutes of processor time. If run times tend to vary from one execution to the next, it is sometimes reasonable to perform multiple runs and present average values. All our solvers – and we assume this to hold for Z3 as well – are completely deterministic and, thus, the fluctuations of the needed computation time are marginal in practice.

Consider Figure 2.6 for an illustrative example of benchmark results. For every solver (here only Z3), we show the number of satisfiable and unsatisfiable instances that were correctly solved, as well as the average run time for the solved instances. Additionally, we show the overall number of solved instances and also give the percentage of solved instances from the overall benchmark set. In this case, Z3 solves 10103 instances (from these 5004 satisfiable and 5099 unsatisfiable) which are 87.9% of the benchmark set (that consists of 11489 instances altogether). On the remaining 1386 instances, Z3 exhausted the resources (either time or memory). No solver presented in this thesis yielded incorrect results.

CDCL-style SAT solving

One of the most basic logics is *propositional logic*. It exclusively deals with Boolean variables (sometimes called atomic propositions or propositional variables) and Boolean combinations thereof. While the formal syntax is one of the simplest available, it already proves to be very powerful: the satisfiability problem for propositional logic is one of the original NP-complete problems from [Kar72].

Definition 3.1: Propositional logic

Let $\mathcal{B}$ be the set of Boolean variables and $\mathbb{B} = \{\mathit{false}, \mathit{true}\}$ the Boolean constants. The syntax of a *propositional logic* formula φ is defined as follows.

$$\varphi ::= b \mid c \qquad b \in \mathcal{B}, c \in \mathbb{B}$$
$$\varphi ::= \varphi \vee \varphi \mid \neg\varphi$$

We assume that $\neg$ and $\vee$ have the common semantics of negation and conjunction. Further Boolean connectives, like disjunction or exclusive or, can be defined in terms of negation and conjunction as syntactic sugar.

Note that we do not allow for explicit quantification of variables – like for example *quantified Boolean formulae* (QBF) does – but rather push the quantification to the question we ask about the formula at hand. Given a propositional logic formula $\varphi(\overline{x})$, we aim to decide whether it is satisfiable – $\exists\overline{x}.\ \varphi(\overline{x})$ – and reuse this to decide whether it is a tautology – $\forall\overline{x}.\ \varphi(\overline{x}) \equiv \neg\exists\overline{x}.\ \neg\varphi(\overline{x})$.

Definition 3.2: Propositional satisfiability problem

Let φ be a propositional logic formula over variables $\overline{x}$. The *propositional satisfiability problem* (SAT) is to decide whether there is a variable assignment for $\overline{x}$ that satisfies φ, or more formally, whether the following statement is true:

$$\exists\overline{x}.\ \varphi(\overline{x})$$

Practical algorithmic approaches to this problem usually follow one of two paths that we identify as *enumeration* or *deduction*. The former works on the variable assignment and exploits that the number of variable assignments is finite, as the number of variables and the number of possible values for each variable is finite. Given this finiteness, we can simply try all possible assignments and eventually find a satisfying one or find that there is none. The latter uses logical reasoning on the formula to derive new knowledge about the solution space, aiming for a proof that there is no solution.

3.1 Satisfiability via enumeration

The arguably simplest approach to determine satisfiability is to try all possible assignments and check whether any of them satisfies the formula. A naive implementation of this idea is shown in Algorithm 3.1. Assuming n variables with two possible values each – *true* and *false* – we obtain at most 2^n variable assignments that we have to check as we can determine satisfiability as soon as one assignment is satisfying.

Algorithm 3.1: Solving the SAT problem via enumeration

```
1 Function Satisfiable(φ)
2     for every model α ∈ A do
3         if φ(α) then
4             return SAT
5     return UNSAT
```

We observe that the formula is hardly used here, in particular, we do not even attempt to make use of certain structures a formula may have. For example, if our formula has the form $\varphi = x_1 \vee \varphi'$, we could determine satisfiability by assigning $x_1 \mapsto \mathit{true}$ and avoid looking at a – possibly huge – remaining formula φ'. This idea of exploiting formula structure leads to the following recursive Algorithm 3.2.

Algorithm 3.2: Solving the SAT problem via recursive enumeration

```
1 Function Satisfiable(φ)
2     x := select unassigned variable from φ
3     for value ∈ {true, false} do
4         if Satisfiable(φ[x/value]) = SAT then
5             return SAT
6     return UNSAT
```

This version of enumeration essentially builds partial assignments one variable at a time and tries substituting all possible values (*true* and *false*) for some variable. The substitution allows to simplify the formula before extending the partial assignment and thereby could solve the above example $x_1 \vee \varphi'$ immediately – given that we select x_1 and assign it to *true*.

There are a few fundamental ideas that are already part of this very simple algorithm. Firstly, we try to make use of the structure of the formula at hand. Secondly, we reduce our problem to one or more simpler problems and inherit or combine the results. We see that even if we have no idea how to do a reduction, we can split into multiple cases – this is what we do in Line 3 of Algorithm 3.2 – and combine the results of all possible cases afterward. Thirdly, we have choices to make, for example, which variable to select or in which order to process our cases.

3.2 Satisfiability via deduction

Given the nature of enumeration, it is only suited for questions of satisfiability and can not be applied for more complex questions, for example, variable elimination. Deduction is a general approach that aims at inferring new facts that are implied by

the current formula. If we apply a (sound) deductive proof system, we construct a proof for such a new fact from our formula. We usually try to infer *false*, which proves unsatisfiability.

The most prominent example of such a deductive proof system is the (Boolean) *resolution proof system*. Though it only consists of a single rule – which is sometimes argued to be a *template* that can be instantiated to form arbitrarily many rules – it is a sound and complete proof system for propositional logic.

Definition 3.3: Resolution proof system

Assume two clauses that share some variable y, though with opposite polarity. The *resolution rule* states that

`Resolution`:

$$\frac{(x_1 \vee \cdots \vee x_i \vee y), (\neg y \vee z_1 \vee \cdots \vee z_j)}{(x_1 \vee \cdots \vee x_i \vee z_1 \vee \cdots \vee z_j)}$$

We say that we *perform resolution on* y and write $\mathit{Resolution}_y(\cdot, \cdot)$, sometimes skipping the variable we perform resolution on. The *resolution proof system* consists of only the `Resolution` rule.

Theorem 3.1: Soundness & completeness

The resolution proof system, as defined in Definition 3.3, is *sound* and *refutationally complete* for propositional logic.

Soundness. Let $\varphi = (x_1 \vee \cdots \vee x_i \vee y) \wedge (\neg y \vee z_1 \vee \cdots \vee z_j)$ and $\alpha \in \mathcal{A}$ such that $\alpha \models \varphi$. Thus, $\alpha \models (x_1 \vee \cdots \vee x_i \vee y)$ and $\alpha \models (\neg y \vee z_1 \vee \cdots \vee z_j)$. We consider two cases: $\alpha \models y$ and $\alpha \models \neg y$.
If $\alpha \models y$ we can infer $\alpha \models (z_1 \vee \cdots \vee z_j)$ and thus $\alpha \models (x_1 \vee \cdots \vee x_i \vee z_1 \vee \cdots \vee z_j)$. In the same way, if $\alpha \models \neg y$ we can infer $\alpha \models (x_1 \vee \cdots \vee x_i)$ and thus $\alpha \models (x_1 \vee \cdots \vee x_i \vee z_1 \vee \cdots \vee z_j)$. Hence, we get $\alpha \models (x_1 \vee \cdots \vee x_i \vee z_1 \vee \cdots \vee z_j)$ in both cases, and thus $\varphi \models (x_1 \vee \cdots \vee x_i \vee z_1 \vee \cdots \vee z_j)$. □

To prepare our proof for (refutational) completeness, we give Algorithm 3.3 that only uses the resolution rule to determine the satisfiability of a formula in CNF. The main idea – which seems to originate in [DP60] – is to eliminate one variable after another and eventually obtain a formula without variables, either *true* or *false*. It does so by selecting an arbitrary variable x from φ splitting the clauses in three parts: φ_x that contains x, $\varphi_{\neg x}$ that contains $\neg x$, and the rest φ_R. Note that no clause contains both x and $\neg x$ as such clauses are trivial tautologies that we (implicitly) remove from φ.

Refutational completeness. We proved that Algorithm 3.3 is sound for all CNF formulae. We observe that at the end of the loop, φ does no longer contain x as φ_R did not contain x in the first place and the clauses resulting from performing resolution on x do neither. Therefore, φ eventually contains no variables and we get either an empty set of clauses (*true*) or the empty clause (*false*). □

Algorithm 3.3: Solving the SAT problem via binary resolution

```
1  Function Satisfiable(φ)
2      for x a variable of φ do
3          φ_x := {c ∈ φ | x ∈ c}
4          φ_¬x := {c ∈ φ | ¬x ∈ c}
5          φ_R := φ \ (φ_x ∪ φ_¬x)
6          φ := φ_R ∪ {Resolution_x(c, c') | c ∈ φ_x, c' ∈ φ_¬x}
7      if φ ≡ true then
8          return SAT
9      else
10         return UNSAT
```

Note that Algorithm 3.3 does not immediately provide us with a satisfying assignment. We can, however, create a model based on the intermediate clause sets. Starting with an (initially empty) partial model, we substitute the model into the intermediate clause sets and consider the clauses that now only contain the next variable x.

We observe that there are only two (syntactically) possible clauses (x) and $(\neg x)$. If none are present, we choose any value for x. If a single one is present, we choose the appropriate value for x. The case that both are present is impossible, as it implies the presence of $(\neg\alpha, x)$ and $(\neg\alpha, \neg x)$ (which evaluate to (x) and $(\neg x)$), which would have allowed for the resolution with the result $(\neg\alpha)$, invalidating the partial model α.

Unfortunately, the elimination of a single variable may incur a quadratic increase of clauses which may even grow in size. Thus we may exhibit a doubly exponential time and space complexity.

3.3 Davis–Putman procedure

We have shown that Boolean resolution is an effective technique for working on propositional formulae on its own, though with limited efficiency. It is oftentimes only used as one of the ingredients, already [DP60] is actually concerned with deciding the satisfiability of *first-order logic* formulae – though with predicates without fixed semantics – and proposes a method that roughly works as described below.

Firstly, the formula is transformed into what we now call *Skolem normal form*: we introduce new function symbols and obtain a formula in *prenex normal form* with only universal quantifiers and (implicitly) ask for the existence of the *Skolem functions*. Secondly, all (countably many) possibilities to *ground* the formula are enumerated to instantiate the universally quantified variables. Finally, after every instantiation, we check for the satisfiability of the conjunction of all the ground formulae. If we eventually find one of the formulae to be unsatisfiable, this implies the unsatisfiability of the original formula.

For our purpose, we focus on the last step only that solves the satisfiability problem for propositional logic. Three rules are proposed in [DP60, Section 4] that are applied in a specific order and eventually derive either the empty clause or the empty formula, yielding *unsat* or *sat*.

Definition 3.4: Davis–Putnam procedure for propositional logic

Let φ be a propositional formula in CNF. We define three rules that we call `One-literal` rule, `Affirmative` rule and `Elimination` rule.

The `One-literal` rule comes in three flavors, all aimed at eliminating variables that occur in clauses that only have a single literal.

`One-literal:`

$$\frac{(x), (\neg x), \varphi}{false} \qquad \frac{(x), \varphi}{\varphi[x/true]} \qquad \frac{(\neg x), \varphi}{\varphi[x/false]}$$

The `Affirmative` rule is aimed at eliminating variables, that only occur positively (or negatively) throughout the whole formula.

`Affirmative:`

$$\frac{\varphi, \neg x \notin \varphi}{\varphi[x/true]} \qquad \frac{\varphi, x \notin \varphi}{\varphi[x/false]}$$

The `Elimination` rule is aimed at eliminating any variable by pairwise resolution, similar to Algorithm 3.3. Note that this rule technically uses non-CNF formulae, but can be implemented on clauses only easily.

`Elimination:`

$$\frac{(\varphi_x \vee x) \wedge (\varphi_{\neg x} \vee \neg x) \wedge \varphi_R, x \notin \varphi_R, \neg x \notin \varphi_R}{(\varphi_x \vee \varphi_{\neg x}) \wedge \varphi_R}$$

The algorithm presented in [DP60] employs these three rules as follows:

Algorithm 3.4: Davis–Putnam procedure for propositional logic

```
1  Function Satisfiable(φ)
2      while true do
3          if φ ∈ {true, false} then
4              return φ
5          if One-literal rule is applicable on φ then
6              Apply One-literal rule to φ
7          else if Affirmative rule is applicable on φ then
8              Apply Affirmative rule to φ
9          else
10             Select some variable x from φ
11             Apply Elimination rule to x in φ
```

This method is interesting to us, as it can be seen as a first combination of the two paradigms of enumeration and deduction. The `Elimination` rule implements the variable elimination idea that we have used in the purely deductive approach. The other two rules also eliminate variables in a deductive fashion, but in a more constructive way that allows to immediately extend a (partial) model.

3.4 Davis–Putnam–Logemann–Loveland procedure

Shortly after [DP60] put forward a theoretical algorithm that they used to solve the problem manually, they described their experience with an actual software implementation in [DLL62]. Though most of this work is actually concerned with considerations concerning the hardware available at the time, it contains a single change to the abstract algorithm which proved to be groundbreaking. Let us reconsider the `Elimination` rule from [DP60] which is used for eliminating a variable.

`Elimination:`

$$\frac{(\varphi_x \vee x) \wedge (\varphi_{\neg x} \vee \neg x) \wedge \varphi_R, x \notin \varphi_R, \neg x \notin \varphi_R}{(\varphi_x \vee \varphi_{\neg x}) \wedge \varphi_R}$$

During implementation, the authors realized that this rule has an important catch. Each clause from φ_x (and $\varphi_{\neg x}$) has one literal less than its origin from φ, but when they get combined in $\varphi_x \vee \varphi_{\neg x}$ the clauses tend to grow significantly in size. This effect can be mitigated by replacing the `Elimination` rule by a new `Splitting` rule:

`Splitting:`

$$\frac{(\varphi_x \vee x) \wedge (\varphi_{\neg x} \vee \neg x) \wedge \varphi_R, x \notin \varphi_R, \neg x \notin \varphi_R}{(\varphi_x \wedge \varphi_R), (\varphi_{\neg x} \wedge \varphi_R)}$$

This avoids the growth of clauses, indeed clauses will get smaller because the variable x is removed. However, it comes with the cost of yielding two sub-problems: $\varphi_x \wedge \varphi_R$ and $\varphi_{\neg x} \wedge \varphi_R$ that we have to check one after the other. There is an additional benefit, though: the `Splitting` rule – producing smaller clauses – tends to allow significantly more applications of the `One-literal` rule compared to the `Elimination` rule.

Algorithm 3.5: Davis–Putman–Logemann–Loveland procedure

```
1  Function Satisfiable(φ)
2      while true do
3          if φ ∈ {true, false} then
4              return φ
5          if One-literal rule is applicable on φ then
6              Apply One-literal rule to φ
7          else if Affirmative rule is applicable on φ then
8              Apply Affirmative rule to φ
9          else
10             Select some variable x from φ
11             Apply Splitting rule and obtain φ′, φ″
12             if ¬ Satisfiable(φ′) then
13                 return false
14             if ¬ Satisfiable(φ″) then
15                 return false
16             return true
```

This is what we know as the original *DPLL algorithm* that is still the (conceptual) basis of most modern SAT solvers. We observe that, conceptually, we only use the simplest forms of deductive reasoning here – the `One-literal` rule and the `Affirmative` rule. The `Splitting` rule can instead be seen as some kind of enumeration as it enumerates all possible values for the selected variable. In this sense, Algorithm 3.5 is one of the first effective combinations of deduction and enumeration.

3.5 Towards modern DPLL

The above Algorithm 3.5 has since been improved and extended in various ways that are standard in most of today's solvers. We now review the most important innovations.

We observe that the original version stores new formulae φ' and φ'' in memory whenever the `Splitting` rule is applied. Though they could replace the original formula φ, this still usually incurs a significant memory overhead, in particular, due to the recursive nature of this process. We can instead maintain the splits – and the results of the other rules – separately without actually changing the representation of the formula. We store them in what is commonly called a *trail*, essentially a list of variable assignments, and consider the formula with respect to the trail.

All three rules from [DLL62] essentially assign a value to a variable, though in a slightly different manner. The `One-literal` rule identifies a logical implication in the sense that the variable assignment is enforced by the formula at hand. The `Affirmative` rule identifies assignments that are sound in the sense that they are consistent with the formula. The number of models may reduce, though not to zero. In contrast, the `Splitting` rule is merely a guess and we may be forced to consider the other choice as well. Hence, we distinguish between two types of assignments that we call *propagations* and *decisions* as defined in Definition 3.5.

Definition 3.5: DPLL trail

The DPLL algorithm works on a what we call the *trail*. A trail M is a sequence of elements of one of the two forms:

L	Boolean decision of literal L
$C \to L$	Boolean implication of literal L due to a clause C

We call L a *decision* and $C \to L$ a *propagation*. We say that a literal L is *assigned to true* (*false*) if $L \in M$ ($\neg L \in M$), otherwise L is *unassigned*.

The notion of considering the formula *with respect to a trail* – or the partial assignment represented by the trail – changes how we apply the original DPLL rules. The `One-literal` rule only works on clauses with a single literal, but these only rarely exist if we do not change the formula. Instead, we consider the `Boolean constraint propagation` rule as given in Definition 3.6.

Definition 3.6: Boolean constraint propagation

Let M be some DPLL trail and $C = (L \vee L_1 \vee \cdots \vee L_k)$ a clause. The `Boolean constraint propagation` rule infers a variable assignment as follows:

Boolean constraint propagation:

$$\frac{M, C}{[\![M, C \to L]\!]} \quad \textbf{if} \quad \begin{array}{l} \neg L_i \in M \text{ for all } i = 1 \dots k \\ L, \neg L \notin M \end{array}$$

Given a trail M and a set of clauses $\mathcal{C}$ we define BCP to apply the above rule until *no suitable clause* $C \in \mathcal{C}$ exists anymore. We extend BCP to check for *conflicting clauses* – whose literals all evaluate to *false* – and let it return *false* if it ever finds such a conflicting clause and *true* otherwise.

The Affirmative rule proves to be rather expensive to be employed in practice. In order to check whether it can be employed, we need to establish a *global information* about the number of *unsatisfied clauses* containing a certain literal. Therefore it is usually not used in the core of modern DPLL-based SAT solvers. The Splitting rule creates what we called a *decision* and is given in Definition 3.7.

Definition 3.7: Decision

Let M be some DPLL trail and L a literal that is unassigned in M. The Decision rule decides upon a variable assignment as follows:

Decision:

$$\frac{M}{[\![M, L]\!]} \quad \textbf{if} \quad L, \neg L \notin M$$

Given a trail M, we define Decide to apply the decision rule *once* and return *true* if possible. If the rule is not applicable – because all literals are already assigned – it returns *false*.

Now that we store the current partial assignment in a trail, and that parts of this assignment are only guesses, we need a way to *undo assignments*. Note that this idea of undoing assignments is not fundamentally new: Algorithm 3.5 returns from checking the satisfiability of φ' and continues to explore φ'' afterward. We essentially only reformulate this process as an iterative procedure, compared to the recursive formulation of Algorithm 3.5.

Definition 3.8: Backtracking

Let M be some DPLL trail and such that some clause is *conflicting*, that means it evaluates to *false* under M. Furthermore, let us *tag decisions* with a Boolean flag, being *false* by default. We define *basic DPLL backtracking* as follows:

Backtracking:

$$\frac{[\![N_1, L_{false}, N_2]\!], C = (L_1 \vee \dots \vee L_k)}{[\![N_1, \neg L_{true}]\!]} \quad \textbf{if} \quad \begin{array}{l} \neg L_1, \dots, \neg L_k \in [\![N_1, L_{false}, N_2]\!] \\ L \text{ is the } \textit{rightmost} \text{ decision} \\ \text{that is } \textit{tagged} \text{ with } \textit{false} \end{array}$$

We define the Backtrack method to apply the above rule and return *true* if possible. Otherwise, that is if no such decision can be found and thus the conflict can not be resolved, it returns *false*.

The basic version of backtracking defined in Definition 3.8 exactly models the recursive exploration from Algorithm 3.5: it simply records whether the opposite branch of the search has already been explored – by default is has not – and also checks it.

Note that we assume BCP to work in an *exhaustive* manner. Thus, in any trail a decision is followed by *all* propagations that are possible. In particular, the last decision is always part of the reason for a conflict and changing this decision *always* resolves this particular conflict. Of course, changing the decision can still lead to another conflict, possibly even given by the very same clause, but the list of propagations is different.

Given these reformulations of the DPLL rules, and the specifications of BCP, Decide, and Backtrack, we now give what we call the *DPLL algorithm* in Algorithm 3.6.

Algorithm 3.6: DPLL algorithm

```
1 Function Satisfiable(φ)
2     while true do
3         while ¬BCP() do
4             if ¬Backtrack() then
5                 return false
6         if ¬Decide() then
7             return true
```

3.6 Conflict-driven clause learning

Many improvements to the DPLL algorithm from Algorithm 3.6 have been proposed that we summarize under the term *conflict-driven clause learning*, fully aware that clause learning is only one of the techniques we present. *Conflict-driven clause learning* – for short CDCL – is the common term used for "modern DPLL-style SAT solving" that comprises all the other techniques and clause learning is arguably the most important of them. A fundamental term for CDCL is what we call a *decision level*.

Definition 3.9: Decision level

Let M be some trail and M_D the subsequence of M consisting only of the decisions from M. We call $|M_D|$ the *current decision level of* M and sometimes write $DL(M)$.
Let M_k be the longest prefix of M that has decision level k. We say that some literal L *has decision level* k if $L \in M_k$ but $L \notin M_{k-1}$, and *something holds at decision level* k if it holds for M_k. We sometimes write $DL(L)$ and $DL(C)$ for the largest decision level of some literal from a clause C.

3.6.1 Clause learning

We have seen in Definition 3.8 that having a clause evaluate to *false* somehow implies the negation of the last decision, though the premise was somewhat vague. Essentially, we have seen that a decision L *did not work out*, and thus – under the assumptions represented by the preceding part of the trail – either $\neg L$ is correct, or said preceding part of the trail was already incorrect. *Clause learning* aims to make this reasoning *explicit* and *persistent* so that it can be reused later on.

Assume a trail $[\![N_1, L, N_2]\!]$ where L is the rightmost decision and some clause C is conflicting. The goal is to construct a new clause D that represents the logical implication $N_1 \implies \neg L$ from C and the information contained in N_2. Conceptually, we can understand the task as *eliminating variables from* C such that only variables from N_1 remain.

Algorithm 3.7: Conflict resolution

```
1 Function ConflictResolution(M, C)
2     while M ≠ [[]] do
3         T := removeLastFrom(M)
4         if T = E → L and ¬L ∈ C then
5             C := Resolution_L(C, E)
6         if T = L and ¬L ∈ C then
7             break
8     return C
```

The given Algorithm 3.7 achieves exactly what we described before: it works its way backward through the trail, eliminates literals from the conflict clause – by using Boolean resolution – until we find a decision literal whose negation is part of the clause. It also already performs the backtracking, exactly as Algorithm 3.6 did.

Note that we usually generalize this method: instead of generating a clause that implies the negation of some decision we allow to learn *any* clause that is conflicting on the current trail and implies *some literal* after appropriate backtracking. We would usually stop the resolution process in Algorithm 3.7 as soon as C contains only a single literal from the *last decision level*, thereby implying this single literal after we have backtracked the whole decision level. We call such a clause *asserting* and the first literal where the clause becomes asserting is called the *first unique implication point* [MMZ+01].

The backtracking checks whether the resulting clause is the empty clause, in which case it returns *false*, or returns the clause for propagation. We can go a step further and *learn* the clause: we add it to the set of clauses $\mathcal{C}$ and thus can use it for propagation at any point in the future. Slightly varying schemes for clause learning have been introduced in [SS96] and [BS97].

Algorithm 3.8: Backtracking with clause learning

```
1 Function Backtrack(M, C)
2     D := ConflictResolution(M, C)
3     if D ≠ false then
4         C := C ∪ {D}
5     return D
```

3.6.2 Non-chronological backtracking

We observe that (intuitively) propagations are *never harmful* because they only implement logical deductions that are sound. We never have to backtrack *because of* a propagation, they may only show that a decision was wrong so that we have to backtrack *because of that decision*. Decisions, on the other hand, are hazardous: they

are just guesses and a bad guess may induce any amount of pointless calculations. This is already reflected to some degree in our algorithm, as we eagerly perform propagation exhaustively and only resort to decisions afterward.

When learning a new clause, like when using backtracking with clause learning, it might be the case that the new clause is already unit at an earlier point in the trail.

Example 3.1: Early propagation of learned clauses

Let $\varphi = (\neg a \vee \neg c \vee d) \wedge (\neg c \vee \neg d) \wedge \varphi'$ where φ' also contains a Boolean variable b. Assuming the trail $M = [\![a, b, c, (\neg a \vee \neg c \vee d) \to d]\!]$, `BCP` finds the conflicting clause $C = (\neg c \vee \neg d)$.
Algorithm 3.7 would now remove $(\neg a \vee \neg c \vee d) \to d$ from the trail and compute the new conflict clause $C \coloneqq (\neg a \vee \neg c)$. In the next step, it finds the decision c with $\neg c \in C$ and, therefore, returns C while the remaining trail is $M = [\![a, b]\!]$. We observe that $C = (\neg a \vee \neg c)$ is now unit and `BCP` propagates $C \to \neg c$ in the next step. However, we could backtrack even further and – relative to the search space exploration – utilize the new propagation *earlier*. In this case, we backtrack to $M = [\![a]\!]$ and already then use C to obtain $M = [\![a, C \to \neg c]\!]$.

We can see in Example 3.1 that the clause that results from Algorithm 3.7 can be unit on a *smaller decision level*. Hence, it seems reasonable to already use this propagation on a smaller decision level in the hope to change how the search proceeds (in a positive way). Of course, backtracking further undoes some work that we may just repeat, but experimental experience shows that this potentially redundant work is worth it.

It remains to determine *which decision level to backtrack to*. Given that we want to utilize the clause C for propagation, C must still be unit and thus all but one literal must be assigned. If we consider the set of decision levels of the literals of C, we observe that conflict literal $\neg L$ is the single literal at the highest decision level (say l). Let now $k < l$ be the largest decision level present in C except for $\neg L$.

As all literals from C – except for $\neg L$ – have a decision level of at most k, C is still unit at decision level k. While the above methods only backtrack to decision level $l - 1$, we can thus also backtrack to decision level k – actually any decision level between k and $l - 1$. This is what we call *non-chronological backtracking*, first applied to satisfiability checking in [SS96], and shown in Algorithm 3.9.

Algorithm 3.9: Non-chronological backtracking with clause learning

```
1 Function Backtrack(M, C)
2     C ≔ ConflictResolution(M, C)
3     if C ≠ false then
4         𝒞 ≔ 𝒞 ∪ {C}
5     Backtrack to DL(C \ {¬L})
6     return C
```

3.6.3 Watched literal scheme

Boolean constraint propagation proves to be incredibly powerful in practice but heavily depends on the ability to perform it efficiently. In particular, it is not feasible – for large formulae – to linearly scan the set of clauses to check whether some clause is unit.

Instead, we need some way to quickly obtain a (small) set of clauses that *might be unit*.

One technique that has significantly improved performance of this across the board is the *two watched literal scheme* introduced in [MMZ+01]. The fundamental idea is based on the following two observations: a clause can only become unit if 1. it had (at least) two unassigned literals and 2. one of these has then been assigned to *false*. Note that we conveniently ignore clauses with a single literal here.

Therefore, we *watch* two literals from each clause and have a closer look at this specific clause only when one of these literals is assigned to *false*. In practice, we maintain a mapping from every literal to a set of clauses and put every clause in the list of *two* of its literals. Whenever we assign a literal L (either by decision or propagation), we review the clauses from the *watch list of* $\neg L$, distinguishing whether the other watched literal which is 1. unassigned, 2. assigned to *true*, or 3. assigned to *false*.

If the other watched literal is assigned to *true*, the clause is satisfied and there is nothing to do. If the other watched literal is not yet assigned or assigned to *false*, the clause *could* be unit. We consider the remaining literals of this clause and move the watch if we find another literal that is either unassigned or assigned to *true*. If such a literal does not exist, all but the other watched literal are assigned to *false* and the considered clause is either suitable for propagation (if the other watched literal is not yet assigned) or conflicting (if the other watched literal is assigned to *false*).

We note that the actual implementation of this scheme is very technical and must be done in a way that is compatible with the backtracking in that the watches must end up in some consistent state after undoing assignments. As this is highly specific to the actual implementation, we refer to [MMZ+01] for more details.

3.6.4 Restarts and clause removal

Most of the above techniques aim to accumulate more and better clauses and make use of available clauses faster. Practical experiments showed, however, that *bad decisions* early on in the search sometimes lead to a particularly hard part of the search space. Allowing the solver to *restart* – clearing the trail, but retaining the learned clauses – provides the opportunity to switch to another, easier part of the search space.

The idea to regularly restart the solver is usually attributed to [GSK98], though it was combined with randomization of the solving process. The concept of restarting the solver was then decoupled from randomization – for example in [MMZ+01] – and subsequently the frequency of restarts was made adaptive in [ES03]. Note that performing restarts without randomization only makes sense if the *variable ordering* – that we discuss in the next section – is dynamic. Otherwise, the solver simply replays the very same decisions and propagations after a restart.

Given that a clause is added to the clause database for every conflict, we might collect a large number of clauses that eventually slow down the solver. Furthermore, many of these clauses may be redundant – not only in the theoretical sense (because all learned clauses are redundant), but in the practical sense that we never use them again.

Assume that some part of the search space (say $[\![a \mapsto \mathit{false}, b \mapsto \mathit{false}]\!]$) has been excluded (by $(a \vee b)$) but it required several intermediate conflicts to construct this clause, for example $(a \vee b \vee c)$, $(a \vee b \vee \neg c \vee d)$ and $(a \vee b \vee \neg c \vee \neg d)$. We observe that the (smaller) clause $(a \vee b)$ is a much *stronger* statement than the other clauses, and it

might make sense to *forget* the other (larger) clauses. Actual schemes how to determine which clauses to remove are discussed in [MMZ+01] or [ES03].

Regular restarts, in particular in conjunction with clause removal, have the potential to lead to nontermination. If all clauses that were learned since the last restart are removed – and there is no reason why this is not possible in general – we may only repeat the same reasoning over and over again without any progress that persists beyond the next restart.

The common solution is already mentioned in [MMZ+01]: the periodicity of restarts is increased over time so that the whole solving process eventually fits in between two consecutive restarts. One variant that was proposed in [Hua07] and has become very popular since is to use a *Luby sequence*, which intuitively alternates frequent restarts and increasingly long periods without a restart.

3.6.5 Decision and value heuristics

The methods we have described give rise to several heuristic choices, most prominently the *choice of a decision variable*, but also which polarity to decide for, when to restart, or which clauses to forget.

The importance of the heuristic for selecting the decision variable – usually called *decision heuristic* or *branching heuristic* – was recognized early on and many different approaches were presented. Prominent examples are heuristics due to Böhm [BB92], the *MOM heuristic* [Fre95], *dynamic largest individual sum* [SS96], or *Jeroslow-Wang* [JW90]. All of them fall in one of two classes: they are either *static* – computed once at the beginning – or *dynamic* – computed for every decision. While the static ones are very fast, the dynamic ones tend to make better decisions.

Eventually, a class of heuristics that we call *quasi-static* was proposed that is both adaptive and very fast when making a decision. First presented in [MMZ+01] and subsequently improved in [ES03], we call them (exponential) *variable state independent decaying sum* (VSIDS). VSIDS maintains an *activity* for every variable that is *increased* when this variable is part of a conflict and decays over time, such that recent conflict have a higher weight than older conflicts.

Though it may seem counter-intuitive to aim for conflicts – instead of satisfying assignments – the practical performance of VSIDS is sufficiently convincing so that essentially all state-of-the-art solvers employ some variant of VSIDS nowadays. Nonetheless, improvements are still developed, for example in [LGZ+15] or [LGP+16], mostly trying to exploit structural properties of the formula (like the “community structure”) or operational properties of the solver (like “global learning rate”).

Satisfiability modulo theories solving

We have seen how algorithmic approaches to decide the satisfiability of a propositional logic formula have developed and improved over the years, starting from [DP60] to the current state-of-the-art `CDCL` with a large amount of further techniques and heuristics. We note, however, that these developments have oftentimes been driven by the interest in richer logics like *first-order logic*, not least the original work from [DP60].

Aiming for automatic solvers for these richer logics, three fundamentally different approaches have emerged. Before the rise of efficient SAT solvers, the predominant approach was to work solely on a set of theory constraints and not bother with the Boolean structure. The prime examples for this are arguably simplex-based linear programming solvers, for example, Gurobi or SCIP. In many cases, a certain amount of Boolean structure can be encoded within the theory as well, which is for example known as *0-1-programming*.

As soon as the propositional satisfiability problem could be solved (sufficiently) quickly, it was leveraged to help solving richer logics. The early approaches – that we summarize as *eager SMT solving* – transform the problem at hand into an equisatisfiable propositional logic formula and solve that one with a SAT solver. At some point, *lazy SMT solving* emerged that uses a SAT solver to process the Boolean structure first and issues *theory queries* (concerning only conjunctions of theory constraints) to dedicated theory solvers.

4.1 Eager SMT solving

The first approaches that employed SAT solvers for richer logics used them as simple black-boxes. In [SSB02], for example, separation logic formulae are encoded into propositional logic and then solved by some SAT solver (in this case Chaff). This approach is particularly straightforward in that no modifications to the SAT solver are necessary and, thus, the core solving engine can be replaced without any problems.

Definition 4.1: Eager SMT solving

Let S be a SAT solver and T a theory reasoning engine that transforms an input formula with some theory into an equisatisfiable propositional logic formula. We call a solver that first transforms the input to propositional logic using T and then determines the satisfiability using S an *eager SMT solving* program.

This approach somewhat restricts what we can argue about in a meaningful way. If the *richer logic* is also *more expressive*, we should not expect an encoding into propositional logic to exist – at least not a *nice* one, whatever *nice* may be. Theories that argue about real numbers are prime examples where a propositional encoding is notoriously difficult or impossible.

We observe that a SAT solver only has a finite state space: it only reasons about the 2^n possible variable assignments for n propositional variables from the input formula. For a real variable, however, a single variable induces an infinitely large state space. Even if we argue that we could do some kind of discretization, it is unclear how to obtain a suitable discretization and how large its state space would turn out to be.

It is important to realize that this property of *being reducible to propositional logic* is not connected to the asymptotic complexity of the problem at hand, all the more if we aim for practical efficiency. A set of linear real arithmetic constraints – without Boolean combinations – can be solved in polynomial time and thus asymptotically faster than propositional logic, but oftentimes the fastest method in practice is the simplex method – with exponential worst-case complexity – and encoding to propositional logic is not sensible, as discussed above.

Note that the reduction to propositional logic is – in theory – possible for every decidable logic. We can *simply solve the problem* and give the propositional encoding *true* or *false*. We consider it obvious that there is no point in doing this, though. If the theory admits *quantifier elimination* – for example cylindrical algebraic decomposition – we could use that to obtain the (trivial) encoding.

We can, however, also use it to decompose the state space into finitely many regions – this is what the cylindrical algebraic decomposition does after all – and confer on the SAT solver to choose a region and make sure that this region satisfies the Boolean structure. Note that it is probably worthy of discussion whether this would be considered *eager SMT solving*, as also considering the Boolean structure could easily be integrated into the theory reasoning part – the original cylindrical algebraic decomposition works on arbitrary formulae.

Though our common perception of *eager SMT solving* is that we transform the formula and issue a single call to the SAT solver, we can very well craft somewhat more complex schemes in the spirit of *eager SMT solving*. Consider for example the approach to solve first-order logic from [DP60] – which produces a sequence of SAT problems – or modern CEGAR-style approaches like in [CKS+04], which can arguably be described as *eager methods* to answer *SMT-like* questions.

4.2 Lazy SMT solving

The *eager SMT solving* scheme quickly became infeasible for richer logics and alternative structure started to emerge towards the end of the 1990s. In [WW99] the authors essentially describe what we now call a *lazy SMT solver* using the simplex method as a theory solver and even denominate some requirement that we, later on, subsume as *SMT compliancy*. Other examples of early solutions that use schemes similar to *lazy SMT* include [ACG99; MR02; BDS02] and we refer to [BHM+09] for more details on both the history and a properly thorough discussion of lazy SMT solving, as we only give an intuitive explanation now.

What we consider the modern *lazy SMT solving* approach consists of a slightly modified SAT solver and a theory solver that solely works on *sets of theory constraints*. The SAT solver works on a *Boolean abstraction* of the input formula and enumerates models for this Boolean abstraction. These models are translated to sets of constraints that should be consistent – if the model translates to the existence of theory assignment – which is what the theory solver decides upon. If the theory solver deems such a set of constraints consistent, we have proven that the input formula is satisfiable.

Definition 4.2: Boolean abstraction

Let A_T be the set of all theory atoms from some theory T and $\bar{b}$ a set of fresh Boolean variables. We call an injective function $B_T : A_T \to \bar{b}$ an *abstraction function* and extend it to arbitrary formulae by point-wise application.
Let φ be some formula over a theory T. We call $\varphi_{abs} = B_T(\varphi)$ the *Boolean abstraction* (or sometimes *Boolean skeleton*) of φ.

The *Boolean abstraction* is an *over-approximation* of the input formula φ in the sense that it ignores the underlying theory and only retains the Boolean structure of the formula. The resulting formula φ_{abs} is now processable for a regular SAT solver which can search for a model. Note that this satisfying *Boolean assignment* is not a model for the input formula φ but only for φ_{abs}.

Definition 4.3: Theory concretization

Let φ be some formula over a theory T, $\varphi_{abs} = B_T(\varphi)$ the *Boolean abstraction* of φ, and $\mathcal{A}$ a model for φ_{abs}. We define the set of theory atoms that correspond to $\mathcal{A}$ as

$$C_\varphi(\mathcal{A}) = \{a \mid \mathcal{A}(B_T(a)) = \mathit{true}\} \cup \{\neg a \mid \mathcal{A}(B_T(a)) = \mathit{false}\}$$

and call $C_\varphi(\mathcal{A})$ the *concretization of* $\mathcal{A}$ *over* φ.

The SMT solver needs to check whether the Boolean assignment corresponds to a feasible *selection of theory atoms*. In order to do this, we *concretize* the Boolean assignment – essentially we apply the inverse of the abstraction – and feed the resulting set of theory atoms to a *theory solver*. This theory solver determines whether a set of theory atoms is *consistent*, that is whether there is a theory model that satisfies them.

If the theory solver confirms that the set of theory atoms is consistent, we determine the satisfiability of φ. Otherwise, the current Boolean assignment can not be transformed into a theory model – and is not a *model* in this sense – and the SAT solver searches for another Boolean assignment. Note that the theory solver oftentimes provides the SAT solver with a *reason for unsatisfiability* – usually a subset of the set of theory atoms – which the SAT solver uses as a new *conflict clause* to perform regular conflict analysis.

Once the SAT solver can not find any further satisfying Boolean assignments – all Boolean assignments have been rejected by the theory solver – we determine unsatisfiability. Note that a *proof for unsatisfiability* is much more difficult than just giving a model in the case of satisfiability. We not only have the Boolean reasoning which essentially consists of the resolution steps, but also the theory reasoning that rejected all Boolean assignments.

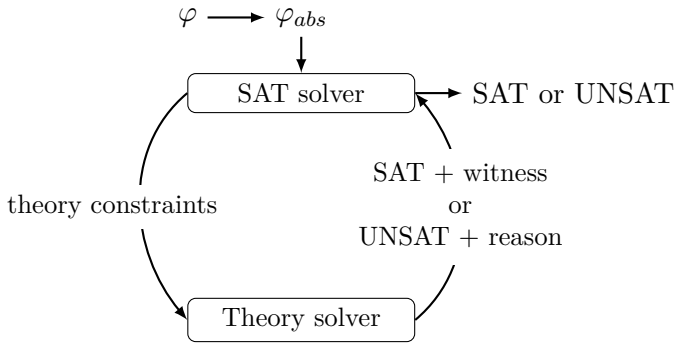


Figure 4.1: Schematic overview of `CDCL(T)`-style SMT solving

The described scheme for an SMT solver is completely agnostic of the theory that we try to solve, as long as we have a theory solver that can determine the satisfiability of a set of theory atoms. We call this *`CDCL(T)`-style SMT solving*, alluding to the fact that we usually use a `CDCL`-style SAT solver and a theory solver for some theory T. See Figure 4.1 for a schematic overview. Further functionality, like providing a theory model or a reason for unsatisfiability, can greatly improve practical performance but is not strictly necessary.

We usually distinguish two variants of lazy SMT solving, namely, *full-lazy SMT solving* and *less-lazy SMT solving*, which differ in when (or how often) the theory solver is called. A full-lazy SMT solver only issues a theory call if the SAT solver obtained a *full Boolean model*, that is if all Boolean variables have been assigned and the Boolean abstraction is satisfied. In contrast, a less-lazy SMT solver issues theory calls more frequently – for example after every decision level of the `CDCL`-style SAT solver – on a *partial Boolean model* if the Boolean abstraction is not yet conflicting.

We observe that the two variants only differ if the theory call returns feasibility: if the Boolean model is a full assignment we derive satisfiability of the whole formula if it is however only partial we need to let the SAT solver continue.

Note that these variants have a nontrivial trade-off. Less-lazy SMT solvers issue more theory calls and require a deeper integration into the SAT solver. The hope is that the additional theory calls are not that expensive – due to what is called "incrementality" and discussed in the following – but instead allow to detect conflicts earlier, possibly avoiding certain harder theory calls altogether. In practice, less-lazy SMT solving seems to beat full-lazy SMT solving, at least for "easier" logics like linear arithmetic.

4.3 SMT compliancy

We discussed the fundamental requirement for theory solvers – being able to decide upon the consistency of a set of theory atoms – but also mentioned some advanced properties. While these are not strictly necessary, they can greatly enhance applicability or practical performance. The common term *SMT compliancy* usually only subsumes *incrementality*, *backtracking* and the generation of *reasons for unsatisfiability*. However, we feel that we expect more properties from a theory solver that are not necessarily self-evident, and other more advanced properties can be very convenient.

4.3.1 Automation

We usually understand the task of SMT solving as a completely automated process: given an input formula, we derive whether the formula is satisfiable with no user interaction, requiring the theory solver to be completely automated as well. This is in stark contrast to approaches like *interactive theorem proving* that rely on the user to impose the general solving strategy. We instead need all components to bring about decisions on heuristics on their own.

4.3.2 Soundness & completeness

We have not (yet) allowed a solver to *fail*, either by providing an *incorrect result* or being *unable to answer* the posed question. Statistical approaches for the question of satisfiability exist, and most of them may be incorrect – though with a small probability.

More commonly, however, certain methods realize that they can not solve a certain problem or do not terminate in certain cases. For example, *virtual substitution* may abort if the degree of some polynomial grows and *Gröbner bases* may fail to prove either satisfiability or unsatisfiability. As for termination, regular implementations of *branch and bound* have well-known cases that lead to infinite sequences of branching.

The presented `CDCL(T)` scheme is commonly enhanced such that theory solvers may also return *unknown*, making the SAT solver skip this particular assignment. If we find the formula to be satisfiable later on, the one *unknown* answer can be ignored. Otherwise, the solver as a whole is forced to return *unknown* as well, as we can not be sure whether the formula is satisfiable.

4.3.3 Model generation

Though we formally ask for the *satisfiability* of some formula, we usually want to have a model in case the formula is indeed satisfiable. For most theory solvers, we can simply read off the model when we determine satisfiability, for example, when using simplex, interval constraint propagation, virtual substitution, or cylindrical algebraic decomposition. In other cases, we have to construct a model separately, though the effort is very low as, for example, for Fourier–Motzkin variable elimination.

We thus essentially assume all theory solvers to support the generation of a model. Of course, some exceptions exist: most notably this is the case for Gröbner bases, though some possible approaches have been proposed in [Jun12].

4.3.4 Reasons for unsatisfiability

When a theory solver determines that the given set of theory constraints is unsatisfiable, the SAT solver uses this fact to advance its own search. We construct a new clause encoding that the theory query $\bigwedge c_i$ is inconsistent and obtain $\bigvee \neg c_i$ – at least one constraint needs to be *false*. Assume the theory query involved k of n theory constraints, this clause *excludes* 2^{n-k} possible assignments from the Boolean search space.

Of course, we would like to exclude as many assignments as possible. If we can find a *subset* of the theory constraints that is already unsatisfiable, we can construct a smaller clause. This is a benefit in itself, but also excludes more possible assignments. While we can find such subsets in theory – we can simply try all of them – it heavily depends on the theory solver whether there is an efficient way to do so.

We call such subsets *infeasible subsets* and are usually interested in a (locally) minimal subset. To find a (globally) minimum subset is a hard problem in most cases and not worth the effort in practice. Hence, we usually use the term *minimal infeasible subset*. Note that in general, neither *minimal* nor *minimum* infeasible subsets are unique.

Definition 4.4: Minimal infeasible subset

Let C be a set of theory constraints with $C \models \mathit{false}$. We call C *infeasible* and $M \subseteq C$ with $M \models \mathit{false}$ an *infeasible subset (of C)*. We say that M is *minimal* if there is no $M' \subsetneq M$ with $M' \models \mathit{false}$. If M is of *minimal cardinality* among all infeasible subsets, we call M a *minimum infeasible subset*.

We have two fundamentally different approaches to obtain an infeasible subset that mainly depend on the *locality* of the underlying conflict. We can either enhance the algorithm at hand in a way that (more or less) immediately yields an infeasible subset, or we can mount an a-posteriori analysis in a second stage.

The simplex method, for example, finds unsatisfiability with a local criterion – when some variable can not be pivoted – which conveniently yields a local conflict that can be used as an infeasible subset as is described for example in [MB08b]. The CAD method, on the other hand, only determines unsatisfiability when it exhausted all possible solution candidates, essentially leaving us with no particular information about the conflict.

4.3.5 Incrementality

In the context of lazy SMT solving – in particular less-lazy SMT solving – a theory query is not an *isolated computation*. We can rather understand the sequence of theory queries as an ever-changing problem that is only *extended* or *restricted* by the SAT solver by adding (or removing) constraints to the current set of constraints.

If the theory solver manages to retain information from the previous theory call and merely extends its internal state when a new constraint is added, we can significantly improve practical performance. Gröbner Bases make for a nice example here, as they essentially only combine existing polynomials to construct new ones until a fixed point is reached. When adding a new polynomial, we can simply start from the result of the previous computation without having to replicate any work we have already done.

4.3.6 Backtracking

Backtracking is the inverse process of incrementally adding constraints, we remove individual constraints from the current set of constraints. Again, we can benefit if we manage to retain as much information as possible. This usually means that we have to track which pieces of information originated from which constraints and consequently remove only what originated from the constraints to be removed.

In most cases, this is arguably the most difficult part when integrating a decision procedure into SMT. It oftentimes involves a lot of bookkeeping and usually has certain trade-offs between the granularity or specificity of tracking the origins of data and the computational overhead. The best strategy is very specific to the decision procedure at hand, and can be anything from "merely disabling" certain parts of the state like for simplex [DM06] to completely removing currently unused data as proposed in [KÁ20].

The term "backtracking" oftentimes implicitly means *in-order* backtracking. It assumes that constraints are added in a certain order and the removal happens in the inverse order. This essentially matches how a CDCL-style SAT solver manages assignments on a trail. We note, however, that practical examples exist that motivate *out-of-order* backtracking, allowing for the removal of constraints in an arbitrary order.

Though out-of-order backtracking paves the way for great additional savings, as we can see in the following Example 4.1, it also proves to be a burden on some theory solvers that might exploit the ordering of constraints otherwise.

Example 4.1: Benefits of out-of-order backtracking

Consider the following first-order logic formula over a real variable x with p some *difficult* polynomial and its abstraction below.

$$(x < 0 \vee x > 0) \wedge (x \geq 0 \vee p > 0) \wedge (x \leq 0 \vee p > 0)$$

$$(a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee c)$$

A typical CDCL-style SAT solver would assign $\neg a, b, c$ and issue a theory call with $\{x \geq 0, x > 0, p > 0\}$. Assuming that this theory call yields UNSAT, it would (after conflict resolution) learn the new clause (a) and backtrack *all assignments* to revise the decision $\neg a$. Hence, the next assignments would be a, c and the following theory call $\{x < 0, p > 0\}$.
Note that $p > 0$ was removed and then added again, just so that the SAT solver can maintain its trail properly. Instead, we could have removed $x > 0$ and $x \geq 0$, added $x < 0$, and not changed $p > 0$. Assuming that $p > 0$ gives rise to a lot of difficult computation in the theory solver, this simple change could provide significant improvements, given that the theory solver can exploit it.

4.3.7 Lemma generation

Most methods that we use within a theory solver provide more information than we need to decide upon the satisfiability. It is oftentimes possible to extract some pieces of information and formulate them as a *lemma* – or a *tautology* – that can be used to *teach the SAT solver* about the theory. Assuming $x^2 < 0$ being a literal in our formula, we could teach the SAT solver that $\neg(x^2 < 0)$ is a tautology and thus the literal must be assigned to *false*, possibly independent of any specific theory query.

4.4 Common theory solvers

Finally, we want to give a rough overview of the techniques that are typically used for theory solving in modern SMT solvers. Note that the topic of this thesis – nonlinear arithmetic – is both comparably new and uncommon in the SMT community, and thus the options we have discussed in Section 1.1 only apply to very few other SMT solvers.

The better part of the SMT community is concerned with linear arithmetic or completely different theories like *uninterpreted functions* or *bit-precise data types*. We briefly describe them in the following and refer to [KS08] for more explanation and further literature on these topics.

4.4.1 Uninterpreted functions

The theory of *(equalities and) uninterpreted functions* introduces function symbols without any semantic other than *being a function*. They are commonly used to *abstract* from the inner workings of some function or to identify possible implementations for a given specification.

This theory is commonly solved by what is called *congruence closure*: forming equivalence classes based on the equalities and checking them against any disequalities afterward. Though this approach is arguably simple, actual implementations require quite some considerations to make it efficient in practice. Note that this theory predates the classical SMT framework and thus some common techniques exist that work outside of the SMT framework, notably the *Ackermann reduction* and *Bryant's reduction*.

4.4.2 Bit-precise data types

In particular, if SMT solvers are used for software verification, it is desirable to be able to model data in a bit-precise way. Accordingly, theories for fixed-width integers – commonly known as "bit-vectors" – or fixed-width floating-point numbers exist. While bit-vector arithmetic has long been supported by many SMT solvers, floating-point arithmetic has only recently found more widespread adoption, most notably since it was added to the SMT-LIB standard after [RW10].

Though the solving techniques for both are extremely similar – both are ultimately reduced to propositional logic in most cases – floating-point arithmetic has proven to be significantly more difficult in practice for two reasons. While one can oftentimes ignore most bits for bit-vector arithmetic (assuming that a satisfying model is usually "small") this is not the case for floating-point arithmetic. As the interrelations between different bits are much more involved for floating-point arithmetic, one typically needs to consider more bits and thus practical run times are much larger. Secondly, floating-point arithmetic contains way more theory operations – for example, square roots or fused multiplication and addition – and technical details that need to be considered, like different rounding modes or values for infinity or "not a number".

4.4.3 Linear arithmetic

The most common theory that incorporates real arithmetic is linear real arithmetic. Given its long history in *linear optimization* (or *operations research*) many applications are routinely formulated as linear real formulae and, thus, it is an important theory for the SMT community as well.

The predominant decision procedure is the simplex method and most current solvers use an integration in the spirit of [DM06]. Other methods, like Fourier–Motzkin variable elimination, are mostly used for preprocessing or in special contexts like the one we describe in Section 8.3.3. More specialized methods exist for restricted theories like *difference logic* that even found its way into SMT-LIB [BFT16] as a separate theory.

4.4.4 Theory combination

One sometimes wishes to *combine* multiple theories, usually because the system that one wants to analyze makes use of multiple different formalisms: for example, software may use both native integers and floating-point numbers. Furthermore, uninterpreted

functions are sometimes used to *abstract* from functionality that is either not relevant or very hard to argue about, for example, during linearization as described in [CGI+18].

The goal is to have a framework that allows the modular combination of two (or more) theory solvers for individual theories into a single theory solver for the combined theory. The first such framework is commonly called *Nelson-Oppen theory combination* and goes back to [NO79], but unfortunately, it imposes severe restrictions on the involved theories that are rather hard to lift. More recently, a more flexible approach called *delayed theory combination* was proposed [BBC+05; MB08a] that lifts most of these and is thus used by most modern SMT solvers.

4.5 CDCL(T) as a proof system

It is sometimes convenient to formulate methods like CDCL(T) differently than as an algorithm. If we want to argue about certain theoretical properties, we oftentimes use a more declarative approach that is somewhat similar to the proof rules from Section 3.2. This gives us a *CDCL(T) proof system* like presented in [NOT06].

We define CDCL(T) such that its proof rules work on a *state*, consisting of what we call a *trail* – essentially recording the (partial) Boolean assignment – and the set of clauses that the solver is arguing about. As for the trail, we simply reuse the DPLL trail from Definition 3.5, which slightly differs from [NOT06] in that it records the clause that leads to a propagation, though it is not explicitly used in the subsequent proof system. Furthermore, we allow this clause to be *constructed lazily* as described in [NOT06, Section 5]. Note that the DPLL trail contains *literals* which may now be *theory atoms* instead of *Boolean variables* (or their negation) as well.

Definition 4.5: CDCL(T) state

Let M be a *DPLL trail* as specified in Definition 3.5. We call the combination of a DPLL trail and a set of clauses $\mathcal{C}$ a *CDCL(T) state* and write $\langle M, \mathcal{C} \rangle$.

For CDCL(T), we strictly distinguish between Boolean reasoning and theory reasoning where the proof system we present is only concerned with Boolean reasoning. If we write $M \models C$ for a clause C, we claim that $L \in M$ for some literal $L \in C$ – a statement arguing only about the Boolean state contained in the trail. In contrast to that $M \models_T C$ denotes *entailment in the theory* T.

The set of rules presented here is taken from [NOT06], but we allow for strong theory derivations as presented in [RKG18], which essentially turns the T-Learn rule into the T-Learn* and gives us the stronger *CDCL*(T) proof system*. Though we change the notation a bit, we feel that the equivalence of the rules (to [NOT06]) is straightforward.

Definition 4.6: CDCL(T) and CDCL*(T) proof systems

The *CDCL(T) proof system* consists of the following proof rules.

Decide:

$$\frac{\langle M, \mathcal{C} \rangle}{\langle [\![M, L]\!], \mathcal{C} \rangle} \quad \textbf{if} \quad \begin{array}{l} L \text{ or } \neg L \text{ occurs in } \mathcal{C}, \\ L \text{ is undefined in } M \end{array}$$

Fail:

$$\frac{\langle M, \mathcal{C} \cup \{C\}\rangle}{FailState} \quad \textbf{if} \quad \begin{array}{l} M \models \neg C, \\ M \text{ contains no decision literals} \end{array}$$

UnitPropagate:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![M, C \to L]\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} C = D \vee L \in \mathcal{C}, \\ M \models \neg D, \\ L \text{ is undefined in } M \end{array}$$

TheoryPropagate:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![M, (D \vee L) \to L]\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} M \models_T D \vee L \text{ and } M \models \neg D, \\ L \text{ or } \neg L \text{ occurs in } \mathcal{C}, \\ L \text{ is undefined in } M \end{array}$$

As already mentioned, we allow the clause which causes a propagation to be constructed lazily. Following [NOT06], we assume that $(D \vee L)$ is usually only constructed when the `T-Backjump` rule is applied and merely serves as a placeholder here and in the trail.

T-Backjump:

$$\frac{\langle [\![M, L, N]\!], \mathcal{C}\rangle}{\langle [\![M, (C' \vee L') \to L']\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} C \in \mathcal{C} \text{ with } [\![M, L, N]\!] \models \neg C, \\ \text{there is some clause } C' \vee L' \text{ such that:} \\ \quad \mathcal{C} \models_T C' \vee L' \text{ and } M \models \neg C', \\ \quad L' \text{ is undefined in } M, \\ \quad L' \text{ or } \neg L' \text{ occurs in } \mathcal{C} \text{ or in } [\![M, L, N]\!] \end{array}$$

T-Learn:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle M, \mathcal{C} \cup \{C\}\rangle} \quad \textbf{if} \quad \begin{array}{l} \text{each atom of } C \text{ occurs in } \mathcal{C} \text{ or in } M, \\ \mathcal{C} \models_T C \end{array}$$

T-Forget:

$$\frac{\langle M, \mathcal{C} \cup C\rangle}{\langle M, \mathcal{C}\rangle} \quad \textbf{if} \quad \mathcal{C} \models_T C$$

Restart:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![]\!], \mathcal{C}\rangle}$$

We call the proof system obtained by replacing `T-Learn` by the following `T-Learn`* rule the *CDCL*(T) proof system*.

T-Learn*:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle M, \mathcal{C} \cup C\rangle} \quad \textbf{if} \quad \mathcal{C} \models_T C$$

The `CDCL(T)` proof system (as well as `CDCL`*`(T)`) either ends up in $FailState$ – indicating that the input formula is unsatisfiable – or any other state where no further rule can be applied – indicating that the input formula is satisfiable and the trail contains a complete Boolean model (but not the theory model).

Part II

Cylindrical Algebraic Decomposition for SMT solving

Cylindrical Algebraic Decomposition

As already discussed, the *cylindrical algebraic decomposition* (CAD) is the most popular *complete* method to deal with nonlinear real arithmetic problems. For some background on its history and alternative approaches we refer to Section 1.1. We now go into some more detail on what CAD is about, how we compute it, and what we can do with the results. As always, we put special emphasis on parts relevant for embedding CAD into an SMT solver and pass over many theoretical issues here. We encourage anyone interested in the mathematical details to study the papers mentioned throughout this chapter and hope to at least mention all noteworthy issues.

The term "cylindrical algebraic decomposition" describes both a mathematical object with certain interesting properties and an algorithm to construct such objects. We thus use this term for either and usually rely on the context for which of the two we mean.

5.1 General idea

The fundamental idea of the CAD is to reduce a question that is concerned with $\mathbb{R}^n$ to an equivalent question about a finite number of "things". We observe that $\mathbb{R}$ has uncountably many elements and, therefore, any naive approach – one that fails to abstract from a single element from $\mathbb{R}$ – is doomed to fail as it is already impossible to enumerate all candidates, not to mention the issues of termination or run time complexity. By producing an equivalent problem over a finite number of representatives (for certain subsets of $\mathbb{R}$), the solutions to many questions get more approachable.

We already used the term *representative*, and it indicates a general idea of how to obtain such a finite set. We decompose $\mathbb{R}^n$ into finitely many subsets such that all elements of a particular subset are in some sense *the same* (*equivalent*) for the question we want to answer. Intuitively, we may expect these subsets to look *reasonable* in some sense, for example, they should be connected, disjoint, together cover $\mathbb{R}^n$ (thus we also call them a *partition*), and be somewhat *smooth*.

The smallest decomposition – in the number of subsets – is induced by the equivalence classes given by the above notion of equivalence. Obtaining it may, however, be costly and not worth the effort in practice, and CAD usually gives us a finer decomposition that exhibits further structural properties. We also note that an incomplete decomposition – in which elements that we consider different in general are in the same subset – may also be sufficient to solve certain problems, in particular *satisfiability*.

There are two fundamentally different ways to represent a partition of $\mathbb{R}^n$ and it heavily depends on the application which of these is preferable. As already indicated, we can choose *a representative* for every partition as a *single point*. We may also want to store the *borders of the partition* – assuming that a partition is a connected set – which tends to give us a lot more information. The CAD works with representatives internally, but we can also extract *full cell representations* consisting of the description of the borders without a lot of additional effort.

To approach the question of how to obtain such a decomposition, we start with a quick look at the equivalence relation that we use to partition $\mathbb{R}^n$ in our application. We consider how many equivalence classes exist in theory and how we can identify the border between two subsets.

Any (first-order logic) real arithmetic question of the form *Does X hold?* results in a statement (a logical sentence) that evaluates to either *true* or *false*, thereby splitting $\mathbb{R}^n$ into the part where X holds and the rest where X does not hold. When we only consider this logical level, the equivalence at its core is discrete: there is no "process" that makes the evaluation *less true* when moving from a satisfying assignment towards a conflicting assignment, but only a sudden change from satisfiability to unsatisfiability. In this sense, there are only two equivalence classes: the *true set* and the *false set*.

Definition 5.1: Equivalence classes of a formula

Let φ be a quantifier-free first-order logic real arithmetic formula. We define the *equivalence classes of* φ for *true* and *false* inductively by

$$\overset{true}{\sim}(\varphi) := \begin{cases} \mathbb{R}^n & \text{if } \varphi = true \\ \emptyset & \text{if } \varphi = false \\ \{\mathcal{A} \mid \mathcal{A}(x) \equiv true\} & \text{if } \varphi = x \in \mathcal{B} \\ \{\mathcal{A} \mid \mathcal{A}(p) \equiv true\} & \text{if } \varphi = p \in \mathscr{A} \\ \overset{false}{\sim}(\varphi') & \text{if } \varphi = \neg\varphi' \\ \overset{true}{\sim}(\varphi_1) \cup \overset{true}{\sim}(\varphi_2) & \text{if } \varphi = \varphi_1 \vee \varphi_2 \end{cases}$$

and $\overset{false}{\sim}(\varphi) := \mathbb{R}^n \setminus \overset{true}{\sim}(\varphi)$.

This somewhat changes if we peek into the statement X, or more precisely the quantifier-free part of X. We assume that we pick a certain variable assignment – we instantiate all quantifiers – and analyze how our equivalence relation is composed for each of the possible ways to construct a formula. We observe that we still have this binary decomposition for Boolean variables and predicates, but we may very well have changes for the evaluation of parts of the formula (for individual predicates or subformulae) without changing the overall evaluation of X.

We observe, maybe not particularly unexpected, that the equivalence classes of a formula are separated only when a predicate changes its evaluation result (discarding the separation due to Boolean variables). Hence, we can obtain the borders of all partitions that (possibly) separate the equivalence classes by considering all points where any of the predicates change their evaluation.

Constructing a decomposition based on where predicates change their evaluation might not yield the *smallest decomposition* as discussed before, but may very well result in

more partitions than necessary. If a predicate changing its evaluation does not change the evaluation of the whole formula, we construct separate partitions that could both be merged into the same equivalence class.

We oftentimes chose not to do this for two reasons: firstly, once we have the two partitions and checked their evaluation result, we are done working on these partitions and merging them is only additional work without any benefit. Secondly, we usually assume certain structural properties about these partitions – for example, *cylindricity* – that oftentimes get lost when merging. For an example where merging actually makes sense, and a discussion of what needs to be considered, we refer to [Neu18a].

Reviewing the type of constraints we consider here finally brings us to the notion of *sign-invariant regions*. Recall that constraints are of the form $p \,\sigma\, 0$ where p is a polynomial and σ is a *sign condition* or *relation symbol*. A constraint may thus change its evaluation only at variable assignments where the evaluation of the polynomial changes its sign, which are exactly the real roots of the polynomial. We call the partitions where no polynomial from a given set changes its *sign-invariant* regions.

Definition 5.2: Sign-invariant regions

Let $P \subset \mathbb{Q}[\overline{x}]$ be a set of polynomials. We call $R \subseteq \mathbb{R}^n$ a *sign-invariant region* of P if $\forall p \in P.\ \forall r_1, r_2 \in \mathbb{R}^n.\ \operatorname{sgn}(p(r_1)) = \operatorname{sgn}(p(r_2))$.

This gives us a way to construct a decomposition of the real space by constructing all real roots of the considered polynomials. Furthermore, it shows why a *finite* decomposition of $\mathbb{R}^n$ that represents the equivalence classes exists in the first place: every set of points that is not separated by a root of some polynomial is equivalent, every one of the finitely many polynomials only has a finite number of (connected) root surfaces, and, being polynomials, their root surfaces only cross finitely many times. As we further observe that sign-invariance with respect to *all polynomials* from a given formula immediately implies *truth-invariance* of the formula, we can use this not only to study sets of polynomials but arbitrary formulae involving such polynomials.

This reduction of a problem about *formulae* to a problem dealing with *polynomials* introduces an abstraction that may force us to consider more partitions than required: though a polynomial changes its sign, the formula might not. Reconsidering that we study formulae may be beneficial later on for what we call a *truth-table invariant CAD* [Bro98; BDE+16], possibly paving the way for future optimizations. For now, we only consider a CAD arguing about polynomials which is sufficient and defer this point, for example to Section 6.6.

The CAD method provides an algorithmic framework for how to construct such a finite decomposition effectively. It proceeds *dimension-wise* and constructs a CAD in some dimension in a way such that it can be *extended* to a higher-dimensional CAD. This extension is done in a very direct way: given the representatives of a k-dimensional CAD, we obtain the representatives of the $(k+1)$-dimensional CAD by extending every representative with one or more values from the $(k+1)$st dimension.

In other words, the representatives of the k-dimensional CAD are exactly the representatives of the $(k+1)$-dimensional CAD *projected onto* $\mathbb{R}^k$. Similarly, the *full-cell representations* of a k-dimensional cell can directly be extended to the representations of the corresponding $(k+1)$-dimensional cells.

Of course, we want to construct the $(k+1)$-dimensional CAD, irrespective of *which* sample points were used for the k-dimensional CAD. As we argued that all sample points from one region are equivalent, it should not make a difference which one was selected. This immediately yields that the projections of two $(k+1)$-dimensional CAD cells onto $\mathbb{R}^k$ must be *either identical or disjoint*. The intuitive argument is as follows: if the projections of two cells C_1, C_2 overlap but are not identical, it would be possible to select a representative $s_k \in C_1 \setminus C_2$ – recall that all samples within a cell are supposed to be equivalent. However, this representative can not be extended to a sample point $s_{k+1} \in C_2$ and, thus, we may not obtain a sample point from C_2 at all.

5.1.1 Cylindricity and delineability

The above observation about how two CAD cells relate results in two crucial concepts that describe how a CAD is built: *cylindricity* and *delineability*. While *cylindricity* describes how the cells look like and how they are arranged relative to each other, *delineability* gives a criterion to identify such cells. We now give very brief definitions and some intuitive descriptions of these concepts here and refer to any of [Col75; ACM84; McC88] for more detailed definitions and more extensive discussions. As we have already argued, the projections of two $(k+1)$-dimensional CAD cells onto $\mathbb{R}^k$ must be *either identical or disjoint*, and we formalize this statement in Definition 5.3.

Definition 5.3: Cylinders and cylindrically arranged cells

Let C be a set of k-dimensional cells. We say that the cells C are *arranged in cylinders* (or simply *are cylindrical*) if

$$\forall c_1, c_2 \in C.\ \left(\mathrm{proj}_{k-1}(c_1) = \mathrm{proj}_{k-1}(c_2)\right) \vee \left(\mathrm{proj}_{k-1}(c_1) \cap \mathrm{proj}_{k-1}(c_2) = \emptyset\right)$$

where proj_{k-1} denotes projection onto $(k-1)$-dimensional space. For any $c \in C$, we call $c \times \mathbb{R}$ a *(k-dimensional) cylinder over c*.

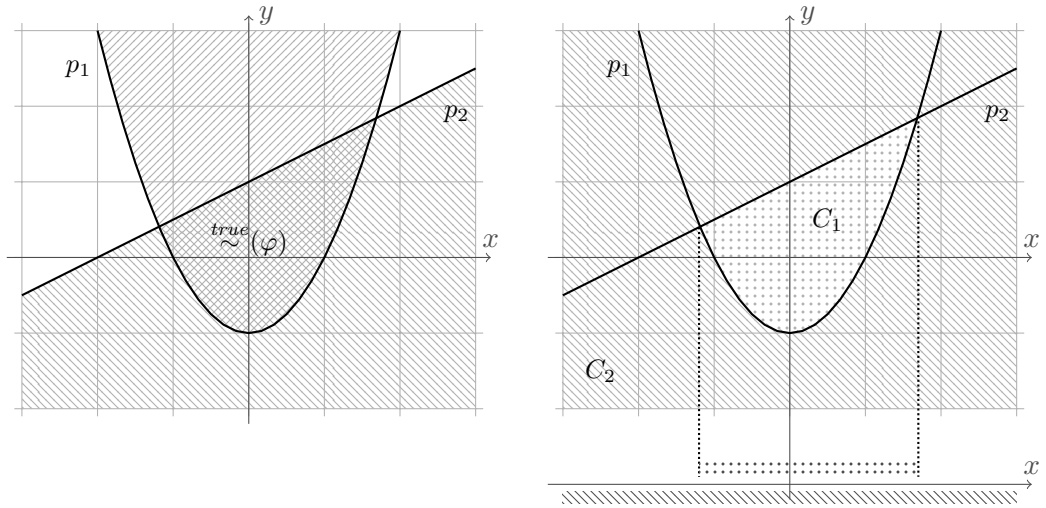
For cells that are *arranged in cylinders*, sample points for all $(k+1)$-dimensional cells from a particular cylinder can be obtained by extending *any* sample point from the k-dimensional cell underlying this cylinder, as the projection of every cell from this cylinder onto $\mathbb{R}^k$ *is identical*.

To obtain a set of cells that is cylindrical, we need some way to identify cells C such that their sign-invariant regions over C are arranged cylindrically – in particular C must be so that the projection onto R^k of every sign-invariant cell that intersects $C \times \mathbb{R}$ is C, and not only some subset of C. We can reformulate this in terms of the root surfaces of the involved polynomials as what we call *delineability*: we can use a cell C if the *number and order* of roots of all polynomials are invariant over C.

Definition 5.4: Delineability

Let $P \subsetneq \mathbb{R}[x_1, \ldots, x_k + 1]$ be a set of polynomial over $k+1$ variables and C a k-dimensional cell. We call P *delineable over* C if the number of real roots of polynomials from P and their multiplicities are constant on C.

Intuitively, a set of polynomials is delineable over a cell if their root surfaces are properly "stacked" above this cell in the sense that every two root surfaces are either identical (over this cell) or do not touch each other (over this cell).



(a) Solutions of φ (b) Projection of cells onto x-axis

Figure 5.1: Decompositions based on two constraints

Apparently, we need some way to construct such a lower-dimensional CAD, in particular, we need to obtain an appropriate set of lower-dimensional polynomials that induce the lower-dimensional CAD. The characterization of *delineability* paves the way to a constructive method by identifying polynomials that have roots where the *number or order* of roots of any polynomials change. We discuss how we identify and characterize these places (or rather borders) in the following Section 5.2.

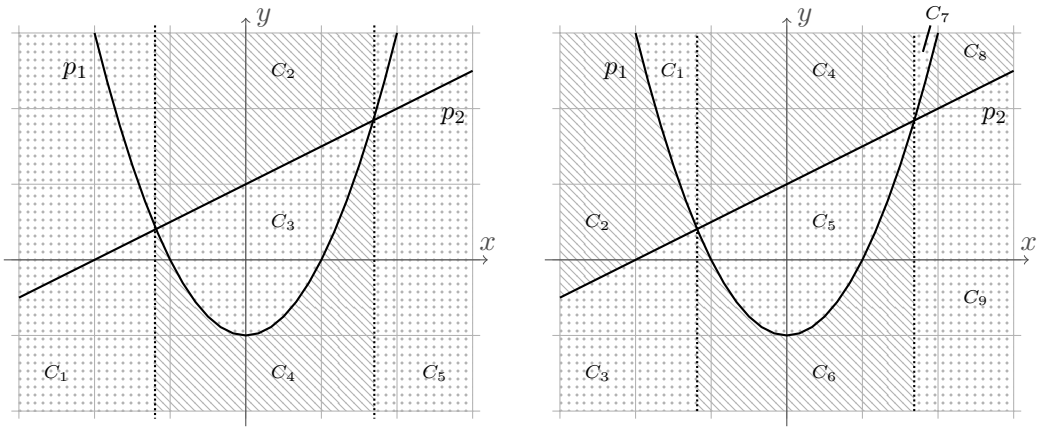
Inspired by the relation between the CAD cells – defined via the projection of higher-dimensional cells – we call this process *projection*, while the stepwise construction of sample points is called *lifting*. In literature, the terms *elimination* and *construction* are sometimes used as well. Before we continue with more detailed descriptions of these individual components, let us consider a full example for a CAD that illustrates the aforementioned concepts.

5.1.2 CAD by example

We now show the construction of a full CAD by the example of the two constraints $y - x^2 + 1 > 0$ and $2y - x - 2 < 0$, combined into the input formula $\varphi \coloneqq (y - x^2 + 1 > 0) \wedge (2y - x - 2 < 0)$. In the following figures, we show the root surfaces of the corresponding polynomials $p_1 \coloneqq y - x^2 + 1$ and $p_2 \coloneqq 2y - x - 2$.

The satisfying regions for each constraint are depicted in Figure 5.1a as striped areas and we observe that the whole formula is satisfied by the region in between the two curves. Note that we only consider what are commonly called *full-dimensional* (or *open*) cells in this example.

We discussed that every CAD cell is *sign-invariant* with respect to the polynomials from φ, or at least *truth-table invariant* with respect to φ. A first attempt might be to construct two cells: one for the satisfying region and one for the rest as depicted in Figure 5.1b. Considering the projections of these cells C_1 and C_2 onto the x-axis, however, we observe that they are not arranged cylindrically as their projections onto the x axis intersect but are not identical.



(a) Truth-table invariant cells (b) Sign-invariant cells

Figure 5.2: CAD cells based on two constraints

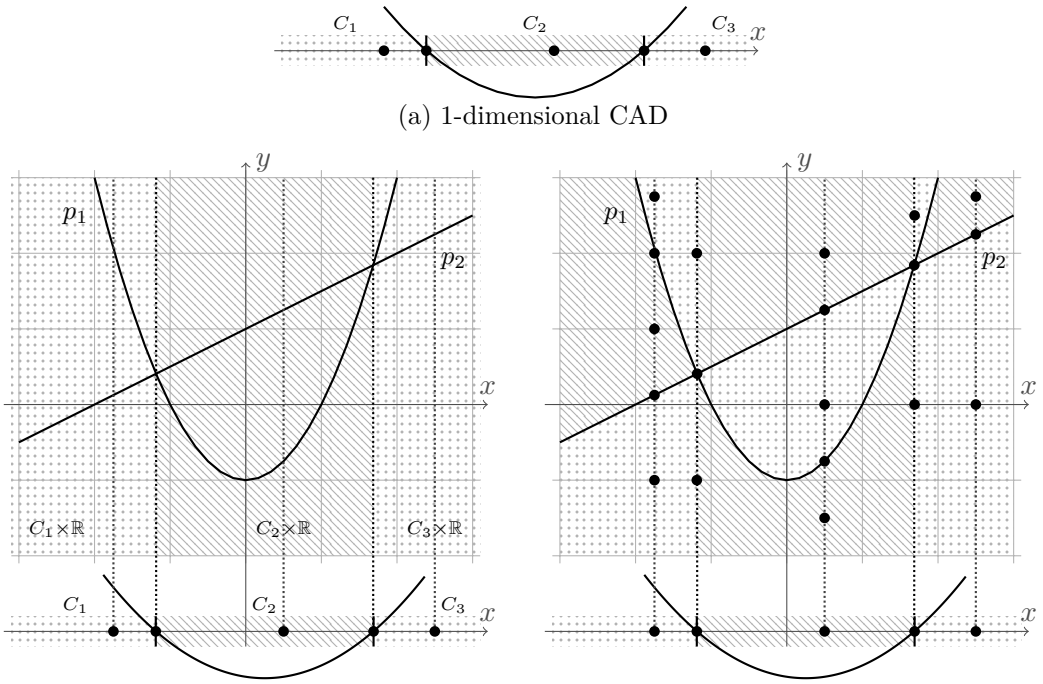
To obtain a cylindrical arrangement of cells, we need to *split* C_2 at the boundaries of C_1 into multiple cells as shown in Figure 5.2a. This already yields five cells that are now cylindrically arranged and *truth-table invariant*. Note how both C_1 and C_5 are not *sign-invariant* as both c_1 and c_2 have root surfaces within these cells. A sign-invariant CAD thus has both C_1 and C_5 separated into multiple cells by the root surfaces of c_1 and c_2 as shown in Figure 5.2b. Also note that we have ignored the fact that sign-invariance also recognizes the root surfaces as cells themselves and instead assumed the notion of open CAD (as described in Section 5.2.9.1).

It is usually difficult to obtain a truth-table invariant CAD directly, at least if no equational constraints as described in Section 5.2.8 are present and allow for techniques like the ones from [BDE+16]. We would need to compute a sign-invariant CAD first and then merge adjacent cells, if appropriate. We thus assume to work with sign-invariant CADs for most of this work as the effort of merging cells is not necessary to decide upon the satisfiability of a formula. A notable exception is quantifier elimination as described in Section 6.9, where merging cells allows for smaller resulting formulae.

The algorithmic construction of this CAD proceeds in the aforementioned two stages: projection and lifting. While the projection constructs lower-dimensional polynomials from the input polynomials, the lifting uses these dimension-wise to build sample points that represent the regions depicted in Figure 5.2b. As we give more detailed descriptions of both the projection and the lifting in the two subsequent sections, we only give a very brief overview here.

From φ we extract the polynomials $P = \{y - x^2 + 1, 2y - x - 2\}$. The projection additionally yields $Proj(P) = \{2x^2 - x - 4\}$ which has real roots exactly at the boundaries of the projections of the cells that we identified in Figure 5.1b: ≈ -1.19 and ≈ 1.69. The lifting procedure now constructs a 1-dimensional CAD, represented by 1-dimensional sample points, as shown in Figure 5.3a.

The first step is to construct 2-dimensional cylinders over the 1-dimensional CAD cells, as depicted in Figure 5.3b. Note how every sample that represents a cell of the 1-dimensional CAD induces a line in every 2-dimensional cylinder: all representatives of 2-dimensional cells will be constructed on the corresponding line. To subdivide the cylinders into 2-dimensional cells, we consider the root surfaces of every 2-dimensional



(a) 1-dimensional CAD

(b) 2-dimensional cylinders

(c) 2-dimensional CAD

Figure 5.3: Building a 2-dimensional CAD

polynomial. As we ensure delineability, we know that for every representative of the 1-dimensional CAD – and thereby every line in the 2-dimensional cylinder – the real roots of the polynomials over this representative – or its intersections with the line – are *equivalent* in the sense that their number and order remains the same.

The construction of these representatives is shown in Figure 5.3c. We start by computing the real roots of every 2-dimensional polynomial over every 1-dimensional representative – recall Section 2.5.3 for how to do this – and additionally select sample points below the smallest root, between every two consecutive roots, and above the largest root. Altogether, we obtain 21 sample points (of dimension 2) for this example, including two sample points for the rightmost cylinder that lie outside of Figure 5.3c.

In the following, we present different existing methods for the *projection operator* we denoted by $Proj$ in Section 5.2 and provide some more details on the lifting procedure in the subsequent Section 5.3.

5.2 Projection operators

We discussed that – given a set of "input" polynomials – we want to construct a set of lower-dimensional polynomials that induce a lower-dimensional CAD, and then use the sample points of this lower-dimensional CAD to construct sample points for our current problem. The main requirement for the lower-dimensional polynomials is that they should have roots wherever the *number or order* of the input polynomials *real roots* change, that is where they "stop being delineable". To construct these lower-dimensional polynomials, we now make use of *reducta*, *resultants* and *discriminants* as already defined in Section 2.2.

Since CAD was proposed in [Col75], several methods to compute such lower-dimensional polynomials have been developed and we call them *projection operators*. We focus on the projection operators suitable for arbitrary inputs and briefly mention a few more that are intended for more specialized cases at the end of this section.

The first projection operator is due to Collins in [Col74], but Collins himself already notes in [Col75, Section 5] that many polynomials are unnecessary in most cases. Hong showed in [Hon90] how to reduce Collins' projection operator significantly to what we call *Hong's projection operator*.

McCallum used a rather different mathematical point of view in [McC88] to motivate another projection operator. Though it uses the same ingredients, it is again substantially smaller than the ones due to Collins or Hong. *McCallum's projection operator* however has an important caveat: for problems of dimension larger than three, the CAD construction may fail or be incomplete in the sense that some cells may be missing from the result – or rather cells are incorrectly "merged" and in this sense a representative is missing. We refer to this issue as *completeness* (or *incompleteness*) of a projection operator. Brown improved upon McCallum's operator again in [Bro01] with respect to the number of constructed polynomials, but still suffers from the deficiency of McCallum's operator and even adds additional sources of incompleteness.

Lazard published another improvement on McCallum's projection operator in [Laz94]. In contrast to McCallum's and Brown's operators, Lazard ensured correctness for all degrees by a slight modification in the lifting process. However, it never came into widespread use as a gap in his proof was noticed in [Col98] and [Bro01]. Lazard's projection operator only came to the fore when McCallum and Hong closed this gap in [MH16] – a full proof was just proposed in [MPP19] – and thereby provide us with a *complete* projection operator that essentially supersedes all but Brown's operator.

5.2.1 Intuition

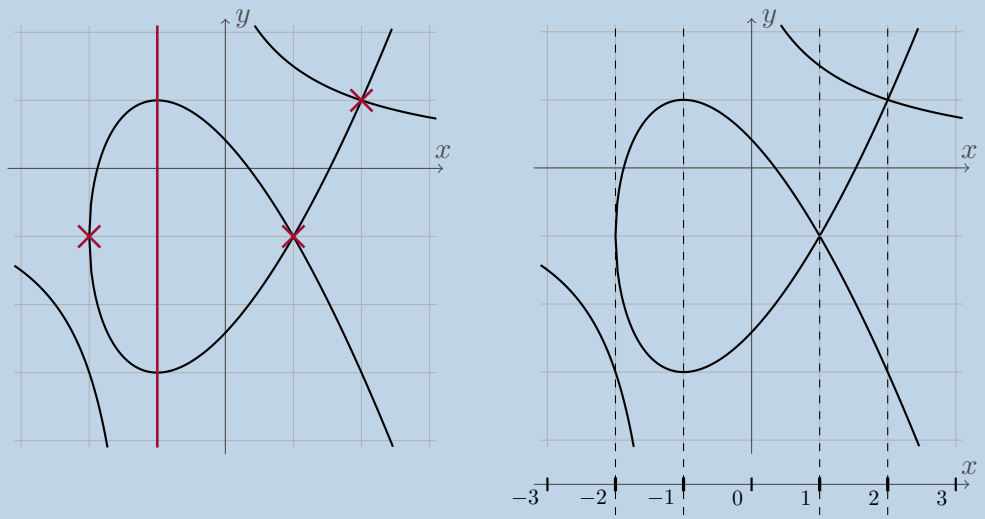
Though we do not dive into the strict mathematical reasoning behind the different projection operators, we give some intuition on what a projection operator does and what it means geometrically. For the purpose of this section, we consider *McCallum's projection operator* – ignoring preconditions and the question of completeness. This projection operator is (roughly) defined as follows.

$$Proj(P) = \{\text{coeffs}(p), \text{disc}(p) \mid p \in P\} \cup \{\text{res}(p, q) \mid p, q \in P\}$$

We observe that we essentially have three components: coefficients of a polynomial, the discriminant of a polynomial, and the resultant of two polynomials, and give some intuition on the role of these components in Example 5.1.

Example 5.1: CAD projection

We consider an input formula with the two polynomials $p = (y+1)^2 - x^3 + 3x - 2$ and $q = (x+1) \cdot y - 3$. We observe that the roots of the first polynomial form a tie-like shape, while the second polynomial is responsible for the two hyperbolas.



In the x-dimension (that is shown at the bottom of the right figure) we can identify four points where the *number or order of roots* change, and thus separate sign-invariant regions, which fall into the following four categories: 1. $x = 2$ because p and q intersect at $(2, 1)$, 2. $x = 1$ because p intersects with itself at $(1, -1)$, 3. $x = -2$ because p *turns around* at $(-2, -1)$, and 4. $x = -1$ because q has an asymptote at $x = -1$. Let us consider the individual components of the projection set $Proj(\{p, q\})$, cleaned from constant and multiple factors as described at the end of Section 2.2:

$$\begin{aligned} Proj(P) = \{ \quad & \text{coeffs}(p) = \{x^3 - 3x + 1\}, \\ & \text{coeffs}(q) = \{x + 1\}, \\ & \text{disc}(p) = (x-1)^2 \cdot (x+2), \\ & \text{disc}(q) = 1, \\ & \text{res}(p, q) = (x-2) \cdot (x^4 + 4x^3 + 6x^2 + 7x + 7) \quad \} \end{aligned}$$

We observe that the resultant covers all intersections of p and q – note that the second factor of the resultant has no real roots. The discriminant, on the other hand, covers the cases where a polynomial induces a separation by itself, either by self-intersection or by turning around. Roots in the coefficients finally indicate singularities, in this case at -1.
The sets of cylinder boundaries may very well overlap if some of these components happen to share common polynomial factors, for example, because multiple of the above criteria coincide. Note, however, that all projection operators may also be subject to *imprecision* in the sense that polynomial factors are generated that indeed yield new cell boundaries that are *spurious*.

5.2.2 Collins' projection operator

In the first publications on the cylindrical algebraic decomposition, multiple (improving) projection operators were proposed. Though the authorship is not completely clear, we attribute this first group of projection operators to Collins. We call all of them *Collins' projection operator* and mean the last of them in Definition 5.7 if not further specified.

We start with the formulation of Collins' projection operator from the first paper [Col74]. Having already seen the general structure of more recent projection operators in the last section, it is the only one that does not follow the pattern we have outlined. Instead, it really looks like the first working solution to finding a projection operator at all, essentially including everything that might be useful.

Definition 5.5: First version of Collins' projection operator

Let $P \subset \mathbb{Z}[\overline{x}]$ and $Q = P \cup \{p \cdot q \mid p, q \in P, p \neq q\}$. *Collins' first projection operator* as of [Col74] is defined as follows:

$$Proj_{Collins}(P) = \bigcup_{q \in RED(Q)} PSC(q, q')$$

In the very next year, [Col75] already contains a significantly improved version that starts to show the features we have highlighted above: the separation into coefficients, something for every polynomial individually and as well for every pair of polynomials. In particular, note that we get rid of the multiplication of polynomials that usually leads to an expensive degree growth – though of course another degree growth is hidden within the principal subresultant coefficients.

Definition 5.6: Second version of Collins' projection operator

Let $P \subset \mathbb{Z}[\overline{x}]$. *Collins' second projection operator* is defined as follows:

$$\begin{aligned} Proj_{Collins}(P) = &\{\text{lcoeff}(p) \mid p \in RED(P)\} \\ &\cup \bigcup_{p \in RED(P)} PSC(p, p') \\ &\cup \bigcup_{p,q \in RED(P)} PSC(p, q) \end{aligned}$$

Later on, yet another variant was presented in [ACM84] that again features an improvement, though much more subtle than the previous one. Observe that the combination of $\text{lcoeff}(p)$ and $PSC(p, p')$ is really just a rewriting, but the modification in the last part actually removes $PSC(p^*, q^*)$ where p^* and q^* are from the reducta set of the same polynomial.

Definition 5.7: Third version of Collins' projection operator

Let $P \subset \mathbb{Z}[\overline{x}]$. *Collins' projection operator* is defined as follows:

$$\begin{aligned} Proj_{Collins}(P) = &\bigcup_{p \in RED(P)} (\{\text{lcoeff}(p)\} \cup PSC(p, p')) \\ &\cup \bigcup_{\substack{p,q \in P \\ p<q}} \bigcup_{\substack{p^* \in RED(p) \\ q^* \in RED(q)}} PSC(p^*, q^*) \end{aligned}$$

This last version from Definition 5.7 is what is usually considered to be *the* projection operator due to Collins and we use it as such for the following experiments.

5.2.3 Hong's projection operator

A few years later, Hong managed to improve Collins' projection operator in [Hon90] in a seemingly small but (in practice) significant way. Note that we can use the essentially same proofs for this modified construction, also retaining full theoretical completeness and the same mathematical foundations. Hong showed that for the pairwise computations of the pseudo resultant coefficients, we only need to consider the whole reducta set for one of the polynomials.

An interesting remark appears in [SS03], noting that "the construction of the chain of reducta can be stopped as soon as the first constant leading coefficient appears". This observation immediately transfers to leading coefficients that do *not vanish* anywhere (like $x^2 + 1$) and also holds for Collins' projection operator. Furthermore, the same argument also applies to McCallum's projection operator that is shown below.

Definition 5.8: Hong's projection operator

Let $P \subset \mathbb{Z}[\overline{x}]$. *Hong's projection operator* is defined as follows:

$$\begin{aligned} Proj_{Hong}(P) = & \bigcup_{p \in RED(P)} (\{\text{lcoeff}(p)\} \cup PSC(p, p')) \\ & \cup \bigcup_{\substack{p,q \in P \\ p<q}} \bigcup_{p^* \in RED(p)} PSC(p^*, q) \end{aligned}$$

5.2.4 McCallum's projection operator

Around the same time, McCallum devised another improvement in [McC84] and – only for dimension three – in [McC88]. This time the projection is approached from a different mathematical angle, namely from complex analytic geometry, based on [Zar65].

We previously considered the whole set of principal subresultant coefficients for every polynomial (and its derivative) from the reducta set and for every pair of polynomials (where one is from the reducta set). McCallum showed that we can replace all these by a single discriminant (for every polynomial) and a single resultant (for every pair of polynomials). We observe that this construction is not fundamentally different though, as we know that $psc_0(p, q) = \text{res}(p, q)$, but "only" allows us to consider the polynomial (instead of its reducta set) and the resultant (instead of the set of principal subresultant coefficients), yielding way fewer polynomials in most cases.

Definition 5.9: McCallum's projection operator

Let $P \subset \mathbb{Z}[\overline{x}]$ be a set of polynomials, $\text{cont}(P)$ their content and P' the finest square-free basis of their primitive parts $\text{prim}(P)$. *McCallum's projection operator* is defined as follows:

$$\begin{aligned} Proj_{McCallum}(P) = & \{\text{coeffs}(p), \text{disc}(p) \mid p \in P'\} \\ & \cup \{\text{res}(p, q) \mid p, q \in P'\} \cup cont(P) \end{aligned}$$

This greatly simplified projection operator has a significant drawback compared to those of Hong or Collins. In certain cases, the projection is *incomplete* in the sense that we may not be able to do the lifting process properly. We discuss the reasons and

consequences in Section 5.3.1. Furthermore, this projection operator assumes that P is not some arbitrary set of polynomials, but a *finest square-free basis*. We discuss this issue briefly in Section 6.5.3 and note that it can be achieved by completely factorizing all polynomials, which appears reasonable anyway. This even yields an *irreducible basis*, which is in some sense even stronger than a square-free basis.

Similar to the remark about Hong's projection operator from [SS03], it is sufficient to consider the coefficients in McCallum's projection operator only until a constant (or rather not vanishing anywhere) coefficient appears, starting from the leading coefficient. We call this variant *partial McCallum's projection operator*.

5.2.5 Brown's projection operator

Brown improved on McCallum's projection operator in [Bro01] and showed that it is oftentimes enough to consider only the leading coefficient instead of all coefficients. Naturally, it inherits the downsides of McCallum's projection operator concerning its incompleteness. Removing all other coefficients from the projection may even lead to additional problems in the lifting phase, though a method is given to detect and handle these cases in [Bro01].

Definition 5.10: Brown's projection operator

Let $P \subset \mathbb{Z}[\overline{x}]$ be a set of polynomials, $\text{cont}(P)$ their content, and P' the finest square-free basis of their primitive parts $\text{prim}(P)$. *Brown's projection operator* is defined as follows:

$$\begin{aligned} Proj_{Brown}(P) =& \{\text{lcoeff}(p), \text{disc}(p) \mid p \in P'\} \\ & \cup \{\text{res}(p, q) \mid p, q \in P'\} \cup \text{cont}(P) \end{aligned}$$

5.2.6 Lazard's projection operator

Lastly, *Lazard's projection operator* was published in [Laz94] but it was mostly disregarded as its correctness proof contained a flaw that was only corrected more than twenty years later in [MH16; MPP19]. The projection operator itself is very similar to McCallum's and Brown's projection operators – it uses the leading and the trailing coefficient. One could very well argue that Lazard's main contribution is a slight change in the lifting process that resolves the incompleteness issue.

Definition 5.11: Lazard's projection operator

Let $P \subset \mathbb{Z}[\overline{x}]$ be a set of polynomials, $\text{cont}(P)$ their content, and P' the finest square-free basis of their primitive parts $\text{prim}(P)$. *Lazard's projection operator* is defined as follows:

$$\begin{aligned} Proj_{Lazard}(P) =& \{\text{lcoeff}(p), \text{tcoeff}(p), \text{disc}(p) \mid p \in P'\} \\ & \cup \{\text{res}(p, q) \mid p, q \in P'\} \cup \text{cont}(P) \end{aligned}$$

Similar to the projection operators due to McCallum and Brown, a problem – in the correctness proof – arises if we substitute a partial sample point into a polynomial to compute new sample points and this polynomial vanishes identically on this partial

sample point. The solution to this, at least for Lazard's projection operator, is rather simple at first glance. We substitute one variable at a time (say $x \mapsto \alpha_x$) and if the polynomial vanishes identically we know that $(x - \alpha_x)$ divides the polynomial q. In this case we simply replace q by $q/(x - \alpha_x)$ and continue from there. We discuss the details of this approach in Section 5.3.2.

5.2.7 Relation between projection operators

We have presented five different, though roughly similar, projection operators which naturally yields the questions of how they relate to each other. We give some brief results on their qualitative relation of the resulting sets of polynomials and the (theoretical) applicability in a complete way, as well as a quantitative comparison of the resulting projection sets. We acknowledge that there are further questions of possible interest, ranging from a comparative analysis of their mathematical background and motivation over a more detailed quantitative comparison to the actual impact on the overall performance for a practical solver that we do not consider here.

We observe that $Proj_{Hong}(P) \subseteq Proj_{Collins}(P)$ as the only difference is that Hong's projection operator only includes $psc_k(p, q)$ for $q \in P$ instead of $q \in RED(P)$, and we know that $P \subseteq RED(P)$. The case is even easier for the projection operators based on McCallum. $Proj_{Brown}(P)$, $Proj_{Lazard}(P)$, and $Proj_{McCallum}(P)$ only differ in which coefficients are included: we have the leading coefficient for $Proj_{Brown}$, leading and trailing coefficient for $Proj_{Lazard}$, and all coefficients for $Proj_{McCallum}$.

To relate $Proj_{Hong}(P)$ and $Proj_{McCallum}(P)$, we note that the first principal subresultant coefficient *is* the resultant – this is why we can use the principal subresultant coefficients to compute the resultant in the first place – and thus we always have $\text{res}(p, q) \in PSC(p, q)$. Combined with how we defined the reducta set, this almost immediately yields that $Proj_{McCallum}(P)$ is included in $Proj_{Hong}(P)$. Thus, we get

$$\begin{aligned} Proj_{Brown}(P) &\subseteq Proj_{Lazard}(P) \\ &\subseteq Proj_{McCallum}(P) \\ &\subseteq Proj_{Hong}(P) \\ &\subseteq Proj_{Collins}(P) \end{aligned}$$

Recall, however, that $Proj_{Brown}$ and $Proj_{McCallum}$ have the important unpleasant property that they may be incomplete on certain examples, even if these are rather pathological. In [Laz94], alongside $Proj_{Lazard}$, a (slightly) modified lifting procedure was introduced that resolves any incompleteness issues with $Proj_{Lazard}$ – and thereby also $Proj_{McCallum}$, at least if the trailing coefficient is retained and not removed as proposed in [SS03] – which we discuss in Section 5.3.2. $Proj_{Brown}$ still suffers from other sources of incompleteness, though.

This mix and match approach to combining projection operators, special techniques (like equational constraints as shown in Section 5.2.8), and changes to the lifting process allows to quickly come up with interesting variants of CAD. It is, however, important to keep in mind that the theoretical foundation for the correctness hangs by a thread: while it is oftentimes retained by rather simple arguments, it can be invalidated by seemingly innocent changes. For example, $Proj_{McCallum}$ can be used

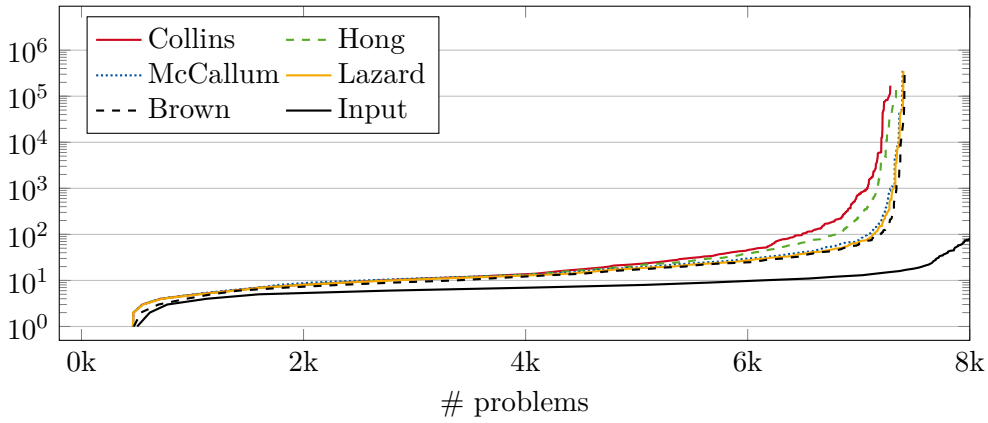


Figure 5.4: Overall projection size

to form a complete CAD when we employ Lazard's lifting scheme. However, while removing coefficients as proposed in [SS03] for $Proj_{McCallum}$ is "safe" (it stays "as incomplete" as it is), it destroys the theoretical foundation of $Proj_{Lazard}$ and thereby the reasoning why Lazard's lifting scheme makes $Proj_{McCallum}$ complete as well.

The above discussion points to three possible directions for future research on projection operators: 1. finding new projection operators that are sound without an adapted lifting, most probably between Hong's and McCallum's projection operators, 2. exploiting special cases or specific structures of constraints and formulae like equational constraints that we discuss in the following or 3. employ new basic building blocks that replace (sub-)resultants. Current work mostly focuses on the second one, as research on CAD, in general, seems to focus more on integrating CAD into specific applications recently. After all, the whole work of this thesis arguably falls into this category.

While we performed some more extensive analysis about the shape and size of projections for different projection operators in [Vie16; VKÁ17], we give a brief overview here to get a rough feeling for the magnitude of differences in size. In Figure 5.4, we show the results of computing a *full projection* of *all polynomials* within a formula for all nonlinear real formulae from SMT-LIB and extracting the overall number of polynomials (if possible within at most 30 min and 8 GB). While this estimate is rather pessimistic – usually the Boolean structure does not require having all polynomials in a CAD at once – it gives a pretty good handle on the approximate size.

Following the previous discussion, the results in Figure 5.4 show that the overall sizes of the projection sets follow a strict ordering. Also, the distances between the individual curves roughly match our expectation: both moving from Collins' to Hong's and from Hong's to McCallum's projection operator makes a significant difference while McCallum's, Lazard's, and Brown's projection operators are pretty close to each other.

As we discuss at various places, for example, in [VKÁ17; KÁ20], having a smaller projection in terms of the number of polynomials does not automatically make the overall solving process faster. For example, it may be beneficial to have additional polynomials with smaller degrees that are preferred for the lifting. Additionally, the actual run time depends on several other factors that have nothing at all to do with the projection. Some experimental results on the impact of different projection operators in practice are given in Section 6.4.

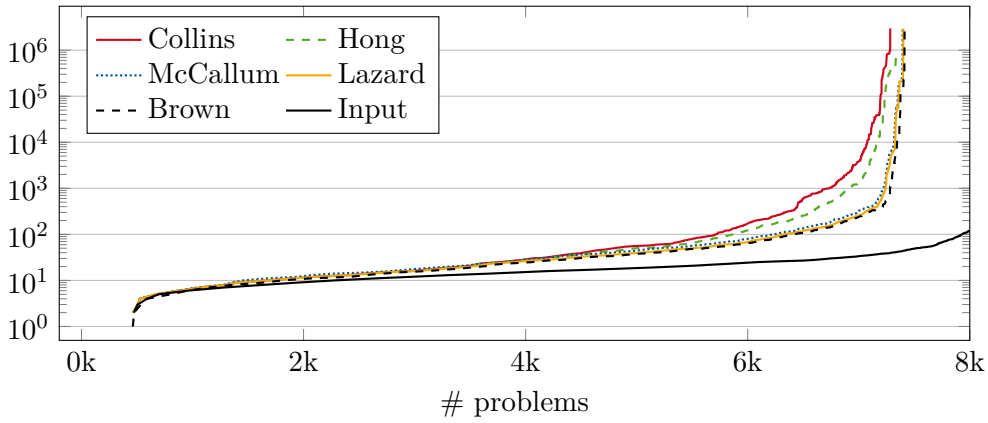


Figure 5.5: Polynomial degrees of projections

We show another indicator in Figure 5.5, namely the sum of the degrees of all polynomials. For comparison with the input problems, the dotted line shows the sum of the degrees of all input polynomials. As we can see, the number of polynomials, as well as their degrees, starts growing quickly at some point although the degree of the input stays comparably small. Hence the steep growth is not due to large input problems, but rather indicates an "algebraic hardness" of some kind.

Altogether, these observations roughly match conventional wisdom about CAD, combined with our analysis of the benchmark set: many problems can be solved reasonably fast, but it is not uncommon to see (near) worst-case behavior in practice. Also keep in mind, that this only considers two contributing factors – the number of polynomials and their degree – while we have not looked at coefficient growth and how these effects multiply once we use these polynomials to construct algebraic numbers in the lifting.

5.2.8 Equational constraints

We observer that the aforementioned projection operators solely work on polynomials and completely ignore the sign conditions attached to the input polynomials. They provide everything needed to produce a *sign-invariant CAD* for the input polynomials, though we usually construct a CAD with respect to input *constraints*, or even a formula containing constraints. In these cases it is sufficient to construct a CAD that is *truth-invariant* (or *truth-table-invariant*) with respect to the input constraints.

In general, we can avoid lifting partial sample points that falsify the input formula as described in [Hon90], but this, unfortunately, does nothing to simplify the projection. Collins found a way to do exactly that in [Col98] by exploiting *equational constraints* – equations that are implied by the input formula. Essentially, we not only avoid lifting sample points that falsify an equation but also skip certain projection steps whose results can not contribute to a satisfying sample for this equation.

While Collins's proposal in [Col98] contains the important ideas but is somewhat vague on the details, McCallum subsequently provided correctness proofs and more details in [McC99] and [McC01]. Let p be the polynomial of an equational constraint, then the fundamental observation is that we only need to work on the cells that satisfy $p = 0$ and can safely ignore cells where $p \neq 0$ – and in particular higher-dimensional cells and cell boundaries within these.

Considering the role of the individual components of the projection operators as described in Example 5.1, Collins essentially observed the following: the projection of p alone is sufficient to describe the cells where $p = 0$ and considering the resultants of p and the remaining polynomials also allows us to take care of all intersections of these other polynomials with p. This immediately motivates the *restricted projection* as defined in [Col98] and analyzed more closely in [McC99] and [McC01].

Definition 5.12: Restricted projection operator

Let $P \subset \mathbb{Z}[\overline{x}]$ be a set of polynomials, $\text{cont}(P)$ their content, and P' the finest square-free basis of their primitive parts $\text{prim}(P)$. Furthermore, let $E \in P'$ be an equational constraint polynomial. The *restricted projection operator* is defined as follows:

$$Proj_{restricted}(P) = \text{cont}(P) \cup \text{coeffs}(E) \cup \{\text{disc}(E)\} \cup \bigcup_{p \in P'} \text{res}(E, p)$$

Note that the *restricted projection operator* is based on McCallum's projection operator, but can be defined analogously for all projection operators defined above (with the exception of Collins' first projection operator). The rigorous proof in [McC99] works for up to three dimensions only, but [McC01] generalizes to more dimensions with a small caveat. For any dimension, we can safely use the *restricted projection operator* for the first and the last projection step. For all projection steps between, however, we need to use the *semi-restricted projection operator* instead. Furthermore, we can only use the (semi-) restricted projection operator if it has already been used for all preceding dimensions. *Interruptions* – using a regular projection operator at some point and continuing using the restricted one again in a lower dimension – are technically possible, but not shown to be sound.

Definition 5.13: Semi-restricted projection operator

Let $P \subset \mathbb{Z}[\overline{x}]$ be a set of polynomials, $\text{cont}(P)$ their content, and P' the finest square-free basis of their primitive parts $\text{prim}(P)$. Furthermore let $E \in P'$ be an equational constraint polynomial. The *semi-restricted projection operator* is defined as follows:

$$Proj_{semi}(P) = \text{cont}(P) \cup \text{coeffs}(E) \cup \bigcup_{p \in P'} (\text{disc}(p) \cup \text{res}(E, p))$$

The possibility to use the (semi-) restricted projection operator, of course, depends on the *existence* of an equational constraint in the *correct dimension*. While there is nothing we can do if no equational constraint is present, Collins already presented a way to exploit two equational constraints in the *same dimension* by using them to infer an equational constraint in a lower dimension – that is not part of the input.

Let $E_1 = 0$ and $E_2 = 0$ be two equational constraints in the same dimension. We observe that every satisfying sample point must thus satisfy $E_1 = 0$ and $E_2 = 0$, and, furthermore, that the resultant of two polynomials describes the *common roots* of these polynomials. Hence, $\text{res}(E_1, E_2) = 0$ can be used as an equational constraint, something that Collins calls the *resultant rule* in [Col98].

Definition 5.14: Resultant rule

Let $E_1 = 0, E_2 = 0$ be two equational constraints and x some variable. Then

$$(E_1 = 0 \wedge E_2 = 0) \implies \mathrm{res}_x(E_1, E_2) = 0$$

and, in particular, $\mathrm{res}_x(E_1, E_2)$ can be used as an equational constraint in the (semi-) restricted projection.

Note that using the (semi-) restricted projection operator in some dimension reduces the (asymptotic) size of the resulting set of polynomials from quadratically many (due to the pairwise resultants) to linearly many polynomials. This essentially moves the complexity of CAD due to the number of polynomials from 2^{2^n} to $2^{2^{n-1}}$, as shown in [EBD15]. The practical effect of the usage of equational constraints in the context of SMT solving has been studied in [Hae18] and [HKÁ18], though the resultant rule has not been considered there.

We observe that the *resultant rule* is not specific to using equational constraints in CAD projections, but is a general scheme to infer additional equalities from a given set of equality constraints. However, it needs some *guidance* which variable to compute the resultant for, which is why we can nicely integrate it in a CAD method that inherently works with respect to some variable ordering.

The theory about equational constraints that we have discussed here is only proven to be sound for McCallum's projection operator. Though it is generally expected that it can be transferred to the other projection operators, no formal proof has been given so far. Some first results to do exactly this for Lazard's projection operator have been presented in [NDS19] that justify a reasonable hope that the restricted projection can be used as a generic template for all projection operators. In our implementation, the restricted projection operator is exactly that: a generic template that can be used for all projection operators, possibly even with interruptions and using the restricted projection operator for all dimensions, disregarding all warranted concerns on the formal correctness of the implementation.

5.2.9 More projection operators

Besides the (more or less) well-known projection operators presented above, other projection operators have been proposed that we consider less relevant here, mostly because they are not meant for constructing a general cylindrical algebraic decomposition. All of the projection operators below are instead targeted at special applications like open CAD, CAD for parameterized systems, or the generation of single CAD cells.

5.2.9.1 Projection for open CAD

For some applications it is sufficient to only consider *open regions* – regions that have full dimension – and discard regions where a polynomial vanishes that have *volume zero* (in the sense of the Lesbesgue measure). The reasoning is usually that open regions capture *almost all* solutions, or that such regions where a polynomial vanishes represent a *tipping point* that is either unstable or at the edge of the permissible parameter space, and thus not of interest. This variant of CAD is sometimes called *open CAD* and we refer to [McC93; Str00] for a more thorough discussion.

Most importantly, open CAD allows for two significant simplifications compared to a regular CAD computation. Firstly, as we discard the sample points that correspond to polynomial roots, all our sample points are rational and thus all the difficulties concerning the handling of and working with real algebraic numbers disappear. Secondly, all the issues that cause incompleteness for some of the projection operators manifest over regions where a polynomial vanishes and we can thus greatly simplify the projection operator while retaining completeness as shown in [Str00].

Definition 5.15: Strzeboński's projection operator for open CAD

Let $P \subset \mathbb{Z}[\overline{x}]$ be a set of polynomials, $\text{cont}(P)$ their content, and P' the finest square-free basis of their primitive parts $\text{prim}(P)$. *Strzeboński's projection operator for open CAD* is defined as follows:

$$\begin{aligned} Proj_{StrzOpen}(P) = &\{\text{lcoeff}(p), \text{disc}(p) \mid p \in P\} \\ &\cup \{\text{res}(p, q) \mid p, q \in P\} \end{aligned}$$

5.2.9.2 Seidl and Sturm's projection operator

Seidl and Sturm in [SS03] make the case that for the application of general quantifier elimination we can essentially add assumptions to our formula in an ad-hoc manner. We explore their reasoning and the resulting simplifications in the following.

When doing general quantifier elimination, we assume a quantified formula φ from which we want to eliminate some variables $\overline{x}$ while we keep other variables $\overline{y}$. We usually call $\overline{y}$ *parameters*. We would apply CAD such that we project $\overline{x}$ first and extract a description of the solutions of φ in $\overline{y}$ only after lifting $\overline{y}$.

The authors argue that it is sensible to allow our projection to assume certain polynomials in $\overline{y}$ *not to vanish*, as these assumptions directly correspond to "easily interpretable" *theory assumptions* that the user had assumed anyway in most cases. These theory assumptions are generated in an ad-hoc manner to validate the following simplified projection operator based on Hong's projection operator.

Definition 5.16: Seidl and Sturm's projection operator

Let $P \subset \mathbb{Z}[\overline{x}]$. *Seidl and Sturm's projection operator* is defined as follows:

$$\begin{aligned} Proj_{S\&S}(P) = &\bigcup_{p \in GRED(P)} (\{\text{lcoeff}(p)\} \cup GPSC(p, p')) \\ &\cup \bigcup_{\substack{p,q \in P \\ p<q}} \bigcup_{p^* \in GRED(p)} GPSC(p^*, q) \end{aligned}$$

where we refer to [SS03] for the exact definitions of $GRED$ and $GPSC$.

The essential idea is to *stop* collecting the reducta for the reducta set as soon as the leading coefficient is only defined over $\overline{y}$, and, equivalently, do the same for the principal subresultant coefficients. These reducta (or principal subresultant coefficients) are then only defined over $\overline{y}$, and the authors argue that they should not vanish due to the implicit theory assumptions of the user.

Note that the underlying idea has already been discussed for Hong's projection operator, but we could only exploit it when we could show that the respective coefficient would not vanish – for example, for constants.

In the context of general quantifier elimination, this approach provides the user with two pieces of information. Firstly, a list of the automatically generated algebraic assumptions, and secondly, a formula that is equivalent to the input formula under these assumptions. The authors argue that the assumptions are "easily interpretable" and their examples suggest that they mostly characterize degenerate cases that the user did not care for anyway.

For the purpose of SMT solving, however, we aim at determining satisfiability for a specific formula in a fully automated way. Handing assumptions back to the user during the solving is not desirable here, because this might happen very frequently – for every theory call – and there may not even be a user involved. Hence, we do not consider this projection operator in this work.

5.2.9.3 Local projections

In some applications, we may only want to look at a very specific part of a CAD – usually, a single region that is identified by a given sample point. In such a scenario, it seems natural to remove polynomials from the projection that do not contribute to the "interesting" part of the CAD.

The question of generating *a generalization* or *an abstraction* of a single point to a single CAD cell was first motivated in [JM12] – a special case of the `MCSAT` approach that we present in Chapter 7 – but was adopted as an interesting question in general, for example, in [Bro13; Str14; BK15].

Multiple variants of *local projections* have been proposed with various benefits and downsides, and we only give a rough overview here. A more thorough description of the ones we use in Chapter 7 is given there. Note that most of the lifting phase is essentially skipped here because the partial assignment is already given. We only need to "lift" a single sample point in every dimension to obtain the borders of the cell we want to construct.

Jovanović and de Moura's model-based projection operator. The proposal of `NLSAT` in [JM12] – the instantiation of `MCSAT` with a CAD-based explanation function – sparked the need for a method that *explains why a partial assignment is unsatisfiable*. One possible implementation, the one proposed in [JM12], is using a CAD-based method to obtain the region that contains the partial assignment.

Their version of a local projection essentially takes Collins' projection operator and for every *component* – that is the coefficients, principal subresultant coefficients of a polynomial and its derivative, and the pairwise principal subresultant coefficients – only considers the first one that does not vanish on the given partial assignment.

Brown and Košta's single-cell method. With [Bro13], Brown departs from the traditional separation into projection and lifting, but instead understands the problem as *refining the cell* incrementally using the input constraints, where every refinement is done by adding a polynomial to the projection. In this approach, the projections may be smaller and also the region may be larger – conceivably beneficial for `NLSAT` that

can exclude a larger region from its search space. The original method from [Bro13] only considers the case of open cells, but [BK15] extends it to work for all cells – at the cost of various special cases and additional algorithmic complexity.

Strzeboński's local projection operator. Given the above ideas, Strzeboński proposed another projection operator in [Str14] that adaptively uses either McCallum's or Hong's projection operator, depending on whether a source of incompleteness was detected while using McCallum's projection operator. Additionally, it is complemented with the use of equational constraints.

5.3 Lifting

Recall that we aim to construct a CAD and decided to represent a single cell using what we called a *sample point*. This sample point is a *representative* for a cell, as the cell is *sign-invariant* on the set of input polynomials (or at least *truth-invariant* under the input formula). We already devised *projection operators* that project certain properties of our CAD into lower dimensions and now describe how we can effectively construct a set of sample points that finally represent a CAD.

The fundamental intuition is the following: given a k-dimensional CAD C_k, every cell $C \in C_k$ induces a cylinder $C \times \mathbb{R}$ which is separated in $(k+1)$-dimensional cells by the roots of the $(k+1)$-dimensional polynomials in *a nice way* – they are *delineable* as described in Section 5.1.1. Having delineable cells yields that it does not matter which k-dimensional point we select from a cell, they all yield an equivalent set of $(k+1)$-dimensional sample points. The $(k+1)$-dimensional CAD C_{k+1} consists of exactly these $(k+1)$-dimensional cells over every k-dimensional cell from C_k, represented by $(k+1)$-dimensional sample points.

Furthermore, delineability (and cylindricity of the cells) implies a particularly nice relation between k-dimensional and $(k+1)$-dimensional sample points. Given $s_k \in C \in C_k$, we can construct sample points $\overline{s_{k+1}} \in C \times \mathbb{R}$ by simply *appending* certain values for dimension $k+1$ to s_k, and these values are derived from the roots of the $(k+1)$-dimensional polynomials of the projection.

Our goal is to select one sample point from each sign-invariant region of the polynomials over some $C \in C_k$ represented by some sample point s_k. As we have already identified the roots of polynomials as *boundaries* of these sign-invariant regions, the selection is rather natural: apart from the roots themselves – representing all regions where some polynomial vanishes – we select one sample point between every two consecutive roots, as well as one value below the smallest root and above the largest root.

Definition 5.17: Lifting operator

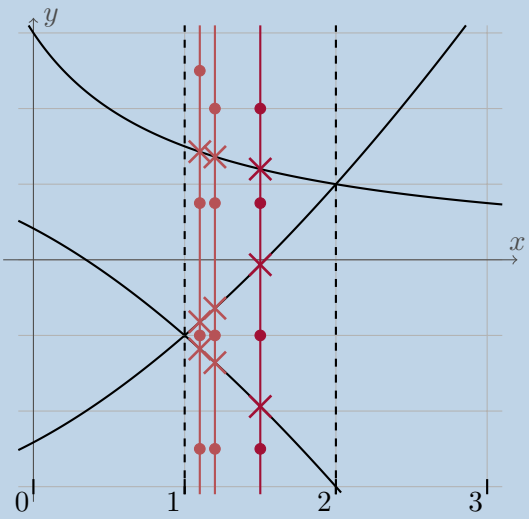
Let $P \subset \mathbb{Q}[x_1, \dots x_{k+1}]$ be a set of polynomials and s_k a k-dimensional sample point. The result of the *lifting operator* of P with s_k is the following set of $(k+1)$-dimensional sample points where $\alpha_1, \dots \alpha_m$ are the (ordered) real roots of P with respect to the model induced by s_k.

$$\{s_k\} \times \{\alpha_1, \dots \alpha_m, r_0, \dots p_m \mid r_0 < \alpha_1 < r_1 < \alpha_2 < \dots < \alpha_m < r_m\}$$

We sometimes call α_i the *root samples* and r_i the *non-root samples* or *intermediate samples* of s_k over P.

The delineability of P ensures that all s_k from a particular k-dimensional cell yield an equivalent – though not necessarily identical – result in the sense that the number or roots m remains constant, the α_i from each s_k belongs to the same surface, and the p_i for every s_k belongs to the same open sign-invariant region. We can also see this graphically in the following Example 5.2.

Example 5.2: CAD lifting



Recall the polynomials $p = (y+1)^2 - x^3 + 3x - 2$ and $q = (x+1) \cdot y - 3$ from Example 5.1. We focus on the one-dimensional cell $(1, 2)$ in the x-dimension and the cylinder above it. We chose $x = 1.5$ as indicated by a vertical line.
To lift it, we substitute $x = 1.5$ into all polynomials and obtain $p[x/1.5] = (y+1)^2 - 0.875$ and $q[x/1.5] = 2.5 \cdot y - 3$. The roots of these polynomials are -0.065, -1.94, and 1.2, respectively, indicated by the red crosses. Note that by construction, these are exactly the intersections of the vertical line and the polynomials' varieties.
Assume we instead lift other values for x, as indicated by the paler lines. Though the resulting sample points change (in general the intermediate sample points might change as well), there is a direct one-to-one correspondence to the sample points for every other value for x. Observe that the roots converge (and eventually collapse) when we move towards the cylinder boundary $x = 1$, thus changing the number of roots and posing a boundary to the region to maintain delineability.

5.3.1 Incompleteness of projection operators

We already discussed that some projection operators are what we called "incomplete" for certain inputs. While it is not that easy to construe an example that witnesses incompleteness (by provoking an incorrect result) – after all, we need at least four variables – it is reasonably easy to get a rough feeling for the fundamental problem.

The main idea is that under special circumstances we may *lose roots* during the lifting because a polynomial *vanishes identically*. If we lift $x = 0$ with $p = x \cdot y$ we obviously "lose" the root $y = 0$ because $p[x/0]$ vanishes identically. Of course, this example would not lead to incorrect results, simply because it actually *is irrelevant* how we

choose y for $x = 0$ in this example. More complex examples exist, however, where roots are lost in this way and consequently we fail to compute a full CAD and may return incorrect results. Note, however, that this case is very rare in practice: for a regular SMT strategy we have not observed any incorrect results due to this issue; the `MCSAT` approach discussed later, in particular the approach presented in Section 8.3.1, however, triggered this issue in our experiments.

It may be interesting to realize that the underlying mathematical effects and their consequences for the respective proofs of correctness differ for different projection operators: the term "well-orientedness" – and in particular the scenarios that break it – are substantially different in [McC84; McC88] and in [Bro01], and for equational constraints in [McC99; McC01] the notion is different once again. Essentially it seems that "well-orientedness" is always used for "everything is fine" for whatever this means for the projection operator at hand.

This is also reflected by how these issues can be resolved or at least mitigated. For McCallum's projection operator, we can oftentimes resolve the issue by adding so-called *delineating polynomials*, for example, based on partial derivatives of the vanishing polynomial. While the modified lifting process due to Lazard – which we discuss in the following Section 5.3.2 – resolves these cases even without the need for such *delineating polynomials*, Brown's projection operators adds additional sources of incompleteness.

5.3.2 Lazard's lifting process

As already mentioned in Section 5.2.6 and presented in [Laz94], Lazard's projection operator requires a change to the regular lifting process to maintain correctness. We have seen that for the regular lifting we take a partial assignment over $\{x_1, \dots, x_k\}$ and a polynomial q over $\{x_1, \dots, x_{k+1}\}$ and consider *isolating the real roots* of q in x_{k+1} with respect to the partial assignment as a *single, atomic* operation.

Unfortunately, q may vanish identically on the partial assignment, thus leaving no polynomial – or, more specifically, the zero polynomial – to compute the roots of. This is the underlying reason for the incompleteness of *McCallum's projection operator* and *Brown's projection operator* that we discussed in Section 5.3.1.

The proposed modification to the lifting aims to directly avoid this *nullification*: we substitute the assignment α_x for every $x \in \{x_1, \dots, x_k\}$ *individually* and check whether the polynomial already vanishes identically. If so, we know that $x - \alpha_x$ divides the polynomial and we replace q by $q/(x - \alpha_x)$, as shown in Algorithm 5.1. Ultimately, we thereby discover real roots of all factors of q, even if they would usually be "shadowed" by other factors that vanish identically over the given variable assignment. The order in which we consider the variables is crucial for the correctness of the whole procedure and needs to coincide with the variable ordering or the current CAD computation, following [MH16; MPP19].

As already discussed in Section 2.5, we can not easily compute with real algebraic numbers (unless we work in some powerful computer algebra system). Algorithm 5.1 suggests to compute with polynomials whose coefficients stem from some extension field $\mathbb{Q}(\alpha)$ to perform the polynomial division, but it is not immediately clear how to properly work with such coefficients. Just like in Section 2.5, we exploit that $\mathbb{Q}(\alpha)$ is isomorphic to $\mathbb{Q}[\xi]/\langle p_\alpha \rangle$ and perform these operations in $\mathbb{Q}[\xi]/\langle p_\alpha \rangle[x]$ instead, representing α symbolically in our polynomials using a fresh variable ξ.

Algorithm 5.1: Lifting with Lazard's projection

```
1 Function LazardLifting(α, q)
2     for x ↦ α_x ∈ A do
3         while q[x/α_x] ≡ 0 do
4             q := q/(x − α_x)
5         q := q[x/α_x]
6     return roots(q)
```

Note that we only care about whether q *vanishes identically* and it is thus sufficient to consider what we called *algebraic cancellation* in Section 2.5 in the loop of Algorithm 5.1. Hence, we only need to slightly extend Algorithm 2.2 to adapt Algorithm 5.1 as shown in the following Algorithm 5.2.

Instead of directly constructing the next extension field like in Line 8 of Algorithm 2.2, we need to split this process into multiple steps. We check whether q *would vanish identically* over the *next* extension field $\mathbb{Q}^*[\xi_x]/\langle f\rangle$ but perform the division over the *current* extension field $\mathbb{Q}^*$. Only when q no longer vanishes identically (over $\mathbb{Q}^*[\xi_x]/\langle f\rangle$), we actually update $\mathbb{Q}^*$ to move to the next extension field in Lines 11 and 12.

Algorithm 5.2: Lifting with Lazard's projection

```
1  Function Lazard(A, q)
2      Q* := Q, A* := ∅
3      for x ↦ α_x ∈ A do
4          Let p_x be the minimal polynomial of α_x        ▷ p_x in variable ξ_x
5          Let f ∈ factors(p_x, Q*[ξ_x]) such that f(A*, ξ_x ↦ α_x) = 0
6          if f is linear in ξ_x then
7              Substitute ξ_x − f for x in q
8          else
9              Let q := q[x/ξ_x]
10             while id_{Q*[ξ_x]/⟨f⟩}(q) ≡ 0 do
11                 Let q := q/p                            ▷ Calculation in Q*
12             Let Q* := Q*[ξ_x]/⟨f⟩
13             Let q := id_{Q*}(q)                         ▷ Embedding q into new Q*
14             Let A* := A* ∪ {ξ_x ↦ α_x}
15     return roots(q, A*)
```

Given that Algorithm 5.2 is one of the main building blocks of a CAD implementation, we should try to make it as efficient as possible in practice. First of all, we can greatly simplify the process if α_x is rational. Another interesting idea is to cache the sequence of extension fields for consecutive lifting steps on the same variable assignment or variable assignments with a common prefix. This of course only works if a static variable ordering is used.

It seems tempting to try to modify the variable ordering, hoping to simplify the algebraic operations. For example, one could try to substitute rational assignments first, leaving polynomials with less variables for computationally more expensive algebraic operations.

However, the whole theory behind the modified lifting procedure relies on a single static variable ordering. We can directly see that modifying the variable ordering changes the results in the following – not overly contrived – Example 5.3.

Example 5.3: Different variable orderings

Let $\mathcal{A} = \{x_1 \mapsto 0, x_2 \mapsto 0\}$ be the current variable assignment and $p = x_1^2 x_2 x_3 + x_1 x_2^2 (x_3 - 1)$ the polynomial we want to use for lifting. We now use the modified lifting procedure due to Lazard with the two different variable orderings $x_1 < x_2 < x_3$ and $x_2 < x_1 < x_3$. In both cases, we aim to extend the sample point represented by $\mathcal{A}$ to the third dimension.
For $x_1 < x_2$, we first consider p with respect to $x_1 \mapsto 0$, realize that p vanishes identically and divide by x_1 to obtain $x_1 x_2 x_3 + x_2^2 (x_3 - 1)$. After substitution, we have $x_2^2 (x_3 - 1)$ and substituting x_2 in the same way leaves us with $x_3 - 1$ such that we obtain 1 as the single real root of p in x_3 over $\mathcal{A}$.
For $x_2 < x_1$, we first consider p with respect to $x_2 \mapsto 0$ and see that p vanishes identically as well. We divide by x_2 to obtain $x_1^2 x_3 + x_1 x_2 (x_3 - 1)$. Substitution leaves the other part of the polynomial which is $x_1^2 x_3$ and substituting x_1 in the same way results in x_3. Now, the single real root of p over $\mathcal{A}$ is 0.

Example 5.3 shows that the results of the modified lifting scheme depend on the variable ordering. The underlying theory from [Laz94; MH16; MPP19] only guarantees correctness if the variable ordering used by the entire CAD is employed, and we thus need to stick to the single static variable ordering.

Cylindrical Algebraic Decomposition for SMT solving

We have seen in the previous chapter that the cylindrical algebraic decomposition can be a powerful tool to analyze nonlinear arithmetic formulae. Our aim is to leverage it to allow for reasonably efficient SMT solving for nonlinear arithmetic formulae. Given what we have described so far, the naive approach is straightforward: upon a theory query, construct the corresponding CAD and figure out whether some cell is satisfying.

Within this chapter, we first work on how we understand CAD and go from a monolithic black-box to a modular incremental proof system. Furthermore, we propose a whole range of optimizations to improve upon the naive solution in terms of practical performance and applicability. This includes supporting *incrementality* and *backtracking*, generating *reasons for unsatisfiability*, but also extending CAD to *integer problems*.

The majority of the underlying ideas are already contained in [KÁ20], though we focus on algorithms and data structures for a practical implementation there while we aim for a more conceptual description here.

6.1 Changing perception

We have introduced and understood CAD as a rather monolithic method to decompose the solution space into regions that are invariant with respect to the evaluation of the input formula. These regions then allowed us to answer certain questions about the formula, including whether it is satisfiable.

In this section, we build another intuition about the nature of CAD, which helps us motivate what we call *incremental CAD*. We restrict to the case of SMT solving, that is our formula is purely existentially quantified. The main insight to gain is that we are looking for a method that produces a satisfying witness, and once we have obtained such a satisfying assignment, we discard all the other information a CAD offers.

Recall the two components of CAD (projection and lifting) and their intuitive meanings. The projection accumulates polynomials that induce borders of sign-invariant regions and as such holds the algebraic information about the solution space. The lifting, on the other hand, constructs individual sample points that represent these regions.

Slightly twisting our view on the lifting process, we can also understand it as a *heuristic* process of *guessing* and the projection as merely guiding it. Adopting this notion, we can understand the entire CAD method as a procedure to guess sample points that is guided by the algebraic structure of the input, yielding some interesting observations.

For once, we can terminate the CAD method as soon as the lifting has found a single satisfying sample point because the formula is only quantified existentially. We do not have to care about all the other regions and sample points as a single satisfying sample point is all we need to infer satisfiability. This is essentially an application of [CH91] to the purely existential case.

Furthermore, we could hope to find the satisfying sample point without even considering a (full) projection. If only a few polynomials in the projection are enough to guide the lifting towards a satisfying sample point, we can happily ignore the rest of the projection. Finally, having identified the lifting as the central component for our purpose, we can turn the program flow of the CAD method around. We start with the lifting process and occasionally – for example, when no or only expensive lifting steps can be done – spend a bit of work on the projection and thereby try to *avoid computing* a full projection altogether.

We can summarize our approach with three goals that seem pretty simple: 1. if a satisfying sample point exists, find it as fast as possible and avoid any additional work, 2. if no satisfying sample point exists, eventually produce a regular CAD and inherit all formal guarantees, 3. do this in a way that allows for incremental computations and backtracking (in the spirit of Section 4.3).

In the following, we use this intuition to define a *proof system* for nonlinear real arithmetic constraints based on CAD. The following presentation does not shed a lot of light on how to implement this approach efficiently, thus we refer to [KÁ20] for details on efficient data structures.

6.2 Proof system

The incremental nature of the CAD method proposed here heavily relies on postponing operations by putting them in a queue and trying to terminate without executing all operations that are still enqueued. Following our observation from Example 5.1, we decompose the projection into *projection steps*, while the lifting is naturally separated into *lifting steps*, as defined in the following.

Definition 6.1: Projection step

Let P be a set of polynomials and $Proj$ some CAD projection operator. We call the execution of $Proj(\{p, q\})$ with $p, q \in P$ a *projection step*. We denote a *projection candidate* – representing a future, not yet executed, projection step – by (p, q) and call it *single* (*paired*) if $p = q$ ($p \neq q$).

Note that (almost) all the projection operators we presented in Section 5.2 naturally decompose into a sequence of projection steps. It is important to realize that the notion of projection candidates immediately leads to an incremental building of the projection as executing (or evaluating) projection candidates (usually) yields new polynomials that then give rise to new projection candidates.

Definition 6.2: Lifting step

Let s be a (partial) sample point of dimension k and p a polynomial (from the projection) of dimension $k + 1$. In particular, we also allow $k = 0$ and thus s be

the *empty sample point*. We call the process of substituting s into p and using the real roots of the resulting polynomial to extend s to a set of $(k+1)$-dimensional sample points a *lifting step*. Similar to a projection candidate, we call (s, p) a *lifting candidate*, representing a future, not yet executed, lifting step.

Note that the generation of $(k+1)$-dimensional sample points needs to be performed with respect to the already existing sample points over s. As we assume that the sample points are stored within some tree structure, the existing sample points over s are easily accessible without additional overhead.

As for projection candidates, the notion of lifting candidates immediately yields an incremental method to compute the lifting. Making both the projection and lifting incremental yields a system that always has to choose 1. whether to perform a projection step or a lifting step and 2. which projection (or lifting) step to perform. We conceptually combine projection candidates and lifting candidates in a single queue to obtain maximum flexibility and discuss how to implement this heuristic in Section 6.5.

In order to properly support the removal of constraints, we also need some way to track various dependencies: which polynomials originate from a certain constraint, which polynomials are the result of which projection candidates, and which sample points were produced by lifting of which sample point with which polynomial.

As for the sample points producing other sample points by means of lifting steps, we propose to store them in a *lifting tree*. Every sample point has a parent – the sample point that was used to construct it – with the *empty sample point* being the (unique) root and removing a sample point implicitly removes the whole subtree rooted in it. Though multiple sample points may produce the same *value* on a higher dimension, we consider these different as we identify a sample point in our tree with the full k-dimensional variable assignment.

All other dependencies do not come in a nice tree structure: multiple different projection steps may produce the same polynomial and there is no point in storing them separately. Therefore, we propose the concept of *origins* which essentially list all the reasons for a projection polynomial (or a sample point) to exist and thereby also allow to recognize when it can be removed.

Definition 6.3: Origins of polynomials and sample points

Let p be a polynomial and s a sample point. We call o an *origin of a projection polynomial* p if either $o = \{p\}$ and $p \,\sigma\, 0$ is an input constraint or o is a set of polynomials and $p \in Proj(o)$. We call $o = \{p\}$ an *origin of a sample point* s if p is a polynomial and s is the result of a lifting step involving p.
We denote the set of multiple such origins for p (or s) by $origins(\cdot)$. We denote adding an origin o to this set by $origins(\cdot)$ `+= o`. We write $origins(\cdot)$ `-= p` for the removal of all origins that contain the polynomial p.
If $origins(p) = \emptyset$ (or $origins(s) = \emptyset$) we say that p (or s) *has no origins*, implying that we can remove it from the state of our proof system.

Intuitively, an *origin* is the reason for something to exist in our proof system and *origins* collect several of these. If the origins are empty, any reason for this projection polynomial (or sample point) *ceased to exist* and we can (and maybe should) remove it. Note that when we remove a projection polynomial, we also need to remove this

polynomial from the origins of all other polynomials (and sample points), possibly triggering a whole series of removals that descends through the levels.

Intermediate sample points are a notable exception to this intuition. Their reason to exist is not simply a polynomial, but the existence of (neighboring) root sample points. We thus do not assign origins to intermediate sample points, but instead create (and remove) them as we create (and remove) root sample points. We trust that the handling of intermediate sample points is simple enough for anyone aiming to implement this, so that we only give a rather superficial description of what needs to be done in the following proof system, in particular in the rules `Add-Sample-Finished` and `Delete-Poly-Finished`.

For the presentation of the proof system for CAD, we now fix a common notation. We denote the set of input constraints by C and use P and S for sets of polynomials and sample points, respectively. As in the above definitions, (p, q) denotes projection candidates while (s, p) denotes lifting candidates and we call the queue containing all of them Q.

Our proof system can be in one of the following states, always starting in the default state. The Add-P (Remove-P) states are used do model some details of adding (removing) polynomials to (from) our system while Add-S does the same for adding sample points. We have additional states SAT and UNSAT to indicate satisfiability and unsatisfiability.

Default: $\langle C, P, S, Q\rangle$	Default state
Add-P: $\langle C, P, S, Q\rangle +_o P'$	Add polynomials P' with origins o
Remove-P: $\langle C, P, S, Q\rangle - P'$	Remove polynomials P'
Add-S: $\langle C, P, S, Q\rangle +_o S'$	Add samples S' with origins o
SAT: $\langle C, P, S, Q\rangle \vdash SAT$	Found constraints to be consistent
UNSAT: $\langle C, P, S, Q\rangle \vdash UNSAT$	Found constraints to be inconsistent

Note that we deliberately did not define the SAT and UNSAT states to be final. Instead, we allow our proof system to continue from there by adding new constraints or removing existing constraints to form a new set of input constraints. This, of course, requires our proof system to be able to continue from an already computed CAD – possibly incompletely computed in the case of SAT.

Compared to most other proof systems, we allow for two extensions, namely the *composition of rules* and taking *external input* as described in Section 2.4. We define our initial state to be $\langle \emptyset, \emptyset, \emptyset, \emptyset\rangle$ which requires adding constraints before any computation can take place. We now present the actual proof rules, starting with the ones to add and remove constraints.

`Initial:`

$$\frac{}{\langle \emptyset, \emptyset, \emptyset, \emptyset\rangle}$$

`Continue from SAT:`

$$\frac{\langle C, P, S, Q\rangle \vdash SAT}{\langle C, P, S, Q\rangle}$$

`Continue from UNSAT:`

$$\frac{\langle C, P, S, Q\rangle \vdash UNSAT}{\langle C, P, S, Q\rangle}$$

`Add-Constraint` c**:**

$$\frac{\langle C, P, S, Q\rangle}{\langle C \cup \{c\}, P, S, Q\rangle +_c P'} \quad \textbf{if} \quad P' = \{p \mid c = p\,\sigma\,0\}$$

`Delete-Constraint` c**:**

$$\frac{\langle C \uplus \{c\}, P, S, Q\rangle}{\langle C, P, S, Q\rangle - P'} \quad \textbf{if} \quad P' = \{p \mid c = p\,\sigma\,0\}$$

These rules allow to continue from the SAT and UNSAT states, usually to modify the set of constraints and then solve the modified problem. Note that we technically allow to add and remove constraints whenever we are in the default state – essentially anytime. Though this is possible and sound, we usually assume that constraints are only added and removed directly after we left the SAT or UNSAT state and before any other reasoning takes place.

`Project:`

$$\frac{\langle C, P, S, Q \uplus \{(p, q)\}\rangle}{\langle C, P, S, Q\rangle +_{(p,q)} P'} \quad \textbf{if} \quad P' = Proj(p, q)$$

`Add-Poly:`

$$\frac{\langle C, P, S, Q\rangle +_o P' \uplus \{p\}}{\langle C, P \cup \{p\}, S, Q'\rangle +_o P'} \quad \textbf{if} \quad \begin{array}{l} origins(p) \texttt{ += } o \\ Q_P = \{(p, q) \mid q \in P \cup \{p\} \wedge level(p) = level(q)\} \\ Q_L = \{(s, p) \mid s \in S \wedge level(s) = level(p)\} \\ Q' = Q \cup Q_P \cup Q_L \end{array}$$

`Add-Poly-Finished:`

$$\frac{\langle C, P, S, Q\rangle +_o \emptyset}{\langle C, P, S, Q\rangle}$$

The execution of a projection candidate is performed by the `Project` rule which switches to the Add-P state. `Add-Poly` (leading to `Add-Poly-Finished` eventually) takes care of actually inserting the polynomials into P and adding the appropriate queue entries.

`Lift:`

$$\frac{\langle C, P, S, Q \uplus \{(s, p)\}\rangle}{\langle C, P, S, Q\rangle +_p R} \quad \textbf{if} \quad R = \text{roots}(p, s)$$

`Add-Sample:`

$$\frac{\langle C, P, S, Q\rangle +_p S' \uplus \{s\}}{\langle C, P, S \cup \{s\}, Q'\rangle +_p S'} \quad \textbf{if} \quad \begin{array}{l} origins(s) \texttt{ += } p \\ Q' = Q \cup \{(s, q) \mid q \in P \wedge level(s) = level(q)\} \end{array}$$

`Add-Sample-Finished:`

$$\frac{\langle C, P, S, Q\rangle +_p \emptyset}{\langle C, P, S \cup S', Q \cup Q'\rangle} \quad \textbf{if} \quad \begin{array}{l} S' = \text{missing intermediate sample points} \\ Q' = \{(s, p) \mid s \in S', p \in P \wedge level(s) = level(p)\} \end{array}$$

Executing a lifting step is rather similar to executing a projection step in that we immediately execute the lifting step and then process the set of root sample points that should be added. The intermediate sample points that need to be considered in addition to the real roots R are generated with respect to the already existing sample points from S and inserted after all root sample points have been processed. Note that the intermediate sample points have no origins themselves.

Delete-Poly:

$$\frac{\langle C, P \cup \{p\}, S, Q\rangle - P' \uplus \{p\}}{\langle C, P, S, Q\rangle - P' \cup P''} \quad \textbf{if} \quad \begin{array}{l} \forall p \in P.\ origins(p) \texttt{-=} p \\ \forall s \in S.\ origins(s) \texttt{-=} p \\ P'' = \{p \in P \mid origins(o) = \emptyset\} \end{array}$$

Delete-Poly-Finished:

$$\frac{\langle C, P, S, Q\rangle - \emptyset}{\langle C, P, S', Q'\rangle} \quad \textbf{if} \quad \begin{array}{l} S_O = \{s \in S \mid origins(s) \neq \emptyset\} \\ S_I = \text{ intermediate sample points} \\ \text{Prune obsolete sample points from } S_I \\ S' = S_I \cup S_O \\ Q' = Q \cap ((P \times P) \cup (S' \times P)) \end{array}$$

As discussed before, the removal of polynomials is mainly organized using the origins. To remove a single polynomial p, we remove it from the set of projection polynomials p and prune it from the origins of all other polynomials and all sample points. We then add all polynomials with empty origins to the set of polynomials that are still to be removed. Once all polynomials have been removed, we remove all sample points with empty origins together with all intermediate sample points that are now obsolete, and additionally remove all queue entries that contain a polynomial or a sample point that has been removed.

Remember that we implicitly remove the whole subtree of a sample point, though we denote the removal of sample points using sets here.

SAT:

$$\frac{\langle C, P, S, Q\rangle}{\langle C, P, S, Q\rangle \vdash SAT} \quad \textbf{if} \quad \exists s \in S.\ s \models C$$

Considering that we regard CAD as a search method for a satisfying sample, detecting satisfiability is conceptually simple. As soon as some sample point that satisfies all input constraints exists, we allow to switch to the SAT state immediately. Note that we retain the computed state in order to continue with adding (or removing) constraints.

UNSAT:

$$\frac{\langle C, P, S, \emptyset\rangle}{\langle C, P, S, \emptyset\rangle \vdash UNSAT} \quad \textbf{if} \quad \neg\exists s \in S.\ s \models C$$

The UNSAT rule is based on the claim that this proof system eventually produces a full CAD which we discuss in the following Section 6.2.1 in more detail. Once the queue is empty, the underlying CAD is complete and we can soundly conclude unsatisfiability from the absence of a satisfying sample point. Once again, we retain the computed state to be able to continue after adding (or removing) constraints.

Most simple optimizations can be integrated into this rule system, and we showcase this for the example of adding polynomials as their set of factors. We first define two additional proof rules to factorize polynomials before adding (or deleting) them.

`Add-Poly-Factorized:`

$$\frac{\langle C, P, S, Q\rangle +_o P'}{\langle C, P, S, Q\rangle +_o P''} \qquad \textbf{if} \quad P' \neq P'' = \cup_{p\in P'} \text{factors}(p)$$

`Delete-Poly-Factorized:`

$$\frac{\langle C, P, S, Q\rangle - P'}{\langle C, P, S, Q\rangle - P''} \qquad \textbf{if} \quad P' \neq P'' = \cup_{p\in P'} \text{factors}(p)$$

The integration works by constructing a new proof system by replacing some of the old proof rules by new composed proof rules as follows:

$$\begin{aligned} \texttt{Add-Constraint} &\to \texttt{Add-Constraint} \circ \texttt{Add-Poly-Factorized} \\ \texttt{Project} &\to \texttt{Project} \circ \texttt{Add-Poly-Factorized} \\ \texttt{Delete-Constraint} &\to \texttt{Delete-Constraint} \circ \texttt{Delete-Poly-Factorized} \end{aligned}$$

6.2.1 Correctness and completeness

For the presented proof system to be useful, we want it to be correct and complete. While correctness means that the answer is correct if we terminate, (refutational) completeness ensures the existence of a finite proof for every finite input, and thus termination on every (finite) input. For this proof system, we thus need to prove that 1. the input is satisfiable if we enter the SAT state, 2. the input is unsatisfiable if we enter the UNSAT state, and 3. we always eventually enter the SAT or UNSAT state.

We observe that the proof system allows for multiple satisfiability checks in a sequential manner. For the purpose of this proof, we only consider starting from a default state and disallowing adding or removing constraints until we enter either the SAT state or the UNSAT state. The whole argument that follows rests on the following claim: the internal state of the proof system – the polynomials in P and the sample points in S – *converges* against a full CAD and eventually *is* a full CAD, once the queue Q is empty.

This claim rests on the observation that the presented proof system performs a regular CAD computation where the individual computations are merely *reordered*. If we indeed perform the very same computations – only in a different order – we *inherit* all formal guarantees, in particular, that the problem is unsatisfiable if no satisfying sample point has been found. Of course, we inherit all limitations as well, in particular, the incompleteness due to the projection operators of McCallum or Brown.

The basic steps of CAD are 1. compute all projection polynomials, 2. lift all sample points with all projection polynomials, and 3. check for satisfying sample points. We observe that our definition of *projection steps* and *projection candidates* were defined such that they merely *decompose* any of the existing projection operators and allow for an incremental execution where the queue stores the current progress. Hence, once the queue is empty, the projection operator has been completely executed. Similarly, *lifting steps* and *lifting candidates* only decompose the (usually recursive) lifting process, and once the queue is depleted all sample points have been constructed.

We can easily conclude the above properties from here. If we enter the SAT state then we have found a satisfying sample point. If we enter the UNSAT state then there is no satisfying sample point and, as at this point $Q = \emptyset$, the CAD is complete. As we compute a full CAD – in particular nothing more – we eventually terminate, and once we have $Q = \emptyset$ the only two remaining options are to enter the SAT state or the UNSAT state (in the slightly restricted scenario we consider for this argument).

6.3 Variants of incrementality

While the proof system presented before provides a very flexible framework, it might make sense to restrict it, allowing for a simpler implementation. The choice of which level of incrementality to allow has by far the largest impact, and we discuss this issue in more depth in [KÁ20]. Note that we always assume that the lifting is performed incrementally (or rather partially in the spirit of [CH91]), but vary which constraints are considered and how the projection is performed.

The easiest variant would be to compute a full CAD from scratch for every theory call, using no form of incrementality at all. This variant not only avoids any bookkeeping – like *origins* that we described before – but also allows integrating external tools that are not built with incremental operations in mind. We call this variant *no incrementality* (`CAD-None`). Please consider [Kre18] for some experiences with integrating external tools for this purpose.

Secondly, we can compute a full CAD for every individual theory call but retain the information from the previous theory call and only extend it. This approach exploits that consecutive theory calls are usually very similar, but ignores the possibility to only consider part of the constraints to obtain a conclusive answer. We call this variant *naive incrementality* (`CAD-Naive`).

The next step is to only consider the constraints one after another. In what we call *simple incrementality* (`CAD-Simple`) we only add a single constraint, complete the CAD, and check whether a solution can be found. This allows to avoid considering *hard* constraints in some cases.

Finally, we have implemented the fully incremental projection from the above proof system which we call *full incrementality* (`CAD-Full`) and a variant where polynomials are only *hidden* instead of removed (`CAD-Full`-hide). The latter one was implemented primarily to mitigate frequent recomputations when using *equational constraints* as described in Section 6.6, though equational constraints are not exploited here.

Note that we may also consider to allow backtracking constraints in a different order than how they were added. We call this *non-chronological backtracking* and discuss it in some depth in [KÁ20]. It comes rather naturally with the presented notion of origins but has only a negligible impact on practical performance.

Some results for these different variants of incrementality are presented in Table 6.1, showing that the benefits of incremental CAD computations significantly outweigh the costs of additional bookkeeping.

We can see from Figure 6.1 that the gain is much more substantial for unsatisfiable problems. This may be surprising, considering that we need to compute a full CAD in the case of unsatisfiability which can not benefit from incrementality at all. However, unsatisfiable problem instances oftentimes yield a sequence of theory calls (of which

Solver	SAT		UNSAT		overall	
CAD-Naive	4285	0.34 s	3496	1.33 s	7781	67.7 %
CAD-None	4293	0.35 s	3507	1.40 s	7800	67.9 %
CAD-Simple	4281	0.39 s	3889	1.68 s	8170	71.1 %
CAD-Full	4328	0.37 s	4250	1.69 s	8578	74.7 %
CAD-Full-hide	4328	0.35 s	4255	1.60 s	8583	74.7 %

Table 6.1: Experimental results for different variants of incrementality.

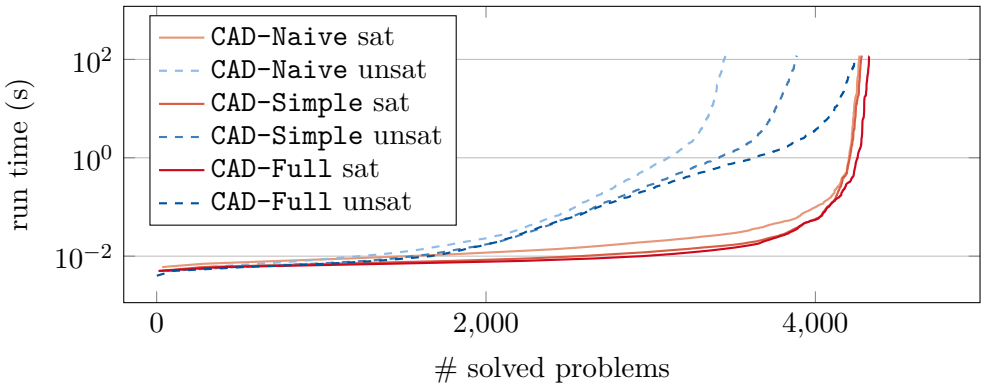


Figure 6.1: Experimental results for CAD-Full for SAT and UNSAT.

many are satisfiable individually) which can benefit significantly from incrementality. Satisfiable problem instances, on the other hand, are usually either solved very quickly or never and thus the improvements due to incrementality are not that significant here. One view could be that one needs to be "lucky" to find a satisfying assignment while unsatisfiability is often found more systematically.

Figure 6.1 shows that after more than one second, only very few problems are found to be satisfiable (less than 2%) but a significant number of unsatisfiable problems are still solved (more than 14%). Naturally, only instances that take a considerable time to solve in the first place can be subject to significant run time improvements.

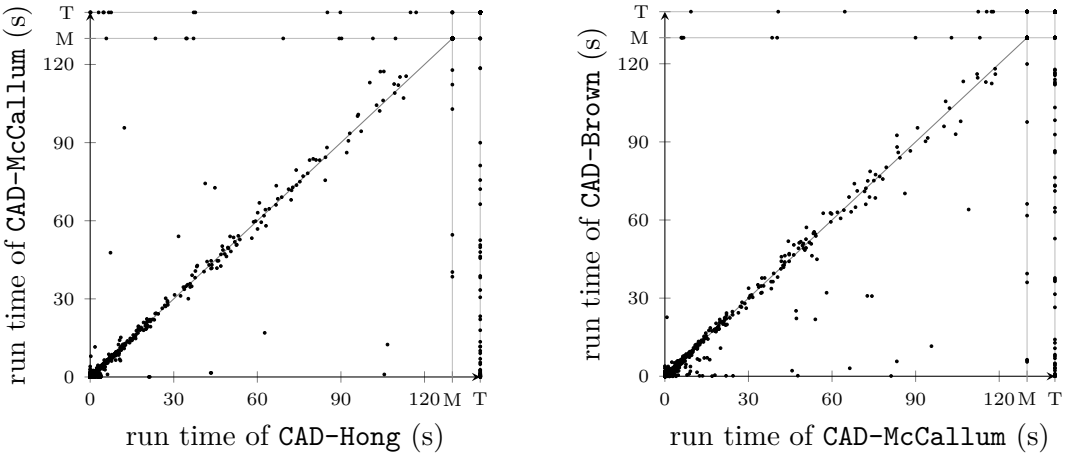
6.4 Projection operators

There exists quite a variety of different projection operators, and we have presented the most important ones in Section 5.2. It is well-known that different projection operators produce sets of polynomials that differ significantly in their size for a traditional cylindrical algebraic decomposition, oftentimes making a huge difference in whether a particular problem can be solved in practice. In general, a smaller set of polynomials not only reduces the computational effort in the projection itself but also yields less sample points that need to be computed. We have given some results on the overall size of the projection in Section 5.2.7.

In our scenario, the same arguments hold in case of unsatisfiability – as we need to generate a full CAD – but whether satisfiable instances can benefit from this, in general, is not immediately obvious. While one might argue that a smaller projection is beneficial as we can exclude unsatisfiable cells faster, it may very well be better to

Solver	SAT		UNSAT		overall	
`CAD-Collins`	4292	0.30 s	4150	1.26 s	8442	73.5 %
`CAD-Hong`	4301	0.29 s	4189	1.46 s	8490	73.9 %
`CAD-McCallum`	4320	0.34 s	4216	1.52 s	8536	74.3 %
`CAD-Lazard`	4322	0.32 s	4229	1.55 s	8551	74.4 %
`CAD-McCallum-partial`	4322	0.30 s	4237	1.64 s	8559	74.5 %
`CAD-Brown`	4328	0.37 s	4250	1.69 s	8578	74.7 %

Table 6.2: Experimental results for different projection operators.



(a) `CAD-Hong` vs. `CAD-McCallum` (b) `CAD-McCallum` vs. `CAD-Brown`

Figure 6.2: Comparison of projection operators.

have more polynomials of which some are "easy" if these easy polynomials are sufficient to heuristically guide the lifting process towards a satisfying cell.

Some further discussion, as well as experiments on the effects of the different projection operators for SMT solving in practice, can be found in [Vie16; VKÁ17], though we need to caution that some implementation issues were discovered after the publication of [Vie16] that affect the last section of the experimental results. An overall summary of the solver performance with varying projection operators is shown in Table 6.2.

These results support the theory that the solver benefits from smaller projection sets, though the impact may not be as strong as expected. A more detailed analysis in Figure 6.2 shows no surprising behavior – contrary to the first (erroneous) findings in [Vie16, Chapter 6]. We see that for the most part `CAD-Hong`, `CAD-McCallum`, and `CAD-Brown` solve the same problem instances in a similar amount of time. As always, a limited number of outliers exist that benefit from a larger projection set in some way.

Another issue we already discussed in Section 5.2 is the problem of completeness, or rather incompleteness, of some projection operators. Examples that provoke this behavior exist in the literature, but as we have already noted in [VKÁ17], this does not seem to be a real issue on our benchmark set. Recall that using McCallum's or Brown's projection operators may lead to missing certain sample points and thereby determining unsatisfiability though a satisfying solution exists.

A key contribution of Lazard is not only its projection operator but also its modified lifting process we discussed in Section 5.3.2. Using this lifting process, McCallum's projection operator (as a superset of Lazard's projection operator, if the optimization from [SS03] is not used) immediately becomes complete as well, and thus only Brown's projection operator is left as incomplete.

Note that we do not implement the "mitigations" discussed in Section 5.3.1 (for example adding delineating polynomials) but simply rely on Lazard's lifting procedure for all projection operators and accept the theoretical incompleteness of Brown's projection operator, verifying that it does not lead to incorrect results on our benchmark set.

6.5 Heuristic choices

The proof system presented before provides a framework to determine the satisfiability of a sequence of sets of constraints by employing a CAD-style method. However, it does not yet yield a deterministic method as it leaves a number of choices how to use the proof system. We now discuss these choices, present some possible heuristics, and evaluate their impact in practice.

For an experimental comparison, we use one particular combination of heuristics as a baseline and vary one single heuristic in every experiment. This particular baseline solver considers only the degree for the projection order and both the level and the type for the lifting order, denoted by `CAD-PO-D` and `CAD-LO-lt`, respectively.

6.5.1 Variable ordering

While most of the presented heuristics have no notable effect if one aims for a complete CAD anyway, the *variable ordering* is known to have a significant impact on the size of a complete CAD. The problem of selecting a good variable ordering has already been studied in the literature, for example, in [Bro04; DSS04; HEW+14; EBD+14]. Though we carried out our own experiments here, we would generously describe them as fruitless: we could mostly confirm that the complexity of CAD seems to grow with some "inherent algebraic hardness".

In most cases, changing the variable order does not change the solver behavior, and if it does the effects are extremely difficult to predict. In our view, this is also reflected by the fact that current research on this topic makes use of machine learning techniques, which could be interpreted malignly as surrendering to an inherently difficult problem.

Beyond that, we have an additional level of complications in our scenario: we not only want to compute one complete CAD but a sequence of possibly incomplete CADs. Even worse, we can not even say beforehand *which CADs* we need to compute as this depends on how the SAT solver behaves, which in turn depends on the infeasible subsets we generate. We discuss these issues to some degree in [NKÁ19], though in the context of `MCSAT`, but the underlying problem is essentially the same.

Therefore, we refrain from further analysis of variable orderings here and just note that we use what is called the *triangular ordering* in [EBD+14] based on all constraints from the formula and keep it static over the whole run. We acknowledge that this heuristic was suggested in the context of the RegularChains library we briefly described in Section 1.1.4 and thus it is not clear whether it is a particularly good heuristic for a regular CAD projection.

Solver	SAT		UNSAT		overall	
CAD-PO-LD	4323	0.31 s	4224	1.50 s	8547	74.4 %
CAD-PO-lD	4315	0.35 s	4245	1.55 s	8560	74.5 %
CAD-PO-PD	4326	0.32 s	4248	1.62 s	8574	74.6 %
CAD-PO-SD	4327	0.34 s	4249	1.62 s	8576	74.6 %
CAD-PO-D	4328	0.37 s	4250	1.69 s	8578	74.7 %

Table 6.3: Experimental results for different projection orders.

Solver	SAT		UNSAT		overall	
CAD-LO-s	4312	0.30 s	4247	1.58 s	8559	74.5 %
CAD-LO-tsa	4319	0.50 s	4247	1.61 s	8566	74.6 %
CAD-LO-ts	4320	0.51 s	4248	1.60 s	8568	74.6 %
CAD-LO-ls	4326	0.44 s	4249	1.64 s	8575	74.6 %
CAD-LO-lta	4328	0.44 s	4248	1.65 s	8576	74.6 %
CAD-LO-ltsa	4328	0.46 s	4248	1.62 s	8576	74.6 %
CAD-LO-tlsa	4329	0.48 s	4247	1.59 s	8576	74.6 %
CAD-LO-lts	4328	0.49 s	4249	1.65 s	8577	74.7 %
CAD-LO-t	4329	0.35 s	4248	1.58 s	8577	74.7 %
CAD-LO-lt	4328	0.37 s	4250	1.69 s	8578	74.7 %

Table 6.4: Experimental results for different lifting orders.

6.5.2 Queue ordering

Another major heuristic is of course when to work on which queue element from Q, that is when to apply `Project` or `Lift` on which projection or lifting candidate. We model this as a global ordering on the elements in Q and call this ordering the *CAD scheduler*. In practice, we usually decompose this scheduler into one part that decides whether we should project or lift and separate orderings on the projection candidates and lifting candidates. Our current (rather naive) heuristic is to always perform a complete lifting before the next projection step is computed.

We also experimented with other heuristics – for example, completing lifting on all rational sample points before computing a projection step and only considering real algebraic sample points when the projection is complete, and some variants thereof – but contrary to our expectations, we could not match the performance of the naive approach. We thus only show results for this naive variant here, still convinced, though, that significant improvements are possible, in particular on restricted inputs.

Within this general ordering that one could describe as *lifting first*, we can now impose an ordering on the projection candidates and the lifting candidates separately. Our approach is to apply *lexicographical* orderings to combine multiple properties of each candidate into one consistent ordering.

For projection candidates, we consider the level of the involved polynomials (`L` and `l` prefer *high* and *low* levels), their type (`P` and `S` prefer *paired* and *single* projection candidates), and finally their degree (`D`). Some results for varying *projection orders* based on these properties are shown in Table 6.3.

For the lifting candidate, or rather its sample points, we consider the absolute value (`a`), the level (`l`), the approximate size of its representation in memory (`s`), and its type (`t`, preferring integers over rationals over real algebraic numbers). The respective experimental results for some combinations of these properties are shown in Table 6.4.

As we can see, both orderings have a limited but noticeable impact on the overall performance of the solver. However, our experience shows that the differences do not persist over time – or rather implementation and configuration changes – but seem to correlate with other components of the solver in obscure ways. In [KÁ20], we evaluated the very same orderings on projection and lifting candidates, but of course, the solver around it has changed since, as well as the benchmarks that are considered. While we noted in [KÁ20] that using the level of a projection candidate seems to be beneficial, the most recent results shown in Table 6.3 suggest the opposite conclusion.

6.5.3 Other heuristics

Apart from these obvious heuristics, one must make a number of further choices when implementing the presented proof system.

Syntactic variable elimination. The main driver of the asymptotic complexity of the CAD method is the number of variables. Hence, eliminating variables should improve the performance drastically, at least if we can do so cheaply. Therefore, we propose to check for equalities that yield a unique solution for one variable (usually equalities that are *linear* in this variable) and use them to eliminate this variable from the set of constraints before starting the actual CAD computation.

However, changing the set of variables is fundamentally opposed to the idea of incrementality. We thus propose to keep the full variable ordering intact and simply let the polynomials skip the levels that correspond to eliminated variables.

Factorization. As already mentioned, it can make sense to factorize every polynomial and add the factors to the projection instead of the original polynomial. Given that we need to make the set of polynomials a *finest square-free basis* anyway for most projection operators – notably McCallum's, Lazard's, and Brown's – we always factorize all polynomials and thereby get a *finest irreducible basis*.

We used to simply ignore that we need a finest square-free basis in the past and note that this did neither cause incorrect results nor did the overall performance seem to degrade significantly. Obtaining a finest square-free basis without computing a full factorization turns out to be much more difficult than one would think, mostly because of the incremental nature of our projection and in particular the removal of polynomials as we already note in [KÁ20]. We thus now always use full factorization as it not only resolves all theoretical issues but also provides a (very small) performance increase.

Sample point generation. The proof system features the `Lift` rule that vaguely states that one should construct the "samples for R", R being a list of real roots. While the samples need to cover all sign-invariant regions – one below the smallest root, one above the largest root, and one between every two consecutive roots – it is not clear how exactly to select them.

While a number of possible heuristics may seem reasonable – the midpoint, an integer or more towards zero, maybe even selecting more than a single point – the more important question is whether it actually makes a difference. We have looked at this in [KÁ20] and concluded that the effect of this selection seems to be negligible.

Subroutines. Most parts of the proof system make use of more or less complex subroutines that all come with their own heuristics that influence their exact result (or at least their performance) and how they should be chosen in the context of CAD – or even in combination with the other heuristics – is mostly unclear. We briefly mention a few of them to give a rough feeling for the breadth of this problem.

Even for storing numbers as our most basic elements, there is a significant variety of possible options. We have already discussed the representation of real algebraic numbers in Section 2.5, which implicitly assumed some way to store rationals. Multiple options exist here, either based on fractions of arbitrarily large integers as in GMP [Gt19] or CLN [HK96], or based on arbitrarily large floating-point numbers as in MPFR [FHL+07]. For certain applications, *binary rationals* or *dyadic rationals* (with denominators being powers of two) have been shown to be particularly efficient as well [MP13]. We use rationals from GMP unless explicitly stated otherwise.

The basic objects we use in CAD are polynomials, which again feature a vast amount of different representations, each with technical implementation details that are crucial for practical performance. Even well-established software packages like Maple still change their internal representations for multivariate polynomials significantly from time to time, as witnessed by [MP14].

For more specialized cases like univariate polynomials or fully factored polynomials (as used by `QEPCAD B`), other representations are usually preferred. Our implementation features sparse multivariate polynomials (as a list of terms with sparse exponent vectors) and dense univariate polynomials (with possibly multivariate coefficients).

Resultants and discriminants can be obtained in a variety of ways. They are usually defined as determinants of the Sylvester matrix – or some of its variants or the Bézout matrix – but are usually computed more efficiently via subresultant sequences. Methods to compute them have a long history and are continuously improved, for example [Col67; Col71a; GCL92; Duc00], though with sometimes unclear benefits and trade-offs. Our current implementation is based on [Duc00].

Real root isolation is used in the `Lift` rule to construct new sample points. A range of different possibilities was explored in [Kre13], mostly aiming to exploit approximative methods as preconditioning to arguably naive bisection in the spirit of [CA76] or [Sag12]. Many questions arise in this context: should we fully factor a polynomial before isolating its roots? Where should we perform splits for bisection? Is it worth using numerical approximations beforehand? How should we count the roots within an interval?

Given the sheer number of design decisions that must be made, combined with the many side effects that any of these could have on all the others, one quickly realizes that it is essentially hopeless to obtain an optimal configuration. We rather try to optimize every issue "locally" – or in combination with a very limited number of other heuristics – and simply hope that the side effects on uncared for subroutines are negligible.

6.6 Equational constraints

An important optimization for CAD that we already discussed in Section 5.2.8 are *equational constraints* as initially described in [Col98]. Essentially, we employ equalities to not only simplify the lifting process – by removing sample points that do not satisfy the equalities – but also to avoid certain projection steps. Given the significant practical impact on some formulae, we now integrate this optimization into the CAD proof system.

We have already discussed some details of using equational constraints in Section 5.2.8, in particular the existence of both the *restricted* and *semi-restricted* projection operator and the restrictions on their applicability. We do not aim to incorporate these in detail here, but rather give a rough template on how to do so.

The main idea is to store which equalities are used as equational constraints in a new part of the state we call EQ and to maintain a separate queue called Q_* that contains everything that is disabled due to the equational constraints used to simplify the projection. For the sake of an easy presentation, we now present a simple – and probably suboptimal – scheme to deal with this issue and refer more efficient implementations to the reader. We assume that making the following adaptions reasonably efficient is rather straightforward for anyone implementing an incremental CAD along the lines of what was described before.

Algorithm 6.1: Reduce projection based on equational constraints

Input: constraints C, polynomials P, equational constraints $EQ \subseteq C$

Function `ReduceProjection`(C, P, EQ)
- $P' = P \cap \{p \mid p\,\sigma\,0 \in C\}$ ▷ `polynomials from input constraints`
- **while** P' *changed* **do**
  - Let $O = \{\{p\}, \{p, q\} \mid p \in P', q \in EQ\}$
  - $P' = P' \cup \{p \in P \mid origins(p) \cap O \neq \emptyset\}$
- **return** P'

Let us first define a helper method to remove polynomials from the projection due to equational constraints in Algorithm 6.1. This method, that we call `ReduceProjection`, iteratively collects polynomials that are still active under a restricted set of origins, consisting of single projection steps or projection steps involving one of the equational constraint polynomials.

Algorithm 6.2: Reduce queue based on equational constraints

Input: polynomials P, sample points S, queue Q,
equational constraints $EQ \subseteq C$

Function `ReduceQueue`(P, S, Q, EQ)
- **return** $Q \cap (\{(p, p) \mid p \in P\} \cup (P \times EQ) \cup (EQ \times P) \cup (S \times P))$

Similarly, we define the helper method `ReduceQueue` in Algorithm 6.2 that restricts the queue to all elements consistent with the new set of projection polynomials and sample points and also removes projection candidates that would yield inactive projection polynomials due to the equational constraint polynomials.

This now allows us to define the new proof rule `EqC-Cleanup` that moves everything from either P, S, or Q to Q_* which should be disabled due to the active equational constraint polynomials EQ. We also define the inverse rule `EqC-Restore` to restore all elements from Q_*, intended to be called when a polynomial is removed from EQ.

`EqC-Cleanup`:

$$\frac{\langle C, P, S, Q, Q_*, EQ\rangle}{\langle C, P', S' \cup S_I, Q', Q'_*, EQ\rangle} \quad \textbf{if} \quad \begin{array}{l} P' = \texttt{ReduceProjection}(C,\ P,\ EQ) \\ S' = \{s \in S \mid \exists o \in \mathit{origins}(s).\ o \subseteq P'\} \\ S_I = \text{intermediate samples from } S \text{ required for } S' \\ Q' = \texttt{ReduceQueue}(P',\ S' \cup S_I,\ Q,\ EQ) \\ Q'_* = Q_* \cup (P \setminus P') \cup (S \setminus (S' \cup S_I)) \cup (Q \setminus Q') \end{array}$$

`EqC-Restore`:

$$\frac{\langle C, P, S, Q, Q_*, EQ\rangle}{\langle C, P', S', Q', \emptyset, EQ\rangle} \quad \textbf{if} \quad \begin{array}{l} P' = P \cup \mathit{polynomials}(Q_*) \\ S' = S \cup \mathit{samples}(Q_*) \\ Q' = Q \cup (Q_* \setminus (P' \cup S')) \end{array}$$

To actually make use of this technique, we invoke the new proof rule `EqC-Cleanup` whenever `Add-Poly-Finished` was used and let any invocation of `EqC-Restore` be followed by `EqC-Cleanup`:

$$\texttt{Add-Poly-Finished} \to \texttt{Add-Poly-Finished} \circ \texttt{EqC-Cleanup}$$
$$\texttt{EqC-Restore} \to \texttt{EqC-Restore} \circ \texttt{EqC-Cleanup}$$

Note that we did not yet define how the set of equational constraint polynomials EQ should be maintained. We define a new proof rule `EqC-Select` that enables as many equational constraints from C as possible by adding their polynomials to EQ and analogously a proof rule to deactivate a given equational constraint polynomial. Both are designed to be integrated with `Add-Constraint` and `Delete-Constraint`, respectively.

`EqC-Select`:

$$\frac{\langle C, P, S, Q, Q_*, EQ\rangle}{\langle C, P, S, Q, Q_*, EQ \cup N\rangle} \quad \textbf{if} \quad \begin{array}{l} L(P) \coloneqq \{\mathit{level}(p) \mid p \in P\} \\ N = \{p \mid p = 0 \in C\} \text{ such that} \\ |L(EQ \cup N)| = |EQ \cup N| \end{array}$$

`EqC-Unselect` c:

$$\frac{\langle C, P, S, Q, Q_*, EQ\rangle}{\langle C, P, S, Q, Q_*, EQ'\rangle} \quad \textbf{if} \quad EQ' = EQ \setminus \{p \mid c = p\,\sigma\,0\}$$

Consequently, we invoke `EqC-Select` after `Add-Constraint` and forward the removed constraint from `Delete-Constraint` to `EqC-Unselect`. Note that after unselecting an equational constraint polynomial we may be able to select another one on the same level and we therefore invoke `EqC-Select` again. Additionally, we add the new state components Q_* and EQ to all other proof rules and let them be the empty set initially.

$$\texttt{Add-Constraint} \to \texttt{Add-Constraint} \circ \texttt{EqC-Select}$$
$$\texttt{Delete-Constraint}\ c \to \texttt{Delete-Constraint}\ c \circ \texttt{EqC-Unselect}\ c \circ \texttt{EqC-Select}$$

Solver	SAT		UNSAT		overall	
CAD-ECS-Hong	4303	0.25 s	4200	1.47 s	8503	74 %
CAD-Hong	4304	0.28 s	4199	1.46 s	8503	74 %
CAD-EC-Hong	4304	0.30 s	4201	1.55 s	8505	74 %
CAD-ECS-McCallum	4320	0.33 s	4227	1.56 s	8547	74.4 %
CAD-McCallum	4321	0.35 s	4226	1.53 s	8547	74.4 %
CAD-EC-McCallum	4320	0.33 s	4228	1.59 s	8548	74.4 %
CAD-EC-Brown	4327	0.32 s	4256	1.63 s	8583	74.7 %
CAD-Brown	4328	0.35 s	4255	1.60 s	8583	74.7 %
CAD-ECS-Brown	4328	0.35 s	4257	1.64 s	8585	74.7 %

Table 6.5: Experimental results for equational constraints.

As we have already mentioned, the actual implementation may look somewhat different. In particular, one might want to employ a more efficient mechanism to restore elements from Q_* – instead of restoring everything and filtering again – and implement `ReduceProjection` level-by-level instead of a generic fixed-point iteration. Another interesting question is which heuristic should be implemented to select an equational constraint in `EqC-Select`.

A brief overview of the impact of using equational constraints as implemented in [Hae18; HKÁ18] is given in Table 6.5. We compare (for Hong's, McCallum's, and Brown's projection operators) our implementation without equational constraints, with the restricted projection operator (`EC`), and the semi-restricted projection operator (`ECS`). Though literature indicates that equational constraints can have a substantial impact, it makes no consistent difference in our use case. We conjecture that the possible effect of equational constraints is reduced by the incremental setting and superseded by the syntactic variable elimination that we describe in the following Section 6.6.1.

6.6.1 Variable elimination and the resultant rule

An interesting topic concerning equational constraints is the *resultant rule* as described in Definition 5.14, taken from [Col98]. We could integrate the resultant rule into the proof system defined above, as well as into an actual implementation thereof. Following the idea of postponing all computations as long as possible, it seems tempting to apply the resultant rule lazily and generate new equational constraints dynamically.

However, we want to caution everyone planning to go down this road for several reasons. First of all, this adds a whole new layer of complexity that we have to deal with. To just give one example, enabling an equational constraint may invalidate origins of another equational constraint on a lower level and thus lead to disabling another one – which in turn might reactivate yet another one even further down. Getting all this right – on top of all the bookkeeping we have to do anyway – is anything but trivial.

Secondly, we observe a very fundamental issue in the interaction of equational constraints and incrementality. Adding constraints only leads to *things being added* to our CAD and removing constraints only leads to *things being removed* from our CAD. Also, we only remove polynomials (and sample points) that we are sure we do not need anymore (except if the constraint is later added again). When adding an equational constraint, however, we also remove, and when removing an equational constraint

Solver	SAT		UNSAT		overall	
CAD-hide	4323	0.31 s	3887	2.34 s	8210	71.5 %
CAD-ECS	4323	0.31 s	3890	2.41 s	8213	71.5 %
CAD+RR-hide	4325	0.31 s	3921	2.26 s	8246	71.8 %
CAD-ECS+RR	4326	0.36 s	3922	2.27 s	8248	71.8 %
CAD+VE+RR-hide	4328	0.35 s	4255	1.60 s	8583	74.7 %
CAD-ECS+VE+RR	4328	0.35 s	4257	1.64 s	8585	74.7 %
CAD+VE-hide	4332	0.35 s	4346	1.91 s	8678	75.5 %
CAD-ECS+VE	4333	0.35 s	4348	1.90 s	8681	75.6 %

Table 6.6: Experimental results for the resultant rule.

we restore what we removed before. This significantly raises the amount of "induced changes", and possibly the amount of information we need to recompute.

Finally, we may very well argue that eagerly applying the resultant rule can be very beneficial, even if we disregard the advantages due to less bookkeeping and simpler code or even using equational constraints at all. The resultant rule gives rise to new equational constraints that might allow to *syntactically eliminate variables* as mentioned in Section 6.5.3 and shown in Example 6.1, and the number of variables is one of the main drivers of run-time complexity in CAD. An eagerly applied combination of the resultant rule and syntactic variable elimination may thus be very beneficial and outweigh the possible benefits of incrementality by far, also allowing to be applied to CAD implementations that do not exploit equational constraints internally.

Example 6.1: Variable elimination with the resultant rule

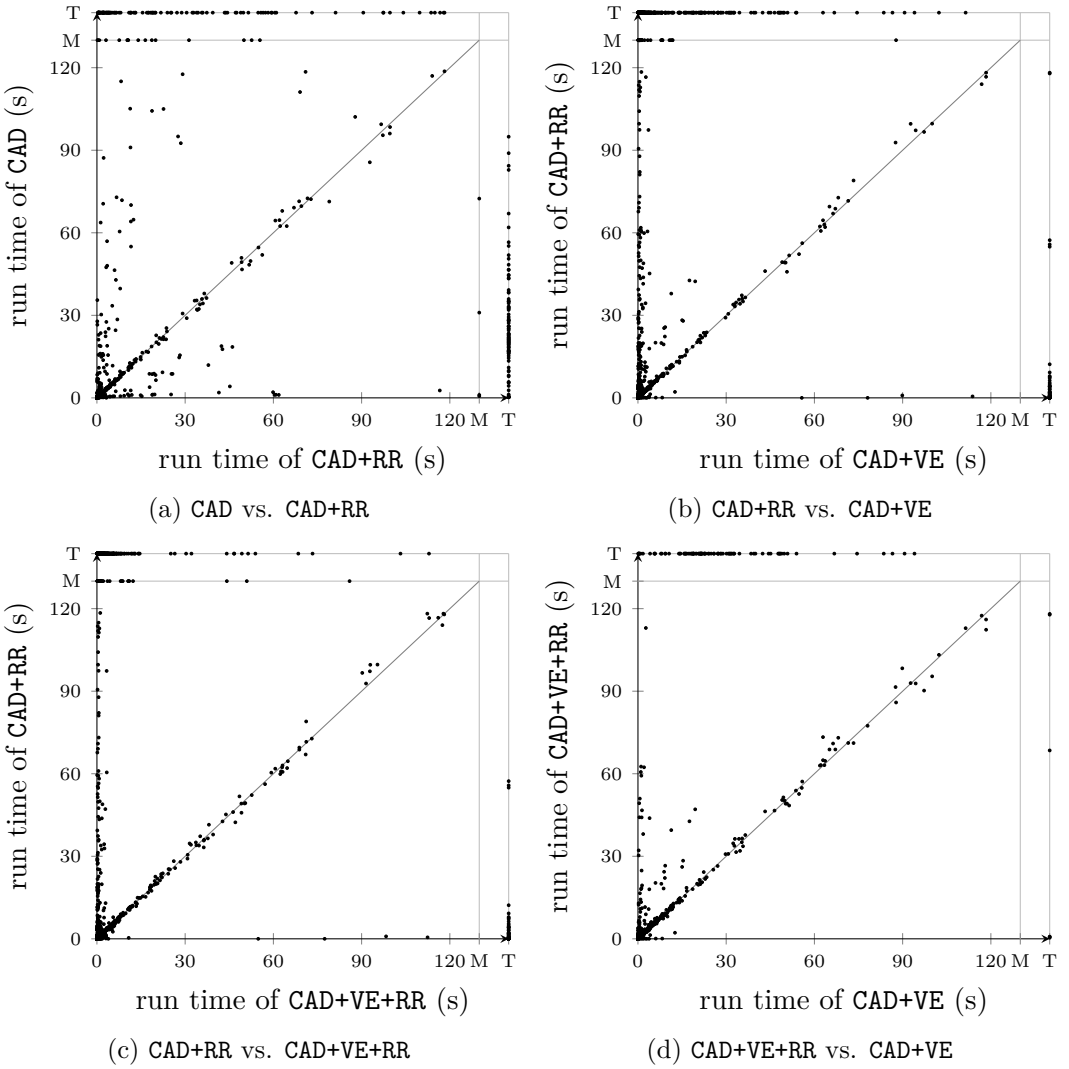
Consider the input formula $\varphi \equiv y^2x^2 = 0 \wedge y^2 - 1 = 0$ and assume that we eliminate y first. The resultant rule states that

$$(y^2x^2 = 0 \wedge y^2 - 1 = 0) \implies \operatorname{res}_y(y^2x^2, y^2 - 1) = 0$$

Observe that neither $y^2x^2 = 0$ nor $y^2 - 1 = 0$ can be used to syntactically eliminate y (or x) because neither defines a unique solution for either variable. While $y^2x^2 = 0$ yields a solution of the form $y = 0 \vee x = 0$, $y^2 - 1 = 0$ gives $y = 1 \vee y = -1$. The resultant rule however gives $x^4 = 0$ with the unique solution $x = 0$ that we can use to eliminate x and obtain $\varphi' \equiv y^2 - 1 = 0$.

We thus devise four variants to evaluate the impact when equational constraints are used and when they are not: without any preprocessing (CAD and CAD-ECS); with the resultant rule only (CAD+RR and CAD-ECS+RR); with syntactic variable elimination only (CAD+VE and CAD-ECS+VE); with the resultant rule and syntactic variable elimination (CAD+VE+RR and CAD-ECS+VE+RR).

Additionally, we show results for the regular (fully incremental) CAD and when equational constraints are used. We see in Table 6.6 that these preprocessing techniques have essentially no impact for satisfiable instances but can make a big difference in case of unsatisfiability. Both, the resultant rule and the variable elimination, improve the solver significantly, combining them however yields worse results than using variable elimination alone.



(a) CAD vs. CAD+RR

(b) CAD+RR vs. CAD+VE

(c) CAD+RR vs. CAD+VE+RR

(d) CAD+VE+RR vs. CAD+VE

Figure 6.3: Comparison of unsatisfiable instances.

Furthermore, the comparisons in Figure 6.3 show that these different heuristics essentially follow a strict order in that a "worse" heuristic solves almost no examples a "better" heuristic can solve. We see this from the fact that only very few benchmark instances end up in the lower right triangle.

We observe that the resultant rule in some way *emulates* variable elimination in that computing the resultant of the form $\mathrm{res}_x(x - p, q)$ – where p does not contain x – is essentially equivalent to substituting $x = p$ into q. The performance of the resultant rule – compared to variable elimination – thus seems to suggest that it merely does a poor job at simulating variable elimination, but has no real benefit on its own, at least on this benchmark set and if (syntactic) variable elimination is performed.

This even holds true when equational constraints are used, which should benefit in particular from new equalities generated by the resultant rule. We thus consider the resultant rule a nice tool that one should consider for particular problem classes, but not beneficial in general for our use case.

6.7 Infeasible subsets

In the case of an inconsistent set of constraints, we would like to provide the SAT solver with an *infeasible subset* of constraints as already discussed in Section 4.3.4. Unfortunately, we can not easily derive any information from our state when we determine unsatisfiability, mostly because we have no "constructive" criterion that we check – like we would, for example, for the simplex method – but only run out of solution candidates. In this sense, we can only perform an *a-posteriori* analysis of an (almost) regular CAD computation. Therefore, we employ a technique that is very similar to the one described in [JDF15] and directly based on the work presented in [Hen17]. In the following, we assume that our CAD method has run to completion and found no sample points to be satisfying.

When we found the problem to be unsatisfiable, we have for every leaf node in the sample tree at least one constraint that evaluates to *false* on this leaf node. Note that the leaves may not be of full dimension: we may stop lifting a partial sample point if it is already invalidated by a constraint, roughly following [CH91]. Fundamentally, an infeasible subset is any subset of constraints such that every sample point is still invalidated (or "covered") by this subset.

We approach this problem in the spirit of a set cover problem and note the similarity to minimal infeasible subsets from Definition 4.4.

Definition 6.4: Set cover problem

Let Ω be a finite set and $S \subseteq \mathcal{P}(\Omega)$. We call $C \subseteq S$ a *set cover of* Ω if

$$\bigcup_{s \in C} s = \Omega \quad \text{or equivalently} \quad \forall o \in \Omega.\ \exists s \in S.\ o \in s$$

We call C a *minimum set cover* if it has minimal cardinality among all set covers and a *minimal set cover* if no proper subset of C is a set cover. Finding a *minimum set cover* C for some Ω and S is called the *set cover problem* $SC(\Omega, S)$.

The set cover problem is one of the classical NP-hard problems from [Kar72], suggesting that computing an optimal solution is oftentimes infeasible. There is, however, a simple greedy heuristic which achieves an essentially optimal approximation [Chv79]. We use the set cover problem to compute infeasible subsets as follows.

Definition 6.5: Infeasible subsets by set covers

Let C be constraints and S the sample points constructed by CAD. We know that $\forall s \in S.\ \exists c \in C.\ c(s) = \mathit{false}$ which essentially matches the alternative formulation of the set cover problem. Let $S_c = \{s \in S \mid c(s) = \mathit{false}\}$ for every $c \in C$, then finding an infeasible subset is equivalent to the set cover problem $SC(S, \{S_c \mid c \in C\})$.

Theorem 6.1: Correctness of infeasible subsets by set covers

Any set cover X that is a solution to $SC(S, \{C_c \mid c \in C\})$ is a proper infeasible subset. If X is a minimal (minimum) set cover, it is a minimal (minimum) infeasible subset.

Proof. We defined the set cover problem such that the elements of the universe are sample points and a sample point is *covered* if a constraint evaluating to *false* for this sample point is selected. As a set cover *covers* all sample points, no sample point is satisfiable and the selection of constraints is an infeasible subset. If X is a minimal set cover, for each of its proper subsets some sample point remains uncovered and thus is satisfiable. Hence, X has no proper subset that is an infeasible subset and X is a minimal infeasible subset. Analogously, if X is a minimum set cover, it is also a minimum infeasible subset. □

As we have transformed our SMT specific question for infeasible subsets into a rather simple and well-understood set cover problem, we can now use this new formalization to actually compute infeasible subsets. As already argued, *minimum infeasible subsets* may be hard to compute as they are one of the original NP-hard problems, so we instead use heuristics in the spirit of [Chv79] to obtain *minimal infeasible subsets*.

Though this gives us a nice way to obtain infeasible subsets, there are still a few open questions when integrating this approach with an actual CAD implementation. First of all, we need to obtain the sets C_c from our CAD and may realize that our CAD, in fact, does not evaluate all constraints on every sample point. We usually only evaluate until we find a single constraint that invalidates a particular sample point and avoid further evaluations to eliminate unnecessary overhead.

We could go ahead and use these partial sets, accepting some interesting consequences. In this case, all sets are singleton sets and thus we essentially have no combinatorial problem. We can simply collect all constraints that are used at some point to invalidate a sample point and are done, potentially saving a lot of work. It however also means that which constraints are part of the infeasible subset is determined by the order in which we evaluate constraints on a sample point.

Alternatively, we can evaluate all sample points with all constraints and use the full sets. This gives us full flexibility on which covers to compute and allows finding minimal infeasible subsets of the actual problem. We find that these additional evaluations are not that costly in practice, at least when compared to the effort we put in the CAD computation in the first place.

We can also understand this problem as a linear programming problem as described in [JDF15]. We want to note, however, that the set cover formulation is quite promising as we usually have a large number of sample points and only a few constraints, thus also the underlying implementation of [JDF15] employs a set cover heuristic.

We finally observe that the set cover problem is usually very regular and has a lot of redundancies. It is common, that only a single constraint invalidates a particular sample point. Such an *essential constraint* must be part of any infeasible subset and we can thus eliminate it from the set cover problem, as well as all sample points that are covered by it. We also know that the CAD usually has multiple sample points for a single sign-invariant region, and these sample points should be completely identical within the set cover problem. We can thus remove such *duplicate columns*, conceding that this may very well change the selections of the greedy heuristics from [Chv79].

Systematically applying these techniques is surprisingly powerful in practice, sufficiently reducing the problem size to allow for an optimal brute force solution of the set cover problem instead of the heuristic approach referenced above, as shown in [Hen17].

Solver	SAT		UNSAT		overall	
`CAD-MIS-Greedy-PP`	4327	0.32 s	4249	1.67 s	8576	74.6 %
`CAD-MIS-Greedy-Weighted`	4329	0.36 s	4247	1.60 s	8576	74.6 %
`CAD-MIS-Hybrid`	4328	0.33 s	4248	1.62 s	8576	74.6 %
`CAD-MIS-Trivial`	4325	0.32 s	4251	1.66 s	8576	74.6 %
`CAD-MIS-Exact`	4328	0.34 s	4249	1.66 s	8577	74.7 %
`CAD-MIS-Greedy`	4328	0.37 s	4250	1.69 s	8578	74.7 %

Table 6.7: Experimental results for heuristics for minimal infeasible subsets.

A classical question in the context of infeasible subsets is whether the size of the infeasible subset is the best ranking criterion. One could imagine that selecting *easy constraints* is beneficial – as this increases their activity in the SAT solver and thus encourages working on easier parts of the problem – or selecting constraints of *small decision level* could speed up the search – by excluding larger parts of the Boolean search space. We can easily enhance the set cover problem to a *weighted set cover problem* to allow for arbitrary weights of the constraints, and appropriate heuristics exist for the weighted variant as well.

Note, however, that this might only end up as a proxy to modify the SAT solvers decision heuristic. Past research – as well as our practical experience – suggests that it is usually a bad idea to mess with the decision heuristic. Adding these criteria almost consistently either does not change the overall performance at all or even impairs it.

This should not come as a surprise as recent progress on SAT decision heuristics points towards structural properties of the clauses (like *literal block distance* [AS09]) or the whole formula (like *community structure* [LGZ+15]) or even optimizing for behavioral properties of the solver (like *learning rate* [LGP+16]), but away from properties of the individual literals. It is not clear to us, however, how such structural properties can be properly integrated into the approach described above and we, therefore, leave it for future research.

Another interesting direction would be to investigate whether adding multiple infeasible subsets at once can be beneficial. It could be possible to construct different infeasible subsets and provide the SAT solver with more diverse knowledge. We conjecture, however, that it might be difficult to have multiple infeasible subsets that do not introduce a lot of redundant information.

To evaluate these different possibilities, we propose the following different heuristics for computing infeasible subsets where "preprocessing" includes selecting essential constraints and removing duplicate columns as described in [Hen17]: `CAD-MIS-Trivial` returns the trivial infeasible subset; `CAD-MIS-Greedy` employs the standard greedy heuristic from [Chv79]; `CAD-MIS-Greedy-PP` applies preprocessing before using the greedy heuristic; `CAD-MIS-Greedy-Weighted` uses the standard greedy heuristic where constraints are weighted according to their complexity; `CAD-MIS-Exact` computes a *minimum* covering after preprocessing; `CAD-MIS-Hybrid` first uses preprocessing, then the greedy heuristic until at most 12 constraints are left, and then computes a *minimum* covering. The threshold of 12 constraints is taken from [Hen17] and makes sure that we only compute a minimum covering if it is sufficiently fast.

We see the impact of the different heuristics – or rather the lack thereof – in Table 6.7.

As already noted, the choice of the heuristic is almost irrelevant on the considered benchmark set and the differences do not seem to be statistically significant. Therefore, we refrain from a deeper analysis here and refer to [Hen17] which contains some interesting observations: for example, in these experiments the number of required theory calls decreased significantly with `CAD-MIS-Hybrid` (compared to `CAD-MIS-Greedy`), though this is not reflected in the overall solver performance.

We note that many interesting questions and possible further heuristics are left open and untested here. The impact of heuristics for infeasible subsets have proven to be minor at best on our benchmarks – mostly due to the low Boolean complexity – but some results from [Hen17] and practical experience with other theories indicate that one should nevertheless keep this topic in mind for future research.

6.8 Integer problems

We have presented and understood CAD as a decision procedure for nonlinear real problems. A common requirement in practice is that assignments should be integral, and it is well-known that the theory of nonlinear integer arithmetic is undecidable – though the search space becomes *smaller*. This is somewhat similar to the linear case, where polynomial algorithms exist for the reals, but linear integer problems have exponential complexity (if we assume $P \neq NP$).

Arguably the most common approach to tackle linear integer problems is called *branch and bound*. It essentially computes a real solution and – if this solution is not integral – excludes some region around this solution so that the removed part does not contain any integral solution.

The simplest variant of branch and bound roughly works as follows: Assume the real solution to our formula φ is $x = 0.5$, we perform two recursive calls with $\varphi \wedge x \leq 0$ and $\varphi \wedge x \geq 1$, respectively. This recursion either eventually finds an integer solution, excludes the whole search space, or continues indefinitely.

The exponential growth due to the recursive calls can be a problem for which we essentially have two approaches. Instead of issuing recursive calls, we can *lift* the lemma $x \leq 0 \vee x \geq 1$ to the SAT solver and let the SAT solver decide, providing the possibility to retain knowledge from different parts of the search space. Alternatively, there are other ways to create cuts that do not (necessarily) induce case splits like Gomory cuts or cutting planes. However, these other cuts usually have other issues concerning termination.

For nonlinear integer problems, however, a completely different approach is commonly used. Most solvers employ *bit blasting* in the spirit of [FGM+07] in some fashion, essentially using a bit-precise encoding of the integer variables – up to a certain number of bits – into propositional logic. This approach is conceptually simple and impressively efficient if a *small* satisfying assignment exists (in terms of the bits necessary to represent it), but also has significant drawbacks: finding *large* models requires a large number of bits which entails a superlinear growth of the propositional formula; determining unsatisfiability is all but trivial and requires additional machinery, if possible at all.

We have presented an integration of nonlinear decision procedures (including CAD) into branch and bound in [KCÁ16] and showed that it nicely complements *bit blasting*. Most prominently, algebraic procedures can find unsatisfiability which is only rarely possible

Solver	SAT		UNSAT		overall	
NIA-B&B	1044	1.63 s	518	4.48 s	1562	6.5 %
NIA-Blast	4367	13.11 s	26	0.68 s	4393	18.4 %
NIA-Full	4490	13.12 s	508	7.68 s	4998	20.9 %

Table 6.8: Experimental results for integer problems.

using bit blasting. Furthermore, problem instances whose satisfying assignments are *large* but have a comparably simple algebraic structure can significantly benefit from algebraic procedures. While our variant lifts *splits* (or *branches*) to the SAT solver, it might be an interesting direction for future research to look for possibilities more similar to cutting planes to avoid case splits.

Some experimental results – on the SMT-LIB `QF_NIA` benchmark set – are shown in Table 6.8. Though the branch-and-bound approach on its own is inferior to bit blasting, it complements bit blasting well. In particular, it allows solving a significant number of unsatisfiable problem instances but also helps in case of satisfiability. For a more thorough analysis that also discusses the other theory methods involved – both `NIA-B&B` and `NIA-Full` also employ simplex and virtual substitution equipped with branch and bound – we refer to [KCÁ16]. Please note that the results in [KCÁ16] were obtained on the set of SMT-LIB `QF_NIA` benchmarks of the day which have significantly changed since.

6.9 Quantifier elimination

The CAD proof system we presented is concerned with determining the satisfiability of a given problem, as is the overwhelming part of this whole thesis. However, the informed reader may remark that CAD was originally devised to solve a more general problem, namely *quantifier elimination*, and ask whether the presented techniques can be employed for tackling such a quantifier elimination problem.

Definition 6.6: Quantifier elimination

Let $\varphi = Q\overline{x}.\ Qy.\ \varphi'$ be a formula where y is quantified (existentially or universally). The *(single) quantifier elimination problem* is to construct a new *quantifier-free* formula ψ such that $Q\overline{x}.\ \psi$ is equivalent to φ.
The *general quantifier elimination problem* is to eliminate some or all quantifiers from an arbitrary formula in prenex normal form.

Most importantly, quantifier elimination does not search for a single satisfying sample point but indeed needs a decomposition – unless all quantifiers are existential quantifiers. Therefore, most presented ideas that aim for early termination and avoiding projection steps do not work here as a full CAD is needed anyway. It is nevertheless valuable to know that an implementation of CAD implementing all these techniques can still be used for quantifier elimination, and we do not need to duplicate multiple versions of the projection and lifting mechanisms.

Our solver `SMT-RAT` was enhanced in [Neu18a] to support quantifier elimination based on the incremental CAD computations. The implemented approach is mostly based

on the descriptions from [Bro99]. Though we can not benefit from most features at the core of `SMT-RAT`, the quantifier elimination part performs reasonably well when compared to other tools like `QEPCAD B` or Maple.

Furthermore, we see striking similarities between the inner workings and practical problems of this form of quantifier elimination and local cell constructions like One-Cell CAD from [Bro13], non-uniform CAD [Bro15] (that we both already mentioned in Section 1.1.4), or `MCSAT`-style explanations as described in Section 8.3.

We mostly think of the process of extracting a particular cell from the CAD and either finding a proper formalism to represent it – like extended polynomial constraints as defined in Section 2.3.1 or "extended Tarski formulae" with *indexed root expressions* from [Bro99] – or converting it to an expression in our standard first-order language – called "simple solution formulae" in [Bro99]. We thus think that future work on this connection could provide additional synergies.

6.10 Optimization

Another extension that gained substantial popularity in the SMT community recently is the support for optimization queries, for example, in [BPF15; ST15b]. Instead of looking for *some* feasible solution to a first-order logic formula, the goal is to find an *optimal* solution with respect to a given objective function. This task is oftentimes called *optimization modulo theories* and expands the applicability of SMT (or rather OMT) significantly.

The most popular approach extends both the SAT solver and the theory solver as presented in [ST15a]. In particular, it requires the theory solver to compute an optimal solution with respect to the given set of constraints. While the simplex method – being a method meant for optimization tasks in the first place – provides for this naturally, optimization within this framework for nonlinear problems has not found a lot of traction yet.

Though we never made time to implement the following approach, we propose to employ CAD for this task. It not only allows for a *complete* method that does not rely on convergence or special problem properties (like convexity) but also permits interesting ways to obtain *almost-optimal* solutions more quickly with strong guarantees on the quality of those.

Let us assume that our objective function is a single variable instead of an arbitrary function. We can simply reduce any objective function f by adding an equality $v = f$ to the set of constraints with v being a fresh variable that we use as a new objective. Let us furthermore assume, without loss of generality, that we always *minimize*.

The fundamental idea is very simple. We first impose a variable ordering where v is the last variable in the projection, and thus the *first to be lifted*, and compute a *full projection*. We then split the lifting into two parts: for v we select the *smallest* cell (we have not tried yet) and afterward perform the regular lifting procedure above this selection of v. As soon as we find a solution, it is optimal in the sense that it comes from the *optimal cell* for v.

If we are only looking for *a very good* solution we can simply select a sample point that is very close to the lower bound of the cell and give extremely strong guarantees on the value of the objective. For s a satisfying sample point with $s_v \in (\alpha_1, \alpha_2)$ we can

give an *absolute* bound on the quality of the solution, namely $s_v - \alpha_1$. We can even allow the user to provide an acceptable error and simply select an appropriate s_v, all without the need for any approximation.

Of course, we can even give exact values if the optimal satisfying cell is a closed cell. In this case, the optimal value s_v is exactly at a root, and the regular CAD lifting provides us with an optimal assignment. Unboundedness can also be detected easily when the very first cell – that is unbounded – is satisfiable. With this, we claim to be on a par with other optimization techniques, at least in terms of expressivity.

We additionally would like to compute solutions for unbounded cells and exact optima from open cells. While we have not investigated this issue deeply, we propose to borrow terminology and machinery from *virtual substitution*, which allows computing with $\pm\infty$ as well as terms of the form $r \pm \varepsilon$. These are eliminated within the individual rules of virtual substitution, and we assume that similar techniques can be applied within a lifting step.

Part III

Model-Constructing Satisfiability Calculus

Proof system

The *lazy SMT solving* approach presented so far is guided by the idea to have a clear separation between the Boolean reasoning and the theory reasoning. It is possible to implement a Boolean SAT solver that can be used for lazy SMT solving completely irrespective of the theory we use it with. The theory reasoning, on the other hand, does not need to bother with clauses and logical connectives, but purely focuses on sets of theory constraints.

While this has been the predominant approach for two decades and this *separation of concerns* simplifies the implementation, it is by no means the only conceivable solution. *Eager SMT solving* is still viable for certain logics and has been used a lot in the past, even [DP60] can be seen as an eager approach to solve first-order logic problems.

In [JM12] and [MJ13], a new approach called *MCSAT* was presented that makes a certain part of the theory reasoning a *first-class citizen* in the – beforehand only Boolean – core reasoning engine. Similar to how a SAT solver builds a *Boolean model* for the formula over time, the *MCSAT* solver also builds a *theory model*. There is still a theory reasoning component that is decoupled from the core MCSAT solver, but it *only* needs to explain why a certain (partial) theory model is not feasible.

We first give a proper definition and introduction to the MCSAT proof system. Afterward, we discuss our main contributions concerning MCSAT, consisting of some adaptations to the proof system, both improved and completely novel methods for theory reasoning, how to implement it within an existing SMT solver, and how to select proper heuristics. The implementation is then experimentally evaluated. Finally, we compare the MCSAT proof system to other proof systems from a theoretic perspective, firstly using the notion of *proof complexity* in comparison to the *resolution proof system* and secondly using the notion of *polynomial simulation* in comparison to the CDCL(T) proof system.

7.1 Definition

MCSAT is given as a proof system in [MJ13] and we start with this presentation as well. Our work on MCSAT includes the integration of MCSAT into a regular SAT solver, and we give an algorithmic description later on.

As we later see, the common termination argument for the MCSAT proof system relies on the *finite basis property*. Essentially, we require that all new theory constraints come from some finite set of constraints. We call such a set a *finite basis* $\mathcal{B}$ and note that it usually depends not only on the input constraints but also on which theory methods are employed for the explanation function that we discuss in Section 8.3.

Similar to CDCL(T) in Section 4.5 and, in particular, Definition 4.5, MCSAT works on a *MCSAT state* as defined below. The proof rules look similar to the ones from Section 4.5 but have some notable differences. We start with a few special notations and then define the MCSAT proof system.

Definition 7.1: MCSAT state and trail

Let an *MCSAT trail* extend a DPLL trail with an additional type of elements:

$$x \mapsto \alpha_x \qquad \text{Theory assignment of } x \text{ to theory value } \alpha_x$$

Following Definition 2.6, we use the following notation for a trail M:

$$M = [\![N, L_1, C_1 \to L_2, \neg L_3, x \mapsto 2]\!]$$

where N is the prefix of M and itself a trail, L_1 and $\neg L_3$ are Boolean decisions, $C \to L_2$ is a Boolean propagation and $x \mapsto 2$ is a theory assignment. We call a trail *complete* if it contains assignments for all (Boolean and theory) variables. We call a trail *satisfying* if its model satisfies the formula, usually implying that it is also *complete*. We call the combination of an MCSAT trail and a set of clauses $\mathcal{C}$ a *MCSAT state*. We distinguish two types of states, an *MCSAT search state* denoted as $\langle M, \mathcal{C} \rangle$ and a *MCSAT conflict state* denoted as $\langle M, \mathcal{C} \rangle \Vdash C$ where C is a clause such that $M \models \neg C$.

In regular CDCL (and CDCL(T)), a literal's value is always determined by a *Boolean* assignment – either a decision or propagation – unless it is still unassigned. As MCSAT incorporates a theory model, a literal can now also have a value due to the theory model if it evaluates to *true* or *false* over this theory model. This carries the risk of having *conflicting* assignments: having a literal that is propagated to be *true* but evaluating to *false* on the theory model. We impose a certain consistency on the trail to avoid such a case.

Definition 7.2: MCSAT evaluation of literals

Let M be a trail and L a literal. We call $\text{value}_{\mathbb{B}}(L, M)$ and $\text{value}_T(L, M)$ the *Boolean value of L (under M)* and the *theory value of L (under M)*, respectively, and define them accordingly.

$$\text{value}_{\mathbb{B}}(L, M) = \begin{cases} \mathit{true} & \text{if } L \in M \\ \mathit{false} & \text{if } \neg L \in M \\ \mathit{undef} & \text{otherwise} \end{cases}$$

$$\text{value}_T(L, M) = \begin{cases} \mathit{true} & \text{if } v[M](L) = \mathit{true} \\ \mathit{false} & \text{if } v[M](L) = \mathit{false} \\ \mathit{undef} & \text{otherwise} \end{cases}$$

We call a trail *consistent* if it represents a proper partial assignment and the Boolean value and the theory value are compatible:

$$\forall L \in M.\ (\neg L \notin M \wedge \text{value}_T(L, M) \neq \mathit{false})$$

This allows us to soundly define the *value of* L $\text{value}(L, M)$:

$$\text{value}(L, M) = \begin{cases} \text{value}_{\mathbb{B}}(L, M) & \text{if } \text{value}_{\mathbb{B}}(L, M) \neq \mathit{undef} \\ \text{value}_{T}(L, M) & \text{otherwise} \end{cases}$$

We extend this function to obtain the *value of a clause* C and analogously the *the value of a set of clauses* $\mathcal{C}$ as follows:

$$\text{value}(C, M) = \bigvee_{l \in C} \text{value}(l, M) \qquad \text{value}(\mathcal{C}, M) = \bigwedge_{C \in \mathcal{C}} \text{value}(C, M)$$

Another interesting property that is checked at multiple places in the proof system we are about to define is *infeasibility*. Informally, we say that a trail is infeasible if it can not be extended to contain a *full model* (both Boolean and theory) in a consistent way.

Definition 7.3: `MCSAT` trail infeasibility

Let M be an `MCSAT` trail. We call M *feasible* if it can be extended to a trail $N = [\![M, \ldots]\!]$ such that N is *consistent* and *satisfying*.

We already note here that this definition has a *global view* on feasibility in that deciding upon the feasibility of a (possibly empty) trail is effectively identical to our original problem of deciding upon the satisfiability of a formula. Therefore, practical implementations usually perform a *partial check* for feasibility only, where *partial* refers to the "lookahead": like proposed in [JM12], we only check whether the next theory variable (according to some ordering) can be assigned properly. When we advance towards a *complete* trail this partial check eventually becomes a complete check. For a more detailed discussion of this, we refer to Section 9.3.

We now define the `MCSAT` proof system. Similar to [MJ13], we split the proof rules into multiple parts that we call *Boolean reasoning*, *conflict analysis*, and *theory reasoning*.

Definition 7.4: `MCSAT` – Boolean reasoning

The `MCSAT` proof rules for *Boolean reasoning* are defined as follows.

`Decide:`

$$\frac{\langle M, \mathcal{C} \rangle}{\langle [\![M, L]\!], \mathcal{C} \rangle} \quad \textbf{if} \quad L \in \mathcal{B} \text{ and } \text{value}(L, M) = \mathit{undef}$$

`Propagate:`

$$\frac{\langle M, \mathcal{C} \rangle}{\langle [\![M, C \to L]\!], \mathcal{C} \rangle} \quad \textbf{if} \quad \begin{array}{l} C = (L_1 \vee \cdots \vee L_n \vee L) \in \mathcal{C}, \\ \forall i.\ \text{value}(L_i, M) = \mathit{false}, \\ \text{value}(L, M) = \mathit{undef} \end{array}$$

`Conflict:`

$$\frac{\langle M, \mathcal{C} \rangle}{\langle M, \mathcal{C} \rangle \Vdash C} \quad \textbf{if} \quad C \in \mathcal{C},\ \text{value}(C) = \mathit{false}$$

`Sat:`

$$\frac{\langle M, \mathcal{C} \rangle}{\mathit{sat}} \quad \textbf{if} \quad \text{value}(\mathcal{C}, M) = \mathit{true}$$

Forget:

$$\frac{\langle M, \mathcal{C} \rangle}{\langle M, \mathcal{C} \setminus \{C\} \rangle} \quad \textbf{if} \quad C \in \mathcal{C} \text{ is a learned clause}$$

Restart:

$$\frac{\langle M, \mathcal{C} \rangle}{\langle [\![]\!], \mathcal{C} \rangle}$$

The part of the MCSAT proof system shown in Definition 7.4 takes care of *Boolean exploration* (Boolean decisions and propagations), detecting conflicts or satisfiability, and "maintenance tasks" such as removing learned clauses or restarting the solver. Note that the original presentation in [JM12] does not contain the Restart rule. Once a conflict has been detected we switch from the *search state* to the *conflict state*.

Definition 7.5: MCSAT – Conflict analysis

The MCSAT proof rules for *conflict analysis* are defined as follows.

Resolve:

$$\frac{\langle [\![M, D \to L]\!], \mathcal{C} \rangle \Vdash C}{\langle M, \mathcal{C} \rangle \Vdash R} \quad \textbf{if} \quad \neg L \in C, \; R = \text{Resolution}_L(C, D)$$

Consume$_1$:

$$\frac{\langle [\![M, D \to L]\!], \mathcal{C} \rangle \Vdash C}{\langle M, \mathcal{C} \rangle \Vdash C} \quad \textbf{if} \quad \neg L \notin C$$

Consume$_2$:

$$\frac{\langle [\![M, L]\!], \mathcal{C} \rangle \Vdash C}{\langle M, \mathcal{C} \rangle \Vdash C} \quad \textbf{if} \quad \neg L \notin C$$

Backjump:

$$\frac{\langle [\![M, N]\!], \mathcal{C} \rangle \Vdash C}{\langle [\![M, C \to L]\!], \mathcal{C} \rangle} \quad \textbf{if} \quad \begin{array}{l} C = (L_1 \vee \cdots \vee L_n \vee L), \\ \forall i.\ \text{value}(L_i, M) = \mathit{false}, \\ \text{value}(L, M) = \mathit{undef}, \\ N \text{ starts with a decision} \end{array}$$

Unsat:

$$\frac{\langle M, \mathcal{C} \rangle \Vdash \mathit{false}}{\mathit{unsat}}$$

Learn:

$$\frac{\langle M, \mathcal{C} \rangle \Vdash C}{\langle M, \mathcal{C} \cup \{C\} \rangle \Vdash C} \quad \textbf{if} \quad C \notin \mathcal{C}$$

Contrary to CDCL(T) where conflict resolution is essentially unspecified, we provide reasonably detailed rules for how to perform conflict resolution in Definition 7.5. However, these rules specify exactly what modern CDCL-style SAT solvers do: resolution-based backtracking until the first unique implication point with clause learning – though the clause learning is technically optional here.

Definition 7.6: MCSAT – Theory reasoning

The MCSAT proof rules for *theory reasoning* are defined as follows.

T-Propagate:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![M, E \to L]\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} L \in \mathcal{B}, \text{value}(L, M) = \mathit{undef}, \\ \texttt{infeasible}([\![M, \neg L]\!]), \\ E = \texttt{explain}([\![M, \neg L]\!]) \end{array}$$

T-Decide:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![M, x \mapsto \alpha_x]\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} x \text{ is a theory variable from } \mathcal{C}, \\ v[M](x) = \mathit{undef}, \\ \texttt{consistent}([\![M, x \mapsto \alpha_x]\!]) \end{array}$$

T-Conflict:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle M, \mathcal{C}\rangle \Vdash E} \quad \textbf{if} \quad \begin{array}{l} \texttt{infeasible}(M), \\ E = \texttt{explain}(M) \end{array}$$

T-Consume:

$$\frac{\langle [\![M, x \mapsto \alpha_x]\!], \mathcal{C}\rangle \Vdash C}{\langle M, \mathcal{C}\rangle \Vdash C} \quad \textbf{if} \quad \text{value}(C, M) = \mathit{false}$$

T-Backjump-Decide:

$$\frac{\langle [\![M, x \mapsto \alpha_x, N]\!], \mathcal{C}\rangle \Vdash C}{\langle [\![M, L]\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} C = (L_1 \vee \cdots \vee L_n \vee L), \\ \exists i.\ \text{value}(L_i, M) = \mathit{undef}, \\ \text{value}(L, M) = \mathit{undef} \end{array}$$

Definition 7.6 finally enhances this proof system with theory-specific reasoning. The T-Propagate rule enables us to inject new implications when the current theory model implies a theory constraint. T-Decide performs a theory decision and thereby extends the theory model. T-Conflict recognizes clauses that are conflicting *in the theory*, rather than on the Boolean level. Finally, T-Consume and T-Backjump-Decide extend the conflict resolution from Definition 7.5 to gracefully handle theory decisions.

Definition 7.7: MCSAT proof system

The *MCSAT proof system* consists of the proof rules for *Boolean reasoning*, *conflict analysis*,ysy and *theory reasoning* as defined above.

7.2 Intuition

The MCSAT proof system is rather similar to CDCL(T) in that it incorporates all the techniques for Boolean reasoning, but it adds theory reasoning into the core solving engine. We try to give some insight into the main ideas of MCSAT and how it may help in practical solving tasks.

Let us first recall how CDCL(T) deals with its theory. The CDCL(T) proof system almost exclusively deals with the Boolean reasoning and offloads all theory reasoning to a few very abstract proof rules. In practice, a CDCL(T) solver regularly calls out to a theory solver to check whether the current trail is consistent in the theory. In the case of inconsistency, the negation of a subset of the trail – an *infeasible subset* – is learned as a conflict clause and triggers the conflict analysis.

Any other form of theory learning through the generation of lemmas is pretty difficult, as the theory solver has no indication of what to generate. Literally, the task is *generate something we do not know yet*. Practical experience shows that there is only a fine line between not being able to construct any lemmas and flooding the Boolean reasoning with large amounts of irrelevant or essentially redundant lemmas.

`MCSAT` moves some part of the theory reasoning into the core solving component: it integrates the construction of a (partial) theory model. Now, the solver does not only work its way through the Boolean search space but at the same time explores the space of theory solutions. This allows the solver to guide the Boolean search using knowledge (or assumptions) about the theory fairly easily, hopefully avoiding dead ends that are easy to see in the theory. Furthermore, the partial model provides a hint to the theory solver *what lemmas to generate*.

7.3 Constructing theory assignments

The first novel component of `MCSAT`, compared to regular `CDCL(T)`-style SMT solving, is the creation of theory assignments. Note that we do not allow to add *any* theory assignment but require the resulting trail to be consistent – the theory assignment should at least be consistent with the theory literals already present in the trail. In general, we define an *assignment finder* as follows:

Definition 7.8: Assignment finder

Let M be an `MCSAT` trail and v a theory variable with domain D such that v is the smallest variable that is unassigned in M. We call a function

$$f(M, v) \to D \cup \mathscr{P}(M)$$

an *assignment finder* if the following holds: If $[\![M, v \to \alpha_v]\!]$ is consistent for some $\alpha_v \in D$, such an α_v is returned. Otherwise some $P \in \mathscr{P}(M)$ that is already infeasible over the partial model induced by M is returned.

We observe that *simply guessing* some value – for example by *random sampling* – may not be a good idea as we need to realize if no value exists. Instead, we need a way to ensure that we find a suitable value – if one exists – but can also detect if no suitable value exists with certainty. In this case, the assignment finder should not only indicate infeasibility but provide a *witness* P which is very similar to an infeasible subset.

Though we discuss the practical implementation of such an assignment finder in more detail later, we give a brief idea of the fundamental concept here, at least for a minimalistic assignment finder for an arithmetic theory.

Let us assume that we want to assign v – all *smaller* variables are already assigned and all *larger* variables are still unassigned. We observe that we only need to consider constraints that involve v and – possibly – *smaller* variables. We can then (at least conceptually) substitute the model for the smaller variables into these constraints and obtain a *univariate* problem.

Decomposing the possible values for v into finitely many regions through real root isolation is then essentially a lifting step in CAD as discussed in Section 5.3. Thus, many concepts we discussed in the context of CAD carry over as well, in particular, the notion

of *sign-invariant* regions and the idea that sample points are seen as *representatives* of such sign-invariant regions. In this sense, an assignment finder is looking for a satisfying region that can be used to construct a feasible assignment, or certifies infeasibility if no satisfying region exists.

7.4 Explanation functions and termination

MCSAT itself is agnostic of the theory being used and the most important component that performs theory reasoning is the explanation function. We define an explanation function, roughly following [MJ13], as follows.

Definition 7.9: MCSAT explanation function

Let M be an MCSAT trail that is *infeasible*. We call

$$E(M) \to \Phi_{\mathcal{T}}$$

an *explanation function* if for every infeasible trail M it generates a *valid theory lemma* that is *inconsistent* with M. We call E *incomplete* if it fails to generate a valid theory lemma under certain conditions. Unless stated otherwise, we assume all explanation functions to be *complete*.

Let us illustrate this definition using a few examples. What the explanation looks like mainly depends on the theory and why the trail is infeasible. The easiest case occurs if the trail is infeasible only due to theory literals from the trail as shown in Example 7.1. Here we essentially employ the same reasoning as in a regular lazy SMT-style theory solver: we certify infeasibility of a set of theory atoms using an infeasible subset of these theory atoms.

Example 7.1: Boolean conflict in a trail

Let $M = [\![y < 0, x^2 > 2, x \mapsto 3, y > 1]\!]$ and observe that M is consistent but infeasible. A suitable explanation is $(y \geq 0 \vee y \leq 1)$.
In general, if literals $L_1 \wedge \cdots \wedge L_k$ are infeasible with $L_1, \ldots, L_k \in M$ then $(\neg L_1 \vee \cdots \vee \neg L_k)$ is a suitable explanation.

Unfortunately, it may get more complicated if theory assignments need to be considered. Let us first review the effects of blindly applying the above technique to a slightly changed input problem in Example 7.2.

Example 7.2: Naive handling of theory conflicts

Let $M = [\![x + y < 0, x^2 > 2, x \mapsto 3, y > 1]\!]$ and observe that M is consistent but infeasible: we substitute $x \mapsto 3$ into $x + y < 0$ and obtain $y < -3$ which conflicts with $y > 1$. Generating a suitable explanation as before – where we use $x \neq 3$ as the negation of $x \mapsto 3$ – we obtain $(x + y \geq 0 \vee x \neq 3 \vee y \leq 1)$. We observe that we can satisfy the new explanation clause easily, for example, by $x \mapsto 3.1$. However, this not only leads us into the same conflict, it also indicates that we can do this infinitely often by slightly changing the assignment for x:

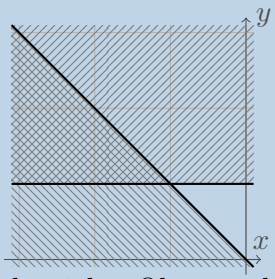
$$x \mapsto 3.1, x \mapsto 3.01, x \mapsto 3.001, \ldots$$

Toherefore, simply negating a theory assignment $x \mapsto 3$ by constructing a disequality $x \neq 3$ is insufficient, as it might pave the way to nontermination. Let us reconsider the conceptual idea of finding the assignment $x \mapsto 3$ in Section 7.3, in particular the idea that we identify a *satisfying sign-invariant region* and only select $x \mapsto 3$ as a representative from this region.

Following this intuition, the negation of the theory assignment $x \mapsto 3$ should cover a whole region of assignments (though not necessarily the same region that we constructed in the assignment finder). Let us rework the last example and try to find such a region we can exclude in Example 7.3.

Example 7.3: Regions for theory conflicts

As before, let $M = [\![x + y < 0, x^2 > 2, x \mapsto 3, y > 1]\!]$ and observe that M is still consistent but infeasible due to $x + y < 0$, $x \mapsto 3$ and $y > 1$. Let us now identify the underlying conflict here, independent of the concrete assignment of x. We rewrite $y > 1$ to $-y < -1$ and add it to $x + y < 0$ to obtain $x < -1$ which conflicts with $x \mapsto 3$.



We can also deduce this graphically from the plot on the right. Observe that the common solution space of the two inequalities $x + y < 0$ and $y > 1$ expands to the left starting at (but not including) $x = -1$, nicely coinciding with the constraint we obtained algebraically.
Following this line of argument, we can thus infer that $x < -1$ and generate a new explanation $(x + y \geq 0 \vee y \leq 1 \vee x < -1)$.

Excluding *some* region instead of a single assignment is not enough, as we could still come across an infinite sequence of such regions as we show in Example 7.4 and – for an arguably more plausible explanation function – Example 7.5.

Example 7.4: Regions of minimal size for theory conflicts

As before, let $M = [\![x + y < 0, x^2 > 2, x \mapsto 3, y > 1]\!]$ and observe that M is still consistent but infeasible due to $x + y < 0$, $x \mapsto 3$ and $y > 1$.
Let us assume the explanation function tries to avoid sequences like $x \mapsto 3.1$, $x \mapsto 3.01$, $x \mapsto 3.001$, … by excluding regions of a certain minimum size, say 1. For $x \mapsto 3$ we could generate the valid explanation $(x + y \geq 0 \vee y \leq 1 \vee x < 3 \vee x > 4)$. This however only shifts the problem a bit, allowing for a sequence like $x \mapsto 3$, $x \mapsto 5$, $x \mapsto 7$, … and, furthermore, may fail if the region we could exclude is smaller than the minimum size we impose.

While the very simple explanation function from Example 7.4 highlights the fundamental problem, one might argue that it is not only naive but also rather contrived to make the point. Therefore, we propose another explanation function that we hope looks somewhat more reasonable. We already observed that real roots separate the sign-invariant regions and our approach allows us to argue about univariate problems, and, thus, it might seem promising to employ the numerical go-to tool for such problems: Newton's method.

Example 7.5: Employing Newton iteration for theory conflicts

Let us consider $M = [\![y = x^2, y \leq 0, x \mapsto 10]\!]$. We observe that M is consistent but infeasible. It might seem reasonable to employ a classical method for such a problem like the Newton method.
We propose two possibilities to employ Newton-style reasoning. Let $f(x) = x^2$ and $x^* = 10$ and observe that we must cross a root of $f(x)$ in order to reach a satisfiable region due to $y \leq 0$.
We first consider how to exclude the half-open spaces towards $-\infty$ and ∞. If we can show that no root exists for any $x > x^*$ (or $x < x^*$) we can exclude $[x^*, \infty)$ (or $(-\infty, x^*]$) by reasoning that $f(x^*) \neq 0$ and $f'(x) \neq 0$ for all $x > x^*$ (or $x < x^*$). Here, we can generate the explanation $(y \neq x^2 \vee y > 0 \vee x < 10)$.
From x^* towards the roots of $f(x)$ we can employ the Newton iteration as follows. If we can show that we did not step over a root of $f(x)$ with a single Newton step, we can exclude the interval between the last and the current x value:

$$x' = x^* - \frac{f(x^*)}{f'(x^*)} = 10 - \frac{10^2}{2 \cdot 10} = 5$$

To certify that $f(x)$ has no root for $x \in [5, 10]$, we can employ interval arithmetic to see that $0 \notin f([5, 10]) = [25, 100]$, use the *intermediate value theorem* giving us $f(5) > 0$, $f(10) > 0$, $f'([5, 10]) > 0$, and thus $f(x) \neq 0$ for all $x \in [5, 10]$, or even algebraic tools like Sturm's theorem. Consequently, we can provide the explanation $(y \neq x^2 \vee y > 0 \vee x < 5 \vee 10 < x)$.
Subsequent Newton steps could however yield smaller and smaller intervals that converge towards $x = 0$, but never actually reach it. The sequence

$$\begin{array}{lll}
x \mapsto 4 & x' = 4 - \dfrac{f(4)}{f'(4)} = 4 - \dfrac{16}{8} = 2 & 0 \notin f([2, 4]) \\
x \mapsto 1 & x' = 1 - \dfrac{f(1)}{f'(1)} = 1 - \dfrac{1}{2} = 0.5 & 0 \notin f([0.5, 1]) \\
x \mapsto 0.25 & x' = 0.25 - \dfrac{f(0.25)}{f'(0.25)} = 0.25 - \dfrac{0.0625}{0.5} = 0.125 & 0 \notin f([0.125, 0.25])
\end{array}$$

excludes the intervals $[2, 4]$, $[0.5, 1]$, $[0.125, 0.25]$ and may continue indefinitely, but never actually reach zero.

Considering the previous examples of unsatisfactory explanation functions, we now give a criterion that ensures termination and is met by most of the explanation function that we present afterward. It comes back to one of the first concepts introduced in Section 7.1, the *finite basis*. In all of the above examples, we failed to construct explanations from such a finite basis but instead allowed the explanation function to construct infinitely many different theory atoms for a single input formula.

Definition 7.10: Finite-basis property

Let E be an explanation function and φ some input formula. We say that E *satisfies the finite-basis property* if for every φ all new theory atoms that E may generate for this φ come from a finite basis $\mathcal{B}$.

The intuition is essentially that an explanation function may not use the (partial) model to *construct* the explanation, but only to *guide* which explanation (that is constructed from the formula) should be returned. Theories over finite domains are an exception here, as only finitely many models exist anyway and, thus, it is sufficient to exclude models individually – though we probably could achieve significant improvements by excluding whole regions.

Note that an explanation function that satisfies the finite-basis property ensures termination of `MCSAT`, formalized in the following Theorem 7.1. We refer to [MJ13, Theorem 1] for a proof.

Theorem 7.1: Finite-basis property implies termination

Let E be a (complete) explanation function that satisfies the finite basis property. Then `MCSAT` equipped with E always terminates.

We feel that discussing what methods may be usable as an explanation function and which properties imply the finite-basis property can be rather insightful. Very abstractly, we can characterize the task of an explanation function as follows: given a problem in n variables and a model in $k < n$ variables that falsifies the input problem, provide a formula in these k variables that 1. follows from the given problem and 2. is still falsified by the model.

This closely resembles problems like *finding interpolants* or *quantifier elimination* in general, and, in fact, most explanation functions we present later are based on quantifier elimination procedures. The fundamental idea is to eliminate the variables not contained in the model and let the resulting formula describe a region *around the model* to generalize from a single theory assignment.

All quantifier elimination methods – at least all that we discuss in this context – share an interesting property: they heavily rely on a fixed variable ordering to iteratively eliminate these variables. Furthermore, the (partial) theory model is not used in the quantifier elimination process itself, but only *selects* the appropriate part of the resulting formula that is then used for the explanation. Thus, we can give a meta-argument for termination, based on this observation.

Theorem 7.2: Finite-basis property by quantifier elimination

Let E_x be a *quantifier elimination procedure* that eliminates x from a given formula and $\mathcal{A}$ a theory model. We assume that E_x returns a *finite* formula and only uses constraints of a higher theory dimension to create new constraints of a lower theory dimension. An explanation function based on E_x can be obtained as follows: 1. apply E_x for every $x \notin \mathcal{A}$, 2. convert the resulting formula to conjunctive normal form, and 3. select a clause C such that $\mathcal{A} \not\models C$.
Such an explanation function satisfies the finite basis property.

Proof. We observe that every application of E_x only creates new constraints that no longer contain x and are thus on a *lower theory level*. Furthermore, the set of new constraints is *finite* – as E_x returns a finite formula – and E_x works independently of $\mathcal{A}$ and only constraints of a higher decision level contribute to new constraints of a lower decision level.

We immediately derive that E_x never constructs new constraints on the *highest theory level*. Thus we can obtain all possible constraints on the second-highest decision level by applying E_x on all (finitely many) possible combinations of the input constraints. Applying this argument iteratively gives all constraints on all lower theory levels. As the result of E_x is always finite, we obtain finitely many constraints that constitute our finite basis. □

Intuitively, we exploited that information *flows downwards* with respect to the theory levels, thus simply "letting any information flow down" allows us to compute all constraints that may ever come up. Also note that this argument is not necessarily specific to quantifier elimination procedures, but could be applied more generally to explanation functions that exhibit this kind of information flow.

This discussion immediately gives us some insight into when the finite-basis property may break: if constraints are, whatever reason, constructed using other constraints from a lower level. While this may just be how the method works, we can also end up in such a case if we (repeatedly) change the variable ordering.

7.5 Model-Refining Satisfiability Calculus

While many different factors contribute to the practical efficiency of modern solving technologies like SAT solvers, one important one is the ability to *pick up valuable information* when its *cheap to obtain* and then use it again afterward. In particular, Boolean constraint propagation, but also conflict resolution with clause learning, collects such information when it is easy to do and makes it explicit to the other components of the solver – either by adding it to the trail or crafting a new clause.

Of course, one would like to have similar options in `MCSAT` to learn new facts in the theory cheaply. While `T-Propagate` aims to do this, we found it hard to apply and computationally expensive in many cases. One major shortcoming in practice that we did not seem to be able to overcome was that any theory reasoning was restricted to the theory variables already assigned (and the next one due to be assigned). We thus propose to extend `MCSAT`, allowing not only to *construct* the theory model but also to *refine* the *not yet assigned* parts of the model.

Definition 7.11: Model-refinement satisfiability calculus

Let $CD_x \subseteq D_x$ be the *current domain* of a variable x, implying that for every satisfying assignment α we have that $\alpha_x \in CD_x$, and write $x \in CD_x$. We introduce a new type of trail elements of the form $E \to x \in CD_x$ where E is a subset of the preceding trail and add the following proof rule:

`Refine-Model:`

$$\frac{\langle M, \mathcal{C} \rangle}{\langle [\![M, E \to x \in CD'_x]\!], D, \mathcal{C} \rangle} \quad \textbf{if} \quad \begin{array}{l} E \subseteq M \text{ such that} \\ E \implies x \in CD'_x \subsetneq CD_x \in M \end{array}$$

We allow to use the information that $\overline{x} \in CD_{\overline{x}}$ throughout the remaining proof system, in particular when evaluating literals with not yet assigned variables in the value function or to detect conflicts if a current domain becomes empty.

This extension from Definition 7.11 that we call *model-refinement satisfiability calculus* shall maintain a *current domain* for every variable, usually an interval (for arithmetic variables). In regular `MCSAT`, this current domain is (∞, ∞) and becomes a point interval once we invoke `T-Decide` for this variable.

We note that the propagations added to the trail integrate seamlessly with conflict resolution, but may introduce problems with the issue of termination. In particular, we need to impose an equivalent of the finite-basis property on the implementation of the `Refine-Model` rule as well, even worse on the *combination* of the `Refine-Model` rule and the `explain` function.

An implementation of the `Refine-Model` rule could use interval constraint propagation (ICP) in the spirit of [BG06; MKC09; Sch13] on the current domains, refining them based on constraints and theory decisions from the trail. Apart from regular propagations, one could even implement *splits* as decisions of new literals in `Decide`. Note, however, that ICP-style refinements most probably do not satisfy such a finite-basis property.

We can finally exploit this information in multiple ways. If any current domain becomes empty, we immediately have a Boolean conflict consisting of the propagations that imply the empty set. Whenever we perform a theory decision, we can restrict the search for a feasible assignment to the current domain of the respective variable. Whenever we evaluate constraints – either to find conflicts, perform theory propagations, or check for semantic propagations – we can not only evaluate over the theory model but also over the current domains for not yet assigned variables.

One can very well argue that this *only integrates ICP into `MCSAT`* or that we *only make knowledge explicit* that is already there. While this may be true, both of this can be very valuable: after all, *making existing knowledge explicit* is the whole point of these kinds of solvers and the integration of ICP into `MCSAT` might give us the power to reason about not yet assigned variables as well, making our solver more robust against possibly bad variable orderings.

Furthermore, this technique is more general than iteratively restricting the model with ICP. It could, for example, 1. allow to decide meaningful new literals in `Decide` based on the current domain (other than splits), 2. integrate other propagation schemes beyond ICP, for example taking into account the Boolean structure of the problem, or 3. be applied to any other theory that is beyond ICP. Note that we have not implemented any of this yet, and thus any claims about the benefits and possible improvements are only (more or less informed) guesses.

7.6 Optimization

As already noted in Section 6.10, extending SMT approaches to support optimization tasks is both interesting in itself, but also greatly beneficial for practical applications. Having seen `MCSAT` as an alternative approach to SMT, we propose to extend `MCSAT` to optimization queries as well. As before, we assume optimization to mean minimization.

Similar to what we presented in Section 6.10, we propose a rather simple approach that we have not yet implemented, though. When using regular CAD, we could exploit *global knowledge* about the problem – in the form of a full projection – to immediately select an optimal assignment. We can not do that in `MCSAT`, but rather have to iteratively search for an optimal solution.

We keep the idea to reduce our objective function to a single variable and assign this objective variable *first*. Also note that the fundamental issue concerning unboundedness and optimal solutions from open cells remain. To perform optimization, we propose an adapted version of `MCSAT` as shown in Algorithm 7.1.

Algorithm 7.1: Optimization with `MCSAT`

```
1  Function OptMCSAT(φ, v)
2      while true do
3          Select optimal feasible region R for v with T-Decide
4          if no feasible region exists then
5              return UNSAT
6          else if R is bounded then
7              T-Decide v ↦ r (almost) optimal from R
8              if MCSAT finds satisfiability then
9                  return SAT and (almost) optimal assignment
10         else
11             repeat
12                 Backtrack until v is unassigned
13                 Let last be the last assignment to v
14                 T-Decide v ↦ r ∈ R with r = 2 · min{R̃, last, −1}
15                 if threshold reached then
16                     return UNBOUNDED
17             until MCSAT finds unsatisfiability
18         MCSAT Restart to empty the trail
```

The method shown in Algorithm 7.1 proceeds as follows. We first obtain the optimal feasible region for the objective variable with respect to the univariate constraints. If no such region exists we determine infeasibility in Line 5, otherwise, we have a region R that contains the optimal solution. If R is bounded we can use `T-Decide` to assign $v \mapsto r$ where r is the optimal – or "a good" – value and return SAT in case this assignment leads to a full model. Otherwise `MCSAT` automatically discards the region around r and we continue with selecting a new region R.

We need to make sure that the excluded regions for the objective variable – for example if `MCSAT` does not find satisfiability in Line 8 – are "visible" when we identify the currently optimal region. Otherwise, we may enter an infinite loop, selecting the same region over and over again. To ensure this, we need to employ a variable ordering that at least processes *univariate* literals as Boolean decisions *before* performing a theory decision on v. Though this may seem self-evident, it conflicts with some of the heuristics we present in Section 8.5.

The region R may, however, be unbounded as well. In this case, we enter a nested loop in Line 11 that assigns v to exponentially growing values r until we either find the assignment to be unsatisfiable – in this case we let `MCSAT` exclude a region around this r and try with a new optimal region in the hope that the new optimal region is bounded now – or reach a threshold and consider the problem to be unbounded. The exact value for r can be one of three variants: if we just started testing values we start with the upper bound of the region (the lower bound is $-\infty$); if we already did some

iterations, we continue with the last assigned values and only make it smaller; finally, we make sure that we start with a negative value so that we have a proper initial value and "making it smaller" works by simply multiplying it with a constant.

Note that the threshold may bound the size of r (possibly depending on the coefficients of the constraints) but could also consider the number of iterations or similar indicators. Finally, after we have processed a particular region R we restart `MCSAT` and start over with a new optimal region.

Given a fixed threshold for the case of unboundedness, this method terminates as it only explores finitely many regions R – following the argument for termination of regular `MCSAT` – and the inner loop only checks finitely many values as well.

Implementation

We defined MCSAT to be agnostic of the actual theory that is being used, given that appropriate subroutines for the theory reasoning are provided. Due to the scope of this thesis, and to ease the presentation, we now focus on the case of real arithmetic. Hence, all theory variables range over $\mathbb{R}$ and only subroutines for (linear or nonlinear) real arithmetic are discussed. Implementations for other theories exist, though, for example, for the theory of uninterpreted functions as described in [JBM13], for nonlinear integer arithmetic in [Jov17], for bit-vectors in [ZWR16; GJ17], or generic improvements for theory combination in [BGM+18].

We have already observed in the definition of MCSAT in Section 7.1 that one major part of MCSAT is the Boolean reasoning which essentially is a CDCL-style SAT solver. Over time, all competitive implementations of SAT solvers have accumulated many techniques that make their usage fast in practice that are not (directly) mentioned in this definition. Well-known examples of this are decision heuristics (see Section 3.6.5), restarts and clause removal (see Section 3.6.4), or the *two watched literal scheme* (see Section 3.6.3) – not to mention low-level implementation tricks to make all this (for example) cache efficient.

It seems both pointless and infeasible to replicate all this in the context of a novel MCSAT solver. Instead, we propose to enhance an existing CDCL-style SAT solver – or even a CDCL(T)-style SMT solver – to support MCSAT-style SMT solving. Thereby, we can simply inherit all the infrastructure and implementation for the SAT-solving part and "only" need to extend it appropriately.

We concede that we are giving up some flexibility in how we integrate the Boolean reasoning within the MCSAT proof system. Furthermore, we may inherit a certain amount of *technical debt* or simply design decisions that were appropriate for a SAT solver, but may not be for an MCSAT solver. However, we hope to profit from a mature implementation of the Boolean reasoning.

In the following, we discuss how we extended SMT-RAT, usually used as a CDCL(T)-style SMT solver as described in [CKJ+15], for the MCSAT framework. This discussion includes not only our current implementation but also future extensions and alternative design choices (that we rejected for certain reasons). Subsequently, we discuss our methods for finding assignments and generating explanations and, finally, present a selection of experimental results.

8.1 Extending CDCL to MCSAT

The SAT solver in SMT-RAT consists of the MiniSAT solver presented in [ES03], adapted for CDCL(T)-style SMT solving so that it incorporates most state-of-the-art techniques for SAT solving and supports working with theory constraints. It thus implements all rules from Definitions 7.4 and 7.5 and combines them in the usual way.

Algorithm 8.1: MiniSAT implementation

```
1  while true do
2      while ¬BCP() do
3          if CDCL-style conflict resolution fails then
4              return UNSAT
5      restart heuristically
6      if unassigned Boolean variable exists then
7          var := pick Boolean decision variable
8          perform Boolean decision on var
9      else
10         all variables are assigned, return SAT
```

We show how this solver works internally – abstracted and simplified – in Algorithm 8.1. It essentially corresponds to the MiniSAT method `search` that implements the search procedure and makes use of *Boolean constraint propagation* (BCP), heuristic restarts, heuristic variable decisions, and CDCL(T)-style conflict resolution. Some technical details have been removed, for example, what MiniSAT calls *assumptions* (clauses with only a single literal) or the occasional removal of rarely used learned clauses.

Compared to the MCSAT proof system from Definition 7.7, Algorithm 8.1 implements the proof rules from Definitions 7.4 and 7.5 as follows: Decide in Lines 7 and 8; Propagate and Conflict in Line 2; Sat in Line 10; Restart in Line 5; Resolve, Consume, Backjump, Unsat and Learn in Line 3. We now add the proof rules from Definition 7.6 to this algorithm to obtain Algorithm 8.2.

The first rule T-Propagate is very generic and can, in theory, be used to inject theory lemmas at any time. While one may very well think about other possibilities to do so, we only check whether a constraint can be propagated after it has been selected for a Boolean decision in Line 17 right now.

The variable selection in Line 9 has been extended to select either a Boolean or a theory variable. In the latter case we perform a theory decision as specified by T-Decide if a suitable assignment exists in Line 12 or recognise a conflict using T-Conflict in Line 14. The last two rules T-Consume and T-Backjump-Decide are integrated into the conflict analysis and are thus not explicitly shown here.

To take care of the proper integration of theory reasoning into the Boolean reasoning, we perform two additional checks that do not directly correspond to any of the proof rules. In Line 6 we check whether any constraint fully evaluates over the current theory model but is not yet assigned in the Boolean model. If so, we inject a *decision* for this constraint. Note that one could also aim to perform some kind of propagation here, however simply using decisions worked well enough in our setting.

Algorithm 8.2: MCSAT implementation

```
1  while true do
2      while ¬BCP() do
3          if CDCL-style conflict resolution fails then
4              return UNSAT
5      restart heuristically
6      Check for semantic propagations
7      Check for inconsistent trail (due to BCP)
8      if unassigned variable exists then
9          var := pick decision variable
10         if var is theory variable then
11             if feasible assignment for var exists then
12                 perform theory decision
13             else
14                 CDCL-style conflict resolution on explanation
15         else if var is Boolean variable then
16             if var or ¬var can be propagated (in the theory) then
17                 use explanation to propagate var or ¬var
18             else
19                 perform Boolean decision
20     else
21         All variables are assigned, return SAT
```

Additionally, we need to check in Line 7 whether the trail became inconsistent. This may be somewhat surprising at first glance, given that the Boolean assignment is always chosen to be compatible with the theory value and the theory decision always respects the Boolean assignment. How can the trail become inconsistent in the first place under these circumstances?

Note that the theory decision (usually) considers only constraints that are *univariate* over the current theory model, but the assignment might fully evaluate a "not-yet-univariate" constraint as well. Consider for example $x \cdot y \neq 0$ being assigned to *true*, and a theory decision setting $x \mapsto 0$ as no univariate constraint is available. Though we could leave this issue to the theory decision to detect, it has proven to be significantly easier to do this explicitly beforehand.

In contrast to the more theoretic definition in [MJ13], and also the description of an actual implementation in [JBM13], Algorithm 8.2 gives a more detailed description of how the proof rules interact and how they are scheduled.

8.2 Assignment finder

The first novel subroutine we need for `MCSAT` is for generating a new assignment in the `T-Decide` rule such that `consistent`$(\llbracket M, x \mapsto \alpha_x \rrbracket)$. We have given some intuition on how such an assignment finder could work in Section 7.3.

We first present a variant that employs real root isolation to explore all sign-invariant regions of univariate constraints similar to the CAD lifting process. Then we show an alternative based on a general SMT solving engine that allows us to explore multiple variables at once.

Note that we allow considering constraints that are not *syntactically* univariate here. While implicitly forbidden in [JM12] – as only syntactically univariate clauses can be *selected* – we allow to use them in line with [MJ13]. Though this implies some interesting corner cases that the assignment finder has to deal with, it also enables significant improvements as the following Example 8.1 shows. Of course, it only makes sense to enhance assignment finders if we also reason about these non-univariate literals in Boolean decisions and propagations – otherwise, non-univariate literals are never added to the trail.

Example 8.1: Non-univariate literals in assignment finding

Let $M = [\![x_1 \cdot x_n + x_2 > 0, x_1 \mapsto 0]\!]$ and assume that we try to apply `T-Decide` to find an assignment for x_2 next.
If we were to ignore the constraint – following [JM12] as it is not univariate – we might select $x_2 \mapsto 0$ and may further spend a lot of effort on theory variables $x_3 \dots x_{n-1}$ until we eventually realize that $x_2 \mapsto 0$ gives rise to a conflict that is only due to $x_1 \mapsto 0$ and $x_2 \mapsto 0$.
If we instead consider the constraint right away, we can make use of $x_2 > 0$ and avoid a lot of theory reasoning for a case we can already prove to be infeasible. This case may even be more obvious if the non-univariate constraint leads to a direct conflict as for $M = [\![x_1 \cdot x_n > 0, x_1 \mapsto 0]\!]$.
We can immediately enter conflict resolution if the constraint is added to the trail while [JM12] would wait for theory level n to even look at the constraint and realize its erroneous theory decision early on.

8.2.1 Assignment finding based on real root isolation

Considering the knowledge about CAD from the previous part of this work, we propose to find assignments as described in the following. Note that we assume this to be how other implementations work as well – for example [JM12; JBM13] – though they do not provide many details on this point.

The approach consists of the following three steps: 1. partially evaluate all theory literals from M over the partial model induced by M and select those whose result is a univariate constraint in the theory variable v that is to be assigned, 2. compute all real roots of these univariate constraints to obtain a sign-invariant decomposition of the real space for v, and 3. either identify a sign-invariant region that satisfies all constraints or realize that no such region exists and compute an infeasible subset, very similar to what is shown in Section 6.7.

The first two steps are essentially equivalent to a single lifting step in CAD and we, therefore, refer to Section 5.3 for a more detailed description. If the model contains real algebraic numbers, it may not be possible to directly substitute the model into a constraint – see the discussion in Section 2.5 – and the underlying method to isolate real roots should also be able to deal with the case that the polynomial turns out not to be univariate. We abstract from this detail to simplify the presentation.

If a satisfying region exists in the third step, we only need to return some value from this region. If it does not exist, though, we need to find an infeasible subset and we refer to Section 6.7 for more details on this issue. This yields the following Algorithm 8.3.

Algorithm 8.3: Find an assignment by real root isolation

```
1 Function FindAssignment_RRI(M, v)
2     α := model from M
3     C := {L ∈ M | L[α] is univariate in v}
4     R := roots(P, α) where P are all polynomials from C
5     S := sample points based on R
6     if satisfying s ∈ S then
7         return s
8     else
9         return cover of C for S
```

8.2.2 Generic SMT-based assignment finding

Let us take a step back and abstract a bit from what we defined to be an assignment finder. We try to find a new theory assignment for some variable that satisfies some set of constraints or gives a subset of these constraints that certifies that no such assignment exists. A more general formulation of this would be to *extend the model for a set of constraints or produce an infeasible subset* – which is almost what we called a *theory query* in the regular SMT framework.

Therefore, we propose to use an appropriate *SMT compliant theory solver* for this task, slightly enhanced by a wrapper that takes care of the partial model we already have. Note that if the constraints contain more than just a single unassigned variable we technically invest more work than necessary as we also obtain a model for all other unassigned variables. We might, however, be able to leverage this additional work as well. If we obtain a model, we can perform multiple theory decisions at once – though it is not clear whether this is beneficial – or cache the additional assignments and use them for the next theory decision if they are still consistent with the trail.

We may, however, also be able to identify a conflict earlier now as we essentially have a lookahead into further theory variables and thus can avoid some work, as the following Example 8.2 illustrates.

Example 8.2: Benefits of an assignment finder with lookahead

Let $M = [\![x_1 > 0, x_n > 0, x_1 + x_n < 0]\!]$ and assume we try to invoke `T-Decide` on x_1 next. A regular assignment finder without lookahead could select $x_1 \mapsto 1$ and continue from there, possibly with a lot of unnecessary reasoning about $x_2, \dots x_{n-1}$, until we eventually realize that we have a conflict due to this assignment. An assignment finder with lookahead, however, could try to find a full model for these three constraints and immediately identify a conflict.

We formulate the following Algorithm 8.4 as a thin wrapper around an arbitrary theory solver. We deliberately do not specify how the model α is combined with the theory literals C, as this depends on the exact model and the capabilities of the theory solver.

We usually try to substitute α into C if possible, however, we may leave the realm of our constraints – for example, due to real algebraic numbers in our model. In this case, we may need to inject new constraints that explicitly state equality to real algebraic numbers if the theory solver supports these constraints. We may also be forced to fail and fall back to the regular assignment finder $\texttt{FindAssignment}_{RRI}$.

Algorithm 8.4: Find an assignment by theory solving

```
1 Function FindAssignment_SMT(M, v)
2     α := model from M
3     C := theory literals from M
4     return call_theory(C ∧ α)
```

As already indicated, it could make sense to combine different methods for finding assignments. Different instantiations of $\texttt{FindAssignment}_{SMT}$ – with different theory solvers – could be used that fail if the model is not compatible with the respective theory solver and hand the task over to the next assignment finder, or ultimately $\texttt{FindAssignment}_{RRI}$. However, we could also employ a whole theory solving strategy, for example, as described in [CKJ+15], to instantiate $\texttt{FindAssignment}_{SMT}$.

8.3 Explanation functions

MCSAT itself is theory-agnostic but requires an explanation function as described in Section 7.4 as an important theory reasoning component. We now describe several different explanation methods (for arithmetic theories) that we have implemented. As already noted, some explanation functions are *incomplete* (for example, restricted to linear constraints) and we let such explanation functions return $\bot$ if they are unable to construct an explanation. Recall that we required an assignment finder to produce a set of conflicting constraints in case of infeasibility, and thus we assume any explanation function to work on a *minimal infeasible* set of constraints.

8.3.1 CAD-based explanations

The first explanation method, that was originally proposed in [JM12], is based on the idea to construct a single CAD cell around the current (partial) model. An example of such a single CAD cell is depicted in Example 8.3.

Example 8.3: A single CAD cell

Consider the CAD cell depicted in Figure 8.1. The first variable x_1 is bounded by constants $(0 < x_1 < 1)$ illustrated by the bold line at the bottom. In the second dimension, x_2 is bounded by polynomials in terms of x_1 $(x_1^3 - 2x_1^2 < x_2 < 2 - x_1^2)$ and the resulting two-dimensional CAD cell is shown by the blue area.
In the third dimension, x_3 is bounded by polynomials in x_1 and x_2 – here we have $0 < x_3 < 2 - 0.2x_1^2 - 0.2x_2^2$. The lower bound 0 is illustrated by the yellow plane while the red surface represents the upper bound. The full three-dimensional CAD cell is thus the space between the yellow and red surfaces.
Note how the projection of the higher-dimensional cells onto the lower dimensions are identical to the lower-dimensional cells. A possible model within this cell would be $(0.5, 0.25, 1)$, indicated by the black dot.

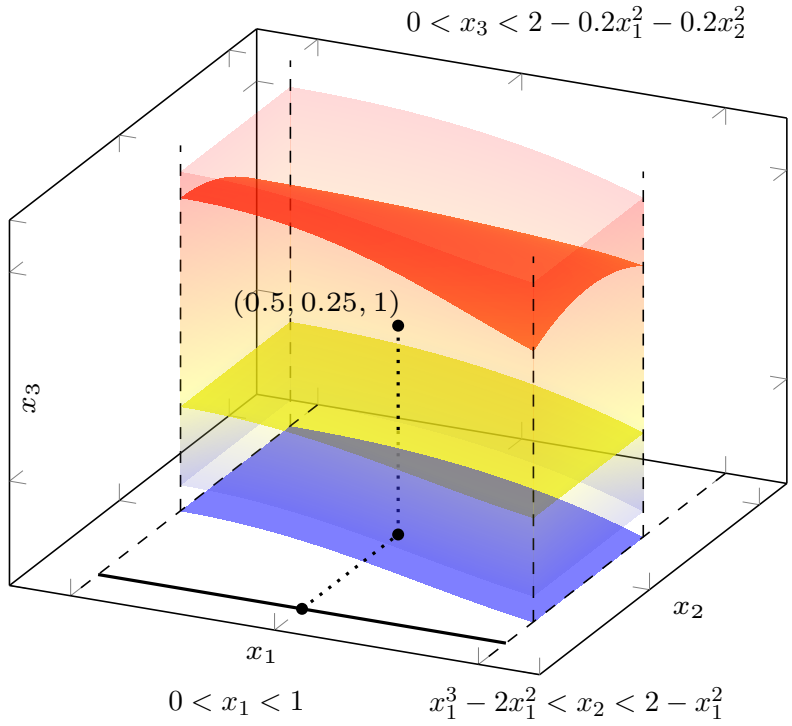


Figure 8.1: Example of a single CAD cell

To generate such a CAD cell, we use a somewhat reduced projection and then only explore the lifting around the partial model to obtain the borders of the cell that contains the partial model. An algorithmic description is given in Algorithm 8.5.

Algorithm 8.5: CAD-based explanation function

```
1  Function Explain_CAD(M)
2      α := model from M
3      L := literals that cause the infeasibility of M
4      P := model based projection of polynomials from L
5      C := true
6      for k = 1, 2, ... do
7          Z := real roots of P_k[α_1, ..., α_{k-1}]
8          if α_k ∈ Z then
9              C := (C ∧ x_k = α_k)
10         else
11             l, u := closest roots from Z below and above α_k
12             C := (C ∧ l < x_k ∧ x_k < u)
13     return L ⟹ ¬C
```

Note that the constraints constructed in Lines 9 and 12 are denoted as regular constraints, comparing x_k with the root of some polynomial. However, we actually need to construct *extended polynomial constraints* as defined in Section 2.3.1 from the respective roots, and we only use the above notation to make the algorithm more concise.

In [JM12], an adaptation of *Collins' projection operator* is presented that makes use of the partial model to avoid certain projection factors. We instead decided to employ *Lazard's projection operator* here (in combination with the modified lifting) and exploit the partial model in the spirit of [BK15], that is we only compute resultants that involve those polynomials that yield the closest bounds l and u.

Note that we initially used *McCallum's projection operator* instead, but occasionally observed incorrect explanations (excluding a region that was too large) witnessing the incompleteness of this projection operator – something we did not observe in the wild for regular SMT-style theory solving. Before switching to *Lazard's projection operator*, we used *Hong's projection operator* to obtain a sound explanation, which proved to be not significantly worse in terms of the overall solver performance.

This explanation function satisfies the finite-basis property following Theorem 7.2, as it only ever introduces new constraints whose polynomials are from the projection of the input constraints. If the variable order is never changed, all new constraints ever introduced stem from the (finite) full projection of all input polynomials.

8.3.2 OneCell explanations

Shortly after [JM12] motivated the need to efficiently construct individual CAD cells, another possibility we call *OneCell* was proposed. While the first version in [Bro13] only considered the case of *open cells*, all corner cases are covered in [BK15]. Though the fundamental idea of restricting the projection using the knowledge of the partial model is still similar to the previous approach, *OneCell* rather resembles a depth-first search traversing the projection polynomials. In many cases, OneCell needs to consider significantly less polynomials and thus yields larger cells.

We refrain from a more detailed explanation here and refer to [Bro13] for a somewhat gentle introduction and [BK15] for a full technical description of the method. Our own implementation – being the first complete implementation of [BK15] to the best of our knowledge – is described in [Neu18b].

We can use the same argument for termination as for the general CAD-based explanation function: as all new constraints are based on polynomials from the full projection of the input polynomials the finite-basis property holds, following Theorem 7.2.

8.3.3 Fourier–Motzkin variable elimination

Practical experience shows that we can identify a linear conflict in many cases, even if the overall input is nonlinear. In such cases, we might want to use a method suited for linear problems instead of calling a rather time-consuming method based on CAD. The somewhat obvious choice – that is also suggested in [MJ13] and used in [JBM13] – is the Fourier–Motzkin variable elimination. While we assume a fundamental knowledge of this method in general, we feel that it is work noting a few interesting issues.

Analogously to the argument for CAD-based explanation functions, the finite-basis property is once again satisfied due to Theorem 7.2 as all constraints stem from the set of constraints obtained by performing Fourier–Motzkin variable elimination, as long as the variable ordering remains unchanged.

Figure 8.2: Possible linear conflicts

8.3.3.1 Nonlinear constraints

The Fourier–Motzkin approach is not technically constrained to linear problems, but only to those where the variable that is to be eliminated only occurs linearly. This allows to extend it to an interesting subclass of nonlinear arithmetic where we can compute explanations without relying on more expensive methods like CAD or VS.

Considering two bounds $q_1 < p_1 \cdot x$ and $p_2 \cdot x < q_2$ we can eliminate x, resulting in $q_1 \cdot p_2 < q_2 \cdot p_1$. Additionally, we need to make sure that p_1 and p_2 do not change their sign by adding side conditions like $p_1 < 0$ and $p_2 > 0$. This comparably simple extension of Fourier–Motzkin variable elimination covers many cases where the variable that could not be assigned is only used linearly within nonlinear constraints.

Note that there are two possible improvements: firstly, generating an explanation with Fourier–Motzkin should be significantly faster compared to a VS-based or CAD-based method; secondly, the explanations due to Fourier–Motzkin tend to cover larger regions and thus possibly reduce the overall number of theory conflicts we need to process.

8.3.3.2 Disequality constraints

The Fourier–Motzkin variable elimination, in general, only works on sets of linear inequalities. While equalities can easily be converted to two weak inequalities, a disequality essentially introduces a case split and can thus not be dealt with. The previously suggested approach from [JBM13] introduces an explicit case split that is lifted to the Boolean reasoning engine. Though this solution looks nice and clean at first sight, it has a significant drawback: it does not *enforce* a Boolean decision but rather "offers" it to the SAT solver which may very well ignore it.

We instead propose another (a bit more technical) variant. Let us inspect the *covered region* of a constraint, that is the part of the real line that *conflicts* with this constraint. The (minimal) set of conflicting constraints has an interesting property: given that the variable we consider only occurs linearly, every inequality constraint covers a region that is bounded on one side but unbounded on the other. Equality constraints, on the other hand, exclude everything but a single point and disequality constraints exclude only a single point. Thus a conflict can only have one of two forms as Figure 8.2 illustrates: 1. two regions overlap or 2. two regions with strict bounds meet (possibly an equality) and the remaining point interval is excluded by a disequality.

If the two inequalities stem from an equality – we have $x \neq p$ and $x = q$ where p and q evaluate to the same value under the theory model – we propose to use the explanation $(x \neq p \wedge x = q) \implies p \neq q$. Otherwise – with $x \neq p$ and two weak inequalities $x \leq q_1$ and $x \geq q_2$ where p, q_1, and q_2 all evaluate to the same value under the theory model – we construct the explanation $(x \neq p \wedge x \leq q_1 \wedge x \geq q_2 \wedge q_1 = q_2) \implies q_1 \neq p$. In both cases, we push the conflict to a lower theory level, forcing the solver to backtrack the previous theory decision or the assignment of one of the constraints.

8.3.4 Virtual substitution

Another possibility that is in some sense between Fourier–Motzkin for linear constraints and CAD for nonlinear constraints is the virtual substitution. Virtual substitution is a quantifier elimination method for constraints of bounded degrees. The common virtual substitution is based on the existence of solution formulae and thus only works up to degree four [Abe26], though implementations tend to provide only support for up to degree two due to the complexity of the solution formulae of higher degree.

Virtual substitution can also be adapted to be used as an explanation function as shown in [ÁNK17] and [Nal17]. Our implementation fully supports constraints up to degree two and allows for (limited) cooperation with CAD-based explanation functions in that it supports root expressions of low degree. Once again, this approach satisfies the finite-basis property due to Theorem 7.2.

Note that variants of the virtual substitution exist that employ a symbolical representation of polynomial roots. Thereby, the need for an explicit solution formula is avoided and the method can be extended to arbitrary – but fixed – degrees. More on this topic can be found in [KS15] and [Kos16], though this is not used in our implementation.

8.3.5 Interval constraint propagation

We have already mentioned *interval constraint propagation* (ICP) as a generic method for solving constraint systems using domain propagation by interval arithmetic. Though it is not a quantifier elimination method, we can adapt it for an explanation function anyway and use it as an example of how to employ regular SMT-style theory solvers.

For this, we depart from the notion of *constructing new constraints* that describe a cell and rather *return* to the regular notion of SMT-style theory solving. We simply take all constraints from the trail and hope that they yield a conflict *independent of the theory model*. If this is the case, we argue that obtaining a conflict consisting solely of existing constraints is in general preferable to an explanation with new constraints.

Our reasoning is that introducing new literals grows the Boolean search space and should, therefore, be avoided, if possible. If we can instead resolve the current conflict by purely Boolean reasoning, we should do so. We acknowledge that this goes against the idea of `MCSAT` to some degree, but we rather see it as a supplementary solving technique, in some sense running `MCSAT` explanations and regular SMT-style theory solving in parallel.

We chose ICP for this experiment as we can easily integrate limited reasoning specific to the theory model in addition to regular SMT-style theory solving. In particular, we do not need to determine infeasibility but can also check easily whether the theory model has been excluded. Furthermore, we hope that its ability to reason across all variables, independent of any variable ordering, may be a valuable addition to the other explanation functions that are heavily oriented towards a common variable ordering.

For ICP, in particular, we collect all constraints from the trail and run standard propagations in the spirit of [BG06; Sch13] until we either *exclude the current theory model* or find *infeasibility* of the whole constraint set, roughly as shown in Algorithm 8.6.

The experimental results we present in the following show that interval constraint propagation is a valuable addition to the other explanation functions which are mostly based on algebraic methods.

Algorithm 8.6: ICP-based explanation function

```
1  Function Explain_ICP(M)
2      α := model from M
3      I_v := (−∞, ∞) for every variable v
4      Q := {(c, 1) for every constraint c}
5      while true do
6          if Q = ∅ then
7              return ⊥
8          if I_v = ∅ for some v then
9              return all c that contributed to I_v = ∅
10         if α_v ∉ I_v for some v then
11             Let e be a constraint that separates α_v from I_v
12             return e and all c that contributed to α_v ∉ I_v
13         if |I_v| < threshold for some v then
14             return ⊥
15         Remove some (c, priority) from Q
16         if priority ≥ threshold then
17             Use c to contract some I_v
18             Update priority accordingly
19             Insert (c, priority) into Q
```

Note that interval constraint propagation is notorious for its parameters and their intricate intra-dependencies, and we did not spend a lot of effort on tuning them for this purpose yet. In particular, we rely on thresholds for the size of the intervals and the priorities of the constraints for termination, but also need to properly update these priorities and select which variable should be updated by a specific constraint.

We attribute the improvements in practice mostly to the ability to argue independently of the variable ordering, considering not yet assigned variables as well. As we consider this a fundamental flaw – or rather a potential for improvement – in `MCSAT`, we propose an alternative, more fundamental, integration of interval constraint propagation into `MCSAT` in Section 7.5 that we think is even more promising.

8.3.6 Composition of explanation functions

Following the spirit of [CKJ+15], we aim to combine different solving techniques in the hope to employ the best method for every individual task. We propose two possible compositions of multiple explanation functions detailed in the following.

Definition 8.1: Sequential composition

Let $E_1, \ldots, E_k$ be explanation functions and allow all but E_k to fail under certain conditions. Let E be a new explanation function defined as follows:

$$E(M) := E_i(M) \text{ with } i = \min\{i \mid E_i(M) \text{ does not fail }\}$$

We call E the *sequential composition* of $E_1, \ldots, E_k$.

Sequential composition can be used for multiple explanations that have a clear priority. For example, we might want to try Fourier–Motzkin first, continue with virtual substitution if nonlinear constraints are present, and finally resort to a CAD-based explanation if high degrees are present.

Definition 8.2: Parallel composition

Let $E_1, \ldots, E_k$ be explanation functions and E a new explanation function defined as follows:

$$E(M) \coloneqq E_i(M) \text{ with } i = \arg\min_i \{\text{run time of } E_i(M)\}$$

We call E the *parallel composition* of $E_1, \ldots, E_k$.

Parallel composition aims to use the fastest result in a multi-threaded setting. We could, for example, run multiple versions of the Fourier–Motzkin explanation with different selections of constraints. Note that this composition can easily be changed to use the *simplest* explanation (instead of the fastest).

Under certain conditions, the composition of multiple explanation functions preserves the finite-basis property – given that the variable order remains unchanged. We formalize this in the following Theorem 8.1 based on Theorem 7.2.

Theorem 8.1: Finite-basis property of compositions

Let $E_1, \ldots, E_k$ be explanation functions that satisfy the finite-basis property for the reasons described in Theorem 7.2 with *equivalent variable orderings* and E their (sequential or parallel) composition. Then, E also satisfies the finite-basis property for a static variable order.

Proof. We resume the proof of Theorem 7.2 and change the iterative application of the explanation function as follows: instead of applying a single explanation function, we apply all explanation functions $E_1, \ldots E_k$ in parallel and afterwards conjoin their results. By the same reasoning as before, the combined explanation function satisfies the finite-basis property. □

8.4 Heuristics

As already discussed, the `MCSAT` proof system is rather a framework than an actual algorithm and we need to implement various heuristics. These include the overall scheduling of rules, the choice of an assignment finder, and an explanation function, but also topics we have not covered yet, in particular the variable ordering. Of course, a wide range of further heuristics is present, much like what we presented in Section 6.5.3, that we do not discuss here.

8.4.1 Variable ordering

In Line 9 of Algorithm 8.2 we pick a decision variable without any further specification other than that it can be either a Boolean or a theory variable and is not yet assigned. One possible implementation can be found in [JM12] with a static theory ordering

and an ordering on Boolean variables governed by whether the respective constraint is univariate under the partial theory model. Another possibility that is used for linear arithmetic only in [JBM13] treats theory variables and Boolean variables more uniformly and allows the (theory) variable ordering to change.

We analyzed those and more variants in [NKÁ19], showing that selecting a proper variable ordering – or rather a dynamic heuristic to select one – is both an important factor for practical performance and all but trivial. In fact, we find in [NKÁ19] – which contains a more detailed discussion of these issues than presented here – that our two best heuristics perform almost equally good, but for apparently different reasons as a hybrid between the two shows rather poor results.

While a dynamic ordering of theory variables seems to be beneficial, it also brings new problems for the `MCSAT` framework itself. As already discussed in Section 2.3.1, we might see extended polynomial constraints that can not be evaluated as the only remaining variable is not its left-hand side, but one from its root expression. Under a static variable ordering – like described in [JM12] – the left-hand side is always the largest among all involved variables, preventing this issue. Our approach to resolving this issue is to *disable* constraints from an *incompatible* variable ordering and reactivate them once the variable ordering is compatible again as described in [NKÁ19].

Furthermore, all arguments for the (refutational) completeness of `MCSAT` depend on a static theory variable ordering, which, in fact, is not just a technicality. Considering the following Example 8.4 we see that a dynamic ordering on the theory variables indeed has the potential to produce infinite loops.

Example 8.4: Incompleteness under dynamic theory ordering

Let $x_1 = 2 \wedge x_1 = 2x_2 \wedge x_2 = 2x_1$ be the input formula and assume we use an explanation function based on Fourier–Motzkin variable elimination (from Section 8.3.3) or virtual substitution (from Section 8.3.4). Consider the following sequence of explanations:

$$\begin{aligned}
[\![x_1 = 2x_2, x_1 = 2, x_2 \mapsto 2]\!] &\implies x_2 = 1 \\
[\![x_2 = 2x_1, x_2 = 1, x_1 \mapsto 2]\!] &\implies x_1 = 0.5 \\
[\![x_1 = 2x_2, x_1 = 0.5, x_2 \mapsto 2]\!] &\implies x_2 = 0.25 \\
[\![x_2 = 2x_1, x_2 = 0.25, x_1 \mapsto 2]\!] &\implies x_1 = 0.125
\end{aligned}$$

We trust the reader to realize that we can continue this sequence arbitrarily and that it is only made possible because we can switch arbitrarily between the variable ordering $x_1 < x_2$ and $x_2 < x_1$.

On a practical note, we neither observed such issues in practice nor do they seem particular likely. On the contrary, we rather expect a dynamic theory variable ordering to settle on some ordering that stays (almost) static instead of changing over and over again. To settle this issue in theory, we suggest to impose a restriction on when the variable ordering may change similar to when we may restart the whole solver. If we force the variable ordering to stay constant for growing periods of time, the solver is bound to eventually solve the problem within one such period, following the argument from Section 3.6.4.

8.5 Experimental results

We have identified three major heuristics for MCSAT: the assignment finder, the explanation function, and the variable ordering. Most of the issues discussed in Section 6.5.3 – factorizing polynomials, how to generate sample points, and many heuristics in various subroutines – apply to MCSAT as well, in particular to the different explanation functions. However, we focus on the aforementioned three issues now.

For these experiments, we use one base solver and analyze the effects of changing every heuristic individually. As this evidently fails to explore the whole range of possibilities, a more extensive analysis seems meaningful. Our base solver uses the assignment finder based on real root isolation, combines the explanations based on Fourier–Motzkin variable elimination, OneCell, and CAD and uses the *Theory first* variable ordering. It is called MCSAT-RRI, MCSAT-FMOCCAD, and MCSAT-Tf in the three comparisons.

8.5.1 Assignment finder

We have described two possible implementations for an assignment finder in Section 8.2. The first one – which we call simply *the assignment finder* and denote by MCSAT-RRI – considering only univariate constraints, uses real root isolation to identify a region that satisfies all constraints and samples some value from this region. The second one – called *SMT assignment finder* amd denoted by MCSAT-SMT – employs an arbitrary SMT strategy for theory calls.

We have made the experience that using a full-fledged strategy for the SMT assignment finder essentially bypasses MCSAT and thereby worsens performance, thus we propose to use an incomplete strategy. We are simply using a linear theory solver – based on the simplex method – which constructs an assignment based on the linear constraints, falling back to the regular assignment finder if this *linear model* does not satisfy the nonlinear constraints.

Solver	**SAT**		**UNSAT**		**overall**	
MCSAT-SMT	4606	0.43 s	4558	1.43 s	9164	79.8 %
MCSAT-RRI	4694	0.47 s	4696	1.35 s	9390	81.7 %

Table 8.1: Experimental results for different assignment finders.

We show a comparison of MCSAT solvers with the two different assignment finders in Table 8.1 and, as already indicated, MCSAT-SMT performs significantly worse than MCSAT-RRI on both satisfiable and unsatisfiable inputs – at least on the benchmark set we consider. Nevertheless, we conjecture that a more careful adaption of an SMT-based assignment finder, or any other scheme that employs a larger lookahead, could provide improvements in other scenarios.

8.5.2 Explanation functions

Apart from all the explanation functions, we have also discussed how to combine them. Given that this work is not concerned with concurrency (or parallelism) within an SMT solver, we ignore the possibility of parallel composition and only consider sequentially composed explanation backends.

Solver	SAT		UNSAT		overall	
MCSAT-CAD	4610	0.40 s	4597	1.15 s	9207	80.1 %
MCSAT-OCCAD	4647	0.44 s	4613	1.16 s	9260	80.6 %
MCSAT-FMOCCAD	4694	0.47 s	4696	1.35 s	9390	81.7 %
MCSAT-FMICPOCCAD	4693	0.52 s	4781	1.19 s	9474	82.5 %
MCSAT-FMVSOCCAD	4724	0.68 s	4819	1.47 s	9543	83.1 %
MCSAT-FMICPVSOCCAD	4723	0.67 s	4884	1.41 s	9607	83.6 %

Table 8.2: Experimental results for different explanation functions.

In our implementation, the CAD-based explanation function is the only *complete* method for nonlinear arithmetic. The OneCell explanation uses McCallum's projection, following [BK15], and fails in cases where correctness can not be ensured. We consider all other explanation functions to be *easy shortcuts* to the CAD-based explanations and thus all proposed explanation backends eventually fall back to them.

For this evaluation, we consider the following explanation backends: 1. CAD (`MCSAT-CAD`), 2. OneCell and CAD (`MCSAT-OCCAD`), 3. Fourier–Motzkin, OneCell and CAD (`MCSAT-FMOCCAD`), 4. Fourier–Motzkin, ICP, OneCell and CAD (`MCSAT-FMICPOCCAD`), 5. Fourier–Motzkin, VS, OneCell and CAD (`MCSAT-FMVSOCCAD`), and 6. Fourier–Motzkin, ICP, VS, OneCell and CAD (`MCSAT-FMICPVSOCCAD`).

A comparison of these explanation backends is given in Table 8.2, highlighting significant differences in practical performance. First and foremost, it appears to be extremely beneficial to combine multiple explanation backends as proposed earlier. We doubt that these major improvements are due to better performance, but rather think that "easier" explanation backends like the Fourier–Motzin-based one construct both *easier* but also *more powerful* explanations that exclude larger regions. One reason might be the fact that CAD-based explanations always construct fully-dimensional explanations while Fourier–Motzkin-based explanations only perform a single elimination step.

8.5.3 Variable orderings

As discussed in Section 8.4.1, the variable ordering of both Boolean and theory variables is of major importance and allows for many variants. We have analyzed the impact of different variable orderings in [NKÁ19] and mostly repeat those experiments.

The different orderings are mostly based on the following ingredients: strictly preferring Boolean or theory over the other; employing a strict ordering or an activity-based dynamic ordering in the spirit of VSIDS as discussed in Section 3.6.5; restricting the ordering to only consider *active* literals (or constraints) which are univariate over the current theory model. For some more aspects and a somewhat more detailed discussion, we refer to [NKÁ19]. We consider the following heuristics:

Random **(`MCSAT-Rnd`)** uses a random static ordering across all variables. This is only meant as a reference and naturally not intended for a serious solver.

Boolean first **(`MCSAT-Bf`)** strictly prefers Boolean variables using a dynamic ordering for Boolean variables and a static ordering for theory variables.

Theory first **(`MCSAT-Tf`)** strictly prefers theory variables using a static ordering for theory variables and a dynamic ordering for Boolean variables. Note that due

Solver	SAT		UNSAT		overall	
`MCSAT-Rnd`	4436	0.33 s	4579	0.67 s	9015	78.5 %
`MCSAT-Bf`	4467	0.29 s	4602	0.94 s	9069	78.9 %
`MCSAT-Univar`	4548	0.36 s	4630	0.96 s	9178	79.9 %
`MCSAT-Univar-active`	4576	0.46 s	4659	1.08 s	9235	80.4 %
`MCSAT-Uniform-Tf`	4639	0.76 s	4668	1.42 s	9307	81 %
`MCSAT-Tf-dynamic`	4653	1.04 s	4657	1.46 s	9310	81 %
`MCSAT-NLSAT`	4668	0.81 s	4671	1.49 s	9339	81.3 %
`MCSAT-Tf`	4694	0.47 s	4696	1.35 s	9390	81.7 %
`MCSAT-Uniform`	4664	1.03 s	4768	1.13 s	9432	82.1 %

Table 8.3: Experimental results for different variable orderings.

to semantic propagations only Boolean variables that do not represent theory constraints are decided.

Theory first dynamic **(`MCSAT-Tf-dynamic`)** strictly prefers theory variables like *Theory first*, but uses a dynamic ordering for theory variables.

Uniform **(`MCSAT-Uniform`)** uses a uniform dynamic ordering for all variables, increasing the variable activity for both Boolean and theory variables once for every conflict.

Uniform + Theory first **(`MCSAT-Uniform-Tf`)** follows *Uniform* by using a uniform dynamic ordering for all variables, but strictly prefers theory variables if two variables have the same activity.

Univariate **(`MCSAT-Univar`)** employs a static ordering for theory variables and a dynamic ordering for Boolean variables. Boolean variables are considered for a decision only if the respective theory constraint is univariate under the theory model and the next theory decision is performed when all eligible Boolean variables are assigned.

Univariate + active **(`MCSAT-Univar-active`)** uses the same strategy as *Univariate*, but considers only Boolean variables whose theory constraint is univariate and in addition occur in a *not yet satisfied clause*.

NLSAT **(`MCSAT-NLSAT`)** implements our understanding of the heuristic of [JM12]. Compared to *Univariate + active*, it additionally restricts the Boolean variables to those that occur in *univariate clauses*.

Experimental results for all strategies are given in Table 8.3 and are consistent with what we observed in [NKÁ19]. Note that the results in [NKÁ19] were obtained while preprocessing was enabled while we compare only the `MCSAT` solver itself.

We are still somewhat puzzled here as we obtain the best results by either strictly preferring theory variables in a fixed order (`MCSAT-Tf`) or using uniform variable activities (`MCSAT-Uniform`). All our attempts to find some middle ground and reconcile these strategies yield worse results as we can see at the examples of `MCSAT-Uniform-Tf` and `MCSAT-Tf-dynamic`.

We can think of two possible resolutions here: there could be multiple significantly different strategies that yield somewhat similar results; alternatively, we are looking at a single (yet unknown) strategy that is nontrivially simulated by both `MCSAT-Tf` and `MCSAT-Uniform` while the other strategies for some reason fail to do so.

Theoretical aspects

We have seen MCSAT as a proof system and provided some insights on how to implement it. Given that we employ very similar techniques in our MCSAT solver as for the regular SMT solver before, it is certainly interesting to compare these two variants. While we can do so by benchmarking, we may also want to leverage theoretic tools to assess them. In the following, we compare the proof systems of MCSAT (from Definition 7.7) and CDCL*(T) (from Definition 4.6).

The first part is based on the notion of *proof complexity*, roughly following [RKG18] – which relates CDCL*(T) and Res*(T) that we define in Definition 9.3 – and our own paper on this issue [KÁG19], relating MCSAT and Res*(T). The second part then directly compares CDCL*(T) and MCSAT, establishing that these two approaches are essentially equivalent. Both, however, rely on a pretty theoretical view, making rather strong assumptions about the proof systems that usually do not apply to actual implementations, and we discuss these issues as well. Throughout this work, we can draw some interesting conclusions about the nature of MCSAT, mostly furthering our intuition, that we present in a final section.

9.1 Proof complexity

When having different methods to solve the same problem – for example CDCL*(T) and MCSAT – one would like to have a solid mathematical way to compare them in a meaningful way. The most fundamental properties are *soundness* and *completeness*, accompanied by more pragmatic measures like *asymptotic complexity* of (average case or worst case) run time or memory consumption.

In the case of CDCL*(T) and MCSAT, we know that both of them are sound and complete, and assuming that the Boolean satisfiability problem indeed has exponential complexity ($P \neq NP$), these two proof systems have the same asymptotic complexity. Note that this asymptotic complexity is – at least in the case of nonlinear real arithmetic – dominated by the theory queries and thus the above claim only holds if both the number and complexity of theory queries are comparable among the two proof systems. Note that MCSAT allows for the introduction of new theory atoms, and we, therefore, consider CDCL*(T) instead of CDCL(T).

The notion of *proof complexity* provides another angle at this, essentially asking for the *asymptotic size* of the *smallest proof* for a given class of problems that an algorithm – formulated as a proof system – can construct. We observe that we defined both MCSAT and CDCL*(T) as *deductive proof systems* as presented in Section 2.4 and use these formulations here.

Given that we argue about the *smallest proof*, this approach assumes that all heuristic decisions are nondeterministic – given by some all-knowing oracle – transforming a proof system into a (nondeterministic) algorithm. Furthermore, at least in our scenario, it only deals with *unsatisfiable* instances since a *proof for satisfiability* is always short, simply given by a variable assignment or rather the construction thereof.

9.1.1 A primer on proof complexity

Using proof complexity as a measure for proof systems dates back at least as far as [Tse68] where it is used for the *resolution proof system* for propositional logic – though it is still called the *annihilation rule*. In contrast to Tseitin's work, however, who measured the size of the proof in terms of the *number of clauses*, we consider the *number of rule applications*. The following discussion is based on the definition of proof systems and proofs from Section 2.4.

Definition 9.1: Proof complexity

Let φ be an unsatisfiable formula and $\mathcal{P}_\varphi \subset \boldsymbol{\mathcal{P}}^\vdash$ the set of proofs for $\varphi \models \mathit{false}$ from a proof system $\boldsymbol{\mathcal{P}}$. We call $\mathcal{P} \in \mathcal{P}_\varphi$ with $|\mathcal{P}|$ minimal among $\mathcal{P}_\varphi$ the *shortest proof for* φ (in $\boldsymbol{\mathcal{P}}$) and $|\mathcal{P}|$ the *(proof) complexity of* φ, also denoted as $\boldsymbol{\mathcal{P}}(\varphi)$.

Note that while the (proof) complexity of φ is unique, the shortest proof is not. If we consider the resolution proof system for propositional logic, a proof can be seen as a sequential representation of a *resolution tree*. While there can be multiple different resolution trees that prove the same formula unsatisfiable, there can also be multiple *serializations* of one such tree which all result in proofs of the same length.

Though the above definition specifies the proof complexity as a natural number n, we usually use it as an asymptotic measure with respect to the *size* of formulae from a certain class of formulae. Thus, we usually talk about *polynomially sized* or *exponentially sized* proofs and (ab)use the *big O notation* we know from run-time analysis. We always assume the asymptotic measure to be relative to the size of the formula, usually in terms of the number of symbols used when writing it down.

There are usually two different types of statements about the proof complexity of some formula classes: 1. the formula class has a certain proof complexity within $\boldsymbol{\mathcal{P}}$ or 2. the formula class has a smaller proof complexity within $\boldsymbol{\mathcal{P}}_1$ than within $\boldsymbol{\mathcal{P}}_2$. While the first assertion provides some insight into a single proof system, the second is suited to compare the *power* of two different proof systems. We formally define the relation between two proof systems by introducing the notion of *simulation of proof systems*.

Definition 9.2: Simulation of proof systems

Let $\boldsymbol{\mathcal{P}}_1$ and $\boldsymbol{\mathcal{P}}_2$ be two proof systems and Φ a set of formulae. We say that $\boldsymbol{\mathcal{P}}_1$ *(polynomially) simulates* $\boldsymbol{\mathcal{P}}_2$ if for every $\varphi \in \Phi$ we have that $\boldsymbol{\mathcal{P}}_1(\varphi)$ is at most polynomially longer than $\boldsymbol{\mathcal{P}}_2(\varphi)$ and write $\boldsymbol{\mathcal{P}}_1 \succcurlyeq_\Phi \boldsymbol{\mathcal{P}}_2$. Conversely, if $\boldsymbol{\mathcal{P}}_1 \not\succcurlyeq_\Phi \boldsymbol{\mathcal{P}}_2$ we know that there is some subclass $\Phi' \subset \Phi$ such that $\boldsymbol{\mathcal{P}}_1(\varphi)$ is *superpolynomially longer* than $\boldsymbol{\mathcal{P}}_2(\varphi)$ for all $\varphi \in \Phi'$. We then write $\boldsymbol{\mathcal{P}}_2 \succ_{\Phi'} \boldsymbol{\mathcal{P}}_1$. If $\boldsymbol{\mathcal{P}}_1 \succcurlyeq_\Phi \boldsymbol{\mathcal{P}}_2$ and $\boldsymbol{\mathcal{P}}_2 \succcurlyeq_\Phi \boldsymbol{\mathcal{P}}_1$ we say that $\boldsymbol{\mathcal{P}}_1$ and $\boldsymbol{\mathcal{P}}_2$ are *bisimilar* and write $\boldsymbol{\mathcal{P}}_2 \approxeq_\Phi \boldsymbol{\mathcal{P}}_1$.
If Φ is the whole set of formulae from the respective logic, or if the set of formulae is clear from the context, we only write $\succcurlyeq$, $\succ$, and $\approxeq$.

Note that the relations $\succcurlyeq$ and $\succ$ indicate relations of the *(expressive) power* of proof systems which is reciprocal to the relation of the lengths of the respective proofs. Also note that the simulation relations do not induce a *total ordering* on proof systems in general (on any meaningful class of formulae Φ). There may very well be two proof systems $\mathcal{P}_1$, $\mathcal{P}_2$ such that $\mathcal{P}_1 \not\succcurlyeq \mathcal{P}_2$ and $\mathcal{P}_2 \not\succcurlyeq \mathcal{P}_1$. Of course, we usually have at least either $\mathcal{P}_1 \succcurlyeq_\Phi \mathcal{P}_2$ or $\mathcal{P}_2 \succcurlyeq_\Phi \mathcal{P}_1$ for any sufficiently small class Φ.

9.1.2 Proof complexity of Res*(T) and MCSAT

We now assess the proof complexity of MCSAT by comparing it to another proof system called Res*(T) which has already been studied in [RKG18], and in particular been related to CDCL*(T). Note that most of what is presented in this section originates from [KÁG19] with little to no changes.

We have already given the basic *resolution proof system* that only deals with propositional logic in Definition 3.3. To cope with first-order logic theories, this proof system is commonly enhanced to what we call the *Res(T) proof system*.

Definition 9.3: Res(T) and Res*(T) proof systems

Let φ be an input formula in conjunctive normal form. Assume two clauses from φ that share some theory atom y, though with opposite polarity. The Resolution rule looks as follows:

Resolution:

$$\frac{(x_1 \vee \cdots \vee x_i \vee y), (\neg y \vee z_1 \vee \cdots \vee z_j)}{(x_1 \vee \cdots \vee x_i \vee z_1 \vee \cdots \vee z_j)}$$

Additionally, we can derive tautologies (in the theory) using the Theory derivation rule or the Strong theory derivation rule:

Theory derivation:

$$\frac{\varphi}{\varphi \wedge C} \quad \textbf{if} \quad \begin{array}{l} T \models C, \\ l \in \varphi \text{ for all } l \in C \end{array}$$

Strong theory derivation:

$$\frac{\varphi}{\varphi \wedge C} \quad \textbf{if} \quad T \models C$$

The *Res(T) proof system* consists of the Resolution rule and the Theory derivation rule. The *Res*(T) proof system* consists of the Resolution rule and the Strong theory derivation rule.

Similar to how we extended CDCL(T) to CDCL*(T) in Definition 4.6, we can extend Res(T) to Res*(T), allowing the introduction of new theory atoms. We can observe that Res*(T) $\succcurlyeq$ Res(T) immediately, and [RKG18] even suggests that Res*(T) $\succ$ Res(T).

We know from [RKG18] that Res(T) $\approxeq$ CDCL(T) and as well Res*(T) $\approxeq$ CDCL*(T) – being called DPLL(T) and DPLL*(T) in spite of their own note that encourages to properly distinguish between DPLL and CDCL. Having observed certain similarities of CDCL*(T) and MCSAT, we now aim to compare MCSAT and Res*(T) to state the following theorem for MCSAT, similar to the one given in [RKG18] for CDCL*(T).

Theorem 9.1: Proof complexity of Res*(T) and MCSAT

The Res*(T) proof system and the MCSAT proof system are *bisimilar* with respect to their proof complexity on *first-order logic with any theory*.

We prove Theorem 9.1 in two steps: firstly, we show MCSAT $\succcurlyeq$ Res*(T) and, secondly, show that Res*(T) $\succcurlyeq$ MCSAT. In both cases, we show that we can simulate every rule individually, which actually proves a slightly stronger statement: we not only construct *some other proof* (with polynomial overhead) but essentially *the same proof*. This observation is the starting point of the subsequent comparison in Section 9.2.

Proof. To show that MCSAT $\succcurlyeq$ Res*(T), it suffices to show that MCSAT can simulate both the Resolution rule and the Strong theory derivation rule with only polynomial overhead. Note that both proof systems operate on the same input formula φ in conjunctive normal form, seen as a set of clauses $\mathcal{C}$.

Simulating the Resolution rule. Assuming that the set of clauses $\mathcal{C}$ contains both $(C \vee L)$ and $(D \vee \neg L)$, we need to add $(C \vee D)$ to $\mathcal{C}$.
Let us first handle a special case. Assume that we have some literal in C whose negation is contained in D. In this case, $(C \vee D) \equiv \mathit{true}$ and there is nothing to do. Similarly, if $(C \vee D) \in \mathcal{C}$ we do not need to learn the clause. From here on we assume that $(C \vee D) \not\equiv \mathit{true}$ and $(C \vee D) \notin \mathcal{C}$. Starting from an empty trail, the clause $(C \vee D)$ can be learned using the MCSAT proof rules as follows:

We start by applying the Decide rule for all literals $L_i \in C \cup D$. Note that all $L_i \in \mathcal{B}$ and all L_i are undefined in M as $M = [\![]\!]$ initially.

Decide:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![M, \neg L_i]\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} L_i \in \mathcal{B}, \\ \text{value}(L_i, M) = \mathit{undef} \end{array}$$

Now both C and D evaluate to *false* over the trail and we can apply the Propagate rule with $(C \vee L)$ to propagate L.

Propagate:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![M, (C \vee L) \to L]\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} \text{value}(C, M) = \mathit{false}, \\ \text{value}(L, M) = \mathit{undef} \end{array}$$

Due to $\text{value}(L, M) = \mathit{true}$, and thus $\text{value}(\neg L, M) = \mathit{false}$, the second clause $(D \vee \neg L)$ is conflicting and we apply the Conflict rule.

Conflict:

$$\frac{\langle [\![M, (C \vee L) \to L]\!], \mathcal{C}\rangle}{\langle [\![M, (C \vee L) \to L]\!], \mathcal{C}\rangle \Vdash (D \vee \neg L)} \quad \textbf{if} \quad \begin{array}{l} (D \vee \neg L) \in \mathcal{C}, \\ \text{value}(D \vee \neg L) = \mathit{false} \end{array}$$

We perform conflict resolution using the Resolve rule and obtain $(C \vee D)$.

Resolve:

$$\frac{\langle [\![M, (C \vee L) \to L]\!], \mathcal{C}\rangle \Vdash (D \vee \neg L)}{\langle M, \mathcal{C}\rangle \Vdash (C \vee D)} \quad \textbf{if} \quad \begin{array}{l} L \in (C \vee L), \\ R = \text{Resolution}_L(C \vee L, D \vee \neg L) \end{array}$$

To add this clause to the set of clauses $\mathcal{C}$, we use the Learn rule.

Learn:

$$\frac{\langle M, \mathcal{C} \rangle \Vdash (C \vee D)}{\langle M, \mathcal{C} \cup \{(C \vee D)\} \rangle \Vdash (C \vee D)} \quad \textbf{if} \quad (C \vee D) \notin \mathcal{C}$$

We have achieved our goal of adding $(C \vee D)$ to $\mathcal{C}$ and return to the initial state with the `Restart` rule.

Restart:

$$\frac{\langle M, \mathcal{C} \cup \{(C \vee D)\} \rangle \Vdash (C \vee D)}{\langle [\![]\!], \mathcal{C} \cup \{(C \vee D)\} \rangle}$$

We observe that this sequence of proof rules is polynomial in the size of the clause $(C \vee D)$ (we need $|C| + |D| + 5$ rule applications) and we return to the same initial state afterward. No theory reasoning was used in the `MCSAT` rule applications and we thus pay no *hidden costs* in the form of theory reasoning.

Simulating the `Strong theory derivation` rule. We need to create some arbitrary clause C which is valid in the theory T, that is $T \models C$. Similar to the previous simulation, we assume that $C \not\equiv \mathit{true}$ and $C \notin \mathcal{C}$ as there is nothing to do in these cases.
Our main hurdle is that `MCSAT` does not allow for learning arbitrary clauses but only the current conflict clause. Therefore, we have to construct an (artificial) conflict that yields the desired clause. We assume that our finite basis $\mathcal{B}$ includes all literals that ever occur in the (finite) `Res*(T)` proof. We can construct and learn an arbitrary (valid) clause C using the `MCSAT` proof rules as follows:

We use the `Decide` rule repeatedly to add $\neg L$ for every $L \in C$ to the trail. Note that the `Decide` rule allows to decide any literal from $\mathcal{B}$, independent of whether they appear in the input formula.

Decide:

$$\frac{\langle M, \mathcal{C} \rangle}{\langle [\![M, \neg L]\!], \mathcal{C} \rangle} \quad \textbf{if} \quad \begin{array}{l} L \in \mathcal{B}, \\ \text{value}(L, M) = \mathit{undef} \end{array}$$

As $T \models C$ but $\text{value}(C, M) = \mathit{false}$, we now have $\texttt{infeasible}(M)$ and can apply the `T-Conflict` rule with $E = C$. Note that $\texttt{infeasible}(M)$ might not be capable of detecting this conflict in practical implementations, and we discuss this issue later in Section 9.3.

T-Conflict:

$$\frac{\langle M, \mathcal{C} \rangle}{\langle M, \mathcal{C} \rangle \Vdash C} \quad \textbf{if} \quad \begin{array}{l} \texttt{infeasible}(M), \\ C = \texttt{explain}(M) \end{array}$$

Now we can learn the desired clause C using the `Learn` rule.

Learn:

$$\frac{\langle M, \mathcal{C} \rangle \Vdash C}{\langle M, \mathcal{C} \cup \{C\} \rangle \Vdash C} \quad \textbf{if} \quad C \notin \mathcal{C}$$

Finally, we return to the initial state with the `Restart` rule.

Restart:

$$\frac{\langle M, \mathcal{C} \cup \{C\}\rangle \Vdash C}{\langle [\![]\!], \mathcal{C} \cup \{C\}\rangle}$$

We observe that we need $|C| + 3$ proof rule applications to learn an arbitrary clause (that is neither *true* nor already present in $\mathcal{C}$) and return to the initial state. This concludes our proof that $\texttt{MCSAT} \succcurlyeq \texttt{Res}^*\texttt{(T)}$. □

We now continue with the second part of the proof and show $\texttt{Res}^*\texttt{(T)} \succcurlyeq \texttt{MCSAT}$ by simulating all (relevant) proof rules from MCSAT within $\texttt{Res}^*\texttt{(T)}$.

Proof. We observe that clauses can exist in three separate places within MCSAT: the set of clauses $\mathcal{C}$, the trail M, and the current conflict clause C. When simulating MCSAT with $\texttt{Res}^*\texttt{(T)}$, we make sure that the set of clauses that $\texttt{Res}^*\texttt{(T)}$ operates on includes the set of clauses $\mathcal{C}$, all clauses from M, and the conflict clause C. Thus, when MCSAT concludes unsatisfiability by inferring the empty clause, we can do the same within $\texttt{Res}^*\texttt{(T)}$.
Note that $\texttt{Res}^*\texttt{(T)}$ retains all clauses that it constructs as it is not designed to *forget* clauses while MCSAT may drop clauses occasionally with the Forget rule. Though removing clauses can bring advantages in practice, the number of additional clauses is linear in the number of rule applications – MCSAT needs at least one rule to construct every rule in the first place – and the practical overhead of additional clauses – for example, due to larger lookup tables – are polynomial at most if proper data structures are used.
To prove that $\texttt{Res}^*\texttt{(T)}$ simulates MCSAT, it thus suffices to show that all clauses that ever occur in the MCSAT derivation can also be constructed using the $\texttt{Res}^*\texttt{(T)}$ proof rules. We assume that, initially, both proof systems start with the same set of input clauses and identify the rules where the MCSAT rule system constructs new clauses: Resolve, T-Propagate and T-Conflict.
All other rules either do not manipulate any clauses at all (Decide, Sat, Consume, Unsat, T-Decide, T-Consume), move clauses between any of the three places (Propagate, Conflict, Backjump, Learn), or drop existing clauses (Forget, T-Backjump-Decide). All those can be ignored for the purpose of this proof, as we only aim to provide $\texttt{Res}^*\texttt{(T)}$ with the same clause set.

Simulating the Resolve rule. The Resolve rule takes the two clauses C (the current conflict clause) and D (from the trail M) and produces the resolvent with respect to some literal L. By construction, we know that $\texttt{Res}^*\texttt{(T)}$ has both C and D available and can thus apply its own Resolution rule to produce the same resolvent.

Simulating the T-Propagate and T-Conflict rules. Both the T-Propagate rule and the T-Conflict rule construct a new clause E by calling the MCSAT explain method. This method produces "a valid theory lemma" (as specified in [MJ13]), thus we have $T \models E$ and can obtain these clauses using the Strong theory derivation rule.
As we have simulated all MCSAT rules that can be used to construct new clauses,

we can now take any MCSAT proof and convert it into a Res*(T) proof by using the presented simulations for the Resolve, T-Propagate, and T-Conflict rule and removing all other proof steps. The simulations shown above only require a single proof step each in Res*(T), and thus the proof in Res*(T) is at most as long as the MCSAT proof. □

We have shown MCSAT $\succcurlyeq$ Res*(T) and Res*(T) $\succcurlyeq$ MCSAT, and thus Res*(T) $\approxeq$ MCSAT, concluding the proof for Theorem 9.1. Note that this is yet another indication that MCSAT is essentially equivalent to CDCL*(T), as they are both bisimilar to Res*(T) (in terms of proof complexity) for arbitrary first-order logics. As we already noted, this proof requires the infeasible method to be more powerful than it usually is in practical implementations, and we discuss this issue in Section 9.3.

9.2 Algorithmic equivalency to CDCL*(T)

Given that MCSAT and CDCL*(T) aim to solve the same problem and arguably share many algorithmic ideas, we now discuss how these two methods relate to each other. We have already shown that MCSAT is bisimilar to Res*(T) – and thus also to CDCL*(T) as shown in [RKG18] – in terms of *proof complexity*.

However, we have already noted that the relation between MCSAT and Res*(T) is even stronger. They not only construct *some proof* (with polynomial overhead) but a *logically equivalent proof* and we thus call them *algorithmically equivalent*. Of course, the proofs are not *identical* – after all, the proof rules of MCSAT and Res*(T) argue about different objects – but the core reasoning steps are equivalent. We concretize this fuzzy notion of *algorithmic equivalency* in Definition 9.4.

Definition 9.4: Algorithmic equivalency

Let $\boldsymbol{\mathcal{P}}_1$ and $\boldsymbol{\mathcal{P}}_2$ be two proof systems and $\backsimeq$ a relation on the states of $\boldsymbol{\mathcal{P}}_1$ and $\boldsymbol{\mathcal{P}}_2$ indicating that two states are *logically equivalent*. Let $\mathcal{P}_1 = \Phi_0 \vdash \cdots \vdash \Phi_k$ and $\mathcal{P}_2 = \Psi_0 \vdash \cdots \vdash \Psi_k$ be proofs in $\boldsymbol{\mathcal{P}}_1$ and $\boldsymbol{\mathcal{P}}_2$, respectively. We say that $\mathcal{P}_1$ and $\mathcal{P}_2$ are *equivalent* if $\Phi_i \backsimeq \Psi_i$ for all $i = 0, \dots k$.
We extend this notion of equivalence to proofs that differ in their length. Let $\mathcal{P}_1$ as before and

$$\mathcal{P}_2 = \Psi_0 \vdash \cdots \vdash \Psi_{j-1} \vdash \Psi_j^0 \vdash \cdots \vdash \Psi_j^m \vdash \Psi_{j+1} \vdash \cdots \vdash \Psi_k$$

such that $\Phi_i \backsimeq \Psi_i$ for all $i = 0, \dots j-1, j+1, \dots k$ and $\Phi_j \backsimeq \Psi_j^i$ for all $i = 0, \dots m$. If m is at most polynomially larger than k we consider $\mathcal{P}_1$ and $\mathcal{P}_2$ equivalent and thereby essentially allow $\mathcal{P}_1$ to *squash multiple proof steps (of $\mathcal{P}_2$) into one*. Let $\boldsymbol{\mathcal{P}}_2$ such that it allows for an equivalent proof for every proof in $\boldsymbol{\mathcal{P}}_1$. We say that $\boldsymbol{\mathcal{P}}_2$ *algorithmically simulates* $\boldsymbol{\mathcal{P}}_1$ and write $\boldsymbol{\mathcal{P}}_2 \succeq \boldsymbol{\mathcal{P}}_1$. If both $\boldsymbol{\mathcal{P}}_1 \succeq \boldsymbol{\mathcal{P}}_2$ and $\boldsymbol{\mathcal{P}}_2 \succeq \boldsymbol{\mathcal{P}}_1$, we call $\boldsymbol{\mathcal{P}}_1$ and $\boldsymbol{\mathcal{P}}_2$ *algorithmically equivalent* and write $\boldsymbol{\mathcal{P}}_1 \simeq \boldsymbol{\mathcal{P}}_2$.

Recalling the proof for the bisimilarity of Res*(T) and MCSAT, we observe that we provided simulations for all rules individually and thus essentially already showed *algorithmic equivalency*. Let us now, however, turn to the actual topic of interest, the relation of MCSAT and CDCL*(T). Given that we have MCSAT $\approxeq$ Res*(T) $\approxeq$ CDCL*(T) and MCSAT $\simeq$ Res*(T), it seems only reasonable to assume that, in fact, MCSAT $\simeq$ CDCL*(T).

Theorem 9.2: Algorithmic equivalence of CDCL*(T) and MCSAT

CDCL*(T) and MCSAT are *algorithmically equivalent*.

Our approach to prove Theorem 9.2 is as follows: we define the *equivalence relation* $\backsimeq$ on the states of both proof systems as an equivalence relation on the respective trails and then show how they can *algorithmically simulate* each other. For every rule in one of the two proof systems, we give a polynomial sequence of rule applications in the other proof system that ends in a state that is equivalent according to $\backsimeq$.

9.2.1 Equivalence of states

Recall the definitions of the trails that are used in CDCL*(T) and MCSAT, respectively.

Reminder 9.1: DPLL and MCSAT trails

A DPLL trail contains the following elements:

L	Boolean decision of literal L
$C \to L$	Boolean implication of literal L due to clause C

An MCSAT trail additionally contains the following elements:

$x \mapsto \alpha_x$	Theory assignment of x to theory value α_x

We claim that the theory model is *only* a heuristic way to guide the solver to meaningful rule applications – which in no way should diminish its importance in practice. Our equivalence, therefore, ignores all theory assignments from the MCSAT trail.

Definition 9.5: Equivalence of trails

Let M_{DPLL} be a DPLL trail and M_{MCSAT} an MCSAT trail. We define the *reduced MCSAT trail* $red(M_{\text{MCSAT}})$ as follows:

$$\begin{aligned} red([\![]\!]) &= [\![]\!] \\ red([\![M, L]\!]) &= [\![red(M), L]\!] \\ red([\![M, C \to L]\!]) &= [\![red(M), C \to L]\!] \\ red([\![M, x \mapsto \alpha_x]\!]) &= red(M) \end{aligned}$$

We call two trails *equivalent* if $M_{\text{DPLL}} = red(M_{\text{MCSAT}})$ and write $M_{\text{DPLL}} \backsimeq M_{\text{MCSAT}}$.

We now use this notion of *equivalence on trails* to induce *equivalence on states*. Observe that states of CDCL*(T) and MCSAT are almost identical (consisting of a trail and a set of clauses) with the exception of the MCSAT conflict state that also includes a conflict clause. As the whole process of conflict resolution is squashed anyway, the equivalence in the following Definition 9.6 ignores the conflict clause.

Definition 9.6: Equivalence of states

Let $\langle M_{\text{DPLL}}, \mathcal{C} \rangle$ be a CDCL*(T) state, $\langle M_{\text{MCSAT}}, \mathcal{C} \rangle$ an MCSAT search state, and $\langle M_{\text{MCSAT}}, \mathcal{C} \rangle \Vdash C$ an MCSAT conflict state. If $M_{\text{DPLL}} \backsimeq M_{\text{MCSAT}}$ we call the CDCL*(T) state *equivalent* to the MCSAT search state (or MCSAT conflict state) and write $\langle M_{\text{DPLL}}, \mathcal{C} \rangle \backsimeq \langle M_{\text{MCSAT}}, \mathcal{C} \rangle$ (or $\langle M_{\text{DPLL}}, \mathcal{C} \rangle \backsimeq \langle M_{\text{MCSAT}}, \mathcal{C} \rangle \Vdash C$).

9.2.2 MCSAT algorithmically simulates CDCL*(T)

We show that the MCSAT proof system algorithmically simulates the CDCL*(T) proof system. Similar to the proof in Section 9.1.2, we proceed by giving MCSAT rule applications for every CDCL*(T) rule that can be combined to transform any CDCL*(T) proof into an MCSAT proof following Definition 9.4.

Proof. We show that every rule from CDCL*(T) can be algorithmically simulated in MCSAT with only polynomial overhead. We do not use the T-Decide rule during the simulation of CDCL*(T) proof rules and thus never obtain MCSAT trails that contain theory assignments. This implies $\text{value}(L, M) = \text{value}_{\mathbb{B}}(L, M)$ and $\text{value}_{\mathbb{B}}(L, M) = \mathit{undef}$ if and only if $L, \neg L \notin M$.

CDCL*(T) Decide:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![M, L]\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} L \text{ or } \neg L \text{ occurs in } \mathcal{C}, \\ L \text{ is undefined in } M \end{array}$$

This rule is directly simulated by the MCSAT Decide rule after some reformulation on the condition. We observe that $\mathcal{B}$ contains all literals from $\mathcal{C}$ and L is undefined in M exactly if $\text{value}(L, M) = \mathit{undef}$.

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![M, L]\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} L \in \mathcal{B}, \\ \text{value}(L, M) = \mathit{undef} \end{array}$$

CDCL*(T) Fail:

$$\frac{\langle M, \mathcal{C} \cup \{C\}\rangle}{\mathit{FailState}} \quad \textbf{if} \quad \begin{array}{l} M \models \neg C, \\ M \text{ contains no decision literals} \end{array}$$

The Fail rule can be applied whenever a clause C evaluates to *false* and there is no decision in the trail. We eventually want to apply the MCSAT Unsat rule which, however, requires a conflict state and the conflict clause to be the empty clause. We start by applying the Conflict rule to enter conflict resolution.

Conflict:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle M, \mathcal{C}\rangle \Vdash C} \quad \textbf{if} \quad C \in \mathcal{C}, \text{value}(C) = \mathit{false}$$

Now we transform the current conflict clause to the empty clause. M contains no decision literals and thus only propagations, and, furthermore, for every $L \in C$ we have one propagation $\neg L$ in our trail, as we have $M \models \neg C$. Formally:

$$M = [\![L_1, \ldots, L_n]\!], C = (K_1 \vee \cdots \vee K_m), \forall K_i.\ \neg K_i \in \{L_1, \ldots, L_n\}$$

Therefore, we can apply either the Resolve rule – if the last propagation from the trail is a literal from C – or the Consume rule until the trail is empty. Note that the above statement – the negation of every element from C is in the trail – is invariant under these rules: the Resolve rule eliminates L from C and only adds literals that are *false* in M and thus have negations in M while the Consume rule only removes propagations from the trail that do not appear in C.

Consume:

$$\frac{\langle [\![M, D \to L]\!], \mathcal{C}\rangle \Vdash C}{\langle M, \mathcal{C}\rangle \Vdash C} \quad \textbf{if} \quad \neg L \notin C$$

`Resolve`:

$$\frac{\langle [\![M, D \to L]\!], \mathcal{C} \rangle \Vdash C}{\langle M, \mathcal{C} \rangle \Vdash R} \quad \textbf{if} \quad \begin{array}{l} \neg L \in C, \\ R = \text{Resolution}_L(C, D) \end{array}$$

As argued before, these rules are applicable until the trail is empty. Furthermore, we know that M contains the negation of every literal from C. As $M = [\![]\!]$, we know that $C = () \equiv \mathit{false}$ and we can, therefore, apply the `Unsat` rule.

`Unsat`:

$$\frac{\langle [\![]\!], \mathcal{C} \rangle \Vdash \mathit{false}}{\mathit{unsat}}$$

We note that we apply the `Consume` and `Resolve` rules as many times as we have propagations in our trail, that is at most n times for n variables in the input formula, and thus get a linear overhead.

`CDCL*(T) UnitPropagate`:

$$\frac{\langle M, \mathcal{C} \rangle}{\langle [\![M, C \to L]\!], \mathcal{C} \rangle} \quad \textbf{if} \quad \begin{array}{l} C = D \vee L \in \mathcal{C}, \\ M \models \neg D, \\ L \text{ is undefined in } M \end{array}$$

This rule is directly simulated by the `MCSAT Propagate` rule and the conditions are equivalent after reformulation.

`Propagate`:

$$\frac{\langle M, \mathcal{C} \rangle}{\langle [\![M, C \to L]\!], \mathcal{C} \rangle} \quad \textbf{if} \quad \begin{array}{l} C = (L_1 \vee \cdots \vee L_n \vee L) \in \mathcal{C}, \\ \forall i.\ \text{value}(L_i, M) = \mathit{false}, \\ \text{value}(L, M) = \mathit{undef} \end{array}$$

Note that the clause C that we use as the explanation (or reason) for the propagation is not necessarily unique, but the one used by `CDCL*(T)` is always part of $\mathcal{C}$ so that we can perform the same propagation.

`CDCL*(T) TheoryPropagate`:

$$\frac{\langle M, \mathcal{C} \rangle}{\langle [\![M, (D \vee L) \to L]\!], \mathcal{C} \rangle} \quad \textbf{if} \quad \begin{array}{l} M \models_T D \vee L,\ M \models \neg D, \\ L \text{ or } \neg L \text{ occurs in } \mathcal{C}, \\ L \text{ is undefined in } M \end{array}$$

We simulate this rule by the `MCSAT T-Propagate` rule which essentially serves the same purpose, though the details of how we show the propagation to be sound differ. As above, the `CDCL*(T)` conditions imply $L \in \mathcal{B}$ and $\text{value}(L, M) = \mathit{undef}$. Furthermore, we observe that

$$M \models_T L \Leftrightarrow M \not\models_T \neg L \Leftrightarrow M, \neg L \models_T \mathit{false} \Leftrightarrow \texttt{infeasible}([\![M, \neg L]\!])$$

and can thus apply `T-Propagate`.

`T-Propagate`:

$$\frac{\langle M, \mathcal{C} \rangle}{\langle [\![M, E \to L]\!], \mathcal{C} \rangle} \quad \textbf{if} \quad \begin{array}{l} L \in \mathcal{B}, \text{value}(L, M) = \mathit{undef}, \\ \texttt{infeasible}([\![M, \neg L]\!]), \\ E = \texttt{explain}([\![M, \neg L]\!]) \end{array}$$

We note that the original definition of the `TheoryPropagate` rule from [NOT06] seems to avoid the burden of actually coming up with a reason for the propagation. In theory, this does not matter as we consider the theory reasoning to have zero cost. In practice, however, this burden is only moved to the `CDCL*(T) T-Backjump` rule (that we show later) where `CDCL*(T)` needs to find *some clause* with this property – if it is needed for the backtracking. Note that the original presentation of `MCSAT` [JM12] proposes to essentially do the same by only calculating the explanations lazily when they are actually needed in the conflict analysis phase. Also note that `infeasible` is commonly implemented with a finite lookahead, checking whether the theory model can be extended by another single theory variable without rendering the trail inconsistent. The authors of [MJ13] state, however, that `infeasible`(M) is equivalent to the literals from M being unsatisfiable together with the partial model from M. While having a finite lookahead technically violates the above equivalence $M, \neg L \models_T \mathit{false} \Leftrightarrow$ `infeasible`$([\![M, \neg L]\!])$, it only *defers* the detection of theory conflicts, but never misses them. As soon as the last variable is to be assigned, `infeasible` is complete and detects all possible conflicts, and the `MCSAT` proof system is robust enough to deal with such conflicts that are found later than expected.
This practical issue may very well make our proofs longer – due to the reasoning we have to perform until we eventually detect the conflict in practice – and we discuss this issue in Section 9.3.

`CDCL*(T) T-Backjump`:

$$\frac{\langle [\![M, L, N]\!], \mathcal{C}\rangle}{\langle [\![M, (C' \vee L') \rightarrow L']\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} C \in \mathcal{C} \text{ with } [\![M, L, N]\!] \models \neg C, \\ \text{there is some clause } C' \vee L' \text{ such that:} \\ \quad \mathcal{C} \models_T C' \vee L' \text{ and } M \models \neg C', \\ \quad L' \text{ is undefined in } M, \\ \quad L' \text{ or } \neg L' \text{ occurs in } \mathcal{C} \text{ or in } [\![M, L, N]\!] \end{array}$$

This rule encapsulates the whole process of conflict analysis and backtracking in one rule whereas `MCSAT` details how conflict analysis works by giving a whole set of rules for this case. `CDCL*(T)`, on the other hand, does not specify how the conflict analysis should proceed and – in theory – allows for a completely different scheme for conflict analysis and resolution.
This leaves us in a tight spot: the straightforward simulation using the `MCSAT` conflict analysis rules will only be able to simulate resolution-based approaches, and while this (should) be able to simulate any other approach – due to the (refutational) completeness of resolution – it may fail to do so in polynomial time. We feel that this is, however, the meaningful simulation, as it simulates what `CDCL*(T)` solvers do in practice.
We can avoid this issue by going for a more general solution: we can eliminate the `T-Backjump` rule from the `CDCL*(T)` proof system and essentially replace it by the `T-Learn` and `UnitPropagate` rules. However, this departs from what we think happens in actual solvers in practice. We thus consider the straightforward variant by far more relevant and instructive, but propose the more general variant as well for theoretical completeness. Therefore, we provide both reductions in this order.

Let us assume now that the clause $C' \vee L'$ is obtained by means of resolution on the propagated literals in N and L is the first unique implication point, implying that N only contains propagations and decisions that are not relevant for the conflict. To employ the regular conflict analysis scheme, we first need to enter conflict resolution using the `Conflict` rule. Then, we apply the `Resolve` or `Consume` rules to construct the conflict clause C' and eventually use the `Backjump` rule to leave conflict resolution and perform propagation on the literal L'.

`Conflict`:

$$\frac{\langle [\![M, L, N]\!], \mathcal{C}\rangle}{\langle [\![M, L, N]\!], \mathcal{C}\rangle \Vdash C} \quad \textbf{if} \quad C \in \mathcal{C},\ \text{value}(C) = \mathit{false}$$

We already noted that N only contains propagations or decisions that are not relevant for the conflict. This is almost the same situation as when simulating the `CDCL*(T) Fail` rule and with essentially the same argument we can apply the `Consume` rule – in this case either for propagations or decisions – and the `Resolve` rule until $N = [\![]\!]$ to obtain some conflict clause $C' \vee L'$.

`Consume`:

$$\frac{\langle [\![M, D \to L]\!], \mathcal{C}\rangle \Vdash C}{\langle M, \mathcal{C}\rangle \Vdash C} \quad \textbf{if} \quad \neg L \notin C$$

`Consume`:

$$\frac{\langle [\![M, L]\!], \mathcal{C}\rangle \Vdash C}{\langle M, \mathcal{C}\rangle \Vdash C} \quad \textbf{if} \quad \neg L \notin C$$

`Resolve`:

$$\frac{\langle [\![M, D \to L]\!], \mathcal{C}\rangle \Vdash C}{\langle M, \mathcal{C}\rangle \Vdash R} \quad \textbf{if} \quad \begin{array}{l} \neg L \in C, \\ R = \text{Resolution}_L(C, D) \end{array}$$

Now, we can apply the `Backjump` rule, given that $N = [\![L]\!]$ starts with a decision and the other conditions hold as L is the first unique implication point.

`Backjump`:

$$\frac{\langle [\![M, N]\!], \mathcal{C}\rangle \Vdash C}{\langle [\![M, C \to L]\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} C = (L_1 \vee \cdots \vee L_n \vee L), \\ \forall i.\ \text{value}(L_i, M) = \mathit{false}, \\ \text{value}(L, M) = \mathit{undef}, \\ N \text{ starts with a decision} \end{array}$$

For the more general case, we observe that we can eliminate the `T-Backjump` rule from the `CDCL*(T)` proof system and simulate it by applying the `T-Learn` and `Restart` rules, restoring the trail M and finally the `UnitPropagate` rule. Note that this is not far off our intuition: We somehow learn a new clause, backtrack – by removing something from the trail – and use the learned fact for propagation. We observe that the new clause $C' \vee L'$ used by the `T-Backjump` rule only allows for literals from the current formula as $M \models \neg C'$ and $L' \in \mathit{atoms}(\mathcal{C})$.
Given the `MCSAT` equivalents for all the rules used here, we implicitly get a simulation for the `T-Backjump` rule. The number of rule applications is linear in the size of the trail and thus in the number of variables. Restoring the trail relies on the optimal scheduler to properly rebuild whatever was on the trail before applying the `Restart` rule, using the `Decide` and `UnitPropagate` rules.

CDCL*(T) T-Learn*:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle M, \mathcal{C} \cup \{C\}\rangle} \qquad \textbf{if} \;\; \mathcal{C} \models_T C$$

MCSAT does not allow for learning arbitrary clauses, instead it allows for learning the current conflict clause only. Our approach is, therefore, to restart the solver, construct an (artificial) conflict that yields the desired clause, learn it, and finally reconstruct the trail.

Restart:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![]\!], \mathcal{C}\rangle}$$

We use the Decide rule repeatedly to add $\neg L_i$ for all $L_i \in C$ to the trail. We then have infeasible(M) and apply the T-Conflict rule with $E = C$.

Decide:

$$\frac{\langle [\![]\!], \mathcal{C}\rangle}{\langle [\![\neg L_i, \ldots]\!], \mathcal{C}\rangle}$$

T-Conflict:

$$\frac{\langle [\![\neg L_i, \ldots]\!], \mathcal{C}\rangle}{\langle [\![\neg L_i, \ldots]\!], \mathcal{C}\rangle \Vdash C}$$

Now, we only need to apply the Learn rule to add the conflict clause to $\mathcal{C}$, restart again, and restore the trail.

By simulating the T-Learn* rule, we can also easily simulate the special case T-Learn, exactly as described before.

CDCL*(T) T-Forget:

$$\frac{\langle M, \mathcal{C} \cup C\rangle}{\langle M, \mathcal{C}\rangle} \qquad \textbf{if} \;\; \mathcal{C} \models_T C$$

We observe that this rule almost corresponds to the MCSAT Forget rule, except that MCSAT only allows to forget learned clauses. CDCL*(T) instead allows to forget any *redundant* clauses – in the sense that they may be recovered by the T-Learn rule – irrespective of whether they were part of the original clause set.
If the clause to forget happens to be a learned clause, we can simply apply the MCSAT Forget rule. Otherwise, we do nothing and keep the clause. We keep at most as many additional clauses as we had in the original formula and thus the overhead is polynomial. As all clauses are still entailed by the current clause set $\mathcal{C}$, keeping redundant clauses does not change the semantics of $\models$ or $\models_T$.

CDCL*(T) Restart:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![]\!], \mathcal{C}\rangle}$$

This rule is identical to the MCSAT Restart rule.

We have shown how to algorithmically simulate every individual CDCL*(T) proof rule within the MCSAT proof system and have thus shown that MCSAT algorithmically simulates CDCL*(T). We observe that we did not use any of the rules that deal with theory assignments – T-Decide, T-Consume and T-Backjump-Decide – which indicates that this component of MCSAT does not contribute to its (theoretical) power. We give some more thoughts to this in Section 9.3.4. □

9.2.3 CDCL*(T) algorithmically simulates MCSAT

We have seen in the previous section that MCSAT is at least as powerful as CDCL*(T) in the sense that it algorithmically simulates any CDCL*(T) proof rule. This leaves the possibility that MCSAT is a stronger proof system than CDCL*(T). We now show that they are in fact algorithmically equivalent, as CDCL*(T) algorithmically simulates every MCSAT proof rule.

Proof. We have seen that MCSAT does not rely on theory assignments to simulate CDCL*(T), and this observation motivates how we deal with MCSAT proof rules that use theory assignments: we simply ignore them and show that it is safe to do. A closer look at the MCSAT proof system reveals that theory assignments only restrict the applicability of certain rules, but do not contribute to new clauses in an explicit way.
Furthermore, we have already observed that the level of detail for the specification of the conflict analysis is very different. While MCSAT gives detailed rules on how to perform the conflict analysis, CDCL*(T) just contains a single rule which *does everything*. We can thus ignore all MCSAT steps that deal with conflict resolution and perform everything within a single application of the T-Backjump rule – or the Fail rule if we determine unsatisfiability – when MCSAT leaves the conflict resolution state. The only exception to this is the Learn rule which does not change the trail or the conflict clause, but the formula and we can trivially simulate it with the T-Learn* rule.
MCSAT may introduce new literals occasionally, in particular when explain is called, from the finite basis $\mathcal{B}$. Whenever a new literal $L \notin atoms(\mathcal{C})$ is used, we use the T-Learn* rule to learn the clause $L \vee \neg L$ and get $L \in atoms(\mathcal{C})$.

MCSAT Decide:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![M, L]\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} L \in \mathcal{B}, \\ \text{value}(L, M) = \mathit{undef} \end{array}$$

If the decision literal L is a new literal, we use the above technique to add $L \vee \neg L$ to $\mathcal{C}$ and ensure that $L \in atoms(\mathcal{C})$. Now, we can apply the Decide rule to perform the equivalent task in CDCL*(T).

MCSAT Propagate:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![M, C \to L]\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} C = (L_1 \vee \cdots \vee L_n \vee L) \in \mathcal{C}, \\ \forall i.\ \text{value}(L_i, M) = \mathit{false}, \\ \text{value}(L, M) = \mathit{undef} \end{array}$$

This rule is identical to the UnitPropagate rule up to the usual reformulations.

MCSAT Conflict: We ignore this rule as argued above. We process the whole subsequent conflict analysis when we leave the conflict analysis state with the Backjump, Unsat, or T-Backjump-Decide rules.

MCSAT Sat:

$$\frac{\langle M, \mathcal{C}\rangle}{sat} \qquad \textbf{if } \texttt{satisfied}(\mathcal{C}, M)$$

Instead of a special *sat* state, CDCL*(T) signals satisfiability by terminating in any state other than $FailState$ when no further rule can be applied. Therefore, we do nothing in this case.

MCSAT Forget:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle M, \mathcal{C} \setminus \{C\}\rangle} \qquad \textbf{if } C \in \mathcal{C} \text{ is a learned clause}$$

This rule is a special case of the CDCL*(T) Forget rule, restricted to learned clauses. We can thus apply the Forget rule.

MCSAT Restart: The Restart rules of MCSAT and CDCL*(T) are identical.

MCSAT Resolve: We ignore this rule as argued above. We process the whole subsequent conflict analysis when we leave the conflict analysis state with the Backjump, Unsat, or T-Backjump-Decide rules.

MCSAT Consume: We ignore this rule as argued above. We process the whole subsequent conflict analysis when we leave the conflict analysis state with the Backjump, Unsat, or T-Backjump-Decide rules.

MCSAT Backjump:

$$\frac{\langle [\![M, N]\!], \mathcal{C}\rangle \Vdash C}{\langle [\![M, C \to L]\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} C = (L_1 \vee \cdots \vee L_n \vee L), \\ \forall i.\ \text{value}(L_i, M) = \mathit{false}, \\ \text{value}(L, M) = \mathit{undef}, \\ N \text{ starts with a decision} \end{array}$$

We can simulate this rule by the T-Backjump rule. We have L in the trail as for MCSAT N starts with a decision. Furthermore, the clause $C' \vee L'$ can be chosen to be the MCSAT conflict clause C and has the required properties: it is entailed by $\mathcal{C}$ as it was constructed by resolution from $\mathcal{C}$, $M \models \neg C'$ holds, L' is not assigned, and all literals are from $atoms(\mathcal{C})$ – after we added them as described above.

T-Backjump:

$$\frac{\langle [\![M, L, N]\!], \mathcal{C}\rangle}{\langle [\![M, (C' \vee L') \to L']\!], \mathcal{C}\rangle} \quad \textbf{if} \quad \begin{array}{l} C \in \mathcal{C} \text{ with } [\![M, L, N]\!] \models \neg C, \\ \text{there is some clause } C' \vee L' \text{ such that:} \\ \quad \mathcal{C} \models_T C' \vee L' \text{ and } M \models \neg C', \\ \quad L' \text{ is undefined in } M, \\ \quad L' \text{ or } \neg L' \text{ occurs in } \mathcal{C} \text{ or in } [\![M, L, N]\!] \end{array}$$

MCSAT Unsat:

$$\frac{\langle M, \mathcal{C}\rangle \Vdash \mathit{false}}{\mathit{unsat}}$$

As we would expect, this is essentially a special case of the `Fail` rule with $C = \mathit{false}$. We first add the conflict clause () to $\mathcal{C}$ using the `T-Learn` rule, restart the solver, and finally apply the `Fail` rule with the newly added empty clause.

MCSAT Learn:

$$\frac{\langle M, \mathcal{C}\rangle \Vdash C}{\langle M, \mathcal{C} \cup \{C\}\rangle} \qquad \textbf{if} \quad C \notin \mathcal{C}$$

As argued above, the current conflict clause of `MCSAT` is a logical consequence of $\mathcal{C}$ at any given time. Hence we can apply the `T-Learn`* rule to add C to $\mathcal{C}$.

MCSAT T-Propagate:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![M, E \to L]\!], \mathcal{C}\rangle} \qquad \begin{array}{ll} & L \in \mathcal{B}, \mathrm{value}(L, M) = \mathit{undef}, \\ \textbf{if} & \texttt{infeasible}([\![M, \neg L]\!]), \\ & E = \texttt{explain}([\![M, \neg L]\!]) \end{array}$$

We recall that `explain` always returns a *valid theory lemma* – a tautology – and thus every clause returned by `explain` is a logical consequence of $\mathcal{C}$. We can, thus, use the `T-Learn`* rule to add the clause E to $\mathcal{C}$ and then perform a Boolean propagation via the `UnitPropagate` rule instead of the theory propagation.
We can also, alternatively, eliminate the `T-Propagate` rule within `MCSAT` itself. Given that $\mathrm{value}(L, M) = \mathit{undef}$, we can apply the `Decide` rule to add $\neg L$ to the trail and as $\texttt{infeasible}([\![M, \neg L]\!])$ we can apply the `T-Conflict` rule. The `Backjump` rule immediately resolves this conflict and replaces the decision $\neg L$ by the propagation $E \to L$.

Decide:

$$\frac{\langle M, \mathcal{C}\rangle}{\langle [\![M, \neg L]\!], \mathcal{C}\rangle}$$

T-Conflict:

$$\frac{\langle [\![M, \neg L]\!], \mathcal{C}\rangle}{\langle [\![M, \neg L]\!], \mathcal{C}\rangle \Vdash E}$$

Backjump:

$$\frac{\langle [\![M, \neg L]\!], \mathcal{C}\rangle \Vdash E}{\langle [\![M, E \to L]\!], \mathcal{C}\rangle}$$

MCSAT T-Decide: We ignore this rule as argued above. Theory assignments are not needed to simulate `MCSAT`.

MCSAT T-Conflict: We ignore this rule as argued above. We process the whole subsequent conflict analysis when we leave the conflict analysis state with the `Backjump`, `Unsat`, or `T-Backjump-Decide` rules.

MCSAT T-Consume: We ignore this rule as argued above. We process the whole subsequent conflict analysis when we leave the conflict analysis state with the Backjump, Unsat, or T-Backjump-Decide rules.

MCSAT T-Backjump-Decide:

$$\frac{\langle [\![M, x \mapsto \alpha_x, N]\!], \mathcal{C}\rangle \Vdash C}{\langle [\![M, L]\!], \mathcal{C}\rangle} \quad \begin{array}{ll} & C = (L_1 \vee \cdots \vee L_n \vee L), \\ \textbf{if} & \exists i.\ \text{value}(L_i, M) = \mathit{undef}, \\ & \text{value}(L, M) = \mathit{undef} \end{array}$$

As we ignore theory assignments, we simulate this rule as follows. First, we remove N from the trail – restarting the solver and restoring M – and then use the Decide rule on L, which is possible as $\text{value}(L, M) = \mathit{undef}$.

We have shown how to simulate every MCSAT proof rule with the CDCL*(T) proof system and thereby how to algorithmically simulate MCSAT with CDCL*(T), concluding the proof for Theorem 9.2. This proof not only completely ignored theory assignments, but also did not need to bother with the details of conflict analysis. □

This concludes our proof of Theorem 9.2, showing that MCSAT is, in fact, algorithmically equivalent to CDCL*(T).

9.3 Theory reasoning in practice

The simulations we presented in Section 9.1 and Section 9.2 have a few shortcomings (or rather only hold true under certain assumptions), some of which have already been mentioned. We now discuss these issues, how they affect the theory or practical implementations, and how one might mitigate them.

9.3.1 Completeness of infeasibility check

We have already issued warnings that the constructions shown in Section 9.1 and Section 9.2 require a level of power by the MCSAT subroutines – in particular infeasible – that is backed by the MCSAT framework, but not by any practical implementation.

The definition from [MJ13] uses the predicate infeasible to state that a trail M is not satisfiable in the sense that the literals from M are not satisfiable together with the theory model induced by M. Note that the previous version from [JM12] still used a slightly different version: feasibility only considered the constraints that are *univariate* over the theory model. Though seemingly small, this is a pivotal difference here.

The presented simulations all require the more general notion of infeasibility from [MJ13] to ensure that *feasibility of the trail* coincides with the trail *implying false*. Practical implementations, however, all follow [JM12] where we only consider univariate constraints for feasibility. As far as we know, the SMT-based assignment finder from Section 8.2.2 is the only exception to this, though with rather disappointing results.

One might even reasonably argue that this restriction to only consider univariate constraints is not a technical one, but is a fundamental idea in MCSAT that allows

the theory exploration in the first place. A complete implementation of `infeasible` would essentially prevent the theory exploration and move the whole process of theory exploration into `infeasible` that turns into a regular theory solver – just like the SMT-based assignment finder from Section 8.2.2.

Let us first observe that employing an "incomplete" version of `infeasible` does not make `MCSAT` incomplete. It only requires theory exploration whenever `infeasible` ignores non-univariate constraints, but by gradually exploring the whole space of theory models one eventually correctly determines infeasibility. We thus claim – without proof – that all simulations can be adapted accordingly to accommodate for an incomplete implementation of `infeasible`.

The basic idea is as follows: instead of immediately finding infeasibility, we guess a partial theory assignment (via the `T-Decide` rule) until it becomes infeasible – note that `infeasible` is definitely complete once all but one variable are assigned – exclude a region (via the `T-Conflict` rule) around the partial assignment (in the space of theory assignments) and backtrack. This usually happens multiple times for a certain variable until the whole space is excluded for this variable and we have to change the assignment of the previous variable. We thereby accumulate witnesses for infeasibility and combine them to contain less and less variables until we eventually get back to our starting point and can derive infeasibility from univariate constraints.

While this might only seem like an issue of reformulating the proofs and making them (a lot) more technical and arguably ugly, it gives rise to a fundamental problem. We required the lengths of equivalent proofs to only differ polynomially. However, the theory exploration described above essentially enumerates invariant regions – we usually think about CAD cells here – and there can be exponentially many of them.

This, unfortunately, conflicts with our hope of a *polynomial* reduction. However, this was to be expected: the whole idea of `MCSAT` is to move parts of the theory reasoning into the core proof system while $\texttt{Res}^*\texttt{(T)}$ and $\texttt{CDCL}^*\texttt{(T)}$ both *hide* any theory reasoning within their proof rules (of constant cost). Thus, our core proof system *of course* exhibits bad asymptotic behavior if we consider a hard theory, while the proof system we compare it to completely hides any theory reasoning from its asymptotic complexity.

While the theoretical result still remains if `infeasible` is complete, we also argue that it is relevant in practice as well when `infeasible` is incomplete. We observed that the computational effort spent on theory reasoning is not *new* but is only made explicit in `MCSAT`, and thus the overall computational effort stays (about) the same.

9.3.2 Theory-aware equivalency

We now want to argue that the length of proofs is not a satisfactory measure to assess the computational complexity of performing proofs. The foregoing discussion essentially shows that we can make a proof system arbitrarily more powerful – in the sense of proof complexity or algorithmic simulation – if we make its proof rules more powerful. Conversely, we can make a proof system less powerful by making its proof rules ever more detailed, possibly executing individual processor instructions.

It is, thus, of utmost importance for a meaningful comparison that the proof systems operate on a similar level of abstraction if we assume that all proof rules take a fixed amount of time. We propose to treat proof rules that involve *theory queries* – whatever a *theory query* may be – in a special way. To call proof systems equivalent (either

bisimilar or *algorithmically equivalent*) we require the *number of theory queries* to be (about) the same and let all theory queries have (about) the *same computational effort*.

Reconsidering the proofs for bisimilarity and algorithmic equivalency, we observe that the number of theory queries is always identical. Every proof rule that involves a theory query is simulated by a set of proof rule applications that involve exactly one theory query as well. Thus we do not need to worry about the *number of theory queries*.

A closer look at the reductions involving theory queries reveals that they are used to construct the *exact same* clauses from the *exact same* theory constraints. We can thus argue that any method that would improve upon a particular implementation of the `MCSAT` theory queries can directly be used to improve the `CDCL*(T)` theory queries, and the other way round.

9.3.3 Impact of new literals

`MCSAT` has the ability to add new literals via its `explain` method that is used in the `T-Propagate` and `T-Conflict` rules. Though we could theoretically restrict `explain` to only use existing literals, it would forbid all existing instantiations of `MCSAT` we know of, in particular the CAD-based `NLSAT` [JM12]. The way `explain` is meant to work – by eliminating all unassigned variables – inherently generates new literals.

We know that adding strong theory derivations to `CDCL(T)` brings us from `Res(T)` to `Res*(T)` in theory. Restricting `explain` to what we described above, that is a quantifier elimination procedure that removes the unassigned variables, gives us something between: it provides a way to generate theory lemmas with additional literals, which may make it stronger than `CDCL(T)` without strong theory derivations, but it remains unclear whether we can *practically* generate any arbitrary clause or an equivalent clause – whatever equivalent would mean – like we assume in our simulation of the `Learn` rule.

Note that we do not consider the simulation of strong theory derivations in the reduction from `CDCL*(T)` to `MCSAT` *practical* in that sense. We have shown that we can instrument `MCSAT` to construct any valid clause *that we already know*. The reduction, however, did not provide a constructive way to find *novel clauses*, just like `CDCL*(T)`. Finding meaningful lemmas is a difficult problem and actual implementations of `CDCL*(T)` have a hard time leveraging the power of strong theory derivations.

This might suggest that real-world implementations of `MCSAT` are less powerful than `CDCL*(T)`. Instead, `MCSAT` might have found a sweet spot between `CDCL(T)` and `CDCL*(T)` – and thus between `Res(T)` and `Res*(T)` – in that it shows how to generate certain meaningful lemmas from a restricted class, but not a general scheme to generate all valid lemmas.

9.3.4 Impact of theory decisions

We have seen that theory decisions were not used when we showed the equivalence of `CDCL*(T)` and `MCSAT`. We did not need them to be as powerful as `CDCL*(T)` and could simply ignore them when simulating `MCSAT` with `CDCL*(T)` without harm. This suggests that they are not an essential *theoretical* part of `MCSAT` after all. They seem to be no more than a tool to steer decisions and the generation of new clauses to a good direction in a heuristic way – a very effective one in practice though.

Conclusion

In this work, we presented various techniques for SMT solving of nonlinear real arithmetic with a special focus on CAD-based procedures for theory solving. Our work mainly centered around an embedding of CAD as a theory solver for traditional (lazy) SMT solving as presented in Chapter 6 and the implementation of the more recent `MCSAT` framework with explanation functions based on CAD in Chapter 8.

For both subjects, we recalled basic definitions and techniques, proposed novel techniques concerning both the theoretical framework and an actual implementation, and extended it beyond the traditional scope of SMT solving. In the following, we give some more details on the contributions of this thesis, differentiating it from previous work, and the future work that was (in part) already brought up within this work, but neither implemented nor properly dealt with yet.

10.1 Contributions

We already gave a brief overview of our contributions in Section 1.2 that we extend now that we have seen the details of this work. Note that we include the author's contributions to published work here, but also make explicit where the presentation in this thesis exceeds the published works listed in Section 1.2.1. Unless stated otherwise, everything mentioned below has been implemented in `SMT-RAT` [CKJ+15] which we consider a contribution in itself.

The author's contributions already start in Section 2.5 on the topic of real algebraic numbers, a crucial part of a sufficiently efficient implementation of CAD and most CAD-based methods. Given this importance, it has already been studied quite a bit and, thus, our presentation does not contain any fundamentally novel techniques or insights. However, discussions of how to implement the important methods are extremely sparse: algebraic works tend to simply assume knowledge about *how* to implement real algebraic numbers while others merely use it as a starting point like [MP13]. In this sense, our presentation is unique in that it discusses how to actually implement real algebraic numbers from building blocks that are easy enough for a computer scientist to understand and implement.

Similarly, Chapter 5 (and in particular Section 5.2) does not contain novel techniques for CAD, but a yet not written summary of existing methods for CAD, in particular projection operators and a comparative analysis thereof which is in parts based on [Vie16; VKÁ17]. Again, we did not focus on the theoretical background (which is both mathematically deep and very diverse) but aimed at making the discussion sufficiently easy to understand, giving a feeling for the impact of the different methods in practice.

In Chapter 6 we first presented a novel formulation of CAD as a proof system that aims to bridge the gap between the mathematical presentations of CAD that prevail in the computer algebra community and the much more practical and algorithmic description from [KÁ20]. These formulations of CAD aim to make an *incremental* implementation as easy as possible to allow using it as a theory solver in the sense of [KÁ18]. Of course, the proof system is heavily based on [KÁ20] and could be seen as a mere reformulation of the algorithms presented there.

This proof system was then extended using known techniques like full factorization, equational constraints, equation inference based on the resultant rule, infeasible subset generation, or branch and bound for integer problems. While all of these techniques are more or less well-known, we combined them into a single framework that allows for a seamless combination of them. This of course based on previously published work on equational constraints [Hae18; HKÁ18], infeasible subsets [Hen17] and nonlinear integer arithmetic [KCÁ16].

Furthermore, the implementation – which was targeted at SMT problems and heavily focussed on incrementality – was applied to quantifier elimination and an adaptation for optimization was presented. While quantifier elimination found its way into our actual implementation in [Neu18a], the work on optimization has not, but still waits to be implemented.

We then turned towards the `MCSAT` framework, a comparably recent alternative to what we called *(lazy) SMT solving*, that yielded particularly good results with its implementation for nonlinear real arithmetic in `Z3` [JM12]. Starting from a slightly revised definition of the underlying proof system (compared to [MJ13]), we introduced a novel variant we call *model-refining satisfiability calculus* and proposed a way to employ `MCSAT` for optimization problems, though both have not been implemented yet.

In Chapter 8, we discussed our implementation of `MCSAT`, the only `MCSAT`-based implementation targeted at nonlinear arithmetic apart from `Yices` and `Z3` [JM12] (which arguably can be considered the same, given that the same author implemented them in close succession).

We proposed a novel embedding of `MCSAT` into a `CDCL`-style SAT solver – in some sense approaching implementations of regular SMT solving and `MCSAT` – and proposed a novel assignment finder as well as multiple novel explanation functions and methods to compose them. Again, some of these were already published in [Nal17; ÁNK17; Neu18b]. This novel implementation of `MCSAT` was evaluated, in particular, for varying explanation functions and varying variable orderings in Section 8.5, in parts based on [NKÁ19].

Finally, we picked up a thread concerning a more theoretical view on proof systems (like `CDCL(T)` or `MCSAT`) from [RKG18] and studied the theoretical power of `MCSAT` in comparison to `Res`*`(T)` and `CDCL`*`(T)`. The first part is concerned with the notion of *proof complexity* and establishes the equivalence of `MCSAT` and `Res`*`(T)` in this context, closely based on [KÁG19]. In the second part, we depart from proof complexity and directly show what we call *algorithmic equivalency* between `MCSAT` and `CDCL`*`(T)`. Finally, we discuss certain shortcomings of our proof when compared to actual implementations and how this affects the power of `MCSAT` implementations in comparison to the `CDCL`*`(T)` proof system and implementations thereof, as well as the importance of the novel concepts in `MCSAT` for its practical and theoretical performance.

10.2 Future work

We can easily imagine improvements and adaptations for almost all of the presented concepts and techniques and the scope of this thesis is limited due to restrictions in space and time, but not for the lack of open questions or (more or less) novel ideas that deserve thorough consideration. We thus only give a few directions for future research that we feel are particularly promising and important (in no specific order).

In Section 6.5.2, we discussed how we can arbitrarily interleave the projection and lifting process in an incremental CAD. Though we did some brief experiments on this issue, the most effective heuristic in practice was to strictly prefer lifting steps. We feel that this does not align with concepts like *open CAD* and our general idea of incrementality, speculating that better schedulers should exist.

The proof system of `MCSAT` feels somewhat convoluted and cumbersome and we propose to aim for a more elegant version of `MCSAT`. Our understanding of `MCSAT` is to treat the theory reasoning much like the Boolean reasoning, but the proof system handles them very differently, though we do not see a fundamental reason for this. Furthermore, we propose to simplify the proof system with respect to conflict analysis and define it more alike to `CDCL*(T)`.

The discussion in Section 9.3 indicate that the main reason for the practical effectiveness of `MCSAT` might be that it provides a constructive way to obtain *useful* lemmas while this is a notoriously hard problem within `CDCL*(T)`. It might thus be interesting whether we can embed parts of `MCSAT` into `CDCL*(T)`, purely to generate such useful lemmas. On the other hand, approaching a theory problem in a *conflict-driven* manner may yield interesting new decision procedures. Both NuCAD [Bro15] and the cylindrical algebraic coverings approach from [ÁDE+20] indicate that such methods are competitive in first experiments and deserve more research.

Finally, we observe that the growing number of industrial applications leads to the wish to extend SMT solving beyond the question of satisfiability. While "SMT solving" has always incorporated the question for a model and *unsatisfiable cores* have long been studied for propositional problems and are slowly adapted for SMT solving as well, one important component for real-world applications is *optimization*. While several groups work on *optimization modulo theories*, no approach has gained any traction for *nonlinear* optimization yet. We have laid out how to use a CAD-based theory solver as well as `MCSAT` for optimization, and note that the CAD-based method gives strong guarantees on the results, arguably even stronger than other methods for nonlinear optimizations as we discuss in Section 1.1.6.2. We thus expect that an effective optimization based on CAD may be extremely useful in practice if it scales sufficiently, which is, of course, a notoriously hard problem for any CAD-based approach.

Bibliography

[AB19] John Abbott and Anna M. Bigatti. *CoCoALib: a C++ library for doing Computations in Commutative Algebra*. 2019. URL: http://cocoa.dima.unige.it/cocoalib.

[ABP18] John Abbott, Anna M. Bigatti, and Elisa Palezzato. "New in CoCoA-5.2.4 and CoCoALib-0.99600 for SC-Square". In: *Satisfiability Checking and Symbolic Computation* (SC^2 2018) at FLoC. CEUR Workshop Proceedings vol. 2189, pp. 88–94. URL: http://ceur-ws.org/Vol-2189/paper4.pdf.

[Abe26] Niels H. Abel. "Beweis der Unmöglichkeit, algebraische Gleichungen von höheren Graden als dem vierten allgemein aufzulösen." In: *Journal für die reine und angewandte Mathematik* 1 (1826), pp. 65–84. DOI: 10.1515/crll.1826.1.65.

[ÁCJ+16] Erika Ábrahám, Florian Corzilius, Einar Broch Johnsen, Gereon Kremer, and Jacopo Mauro. "Zephyrus2: On the Fly Deployment Optimization Using SMT and CP Technologies". In: *Dependable Software Engineering: Theories, Tools, and Applications* (SETTA 2016). LNCS vol. 9984, pp. 229–245. DOI: 10.1007/978-3-319-47677-3_15.

[ÁDE+20] Erika Ábrahám, James H. Davenport, Matthew England, and Gereon Kremer. "Deciding the Consistency of Non-Linear Real Arithmetic Constraints with a Conflict Driven Search Using Cylindrical Algebraic Coverings". In: *arXiv e-prints* (2020). Accepted at Journal of Logical and Algebraic Methods in Programming. arXiv: 2003.05633.

[ÁK16] Erika Ábrahám and Gereon Kremer. "Satisfiability Checking: Theory and Applications". In: *Software Engineering and Formal Methods* (SEFM 2016). LNCS vol. 9763, pp. 9–23. DOI: 10.1007/978-3-319-41591-8_2.

[ÁK17] Erika Ábrahám and Gereon Kremer. "SMT Solving for Arithmetic Theories: Theory and Tool Support". In: *Symbolic and Numeric Algorithms for Scientific Computing* (SYNASC 2017), pp. 1–8. DOI: 10.1109/SYNASC.2017.00009.

[ÁK18] Erika Ábrahám and Gereon Kremer. "Incremental CAD". Presented at International Congress on Mathematical Software. 2018.

[ÁNK17] Erika Ábrahám, Jasper Nalbach, and Gereon Kremer. "Embedding the Virtual Substitution Method in the Model Constructing Satisfiability Calculus Framework". In: *Satisfiability Checking and Symbolic Computation* (SC^2 2017) at ISSAC. CEUR Workshop Proceedings vol. 1974. URL: http://ceur-ws.org/Vol-1974/EAb.pdf.

[ACG99] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. "SAT-Based Procedures for Temporal Reasoning". In: *Recent Advances in AI Planning* (ECP 1999). LNCS vol. 1809, pp. 97–108. DOI: 10.1007/10720246_8.

[Arn81] Dennis S. Arnon. "Algorithms for the Geometry of Semi-Algebraic Sets". PhD thesis. University of Wisconsin-Madison, 1981. URL: https://digital.library.wisc.edu/1793/58310.

[ACM84] Dennis S. Arnon, George E. Collins, and Scott McCallum. "Cylindrical Algebraic Decomposition I: The Basic Algorithm". In: *SIAM Journal on Computing* 13 (4 1984), pp. 865–877. DOI: 10.1137/0213054.

[Art91] Michael Artin. *Algebra*. Prentice Hall Inc., 1991.

[ADG07] Eugene Asarin, Thao Dang, and Antoine Girard. "Hybridization methods for the analysis of nonlinear systems". In: *Acta Informatica* 43 (7 2007), pp. 451–476. DOI: 10.1007/s00236-006-0035-7.

[AS09] Gilles Audemard and Laurent Simon. "Predicting Learnt Clauses Quality in Modern SAT Solvers". In: *International Joint Conference on Artificial Intelligence* (IJCAI 2009), pp. 399–404. URL: https://ijcai.org/Proceedings/09/Papers/074.pdf.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *Computer Aided Verification* (CAV 2011). LNCS vol. 6806, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14.

[BMS05] Clark Barrett, Leonardo de Moura, and Aaron Stump. "Design and Results of the First Satisfiability Modulo Theories Competition (SMT-COMP 2005)". In: *Journal of Automated Reasoning* 35 (4 2005), pp. 373–390. DOI: 10.1007/s10817-006-9026-1.

[BDS02] Clark W. Barrett, David L. Dill, and Aaron Stump. "Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT". In: *Computer Aided Verification* (CAV 2002). LNCS vol. 2404, pp. 236–249. DOI: 10.1007/3-540-45657-0_18.

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

[Bar18] Lorena Calvo Bartolomé. "Using Fourier-Motzkin Variable Elimination for MCSAT Explanations in SMT-RAT". Bachelor's thesis. RWTH Aachen University, 2018.

[BPR96] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. "On the Combinatorial and Algebraic Complexity of Quantifier Elimination". In: *Journal of the ACM* 43 (6 1996), pp. 1002–1045. DOI: 10.1145/235809.235813.

[BPR10] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry*. 2nd ed. Vol. 10. Algorithms and Computation in Mathematics. Springer, 2010. DOI: 10.1007/3-540-33099-2.

[BS97] Roberto J. Bayardo Jr. and Robert Schrag. "Using CSP Look-Back Techniques to Solve Real-World SAT Instances". In: *National Conference on Artificial Intelligence* (AAAI 1997), pp. 203–208. URL: https://www.aaai.org/Papers/AAAI/1997/AAAI97-032.pdf.

[BG06] Frédéric Benhamou and Laurent Granvilliers. "Continuous and Interval Constraints". In: *Handbook of Constraint Programming*. Ed. by Francesca Rossi, Peter van Beek, and Toby Walsh. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006, pp. 571–603. DOI: 10.1016/S1574-6526(06)80020-9.

[BHM+09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009.

[BPF15] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. "νZ - An Optimizing SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS 2015). LNCS vol. 9035, pp. 194–199. DOI: 10.1007/978-3-662-46681-0_14.

[BGM+18] François Bobot, Stéphane Graham-Lengrand, Bruno Marre, and Guillaume Bury. "Centralizing equality reasoning in MCSAT". In: *Satisfiability Modulo Theories* (SMT 2018) at FLoC. URL: https://hal.archives-ouvertes.fr/hal-01935591.

[Boo19] Boost development team. *The Boost C++ Library*. 2019. URL: https://boost.org.

[BM90] Robert S. Boyer and J. Strother Moore. "A Theorem Prover for a Computational Logic". In: *Automated Deduction* (CADE-10 1990). LNCS vol. 449, pp. 1–15. DOI: 10.1007/3-540-52885-7_75.

[BBC+05] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Silvio Ranise, Peter van Rossum, and Roberto Sebastiani. "Efficient Satisfiability Modulo Theories via Delayed Theory Combination". In: *Computer Aided Verification* (CAV 2005). LNCS vol. 3576, pp. 335–349. DOI: 10.1007/11513988_34.

[BDE+16] Russell Bradford, James H. Davenport, Matthew England, Scott McCallum, and David Wilson. "Truth table invariant cylindrical algebraic decomposition". In: *Journal of Symbolic Computation* 76 (2016), pp. 1–35. DOI: 10.1016/j.jsc.2015.11.002.

[Bro98] Christopher W. Brown. "Simplification of Truth-Invariant Cylindrical Algebraic Decompositions". In: *International Symposium on Symbolic and Algebraic Computation* (ISSAC 1998), pp. 295–301. DOI: 10.1145/281508.281652.

[Bro99] Christopher W. Brown. "Solution Formula Construction for Truth Invariant CADs". PhD thesis. University of Delaware, 1999. URL: https://www.usna.edu/Users/cs/wcbrown/research/Thesis.html.

[Bro01] Christopher W. Brown. "Improved Projection for Cylindrical Algebraic Decomposition". In: *Journal of Symbolic Computation* 32 (5 2001), pp. 447–465. DOI: 10.1006/jsco.2001.0463.

[Bro03] Christopher W. Brown. "QEPCAD B: A program for computing with semi-algebraic sets using CADs". In: *ACM SIGSAM Bulletin* 37 (4 2003), pp. 97–108. DOI: 10.1145/968708.968710.

[Bro04] Christopher W. Brown. *Companion to the Tutorial Cylindrical Algebraic Decomposition. Presented at ISSAC*. 2004.

[Bro13] Christopher W. Brown. "Constructing a Single Open Cell in a Cylindrical Algebraic Decomposition". In: *International Symposium on Symbolic and Algebraic Computation* (ISSAC 2013), pp. 133–140. DOI: 10.1145/2465506.2465952.

[Bro15] Christopher W. Brown. "Open Non-uniform Cylindrical Algebraic Decompositions". In: *International Symposium on Symbolic and Algebraic Computation* (ISSAC 2015), pp. 85–92. DOI: 10.1145/2755996.2756654.

[BK15] Christopher W. Brown and Marek Košta. "Constructing a single cell in cylindrical algebraic decomposition". In: *Journal of Symbolic Computation* 70 (2015), pp. 14–48. DOI: 10.1016/j.jsc.2014.09.024.

[BGV99] Randal E. Bryant, Steven German, and Miroslav N. Velev. "Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions". In: *Computer Aided Verification* (CAV 1999). LNCS vol. 1633, pp. 470–482. DOI: 10.1007/3-540-48683-6_40.

[Buc65] Bruno Buchberger. "Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal". PhD thesis. University of Innsbruck, 1965.

[BB92] Michael Buro and Hans Kleine Büning. *Report on a SAT competition*. Tech. rep. Fachbereich Mathematik-Informatik, 1992. URL: https://stamm-wilbrandt.de/en/Report_on_a_SAT_competition.pdf.

[Cau28] Augustin Louis Baron Cauchy. *Exercices de mathématiques*. Vol. 3. Bure frères, 1828.

[CJ98] Bob Forrester Caviness and Jeremy R. Johnson. *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer, 1998. DOI: 10.1007/978-3-7091-9459-1.

[CMX+09] Changbo Chen, Marc Moreno Maza, Bican Xia, and Lu Yang. "Computing Cylindrical Algebraic Decomposition via Triangular Decomposition". In: *International Symposium on Symbolic and Algebraic Computation* (ISSAC 2009), pp. 95–102. DOI: 10.1145/1576702.1576718.

[Chu36] Alonzo Church. "An Unsolvable Problem of Elementary Number Theory". In: *American Journal of Mathematics* 58.2 (1936), pp. 345–363. DOI: 10.2307/2371045.

[Chu69] Alonzo Church. "Alfred Tarski. The completeness of elementary algebra and geometry". Review. In: *Journal of Symbolic Logic* 34 (2 1969), p. 302. DOI: 10.2307/2271123.

[Chv79] Vasek Chvátal. "A Greedy Heuristic for the Set-Covering Problem". In: *Mathematics of Operations Research* 4.3 (1979), pp. 233–235. DOI: 10.1287/moor.4.3.233.

[CGI+18] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. "Incremental Linearization for Satisfiability and Verification Modulo Nonlinear Arithmetic and Transcendental Functions". In: *ACM Transactions on Computational Logic* 19 (3 2018), 19:1–19:52. DOI: 10.1145/3230639.

[CGS+13] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. "The MathSAT5 SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS 2013). LNCS vol. 7795. DOI: 10.1007/978-3-642-36742-7_7.

[CKS+04] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. "Predicate Abstraction of ANSI-C Programs Using SAT". In: *Formal Methods in System Design* 25 (2–3 2004), pp. 105–127. DOI: 10.1023/B:FORM.0000040025.89719.f3.

[Col56] George E. Collins. "The Tarski decision procedure". In: *Proceedings of the 1956 11th ACM National Meeting* (ACM 1956), pp. 162–164. DOI: 10.1145/800258.808975.

[Col60] George E. Collins. "A Method for Overlapping and Erasure of Lists". In: *Communications of the ACM* 3 (12 1960), pp. 655–657. DOI: 10.1145/367487.367501.

[Col66] George E. Collins. "PM, A System for Polynomial Manipulation". In: *Communications of the ACM* 9 (8 1966), pp. 578–589. DOI: 10.1145/365758.365770.

[Col67] George E. Collins. "Subresultants and Reduced Polynomial Remainder Sequences". In: *Journal of the ACM* 14 (1 1967), pp. 128–142. DOI: 10.1145/321371.321381.

[Col71a] George E. Collins. "The Calculation of Multivariate Polynomial Resultants". In: *Journal of the ACM* 18 (4 1971), pp. 515–532. DOI: 10.1145/321662.321666.

[Col71b] George E. Collins. "The SAC-1 System: An Introduction and Survey". In: *ACM Symposium on Symbolic and Algebraic Manipulation* (SYMSAC 1971), pp. 144–152. DOI: 10.1145/800204.806279.

[Col73] George E. Collins. "Efficient Quantifier Elimination for Elementary Algebra". Presented at Symposium on Complexity of Sequential and Parallel Numerical Algorithms. 1973.

[Col74] George E. Collins. "Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition–Preliminary Report". In: *ACM SIGSAM Bulletin* 8 (3 1974), pp. 80–90. DOI: 10.1145/1086837.1086852.

[Col75] George E. Collins. "Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition". In: *Automata Theory and Formal Languages* (2nd GI Conference 1975). LNCS vol. 33, pp. 134–183. DOI: 10.1007/3-540-07407-4_17.

[Col85] George E. Collins. "The SAC-2 Computer Algebra System". In: *European Conference on Computer Algebra* (EUROCAL 1985). LNCS vol. 204, pp. 34–35. DOI: 10.1007/3-540-15984-3_235.

[Col98] George E. Collins. "Quantifier Elimination by Cylindrical Algebraic Decomposition — Twenty Years of Progress". In: *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Ed. by Bob F. Caviness and Jeremy R. Johnson. Springer, 1998, pp. 8–23. DOI: 10.1007/978-3-7091-9459-1_2.

[CA76] George E. Collins and Alkiviadis G. Akritas. "Polynomial Real Root Isolation Using Descarte's Rule of Signs". In: *ACM Symposium on Symbolic and Algebraic Computation* (SYMSAC 1976), pp. 272–275. DOI: 10.1145/800205.806346.

[CH91] George E. Collins and Hoon Hong. "Partial Cylindrical Algebraic Decomposition for Quantifier Elimination". In: *Journal of Symbolic Computation* 12 (3 1991), pp. 299–328. DOI: 10.1016/S0747-7171(08)80152-6.

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. "SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving". In: *Theory and Applications of Satisfiability Testing* (SAT 2015). LNCS vol. 9340, pp. 360–368. DOI: 10.1007/978-3-319-24318-4_26.

[CLJ+12] Florian Corzilius, Ulrich Loup, Sebastian Junges, and Erika Ábrahám. "SMT-RAT: An SMT-Compliant Nonlinear Real Arithmetic Toolbox (Tool Presentation)". In: *Theory and Applications of Satisfiability Testing* (SAT 2012). LNCS vol. 7317. DOI: 10.1007/978-3-642-31612-8_35.

[Cro75] John N. Crossley. "Reminiscences of Logicians". In: *Algebra and Logic* 1975. LNM vol. 450, pp. 1–62. DOI: 10.1007/BFb0062850.

[DH88] James H. Davenport and Joos Heintz. "Real Quantifier Elimination is Doubly Exponential". In: *Journal of Symbolic Computation* 5 (1–2 1988), pp. 29–35. DOI: 10.1016/S0747-7171(88)80004-X.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. "A Machine Program for Theorem-Proving". In: *Communications of the ACM* 5 (7 1962), pp. 394–397. DOI: 10.1145/368273.368557.

[DP60] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *Journal of the ACM* 7 (3 1960), pp. 201–215. DOI: 10.1145/321033.321034.

[Des37] René Descartes. *La Géométrie*. 1637. URL: https://journals.openedition.org/bibnum/635.

[Din19] Lloyd L. Dines. "Systems of Linear Inequalities". In: *Annals of Mathematics* 20.3 (1919), pp. 191–199. DOI: 10.2307/1967869.

[DSS04] Andreas Dolzmann, Andreas Seidl, and Thomas Sturm. "Efficient Projection Orders for CAD". In: *International Symposium on Symbolic and Algebraic Computation* (ISSAC 2004), pp. 111–118. DOI: 10.1145/1005285.1005303.

[DS97] Andreas Dolzmann and Thomas Sturm. "REDLOG: Computer Algebra Meets Computer Logic". In: *ACM SIGSAM Bulletin* 31 (2 1997), pp. 2–9. DOI: 10.1145/261320.261324.

[Dri88] Lou van den Dries. "Alfred Tarski's Elimination Theory for Real Closed Fields". In: *The Journal of Symbolic Logic* 53.1 (1988), pp. 7–19. DOI: 10.2307/2274424.

[Duc00] Lionel Ducos. "Optimizations of the subresultant algorithm". In: *Journal of Pure and Applied Algebra* 145 (2 2000), pp. 149–163. DOI: 10.1016/S0022-4049(98)00081-4.

[Dut14] Bruno Dutertre. "Yices 2.2". In: *Computer Aided Verification* (CAV 2014). LNCS vol. 8559, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49.

[DM06] Bruno Dutertre and Leonardo de Moura. "A Fast Linear-Arithmetic Solver for DPLL(T)". In: *Computer Aided Verification* (CAV 2006), pp. 81–94. DOI: 10.1007/11817963_11.

[ES03] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing* (SAT 2003). LNCS vol. 2919, pp. 502–518. DOI: 10.1007/978-3-540-24605-3_37.

[EBD15] Matthew England, Russell Bradford, and James H. Davenport. "Improving the Use of Equational Constraints in Cylindrical Algebraic Decomposition". In: *International Symposium on Symbolic and Algebraic Computation* (ISSAC 2015), pp. 165–172. DOI: 10.1145/2755996.2756678.

[EBD+14] Matthew England, Russell Bradford, James H. Davenport, and David Wilson. "Choosing a Variable Ordering for Truth-Table Invariant Cylindrical Algebraic Decomposition by Incremental Triangular Decomposition". In: *International Congress on Mathematical Software* (ICMS 2014). LNCS vol. 8592. DOI: 10.1007/978-3-662-44199-2_68.

[FOS+18] Pascal Fontaine, Mizuhito Ogawa, Thomas Sturm, Van Khanh To, and Xuan Tung Vu. "Wrapping Computer Algebra is Surprisingly Successful for Non-Linear SMT". In: *Satisfiability Checking and Symbolic Computation* (SC^2 2018) at FLoC. CEUR Workshop Proceedings vol. 2189, pp. 110–117. URL: http://ceur-ws.org/Vol-2189/paper3.pdf.

[Fou25] Jean Baptiste Joseph Fourier. "Sur le Calcul des conditions d'inégalité". In: *Nouveau Bulletin des Sciences par la Société philomathique de Paris* (1825), pp. 66–68. URL: https://biodiversitylibrary.org/page/4153058.

[Fou26] Jean Baptiste Joseph Fourier. "Solution d'une question particulière du calcul des inégalités". In: *Nouveau Bulletin des Sciences par la Société philomathique de Paris* (1826), pp. 99–100. URL: https://biodiversitylibrary.org/page/4453516.

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. "MPFR: A Multiple-Precision Binary Floating-Point Library With Correct ounding". In: *ACM Transactions on Mathematical Software* 33 (2 2007). DOI: 10.1145/1236463.1236468.

[Fra20] Hanna Franzen. "Conflict Driven Cylindrical Algebraic Coverings for Nonlinear Arithmetic in SMT Solving". Master's thesis. RWTH Aachen University, 2020.

[Fre95] Jon William Freeman. "Improvements to Propositional Satisfiability Search Algorithms". PhD thesis. Philadelphia, PA, USA: University of Pennsylvania, 1995. URL: https://repository.upenn.edu/dissertations/AAI9532175.

[FGM+07] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. "SAT Solving for Termination Analysis with Polynomial Interpretations". In: *Theory and Applications of Satisfiability Testing* (SAT 2007). LNCS vol. 4501, pp. 340–354. DOI: 10.1007/978-3-540-72788-0_33.

[Fuj16] Matsusaburô Fujiwara. "Über die obere Schranke des absoluten Betrages der Wurzeln einer algebraischen Gleichung". In: *Tohoku Mathematical Journal, First Series* 10 (1916), pp. 167–171. URL: https://www.jstage.jst.go.jp/article/tmj1911/10/0/10_0_167/_article.

[GKC13] Sicun Gao, Soonho Kong, and Edmund M. Clarke. "dReal: An SMT Solver for Nonlinear Theories over the Reals". In: *Automated Deduction* (CADE-24 2013). LNCS vol. 7898, pp. 208–214. DOI: 10.1007/978-3-642-38574-2_14.

[GCL92] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992. DOI: 10.1007/b102438.

[GS96] Fausto Giunchiglia and Roberto Sebastiani. "Building Decision Procedures for Modal Logics from Propositional Decision Procedures – The Case Study of Modal K". In: *Automated Deduction* (CADE-13 1996). LNCS vol. 1104, pp. 583–597. DOI: 10.1007/3-540-61511-3_115.

[Gt19] Torbjörn Granlund and the GMP development team. *The GNU Multiple Precision Arithmetic Library*. 2019. URL: https://gmplib.org.

[Göd31] Kurt Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I". In: *Monatshefte für Mathematik und Physik* 38 (1 1931), pp. 173–198. DOI: 10.1007/BF01700692.

[GSK98] Carla P. Gomes, Bart Selman, and Henry A. Kautz. "Boosting Combinatorial Search Through Randomization". In: *National Conference on Artificial Intelligence* (AAAI 1998), pp. 431–437. URL: https://www.aaai.org/Papers/AAAI/1998/AAAI98-061.pdf.

[GJ17] Stéphane Graham-Lengrand and Dejan Jovanović. "An MCSAT treatment of Bit-Vectors (preliminary report)". In: *Satisfiability Modulo Theories* (SMT 2017) at CAV. URL: https://hal.archives-ouvertes.fr/hal-01615837.

[GV88] D. Yu. Grigor'ev and N.N. Vorobjov. "Solving Systems of Polynomial Inequalities in Subexponential Time". In: *Journal of Symbolic Computation* 5 (1–2 1988), pp. 37–64. DOI: 10.1016/S0747-7171(88)80005-1.

[Gro17] Marta Grobelna. "Solving Pseudo-Boolean Constraints". Bachelor's thesis. RWTH Aachen University, 2017.

[GJ+10] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. 2010. URL: https://eigen.tuxfamily.org.

[Hae18] Rebecca Haehn. "Using Equational Constraints in an Incremental CAD Projection". Master's thesis. RWTH Aachen University, 2018.

[HKÁ18] Rebecca Haehn, Gereon Kremer, and Erika Ábrahám. "Evaluation of Equational Constraints for CAD in SMT Solving". In: *Satisfiability Checking and Symbolic Computation* (SC^2 2018) at FLoC. CEUR Workshop Proceedings vol. 2189, pp. 19–32. URL: http://ceur-ws.org/Vol-2189/paper10.pdf.

[HK96] Bruno Haible and Richard Kreckel. *CLN – Class Library for Numbers*. 1996. URL: https://ginac.de/CLN.

[Hei70] Lee E. Heindel. "Algorithms for exact Polynomial Root Calculation". PhD thesis. University of Wisconsin, 1970. URL: https://search.library.wisc.edu/catalog/999840854302121.

[Hen17] Wanja Hentze. "Computing minimal infeasible subsets for the Cylindrical Algebraic Decomposition". Bachelor's thesis. RWTH Aachen University, 2017.

[HM97] Holly P. Hirst and Wade T. Macey. "Bounding the Roots of Polynomials". In: *The College Mathematics Journal* 28 (4 1997), pp. 292–295. DOI: 10.1080/07468342.1997.11973878.

[Hon90] Hoon Hong. "An Improvement of the Projection Operator in Cylindrical Algebraic Decomposition". In: *International Symposium on Symbolic and Algebraic Computation* (ISSAC 1990), pp. 261–264. DOI: 10.1145/96877.96943.

[Hon91] Hoon Hong. *Comparison of Several Decision Algorithms for the Existential Theory of the Reals*. Research rep. Johannes Kepler University, 1991, pp. 1–33.

[Hua07] Jinbo Huang. "The Effect of Restarts on the Efficiency of Clause Learning". In: *International Joint Conference on Artifical Intelligence* (IJCAI 2007), pp. 2318–2323. URL: http://dl.acm.org/citation.cfm?id=1625275.1625649.

[HEW+14] Zongyan Huang, Matthew England, David Wilson, James H. Davenport, Lawrence C. Paulson, and James Bridge. "Applying Machine Learning to the Problem of Choosing a Heuristic to Select the Variable Ordering for Cylindrical Algebraic Decomposition". In: *Intelligent Computer Mathematics* (CICM 2014), pp. 92–107. DOI: 10.1007/978-3-319-08434-3_8.

[IEE08] IEEE. "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[Irf18] Ahmed Irfan. "Incremental Linearization for Satisfiability and Verification Modulo Nonlinear Arithmetic and Transcendental Functions". PhD thesis. University of Trento, 2018. URL: http://eprints-phd.biblio.unitn.it/2952/.

[Isi95] Alberto Isidori. *Nonlinear Control Systems*. Springer, 1995. DOI: 10.1007/978-1-84628-615-5.

[IYA+09] Hidenao Iwane, Hitoshi Yanami, Hirokazu Anai, and Kazuhiro Yokoyama. "An Effective Implementation of a Symbolic-Numeric Cylindrical Algebraic Decomposition for Quantifier Elimination". In: *Symbolic Numeric Computation* (SNC 2009), pp. 55–64. DOI: 10.1145/1577190.1577203.

[JDF15] Maximilian Jaroschek, Pablo Federico Dobal, and Pascal Fontaine. "Adapting Real Quantifier Elimination Methods for Conflict Set Computation". In: *Frontiers of Combining Systems* (FroCoS 2015). LNCS vol. 9322, pp. 151–166. DOI: 10.1007/978-3-319-24246-0_10.

[JW90] Robert G. Jeroslow and Jinchang Wang. "Solving Propositional Satisfiability Problems". In: *Annals of Mathematics and Artificial Intelligence* 1 (1–4 1990), pp. 167–187. DOI: 10.1007/bf01531077.

[JD17] Dejan Jovanovic and Bruno Dutertre. "LibPoly: A Library for Reasoning about Polynomials". In: *Satisfiability Modulo Theories* (SMT 2017) at CAV. CEUR Workshop Proceedings vol. 1889. URL: http://ceur-ws.org/Vol-1889/paper3.pdf.

[Jov17] Dejan Jovanović. "Solving Nonlinear Integer Arithmetic with MCSAT". In: *Verification, Model Checking, and Abstract Interpretation* (VMCAI 2017). LNCS vol. 10145, pp. 330–346. DOI: 10.1007/978-3-319-52234-0_18.

[JBM13] Dejan Jovanović, Clark Barrett, and Leonardo de Moura. "The Design and Implementation of the Model Constructing Satisfiability Calculus". In: *Formal Methods in Computer-Aided Design* (FMCAD 2013), pp. 173–180. DOI: 10.1109/FMCAD.2013.7027033.

[JM12] Dejan Jovanović and Leonardo de Moura. "Solving Non-linear Arithmetic". In: *Automated Reasoning* (IJCAR 2012). LNCS vol. 7364, pp. 339–354. DOI: 10.1007/978-3-642-31365-3_27.

[Jun12] Sebastian Junges. "On Gröbner Bases in SMT-Compliant Decision Procedures". Bachelor's thesis. RWTH Aachen University, 2012.

[Kar72] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Complexity of Computer Computations* 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9.

[Kor17] Leonard Korp. "SMT-based Planning for Autonomous Robot Fleets". Bachelor's thesis. RWTH Aachen University, 2017.

[Koš16] Marek Košta. "New Concepts for Real Quantifier Elimination by Virtual Substitution". PhD thesis. Saarland University, Saarbrücken, Germany, 2016. DOI: 10.22028/D291-26679.

[KS15] Marek Košta and Thomas Sturm. "A Generalized Framework for Virtual Substitution". In: *arXiv e-prints* (2015). arXiv: 1501.05826.

[Kre13] Gereon Kremer. "Isolating Real Roots Using Adaptable-Precision Interval Arithmetic". Master's thesis. RWTH Aachen University, 2013.

[Kre18] Gereon Kremer. "Computer Algebra and Computer Science". In: *Applications of Computer Algebra* (ACA 2018). Abstract, p. 27. DOI: 10.15304/9788416954872.

[Kre+20a] Gereon Kremer et al. *Computer Arithmetic and Logic library (CArL)*. 2020. URL: https://github.com/smtrat/carl.

[Kre+20b] Gereon Kremer et al. *Satisfiability-Modulo-Theories Real Algebra Toolbox (SMT-RAT)*. 2020. URL: https://github.com/smtrat/smtrat.

[KÁ18] Gereon Kremer and Erika Ábrahám. "Modular strategic SMT solving with SMT-RAT". In: *Acta Universitatis Sapientiae, Informatica* 10 (1 2018), pp. 5–25. DOI: 10.2478/ausi-2018-0001.

[KÁ20] Gereon Kremer and Erika Ábrahám. "Fully Incremental Cylindrical Algebraic Decomposition". In: *Journal of Symbolic Computation* 100 (2020), pp. 11–37. DOI: 10.1016/j.jsc.2019.07.018.

[KÁG19] Gereon Kremer, Erika Ábrahám, and Vijay Ganesh. "On the Proof Complexity of MCSAT". In: *Satisfiability Checking and Symbolic Computation* (SC^2 2019) at SIAM AG. CEUR Workshop Proceedings vol. 2460. URL: http://ceur-ws.org/Vol-2460/paper3.pdf.

[KCÁ16] Gereon Kremer, Florian Corzilius, and Erika Ábrahám. "A Generalised Branch-and-Bound Approach and Its Application in SAT Modulo Nonlinear Integer Arithmetic". In: *Computer Algebra in Scientific Computing* (CASC 2016). LNCS vol. 9890, pp. 315–335. DOI: 10.1007/978-3-319-45641-6_21.

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

[Krü15] Andreas Krüger. "Bitvectors in SMT-RAT and Their Application To Integer Arithmetics". Master's thesis. RWTH Aachen University, 2015.

[Kuk19] Denis Kuksaus. "SMT-basierte Lösung reell-algebraischer Probleme mittels Linearisierung". Bachelor's thesis. RWTH Aachen University, 2019.

[Lag08] Joseph-Louis Lagrange. *Traité de la résolution des équations numériques*. Courcier, 1808. URL: https://gallica.bnf.fr/ark:/12148/bpt6k1042793z.

[Laz94] Daniel Lazard. "An Improved Projection for Cylindrical Algebraic Decomposition". In: *Algebraic Geometry and its Applications*. Springer, 1994. Chap. 29, pp. 467–476. DOI: 10.1007/978-1-4612-2628-4_29.

[LGP+16] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. "Learning Rate Based Branching Heuristic for SAT Solvers". In: *Theory and Applications of Satisfiability Testing* (SAT 2016). LNCS vol. 9710, pp. 123–140. DOI: 10.1007/978-3-319-40970-2_9.

[LGZ+15] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. "Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers". In: *Hardware and Software: Verification and Testing* (HVC 2015). LNCS vol. 9434, pp. 225–241. DOI: 10.1007/978-3-319-26287-1_14.

[LW93] Rüdiger Loos and Volker Weispfenning. "Applying Linear Quantifier Elimination". In: *The Computer Journal* 36 (5 1993). DOI: 10.1093/comjnl/36.5.450.

[Lös18] Christopher Lösbrock. "Implementing an Incremental Solver for Difference Logic". Bachelor's thesis. RWTH Aachen University, 2018.

[Lot18] Henri Lotze. "Automated Optimization in Production Planning". Master's thesis. RWTH Aachen University, 2018.

[Lou18] Ulrich Loup. "On Solving Real-algebraic Formulas in a Satisfiability-modulo-theories Framework". PhD thesis. RWTH Aachen University, 2018. DOI: 10.18154/RWTH-2018-231963.

[LÁ11] Ulrich Loup and Erika Ábrahám. "GiNaCRA: A C++ Library for Real Algebraic Computations". In: *NASA Formal Methods* (NFM 2011). LNCS vol. 6617. DOI: 10.1007/978-3-642-20398-5_41.

[LSC+13] Ulrich Loup, Karsten Scheibler, Florian Corzilius, Erika Ábrahám, and Bernd Becker. "A Symbiosis of Interval Constraint Propagation and Cylindrical Algebraic Decomposition". In: *Automated Deduction* (CADE-24 2013). LNCS vol. 7898, pp. 193–207. DOI: 10.1007/978-3-642-38574-2_13.

[Mah07] Assia Mahboubi. "Implementing the cylindrical algebraic decomposition within the Coq system". In: *Mathematical Structures in Computer Science* 17 (1 2007), pp. 99–127. DOI: 10.1017/S096012950600586X.

[McC84] Scott McCallum. "An Improved Projection Operation for Cylindrical Algebraic Decomposition". PhD thesis. University of Wisconsin-Madison, 1984. URL: https://research.cs.wisc.edu/techreports/1985/TR578.pdf.

[McC85] Scott McCallum. "An Improved Projection Operation for Cylindrical Algebraic Decomposition". In: *European Conference on Computer Algebra* (EUROCAL 1985). LNCS vol. 204, pp. 277–278. DOI: 10.1007/3-540-15984-3_277.

[McC88] Scott McCallum. "An Improved Projection Operation for Cylindrical Algebraic Decomposition of Three-dimensional Space". In: *Journal of Symbolic Computation* 5 (1–2 1988), pp. 141–161. DOI: 10.1016/S0747-7171(88)80010-5.

[McC93] Scott McCallum. "Solving Polynomial Strict Inequalities Using Cylindrical Algebraic Decomposition". In: *The Computer Journal* 36 (5 1993), pp. 432–438. DOI: 10.1093/comjnl/36.5.432.

[McC99] Scott McCallum. "On Projection in CAD-based Quantifier Elimination with Equational Constraint". In: *International Symposium on Symbolic and Algebraic Computation* (ISSAC 1999), pp. 145–149. DOI: 10.1145/309831.309892.

[McC01] Scott McCallum. "On Propagation of Equational Constraints in CAD-based Quantifier Elimination". In: *International Symposium on Symbolic and Algebraic Computation* (ISSAC 2001), pp. 223–231. DOI: 10.1145/384101.384132.

[MH16] Scott McCallum and Hoon Hong. "On using Lazard's projection in CAD construction". In: *Journal of Symbolic Computation* 72 (2016), pp. 65–81. DOI: 10.1016/j.jsc.2015.02.001.

[MPP19] Scott McCallum, Adam Parusiński, and Laurentiu Paunescu. "Validity proof of Lazard's method for CAD construction". In: *Journal of Symbolic Computation* 92 (2019), pp. 52–69. DOI: 10.1016/j.jsc.2017.12.002.

[McC56] Edward J. McCluskey. "Minimization of Boolean functions". In: *The Bell System Technical Journal* 35 (6 1956). DOI: 10.1002/j.1538-7305.1956.tb03835.x.

[MS02] Maurice Mignotte and Doru Stefanescu. "On an estimation of polynomial roots by Lagrange". Research rep. 2002. URL: https://hal.archives-ouvertes.fr/hal-00129675.

[MP14] Michael Monagan and Roman Pearce. "POLY: A New Polynomial Data Structure for Maple 17". In: *Computer Mathematics* 2014, pp. 325–348. DOI: 10.1007/978-3-662-43799-5_24.

[MKC09] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009. DOI: 10.1137/1.9780898717716.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. "Chaff: Engineering an Efficient SAT Solver". In: *Design Automation Conference* (DAC 2001), pp. 530–535. DOI: 10.1145/378239.379017.

[Mot36] Theodor Samuel Motzkin. "Beiträge zur Theorie der Linearen Ungleichungen". PhD thesis. Universität Basel, 1936.

[MB08a] Leonardo de Moura and Nikolaj Bjørner. "Model-based Theory Combination". In: *Electronic Notes in Theoretical Computer Science* 198 (2 2008), pp. 37–49. DOI: 10.1016/j.entcs.2008.04.079.

[MB08b] Leonardo de Moura and Nikolaj Bjørner. "Proofs and Refutations, and Z3". In: *Logic for Programming, Artificial Intelligence and Reasoning* (LPAR Workshops 2008). CEUR Workshop Proceedings vol. 418, pp. 123–132. URL: http://ceur-ws.org/Vol-418/paper10.pdf.

[MB08c] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS 2008). LNCS vol. 4963, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[MJ13] Leonardo de Moura and Dejan Jovanović. "A Model-Constructing Satisfiability Calculus". In: *Verification, Model Checking, and Abstract Interpretation* (VMCAI 2013). LNCS vol. 7737, pp. 1–12. DOI: 10.1007/978-3-642-35873-9_1.

[MP13] Leonardo de Moura and Grant Olney Passmore. "Computation in Real Closed Infinitesimal and Transcendental Extensions of the Rationals". In: *Automated Deduction* (CADE-24 2013). LNCS vol. 7898, pp. 178–192. DOI: 10.1007/978-3-642-38574-2_12.

[MR02] Leonardo de Moura and Harald Rueß. "Lemmas on Demand for Satisfiability Solvers". In: *Theory and Applications of Satisfiability Testing* (SAT 2002). URL: https://leodemoura.github.io/files/sat02.pdf.

[Mül78] Franz Müller. *Ein exakter Algorithmus zur nichtlinearen Optimierung für beliebige Polynome mit mehreren Veränderlichen*. Hain, 1978.

[NDS19] Akshar Nair, James Davenport, and Gregory Sankaran. "On Benefits of Equality Constraints in Lex-Least Invariant CAD". In: *Satisfiability Checking and Symbolic Computation* (SC^2 2019) at SIAM AG. CEUR Workshop Proceedings vol. 2460. URL: http://ceur-ws.org/Vol-2460/paper6.pdf.

[Nal17] Jasper Nalbach. "Embedding the Virtual Substitution in the MCSAT Framework". Bachelor's thesis. RWTH Aachen University, 2017.

[Nal20] Jasper Nalbach. "A novel adaption of the Simplex algorithm for linear real arithetic". Master's thesis. RWTH Aachen University, 2020.

[NKÁ19] Jasper Nalbach, Gereon Kremer, and Erika Ábrahám. "On Variable Orderings in MCSAT for Non-linear Real Arithmetic (extended abstract)". In: *Satisfiability Checking and Symbolic Computation* (SC^2 2019) at SIAM AG. CEUR Workshop Proceedings vol. 2460. URL: http://ceur-ws.org/Vol-2460/paper5.pdf.

[NO79] Greg Nelson and Derek C. Oppen. "Simplification by Cooperating Decision Procedures". In: *ACM Transactions on Programming Languages and Systems* 1 (2 1979), pp. 245–257. DOI: 10.1145/357073.357079.

[NO80] Greg Nelson and Derek C. Oppen. "Fast Decision Procedures Based on Congruence Closure". In: *Journal of the ACM* 27.2 (1980), pp. 356–364. DOI: 10.1145/322186.322198.

[Neu15] Lukas Neuberger. "Generation of Infeasible Subsets in Less-Lazy SMT-Solving for the Theory of Uninterpreted Functions". Bachelor's thesis. RWTH Aachen University, 2015.

[Neu18a] Tom Neuhäuser. "Quantifier Elimination by Cylindrical Algebraic Decomposition". Bachelor's thesis. RWTH Aachen University, 2018.

[Neu18b] Malte Neuß. "Using Single CAD Cells as Explanations in MCSAT-style SMT Solving". Master's thesis. RWTH Aachen University, 2018.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)". In: *Journal of the ACM* 53 (6 2006), pp. 937–977. DOI: 10.1145/1217856.1217859.

[PRS+99] Amir Pnueli, Yoav Rodeh, Ofer Shtrichman, and Michael Siegel. "Deciding Equality Formulas by Small Domains Instantiations". In: *Computer Aided Verification* (CAV 1999). LNCS vol. 1633, pp. 455–469. DOI: 10.1007/3-540-48683-6_39.

[Pre30] Mojżesz Presburger. "Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchen die Addition als einzige Operation hervortritt". In: *Sprawozdanie z I Kongresu Matematików Krajów Słowiańskich (Comptes-rendus du I^{er} Congrès des Mathématiciens des Pays Slaves)* 1930, pp. 92–101 and 395.

[Ren88] James Renegar. "A Faster PSPACE Algorithm for Deciding the Existential Theory of the Reals". In: *Symposium on Foundations of Computer Science* (SFCS 1988), pp. 291–295. DOI: 10.1109/SFCS.1988.21945.

[RKG18] Robert Robere, Antonina Kolokolova, and Vijay Ganesh. "The Proof Complexity of SMT Solvers". In: *Computer Aided Verification* (CAV 2018). LNCS vol. 10982, pp. 275–293. DOI: 10.1007/978-3-319-96142-2_18.

[Ros36] J. Barkley Rosser. "Extensions of some Theorems of Gödel and Church". In: *The Journal of Symbolic Logic* 1.3 (1936), pp. 87–91. DOI: 10.2307/2269028.

[RW10] Philipp Rümmer and Thomas Wahl. "An SMT-LIB Theory of Binary Floating-Point Arithmetic". In: *Satisfiability Modulo Theories* (SMT 2010) at FLoC. URL: http://www.philipp.ruemmer.org/publications/smt-fpa.pdf.

[SYZ18] Mohab Safey El Din, Zhi-Hong Yang, and Lihong Zhi. "On the Complexity of Computing Real Radicals of Polynomial Systems". In: *International Symposium on Symbolic and Algebraic Computation* (ISSAC 2018), pp. 351–358. DOI: 10.1145/3208976.3209002.

[Sag12] Michael Sagraloff. "When Newton Meets Descartes: A Simple and Fast Algorithm to Isolate the Real Roots of a Polynomial". In: *International Symposium on Symbolic and Algebraic Computation* (ISSAC 2012), pp. 297–304. DOI: 10.1145/2442829.2442872.

[Sal18] Ömer Sali. "Linearization Techniques for Nonlinear Arithmetic Problems in SMT". Master's thesis. RWTH Aachen University, 2018.

[SKB13] Karsten Scheibler, Stefan Kupferschmid, and Bernd Becker. "Recent Improvements in the SMT Solver iSAT". In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen* (MBMV 2013). Vol. 13, pp. 231–241.

[Sch13] Stefan Schupp. "Interval Constraint Propagation in SMT Compliant Decision Procedures". Master's thesis. RWTH Aachen University, 2013.

[ST15a] Roberto Sebastiani and Silvia Tomasi. "Optimization Modulo Theories with Linear Rational Costs". In: *ACM Transactions on Computational Logic* 16 (2 2015), 12:1–12:43. DOI: 10.1145/2699915.

[ST15b] Roberto Sebastiani and Patrick Trentin. "OptiMathSAT: A Tool for Optimization Modulo Theories". In: *Computer Aided Verification* (CAV 2015). LNCS vol. 9206, pp. 447–454. DOI: 10.1007/978-3-319-21690-4_27.

[Sei54] Abraham Seidenberg. "A New Decision Method for Elementary Algebra". In: *Annals of Mathematics, Second Series* 60.2 (1954), pp. 365–374. DOI: 10.2307/1969640.

[SS03] Andreas Seidl and Thomas Sturm. "A Generic Projection Operator for Partial Cylindrical Algebraic Decomposition". In: *International Symposium on Symbolic and Algebraic Computation* (ISSAC 2003), pp. 240–247. DOI: 10.1145/860854.860903.

[Sho79] Robert E. Shostak. "A Practical Decision Procedure for Arithmetic with Function Symbols". In: *Journal of the ACM* 26 (2 1979), pp. 351–360. DOI: 10.1145/322123.322137.

[SS96] João P. Marques Silva and Karem A. Sakallah. "GRASP: a New Search Algorithm for Satisfiability". In: *IEEE/ACM International Conference on Computer-aided Design* (ICCAD 1996), pp. 220–227. DOI: 10.1109/ICCAD.1996.569607.

[Sta84] Ryan Stansifer. *Presburger's Article on Integer Arithmetic: Remarks and Translation*. Tech. rep. Cornell University, 1984. URL: https://hdl.handle.net/1813/6478.

[SSB02] Ofer Strichman, Sanjit A. Seshia, and Randal E. Bryant. "Deciding Separation Formulas with SAT". In: *Computer Aided Verification* (CAV 2002). LNCS vol. 2404, pp. 209–222. DOI: 10.1007/3-540-45657-0_16.

[Str00] Adam W. Strzeboński. "Solving Systems of Strict Polynomial Inequalities". In: *Journal of Symbolic Computation* 29 (3 2000), pp. 471–480. DOI: 10.1006/jsco.1999.0327.

[Str14] Adam W. Strzeboński. "Cylindrical Algebraic Decomposition Using Local Projections". In: *39th International Symposium on Symbolic and Algebraic Computation* (ISSAC 2014), pp. 389–396. DOI: 10.1145/2608628.2608633.

[Stu29] Jacques C. F. Sturm. "Analyse d'un Mémoire sur la résolution des équations numériques". In: *Bulletin de Férussac*. 271st ed. Vol. XI. 1829, pp. 419–422. DOI: 10.1007/978-3-7643-7990-2_24.

[Tar51] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. Research rep. RAND Corporation, 1951. URL: https://www.rand.org/pubs/reports/R109.html.

[Tse68] Grigori S. Tseitin. "On the Complexity of Derivation in Propositional Calculus". In: *Studies in Constructive Mathematics and Mathematical Logic* (1968), pp. 115–125. URL: http://www.decision-procedures.org/handouts/Tseitin70.pdf.

[TKO17] Vu Xuan Tung, To Van Khanh, and Mizuhito Ogawa. "raSAT: an SMT solver for polynomial constraints". In: *Formal Methods in System Design* 51 (3 2017), pp. 462–499. DOI: 10.1007/s10703-017-0284-9.

[Vie16] Tarik Viehmann. "Comparing different projection operators in the Cylindrical Algebraic Decomposition for SMT solving". Bachelor's thesis. RWTH Aachen University, 2016.

[VKÁ17] Tarik Viehmann, Gereon Kremer, and Erika Ábrahám. "Comparing Different Projection Operators in the Cylindrical Algebraic Decomposition for SMT Solving". In: *Satisfiability Checking and Symbolic Computation* (SC^2 2017) at ISSAC. CEUR Workshop Proceedings vol. 1974. URL: http://ceur-ws.org/Vol-1974/RP2.pdf.

[Vol15] Matthias Volk. "Using SAT Solvers for Industrial Combinatorial Problems". Master's thesis. RWTH Aachen University, 2015.

[Wei88] Volker Weispfenning. "The Complexity of Linear Problems in Fields". In: *Journal of Symbolic Computation* 5 (1–2 1988), pp. 3–27. DOI: 10.1016/S0747-7171(88)80003-8.

[Wei97] Volker Weispfenning. "Quantifier Elimination for Real Algebra — the Quadratic Case and Beyond". In: *Applicable Algebra in Engineering, Communication and Computing* 8 (2 1997), pp. 85–101. DOI: 10.1007/s002000050055.

[Win16] Tobias Winkler. "Using Thom Encodings for Real Algebraic Numbers in the Cylindrical Algebraic Decomposition". Bachelor's thesis. RWTH Aachen University, 2016.

[WW99] Steven A. Wolfman and Daniel S. Weld. "The LPSAT Engine & its Application to Resource Planning". In: *International Joint Conference on Artificial Intelligence* (IJCAI 1999), pp. 310–316. URL: https://ijcai.org/Proceedings/99-1/Papers/046.pdf.

[Zam19] Aklima Zaman. "Incremental Linearization for SAT Modulo Real Arithmetic Solving". Master's thesis. RWTH Aachen University, 2019.

[Zar65] Oscar Zariski. "Studies in Equisingularity II. Equisingularity in Codimension 1 (and Characteristic Zero)". In: *American Journal of Mathematics* 87.4 (1965), pp. 972–1006. DOI: 10.2307/2373257.

[ZWR16] Aleksandar Zeljić, Christoph M. Wintersteiger, and Philipp Rümmer. "Deciding Bit-Vector Formulas with mcSAT". In: *Theory and Applications of Satisfiability Testing* (SAT 2016). LNCS vol. 9710, pp. 249–266. DOI: 10.1007/978-3-319-40970-2_16.

Index

Publication list

This list contains all other publications that the author of this thesis contributed to. Please see Section 1.2.1 for a summary of the authors contributions to the relevant publications.

[ÁCJ+16] Erika Ábrahám, Florian Corzilius, Einar Broch Johnsen, Gereon Kremer, and Jacopo Mauro. "Zephyrus2: On the Fly Deployment Optimization Using SMT and CP Technologies". In: *Dependable Software Engineering: Theories, Tools, and Applications* (SETTA 2016). LNCS vol. 9984, pp. 229–245. DOI: 10.1007/978-3-319-47677-3_15.

[ÁDE+20] Erika Ábrahám, James H. Davenport, Matthew England, and Gereon Kremer. "Deciding the Consistency of Non-Linear Real Arithmetic Constraints with a Conflict Driven Search Using Cylindrical Algebraic Coverings". In: *arXiv e-prints* (2020). Accepted at Journal of Logical and Algebraic Methods in Programming. arXiv: 2003.05633.

[ÁK16] Erika Ábrahám and Gereon Kremer. "Satisfiability Checking: Theory and Applications". In: *Software Engineering and Formal Methods* (SEFM 2016). LNCS vol. 9763, pp. 9–23. DOI: 10.1007/978-3-319-41591-8_2.

[ÁK17] Erika Ábrahám and Gereon Kremer. "SMT Solving for Arithmetic Theories: Theory and Tool Support". In: *Symbolic and Numeric Algorithms for Scientific Computing* (SYNASC 2017), pp. 1–8. DOI: 10.1109/SYNASC.2017.00009.

[ÁNK17] Erika Ábrahám, Jasper Nalbach, and Gereon Kremer. "Embedding the Virtual Substitution Method in the Model Constructing Satisfiability Calculus Framework". In: *Satisfiability Checking and Symbolic Computation* (SC^2 2017) at ISSAC. CEUR Workshop Proceedings vol. 1974. URL: http://ceur-ws.org/Vol-1974/EAb.pdf.

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. "SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving". In: *Theory and Applications of Satisfiability Testing* (SAT 2015). LNCS vol. 9340, pp. 360–368. DOI: 10.1007/978-3-319-24318-4_26.

[HKÁ18] Rebecca Haehn, Gereon Kremer, and Erika Ábrahám. "Evaluation of Equational Constraints for CAD in SMT Solving". In: *Satisfiability Checking and Symbolic Computation* (SC^2 2018) at FLoC. CEUR Workshop Proceedings vol. 2189, pp. 19–32. URL: http://ceur-ws.org/Vol-2189/paper10.pdf.

[Kre18] Gereon Kremer. "Computer Algebra and Computer Science". In: *Applications of Computer Algebra* (ACA 2018). Abstract, p. 27. DOI: 10.15304/9788416954872.

[KÁ18] Gereon Kremer and Erika Ábrahám. "Modular strategic SMT solving with SMT-RAT". In: *Acta Universitatis Sapientiae, Informatica* 10 (1 2018), pp. 5–25. DOI: 10.2478/ausi-2018-0001.

[KÁ20] Gereon Kremer and Erika Ábrahám. "Fully Incremental Cylindrical Algebraic Decomposition". In: *Journal of Symbolic Computation* 100 (2020), pp. 11–37. DOI: 10.1016/j.jsc.2019.07.018.

[KÁG19] Gereon Kremer, Erika Ábrahám, and Vijay Ganesh. "On the Proof Complexity of MCSAT". In: *Satisfiability Checking and Symbolic Computation* (SC^2 2019) at SIAM AG. CEUR Workshop Proceedings vol. 2460. URL: http://ceur-ws.org/Vol-2460/paper3.pdf.

[KCÁ16] Gereon Kremer, Florian Corzilius, and Erika Ábrahám. "A Generalised Branch-and-Bound Approach and Its Application in SAT Modulo Nonlinear Integer Arithmetic". In: *Computer Algebra in Scientific Computing* (CASC 2016). LNCS vol. 9890, pp. 315–335. DOI: 10.1007/978-3-319-45641-6_21.

[NKÁ19] Jasper Nalbach, Gereon Kremer, and Erika Ábrahám. "On Variable Orderings in MCSAT for Non-linear Real Arithmetic (extended abstract)". In: *Satisfiability Checking and Symbolic Computation* (SC^2 2019) at SIAM AG. CEUR Workshop Proceedings vol. 2460. URL: http://ceur-ws.org/Vol-2460/paper5.pdf.

[VKÁ17] Tarik Viehmann, Gereon Kremer, and Erika Ábrahám. "Comparing Different Projection Operators in the Cylindrical Algebraic Decomposition for SMT Solving". In: *Satisfiability Checking and Symbolic Computation* (SC^2 2017) at ISSAC. CEUR Workshop Proceedings vol. 1974. URL: http://ceur-ws.org/Vol-1974/RP2.pdf.

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2017-01 *	Fachgruppe Informatik: Annual Report 2017
2017-02	Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity
2017-04	Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE
2017-05	Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems
2017-06	Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy
2017-07	Klaus Leppkes, Johannes Lotz, Uwe Naumann, and Jacques du Toit: Meta Adjoint Programming in C++
2017-08	Thomas Gerlitz: Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts
2017-09	Muhammad Hamad Alizai, Jan Beutel, Jó Ágila Bitsch, Olaf Landsiedel, Luca Mottola, Przemyslaw Pawelczak, Klaus Wehrle, and Kasim Sinan Yildirim: Proc. IDEA League Doctoral School on Transiently Powered Computing
2018-01 *	Fachgruppe Informatik: Annual Report 2018
2018-02	Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2018-03	Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices
2018-04	Andreas Ganser: Operation-Based Model Recommenders
2018-05	Matthias Terber: Real-World Deployment and Evaluation of Synchronous Programming in Reactive Embedded Systems
2018-06	Christian Hensel: The Probabilistic Model Checker Storm - Symbolic Methods for Probabilistic Model Checking
2019-01 *	Fachgruppe Informatik: Annual Report 2019
2019-02	Tim Felix Lange: IC3 Software Model Checking
2019-03	Sebastian Patrick Grobosch: Formale Methoden für die Entwicklung von eingebetteter Software in kleinen und mittleren Unternehmen
2019-05	Florian Göbe: Runtime Supervision of PLC Programs Using Discrete-Event Systems
2020-01 *	Fachgruppe Informatik: Annual Report 2020
2020-02	Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla: Towards an Isabelle Theory for distributed, interactive systems - the untimed case

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.