Formally Verifying Algorithms for Real Quantifier Elimination

Katherine Kosaian

CMU-CS-23-130

August 2023

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This material is based upon work supported by the National Science Foundation under Grant No. CNS1739629, a National Science Foundation Graduate Research Fellowship under Grants Nos. DGE1252522 and DGE1745016, and by the AFOSR under grant number FA9550-16-1-0288. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or AFOSR.

Keywords: quantifier elimination, theorem proving, real arithmetic

Sed Deus Dat Incrementum

Abstract

Statements in the first-order logic of real arithmetic ($\text{FOL}_{\mathbb{R}}$) that involve the "there exists" and "for all" quantifiers arise in various application domains, including the formal verification of cyber-physical systems and robot motion planning. These quantifiers are difficult for both humans and computers to handle. The best way of analyzing quantified formulas in $\text{FOL}_{\mathbb{R}}$ is to reduce them to logically equivalent quantifier-free formulas, through a process known as *quantifier elimination (QE)*. By removing the quantifiers in a logically sound manner, QE makes formulas significantly simpler to analyze (as quantifier-free formulas can be easily evaluated by arithmetic in individual states, whereas quantified formulas cannot).

Given the safety-critical nature of applications involving real quantifier elimination, having correct QE algorithms is crucial. For this, formally verifying QE algorithms—by implementing them in a theorem prover and developing associated proofs of correctness—is very desirable. These proofs of correctness are rigorous, as they rely only on the trusted core of the theorem prover—a (typically small) foundation of trusted code/logical axioms from which all other results are built.

This thesis provides formally verified support for real QE with a two-pronged approach: First, develop support for *efficient* even if *incomplete* QE algorithms (which are specialized to a fragment of real arithmetic), with a focus on filling gaps in the existing body of related work. Second, develop support for a *promising* complete QE algorithm with the potential for eventual efficiency / good complexity. For the first goal, the thesis discusses a verification of linear and quadratic *virtual substitution* with a focus on correctness, experimentation, and optimization; the experiments show that this verified VS implementation is competitive with unverified implementations of VS. For the second goal, the thesis discusses the verification of a complete QE algorithm that uses insights from the influential *Ben-Or, Kozen, and Reif (BKR)* algorithm; although this verified algorithm does not currently exploit *all* insights from BKR and does not yet realize practical efficiency, it lays groundwork for eventual verified complete QE algorithms with strong parallel complexity bounds. This thesis uses the theorem prover Isabelle/HOL for both verifications.

Acknowledgments

Many thanks to my advisor, André Platzer, for his unwavering support throughout the PhD. I have greatly appreciated both his technical mentorship and also his encouragement. On a technical level, I did not have much background in logic and verification when starting graduate school, and André helped me to gradually build necessary background knowledge while learning how to identify and pursue research questions that blended well with my mathematical interests. He has also helped me to grow considerably in the areas of technical writing and speaking. On a more personal level, there are many opportunities which I would not have thought to apply to that André has encouraged me to pursue throughout grad school, many of which have been formative.

I am also very grateful to have had some fantastic collaborators throughout my time at CMU. A special thank you to Yong Kiam Tan, who I enjoyed working with closely throughout graduate school. A second special thank you to Matias Scharager—I had the great pleasure of mentoring Matias while he was still an undergrad, and I wish him the best in his PhD journey. Thank you also to Stefan Mitsch and Fabian Immler; I learned a great deal from both of you and hope that we get the chance to collaborate again in the future. My labmates (both present and former) have also been highly supportive; I am particularly grateful to Rose Bohrer for always being willing to answer my many, many technical questions and thus serving as something of a "logic oracle". Thank you also to my former mentors César Muñoz and Aaron Dutle, who I had the privilege of working with in the summer of 2019 during a NASA internship, and to my thesis committee members—Jeremy Avigad, Dexter Kozen, Lawrence Paulson, and Frank Pfenning—for sharing their time and expertise with me.

Beyond this, I am extremely fortunate to have strong support from a large network of family and close friends. Many thanks to my parents, big sister (Kristin, thank you for almost single-handedly ensuring that I have been well-dressed throughout grad school, and for sometimes being willing to suffer through a shark movie!), brother, and wonderful in-laws for their support and encouragement. A special thank you to my wonderful husband, Jack, for his steadfast support (and for never failing to make me laugh), and to our daughter Gabriella for bringing delight to the thesis writing process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Quantified formulas in the first-order logic of real arithmetic (FOL$_\mathbb{R}$) arise in various application domains, including the formal verification of cyber-physical systems (CPS), geometric theorem proving, and stability analysis of models of biological systems [83]. Intuitively, these formulas naturally capture questions such as: "*For all* possible positions that my robot is in, does *there exist* a safe control choice?" The "there exists" and "for all" quantifiers make these statements difficult to analyze, because they range over the real numbers, which are uncountably infinite. It is extremely useful to reduce quantified formulas to logically equivalent quantifier-free formulas through a process known as *quantifier elimination (QE)*. These quantifier-free formulas are significantly simpler to analyze subsequently, because they can be explicitly evaluated by arithmetic in individual *states* (assignments of values to free variables) [70]. If a formula is *fully quantified* (i.e., it has no free variables), QE is able to reduce it to either *True* or *False*; otherwise, QE will identify what conditions on the free variables are needed to make the formula true. Satisfiability and validity questions are fully quantified (after prefixing existential and universal quantifiers, respectively), whereas other applications may involve QE problems with free variables.

Alfred Tarski proved that algorithms for real QE exist [85]. Thus, *in theory*, all it takes to rigorously answer any real arithmetic question is to verify a complete QE procedure for FOL$_\mathbb{R}$ (in particular, FOL$_\mathbb{R}$ validity and satisfiability are decidable by QE and evaluation). However, *in practice*, complete QE algorithms are complicated and the fastest known one, *cylindrical algebraic decomposition (CAD)* [16] is, in the worst case, doubly exponential in the number of variables. In principle, this complexity bound is not bad, since QE is known to be doubly exponential in the number of *quantifier alternations* [23, 24, 91]. However, in practice, layers of optimizations are needed to help CAD realize its potential for efficiency [6, 17, 56], and work to improve CAD has been going on for decades.

As the applications which require QE are safety-critical [69, 70]; it is vital to have efficient trustworthy support for QE to trust the resulting decisions. Unfortunately, QE algorithms are intricate (intuitively, each QE algorithm has to reduce a highly continuous problem, where each quantifier ranges over the uncountable real numbers, to a discrete problem which can be solved computationally), which makes it difficult to implement them correctly. An ideal solution is to *formally verify* QE algorithms by implementing the algorithms in the rigorous logical setting of a *theorem prover* and developing associated proofs of correctness. The theorem prover will enforce that the proofs of correctness only depend on a (typically small) *trusted core*, a code base which consists of the base logical axioms and their basic implementation. This makes the proofs of correctness, and the associated algorithms, highly trustworthy.

However, developing relatively efficient support for unverified QE took decades [7, 82], and as verification is both newer and more difficult than implementation, there is a dearth of formally verified QE algorithms. In practice, this necessitates the use of unverified tools to solve quantifier elimination problems, even in safety-critical settings. For example, as mentioned previously, QE problems naturally (and frequently) arise when analyzing mathematical models of CPS [71, 79]. Currently, the theorem prover KeYmaera X [38], which is designed to formally verify models of CPS (such as planes [12, 45], trains [47], and robots [4]) uses Mathematica/Wolfram Engine and/or Z3 as blackbox solvers for QE. While these are admirable tools, they are unverified, and their use introduces a weak link [34] into what would otherwise be a (fully verified [3]) trustworthy proof.

There are many algorithms that can be used to solve QE problems [66], and to realize scalability, clever combinations of these algorithms are needed. Many of the most efficient QE algorithms are *incomplete*, in that they target specific fragments of real arithmetic, but are not able to solve *all* QE problems.[1] On the other hand, *complete* QE algorithms, which are capable of solving any QE problem, tend to be complex and inefficient. In practice, unverified tools like Mathematica make extensive use of efficient incomplete methods, preprocessing, and heuristics *and* at least one complete QE algorithm (typically CAD) to achieve efficient QE. A recent thesis by Passmore studies (in an unverified setting) combinations of a range of complete and incomplete QE methods with the goal of systematically realizing strong efficiency benefits [66].

1.1 Our Approach

In this thesis, we are interested in verifying QE algorithms in order to attain strong correctness guarantees. In a nutshell, this thesis aims to advance formally verified support for QE by taking *practical* steps towards verifying efficient incomplete QE methods in conjunction with a promising complete QE algorithm. Our approach is to work in the formal setting of a *theorem prover*, a piece of software which is based on a (typically small) trusted core of logical axioms. We write QE algorithms in a theorem prover and provide corresponding proofs of correctness. Theorem provers are designed so that, in principle, everything written in the theorem prover reduces to that core set of logical axioms, which makes everything formalized in a theorem prover highly trustworthy. Our theorem prover of choice is Isabelle/HOL [65, 67]. While selecting a theorem prover is slightly a matter of taste, Isabelle/HOL is excellent for our purposes: it is well-suited for formalizing mathematics and has a large, centralized, and well-maintained collection of existing proof developments available in the Archive of Formal Proofs (AFP), a number of which we build upon in our work[2]; further, the built-in search tool and Sledgehammer [68] provide invaluable automation for discovering existing theorems and for finishing (easy) subgoals in proofs. A

[1]On an intuitive level, it is precisely this specialization to a fragment of real arithmetic which can allow these algorithms to realize considerable practical performance.

[2]In practice, the AFP is very convenient for us, because it allows us to easily find and load relevant prior formalizations. Further, when Isabelle/HOL undergoes an update (currently, this happens approximately once a year), the AFP formalizations are also updated so that they are consistent with the newest version of the tool.

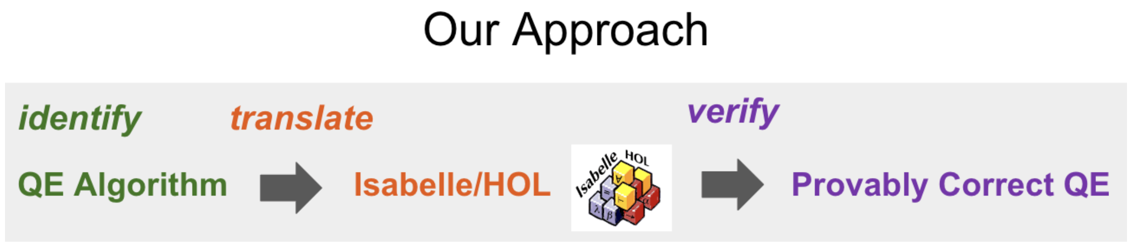


Figure 1.1: A pipeline of our methodology.

pipeline of our approach is visualized in Fig. 1.1.

For any work that seeks to formalize mathematics, there will be challenges on both the mathematical side and the formalization side. On the mathematical side, it is often difficult to write mathematics and mathematical proofs in a theorem prover; specifying details that were initially underspecified and fleshing out proofs can take considerable creativity. On the formalization side, there are often considerable low-level details that must be made explicit to the theorem prover, which can require fleshing out gaps in existing libraries, and particularly intricate pieces of proof may require significant time and effort.

Since working in a verified setting poses considerable challenges compared to an unverified setting (namely, by specifying that all algorithms must be completely logically precise and by enforcing that all results must be accompanied by rigorous correctness proofs), we presently limit our attention to verifying a narrow range of QE methods, with the hope that future works will augment this range.

First, we verify the linear and quadratic cases of *virtual substitution (VS)* [91, 93]. Linear and quadratic VS have considerable practical significance [66] and, correspondingly, are widely used in QE implementations in unverified software, including Mathematica [95], Maple [89, 96], and Redlog [30]. Although these methods are incomplete, as they target problems that contain low-degree polynomial inequalities or equations, they are very impactful. To demonstrate the practical potential of our verified algorithms, we test our verified VS implementation on benchmarks from the literature, comparing to several unverified tools, and show promising performance. In the course of this formalization, there were challenges on both the mathematical side and on the formalization side, but most of the challenges were on the *formalization* side. Intuitively, because the VS algorithm is already highly syntactic, writing it in a formal setting is rather natural, which somewhat lessens the mathematical challenge. However, as we will discuss in more detail later, this work required developing a framework for expressing multivariate QE problems in Isabelle/HOL, and we also had to spend considerable effort developing Isabelle/HOL's multivariate polynomials libraries.

We then verify a complete multivariate QE algorithm that incorporates insights from the *Ben-Or, Kozen, and Reif (BKR)* algorithm [2]. The BKR algorithm was an influential algorithm with good potential for parallelism, and it gave rise to a number of variant QE algorithms by Renegar [75], Canny [10], and others with compelling parallel complexity bounds. In the process of our multivariate formalization, we formalize univariate BKR and the univariate case of Renegar, which are of independent interest, as the insights from univariate BKR have been foundational in a number of later algorithms [9, 59]. In the multivariate case, our work does not currently use

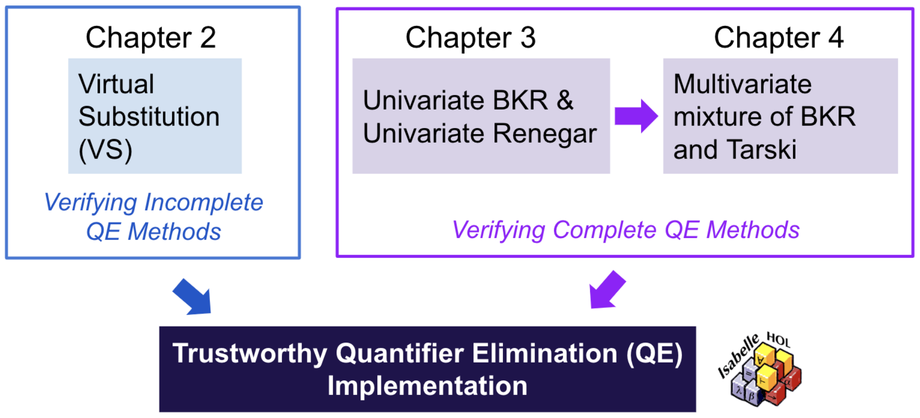

Chapter 2
Virtual Substitution (VS)
Verifying Incomplete QE Methods
Chapter 3
Univariate BKR & Univariate Renegar
Chapter 4
Multivariate mixture of BKR and Tarski
Verifying Complete QE Methods
Trustworthy Quantifier Elimination (QE) Implementation
Isabelle HOL

Figure 1.2: Thesis overview

all of the insights from BKR; rather, we blend theoretical insights from BKR and Tarski's original QE algorithm [85]. To our knowledge, this is the first complete multivariate QE algorithm formalized in Isabelle/HOL, and although we trade off some efficiency for ease of formalization, it is a significant step towards formalized QE algorithms that have good complexity. In the course of this formalization, there were challenges on both the mathematical side and on the formalization side, but most of the challenges were on the *mathematical* side. Intuitively, this can be viewed as being somewhat emblematic of the gap between Theory A research and Theory B research, which my thesis seeks to bridge. The BKR result is a complexity result in the Theory A community that uses sophisticated mathematics; it contains many brilliant ideas but is not immediately well-set-up for formalization, so it is unsurprising that it takes many extra steps to turn it into a formal artifact in the Theory B community. This effort required first turning BKR's mathematical arguments into a recursive function in Isabelle/HOL, then deciding what correctness lemmas to state in order to break apart the proof into manageable pieces, and, finally, fully proving those lemmas.

Finally, to get a unified QE implementation, we link together our verified VS implementation with our verified complete QE algorithm. This allows us to simultaneously have both completeness and the efficiency of virtual substitution.

This thesis is organized into three main technical sections (cross-reference Fig. 1.2). The first discusses our formalization of linear and quadratic virtual substitution. The second discusses our formalization of *univariate* BKR and *univariate* Renegar. The third discusses our formalization of a complete *multivariate* QE algorithm. In the remainder of this introduction, we give a broad overview of related work (related works that are specific to one chapter will be discussed at greater length therein), briefly introduce VS and BKR, and motivate our consideration of each.

1.2 Related Work

Given the safety-critical nature of real arithmetic questions, it is not surprising that considerable attention has been given to formally verifying algorithms for real QE [11, 15, 40, 54, 55, 58, 61, 63, 64, 72]. A variety of works have focused on verifying incomplete QE methods [11, 40, 64, 72], though not always with a focus on practical efficiency. Further, good progress has been made on verifying the *univariate* cases of complete QE algorithms (which can solve QE problems that only involve one variable, and so de facto have at most one quantifier) [54, 61, 63]. Additionally, some progress [15, 55, 58] has been made on fully verifying *complete* multivariate QE algorithms (i.e., algorithms that are capable of resolving *any* real QE problem). We now discuss each of these general categories in more detail.

Incomplete Methods

Regarding incomplete QE methods, virtual substitution (VS) is a particularly well-known efficient algorithm which targets problems with low-degree polynomials. It is natural, then, that previous work has considered formalizing the linear [64] and *quadratic equality* [11] cases of VS. As discussed in more detail in Sect. 2.1, for various reasons we choose not to directly build upon these works in our development (which also formalizes the *quadratic inequality* case of VS, which is considerably more intricate than the equality case, since it requires reasoning not just about roots of quadratic polynomials, but also about points in ranges between roots, of which there are infinitely many).

Other work, by Harrison [40], has considered formalizing incomplete QE methods for the *universal* fragment of real arithmetic (that is, for QE problems that are fully universally quantified) that employ sum-of-squares methods. The basic intuitive idea is that if a polynomial can be decomposed into a sum of polynomial squares, then it is nonnegative, though Harrison also formalizes more general sum-of-squares methods. These include, for example, a way of establishing the inconsistency of a set of polynomial equations and inequalities based on Hilbert's Nullstellensatz (and variations thereof) [40].

There has also been some prior work by Platzer et al. on comparing verified and unverified QE methods which took an experimental focus [72]. This work compared (on 97 benchmarks) unverified complete QE methods in tools including Mathematica, an unverified implementation of virtual substitution that defaults to an optimized version of CAD in Redlog, a verified proof-producing procedure for QE by McLaughlin and Harrison [58] (which we will discuss later in more detail), and both verified and unverified incomplete methods for the universal fragment of real arithmetic, including Harrison's [40] and a new QE method[3] which combines ideas from the real Nullstellensatz with Gröbner bases methods. This experimental approach for evaluating verified QE methods adds a useful practical lens to the literature, especially because some of the methods that were tested had not previously been extensively benchmarked. We take a similarly practical focus in our work on verified virtual substitution, where we do extensive experiments to test the effectiveness of our formalization on real-world benchmarks (see Sect. 2.4).

Univariate Methods

Regarding univariate methods, several univariate cases of complete QE algorithms have been verified independently of their multivariate counterparts. In Isabelle/HOL, Li, Passmore, and

[3]An interesting theoretical contribution of Platzer et al. [72] was identifying a simple set of proof rules that are complete for universal real arithmetic. These proof rules require identifying a useful witness for each QE problem, and Platzer et al.'s new QE method poses a strategy to do this.

Paulson [54] formalized an efficient univariate decision procedure based on univariate cylindrical algebraic decomposition (CAD). Their decision procedure is highly optimized and makes use of Mathematica as an *untrusted oracle*, meaning that Mathematica is used to (efficiently) perform certain computations but that the results are then checked within Isabelle/HOL's trusted core; it achieves strong performance on a set of microbenchmarks [54].

Additionally, two univariate decision procedures have been verified in the theorem prover PVS: `hutch` [61], which is based on univariate CAD, and `tarski` [63], which is based on the Sturm-Tarski theorem and thus shares some conceptual overlap with Tarski's original QE procedure. Originally, `tarski` was limited to the existential conjunctive fragment of universal real arithmetic, but some of my work extended it into a full decision procedure [18]. As expected, `hutch` is typically faster than `tarski`, since it uses a more efficient algorithm; notably, however, `tarski` sometimes outperforms `hutch` [18]. This is likely because the mathematical underpinnings of `hutch` and `tarski` are different, leading to different efficiency tradeoffs. This suggests that, in addition to a range of efficient preprocessing methods, it is desirable to have multiple formalized QE algorithms—problems that are adversarial for one strategy may be resolved more quickly by another strategy.

Complete Multivariate Methods

Unsurprisingly, complete multivariate QE algorithms are significantly more challenging than their univariate counterparts. Multivariate polynomials are unlike univariate polynomials, because they may have infinitely many roots, their leading coefficients are polynomials in fewer variables and so may have zeros, and polynomial division is not always unique. Additionally, whereas univariate QE problems only involve a single quantifier and always reduce to True or False, multivariate QE problems can involve nested quantifier alternations and free variables.

To our knowledge, the main published progress on verifying complete multivariate QE algorithms in theorem provers is threefold: first, Mahboubi [55] *implemented* (but did not yet verify) the fastest-known complete multivariate QE algorithm, CAD [16], in Coq; second, McLaughlin and Harrison developed a *proof-producing* (but not verified) procedure based on the Cohen-Hörmander algorithm in HOL Light [58]; finally, Cohen and Mahboubi verified Tarski's original QE algorithm in Coq [13, 15]. Unfortunately, both Tarski's original QE algorithm and the Cohen-Hörmander algorithm have non-elementary complexity (i.e. the complexity is not bounded by any tower of powers of two) [1, p. 444]. While McLaughlin and Harrison's procedure can solve simple microbenchmarks, they acknowledge considerable experimental limitations [58].[4] Similarly, Cohen and Mahboubi consider their work to be primarily a theoretical contribution [15].

The dearth of efficient formally-verified complete QE algorithms is in part a consequence of the intricacy of the algorithms themselves. Indeed, we observe a tradeoff between the computational efficiency of an algorithm and the tractability of verification [76]. Most notably, the multivariate CAD algorithm is efficient but complex and tremendously difficult to verify, even though the (significantly simpler) univariate case has been independently verified both in Isabelle/HOL [54] and PVS [61]. Further, in order for CAD to realize its full potential for efficiency, many

[4]This is not only due to the complexity of the Cohen-Hörmander algorithm, but also because proof-producing algorithms are not verified once and for all but, instead, have to produce a new proof of correctness per question, which incurs significant overhead compared to fully verified ones [58, 72].

further insights [6, 17, 32, 56] beyond the original development [16] are needed, and improving CAD (and algorithms for real QE at large) is an active area of research [5, 8, 36, 62].

The lack of efficient *verified* QE methods is also a consequence of the challenge posed by verification. Working within the formal setting of a theorem prover adds a considerable layer of rigor but also intricacy, which is why even small progress needs significant effort. For example, Mahboubi [55] discusses the many challenges involved in implementing CAD in Coq—a significantly more arduous and involved task than implementing CAD in an unverified computer algebra system (which also took decades [7, 82]).

Why BKR?

This thesis identifies a potential *sweet spot* within the tradeoff between complexity and verification amenability [76] by focusing on verifying a *complete* multivariate QE algorithm with insights from the *Ben-Or, Kozen, and Reif (BKR)* algorithm [2]. The BKR algorithm was an influential algorithm with good potential for parallelism. It has some theoretical similarity with Tarski's original QE method [85], but BKR includes important insights that lead to better efficiency / complexity. One of its key insights is a clever recursive procedure that computes the set of all consistent sign assignments for an input set of univariate polynomials while carefully managing intermediate steps to avoid exponential blowup from naively enumerating all possible sign assignments (this insight is fundamental for both the univariate case and the multivariate case). Using this insight, the univariate BKR algorithm has good parallel complexity—when optimized, it is an NC algorithm (that is, it runs in parallel polylogarithmic time) [2].

Since multivariate BKR seems to rely fairly directly on the univariate version, we hope that it will be significantly easier to formally verify than multivariate CAD, which is highly complicated. However, it is unlikely that multivariate BKR will be as efficient as CAD in the average case. While BKR states that their multivariate algorithm is computable in parallel exponential time (or in NC for fixed dimension), Canny later found an error in BKR's analysis of the multivariate case [10], which highlights the subtlety of the algorithm and the role for formal verification. Various variants [10, 22, 75] of the BKR algorithm were developed to fix this error; these often achieve highly compelling parallel complexity bounds.[5] Notwithstanding this, it seems highly likely that multivariate BKR will outperform Tarski's original QE algorithm (when its inherent parallelism is exploited), and a highly optimized parallel implementation of BKR, or perhaps of its variants, could (ideally) potentially supplement an eventual formalization of multivariate CAD.

Indeed, as previously noted, it is desirable to have a variety of formally verified decision procedures for arithmetic since different strategies can have different efficiency tradeoffs on different classes of problems [18, 72]. Toward this end, it is beneficial that BKR has a fundamentally different working principle than CAD; like the Cohen-Hörmander procedure, it represents roots

[5]As previous work [42] has drawn a strong distinction between computational complexity and practical efficiency (with particular attention to Renegar [75]), these complexity bounds will not necessarily translate into immediate practical efficiency. However, a followup work [41] argued for the potential of algorithms with strong theoretical complexity bounds to realize efficiency on fragments of real arithmetic. As a promising example of this, in his 2008 thesis, Huntington [43] implemented an experimentally promising algorithm for the existential fragment of real arithmetic based on Canny's variant of BKR. Overall, BKR-based algorithms remain influential.

and sign-invariant regions abstractly, instead of via the computationally expensive real algebraic numbers required in CAD.

Chapter 2

Verifying Virtual Substitution

This chapter discusses the formal verification of linear and quadratic *virtual substitution (VS)* due to Weispfenning [91, 93], which focuses on QE for a quantified variable x occurring in polynomials of at most degree 2 in x, although variations [50, 92] can handle higher degree polynomials. Linear and quadratic VS are of practical significance. They serve to improve QE [66] and SMT tools and are the basis of the experimentally successful [84] Redlog solver [30]. To our knowledge, ours is the first formally verified VS algorithm which includes the intricate quadratic inequality case. We are also unique in that we explicitly focus on achieving promising empirical performance of our verified VS implementation.

As we focus on correct and practical VS, we export our verified Isabelle/HOL code to SML for experimentation. We test our exported formalization of the equality VS algorithm (Sect. 2.2.2) and of the general VS algorithm (Sect. 2.2.3). We compare to four tools that implement real QE: Redlog, SMT-RAT [21], Z3 [25], and Wolfram Engine. With 304 examples, we solve more examples than SMT-RAT in quantifier elimination mode (solves 191) and come close to virtual substitution in Wolfram Engine (solves 322). The remaining tools solve almost all examples; this is to be expected given that those tools have been optimized and fine-tuned (some for decades) and use efficient complete fallback QE algorithms when VS does not succeed. However, as we found 137 inconsistencies in other solvers, it is significant that ours is the only VS implementation with associated correctness proofs (assuming the orthogonal challenge of correct code generation from Isabelle [44]).

The formalization is approximately 23,000 lines in Isabelle/HOL and is available on the Archive of Formal Proofs (AFP) [78].

Collaborators. The material in this chapter is based upon joint work with Matias Scharager, Stefan Mitsch, and André Platzer which appeared at FM 2021 [76]. Fabian Immler also assisted with augmenting the polynomial libraries of Isabelle/HOL during his time at CMU; unfortunately, his subsequent industry position precluded our ability to include him as a coauthor on the final paper.

2.1 Related Work

There has already been some work on formally verified VS: Nipkow [64] formally verified a VS procedure for *linear* equations and inequalities. The building blocks of $\mathrm{FOL}_{\mathbb{R}}$ formulas, or "atoms," in Nipkow's work only allow for linear polynomials $\sum_i a_i x_i \sim c$, where $\sim \in \{=, <\}$, the x_i's are quantified variables and c and the a_i's are real numbers. These restrictions ensure that linear QE can always be performed, and they also simplify the substitution procedure and associated proofs. Nipkow additionally provides a generic framework that can be applied to several different kinds of atoms (each new atom requires implementing several new code theorems in order to create an exportable algorithm). While this is an excellent theoretical framework—we utilize several similar constructs in our formulation—we create an independent formalization that is specific to general $\mathrm{FOL}_{\mathbb{R}}$ formulas, as our main focus is to provide an efficient algorithm in this domain. Specializing to one type of atom allows us to implement several optimizations, such as our modified DNF algorithm, which would be unwieldy to develop in a generic setting.

Chaieb [11] extends Nipkow's work to quadratic equalities. His formalizations are not publicly available, but he generously provided us with the code. While this was helpful for reference, we chose to build on a newer Isabelle/HOL polynomial library, and we focus on VS as an exportable standalone procedure, whereas Chaieb intrinsically links VS with an auxiliary QE procedure.

Other related work includes some unverified solvers. For example, some work has been done in constraint solving with falsification: RSolver [74] was designed for hybrid systems verification and can find concrete counterexamples for fully quantified existential QE problems on *compact* domains; notably, RSolver considers a system to be unsafe if it is *close* to an unsafe system. dReal [39] is based on similar ideas and slightly relaxes the notion of satisfiability to δ-satisfiability, where a formula is considered false if a small numerical perturbation of the formula would render it false. Constraint solving has also been considered in SMT-solving with Z3's nlsat [46], which uses CDCL to decide systems of nonlinear inequalities and equations.

2.2 The Virtual Substitution Algorithm

Informally (and broadly) speaking, VS discretizes the QE problem by solving for the roots of one or more low-degree polynomials $f_1(x), \ldots, f_n(x)$. VS focuses on these roots and the intervals around them to identify and substitute appropriate representative "sample points" for x into the rest of the formula. However, these sample points may contain fractions, square roots, and/or other extensions of the logical language, and so they must be substituted "virtually": That is, VS creates a formula *in $\mathrm{FOL}_{\mathbb{R}}$ proper* that models the behavior of the direct substitution, which would be outside of $\mathrm{FOL}_{\mathbb{R}}$. VS applies in two cases: an equality case and a general case. We formalize both, and discuss each in turn.

Remark 1. *The VS algorithms need to work for* multivariate *polynomials. But as the VS correctness proofs show the equivalence is true for every real value of the free variables, they often implicitly treat all but one variable as having fixed (but arbitrary) real values. That is why most correctness lemmas (but not the top-level algorithmic constructions) suffice for* univariate *polynomials with* real coefficients. *We utilize this trick to simplify difficult proofs for general VS.*

2.2.1 Example

Example 1. *Say that we want to perform QE on the formula* $\exists x.(x^2 = 2 \wedge xy^2 + 2y + 1 = 0)$*. One might notice that* $x^2 = 2$ *forces* $x = \pm\sqrt{2}$ *and accordingly wish to substitute. Direct substitution yields the following expression:* $(\sqrt{2}y^2 + 2y + 1 = 0 \vee -\sqrt{2}y^2 + 2y + 1 = 0)$*. However, as its mention of the* $\sqrt{\cdot}$ *operator makes it an illegal FOL*$_\mathbb{R}$ *formula, we will need some further tricks.*

Cleverly, VS finds that $\sqrt{2}y^2 + 2y + 1 = 0$ *is logically equivalent to* $y^2 \cdot (2y + 1) \leq 0 \wedge 2y^4 - (2y + 1)^2 = 0$*, which is a FOL*$_\mathbb{R}$ *formula.*[1] *Similarly, VS identifies a FOL*$_\mathbb{R}$ *formula that is logically equivalent to* $-\sqrt{2}y^2 + 2y + 1 = 0$*. Then, VS returns the following quantifier-free FOL*$_\mathbb{R}$ *formula which is logically equivalent to* $\exists x.(x^2 = 2 \wedge xy^2 + 2y + 1 = 0)$*:*

$$\begin{aligned}&\big((y^2 \cdot (2y + 1) \leq 0 \wedge 2y^4 - (2y + 1)^2 = 0)\\ &\vee (-y^2 \cdot (2y + 1) \leq 0 \wedge 2y^4 - (2y + 1)^2 = 0)\big).\end{aligned}$$

Remark 2. *If instead our starting formula were* $\exists x.\exists y.(x^2 = 2 \wedge xy^2 + 2y + 1 = 0)$*, where now* y *is quantified, then (following the same method as above) VS would identify the following logically equivalent FOL*$_\mathbb{R}$ *formula with fewer variables:*

$$\begin{aligned}\exists y.&\big((y^2 \cdot (2y + 1) \leq 0 \wedge 2y^4 - (2y + 1)^2 = 0)\\ &\vee (-y^2 \cdot (2y + 1) \leq 0 \wedge 2y^4 - (2y + 1)^2 = 0)\big).\end{aligned} \tag{2.1}$$

Unfortunately, here we are left with a quantified formula with no linear or quadratic equations or inequalities. As we are thus outside of the fragment of FOL$_\mathbb{R}$ *that standard VS applies to, at this point we would want to outsource* (2.1) *to a complete QE algorithm (such as CAD, or the algorithm described in Chapter 4 of this thesis) to eliminate the quantifier on* y*.*

Example 1 was relatively simple, because it involved a quadratic equation with constant coefficients for x. However, nothing in our reasoning was limited to constant coefficients: To perform QE on $\exists x.(x^2 = c \wedge xy^2 + 2y + 1 = 0)$, where c is a polynomial in the variable z, we could handle substituting $x = \pm\sqrt{c}$ in the exact same way as for $x = \pm\sqrt{2}$, but the answer must distinguish the case of $c \geq 0$ symbolically. More difficult is the generalization to inequalities, which seemingly require uncountably infinitely many values to be virtually substituted. We first turn to the general equality case, and then discuss inequalities.

2.2.2 Equality Virtual Substitution Algorithm

Let a, b and c be arbitrary polynomials with real coefficients that do not mention the variable x. Consider the formula $\exists x.(ax^2 + bx + c = 0 \wedge F)$. There are three possible cases: Either $a \neq 0$, or $a = 0$ and b is nonzero, or all of a, b, c are zero (so $ax^2 + bx + c = 0$ is uninformative). Letting F_x^r denote the substitution of $x = r$ for x in F, and solving for the roots of $ax^2 + bx + c$, we have

[1]Notice that if $y = 0$, then both $\sqrt{2}y^2 + 2y + 1 = 0$ and $y^2 \cdot (2y + 1) \leq 0 \wedge 2y^4 - (2y + 1)^2 = 0$ are false. If instead $y \neq 0$, then $\sqrt{2}y^2 + 2y + 1 = 0$ is true exactly when $\sqrt{2} = -(2y + 1)/y^2$, or exactly when $-(2y+1)/y^2 \geq 0 \wedge 2y^4 - (2y+1)^2 = 0$, which is logically equivalent to $y^2 \cdot (2y+1) \leq 0 \wedge 2y^4 - (2y+1)^2 = 0$, as desired.

the following:

$$\begin{aligned}
&\exists x.(ax^2 + bx + c = 0 \land F) \longleftrightarrow \\
&\Big((a = 0 \land b = 0 \land c = 0 \land \exists x.F) \lor \\
&\ (a = 0 \land b \neq 0 \land F_x^{-c/b}) \lor \\
&\ (a \neq 0 \land b^2 - 4ac \geq 0 \land (F_x^{(-b+\sqrt{b^2-4ac})/(2a)} \lor F_x^{(-b-\sqrt{b^2-4ac})/(2a)}))\Big).
\end{aligned}$$

Conditions such as $b^2 - 4ac \geq 0$ are needed to ensure $(-b \pm \sqrt{b^2 - 4ac})/(2a)$ are well-defined; these are symbolic formulas unless a, b, c are concrete numbers.

Similarly as in Example 1, if we substituted $F_x^{-c/b}$, $F_x^{(-b+\sqrt{b^2-4ac})/(2a)}$, and $F_x^{(-b-\sqrt{b^2-4ac})/(2a)}$ directly (for polynomials a, b, and c that do not involve x), the resulting formula would no longer be in $\text{FOL}_{\mathbb{R}}$. Instead, VS avoids directly dividing polynomials or taking square roots with equivalent rewritings in $\text{FOL}_{\mathbb{R}}$. This involves two procedures: one for fractions, and one for square roots.

To virtually substitute a fraction p/q of polynomials where $q \neq 0$ into the atom $\sum_{i=0}^{n} a_i x^i \sim 0$, where $\sim \in \{=, <, \leq, \neq\}$ and each a_i is an arbitrary polynomial expression not involving x, it suffices to normalize the denominator of the LHS, with the caveat that we must not flip the direction of the inequality for $<$ and $\leq$ atoms by normalizing by a value that might be negative. When n is even, $q^n \geq 0$ under any possible valuation, so normalizing by q^n does not flip the inequality. Alternatively, if n is odd, then $q^{n+1} \geq 0$, so we can normalize by q^{n+1} without flipping the inequality. We formalize this in our `linear_substitution` function (see Appendix A.1.1).

Next, we consider substituting $x = \sqrt{c}$ into an atom $\sum_{i=0}^{n} a_i x^i \sim 0$, where c is an arbitrary polynomial expression not involving x that satisfies $c \geq 0$, each a_i is an arbitrary polynomial expression not involving x, and $\sim \in \{=, <, \leq, \neq\}$. Its direct substitution can be separated out into even and odd exponents:

$$\sum_{i=0}^{n} a_i \cdot (\sqrt{c})^i = \sum_{i=0}^{n/2} a_{2i} c^i + \sum_{i=0}^{n/2} a_{2i+1} c^i \sqrt{c}$$

Now our polynomial has the form $A + B\sqrt{c}$, where A and B and c are symbolic polynomial expressions not involving x. Then, we have the following cases:

$$\begin{aligned}
&A + B\sqrt{c} = 0 \longleftrightarrow AB \leq 0 \land A^2 - B^2c = 0 \\
&A + B\sqrt{c} < 0 \longleftrightarrow (A < 0 \land B^2c - A^2 < 0) \lor (B \leq 0 \land (A < 0 \lor A^2 - B^2c < 0)) \\
&A + B\sqrt{c} \leq 0 \longleftrightarrow (A \leq 0 \land B^2c - A^2 \leq 0) \lor (B \leq 0 \land A^2 - B^2c \leq 0) \\
&A + B\sqrt{c} \neq 0 \longleftrightarrow -AB < 0 \lor A^2 - B^2c \neq 0
\end{aligned}$$

The equivalences for $=$ and $\neq$ atoms are derived from the observation that if $B \neq 0$, $A + B\sqrt{c} = 0$ can be solved to find $\sqrt{c} = -A/B$, which holds iff $A^2 = B^2c$ and $-A/B \geq 0$. The inequality cases involve casework to determine when polynomial A is negative and dominates $B\sqrt{c}$ as $A^2 > B^2c$, and when B is negative and $B\sqrt{c}$ dominates A as $B^2c > A$. We formalize the VS procedure for quadratic roots in `quadratic_sub` (see Appendix A.1.2).

2.2.3 General Virtual Substitution Algorithm

As we have seen, QE very naturally leads to finitely many cases (discretizes) for formulas that involve quadratic equality atoms (we call this the *equality case*). The VS algorithm for the *general case*, which also handles inequality atoms, is more involved, because, unlike equalities, inequalities may have uncountably many solutions. General VS only directly applies to a very specific fragment of $\text{FOL}_{\mathbb{R}}$ formulas: conjunctions of polynomials that are at most quadratic in the variable of interest. However, we can extend general VS to apply to more formulas with the help of a disjunctive normal form (DNF) transformation.

As a simple example, consider the formula $\exists x.(p < 0 \land q < 0)$, where p and q are the univariate quadratic polynomials (in variable x) depicted in Fig. 2.1. Noting that the roots of p and q cannot possibly satisfy the strict inequalities, we partition the number line into ranges in between these zeros.

We recognize a key property: In each of the ranges between the roots of p, q, the signs of both p and q do not change. Since the ranges cover all roots of p, q, the truth value of the formula at a single point in a range is representative of the truth value of the formula on the entire range. To discretize the QE problem, we need only pick one sample point for each range.

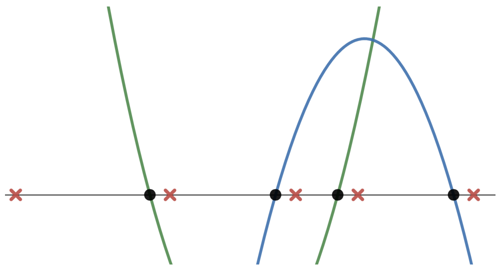


Figure 2.1: Two quadratics, their roots (black dots) and off-roots (red x's)

However, we want to pick appropriate sample points *for any* possible p and q. The points we pick as representatives are called the off-roots, which occur ϵ units away from the roots, where $\epsilon > 0$ is arbitrarily small. We additionally need a representative for the leftmost range, which we represent with the point $-\infty$, where $-\infty$ is arbitrarily negative. Of course, we cannot directly substitute ϵ and $-\infty$: they are not real numbers! However, we can *virtually* substitute them.

Negative Infinity

Given any formula F, the VS of $-\infty$ should satisfy the equivalence $F_x^{-\infty} \longleftrightarrow \exists y.\, \forall x{<}y.\, F(x)$ (where y does not occur in F). Intuitively, this says that $-\infty$ acts as if it is arbitrarily negative (and thus less than the x component of all roots of the polynomials in F) and captures information for the leftmost range on the real number line in any valuation of the non-x variables. If formula $\exists y.\, \forall x{<}y.\, ax^2 + bx + c = 0$ is true, where a, b, c are polynomials that do not involve x, then $ax^2 + bx + c = 0$ holds at infinitely many x; since nonzero polynomials have finitely many roots, this can only happen if $ax^2 + bx + c$ is the zero polynomial in x, i.e., it holds that:

$$(ax^2 + bx + c = 0)_x^{-\infty} \longleftrightarrow a = 0 \land b = 0 \land c = 0 \qquad (2.2)$$

The negation of (2.2) captures the behavior of $\neq$ atoms. For $<$ atoms, note that the sign value at $-\infty$ is dominated by the leading coefficient, so:

$$(ax^2 + bx + c < 0)_x^{-\infty} \longleftrightarrow a < 0 \lor (a = 0 \land (b > 0 \lor (b = 0 \land c < 0)))$$

Finally, $(ax^2 + bx + c \leq 0)_x^{-\infty} \longleftrightarrow (ax^2 + bx + c = 0)_x^{-\infty} \lor (ax^2 + bx + c < 0)_x^{-\infty}$.

In Isabelle/HOL, we formalize that our virtual substitution of $-\infty$ satisfies the desired equivalence (on $\mathbb{R}$ using Remark 1) in the following lemma:

```
lemma infinity_evalUni: shows "(∃y. ∀x<y. aEvalUni At x) =
  (evalUni (substNegInfinityUni At) x)"
```

To explain this lemma, we need to take a slight detour and discuss a few structural details of our framework (which is discussed in greater detail in Sect. 2.3). The datatype `atomUni` contains a triple of real numbers (which represent the coefficients of a univariate quadratic polynomial) and a sign condition:

```
datatype atomUni = LessUni "real*real*real" | EqUni "real*real*real"
  | LeqUni "real*real*real" | NeqUni "real*real*real"
```

The `aEvalUni` function has type `atomUni ⇒ real ⇒ bool`; that is, it takes a sign condition with a triple of real numbers (a, b, c) and a real number x and evaluates whether $ax^2 + bx + c$ satisfies the sign condition. The `evalUni` function has type `atomUni fmUni ⇒ real ⇒ bool`, where an `atomUni fmUni` is a formula that involves conjunctions and disjunctions of elements of type `atomUni` (and "True" and "False"). That is, the `evalUni` function takes such a formula and a real number and evaluates whether the formula is true at the real number. Thus, `infinity_evalUni` states that, given `At` of type `atomUni`, with tuple (a, b, c) and sign condition $\sim\, \in \{<, =, \leq, \neq\}$, $\mathtt{At}_x^{-\infty}$ holds iff $\exists y. \forall x{<}y. ax^2 + bx + c \sim 0$. This captures the desired equivalence.

Note that these definitions are set up for univariate polynomials (coefficients are assumed to be real numbers). This is deliberate. In Sect. 2.3.3, we will discuss how our framework reduces correctness results for multivariate polynomials to univariate lemmas like `infinity_evalUni`.

Infinitesimals

Given arbitrary r (not containing x), VS of $r + \epsilon$ for variable x should capture the equivalence $F_x^{r+\epsilon} \longleftrightarrow \exists y{>}r. \forall x{\in}(r, y].\ F(x)$, where F does not contain y. Intuitively, this says that (in any valuation of the non-x variables) $r + \epsilon$ captures information for the interval between r and the next greatest x-root.

For $=$ and $\neq$ atoms, we proceed in the same manner as we did with $-\infty$, as $(r, y]$ contains infinitely many points and only the zero polynomial has infinitely many solutions. As before, $\leq$ atoms turn into disjunctions of the inequality and equality representations at $r + \epsilon$. We are left only to consider $<$ atoms.

Consider $(p{<}0)_x^{r+\epsilon}$ where $p = ax^2 + bx + c$ with polynomials a, b, c not containing x, and an arbitrary r not containing x. Notice that if $(p{<}0)_x^r$, then because polynomials are continuous, we can choose a small enough y so that $\forall x{\in}(r, y].\ p{<}0$. If instead $(p = 0)_x^r$, then consider the partial derivative of p evaluated at r. If $\frac{\partial p}{\partial x}(r)$ is negative, then $\exists y{>}r. \forall x{\in}(r, y].\ p{<}0$ holds, because p is decreasing in x locally after $x{=}r$. If $\frac{\partial p}{\partial x}(r)$ is positive, then $\exists y{>}r. \forall x{\in}(r, y].\ p{<}0$ *does not* hold, because p is increasing in x after $x{=}r$. If $\frac{\partial p}{\partial x}(r)$ is zero, then to ascertain whether $\exists y{>}r. \forall x{\in}(r, y].\ p{<}0$, we will need to check higher derivatives.

This pattern forms the following recurrence, with the base case $(p < 0)_x^{r+\epsilon} = (p < 0)_x^r$ for polynomials p of degree zero:

$$(p < 0)_x^{r+\epsilon} \overset{\text{def}}{=} (p < 0)_x^r \vee \left((p = 0)_x^r \wedge ((\partial p/\partial x) < 0)_x^{r+\epsilon}\right)$$

We use the VS algorithm from Section 2.2.2 to characterize $(p < 0)^r_x$ and $(p = 0)^r_x$.

In Isabelle/HOL, we show that given a quadratic root r, the virtual substitution of $r + \epsilon$ satisfies the desired equivalence in the following theorem (on $\mathbb{R}$ using Remark 1; we have an analogous lemma for linear roots r):

```
lemma infinitesimal_quad:
  fixes A B C D:: "real"
  assumes "D≠0"
  assumes "C≥0"
  shows "(∃y::real>((A+B * sqrt(C))/(D)).
      ∀x::real ∈{((A+B * sqrt(C))/(D))<..y}. aEvalUni At x)
      = (evalUni (substInfinitesimalQuadraticUni A B C D At) x)"
```

Note that `{r<..y}` in Isabelle stands for the range $(r, y]$. This says that, given `At` of type `atomUni`, with tuple (a, b, c) and sign condition $\sim \in \{<, =, \leq, \neq\}$, $At^{r+\epsilon}_x$ holds iff $\exists y > r.\forall x \in (r, y].ax^2 + bx + c \sim 0$, which is the desired equivalence.

The General VS Theorem

Now that we have explained virtually substituting $-\infty$ and infinitesimals, we are ready to state the general VS theorem.

Let F be a formula of the following shape, where each a_i, b_i, c_i, and d_i is a polynomial that is at most quadratic in variable x:

$$F = \left(\bigwedge a_i = 0\right) \wedge \left(\bigwedge b_i < 0\right) \wedge \left(\bigwedge c_i \leq 0\right) \wedge \left(\bigwedge d_i \neq 0\right).$$

Let $R(p)$ denote the set of symbolic expressions of the form $(g_1 + g_2\sqrt{g_3})/g_4$ that, as in Sect. 2.2.2, are roots of the polynomial p in x, where the g_i's are polynomials not involving x. For the zero polynomial, let $R(0) = \emptyset$. Note that, as in Sect. 2.2.2, the g_i's come with certain well-definedness checks that we retain implicitly in the construction (for example, $g_4 \neq 0$ and $g_3 \geq 0$). We now define:

$$A = \bigcup R(a_i) \quad B = \bigcup R(b_i) \quad C = \bigcup R(c_i) \quad D = \bigcup R(d_i)$$

Then we obtain the following QE equivalence, where for simplicity we elide the relevant crucial well-definedness checks (cross-reference [70, Theorem 21.1]):

$$(\exists x.F) \longleftrightarrow F^{-\infty}_x \vee \bigvee_{r \in A \cup C} F^r_x \vee \bigvee_{r \in B \cup C \cup D} F^{r+\epsilon}_x \tag{2.3}$$

Intuitively, this formula states that if there is a particular x that satisfies F, then it must be the case that x is one of the equality roots from $A \cup C$, or that x falls in one of the particular ranges (including $-\infty$ as a range) obtained by partitioning the number line by the roots in $B \cup C \cup D$.

Equation (2.3) can be optimized further by eliding C from the off-roots:

$$(\exists x.F) \longleftrightarrow F^{-\infty} \vee \bigvee_{r \in A \cup C} F^r_x \vee \bigvee_{r \in B \cup D} F^{r+\epsilon}_x. \tag{2.4}$$

Intuitively, this optimization holds because polynomials are continuous. More precisely, if F has the shape $F = (p{\le}0 \wedge G)$, and if r is an x-root of p, then r already satisfies $p{\le}0$ in any valuation of the non-x variables, so including $r + \epsilon$ as a sample point on account of $p{\le}0$ is redundant. It is possible that G contains some atom $q < 0$ or $q \neq 0$ where r is an x-root of q. In this case, $r + \epsilon$ will already be a sample point on account of q, and we do not need to add it in on account of p. Alternatively, if G does not contain such a q, then, in any valuation of the non-x variables, it is impossible for G to be satisfied by $r + \epsilon$ and not r, meaning that it is redundant to include $r + \epsilon$ as a sample point on account of G.

The general QE theorem is proved in Isabelle/HOL as the following, using Remark 1 to restrict to the univariate case and avoid well-definedness formulas:

```
theorem general_qe:
  defines "R ≡ {(=), (<), (≤), (≠)}"
  assumes "∀rel∈R. finite (Atoms rel)"
  defines "F ≡ (λx. ∀rel∈R. ∀(a,b,c)∈(Atoms rel). rel (a*x^2+b*x+c) 0)"
  defines "Fε ≡ (λr. ∀rel∈R. ∀(a,b,c)∈(Atoms rel). ∃y>r. ∀x∈{r<..y}.
    rel (a*x^2+b*x+c) 0)"
  defines "F_inf ≡ (∀rel∈R. ∀(a,b,c)∈(Atoms rel). ∃x. ∀y<x.
    rel (a*y^2+b*y+c) 0)"
  defines "all_roots ≡ (λ(a,b,c).
     if a=0 ∧ b≠0 then {-c/b} else
     if a≠0 ∧ b^2-4*a*c≥0 then {(-b+sqrt(b^2-4*a*c))/(2*a)}
        ∪ {(-b-sqrt(b^2-4*a*c))/(2*a)} else {})"
  shows "(∃x. F(x)) =  (F_inf ∨
           (∃r∈⋃(all_roots ` (Atoms (=) ∪ Atoms (≤))). F r) ∨
           (∃r∈⋃(all_roots ` (Atoms (<) ∪ Atoms (≠))). Fε r))"
```

Here, ` is the Isabelle/HOL syntax for mapping a function over a set. This theorem says that if a finite-length formula `F` is of the requisite shape, then there exists an $x$ satisfying `F` iff `F` is satisfied at $-\infty$ (captured by $F_{inf}$), or there is a root `r` of one of the $=$ or $\leq$ atoms where `F r` holds, or if there is a root `r` of one of the $<$ or $\neq$ atoms where $F_\varepsilon$ `r` holds. The proof is quite lengthy and involves a significant amount of casework (we choose to split the proof into two implications, proving that the LHS implies the RHS and vice-versa; to show that the RHS implies the LHS, we consider each of the disjunctions on the RHS separately, and showing that the LHS implies the RHS requires proving that we are not losing information by limiting our attention to the sample points); however, because we are working with univariate polynomials thanks to Remark 1, this casework mostly reduces to arithmetic computations and basic real analysis for univariate polynomials, and some of what we need, such as properties of discriminants and continuity properties of polynomials, is already formalized in Isabelle/HOL's standard library.

2.2.4 Top Level Algorithms

We develop several top-level algorithms that perform these VS procedures on multivariate polynomials; these are described in more detail in Appendix A.2. Crucially, each features its own proof of correctness. For example, the following theorem proves the soundness of the `VSEquality` algorithm, which performs equality VS repeatedly. The correctness of this theorem only relies on Isabelle/HOL's trusted core[2].

theorem *VSEquality_eval: "∀ν. eval (VSEquality φ) ν = eval φ ν"*

Here, the *eval* function expresses the truth value of the (multivariate) input formula given a valuation *xs*, represented as a list of real numbers. This theorem says that, over all possible valuations, the truth value of φ is the same as the truth value of *VSEquality φ*; that is, applying *VSEquality* to a formula does not change its logical meaning, i.e. *VSEquality* is sound.

As our algorithms are general enough to handle formulas with high degree polynomials where VS does not apply, we cannot assert that the result is quantifier free (it might not be). To demonstrate the practical usefulness of these algorithms, we export our code to SML and experimentally show that these algorithms solve many benchmarks. The code exports rely on the correctness of Isabelle/HOL's code export, which ongoing work is attempting to establish [44].

2.3 Framework

We turn to a discussion of our framework, which is designed with two key goals in mind: First, perform VS as many times as possible on any given formula. Second, reduce unwieldy multivariate proofs to more manageable univariate ones.

2.3.1 Representation of Formulas

We define our type for formulas in the canonical datatype *fm*:

datatype *(atoms: 'a) fm = TrueF | FalseF | Atom 'a |*
And "'a fm" "'a fm" | Or "'a fm" "'a fm" | Neg "'a fm" |
ExQ "'a fm" | AllQ "'a fm" | ExN "nat" "'a fm" | AllN "nat" "'a fm"

As in Nipkow's previous work [64], we use De Bruijn indices to express the variables: That is, the 0th variable represents the innermost quantifier, and variables greater than the number of quantifiers represent the free variables.

We have two constructors for each type of quantifier: *ExQ F* (resp. *AllQ F*) indicates a single existential (universal) quantifier, and *ExN n F* (resp. *AllN n F*) represents a *block* of *n* existential (universal) quantifiers. These representations are interchangeable and converted back and forth in our algorithm; we include the block representation for variable ordering heuristics (see Appendix A.3.3).

We utilize the multivariate polynomial library [80] to define our atoms:

datatype *atom = Less "real mpoly" | Eq "real mpoly" | Leq "real mpoly"*
| Neq "real mpoly"

Each atom is normalized without loss of generality, so that atom *Less p* means $p < 0$, *Eq p* means $p = 0$, and so on.

[2]The correctness of a theorem in the *mathematical* sense also relies on the correctness of its component definitions. If any of the definitions used in a theorem does not reflect the mathematical intention, then the theorem (while still proved) does not express what it is intended to express, and is thus not correct in a mathematical sense.

For example, the FOL$_\mathbb{R}$ formula $\forall x.((\exists y.xa = y^2b) \wedge \neg(\forall y.5x^2 \leq y))$ is represented in our framework as follows, where `Const n` represents the constant $n \in \mathbb{R}$, and `Var i` represents the ith variable:

```
AllQ (And (ExQ (Atom (Eq (Var 1 * Var 2 - (Var 0)^2 * Var 3))))
       (Neg (AllQ (Atom (Leq (Const 5 * (Var 1)^2 - Var 0)))))).
```

Note that we could restrict ourselves to the $\top, \neg, \vee, \exists$ connectives and normalize $\leq$ and $\neq$ atoms to combinations of $<$ and $=$ atoms, and we could still express all of FOL$_\mathbb{R}$. We avoid this for two reasons: because it would linearly increase the size of the formula, and because we want to handle $\leq$ atoms in the optimized way discussed in Sect. 2.2.3 (see (2.4)). We do, however, allow for the normalization of $p = q$ into $p - q = 0$. This does not affect the size of the formula, and can afford simplifications: For example, $x^3 + x^2 + x + 1 = x^3$ becomes $x^2 + x + 1 = 0$.

2.3.2 Modified Disjunctive Normal Form

Nipkow's prior work [64] avoided incurring cases where linear VS does not apply by constraining atoms to be linear. In order to develop a VS method which can be used, e.g., as a preprocessing method for CAD, we must reason about cases where VS fails to perform QE for a specific quantifier. We would like to use the context of the broader formula to continue the execution of the VS algorithm to the remaining quantifiers to simplify as much as possible. To help with this, we implement a modified disjunctive normal form (DNF) that allows expressions to involve quantifiers.

Contextual Awareness

Let us analyze how to increase the informational content in a formula with respect to a quantified variable of interest.

Say we wish to perform VS to eliminate variable x in the formula $\exists x.F$, where F is not necessarily quantifier free. In linear time, we remove all negations from the formula by converting it into negation normal form. We can then normalize $\exists x.F$ into the following form, where the $A_{n,i}$'s are (quantifier-free) atoms:

$$\exists x. \bigvee_n \Big(\bigwedge_i A_{n,i} \wedge \bigwedge_j (\forall y.F_{n,j}) \wedge \bigwedge_k (\exists z.F_{n,k}) \Big).$$

This normalization procedure is similar to standard DNF, as it handles quantified formulas as if they were atomic formulas. We can distribute the existential quantifier across the disjuncts, which results in the equivalent formula:

$$\bigvee_n \exists x. \Big(\bigwedge_i A_{n,i} \wedge \bigwedge_j (\forall y.F_{n,j}) \wedge \bigwedge_k (\exists z.F_{n,k}) \Big). \tag{2.5}$$

Now we run the VS algorithm, i.e. the input to VS is a conjunction of atomic formulas and quantified formulas in the shape of (2.5). Notice that if equality VS applies to atom $A_{n,i}$, then the relevant roots can be substituted into the quantified formulas $F_{n,j}$ and $F_{n,k}$, but roots from $F_{n,j}$ or $F_{n,k}$ cannot be substituted into $A_{n,i}$ since they feature quantified variables which are undefined in the broader context. So, our informational content is greatest when the number of $A_{n,i}$ atoms is maximized and the sizes of the $F_{n,j}$ and $F_{n,k}$ are minimized.

Innermost Quantifier Elimination

The innermost quantifier has an associated formula which is entirely quantifier free (and thus has no $F_{n,j}$ and $F_{n,k}$). As such, we opt to perform VS recursively, starting with the innermost quantifier and moving outwards, hoping that VS is successful and the quantifier-free property is maintained. This is not always optimal. Consider the following formula:

$$\exists x.(x = 0 \wedge \exists y.\ xy^3 + y = 0).$$

If we attempt to perform quadratic VS on the innermost y quantifier, it is cubic and will fail. However, performing VS on the x quantifier first fixes $x = 0$, which converts the cubic $xy^3 + y = 0$ equality into the linear $y = 0$. So, an (unoptimized) run of inside-out VS would produce $\exists y.y = 0$, and we could completely resolve the QE query by running VS again.

Reaching Under Quantifiers

We would like to recover usable information from the $F_{n,k}$ formulas to increase the informational content going into our QE algorithm. It would be ideal if we could "reach underneath" the existential binders and "pull out" the atoms from the formulas. We can achieve this through a series of transformations. Let k range from 0 to K_n. If we pull out each existential quantifier one by one, we get the following formula, which is equivalent to formula (2.5):

$$\bigvee_n \exists z_0. \cdots \exists z_{K_n}.\exists x.\Big(\bigwedge_i A_{n,i} \wedge \bigwedge_j (\forall y.F_{n,j}) \wedge \bigwedge_k F_{n,k}\Big)$$

This works because the rest of the conjuncts do not mention the quantified variable z_k and adjacent existential quantifiers can be swapped freely (without changing the logical meaning of the formula).

We can then recursively unravel the formulas $F_{n,k}$, moving as many existential quantifiers as possible to the front. Our implementation does this via a bottom-up procedure, starting underneath the innermost existential quantifier and building upwards, normalizing the formula into the form:

$$\bigvee_n \exists z_0. \cdots \exists z_{K_n}.\exists x.\Big(\bigwedge_i A_{n,i} \wedge \bigwedge_j \forall y.F_{n,j}\Big)$$

On paper, these transformations are simple as they involve named quantified variables; however, because our implementation uses a locally nameless form for quantifiers with de Bruijn indices, shifting an existential quantifier requires a "lifting" procedure $A\!\uparrow$ which increments all the variable indices in A by one. This allows for the following conversion: $A \wedge \exists z.F \longleftrightarrow \exists z.((A\!\uparrow) \wedge F)$.

2.3.3 Logical Evaluation

Our proofs show that the input formula and the output formula (after VS) are *logically equivalent*, i.e., have the same truth value under any valuation. This needs a method of "plugging in" the real-valued valuation into the variables of the polynomials. Towards this, we define the `eval` function, which accumulates new values into the valuation as we go underneath quantifiers, and the `aEval` function, which homomorphically evaluates a polynomial at a valuation.

When proving correctness, we focus our attention on one quantifier at a time. By Remark 1, the correctness of general VS follows when considering a formula F with a single quantifier, where F contains only polynomials of at most degree two (otherwise general VS does not apply). With these restrictions, we can substitute a valuation into the non-quantified variables, transforming multivariate polynomials into univariate polynomials. For example, let $a, b,$ and c be arbitrary multivariate polynomials that do not mention variable x. Let $\hat{p} = \gamma(p)$ denote the evaluation of polynomial p at valuation γ ($\hat{p}$ is a real number). We obtain the following conversion between multivariate and univariate polynomials:

$$\textbf{eval}\ (ax^2 + bx + c = 0)\ \gamma \longleftrightarrow \textbf{evalUni}\ (\hat{a}x^2 + \hat{b}x + \hat{c} = 0)\ \hat{x}$$

As such, we develop an alternative VS algorithm for univariate polynomials, where atoms are represented as triples of real-valued coefficients (as seen in Sect. 2.2.3), and show that under this specific valuation, the multivariate output is equivalent per valuation with the output of the univariate case. Thus, we finish the proof of the multivariate case by lifting the proof for the univariate case.

2.3.4 Polynomial Contributions

We build on the polynomials library [80], which was designed to support executable multivariate polynomial operations. This choice naturally comes with tradeoffs, and a number of functions and lemmas that we needed were missing from the library. For example, we needed an efficient way to isolate the coefficient of a variable within a polynomial, which we define in the `isolate_variable_sparse` function. The following particularly critical lemma rewrites a multivariate polynomial in $\mathbb{R}[a_1, \ldots, a_n, x]$ as a nested polynomial $\mathbb{R}[a_1, \ldots, a_n][x]$, i.e., a univariate polynomial in x with coefficients that are polynomials in $\mathbb{R}[a_1, \ldots, a_n]$:

```
lemma sum_over_degree: "(p :: real mpoly) =
  (∑ i≤degree p x. isolate_variable_sparse p x i * Var x^i)"
```

This is needed rather frequently within VS, as we often seek to re-express polynomials with respect to a single quantified variable of interest, and although it is mathematically quite obvious, its verification was somewhat involved.

Additionally, to utilize the variables within polynomials as de Bruijn indices, we implemented various lifting and substitution operations. These include the `liftPoly` and `lowerPoly` variable reindexing functions. These and other contributions to the polynomials library are discussed in Appendix A.2.4.

2.4 Experiments

The benchmark suite consists of 378 QE problems in category CADE09 collected from 94 examples [72], and category Economics with 45 QE problems [60]. CADE09 and Economics examples were converted into decision problems, powers were flattened to multiplications, and CADE09 were additionally rewritten to avoid polynomial division. For sanity checking, we also negated the CADE09 examples [72]. We run on commodity hardware.[3] The benchmark examples, as well as all scripts to rerun the experiments are in [77].

Tools. We compare the performance of *a*) our VSEquality (**E**), VSGeneral (**G**), VSLucky (**L**), and VSLEG (**LEG**) algorithms (Appendix A.2) to *b*) Redlog [30] snapshots 2021-04-13[4] ($\mathbf{R}_{\lightning}$) and 2021-07-16[5] ($\mathbf{R}_{\checkmark}$, which includes bug fixes for contradictions we reported to Redlog developers), *c*) SMT-RAT 21.05[6] [21] quantifier elimination ($\mathbf{S\text{-}QE}_{\checkmark}$) and satisfiability checking ($\mathbf{S\text{-}SAT}_{\lightning}$), *d*) the SMT solver Z3 4.8.10[7] [25] (**Z3**), and *e*) Wolfram Engine 12.3.1(**W-VS**, **W-QE**). All tools were run in Docker containers on Ubuntu 18.04 with 8GB of memory and 6 CPU cores. Tool syntax translations from SMT-LIB format were done prior to benchmarking: For our VS algorithms, examples were translated to SML data structures and compiled with MLton[8]; as a result, measurements do not include parsing. For W-VS and W-QE, examples were translated into Wolfram syntax, including configuration options restricting QE to quadratic virtual substitution in W-VS. For S-QE$_{\checkmark}$, `eliminate-quantifiers` replaced `check-sat`.

Results. Each example has a timeout of 30s. Figure 2.2 summarizes the performance on the CADE09 and Economics examples in terms of the cumulative time needed to solve (return “true”, “false”, “sat”, or “unsat”) the fastest n problems with a logarithmic time axis: more problems solved and a flatter curve is better.

Wolfram Engine solves all problems in the CADE09 category, closely trailed by Redlog, Z3. The near constant computation time offset of Redlog in comparison to Z3, SMT-RAT, and Wolfram Engine may be attributable to the additional step of entering an SMT REPL. Our verified VSEquality (E), VSGeneral (G), VSLucky (L), and VSLEG (LEG) algorithms rank in performance between the basic quantifier elimination implementation in SMT-RAT (S-QE$_{\checkmark}$), virtual substitution in Wolfram Engine (W-VS), full SMT approaches (S-SAT$_{\lightning}$, Z3), and combined virtual substitution plus CAD implementations (R$_{\checkmark}$, W-QE). The reduced startup time of our algorithms is attributable to the omitted parsing step. Overall, VSEquality and VSLucky solve examples fast, but the wider applicability of VSGeneral and VSLEG allows them to solve considerably more examples. Though we have already implemented a number of optimizations for VS (Appendix A.3) we do not expect to outperform prior tools at this stage, as many of them have been optimized over a period of many years.

[3] MacBook Pro 2019 with 2.6GHz Intel Core i7 (model 9750H) and 32GB memory (2667MHz DDR4 SDRAM).

[4] `https://sourceforge.net/projects/reduce-algebra/files/snapshot_2021-04-13/`

[5] `https://sourceforge.net/projects/reduce-algebra/files/snapshot_2021-07-16/`

[6] `https://github.com/ths-rwth/smtrat/releases/tag/21.05`

[7] `https://github.com/Z3Prover/z3/releases/tag/z3-4.8.10`

[8] `http://mlton.org/`

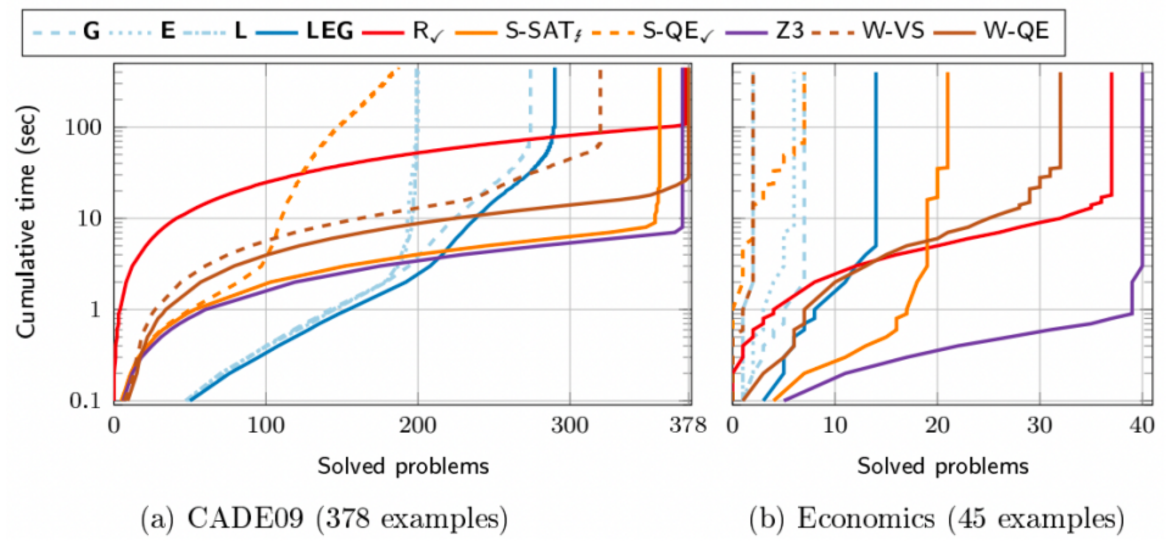


Figure 2.2: Cumulative time to solve fastest n problems (flatter and more is better)

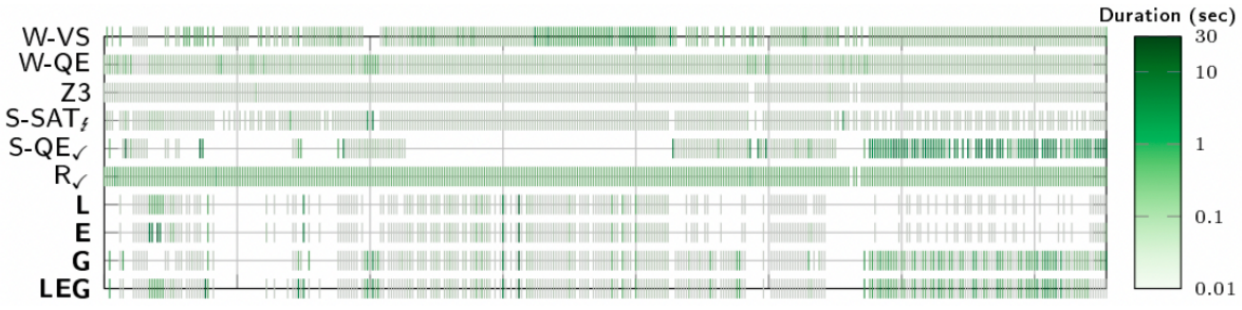


Figure 2.3: CADE09 duration per problem (color indicates duration, lighter is better)

A comparison of duration per problem is in Fig. 2.3. Though there is considerable overlap between VSEquality, VSGeneral, and VSLucky, mutually exclusive sets of solved examples (and considerable performance differences on a number of examples) foreshadow the performance achievable with the combined VSLEG algorithm.

Contradictions. In Fig. 2.4, we compare the CADE09 results to the results on negated CADE09 examples to highlight *contradictions* between answers (e.g., both A and $\neg A$ are claimed to be true). Wolfram Engine and Z3 answer consistently on both formula sets, and solve (almost) all examples. Redlog, the main VS implementation, in R↯ and previous versions in general does not perform well on the negated formulas and reports 96 contradictory answers; the contradictory examples were shared with the developers and triggered several bug fixes that are now available in R✓ (no contradictions found on the benchmark set). SMT-RAT performs better than R↯ on the negated formulas, but in satisfiability mode contradicts itself on 41 examples by silently ignoring quantifiers in the input; in quantifier elimination mode, SMT-RAT supports quantifiers and does not report contradictions, but SMT-RAT then incurs a significant performance loss (S-SAT↯ reports 359 answers while S-QE✓ only solves 187). No contradictions were found across tools,

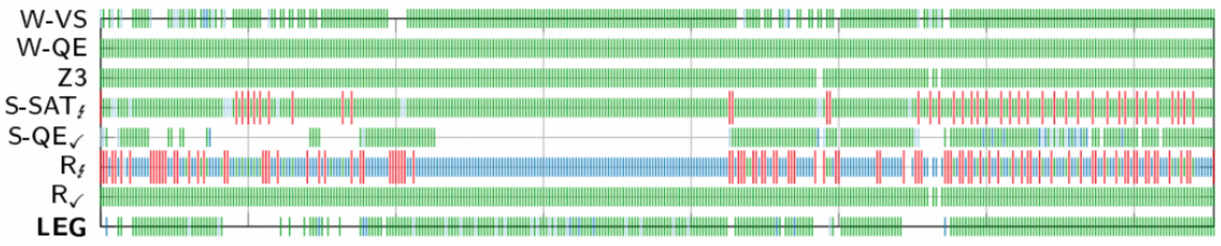


Figure 2.4: CADE09 consistency comparison between original and negated formula: color indicates discrepancies within tools (green (🟩): answer on original and negated formula agree, dark-blue (🟦): only original solved, light-blue (🟦): only negated solved, red+long (🟥): **contradictory** answers (both formulas unsat/proved or both sat/disproved), empty: both timeout/unknown)

i.e., whenever a tool's answers were consistent internally, the answers agreed with those of other tools. Our VSLEG algorithm has similar performance for proving and disproving in terms of absolute number of solved examples, but combining proving and disproving would still solve more examples than just one question individually (as for S-QE$_{\checkmark}$ and W-VS).

In summary, the performance of our verified virtual substitution QE on the benchmark set is encouraging. The number of examples solved by our verified VS is close to other VS implementations (304 examples by our VSLEG vs. 322 by W-VS) and the cumulative solving time reveals that the majority of examples are solved fast.

2.5 Takeaways and Future Directions

This chapter focuses on the formalization of linear and quadratic virtual substitution for real arithmetic. Our algorithms are *provably correct* up to Isabelle/HOL's trusted core and code export. Developing practical verified VS in Isabelle/HOL required significant low-level improvements and extensions to Isabelle's multivariate polynomials library. The extensive experiments both reveal the benefits of our current optimizations and indicate room for future improvements. Adding in some simplification techniques from the literature by, for example, generalizing our existing degree-reduction optimizations (see Appendix A.3.1) or more closely analyzing dependencies of atoms within formulas [31] could be of interest. Further optimizations to Isabelle/HOL's polynomial libraries, such as efficient coefficient lookup for polynomials using red black trees, would be welcome. Expanding our framework to handle formulas that involve polynomial division would also be of practical significance. Continuing to develop our formalization with such improvements is of especial significance given that our experiments found long-standing errors in existing unverified real arithmetic tools. Combined with our promising experimental results, this highlights the benefits of formal verification, which provides an ideal path toward QE implementations that are both useful and correct in practice.

Chapter 3

Verifying Univariate BKR

This chapter discusses the formal verification of the univariate fragment of Ben-Or, Kozen, and Reif's (BKR) decision procedure [2] for first-order real arithmetic in Isabelle/HOL. Our proof combines ideas from BKR and a follow-up work by Renegar [75] that are well-suited for formalization. While formalizing the univariate case of BKR, we simultaneously formalize the univariate case of Renegar. The resulting proof outline allows us to build substantially on Isabelle/HOL's libraries for algebra, analysis, and matrices. Our main extensions to existing libraries are also detailed.

Our formalization of univariate BKR is ≈ 7000 lines. It is available on the Archive of Formal Proofs (AFP) [20]. Our main contributions are:

- In Sect. 3.1, we present an algorithmic blueprint for implementing BKR's procedure that blends insights from Renegar's [75] later variation of BKR. Compared to the original abstract presentations [2, 75], our blueprint is phrased concretely in terms of matrix operations which facilitates its implementation and identifies its correctness properties.
- Our blueprint is designed for formalization by judiciously combining and fleshing out BKR's and Renegar's proofs. In Sect. 3.2, we outline key aspects of our proof, its use of existing Isabelle/HOL libraries, and our contributions to those libraries.

As stated previously, when its inherent parallelism is exploited, an optimized version of univariate BKR is an NC algorithm (i.e. it runs in parallel polylogarithmic time). Our formalization is not yet optimized and parallelized, so we do not yet achieve such efficiency. However, we do export our Isabelle/HOL formalization to SML code and are already able to solve some microbenchmarks with this exported code (Sect. 3.2.3).

Additionally, our univariate formalization is a significant stepping stone towards the multivariate case, which builds inductively on the univariate case. Indeed, this formalization was already instrumental in our verification of a complete multivariate algorithm which mixes ideas from BKR and Tarski (Chapter 4 of this thesis).

Collaborators. The material in this chapter is based upon joint work with Yong Kiam Tan and André Platzer which appeared at ITP 2021 [19].

3.1 Mathematical Underpinnings

This section provides an outline of our decision procedure for univariate real arithmetic and its verification in Isabelle/HOL [65]. The goal is to provide an accessible mathematical blueprint that explains our construction and its blend of ideas from BKR [2] and Renegar [75]; in-depth technical discussion of the formal proofs is largely deferred to Sect. 3.2. Our procedure starts with two transformation steps (Sections 3.1.1 and 3.1.2) that simplify an input decision problem into a so-called restricted sign determination format. An algorithm for the latter problem is then presented in Sect. 3.1.3. Throughout this paper, unless explicitly specified, we are working with *univariate* polynomials, which we assume to have variable x. Our decision procedure works for polynomials with rational coefficients (`rat poly` in Isabelle), though some lemmas are proved more generally for univariate polynomials with real coefficients (`real poly` in Isabelle).

3.1.1 From Univariate Problems to Sign Determination

Formulas of *univariate real arithmetic* are generated by the following grammar, where p is a univariate polynomial with rational coefficients:

$$\phi, \psi \ ::= \ p > 0 \mid p \geq 0 \mid p = 0 \mid \phi \lor \psi \mid \phi \land \psi$$

In Isabelle/HOL, we define this grammar in `fml`, which is our type for formulas. This is very similar to our datatype for multivariate formulas from Sect. 4.1.3. The major difference is that our univariate grammar is simpler because we are explicitly restricting our attention to univariate formulas, and thus we do not need to include quantifiers in our grammar. As a minor difference, unlike in our multivariate grammar, we do not include logical negation of formulas in our datatype; instead, we explicitly include cases for $p < 0$, $p \leq 0$, and $p \neq 0$ in our `fml` type. This does not significantly increase proof overhead as, unlike the VS algorithm, the BKR algorithm is more concerned with the mathematical content of a formula (i.e., the polynomials it involves and their mathematical properties) than with the syntactic structure of the formula.

For formula ϕ, the *universal* decision problem is to decide if ϕ is true for *all* real values of x, i.e., validity of the quantified formula $\forall x\, \phi$. The *existential* decision problem is to decide if ϕ is true for *some* real value of x, i.e., validity of the quantified formula $\exists x\, \phi$. For example, a decision procedure should return false for formula (3.1) and true for formula (3.2) below (left).

$$\forall x\,(x^2 - 2 = 0 \land 3x > 0) \tag{3.1}$$

$$\exists x\,(x^2 - 2 = 0 \land 3x > 0) \tag{3.2}$$

Formula Structure: $\textcircled{A} = 0 \land \textcircled{B} > 0$

Polynomials: $\textcircled{A} : x^2 - 2, \quad \textcircled{B} : 3x$

The first observation is that both univariate decision problems can be transformed to the problem of finding the set of *consistent sign assignments* (also known as realizable sign assignments [1, Definition 2.34]) of the set of polynomials appearing in the formula ϕ.

Definition 1. A **sign assignment** for a set G of polynomials is a mapping σ that assigns each $g \in G$ to either $+1$, -1, or 0. A sign assignment σ for G is **consistent** if there exists an $x \in \mathbb{R}$ where, for all $g \in G$, the sign of $g(x)$ matches the sign of $\sigma(g)$.

For the polynomials $x^2 - 2$ and $3x$ appearing in formulas (3.1) and (3.2), the set of all consistent sign assignments (written as ordered pairs) is:

$$\{(+1,-1),\ (0,-1),\ (-1,-1),\ (-1,0),\ (-1,+1),\ (0,+1),\ (+1,+1)\}$$

Formula (3.1) is not valid because the consistency of sign assignment $(0,-1)$ implies there exists a real value $x \in \mathbb{R}$ such that conjunct $x^2 - 2 = 0$ is satisfied but not $3x > 0$. Conversely, formula (3.2) is valid because the consistent sign assignment $(0,+1)$ demonstrates the existence of an $x \in \mathbb{R}$ satisfying $x^2 - 2 = 0$ and $3x > 0$. The truth-value of formula ϕ at a given sign assignment is computed by evaluating the formula after replacing all of its polynomials by their respective assigned signs. For example, for the sign assignment $(0,-1)$, replacing $\textcircled{A}$ by 0 and $\textcircled{B}$ by -1 in the formula structure underlying (3.1) and (3.2) shown above (right) yields $0 = 0 \wedge -1 > 0$, which evaluates to false. Validity of $\forall x\, \phi$ is decided by checking that ϕ evaluates to true at *each* of its consistent sign assignments. Similarly, validity of $\exists x\, \phi$ is decided by checking that ϕ evaluates to true at *at least one* consistent sign assignment.

Our top-level formalized algorithms are `decide_universal` and `decide_existential`, both with type `rat poly fml ⇒ bool`. The definition of `decide_existential` is as follows (the omitted definition of `decide_universal` is similar):

```
definition decide_existential :: "rat poly fml ⇒ bool"
where "decide_existential fml = (let (fml_struct,polys)=convert fml in
   find (lookup_sem fml_struct) (find_consistent_signs polys) ≠ None)"
```

Here, `convert` extracts the list of constituent polynomials `polys` from the input formula `fml` along with the formula structure `fml_struct`, `find_consistent_signs` returns the list of all consistent sign assignments `conds` for `polys`, and `find` checks that at one of those sign assignments the predicate `lookup_sem fml_struct` is true. Given a sign assignment σ, `lookup_sem fml_struct` σ evaluates the truth value of `fml` at σ by recursively evaluating the truth of its subformulas after replacing polynomials by their sign according to σ using the formula structure `fml_struct`. Thus, `decide_existential` returns true iff `fml` evaluates to true for at least one of the consistent sign assignments of its constituent polynomials.

The correctness theorem for `decide_universal` and `decide_existential` is shown below, where `fml_sem fml x` evaluates the truth of formula `fml` at the real value `x`. This top-level correctness result shows that our decision procedures correctly reduces any quantified univariate input formula to the logically equivalent output of either `True` or `False` (note that univariate QE problems are inherently fully quantified, as each has exactly one quantifier). The correctness of this theorem only relies on the correctness of the trusted core of Isabelle/HOL[1].

```
theorem decision_procedure:
 "(∀x::real. fml_sem fml x) ⟷ decide_universal fml"
 "(∃x::real. fml_sem fml x) ⟷ decide_existential fml"
```

This theorem depends crucially on `find_consistent_signs` correctly finding *all* consistent sign assignments for `polys`, i.e., solving the sign determination problem.

[1]As discussed previously, the mathematical *meaning* of this theorem also relies on the correctness of its component definitions.

3.1.2 From Sign Determination to Restricted Sign Determination

The next step restricts the sign determination problem to the following more concrete format: Find all consistent sign assignments σ for a set of polynomials $q_1, \ldots, q_n$ at the roots of a nonzero polynomial p, i.e., the signs of $q_1(x), \ldots, q_n(x)$ that occur at the (finitely many) real values $x \in \mathbb{R}$ with $p(x) = 0$. Intuitively, the idea here is that the polynomial p should be chosen so that sign information at its roots are representative of sign information for the entire real number line. Then, a key insight of BKR is that this restricted problem can be solved efficiently (in parallel) using purely algebraic tools (Sect. 3.1.3). Following BKR's procedure, we also normalize the q_i's to be coprime with (i.e. share no common factors with) p, which simplifies the subsequent construction for the key step and its formal proof.

Remark 3. *The normalization of q_i's to be coprime with p can be avoided using a slightly more intricate construction due to Renegar [75]. We have also formalized the univariate case of this construction. Intuitively, the overall structure of the algorithms is quite similar, but Renegar's matrix equation is slightly larger than BKR's and uses a slightly more general Tarski query computation than BKR's, in order to store the extra information that can arise when the q_i's are potentially not coprime with p. When generalizing to multivariate polynomials in Chapter 4, we use the Renegar-style matrix equation (which will be described in Sect. 4.1.2), since a coprimality assumption on polynomials does not remove 0 as a potential consistent sign assignment in a multivariate setting[2]. Our formalization of univariate Renegar is available in the AFP alongside our formalization of univariate BKR [20]; as the underlying algorithms are very similar, these formalizations share considerable code.*

Consider as input a set of polynomials (with rational coefficients) $G = \{g_1, \ldots, g_k\}$ for which we need to find all consistent sign assignments. The transformation proceeds as follows:

(1) Factorize the input polynomials G into a set of pairwise coprime factors (with rational coefficients) $Q = \{q_1, \ldots, q_n\}$. This also removes redundant/duplicate polynomials.

    Each input polynomial $g \in G$ can be expressed in the form $g = c \prod_{i=1}^{n} q_i^{d_i}$ for some rational coefficient c and natural number exponents $d_i \geq 1$ so the sign of g is directly recovered from the signs of the factors $q \in Q$. For example, if $g_1 = q_1 q_2$ and in a consistent sign assignment q_1 is positive while q_2 is negative, then g_1 is negative according to that assignment, and so on. Accordingly, to determine the set of all consistent sign assignments for G it suffices to determine the same for Q.

(2) Because the q_i's are pairwise coprime, there is no consistent sign assignment where two or more q_i's are set to zero. So, in any given sign assignment, there is either *exactly one* q_i set to zero, or the q_i's are *all assigned to nonzero* (i.e., +1, -1) signs.

    Now, for each $1 \leq i \leq n$, solve the restricted sign determination problem for all consistent sign assignments of $\{q_1, \ldots, q_n\} \setminus \{q_i\}$ at the roots of q_i. This yields all consistent sign assignments of Q where exactly one q_i is assigned to zero.

[2]While the ring of multivariate polynomials $\mathbb{Q}[x_1, \ldots, x_n]$ is a unique factorization domain (and thus the notions of GCD and coprimality make sense for polynomials in $\mathbb{Q}[x_1, \ldots, x_n]$), even if two polynomials p and q are globally coprime, they may not be coprime in every valuation, and so we would not be able to immediately exclude 0 as a potential consistent sign assignment for q at the roots of p in a multivariate matrix equation. For example, if $p = x$ and $q = 2x - y$, then p and q are coprime in $\mathbb{Q}[x, y]$ but both p and q are 0 when $x = y = 0$.

(3) This step and the next step focus on finding all consistent sign assignments where all q_i's are nonzero. Compute a polynomial p that satisfies the following properties:

(i) p is pairwise coprime with all of the q_i's,

(ii) p has a root in every interval between any two roots of the q_i's,

(iii) p has a root that is greater than all of the roots of the q_i's, and

(iv) p has a root that is smaller than all of the roots of the q_i's.

An explicit choice of p satisfying these properties when $q_i \in Q$ are squarefree and pairwise coprime is shown in Sect. 3.2.1. The relationship between the roots of p and the roots of $q_i \in Q$ is visualized in Fig. 3.1. Intuitively, the roots of p (red points) provide representative sample points between the roots of the q_i's (black squares), and thus cover sign information for all ranges of the number line where all the q_i's are nonzero, as desired.

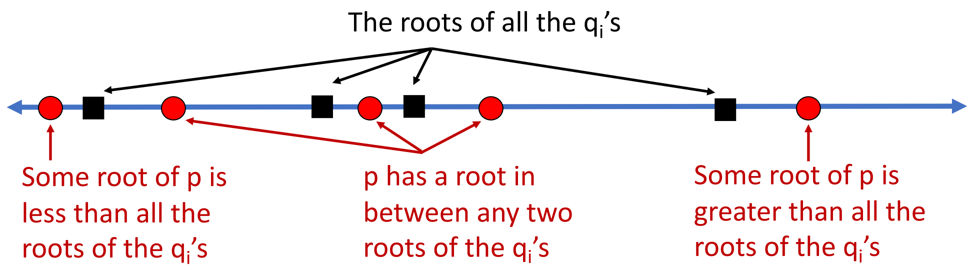

The roots of all the q$_i$'s
Some root of p is less than all the roots of the q$_i$'s
p has a root in between any two roots of the q$_i$'s
Some root of p is greater than all the roots of the q$_i$'s

Figure 3.1: The relation between the roots of the added polynomial p and the roots of the q_i's.

(4) Solve the restricted sign determination problem for all consistent sign assignments of $\{q_1, \ldots, q_n\}$ at the roots of p.

Returning to Fig. 3.1, the q_i's are sign-invariant in the intervals between any two roots of the q_i's (black squares) and to the left and right beyond all roots of the q_i's. Intuitively, this is true because moving along the blue real number line in Fig. 3.1, no q_i can change sign without first passing through a black square. Thus, all consistent sign assignments of q_i that only have nonzero signs must occur in one of these intervals and therefore, by sign-invariance, also at one of the roots of p (red points).

(5) The combined set of sign assignments where some q_i is zero, as found in (2), and where no q_i is zero, as found in (4), solves the sign determination problem for Q, and therefore also for G, as argued in (1).

Our algorithm to solve the restricted sign determination problem using BKR's key insight is called `find_consistent_signs_at_roots`; we now turn to the details of this method.

3.1.3 Restricted Sign Determination

The restricted sign determination problem for polynomials $q_1, \ldots, q_n$ at the roots of a polynomial $p \neq 0$, where each $q_1, \ldots, q_n$ is coprime with p, can be tackled naively by setting up and solving a *matrix equation*. The idea of using a matrix equation for sign determination dates back to Tarski [85] [1, Section 10.3], and accordingly our formalization shares some similarity to Cohen and

Mahboubi's formalization [15] of Tarski's algorithm (see [13, Section 11.2]). BKR's additional insight is to avoid the prohibitive complexity of enumerating exponentially many possible sign assignments for $q_1, \ldots, q_n$ by computing the matrix equation recursively and performing a *reduction* that retains only the consistent sign assignments at each recursive step. This reduction keeps intermediate data sizes manageable because the number of consistent sign assignments is bounded by the number of roots of p throughout. We first explain the technical underpinnings of the matrix equation before returning to our implementation of BKR's recursive procedure. *For brevity, references to sign assignments for $q_1, \ldots, q_n$ in this section are always at the roots of p, because those are representative of sign assignments that are elsewhere.*

Matrix Equation

The inputs to the matrix equation are a set of candidate (i.e., not necessarily consistent) sign assignments $\tilde{\Sigma} = \{\tilde{\sigma}_1, \ldots, \tilde{\sigma}_m\}$ for the polynomials $q_1, \ldots, q_n$ and a set of subsets $S = \{I_1, \ldots, I_l\}$, $I_i \subseteq \{1, \ldots, n\}$ of indices selecting among those polynomials. The set of all consistent sign assignments Σ for $q_1, \ldots, q_n$ is assumed to be a subset of $\tilde{\Sigma}$, i.e., $\Sigma \subseteq \tilde{\Sigma}$.

For example, consider $p = x^3 - x$ and $q_1 = 3x^3 + 2$. The set of all possible candidate sign assignments $\tilde{\Sigma} = \{(+1), (-1)\}$ must contain the consistent sign assignments for q_1 (sign (0) is impossible as p, q_1 are coprime). The possible subsets of indices are $I_1 = \{\}$ and $I_2 = \{1\}$.

The main algebraic tool underlying the matrix equation is the *Tarski query* which provides semantic information about the number of roots of p with respect to another polynomial q.

Definition 2. Given univariate polynomials p, q with $p \neq 0$, the *Tarski query* $N(p, q)$ is:

$$N(p,q) = \#\{x \in \mathbb{R} \mid p(x) = 0, q(x) > 0\} - \#\{x \in \mathbb{R} \mid p(x) = 0, q(x) < 0\}.$$

As an example, for $p = x^3 - x$ and $q = 1$, we have:

$$\begin{aligned} N(p,q) &= \#\{x \in \mathbb{R} \mid x^3 - x = 0, 1 > 0\} - \#\{x \in \mathbb{R} \mid x^3 - x = 0, 1 < 0\} \\ &= \#\{x \in \mathbb{R} \mid x^3 - x = 0\} = 3, \end{aligned}$$

and this captures the information that p has three roots.

Importantly, the Tarski query $N(p, q)$ can be computed from input polynomials p, q using Euclidean remainder sequences *without* explicitly finding the roots of p. This is a consequence of the Sturm-Tarski theorem which has been formalized in Isabelle/HOL by Li [51]. The theoretical complexity for computing $N(p, q)$ is $O(\deg p\ (\deg p + \deg q))$ [1, Sections 2.2.2 and 8.3]. However, this complexity analysis does not take into account the growth in bitsizes of coefficients in the remainder sequences [1, Section 8.3], so it will not be achieved by the current Isabelle/HOL formalization of Tarski queries [51] without further optimization.

For the matrix equation, we lift Tarski queries to a *subset* of the input polynomials:

Definition 3. Given a univariate polynomial $p \neq 0$, univariate polynomials $q_1, \ldots, q_n$, and a subset $I \subseteq \{1, \ldots, n\}$, the *Tarski query* $N(I)$ with respect to p is:

$$\begin{aligned} N(I) = N(p, \Pi_{i \in I} q_i) = \#\{x \in \mathbb{R} \mid p(x) = 0, \Pi_{i \in I} q_i(x) > 0\} \\ - \#\{x \in \mathbb{R} \mid p(x) = 0, \Pi_{i \in I} q_i(x) < 0\}. \end{aligned}$$

The idea here is that, given p and $q_1, \ldots, q_n$, then by considering Tarski queries for a representative set of subsets of $\{1, \ldots, n\}$, we will be able to relate Tarski queries to sign information on the q_i's at the roots of p. For example, when $p = x^3 - x$ and $q_1 = 3x^3 + 2$, the two possible subsets we might consider of $\{1\}$ are $\{\}$ and $\{1\}$. We can then compute: $N(\{\}) = N(p, 1) = 3$ and

$$\begin{aligned} N(\{1\}) = N(p, q_1) &= \#\{x \in \mathbb{R} \mid x^3 - x = 0, 3x^3 + 2 > 0\} - \#\{x \in \mathbb{R} \mid x^3 - x = 0, 3x^3 + 2 < 0\} \\ &= 2 - 1 = 1. \end{aligned}$$

Using these Tarski queries, the sign information for $3x^3 + 2$ at the roots of $x^3 - x$ will then be captured with the matrix equation that we will now describe (see also Fig. 3.2).

The *matrix equation* is the relationship $M \cdot w = v$ between the following three entities:

- M, the l-by-m matrix with entries $M_{i,j} = \Pi_{k \in I_i} \tilde{\sigma}_j(q_k) \in \{-1, 1\}$ for $I_i \in S$ and $\tilde{\sigma}_j \in \tilde{\Sigma}$,
- w, the length m vector whose entries count the number of roots of p where $q_1, \ldots, q_n$ has sign assignment $\tilde{\sigma}$, i.e., $w_i = \#\{x \in \mathbb{R} \mid p(x) = 0, \text{sgn}(q_j(x)) = \tilde{\sigma}_i(q_j) \text{ for all } 1 \leq j \leq n\}$,
- v, the length l vector consisting of Tarski queries for the subsets, i.e., $v_i = N(I_i)$.

Observe that the vector w is such that the sign assignment $\tilde{\sigma}_i$ is consistent (at a root of p) iff its corresponding entry w_i is nonzero. Thus, the matrix equation can be used to solve the sign determination problem by solving for w. In particular, the matrix M and the vector v are both computable from the input (candidate) sign assignments and subsets. Further, since the subsets will be chosen such that the constructed matrix M is *invertible*, the matrix equation uniquely determines w and the nonzero entries of $w = M^{-1} \cdot v$.

The following Isabelle/HOL theorem summarizes sufficient conditions on the list of sign assignments `signs` and the list of index subsets `subsets` for the matrix equation to hold for polynomial list `qs` at the roots of polynomial `p`. Note the switch from set-based representation to list-based representation in the theorem. This formally provides an ordering to the polynomials, sign assignments, and subsets, which is useful for computations.

theorem *matrix_equation:*
assumes *"p≠0"*
assumes *"⋀q. q ∈ set qs ⟹ coprime p q"*
assumes *"distinct signs"*
assumes *"consistent_signs_at_roots p qs ⊆ set signs"*
assumes *"⋀l i. l ∈ set subsets ⟹ i ∈ set l ⟹ i < length qs"*
shows *"M_mat signs subsets $*_v$ w_vec p qs signs = v_vec p qs subsets"*

Here, *M_mat*, *w_vec*, and *v_vec* construct the matrix M and vectors w, v respectively; $*_v$ denotes the matrix-vector product in Isabelle/HOL. The switch into list notation necessitates some consistency assumptions, e.g., that the *signs* list contains *distinct* sign assignments and that the index *i* occurring in each list of indices *l* in *subsets* points to a valid element of the list *qs*. The proof of *matrix_equation* uses a counting argument: intuitively, $M_{i,j}$ is the contribution of any real value x that has the sign assignment $\tilde{\sigma}_j$ towards $N(I_i)$, so multiplying these contributions by the actual counts of those real values in w gives $M_i \cdot w = v_i$.

Note that the theorem does *not* ensure that the constructed matrix M is invertible (or even square). This must be ensured separately when solving the matrix equation for w. We now discuss BKR's inductive construction and its usage of the matrix equation.

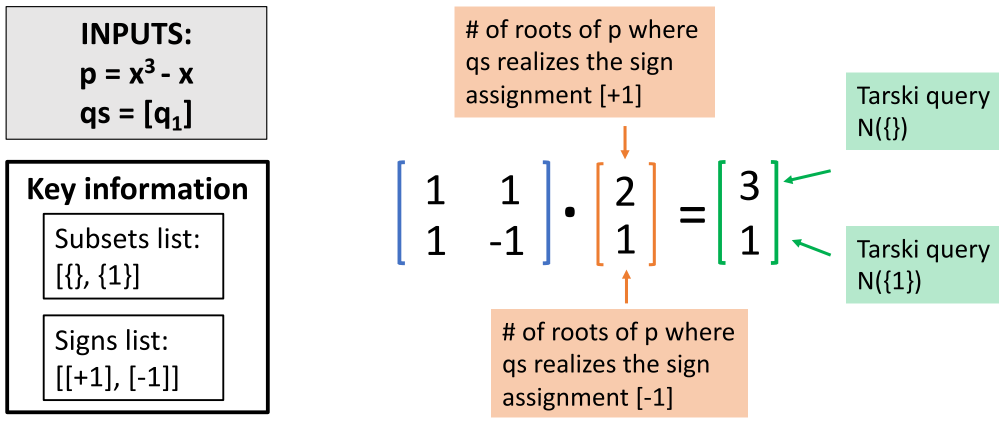


Figure 3.2: Matrix equation for $p = x^3 - x$, $q_1 = 3x^3 + 2$.

Base Case

The simplest (base) case of the algorithm is when there is a single polynomial $[q_1]$. Here, it suffices to set up a matrix equation $M \cdot w = v$ from which we can compute all consistent sign assignments. As hinted at earlier, this can be done with the list of index subsets $[\{\}, \{1\}]$ and the candidate sign assignment list $[(+1), (-1)]$.[3] Further, as illustrated in Fig. 3.2, the matrix M is invertible for these choices of subsets and candidate sign assignments, so the matrix equation can be explicitly solved for w.

Inductive Case: Combination Step

The matrix equation can be similarly used to determine the consistent sign assignments for an arbitrary list of polynomials $[q_1, \ldots, q_n]$. The driving idea for BKR is that, given two solutions of the sign determination problem at the roots of p for two input lists of polynomials, say, $\ell_1 = [r_1, \ldots, r_k]$ and $\ell_2 = [r_{k+1}, \ldots, r_{k+l}]$, one can combine them to yield a solution for the list of polynomials $[r_1, \ldots, r_{k+l}]$. This yields a recursive method for solving the sign determination problem by solving the base case at the single polynomials $[q_1], [q_2], \ldots, [q_n]$, and then recursively combining those solutions, i.e., solving $[q_1, q_2], [q_3, q_4], \ldots,$ then $[q_1, q_2, q_3, q_4], \ldots,$ and so on until a solution for $[q_1, \ldots, q_n]$ is obtained. Importantly, BKR performs a reduction (Sect. 3.1.3) after each combination step to bound the size of the intermediate data.

More precisely, assume for ℓ_1, we have a list of index subsets S_1 and a list of sign assignments $\tilde{\Sigma}_1$ such that $\tilde{\Sigma}_1$ contains all of the consistent sign assignments for ℓ_1 and the matrix M_1 constructed from S_1 and $\tilde{\Sigma}_1$ is invertible. Accordingly, for ℓ_2, we have the list of subsets S_2, list of sign assignments $\tilde{\Sigma}_2$ containing all consistent sign assignments for ℓ_2, and M_2 constructed from $S_2, \tilde{\Sigma}_2$ is invertible. In essence, we are assuming that $S_1, \tilde{\Sigma}_1$ and $S_2, \tilde{\Sigma}_2$ satisfy the hypotheses for the matrix equation to hold, so that they contain all the information needed to solve for the consistent sign assignments of ℓ_1 and ℓ_2 respectively.

Observe that any consistent sign assignment for $\ell = [r_1, \ldots, r_{k+l}]$ must have a prefix that is

[3]In the Isabelle/HOL formalization, we use 0-indexed lists to represent sets and sign assignments, so the subsets list is represented as `[[], [0]]` and the signs list is `[[1], [-1]]`.

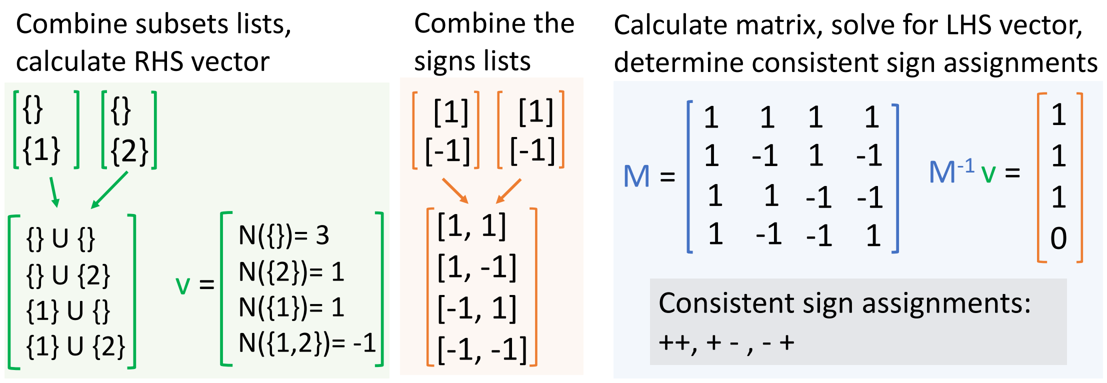

Combine subsets lists, calculate RHS vector
[{} {1}] [{} {2}]
[{} U {}, {} U {2}, {1} U {}, {1} U {2}]
v = [N({})= 3, N({2})= 1, N({1})= 1, N({1,2})= -1]
Combine the signs lists
[1] [-1] [1] [-1]
[1, 1] [1, -1] [-1, 1] [-1, -1]
Calculate matrix, solve for LHS vector, determine consistent sign assignments
M = [1 1 1 1; 1 -1 1 -1; 1 1 -1 -1; 1 -1 -1 1]
M-1 v = [1 1 1 0]
Consistent sign assignments:
++, + - , - +

Figure 3.3: Combining two systems.

itself a consistent sign assignment to ℓ_1 and a suffix that is itself a consistent sign assignment to ℓ_2. Thus, the combined list of sign assignments $\tilde{\Sigma}$ obtained by concatenating every entry of $\tilde{\Sigma}_1$ with every entry of $\tilde{\Sigma}_2$ necessarily contains all consistent sign assignments for ℓ. The combined subsets list S is obtained in an analogous way from S_1, S_2 (where concatenation is now set union), with a slight modification: the subset list S_2 indexes polynomials from ℓ_2, but those polynomials now have different indices in ℓ, so everything in S_2 is shifted by the length of ℓ_1 before combination. Once we have the combined subsets list, we can calculate the RHS vector v with Tarski queries as explained in Sect. 3.1.3.

The matrix M constructed from S, $\tilde{\Sigma}$ is exactly the Kronecker product of M_1 and M_2. Further, the Kronecker product of invertible matrices is invertible, so the matrix equation can be solved for the LHS vector w using M and the vector v computed from the subsets list S. Then the nonzero entries of w correspond to the consistent sign assignments of ℓ. Taking a concrete example, suppose we want to find the list of consistent sign assignments for $\ell = [3x^3 + 2, 2x^2 - 1]$ at the zeros of $p = x^3 - x$. The combination step for $\ell_1 = [3x^3 + 2]$ and $\ell_2 = [2x^2 - 1]$ is visualized in Fig. 3.3.

Reduction Step

The reduction step takes an input list of index subsets S and candidate sign assignments $\tilde{\Sigma}$. It removes the inconsistent sign assignments and then unnecessary index subsets, which keeps the size of the intermediate data tracked for the matrix equation as small as possible.

The reduction step is best explained in terms of the matrix equation $M \cdot w = v$ constructed from the inputs S, $\tilde{\Sigma}$. After solving for w, the reduction starts by deleting all indexes of w_i that are 0 and the corresponding i-th sign assignments in $\tilde{\Sigma}$ which are now known to be inconsistent (recall that w_i counts the number of zeros of p where the i-th sign assignment is realized). This corresponds to deleting the i-th columns of matrix M. If any columns are deleted, the resulting matrix is no longer square (nor invertible). Thus, the next step finds a basis among the remaining rows of the matrix to make it invertible again (deleting any rows that do not belong to the chosen basis). Deleting the j-th row in this matrix corresponds to deleting the j-th index subset in S.

The reduction step for the matrix equation with $p = x^3 - x$ and $\ell = [3x^3 + 2, 2x^2 - 1]$ is visualized in Fig. 3.4. Naively using the matrix equation for restricted sign determination

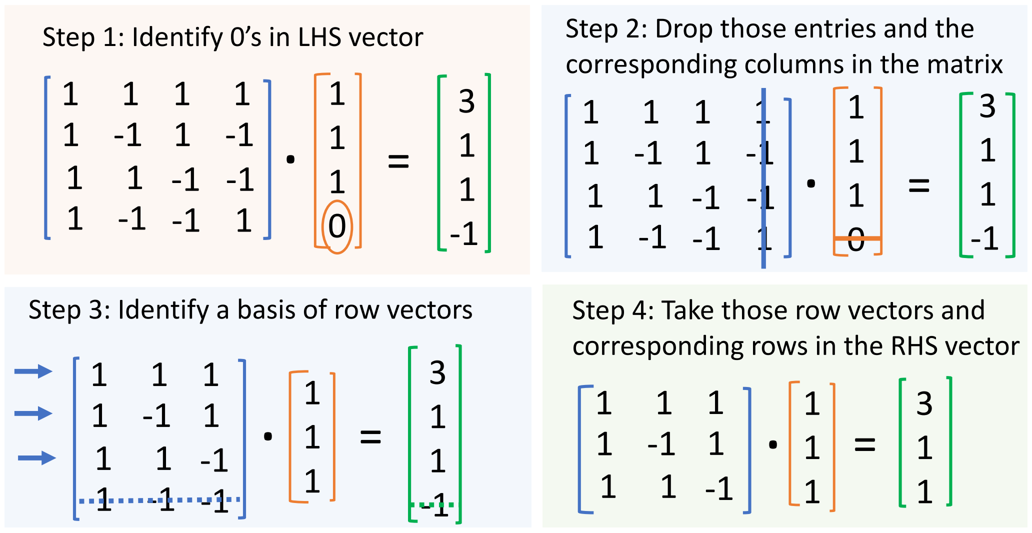

Step 1: Identify 0's in LHS vector
Step 2: Drop those entries and the corresponding columns in the matrix
Step 3: Identify a basis of row vectors
Step 4: Take those row vectors and corresponding rows in the RHS vector

Figure 3.4: Reducing a system.

would require $2^{|\ell|} = 4$ Tarski queries for this example, whereas $2 + 2 + 4 = 8$ queries are required using BKR (2 for each base case, 4 for the combination step). However, for longer lists ℓ, the naive approach requires $2^{|\ell|}$ queries while BKR's reduction step ensures that the number of intermediate consistent sign assignments is bounded by the number of roots of p (and hence $\deg p$) throughout. This difference is shown in Sect. 3.2.3 and is also illustrated by Fig. 3.4, where p has degree 3 and there are 3 consistent sign assignments for ℓ after reduction.

3.2 Formalization

Now that we have set up the theory behind the BKR algorithm, we turn to some details of our formalization: the proofs, extensions to the existing matrix libraries, and the exported code. Our proof builds significantly on existing proof developments in the Archive of Formal Proofs [51, 87, 88]. Isabelle/HOL's builtin search tool and Sledgehammer [68] provided invaluable automation for discovering existing theorems and for finishing (easy) subgoals in proofs. The most challenging part of the formalization, in our opinion, is the reduction step, in no small part because it involves significant linear algebra (further details in Sect. 3.2.2).

3.2.1 Formalizing the Decision Procedure

In this section, we discuss the proofs for our decision procedure in *reverse* order compared to Sect. 3.1; that is, we first discuss the formalization of our algorithm for restricted sign determination `find_consistent_signs_at_roots` before discussing the top-level decision procedures for univariate real arithmetic, `decide_{universal|existential}`. The reader may wish to revisit Sect. 3.1 for informal intuition behind the procedure while reading this section.

Sign Determination at Roots

We combine BKR's base case (Sect. 3.1.3), combination step (Sect. 3.1.3), and reduction step (Sect. 3.1.3) to form our core algorithm `calc_data` for the restricted sign determination problem at the roots of a polynomial. The `calc_data` algorithm takes a real polynomial `p` and a list of polynomials `qs` and produces a 3-tuple `(M, S, Σ)`, consisting of the matrix `M` from the matrix equation, the list of index subsets `S`, and the list of all consistent sign assignments Σ for `qs` at the roots of `p`. Although `M` can be calculated directly from `S` and Σ, it is returned (as part of the algorithm), to avoid redundantly recomputing it at every recursive call.

```
fun calc_data :: "real poly ⇒ real poly list ⇒ (rat mat ×
  (nat list list × rat list list))"
where "calc_data p qs = (let len = length qs in
if len = 0 then (λ(a,b,c).
  (a,b,map (drop 1) c)) (reduce_system p ([1],base_case_info))
else if len ≤ 1 then reduce_system p (qs,base_case_info)
else (let qs1 = take (len div 2) qs; left = calc_data p qs1;
          qs2 = drop (len div 2) qs; right = calc_data p qs2 in
        reduce_system p (combine_systems p (qs1,left) (qs2,right))))"
```

```
definition find_consistent_signs_at_roots :: "real poly ⇒ real poly list
  ⇒ rat list list"
 where "find_consistent_signs_at_roots p qs =
   (let (M,S,Σ) = calc_data p qs in Σ)"
```

The base case where `qs` has length ≤ 1 is handled[4] using the fixed choice of matrix, index subsets, and sign assignments (defined as the constant `base_case_info`) from Sect. 3.1.3. Otherwise, when `length qs` > 1, the list is partitioned into two sublists `qs1, qs2` and the algorithm recurses on those sublists. The outputs for both sublists are combined using the function `combine_systems`, which takes the Kronecker product of the output matrices and concatenates the index subsets and sign assignments as explained in Sect. 3.1.3. Finally, `reduce_system` performs the reduction according to Sect. 3.1.3, removing inconsistent sign assignments and redundant subsets of indices. The top-level procedure is `find_consistent_signs_at_roots`, which returns only Σ (the third component of `calc_data`). The following Isabelle/HOL snippets show its main correctness theorem and important relevant definitions.

```
definition roots :: "real poly ⇒ real set"
  where "roots p = {x. poly p x = 0}"
```

```
definition consistent_signs_at_roots :: "real poly ⇒ real poly list ⇒
  rat list set"
 where "consistent_signs_at_roots p qs = (sgn_vec qs) ` (roots p)"
```

```
theorem find_consistent_signs_at_roots:
  assumes "p ≠ 0"
  assumes "⋀q. q ∈ set qs ⟹ coprime p q"
  shows "set (find_consistent_signs_at_roots p qs) =
    consistent_signs_at_roots p qs"
```

[4]The trivial case where `length qs = 0` is also handled for completeness; in this case, the list of consistent sign assignments is empty if p has no real roots, otherwise, it is the singleton list `[[]]`.

Here, `roots` defines the set of roots of a polynomial `p` (non-constructively), i.e., the set of real values `x` where `p` evaluates to 0 (`poly p x = 0`). Similarly, `consistent_signs_at_roots` returns the set of all sign vectors for the list of polynomials `qs` at the roots of `p`; `sgn_vec` returns the sign vector for input `qs` at a real value and ` is Isabelle/HOL notation for the image of a function on a set. These definitions are not meant to be computational. Rather, they are used to state the correctness theorem that the algorithm `find_consistent_signs_at_roots` (and hence `calc_data`) computes *exactly all* consistent sign assignments for `p` and `qs` for input polynomial `p ≠ 0` and polynomial list `qs`, where every entry in `qs` is `coprime` to `p`.

The proof of `find_consistent_signs_at_roots` is by induction on `calc_data`. Specifically, we prove that the following properties (our inductive invariant) are *satisfied by the base case and maintained by both the combination step and the reduction step*:

1. The signs list is well-defined, i.e., the length of every entry in the signs list is the same as the length of the corresponding `qs`. Additionally, all assumptions on `S` and Σ from the `matrix_equation` theorem from Sect. 3.1.3 hold. (In particular, the algorithm always maintains a distinct list of sign assignments that, when viewed as a set, is a superset of all consistent sign assignments for `qs`.)
2. The matrix `M` matches the matrix calculated from `S` and Σ. (Since we do not directly compute the matrix from `S` and Σ, as defined in Sect. 3.1.3, we need to verify that our computations keep track of `M` correctly.)
3. The matrix `M` is invertible (so $M \cdot w = v$ can be uniquely solved for w).

Some of these properties are easier to verify than others. The well-definedness properties, for example, are quite straightforward. In contrast, matrix invertibility is more complicated to verify, especially after the reduction step; we will discuss this in more detail in Sect. 3.2.2. The inductive invariant establishes that we have a superset of the consistent sign assignments throughout the construction. This is because the base case and the combination step may include extraneous sign assignments. Only the reduction step is guaranteed to produce exactly the set of consistent sign assignments. Thus the other main ingredient in our formalization, besides the inductive invariant, is a proof that the reduction step deletes all inconsistent sign assignments. As `calc_data` always calls the reduction step before returning output, `calc_data` returns exactly the set of all consistent sign assignments, as desired.

Building the Univariate Decision Procedure

To prove the `decision_procedure` theorem from Sect. 3.1.1, we need to establish correctness of `find_consistent_signs`. The most interesting part is formalizing the transformation described in Sect. 3.1.2. We discuss the steps from Sect. 3.1.2 enumerated (1)–(5) below.

(1) Our procedure takes an input list of rational polynomials $G = [g_1, \ldots, g_k]$ and computes a list of their pairwise coprime and squarefree factors[5] $Q = [q_1, \ldots, q_n]$. An efficient method to factor a *single* rational polynomial is formalized in Isabelle/HOL by Divasón et al. [27];

[5]This is actually overkill: we do not necessarily need to *completely* factor every polynomial in G to transform G into a set of pairwise coprime factors. BKR suggest a parallel algorithm based in part on the literature [90] to find a "basis set" of squarefree and pairwise coprime polynomials.

we slightly modified their proof to find factors for a *list* of polynomials while ensuring that the resulting factors are pairwise coprime, which implies that their product $\prod_i q_i$ is squarefree.

(2) This step makes n calls to `find_consistent_signs_at_roots`, one for each $Q \setminus \{q_i\}$.

(3) We choose the polynomial $p = (x - \mathit{crb}(\prod_i q_i))(x + \mathit{crb}(\prod_i q_i))(\prod_i q_i)'$, where $(\prod_i q_i)'$ is the formal polynomial derivative of $\prod_i q_i$ and $\mathit{crb}(\prod_i q_i)$ is a computable positive integer with larger magnitude than any real root of $\prod_i q_i$. The choice of $\mathit{crb}(\prod_i q_i)$ uses a proof of the Cauchy root bound [1, Section 10.1] by Thiemann and Yamada [88]. We prove that p satisfies the four properties of step (3) from Sect. 3.1.2:

   (i) Since $\prod_i q_i$ is squarefree, $(\prod_i q_i)'$ is coprime with $\prod_i q_i$ and, thus, also coprime with each of the q_i's. Because $\mathit{crb}(\prod_i q_i)$ is strictly larger in magnitude than all of the roots of the roots of the q_i's, it follows that p is also coprime with all of the q_i's.

   (ii) By Rolle's theorem[6] (which is already formalized in Isabelle/HOL's standard library), $(\prod_i q_i)'$ has a root between every two roots of $\prod_i q_i$ and therefore p also has a root in every interval between any two roots of the q_i's.

   (iii) and *iv)* This choice of p has roots at $-\mathit{crb}(\prod_i q_i)$ and $\mathit{crb}(\prod_i q_i)$, which are respectively smaller and greater than all roots of the q_i's.

(4) Each polynomial q_i is sign invariant between its roots.[7] Accordingly, the q_i's are sign invariant between the roots of $\prod_i q_i$ (and to the left/right of all roots of the q_i's).

(5) We use the `find_consistent_signs_at_roots` algorithm with Q and our chosen p.

Putting the pieces together, we verify that `find_consistent_signs` finds exactly the consistent sign assignments for its input polynomials. The `decision_procedure` theorem follows by induction over the `fml` type representing formulas of univariate real arithmetic and our formalized semantics for those formulas.

3.2.2 Matrix Library

Matrices feature prominently in our algorithm: the combination step uses the Kronecker product, while the reduction step requires matrix inversion and an algorithm for finding a basis from the rows (or, equivalently, columns) of a matrix. There are a number of linear algebra libraries available in Isabelle/HOL [28, 81, 87], each building on a different underlying representation of matrices. We use the formalization by Thiemann and Yamada [87] as it provides most of the matrix algorithms required by our decision procedure and supports efficient code extraction [87, Section 1]. Naturally, any such choice leads to tradeoffs; we now detail some challenges of working with the library and some new results we prove.

[6]For a differentiable function $f : \mathbb{R} \mapsto \mathbb{R}$ with $f(a) = f(b)$, $a < b$, there exists $a < z < b$ where $f'(z) = 0$.

[7]By the intermediate value theorem (which is already formalized in Isabelle/HOL's standard library), if q_i changes sign, e.g., from positive to negative, between two adjacent roots, then there exists a third root in between those adjacent roots, which is a contradiction.

Combination Step: Kronecker Product

We define the Kronecker product for matrices `A`, `B` over a `ring` as follows:

definition `kronecker_product :: "'a :: ring mat ⇒ 'a mat ⇒ 'a mat"`
where `"kronecker_product A B = (`
```
let ra = dim_row A; ca = dim_col A; rb = dim_row B; cb = dim_col B in
  mat (ra * rb) (ca * cb)
    (λ(i,j). A $$ (i div rb, j div cb) * B $$ (i mod rb, j mod cb)))"
```

Matrices with entries of type `'a` are constructed with `mat m n f`, where `m, n :: nat` are the number of rows and columns of the matrix respectively, and `f :: nat × nat ⇒ 'a` is such that `f i j` gives the matrix entry at position `i`, `j`. Accordingly, `M $$ (i,j)` extracts the `(i,j)`-th entry of matrix `M`, and `dim_row, dim_col` return the number of rows and columns of a matrix respectively.

We prove basic properties of our definition of the Kronecker product: it is associative, distributes over addition, and satisfies the mixed-product identity for matrices `A`, `B`, `C`, `D` with compatible dimensions (for `A * C` and `B * D`):

```
kronecker_product (A * C) (B * D) =
                     (kronecker_product A B) * (kronecker_product C D).
```

The mixed-product identity implies that the Kronecker product of invertible matrices is invertible. Briefly, for invertible matrices `A`, `B` with respective inverses A^{-1}, B^{-1}, the mixed product identity gives:

$$\texttt{(kronecker\_product A B) * (kronecker\_product } A^{-1}\ B^{-1}\texttt{)} = \texttt{kronecker\_product } (A * A^{-1})\ (B * B^{-1}) = I,$$

where `I` is the identity matrix—so `kronecker_product A B` and `kronecker_product` A^{-1} B^{-1} are inverses. We use this to prove that the matrix obtained by the combination step is invertible (part of the inductive hypothesis from Sect. 3.2.1).

Remark 4. *Prathamesh [73] formalized Kronecker products for Isabelle/HOL's default matrix type. For computational purposes, we provide a new formalization that is compatible with the matrix representation of Thiemann and Yamada [87].*

Reduction Step: Gauss–Jordan and Matrix Rank

Our reduction step makes extensive use of the Gauss–Jordan elimination algorithm formalized by Thiemann and Yamada [86]. First, we use matrix inversion based on Gauss–Jordan elimination to invert the matrix M in the matrix equation (Sect. 3.1.3 and Step 1 in Fig. 3.4). We also contribute new proofs surrounding their Gauss–Jordan elimination algorithm in order to use it to extract a basis from the rows (equivalently columns) of a matrix (Step 3 in Fig. 3.4).

Suppose that an input matrix `A` has more rows than columns, e.g., the matrix in Step 2 of Fig. 3.4. The following definition of `rows_to_keep` returns a list of (distinct) row indices of `A`.

definition `rows_to_keep:: "('a::field) mat ⇒ nat list"` **where**
`"rows_to_keep A = map snd (pivot_positions (gauss_jordan_single (`A^T`)))"`

Here, `gauss_jordan_single` returns the row-reduced echelon form (RREF) of `A` after Gauss–Jordan elimination and `pivot_positions` finds the positions, i.e., `(row, col)` pairs, of the first nonzero entry in each row of the matrix; both are existing definitions from the library by Thiemann and Yamada [86]. Our main new result for `rows_to_keep` is:

lemma *rows_to_keep_rank:*
  assumes *"dim_col A ≤ dim_row A"*
  shows *"vec_space.rank (length (rows_to_keep A)) (take_rows A (rows_to_keep A)) = vec_space.rank (dim_row A) A"*

Here `vec_space.rank n M` (defined by Bentkamp [87]) is the finite dimension of the vector space spanned by the columns of `M`. Thus, the lemma says that keeping only the pivot rows of matrix `A` (with `take_rows A (rows_to_keep A)`) preserves the rank of `A`. At a high level, the proof of `rows_to_keep_rank` is in three steps:

1. First, we prove a version of `rows_to_keep_rank` for the pivot *columns* of a matrix and where `A` is assumed to be a matrix in RREF. The RREF assumption for `A` enables direct analysis of the shape of its pivot columns.
2. Next, we lift the result to an arbitrary matrix `A`, which can always be put into RREF form by `gauss_jordan_single`.
3. Finally, we formalize the following classical result that column rank is equal to row rank: *vec_space.rank (dim_row A) A = vec_space.rank (dim_col A) (A^T).* We lift the preceding results for pivot columns to also work for pivot rows by matrix transposition (pivot rows of matrix `A` are the pivot columns of the transpose matrix A^T).

To complete the proof of the reduction step, recall that the matrix in Step 2 of Fig. 3.4 is obtained by dropping columns of an invertible matrix. The resulting matrix has full column rank but more rows than columns. We show that when `A` in `rows_to_keep_rank` has full column rank (its `rank` is `dim_col A`) then `length (rows_to_keep A) = dim_col A` and so the matrix consisting of pivot rows of `A` is square, has full rank, and is therefore invertible.

Remark 5. *Divasón and Aransay formalized the equivalence of row and column rank for Isabelle/HOL's default matrix type [29] while we have formalized the same result for Bentkamp's definition of matrix rank [87]. Another technical drawback of our choice of libraries is the locale argument* `n` *for* `vec_space`*. Intuitively (for real matrices) this carves out subsets of $\mathbb{R}^n$ to form the vector space spanned by the columns of* `M`*. Whereas one would usually work with* `n` *fixed and implicit within an Isabelle/HOL locale, we pass the argument explicitly here because our theorems often need to relate the rank of vector spaces in $\mathbb{R}^m$ and $\mathbb{R}^n$ for $m \neq n$. This negates some of the automation benefits of Isabelle/HOL's locale system.*

3.2.3 Code Export

We export our decision procedure to Standard ML, compile with `mlton`, and test it on 10 microbenchmarks from [54, Section 8]. While we leave extensive experiments for future work since our implementation is unoptimized, we compare the performance of our procedure using BKR sign determination (Sections 3.1.3–3.1.3) versus an *unverified* implementation that naively uses the matrix equation (Sect. 3.1.3). We also ran Li *et al.*'s `univ_rcf` decision procedure [54] which can be directly executed as a proof tactic in Isabelle/HOL (code kindly provided by Wenda

Li). The benchmarks were ran on an Ubuntu 18.04 laptop with 16GB RAM and 2.70 GHz Intel Core i7-6820HQ CPU. Results are in Table 3.1.

The most significant bottleneck in our current implementation is the computation of Tarski queries $N(p, q)$ when solving the matrix equation. Recall for our algorithm (Sect. 3.1.3) the input q to $N(p, q)$ is a *product* of (subsets of) polynomials appearing in the inputs. Indeed, Table 3.1 shows that the algorithm performs well when the factors have low degrees, e.g., ex1, ex2, ex4, and ex5. Conversely, it performs poorly on problems with many factors and higher degrees, e.g., ex3, ex6, and ex7. Further, as noted in experiments by Li and Paulson [53], the Sturm-Tarski theorem in Isabelle/HOL currently uses a straightforward method for computing remainder sequences which can also lead to significant (exponential) blowup in the bitsizes of rational coefficients of the involved polynomials. This is especially apparent for ex6 and ex7, which have large polynomial degrees and high coefficient complexity; these time out without completing even a single Tarski query. From Table 3.1, the BKR approach successfully reduces the number of Tarski queries as the number of input factors grows—the number of queries for BKR is dependent on the polynomial degrees and the number of consistent sign assignments, while the naive approach always requires exactly $(\frac{n}{2} + 1)2^n$ queries for n factors[8] (which are reported in Table 3.1 whether completed or not). On the other hand, there is some overhead for smaller problems, e.g., ex1, ex3, that arises from the recursion in BKR.

The `univ_rcf` tactic relies on an external solver (we used Mathematica 12.1.1) to produce *untrusted* certificates which are then formally checked (by reflection) in Isabelle/HOL [54]. The untrusted solver (Mathematica) is used in two key ways: first, for real root isolation (the solver returns a list of points that it claims are roots to Isabelle/HOL), second, to decide satisfiable existentially quantified formulas (as Li et al. point out, CAD may not even be used in these instances—sometimes these formulas can be decided in external tools by incomplete methods that are faster than CAD, so there is considerable benefit to outsourcing these problems). This procedure is optimized and efficient: except for ex7 where the tactic timed out, most of the time (roughly 3 seconds per example) is actually spent to start an instance of the external solver.

An important future step, e.g., to enable use of our procedure as a tactic in Isabelle/HOL, is to avoid coefficient growth by using pseudo-division [63, Section 3] or more advanced techniques: for example, using subresultants to compute polynomial GCDs (and thereby build the remainder sequences) [33]. Pseudo-division is also important in the multivariate generalization of BKR (as we will see shortly in Chapter 4), where the polynomial coefficients of concern are themselves (multivariate) polynomials rather than rational numbers. The pseudo-division method has been formalized in Isabelle/HOL [54]; this formalization was not publicly available at the time of our univariate work, but it later was added to Isabelle's Archive of Formal Proofs (and we use it in our multivariate formalization).

As a final comment, it is worth noting that Renegar-based sign determination is likely to almost always be significantly slower than BKR-based sign determination, as univariate Renegar involves Tarski queries with polynomials of higher degree. This is of consequence for the multivariate algorithm discussed in the next chapter (Chapter 4), as this algorithm is based on

[8]For n factors, Sect. 3.1.2's transformation yields n restricted sign determination subproblems involving $n - 1$ polynomials each and one subproblem involving n polynomials. Using naive sign determination to solve all of these subproblems requires $n(2^{n-1}) + 2^n = (\frac{n}{2} + 1)2^n$ Tarski queries in total.

Formula	**#Poly**	**#Factor**	**#$N(p,q)$ (Naive)**	**#$N(p,q)$ (BKR)**	**Time (Naive)**	**Time (BKR)**	**Time ([54])**
ex1	4 (12)	3 (1)	20	31	0.003	0.006	3.020
ex2	5 (6)	7 (1)	576	180	5.780	0.442	3.407
ex3	4 (22)	5 (22)	112	120	1794.843	1865.313	3.580
ex4	5 (3)	5 (2)	112	95	0.461	0.261	3.828
ex5	8 (3)	7 (3)	576	219	28.608	8.333	3.806
ex6	22 (9)	22 (8)	50331648	-	-	-	6.187
ex7	10 (12)	10 (11)	6144	-	-	-	-
ex1 $\wedge$ 2	9 (12)	9 (1)	2816	298	317.432	3.027	3.033
ex1 $\wedge$ 2 $\wedge$ 4	13 (12)	12 (2)	28672	555	-	51.347	3.848
ex1 $\wedge$ 2 $\wedge$ 5	16 (12)	14 (3)	131072	826	-	436.575	3.711

Table 3.1: Comparison of decision procedures using naive and BKR sign determination and Li *et al.*'s `univ_rcf` tactic in Isabelle/HOL [54]. All formulas are labeled following [54, Section 8]; formulas with $\wedge$ indicate conjunctions of the listed examples. Columns: **#Poly** counts the number of distinct polynomials appearing in the formula (maximum degree among polynomials in parentheses), **#Factor** counts the number of distinct factors from (1) in Sect. 3.1.2 (maximum degree among factors in parentheses), **#$N(p,q)$** counts the number of Tarski queries made by each approach, and **Time** reports time taken (seconds, 3 d.p.) for each decision procedure to run to completion. Cells with - indicate a timeout after 1 hour.

univariate Renegar (cross-reference Remark 3), and so further optimizing the computation of Tarski queries is of considerable significance.

3.3 Related Work

Our work fits into the larger body of formalized univariate decision procedures. Of these, the most closely related are Li *et al.*'s formalization of a CAD-based univariate QE procedure in Isabelle/HOL [54] and the *`tarski`* univariate QE strategy formalized in PVS [63]. We discuss each in detail.

The univariate CAD algorithm underlying Li *et al.*'s approach [54] decomposes $\mathbb{R}$ into a set of sign-invariant regions, so that every polynomial of interest has constant sign within each region. A real algebraic sample point is chosen from every region, so the set of sample points captures all of the relevant information about the signs of the polynomials of interest *for the entirety of* $\mathbb{R}$. BKR (and Renegar) take a more indirect approach, relying on consistent sign assignments which merely indicate the *existence* of points with such signs. Consequently, although CAD will be faster in the average case, BKR and CAD have different strengths and weaknesses. For example, CAD works best on full-dimensional decision problems [57], where only rational sample points are needed (this allows faster computation than the computationally expensive real algebraic numbers that general CAD depends on).

The Sturm-Tarski theorem is also invoked in Li *et al*'s procedure to decide the sign of a univariate polynomial at a point (using only rational arithmetic) [54, Section 5]. (This was later extended to bivariate polynomials by Li and Paulson [52].) This is theoretically similar to our procedure to find the consistent sign assignments for $q_1, \ldots, q_n$ at the roots of p, as both rely on the mathematical properties of Tarski queries; however, for example, we do not require isolating the real roots of p within intervals. This difference reflects our different goals: theirs is to encode algebraic numbers in Isabelle/HOL and to verify univariate CAD, ours is to perform full sign determination with BKR. It is beneficial for us that we are able to completely avoid calculations with real algebraic numbers. These computations, which are inherent to CAD-based approaches, can be quite costly; typically, they are performed with subterfuges for better efficiency. These subterfuges soundly approximate exact arithmetic, but are often somewhat intricate [54].

PVS's `tarski` uses Tarski queries and a version of the matrix equation to solve univariate decision problems [63]. Unlike our work, `tarski` has already been optimized in significant ways; for example, `tarski` computes Tarski queries with pseudo-divisions. However, `tarski` does not maintain a *reduced* matrix equation as our work does. Further, `tarski` was designed to solve existential conjunctive formulas and requires a potentially costly DNF transformation to be able to solve arbitrary problems [18], whereas our development solves problems directly without requiring a DNF tranformation.

In addition, as previously mentioned, our work is somewhat similar in flavor to Cohen and Mahboubi's (multivariate) formalization of Tarski's algorithm [15]. In particular, the characterization of the matrix equation and the parts of the construction that do not involve reduction share considerable overlap, as BKR derives the idea of the matrix equation from Tarski [2]. However, the reduction step is not present in Tarski's algorithm and is a distinguishing feature of our work when compared to Cohen and Mahboubi's formalization.[9] This again reflects different goals: namely, our goal was to include some of the transformative insights of BKR in a (univariate, and later multivariate) QE algorithm.

3.4 Takeaways and Future Directions

This chapter describes how we have verified the correctness of a decision procedure for univariate real arithmetic in Isabelle/HOL. This formalization lays the groundwork for several future directions, including:

1. Optimizing the current formalization and adding parallelism.
2. Proving that the univariate sign determination problem is decidable in NC [2, 75] and other complexity-theoretic results. This (ambitious) project would require developing a complexity framework that is compatible with all of the libraries we use.
3. Verifying a multivariate sign determination algorithm and decision procedure based on BKR. The next chapter of this thesis discusses our progress towards this goal. Additionally, as mentioned previously, multivariate BKR has an error in its complexity analysis; variants

[9]Actually, Cohen later extended his formalization of Tarski's algorithm to add in a reduction step to the construction of the matrix equation. This work is unpublished and, to our knowledge, was not publicly available prior to the publication of the work in the present chapter of the thesis, but Cohen has since made a writeup of it is available on his webpage [14]. We will discuss this work again in the next chapter of the thesis, when we compare it to our formalization of a complete multivariate QE algorithm.

of decision procedures for $\mathrm{FOL}_{\mathbb{R}}$ based on BKR's insight that attempt to mitigate this error could eventually be formalized for useful points of comparison. Two of particular interest are that of Renegar [75], who develops a complete QE algorithm, and that of Canny [10], in which coefficients can involve some more general terms, like transcendental functions.

Chapter 4

Verifying a Complete Multivariate QE Algorithm

This chapter discusses our formalization of a hybrid mixture of Tarski's original QE algorithm and the Ben-Or, Kozen, and Reif algorithm. Verified complete QE algorithms are rare, and so our formalization is a considerable addition to the body of existing work (see Sect. 1.2).

More specifically, we see this work as making the following contributions: (1) Our work is the first complete multivariate QE algorithm formalized in Isabelle/HOL. (2) To our knowledge, it is the first formalized multivariate QE algorithm to include insights from BKR, and it is a first step towards a less complex verified algorithm (e.g. in the style of Renegar [75]), which could ideally complement an eventual formalized algorithm based on CAD. (3) Because much of the source material is either sparsely written (e.g. [2]) or highly mathematical (e.g. [1, 75]), it was not a priori obvious what the formalized algorithm should look like (this formalization barrier is discussed in Sect. 4.2.1). The rigorous nature of verification forced us to clearly identify the essential building blocks of the algorithm: In our formalization, *all* definitions are mathematically precise and verifiable, and *all* their correctness properties are identified and proved.

The formalization is approximately 8500 lines of code and is available on the Archive of Formal Proofs (AFP) [48]. It includes various advances to Isabelle/HOL's existing libraries, particularly the library for multivariate polynomials, which could help pave the way for future multivariate QE algorithms in Isabelle/HOL.

Collaborators. The material in this chapter is based upon joint work with Yong Kiam Tan and André Platzer which appeared at CPP 2023 [49].

4.1 Quantifier Elimination

Our QE algorithm works by eliminating one quantifier at a time. Hence, if we have polynomials in $n+1$ variables, we can consider them as univariate polynomials in a variable of interest with coefficient polynomials in n variables. For example, if x is our variable of interest, then we can treat $3xyz^2 + 6x^2wv + 5xy + 1$ as the following polynomial in x: $(6wv)x^2 + (3yz^2 + 5y)x + 1$. For clarity, and WLOG, we assume throughout this section that our variable of interest is x.

The multivariate algorithm closely mirrors the univariate algorithm (though formalizing it is considerably more challenging). Working with multivariate polynomials is substantially harder than univariate polynomials (both mathematically and formally). Mathematically, multivariate polynomials are more complex than univariate polynomials—for example, they may have infinitely many roots, and (polynomial) coefficients may be zero in certain valuations. Certain constructs, like the Cauchy root bound (see Sect. 3.2.1), apply only to univariate polynomials and not to multivariate polynomials. Formally, the representation of multivariate polynomials is more difficult—while it is very natural to represent a univariate polynomials by a list of its coefficients, multivariate polynomials have multiple possible representations and it is often convenient to be able to switch between representations—and (in theorem provers) multivariate polynomial libraries are often less well-developed than univariate polynomial libraries.

The key component of both multivariate and univariate BKR is a *sign-determination algorithm* which is concerned with finding all *consistent sign assignments* to a set of polynomials $\{q_1, \ldots, q_k\}$. A *sign assignment* is a mapping that assigns each polynomial to a *sign*, i.e. positive, zero, or negative (represented by $1, 0$, and -1). A sign assignment is called *consistent* if it is actually realized at some real point.

At the heart of the sign-determination algorithm that we formalize is a *matrix equation* that is capable of storing sign information for a set of polynomials in variables $x, y_1, \ldots, y_n$, under a set of assumptions on polynomials in $y_1, \ldots, y_n$. Our overall quantifier elimination algorithm takes a formula and identifies the polynomials that occur in the formula. It then generates a number of matrix equations, each of which captures some sign information for the polynomials, subject to some list of assumptions. Collectively, it is important that the generated matrix equations have exhaustive assumptions—in the sense that for every possible set of assumptions, there is at least one corresponding matrix equation. We call sets of assumptions *branches*. Branches are refined throughout the construction with additional assumptions until each multivariate matrix equation has assumptions that generate a unique matrix equation. Initial branches, which are not fully refined, may still have multiple associated matrix equations.

WLOG, we assume that we are eliminating a $\forall$ quantifier (because $\exists$ quantifiers can be transformed into $\forall$ quantifiers with appropriate negations). We do some initial branching (this is needed to guide the computations of the matrix equations), and for each branch, we check whether *all* of the associated matrix equations describe a sign assignment on our polynomials that satisfies the original formula. We filter our initial branches to pick out the ones that satisfy this property. Finally, we return a disjunction of all assumptions of the initial branches in this filtered list.

Fig. 4.1 visualizes how this QE algorithm works on an example. We begin with formula $\exists y. \forall x. (xy^2 > 0 \vee y^2 + x^2 > 0)$, where our focus is on eliminating the $\forall x$ quantifier. We first identify the polynomials of interest in this formula and view them as univariate polynomials in x (with coefficients that are polynomials in y): these are $y^2 x$ and $x^2 + y^2$. Next, we determine all consistent sign assignments to these polynomials of interest given all *possible*[1] sign assumptions on y^2, where y^2 is significant because it is the *leading coefficient* of $y^2 x$ (technically our

[1]Here, we differ from the BKR algorithm, which would branch on all *consistent* sign assumptions on y^2. That is, we consider a branch where $y^2 < 0$, because this is a possible (but inconsistent) sign assumption: even though y^2 is never negative, our algorithm does not discern this when branching.

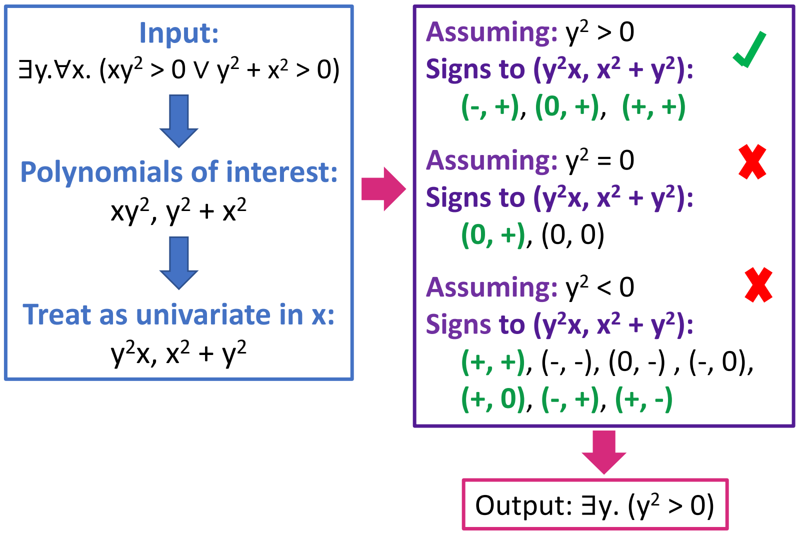


Figure 4.1: A visual overview of the multivariate QE algorithm.

algorithm will do some additional and unnecessary branching, but for the clarity of this example we focus on the branch on y^2; see Sect. 4.1.1 for a more in-depth discussion of the branching). Internally, our algorithm performs sign determination using matrix equation constructions (but this is not pictured in the figure). We then pick out the sign assignments that solve our original QE problem—that is, we are looking for one of our polynomials of interest, y^2x or x^2+y^2, to be positive. Signs that satisfy this condition are pictured in green. Then, we filter our branches to find the ones where *every* sign assignment satisfies the original QE problem. This happens only in the branch where y^2 is assumed to be positive. This means that $y^2 > 0$ is logically equivalent to $\forall x.(y^2x > 0 \vee x^2 + y^2 > 0)$, which means that $\exists y.\forall x.(xy^2 > 0 \vee y^2 + x^2 > 0)$ is logically equivalent to $\exists y.(y^2 > 0)$, whose quantifier $\exists y$ can be eliminated further.

If our original QE question was instead $\exists y.\forall x.(xy^2 \geq 0 \vee x^2 + y^2 > 0)$, then both the branch with assumption $y^2 > 0$ and the branch with assumption $y^2 = 0$ would satisfy our QE problem. This means that the disjunction $y^2 > 0 \vee y^2 = 0$ is logically equivalent to $\forall x.(y^2x \geq 0 \vee x^2 + y^2 > 0)$, and so our output in this case would be $\exists y.(y^2 > 0 \vee y^2 = 0)$.

Here it is important to note that there are many logically equivalent outputs to any given QE problem. For example, if our original QE question were $\forall x.((xy^2 = 0 \wedge x^2 + y^2 = 0) \vee (xy^2 = 0 \wedge x^2 + y^2 < 0))$, then two possible correct outputs that are logically equivalent are $y^2 = 0$, and $y^2 < 0 \vee y^2 = 0$. Here, $y^2 = 0$ is the simplest output. While the output of our QE algorithm is always logically correct, it is *not* guaranteed to be in the simplest form. In particular, assumptions for branches that are inconsistent will often be included in the final disjunction. This has no impact on logical correctness, but it does impact formula complexity (and thus potentially also computational complexity), so it would be potentially impactful to verify the true BKR algorithm without the extra branching in a future work.

We now turn to more detailed descriptions of the sign determination procedure, the multivariate matrix equation, and the full quantifier elimination procedure.

4.1.1 Sign Determination

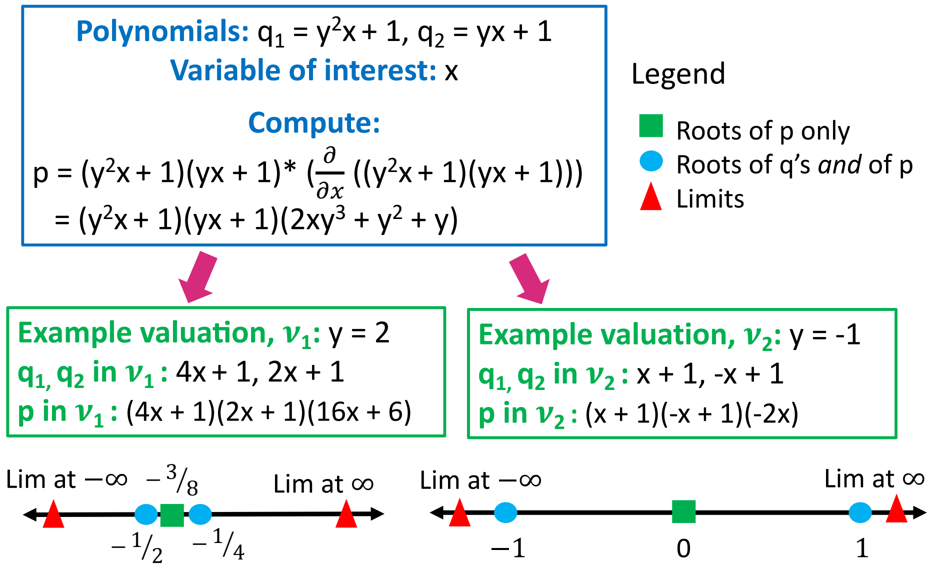
Finding sign information for polynomials $q_1, \dots q_k$ in variables $x, y_1, \dots, y_n$ is, on the surface, a continuous problem—the most obvious way to determine the sign information would be to evaluate $(q_1, \dots, q_k)$ on $\mathbb{R}^k$, which is clearly not computationally viable. To account for this, BKR and Renegar reduce the sign-determination problem to a problem with the following format: find sign information for $q_1, \dots, q_k$ *at the roots* of some cleverly chosen polynomial p. This problem is clearly computationally viable for *univariate* polynomials, because polynomials in one variable only have finitely many roots. It is a (non-obvious) key insight that it is also computationally viable for *multivariate* polynomials [2, 75]. Intuitively, the output of the univariate algorithm only depends on the *signs* of the real polynomial coefficients and not on the actual *values* of those coefficients. Thus, the algorithm lifts to the multivariate case by casing on an exhaustive set of *sign assumptions* on the (multivariate) polynomial coefficients in variables $y_1, \dots, y_n$.

In our multivariate setting, $p = (\prod q_i) \cdot \frac{\partial}{\partial x}(\prod q_i)$ is chosen for p. To see what makes this particular polynomial useful, consider some valuation ν on $y_1, \dots, y_n$ (i.e., some assignment of $y_1, \dots, y_n$ to real values). Let $\nu(f)$ denote the evaluation of polynomial f in valuation ν; note that $\nu(f)$ is univariate in x. Now, the roots of $\nu(p) = (\prod \nu(q_i)) \cdot \frac{\partial}{\partial x}(\prod \nu(q_i)) = (\prod \nu(q_i)) \cdot \frac{d}{dx}(\prod \nu(q_i))$ contain all of the roots of the $\nu(q_i)$'s (since each $\nu(q_i)$ divides $\nu(p)$), as well as sample points from intervals between the roots (by Rolle's theorem [19]). Because these intervals are *sign-invariant*—that is, no $\nu(q_i)$ changes sign in any of these intervals, since no $\nu(q_i)$ can change sign without passing through a root—sign information at a single point within any of these intervals is *representative* of sign information for the entire interval. So, we see that the only intervals which the roots of $\nu(p)$ do not adequately cover are the extreme intervals—the leftmost and rightmost, which lie beyond any of the roots of $\nu(p)$—for which sign information can be computed with a limit calculation on the $\nu(q_i)$'s.[2] So, this polynomial p allows a natural lifting from the univariate QE algorithm to the multivariate case, but the correctness justification needs an extensive covering of the influence of all possibilities for valuation ν.

This is visualized in Fig. 4.2. Here, we have polynomials $q_1 = y^2x + 1$ and $q_2 = yx + 1$, so $p = (y^2x + 1)(yx + 1)(2xy^3 + y^2 + y)$. For the purposes of illustration, we consider two sample valuations: in ν_1, we set $y = 2$, and in ν_2, we set $y = -1$. As depicted, in both valuations, to find sign information for q_1 and q_2, it suffices to find sign information for q_1 and q_2 at the roots of p and the limit points.

We formalize this procedure for sign determination in the `sign_determination` function. The first input to this function is a list of polynomials `qs` of type `rmpoly`, where `rmpoly` is our abbreviation for `real mpoly poly`. Here, `poly` is Isabelle/HOL's type for univariate polynomials, `mpoly` is the type for multivariate polynomials, and `real` is the type for real numbers, so an `rmpoly` is a univariate polynomial whose coefficients are real multivariate polynomials. Say initially we have polynomials in variables $x, y_1, \dots, y_n$; then type `rmpoly` arises when we treat those polynomials as being univariate in x with coefficients in $y_1, \dots, y_n$. Unlike in computer algebra, these polynomials are not restricted to have any particular representation; rather,

[2]In the formalization of the univariate case discussed in the previous chapter, the polynomial p was chosen so as to directly sample from these intervals by using the Cauchy root bound, a mathematical quantity that bounds the roots of a set of univariate polynomials. This followed BKR's original work [2]. However, since the Cauchy root bound is for univariate polynomials only, we must work instead with limit computations as Renegar does [75].



Polynomials: $q_1 = y^2x + 1$, $q_2 = yx + 1$
Variable of interest: x
Compute:
$p = (y^2x + 1)(yx + 1) * (\frac{\partial}{\partial x}((y^2x + 1)(yx + 1)))$
$= (y^2x + 1)(yx + 1)(2xy^3 + y^2 + y)$
Legend
🟩 Roots of p only
● Roots of q's *and* of p
▲ Limits
Example valuation, ν_1: y = 2
q_1, q_2 in ν_1: 4x + 1, 2x + 1
p in ν_1: (4x + 1)(2x + 1)(16x + 6)
Example valuation, ν_2: y = -1
q_1, q_2 in ν_2: x + 1, -x + 1
p in ν_2: (x + 1)(-x + 1)(-2x)
Lim at $-\infty$
$-3/8$
Lim at ∞
$-1/2$
$-1/4$
Lim at $-\infty$
Lim at ∞
-1
0
1

Figure 4.2: An example of sign determination.

they are elements of the free term algebra. The next input to *sign_determination* is a list of initial assumptions of type *(real mpoly × rat) list*, which we abbreviate as *assumps*. Here, *rat* is Isabelle/HOL's type for rational numbers, and so each assumption in the list pairs a real multivariate polynomial with an associated rational number that indicates a sign condition on the polynomial (0, 1, or -1). This type is useful in specifying any known sign information on polynomials in $y_1, \dots, y_n$. The output of *sign_determination* is a list of pairs of assumptions and associated sign assignments to *qs*. Each sign assignment has type *rat list*.[3] The assumptions have type *assumps* (for the same reason as before), and as each assumption may have multiple associated sign assignments, each assumption is paired with a *list* of associated sign assignments, as demonstrated by the *assumps × (rat list list)* type. The output, of type *(assumps × (rat list list)) list*, contains an exhaustive set of assumptions (in order to capture *all* consistent sign assignments for the q_i's).

```
fun sign_determination:: "rmpoly list ⇒ assumps ⇒
  (assumps × rat list list) list"
  where "sign_determination qs assumps =
  (let branches = lc_assump_generation_list qs assumps in
   concat (map (λbranch. let
       poly_p_branch = poly_p_in_branch branch;
       (pos_limit_branch, neg_limit_branch) =
         limit_points_on_branch branch;
       mat_eq_signs_on_branch = extract_signs
         (calculate_data_assumps_M poly_p_branch
         (snd branch) (fst branch)) in
```

[3]Technically, we could use *int list* for sign assignments, since each member of the sign assignment list is 1, 0, or -1, but as noted in the previous chapter, it is easier to work with *rat list* in the matrix equation construction.

```
    map (λ(a, signs).
      (a, pos_limit_branch#neg_limit_branch#signs))
       mat_eq_signs_on_branch) branches))"
```

Here, the `lc_assump_generation_list` function generates an exhaustive list of *possible* branches, `branches`, that contain assumptions on the signs of the leading coefficients of the input polynomials `qs`. An important subtlety is that the leading coefficient of the polynomial q_i may be different in different branches. For example, the leading coefficient of $(y+1)x^2+yx+2$ is $y+1$ in a branch where $y+1$ is assumed to be nonzero, y in a branch where $y+1$ is zero and y is assumed to be nonzero, and 2 in a branch where both $y+1$ and y are assumed to be zero. To best account for this subtlety, each element of `branches` contains both the generated assumptions (which determine the branch) *and* a list of polynomials which contains a simplified version of the `qs`: to be precise, $q_i = c_1x^{d_1} + \cdots + c_mx^{d_m}$ simplifies to $c_jx^{d_j} + \cdots + c_mx^{d_m}$ iff $c_1, \ldots, c_{j-1}$ are all assumed to be zero and c_j is assumed to be nonzero. For example, given a list of input polynomials $[(y+1)x^2+yx+2, y^2+(y+1)x^5]$, an element of `branches` could be: $([(y+1,0),(y,1),(y^2,1)],[yx+2,y^2+(y+1)x^5])$. The list of assumptions $[(y+1,0),(y,1),(y^2,1)]$ specifies that, in this branch, $y+1$ is assumed to be 0 and y and y^2 are assumed to be positive. Under these assumptions, $(y+1)x^2+yx+2$ simplifies to $yx+2$ and $y^2+(y+1)x^5$ simplifies to $y^2+(y+1)x^5$ (as the purpose of the simplification is to determine the leading coefficient, it is not mission critical to fully simplify $y^2+(y+1)x^5$ to y^2, and our code is not optimized to do so).

Currently, `lc_assump_generation_list` naively generates branches by branching on *all possible* sign assignments to the leading coefficients, rather than on all *consistent* ones as BKR would. Thus, branches with inconsistent assumptions can be generated: for example, the branch $([(y+1,0),(y,1),(y^2,-1)],\ [yx+2,y^2+(y+1)x^5])$ could be generated by the function `lc_assump_generation` despite its inconsistent assumptions (y^2 is assumed to be negative). Exploring these inconsistent branches trades off some efficiency in favor of ease of verification, and pruning inconsistent branches early on (as BKR does) would be desirable in a future verified algorithm. Also, although `lc_assump_generation_list` takes an input list of assumptions, `assumps`, as an argument, it does not enforce consistency of the output branches with `assumps`; however, before splitting on the sign of a polynomial f, it will check whether `assumps` already contains sign information for f.

Branching on the signs of the leading coefficients of the `qs` provides important information for two reasons: First, because these signs are relevant for the matrix equation computation (Sect. 4.1.2); and second, because knowing the sign of the first non-zero leading coefficient for every q_i allows us to easily compute the signs at the limit points.[4]

The `sign_determination` function maps over `branches`, and for each computes the polynomial $p = (\prod q_i) \cdot \frac{\partial}{\partial x}(\prod q_i)$, stored in `poly_p_branch` (cross reference Fig. 4.2). Although it would suffice to compute p beforehand, and then simplify it appropriately on each branch given the associated assumptions (for example, in a branch where $y=0$, $q_1 = y^2x+1$, and $q_2 = yx+1$, the polynomial $p = (y^2x+1)(yx+1)(2xy^3+y^2+y)$ simplifies to $p=0$), it is more direct to compute p in each branch.[5] That is, given $q_1 = y^2x+1$, and $q_2 = yx+1$, if in a given branch we

[4]The sign of q_i at ∞ equals the sign of its leading coefficient, whereas the sign of q_i at $-\infty$ is the sign of its leading coefficient multiplied by $(-1)^{\deg q_i}$, where $\deg q_i$ is the degree of q_i.

[5]Our polynomials do not have any fixed representation, and equality checking is a potentially costly operation.

know that $y = 0$, we also know that the leading coefficient of q_1 is 1 and the leading coefficient of q_2 is 1, which means that $q_1 = 1$ and $q_2 = 1$, and so $p = (1 \cdot 1) \cdot (\frac{\partial}{\partial x}(1 \cdot 1)) = 0$.

Next, for each branch, `sign_determination` performs a calculation (formalized in our `limit_points_on_branch` function) to find the signs of `qs` at ∞ and $-\infty$. These are stored in `pos_limit_branch` and `neg_limit_branch`, respectively.

Then, it makes a call to our `calculate_data_assumps_M` function (discussed in Sect. 4.1.2) to calculate a list of matrix equations for each branch, each of which stores sign information under some assumptions (assumptions in our formalization only accumulate, so the output assumptions contain the original branch's assumptions). It pulls out the assumptions and sign conditions from the matrix equations with the `extract_signs` function, which returns a list of type `(assumps × rat list list) list`. This list is stored in `mat_eq_signs_on_branch`.

Finally, the positive and negative limit sign conditions (respectively, `pos_limit_branch` and `neg_limit_branch`) are prepended to each list of sign conditions calculated with the matrix equations (the # operator in Isabelle/HOL prepends an element to a list), and the resulting list of assumptions and associated sign conditions is returned.

It is now time to discuss the matrix equation.

4.1.2 The Multivariate Matrix Equation

The multivariate matrix equation, like the univariate matrix equation, is concerned with finding sign information for a set of polynomials $q_1, \ldots, q_n$ at the roots of an auxiliary polynomial p. One advantage of formalizing a multivariate QE algorithm based on BKR and Tarski is that the construction of the multivariate matrix equation is very similar to the construction of the univariate matrix equation.

Thus, to understand the multivariate matrix equation, we first need to consider the construction of the univariate matrix equation. At its core, the univariate matrix equation relies on computing *Tarski queries*, so we start there.

Computing Multivariate Tarski Queries

Recall (see Definition 2) that for univariate polynomials p, q with $p \neq 0$, the *Tarski query* $N(p, q)$ is defined as:

$$N(p,q) = \#\{x \in \mathbb{R} \mid p(x) = 0, q(x) > 0\} - \#\{x \in \mathbb{R} \mid p(x) = 0, q(x) < 0\}.$$

These Tarski queries can be computed from the Euclidean remainder sequence that starts with p and $p'q$:

Proposition 1. *(Sturm-Tarski Theorem) Let $p \neq 0$ and q be real* univariate *polynomials. Let $p_1 = p$, $p_2 = p'q$, $p_3, \ldots, p_k$ be the Euclidean remainder sequence of p and $p'q$, where*

$$p_i = c_i p_{i+1} - p_{i+2},$$

Further, even if two polynomials are not identically equivalent, they may be so under a branch's assumptions (for example, $y^2 + y + 1$ is equivalent to y^2 if $y + 1$ is assumed to be 0).

for $c_i \in \mathbb{R}[x]$ *and where* $\deg(p_{i+2}) < \deg(p_{i+1})$*. Let* a_i *be the leading coefficient of* p_i *and let* $d_i := \deg(p_i)$*. Let* $S^+(p,q)$ *denote the number of sign changes in the sequence* $a_1, \ldots, a_k$*, and let* $S^-(p,q)$ *denote the number of sign changes in the sequence* $(-1)^{d_1}a_1 \ldots, (-1)^{d_k}a_k$*. Then* $N(p,q) = S^-(p,q) - S^+(p,q)$*.*

To revisit an example from the last chapter of this thesis (see Sect. 3.1.3), given $p = x^3 - x$ and $q = 3x^3 + 2$, then to calculate $N(p,q) = 1$, we would first compute $p_1 = x^3 - x$ and $p_2 = (3x^2 - 1)(3x^3 + 2) = 9x^5 - 3x^3 + 6x^2 - 2$. Then, $p_3 = -x^3 + x$, since $p_1 = 0 \cdot p_2 - (-p_1)$. Next, $p_4 = -6x^2 - 6x + 2$, since $p_2 = (-9x^2 - 6)p_3 - (-6x^2 - 6x + 2)$. Continuing in this manner, we find $p_5 = \frac{1}{3}x - \frac{1}{3}$ and $p_6 = 10$. Now, the sequence of a_i's is $1, 9, -1, -6, \frac{1}{3}, 10$. Thus $S^+(p,q) = 2$. The sequence of $(-1)^{d_i} \cdot a_i$'s is $-1, -9, 1, -6, -\frac{1}{3}, 10$. Thus $S^-(p,q) = 3$, and so $N(p,q) = 3 - 2 = 1$.

Proposition 1 is from the literature [75, Prop. 8.1] (with an unnecessary assumption removed that is not included in other references [1] or in Isabelle's existing formalization [51] of the Sturm-Tarski theorem). Critically, in the Sturm-Tarski theorem, it is not the values of $a_1, \ldots, a_k$ that matter; rather, it is the signs that matter; this is what enables the multivariate generalization [2].

Consider polynomials $p \neq 0$ and q in x with polynomial coefficients in $y_1, \ldots, y_n$ (i.e., $p, q \in \mathbb{R}[y_1, \ldots, y_n][x]$). Then, we can form Euclidean remainder sequences of p and $p'q$ with respect to x. The Euclidean remainder sequence is no longer unique—instead, there are multiple sequences, each depending on the signs of the coefficients of p and q (as coefficients that are polynomials can have different signs at different points). Once we fix a sequence and find the leading coefficients, we need to consider (by branching) *all* possible sign assignments to those coefficients,[6] and output a list of Tarski queries and the assumptions they are subject to.

For example, if we take polynomials $p = y^2x + 1$ and $q = yx + 1$, then if $y^2 = 0$, then $y = 0$ so $p = q = 1$, and the Euclidean remainder sequence is just 1, and $N(p,q) = 0$.[7] However, if $y \neq 0$, then our Euclidean remainder sequence is $y^2x + 1, y^3x + y^2, -(1-y)$, where we have calculated $y^2x + 1 = \frac{1}{y} \cdot (y^3x + y^2) + (1 - y)$, using assumption $y \neq 0$ for $\frac{1}{y}$.

Now, continuing the computation of $N(y^2x + 1, yx + 1)$, we find that the leading coefficients of our Euclidean remainder sequence (assuming $y \neq 0$) are y^2, y^3, and $-(1-y)$. Next, we consider the possible sign assignments to y^2, y^3, and $-(1-y)$. For example, $(+,+,-)$ is one such sign assignment. So, we have Tarski query $N(p,q) = S^-(p,q) - S^+(p,q) = 0 - 1 = -1$ under the assumptions that: $y \neq 0$, $y^2 > 0$, $y^3 > 0$, and $-(1-y) < 0$. Our output for $N(y^2x + 1, yx + 1)$ would be a list of all the Tarski queries under all possible assumptions. This computation is visualized in Fig. 4.3 (where, for purposes of space, only three output branches are shown explicitly).

Note that Euclidean remainder sequences for multivariate polynomials sometimes contain fractions. While we could have chosen to work with Euclidean remainder sequences in a *fraction field*, this would require complicated type switching in the formalization. Instead, we use *pseudo-remainder sequences* for multivariate polynomials. Pseudo-remainder sequences are es-

[6]Full BKR would consider all consistent sign assignments instead. This makes the algorithm highly recursive, which adds a considerable layer of difficulty to its verification.

[7]Technically, our formalization would do more branching than this for two reasons: First, it will branch on $y^2 = 0$, $y^2 > 0$, and (unnecessarily) $y^2 < 0$; and second, because it will not determine that $y^2 = 0$ implies $y = 0$—and so it will not know that $q = 1$ whenever $y^2 = 0$.

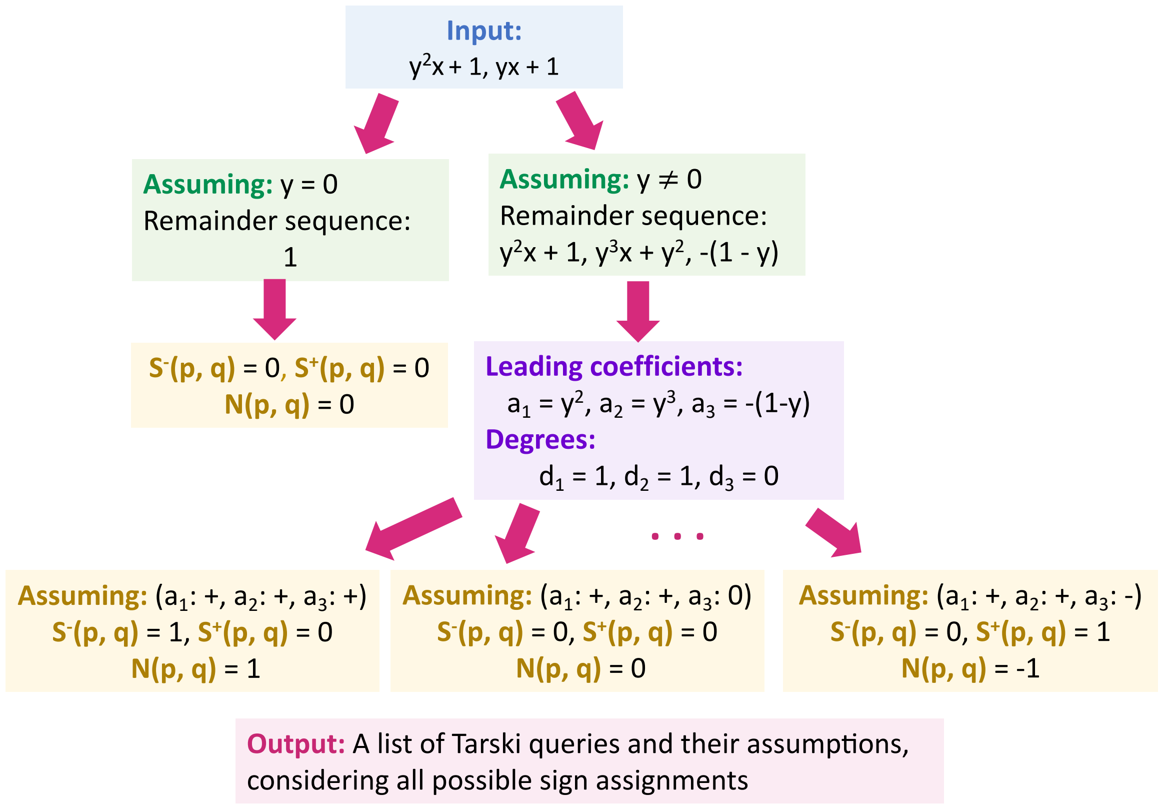

Input:
$y^2x + 1, yx + 1$
Assuming: $y = 0$
Remainder sequence:
1
Assuming: $y \neq 0$
Remainder sequence:
$y^2x + 1, y^3x + y^2, -(1 - y)$
$S^-(p, q) = 0, S^+(p, q) = 0$
$N(p, q) = 0$
Leading coefficients:
$a_1 = y^2, a_2 = y^3, a_3 = -(1-y)$
Degrees:
$d_1 = 1, d_2 = 1, d_3 = 0$
. . .
Assuming: (a_1: +, a_2: +, a_3: +)
$S^-(p, q) = 1, S^+(p, q) = 0$
$N(p, q) = 1$
Assuming: (a_1: +, a_2: +, a_3: 0)
$S^-(p, q) = 0, S^+(p, q) = 0$
$N(p, q) = 0$
Assuming: (a_1: +, a_2: +, a_3: -)
$S^-(p, q) = 0, S^+(p, q) = 1$
$N(p, q) = -1$
Output: A list of Tarski queries and their assumptions, considering all possible sign assignments

Figure 4.3: Computing Tarski queries for $p = y^2x + 1$, $q = yx + 1$.

sentially Euclidean remainder sequences for polynomials, but normalized so as not to contain fractions (ours are additionally normalized so as not to affect the result of the Sturm-Tarski computation [54]). We develop pseudo-remainder sequences for multivariate polynomials of type *rmpoly* (currently, our formalization naively branches on the signs of the leading coefficients of the relevant polynomials). Here, we benefit from prior work: The Sturm-Tarski theorem was formalized in Isabelle/HOL by Wenda Li [51]; Li and Paulson later extended this to *bivariate* polynomials [52] using pseudo-remainder sequences, and Li, Passmore, and Paulson also developed univariate Tarski queries with pseudo-remainder sequences [54].

Remark 6. *For self-containedness, we briefly describe pseudo-remainder sequences.* Polynomial pseudo-quotients *(pquo) and* pseudo-remainders *(prem) satisfy this property [26, 54]:*

$$(lead\_coeff\ q)^{(1+deg\ p-deg\ q)}p = pquo(p,q)\cdot q + prem(p,q),$$

where deg $prem(p,q) < deg\ q$ *or* $q = 0$*. For example, when considering polynomials* $p = yx^2+1$ *and* $q = y^3x+1$ *as univariate polynomials in* x*, then* $pquo(p,q) = y^4x - y$ *and* $prem(p,q) = y^6 + y$*, as* $(y^3)^2p = (y^4x-y)q + (y^6+y)$ *and* $deg(y^6+y) = 0 < deg\ q = 1$*. Notice how there are no fractions in pquo or prem, unlike the fractions in the usual Euclidean remainder sequence (assuming* $y \neq 0$ *for well-definedness).*

We use signed *pseudo-remainder sequences, where* $p_1 = p$*,* $p_2 = p'q$*, and* $p_3, \ldots, p_k$ *satisfy the following equation for a special choice of coefficients* s_i*, explained below:*

$$p_{i+2} = s_i \cdot prem(p_i, p_{i+1})$$

This sequence is normalized so that, in any valuation, the number of sign changes in the evaluated pseudo-remainder sequence is the same as in the Euclidean remainder sequence for the evaluated polynomials, so that the result of the Sturm-Tarski computation is unaffected by the normalization. For this, we follow the style of [54] and normalize as follows: if $(1 + \textit{deg}\, p_i - \textit{deg}\, p_{i+1})$ *is even, we multiply prem*(p_i, p_{i+1}) *by* $s_i = -1$*; else, by* $s_i = -\textit{lead\_coeff}\, p_{i+1}$*. To understand this intuitively, note that the pseudo-remainder prem*(p, q) *effectively normalizes by* $(\textit{lead\_coeff}\, q)^{(1+\textit{deg}\, p - \textit{deg}\, q)}$*. Then, note that remainder sequences in the Sturm-Tarski theorem always negate prem (cross-reference Proposition 1). So, if* $(1 + \textit{deg}\, p - \textit{deg}\, q)$ *is even, we have not changed the sign of prem and we need only negate it. However, if* $(1 + \textit{deg}\, p - \textit{deg}\, q)$ *is odd, we have potentially changed the sign of prem—depending on the sign of* $(\textit{lead\_coeff}\, q)$*—so we not only negate prem but also multiply it by* $(\textit{lead\_coeff}\, q)$*.*

Since QE is concerned with sign information for multiple polynomials simultaneously, it is useful to generalize the notion of Tarski queries to *sets* of polynomials (compare Definition 3) as follows:

Definition 4. Given a polynomial p and a list of polynomials $q_1, \ldots, q_n$, let I and J be subsets of $\{1, \ldots, n\}$. Then, the Tarski query $N(I, J)$ with respect to p is

$$N(I, J) = N(p^2 + \left(\Sigma_{i \in I} q_i^2\right), \Pi_{j \in J}\, q_j).$$

For univariate p and $q_1, \ldots, q_n$, if I and J are subsets of $\{1, \ldots, n\}$, then

$$\begin{aligned} N(I, J) &= N(p^2 + \left(\Sigma_{i \in I} q_i^2\right), \Pi_{j \in J}\, q_j) = \\ &\#\{x \in \mathbb{R} \mid p(x) = 0, \forall i \in I.\ q_i(x) = 0, \Pi_{j \in J}\, q_j(x) > 0\} - \\ &\#\{x \in \mathbb{R} \mid p(x) = 0, \forall i \in I.\ q_i(x) = 0, \Pi_{j \in J}\, q_j(x) < 0\}. \end{aligned}$$

For example, if $p = x^3 - x$, $q_1 = 3x^3 + 2$, $q_2 = x$, and $q_3 = x^2$, then

$$\begin{aligned} N(\{2, 3\}, \{1\}) &= N((x^3 - x)^2 + x^2 + x^4, 3x^3 + 2) \\ &= \#\{x \in \mathbb{R} \mid x^3 - x = 0, x = 0, x^2 = 0, 3x^3 + 2 > 0\} - \\ &\quad \#\{x \in \mathbb{R} \mid x^3 - x = 0, x = 0, x^2 = 0, 3x^3 + 2 < 0\} \\ &= 1 - 0 = 1. \end{aligned}$$

The matrix equation determines the signs of $q_1, \ldots, q_n$ at the zeros of p by computing $N(I, J)$ for a representative set of combinations of subsets I, J of $q_1, \ldots, q_n$ (see Sect. 4.1.2).

There are two key lemmas that we prove about multivariate Tarski queries. The first is a soundness lemma showing that the resulting multivariate Tarski queries agree, on every point satisfying the associated assumptions, with what the univariate Tarski query would have been:

```
lemma multiv_tarski_query_correct:
  assumes inset: "(assumps, tarski_query) ∈
    set(construct_NofI_M p acc I J)"
  assumes val: "⋀f n. (f,n) ∈ set assumps ⟹
    satisfies_evaluation val f n"
  shows "tarski_query = construct_NofI_R (eval_mpoly_poly val p)
    (eval_mpoly_poly_list val I) (eval_mpoly_poly_list val J)"
```

Here, the *construct_NofI_M* function constructs a list of multivariate Tarski queries and the assumptions they are subject to. As input, it takes a polynomial *p*, an initial set of assumptions *acc*, and two lists of polynomials *I* and *J*. Both *p* and all of the polynomials in *I* and *J* have type *rmpoly*, i.e. they are univariate polynomials in x with polynomial coefficients in some variables $y_1, \ldots, y_n$. The *inset* assumption assumes that we have some particular Tarski query *tarski_query* that is subject to the assumptions *assumps*, which are assumptions on polynomials in $y_1, \ldots, y_n$. Now, the *construct_NofI_R* function is the function to compute univariate Tarski queries from our prior work [19], so the conclusion of the lemma is that *tarski_query* is exactly the (unique) univariate Tarski query that would be computed from evaluating *p* and all of the polynomials in *I*, *J* on *val* (using the *eval_mpoly_poly* and *eval_mpoly_poly_list* functions), where *val* is any assignment of real values to $y_1, \ldots y_n$ where the assumptions *assumps* are realized.

The second key lemma is a completeness result:

```
lemma multiv_tarski_queries_complete:
  assumes "⋀f n. (f,n) ∈ set init_assumps ⟹
    satisfies_evaluation val f n"
  shows "∃(assumps, tq) ∈ set (construct_NofI_M p init_assumps I J).
    (∀(p,n)∈set assumps. satisfies_evaluation val p n)"
```

Here, this shows that if initial assumptions *init_assumps* are satisfied by valuation *val*, then there is some resulting assumptions and Tarski query pair *(assumps, tq)* where all final assumptions *assumps* are satisfied by *val*.

Together, these two lemmas give a strong result: the soundness lemma shows that the multivariate results coincide with univariate results in all projections meeting the final assumptions, and the completeness lemma shows that for any projection meeting the initial assumptions, there is some corresponding Tarski query whose associated (final) assumptions are met by the projection. Or, on a more intuitive level, the completeness lemma shows that our function to compute multivariate Tarski queries generates useful output whenever it is given useful input, and the soundness lemma shows that useful output has the desired mathematical meaning.

Using Multivariate Tarski Queries

The matrix equation connects a vector of information about *possible sign assignments* for a set of multivariate polynomials—i.e., sign assignments that are not necessarily consistent—on the LHS, to a vector of multivariate Tarski queries on the RHS.

The univariate matrix equation is defined as follows, where we closely follow the definition of the univariate matrix equation given in the previous chapter, but adapted to our purposes:[8]

[8]Both the previous chapter of this thesis and the corresponding univariate BKR paper [19] follow the matrix equation developed in Ben-Or, Kozen, and Reif's original paper [2], where p is assumed to be coprime with each q_i, so that sign assignments where any q_i is 0 at the roots of p can be automatically excluded from the matrix equation. Because a global coprimality assumption does not ensure coprimality in each valuation for multivariate polynomials, in this chapter we use the matrix equation developed by Renegar [75], which encodes information for sign assignments where the q_i's are 0. While our univariate work formalized both styles of matrix equation [20], only the former is discussed at length in the paper [19] (and in Chapter 3). As mentioned previously, the two univariate formalizations are in fact very similar, and much of the code is shared.

Definition 5. Fix univariate polynomials of interest p and $q_1, \ldots, q_k$. Let $\tilde{\Sigma} = \{\tilde{\sigma}_1, \ldots, \tilde{\sigma}_m\}$ be a set of possible sign assignments to $q_1, \ldots, q_k$, and assume $\tilde{\Sigma}$ contains all consistent sign assignments to $q_1, \ldots, q_k$ at the roots of p. Let S be a set of pairs of subsets $(I_1, J_1), \ldots, (I_l, J_l)$ where for all $1 \leq i \leq l$, $I_i \subseteq \{1, \ldots, k\}$ and $J_i \subseteq \{1, \ldots, k\}$. Then the *matrix equation* for $\tilde{\Sigma}$ and S is the relationship $M \cdot w = v$ between the following three entities:

- M, the l-by-m matrix with entries

$$M_{i,j} = \left(\Pi_{\ell \in I_i}(1 - (\tilde{\sigma}_j(q_\ell))^2)\right) \cdot (\Pi_{\ell \in J_i} \tilde{\sigma}_j(q_\ell)) \in \{-1, 0, 1\}$$

  for $(I_i, J_i) \in S$ and $\tilde{\sigma}_j \in \tilde{\Sigma}$,
- w, the length m vector whose entries count the number of roots of p where $q_1, \ldots, q_k$ has sign assignment $\tilde{\sigma}$, i.e., $w_i = \#\{x \in \mathbb{R} \mid p(x) = 0, \text{sgn}(q_\ell(x)) = \tilde{\sigma}_i(q_\ell) \text{ for all } 1 \leq \ell \leq k\}$,
- v, the length l vector consisting of Tarski queries for the subsets, i.e., $v_i = N(I_i, J_i)$.

Intuitively, as noted in the previous chapter, the meaning of a matrix equation is captured by its associated list of signs and list of (pairs of) subsets. Both the matrix M and the RHS vector v are fully computable from these two lists, and w, which stores information about which possible sign assignments are consistent (sign assignment $\tilde{\sigma}_i$ is consistent iff w_i is nonzero), is calculated as $M^{-1} \cdot v$.

For multivariate polynomials the situation is more complicated. We can still construct a matrix equation for multivariate polynomials—the definition of the matrix M is the same as it was in the univariate setting, but the righthandside vector uses our function to construct a list of Tarski queries for multivariate polynomials. Each RHS vector—and so each matrix equation—comes with an associated list of assumptions which were generated by the multivariate Tarski queries. So, for an input list of multivariate polynomials p and $q_1, \ldots, q_k$, we construct a *list* of multivariate matrix equations that store sign information for these polynomials, subject to certain assumptions on polynomials in one fewer variable.

The overall construction is very similar to that in the univariate case, which was discussed in the previous chapter. It proceeds by induction on the number of q's, so that the base case is for a single q. Smaller matrix equations are successively combined and reduced to form the matrix equation for $q_1, \ldots, q_n$. The reduction is what differentiates the matrix equation of BKR from that of Tarski: information for inconsistent sign assignments is removed at appropriate intervals, which decreases the size of the matrix equation. In the univariate case, the size of the matrix equation is bounded by $\#\{x.\ p(x) = 0\})^2$, where $\#\{x.\ p(x) = 0\}$ is the number of roots of the polynomial p. The size of a multivariate matrix equation is bounded by the number of roots of p in a valuation satisfying the associated assumptions. As the univariate reduction step mainly involves computations on the matrix M, which is unchanged in the multivariate setting, it generalizes quite naturally, and so our hybrid algorithm essentially inherits reduction in the matrix equation construction, thus incorporating insights from BKR into our hybrid algorithm.

We formalize our multivariate matrix equation construction in `calculate_data_assumps_M` (cross reference Sect. 4.1.1), and prove the following two key lemmas:

```
lemma multivariate_calculate_data_correct:
  assumes mat_eq: "(assumps, mat_eq) ∈
    set (calculate_data_assumps_M p qs init_assumps)"
  assumes "⋀p n. (p,n) ∈ set assumps ⟹
```

```
    satisfies_evaluation val p n"
  assumes "eval_p = eval_mpoly_poly val p"
  assumes "eval_qs = map (eval_mpoly_poly val) qs"
  assumes p_nonzero: "eval_mpoly_poly val p ≠ 0"
  shows "calculate_data_R eval_p eval_qs = mat_eq"
```

This first lemma connects the behavior of our multivariate matrix equation constructor function to the Renegar-style univariate matrix equation function (*calculate_data_R*) formalized in our prior work [20]. That is, on any valuation *val* that satisfies the assumptions *assumps*, the associated multivariate matrix equation *mat_eq*, which finds the consistent sign assignments for *qs* at the zeros of some *p* in the valuation *val*, is equal to the univariate matrix equation that find the consistent sign assignments for *eval_qs* at the zeros of *eval_p* , where *eval_p* is *p* evaluated on *val* and *eval_qs* is *qs* evaluated on *val*. This is a soundness lemma, since it explains that whenever our output is useful, it has the correct mathematical meaning.

```
lemma multivariate_calculate_data_complete:
  assumes "⋀p n. (p,n) ∈ set init_assumps ⟹
    satisfies_evaluation val p n"
  shows "∃ (assumps, mat_eq) ∈
    set (calculate_data_assumps_M p qs init_assumps).
    (∀ (p,n) ∈ set assumps. satisfies_evaluation val p n)"
```

This second lemma shows that when *calculate_data_assumps_M* has logically consistent input assumptions, some output with logically consistent assumptions will be generated (i.e., useful input generates useful output). These lemmas are analogous to those discussed for multivariate Tarski queries; taken together, they help us prove key correctness properties of our *elim_forall* method, which serves to eliminate a single universal quantifier. We now turn to a discussion of our top-level QE methods, including *elim_forall*.

4.1.3 Overall Quantifier Elimination Algorithm

To best explain our formalized QE algorithm, we must first touch on the framework we are working with.

We build on the framework discussed in Chapter 2, which detailed the verification of the virtual substitution (VS) algorithm. The VS formalization set up a framework for multivariate QE (including a type for real QE problems and a function to evaluate QE problems at real-valued points); by building on this, we are ultimately able to link together our verified (complete, inefficient) QE method with verified virtual substitution, using this (incomplete but experimentally promising) QE method as a preprocessing step.

Accordingly, we work with formulas of type *atom fm* [76], which have the following grammar (cross reference Sect. 2.3.1):

$$\begin{aligned} F, G \ ::= \ & \text{TrueF} \mid \text{FalseF} \mid (\text{Atom}(\text{Eq}\, p)) \mid (\text{Atom}(\text{Less}\, p)) \mid (\text{Atom}(\text{Leq}\, p)) \mid (\text{Atom}(\text{Neq}\, p)) \mid \\ & \text{And}\ F\ G \mid \text{Or}\ F\ G \mid \text{Neg}\ F \mid \text{ExQ}\ F \mid \text{AllQ}\ F \mid \text{ExN}\ n\ F \mid \text{AllN}\ n\ F, \end{aligned}$$

where p is a real polynomial and $n \in \mathbb{N}$. Here, $(\text{Atom}(\text{Eq}\ p))$ captures the relationship $p = 0$, $(\text{Atom}(\text{Less}\ p))$ captures $p < 0$, $(\text{Atom}(\text{Leq}\ p))$ captures $p \leq 0$, and $(\text{Atom}(\text{Neq}\ p))$ captures

$p \neq 0$. Further, And F G captures the logical meaning of $F \wedge G$, Or F G captures $F \vee G$, and Neg F captures $\neg F$. Finally, ExQ F indicates that formula F is quantified by an existential quantifier, AllQ F indicates that F is quantified by a universal quantifier, ExN n F indicates that F is quantified by a block of n existential quantifiers, and AllN n F indicates that F is quantified by a block of n universal quantifiers.

In these formulas, variables are represented with de Bruijn indices; *Var 0* is the variable quantified by the innermost quantifier, *Var 1* is the variable quantified by the second innermost quantifier, and so on. We operate on quantifiers inside-out, i.e. we start with the quantifier attached to *Var 0*.

Our *elim_forall* function is designed to eliminate a single $\forall$ quantifier. It parallels the method visualized in Fig. 4.1.

```
fun elim_forall:: "atom fm ⇒ atom fm"
  where "elim_forall F = (let qs = extract_polys F;
    univ_qs = univariate_in qs 0;
    reindexed_univ_qs = map (map_poly (lowerPoly 0 1)) univ_qs;
    initial_data = sign_determination reindexed_univ_qs [];
    filtered_data = filter (λ(assumps, signs_list).
    list_all (λ signs. lookup_sem_M F (zip qs signs) = (Some True))
    signs_list
    ) initial_data
  in create_disjunction filtered_data)"
```

Here, *extract_polys* finds the polynomials *qs* in our formula *F*, and *univariate_in qs 0* transforms our polynomials *qs* to have the *rmpoly* type (so that they are univariate polynomials in *Var 0*, with coefficients that are multivariate polynomials in bigger variables). The resulting list of polynomials is called *univ_qs*. Then, in *reindexed_univ_qs*, we transform the coefficients of every polynomial in *univ_qs* (which do not contain *Var 0*) by lowering every variable index by 1. This lowering is crucial for finding all possible signs/assumptions pairs for our multivariate polynomial coefficients (cross reference Sect. 4.1.1), as *sign_determination* expects polynomials in *Var 0*. We then retain all the sign assignments that satisfy our formula of interest, and return a disjunction of the associated assumptions. If our original formula involved polynomials in variables *Var 0*, *Var 1*, ..., *Var n*, then, because of the transformation and reindexing, these assumptions will be polynomials in variables *Var 0*, ..., *Var (n - 1)*. Our new *Var 0*, which was previously *Var 1*, will correctly match to the new innermost quantifier, which was previously the second innermost quantifier, and so on.

Our top-level QE method, named *qe*, heavily relies on *elim_forall* and *elim_exist* (where *elim_exist F* is defined as *Neg (elim_forall (Neg F))*):

```
fun qe:: "atom fm ⇒ atom fm"
  where
    "qe TrueF = TrueF"
  | "qe FalseF = FalseF"
  | "qe (Atom a) = (Atom a)"
  | "qe (And F1 F2) = And (qe F1) (qe F2)"
  | "qe (Or F1 F2) = Or (qe F1) (qe F2)"
  | "qe (Neg F) = Neg (qe F)"
  | "qe (ExQ F) = elim_exist (qe F)"
  | "qe (AllQ F) = elim_forall (qe F)"
  | "qe (AllN n F) = (elim_forall ^^ n) (qe F)"
  | "qe (ExN n F) = (elim_exist ^^ n) (qe F)"
```

Our top-level correctness theorem says that for any assignment ν of the free variables in `F` to real numbers, our original formula `F` has the same truth-value as `qe F`; or, in other words, `F` and `qe F` are logically equivalent:

theorem `qe_correct:`
 fixes `F:: "atom fm"`
 shows `"eval F ν = eval (qe F) ν"`

Here, `eval` is the function formalized in the VS development (see Chapter 2) to evaluate formulas of type `atom fm` on valuations. This function accounts for the reindexing of free variables that naturally takes place during QE. For example, $\forall x.\ x^2y \leq 0$ is logically equivalent to $y \leq 0$, but since variables are represented with de Bruijn indices, where the innermost quantifier corresponds with `Var 0`, $\forall x.\ x^2y \leq 0$ is represented in the `atom fm` type as `AllQ (Leq ((Var 0)^2 · Var 1))` whereas $y \leq 0$ is represented as `Leq (Var 0)`. In `eval`, this subtlety is handled by defining, e.g., `eval (AllQ F) v` as `(∀ x. (eval F (x#v)))`, where `x#v` is the list with head `x` and tail `v`. So, `qe_correct` shows that `F` evaluated on any mapping of free variables to real numbers is equal to `qe F` evaluated on that same mapping, which establishes that `qe` is sound.

We also show that `qe` fully removes quantifiers in the following lemma, where the function `countQuantifiers` counts the number of existential or universal quantifiers in a formula:

theorem `qe_complete:`
 shows `"countQuantifiers (qe F) = 0"`

This result shows that `qe` is complete.

To our knowledge, `qe` is the first sound and complete algorithm for real QE to be formalized in Isabelle/HOL (previous work [54, 64, 76] was sound but not complete). We now turn to some further details regarding our formalization.

4.2 Formalization Details

As in our formalization of univariate BKR (see Chapter 3), Isabelle/HOL is well-suited for us; we not only benefit considerably from the well-developed libraries (including our previous formalizations [19, 76] and other prior work [54]), but also from Isabelle/HOL's support for automated proof search in Sledgehammer [68].

However, at the same time, working in the formal setting of Isabelle/HOL poses considerable challenges. In this section, we begin by discussing some of those challenges, followed by some of the high-level proof techniques that helped us succeed in our formalization. We then discuss some useful low-level details regarding our extensions to Isabelle/HOL's multivariate polynomials library. Finally, we discuss our code export and the performance of our algorithm.

4.2.1 Challenges

Many design decisions for the functions described in Sect. 4.1 were not initially evident. For example, the need to consistently track assumptions and pass them in as an argument to our functions throughout the calculation of the matrix equation was initially not obvious. At first, we wrote a function that was nearly identical to `calculate_data_assumps_M`, with the one major difference that we did not include `assumps` as an argument to this function. While this function was fully capable of generating a multivariate matrix equation, we soon realized we had made a major mistake when we tried to extend it into a larger QE algorithm. After this, we were careful to always include an argument for assumptions in our functions if it could possibly be applicable, regardless of whether or not it seemed immediately relevant.

The challenge of correctly formalizing the algorithm in Isabelle/HOL is heightened because the precision of formalization sometimes identifies details that were underspecified in the source material. Indeed, BKR's discussion of the multivariate QE algorithm was limited to only two pages and proceeds at a very high level [2]. Renegar [75] is considerably more detailed, but is also written in the style of mathematics, which necessitates significant translation to the level of formalization. For example, the way in which the limit point calculation should be formalized, while entirely obvious in retrospect, did not become clear to us until we fixed a method of branching—and indeed, our initial method of formalizing the limit point calculation, which was agnostic to branching, did not make it into the final code for the algorithm. Of this calculation, Renegar writes the following, in which he uses the notation g_i where we use q_i, and f instead of p [75]: ". . . each consistent sign vector of $\{g_i\}_i$ occurs at some real zero of f except, perhaps, for the sign vectors of points to the right or left of all real zeros of $\prod_i g_i$. However, the latter two consistent sign vectors are trivially determined from the leading coefficients of the polynomials g_i." While this completely describes the mathematical use of the limit point calculations, it took some time to translate it into Isabelle/HOL definitions and proofs.

Finally, a last challenge is that even simple details can become complex in the formalized setting of a theorem prover. For example, working with multivariate polynomials in Isabelle/HOL poses a challenge, as the formal setting requires rigor even for operations that are simple on paper but may become much more involved when formalized. For example, the transformation to treat a multivariate polynomial as univariate in some variable of interest is immediate on paper, but in Isabelle/HOL it is more subtle, precisely because the type of our object is changing: $3xyz^2 + 6x^2wv + 5xy + 1$ has type `real mpoly`, whereas $(6wv)x^2 + (3yz^2 + 5y)x + 1$ has type `rmpoly` (see also Sect. 4.1.1).

4.2.2 High Level Proof Techniques

Though treating multivariate polynomials as univariate in some variable of interest poses low-level challenges in our formal setting, it affords significant high-level simplifications. Many of our proofs rely on the technique of universal *projection*—we assume fixed real values for all variables aside from a variable of interest, which lets us work with *truly* univariate polynomials. Projection allows us to connect functions in our multivariate construction to corresponding functions in the univariate construction. This works because the multivariate case of the BKR algorithm builds rather directly on the univariate case, making it amenable to formalization, as noted previously.

In consequence, each key function involved in the construction of the multivariate matrix equation requires two top-level associated lemmas. The first is a soundness lemma which connects the behavior of the multivariate function to a corresponding univariate function [19] through projection. The second is a completeness lemma which establishes that data for all possible projections is captured by the function for some assumptions. Some examples of these soundness and completeness lemmas are seen in Sect. 4.1.2 (e.g. *multiv_tarski_query_correct* and *multiv_tarski_query_complete*); there are many more in the actual proof development. This proof structure does not seek to closely mimic the (highly mathematical) proofs in the source material [2, 75], but rather to translate the key intuition into a shape which is amenable to formalization.

Our construction and proofs are designed to be modular, and we often rely on induction to prove key properties of helper functions. In particular, we found it very helpful to use custom induction theorems, supplementing those automatically generated by Isabelle/HOL. For example, the *spmods_multiv_aux* function shown (abridged) below computes a list of pseudo-remainder sequences for polynomials *p* and *q* together with corresponding sign assumptions on the leading coefficients of the polynomials in each sequence.

```
function spmods_multiv_aux:: "rmpoly ⇒ rmpoly ⇒ assumps ⇒
  (assumps × rmpoly list) list" where
  "spmods_multiv_aux p q assumps = (if q = 0 then [(assumps, [p])]
  else case (lookup_assump_aux (lead_coeff q) assumps) of
  None ⇒
    let lcz = spmods_multiv_aux p (one_less_degree q)
              ((lead_coeff q, 0) # assumps) in
    let lcp = spmods_multiv_aux q (mul_pseudo_mod p q)
              ((lead_coeff q, 1) # assumps) in
    let lcn = spmods_multiv_aux q (mul_pseudo_mod p q)
              ((lead_coeff q, -1) # assumps) in
      ... /* combine lcz, lcp, lcn */
  | (Some i) ⇒  ... /* two recursive branches */ )"
```

The function branches depending on whether *q* is the zero polynomial, otherwise, it recurses on the (possible) signs of its leading coefficient *lead_coeff q*. Here, *assumps* specifies a list of assumed input sign conditions, which are checked for assumptions on *lead_coeff q*. Notably, *spmods_multiv_aux* is ***not*** structurally recursive; its termination uses the fact that, on each recursive call, the degree of the polynomial arguments *one_less_degree q* or *mul_pseudo_mod p q* strictly decreases. For such functions, Isabelle/HOL automatically generates induction theorems, but these theorems lack the usual case-splitting support for structurally recursive functions [94]. The following snippet shows the Isabelle/HOL subgoal (`cases`) structure that results from applying induction with the generated theorem for *spmods_multiv_aux*.

```
// apply (induct ... spmods_multiv_aux.induct)
Proof outline with cases:
  case (1 p q assumps)
  ...
qed
```

Although *spmods_multiv_aux.induct* can, *in principle*, be used to prove the aforementioned soundness and completeness properties for `spmods_multiv_aux`, we found the proofs tedious in practice because they lack the case structuring benefits of Isabelle/HOL's structured proof language [94]. Instead, we manually prove an alternative induction theorem that mimics the branching structure of `spmods_multiv_aux` (one base case, three branches with recursion). As before, a snippet of the Isabelle/HOL subgoal (cases) structure is shown below (comments illustrate the branching structure).

```
// apply (induct ... spmods_multiv_aux_induct)
Proof outline with cases:
  case (Base p q assumps)
  ... // base case (q = 0)
next
  case (Rec p q assumps)
  ... // lookup_assump_aux returns None
next
  case (Lookup0 p q assumps)
  ... // lookup_assump_aux returns Some 0
next
  case (LookupN0 p q assumps r)
  ... // otherwise
qed
```

Though some manual effort is needed to state and prove `spmods_multiv_aux_induct`, our subsequent, repeated use of this customized induction theorem makes it well worth the initial investment. We expect similar induction theorems to be broadly useful for structuring proofs about non-structural recursive functions, including in other proof assistants. Indeed, manual induction theorems are also used elsewhere in the development, particularly to verify invariant properties of the helper function that underlies the branching function `lc_assump_generation_list` (see Sect. 4.1.1).

4.2.3 Library Extensions

We turn to some of our key results for multivariate polynomials and the library extensions they prompted.

As seen in Sect. 4.2.1, we need a function to convert polynomials of type `real mpoly` to polynomials of type `real mpoly poly`. Eberl and Thiemann formalized one such way of doing this in their `mpoly_to_mpoly_poly` definition [35]. We provide the following alternate definition, which is executable:

definition *mpoly_to_mpoly_poly_alt :: "nat ⇒ 'a :: comm_ring_1 mpoly ⇒ 'a mpoly poly"*
 where *"mpoly_to_mpoly_poly_alt x p = (∑i∈{0..MPoly_Type.degree p x} . monom (isolate_variable_sparse p x i) i)"*

This definition applies to multivariate polynomials with coefficients in a commutative ring with unity (denoted by `comm_ring_1`). It relies on the `isolate_variable_sparse` function [78], where `isolate_variable_sparse p x i` finds the coefficient of `x^i` in *p*. For each *i* from 0 to the degree of *x* in *p*, we find this coefficient and construct a monomial of type `poly` with degree *i* and this coefficient. Our final polynomial is the sum of all of these monomials.

We connect our new definition to *mpoly_to_mpoly_poly* in the following lemma:

```
lemma multivar_as_univar:
  shows "mpoly_to_mpoly_poly_alt x p = mpoly_to_mpoly_poly x p"
```

This enables a natural interface between Eberl and Thiemann's work [35] and the large and powerful collection of lemmas regarding *isolate_variable_sparse* [78], from which we benefit in the formalization.

We benefit from Eberl and Thiemann's lemmas regarding *mpoly_to_mpoly_poly* in one of our main results regarding polynomials, which is useful in our correctness proof for *elim_forall* (cross reference Sect. 4.1.3):

```
lemma reindexed_univ_qs_eval:
  assumes "univ_qs = univariate_in qs 0"
  assumes "reindexed_univ_qs = map (map_poly (lowerPoly 0 1)) univ_qs"
  shows "map (eval_mpoly (x#xs)) qs = (map (λp. (poly p x))
    (map (λq. eval_mpoly_poly xs q) reindexed_univ_qs))"
```

This lemma relates the evaluation of multivariate polynomials, of type *real mpoly*, and multivariate polynomials *treated as univariate polynomials* in the variable of interest *Var 0*, of type *rmpoly*. To fully understand it, we must explain a few Isabelle/HOL operators that manipulate multivariate polynomials. Here, *eval_mpoly* is our name for the natural definition of multivariate polynomial evaluation which substitutes real values for variables. Because variables are represented with de Bruijn indices, we can store the values to substitute in a list *L*, where the element of *L* at position 0 is then substituted for *Var 0*, the element of *L* at position 1 is substituted for *Var 1*, and so on. If the length of *L* is shorter than the number of variables, a default value of 0 is substituted for any variables that are not covered by *L*. This definition was implicitly used in the VS formalization [76], but was not explicitly stated and named:

```
definition eval_mpoly:: "real list ⇒ real mpoly ⇒ real"
  where "eval_mpoly L p = insertion (nth_default 0 L) p"
```

The *eval_mpoly_poly* function maps *eval_mpoly* over the coefficients of a *real mpoly poly*.

Continuing to unpack the *reindexed_univ_qs_eval* lemma, the *lowerPoly* function is from the VS formalization (cross reference Appendix A.2.4); here, it serves to reindex variables in multivariate polynomials, so that *lowerPoly 0 1 q* lowers every variable index in *q* by 1. The *univariate_in* operator is our function to perform this multivariate to univariate transformation. Let q_i be the polynomial at the ith index of *qs*, and uq_i be the polynomial at the ith index of *univ_qs*—then the first assumption of *reindexed_univ_qs_eval* says that uq_i is the polynomial that we obtain by treating q_i as univariate in *Var 0*.

Next, the second assumption in *reindexed_univ_qs_eval* says that *reindexed_univ_qs* is the list of polynomials obtained by lowering all variable indices in the *coefficients* of the *univ_qs* by 1. Let us call ruq_i the polynomial at the ith index of *reindexed_univ_qs*. Then, lemma *reindexed_univ_qs_eval* captures the mathematical equivalence of q_i and ruq_i by showing that evaluating q_i on the valuation v = *x#xs* gives the same result as evaluating the *coefficients* of ruq_i on *xs* and then evaluating the resulting univariate polynomial (which now has constant coefficients) on *x*.

The proof of this key lemma required that we first prove the following fundamental extensionality result, which says that if two polynomials *p* and *q* (in n variables) have identical evaluations on $\mathbb{R}^n$, then they are themselves identical:

```
lemma same_evaluations_same_mpoly:
  assumes "(⋀ L. eval_mpoly L p = eval_mpoly L q)"
  shows "p = q"
```

Since real multivariate polynomials are fundamental to many areas of mathematics, it is our hope that our library developments will be useful to others, including in the formalization of other QE algorithms, but also more widely.

4.2.4 Code Export

Our multivariate algorithm is designed so that it is *executable*. Like both its univariate predecessor and the VS algorithm, it can be run directly within Isabelle/HOL, or it can be exported to SML code.[9] Building on the framework of verified virtual substitution (by using the same type for QE formulas and the same evaluation function for formulas) makes the connection with the verified virtual substitution algorithm very easy.[10] This means that we are able to retain efficiency [76] on examples that are tractable for virtual substitution.

However, because virtual substitution is *not* a complete QE method (i.e., it is not able to solve all QE problems), the efficiency, or lack thereof, of our (complete) algorithm is still significant. To get a ballpark sense of what we are able to achieve at present, we ran (within Isabelle/HOL) the multivariate algorithm *without VS preprocessing* on some simple examples. Unfortunately (but not unexpectedly), these simple experiments confirm that, without the link to virtual substitution, our hybrid multivariate algorithm is not at all efficient. We were pleased to see that our algorithm was able to solve some very simple examples, including $\exists x.\ x + 1 < 0$, $\exists x.\ x^2 = 0$, and $\exists x.\ x^4 < 0$. However, it appears to hang on examples that involve more than one variable, or even univariate examples that involve even slightly more complicated polynomials, like $\exists x.\ x^2 - 1 = 0$. While it is possible that experiments with the exported SML code could be slightly more promising (as the code export may remove some overhead), the main efficiency bottlenecks come from within the algorithm itself.

Although inefficiency is not surprising given that even Renegar may not realize practical efficiency in its current state [41, 42], at present, we suspect that part of the efficiency bottleneck for our algorithm is the untenable branching in the computation of the multivariate Tarski queries; this can be significantly reduced in the future by implementing an algorithm that more closely follows BKR. We also believe that our algorithm's lack of inherent optimizations is another contributing factor; as one example, we currently branch unnecessarily on the signs of constant coefficients (this explains why $\exists x.\ x^2 - 1 = 0$ hangs while $\exists x.\ x^2 = 0$ does not). Further, we are not currently exploiting the algorithm's inherent parallelism. However, it does not make sense to focus on optimizing our algorithm at this stage (optimizations may be brittle). Once the branch-

[9]Both running the algorithm in Isabelle/HOL and running the SML code require trusting Isabelle/HOL's code generator in addition to the theorem prover's trusted core. Partial progress has been made on verifying Isabelle's code generator [44].

[10]The top-level correctness theorems for verified virtual substitution have a very similar shape to `qe_correct`, as they state that for each top-level formalized virtual substitution method V and valuation ν, `eval F` ν equals `eval (V F)` ν (cross-reference Sect. 2.2.4). This makes it easy to verify that, for any valuation ν, `eval F` ν equals `eval ((qe ∘ V) F)` ν.

ing reflects the full reduction of BKR, then inefficiencies (such as the unnecessary branching on constant coefficients) should be identified and handled appropriately. We do not consider our algorithm's present inefficiency to be a fatal flaw, since we envision it as being a (major) stepping stone on the way towards an optimized algorithm. As noted previously, unverified computer algebra systems have realized efficient QE in part because many have been extensively optimized over several decades; thus, it is natural that optimized verified algorithms will similarly take time to develop.

4.3 Related Work

From a theoretical standpoint, the most closely related work is one by Cyril Cohen, who formalized a sign-determination algorithm with reduction in Coq that, to our understanding, uses the same matrix equation as our algorithm, although the details of his formalization look quite different from ours.[11] To our knowledge, he has not yet used this improved sign-determination algorithm for a QE algorithm, and this work is unpublished, but a writeup is available on his webpage [14]. Additionally, because the algorithm we verify is a hybrid between Tarski's QE algorithm and BKR, our work shares some theoretical overlap with Cohen and Mahboubi's formalization of Tarski's algorithm in Coq [13, 15].

From a practical standpoint, we benefit from the well-developed Isabelle/HOL libraries. This includes, of course, our previous verification of univariate BKR [19] and our verification of virtual substitution [76], which have already been discussed at length. Additionally, we build on the formalization of pseudo-remainder sequences (recently made available on the AFP [51]) described by Li, Passmore, and Paulson [54]. Although we formalize our own functions to generate pseudo-remainder sequences, which interface well with our assumptions-based framework (and which are specialized to the `rmpoly` type), we derive insights from Li's code and mimic some of his structure in our functions, adapted appropriately to our purposes. We also benefit from proving a connection between our functions and his.

4.4 Takeaways and Future Directions

This chapter describes the formalization of Isabelle/HOL's first complete multivariate quantifier elimination (QE) algorithm for the first-order logic of real-closed fields. Our algorithm mixes ideas from Tarski's original QE algorithm [85] and more efficient algorithms by BKR [2] and Renegar [75]; the formalization requires rigorizing high-level mathematical insights [2, 75]. We realize a number of ideas suggested in the previous chapters of the thesis by extending our univariate formalization of BKR [19] to the multivariate world and by building on our virtual substitution framework [76] in order to use VS as an efficient preprocessing step for our hybrid algorithm. While our algorithm (on its own) currently has prohibitive inefficiency, its nontrivial library extensions and theoretical interest (including its potential to be extended into variant algorithms with promising parallel complexity [10, 22, 75]) make it a meaningful contribution.

[11]This is in part because the setup is considerably different: while we extended a univariate QE procedure with reduction into multivariate, Cohen added reduction to an already multivariate sign-determination procedure.

Future work includes first extending our algorithm to one that realizes the full reduction of BKR [2]; one main challenge is that this will make the algorithm highly recursive because, for example, the computation of each Tarski query will require a recursive call to the top-level sign-determination algorithm. In order to be able to modularize the algorithm—and thus the proofs—in the face of this recursion, one possible approach is to pass an "oracle function" for sign determination as an argument to individual functions (such as the computation of Tarski queries). This oracle function is simply a function which is assumed to satisfy the key properties of the top-level sign-determination algorithm. Then, in the body of the main BKR algorithm, this oracle function will be instantiated as the top-level sign determination function whenever necessary—for example, when a Tarski query computation is performed. The benefit of this approach is that many individual computations can then be moved outside the body of the top-level algorithm, which affords modularization of the proofs.

After this, it would be interesting to identify other areas of inefficiency and aggressively optimize. In addition to fine-tuning the branching to avoid splitting on trivial cases (most notably, on constants), one very significant (and challenging) task will be to optimize the computation of the Tarski queries; this was previously noted in the univariate case also [19]. Overall, our contribution lays considerable groundwork for more optimized verified QE algorithms with inherent parallelism.

Chapter 5

Conclusion

Developing methods for real quantifier elimination (QE) that are both practical and trustworthy is of considerable practical significance, as QE problems arise in safety-critical application domains. For example, rigorous logical proofs that seek to establish safety properties of models of cyber-physical systems (like planes, trains, drones, and surgical robots) often reduce to real QE problems. Currently, in practice, software like Mathematica is then used to solve these problems; since Mathematica (for example) is a blackbox tool, this raises the question of whether the answers it provides can be fully trusted.

This thesis takes practical steps to help develop *formally verified* support for algorithms for real quantifier elimination. This necessitates not only implementing QE algorithms but also providing associated rigorous logical correctness proofs, which make the implementation highly dependable. Working within the theorem prover Isabelle/HOL, we formalize the linear and quadratic cases of virtual substitution (VS); these practically impactful QE methods target problems with low-degree polynomials. We demonstrate the promising efficiency of our verified VS algorithms on real-world benchmarks. We also formalize (in Isabelle/HOL) a complete QE algorithm that mixes Tarski's QE method with insights from the Ben-Or, Kozen, and Reif (BKR) algorithm, which has good potential for parallelism. This result, which bridges the Theory A and Theory B communities, involved formalizing the *univariate* cases of both BKR and a variant due to Renegar, which may be of independent interest, as these are used in other parallel algorithms beyond real QE.

By contributing an experimentally successful formalization of linear and quadratic virtual substitution and by realizing Isabelle/HOL's first formally verified complete QE algorithm, this thesis advances the state of formally verified support for real QE while also laying considerable groundwork for future advances in this area. In the short-term future, it would be interesting to continue to optimize the verified VS implementation, and to combine it with formalizations of other efficient incomplete QE methods. In the long-term future, it would be compelling to develop an optimized version of verified univariate BKR that is in NC, and to formalize variants of (multivariate) BKR with strong parallel complexity bounds. Our considerable advances to Isabelle/HOL's multivariate polynomials library, as well as our contributions to Isabelle/HOL's matrix library, could help to enable such ambitious goals (and may additionally be of independent interest).

Appendix A

Virtual Substitution

This is the appendix associated with Chapter 2 of the thesis. It contains some of the key Isabelle/HOL code used within the Equality VS algorithm (see Sect. 2.2.2) and the General VS algorithm (see Sect. 2.2.3).

A.1 Linear and Quadratic Substitution

A.1.1 Linear Substitution

In Isabelle/HOL, we formalize linear substitution in the `linear_substitution` function. This function takes as input a natural number `var`, which indicates which variable is of interest, two real multivariate polynomials `a` and `b`, and an atom `A`. Calling `linear_substition var a b A` then returns the `atom fm` that is the result of virtually substituting the fraction `a/b` for the variable with de Bruijn index `var` into atom `A` (the `atom` and `atom fm` datatypes are explained in Sect. 2.3.1). To do this, `linear_substition` cases on the structure of A and follows the casework described in Sect. 2.2.2.

```
fun linear_substitution :: "nat ⇒ real mpoly ⇒ real mpoly ⇒ atom
  ⇒ atom" where
  "linear_substitution var a b (Eq p) =
  (let d = MPoly_Type.degree p var in
    (Eq (∑i∈{0..<(d+1)}.
      isolate_variable_sparse p var i*(a^i)*(b^(d-i))))
  )" |
  "linear_substitution var a b (Less p) =
  (let d = MPoly_Type.degree p var in
    let P = (∑i∈{0..<(d+1)}.
      isolate_variable_sparse p var i*(a^i)*(b^(d-i))) in
       (Less (P * (b ^ (d mod 2))))
   )" |
  "linear_substitution var a b (Leq p) =
  (let d = MPoly_Type.degree p var in
    let P = (∑i∈{0..<(d+1)}.
      isolate_variable_sparse p var i*(a^i)*(b^(d-i))) in
       (Leq (P * (b ^ (d mod 2))))
   )" |
  "linear_substitution var a b (Neq p) =
  (let d = MPoly_Type.degree p var in
    (Neq (∑i∈{0..<(d+1)}.
      isolate_variable_sparse p var i*(a^i)*(b^(d-i))))
  )"
```

In each case, the first step of the algorithm is to recover the degree of the polynomial `p` that we wish to substitute into (in the variable of interest `var`). We store this in `d`. Then, we can express the polynomial `p` as a summation of monomials with respect to the variable we are eliminating on (which is `var`). To read off the ith coefficient of the polynomial `p` with respect to `var`, we use the `isolate_variable_sparse p var i` function.

For the $=$ and $\neq$ cases, we multiply the ith coefficient by `a^i * b ^(d-i)`, to reflect that we have normalized by multiplying everything by `b^d` (more specifically, this reflects that $(a^i/b^i) \cdot b^d = a^i \cdot b^{d-i}$). For the $<$ and $\leq$ cases, we additionally need to check the parity of the degree `d`. When `d` is odd, we must multiply the whole polynomial by an additional factor of `b`, as explained in Sect. 2.2.2; in the code, we accomplish this by multiplying by the extra factor of `b^(d mod 2)` in the $<$ and $\leq$ cases.

A.1.2 Quadratic Substitution

In Isabelle/HOL, we formalize quadratic substitution in the `quadratic_sub` function. Here, `quadratic_sub var a b c d A` returns the result of virtually substituting $(a + b\sqrt{c})/d$ for the variable with de Bruijn index `var` in atom `A`. It cases on the structure of `A`. All of the cases use the same helper functions. As explained in Sect. 2.2.2, the cases differ in the final arrangement of the polynomial $A + B\sqrt{c}$, where A and B are multivariate polynomials that do not mention the variable we are eliminating on.

```
primrec quadratic_sub :: "nat ⇒
      real mpoly ⇒ real mpoly ⇒ real mpoly ⇒ real mpoly ⇒
      atom ⇒ atom fm" where
  "quadratic_sub var a b c d (Eq p) = (
    let (p1::real mpoly) = quadratic_part_1 var a b d (Eq p) in
    let (p2::real mpoly) = quadratic_part_2 var c p1 in
    let (A::real mpoly) = isolate_variable_sparse p2 var 0 in
    let (B::real mpoly) = isolate_variable_sparse p2 var 1 in
    And
      (Atom (Leq (A*B)))
      (Atom (Eq (A^2-B^2*c)))
)" |
  "quadratic_sub var a b c d (Less p) = (
    let (p1::real mpoly) = quadratic_part_1 var a b d (Less p) in
    let (p2::real mpoly) = quadratic_part_2 var c p1 in
    let (A::real mpoly) = isolate_variable_sparse p2 var 0 in
    let (B::real mpoly) = isolate_variable_sparse p2 var 1 in
    Or
      (And
        (Atom (Less(A)))
        (Atom (Less (B^2*c-A^2))))
      (And
        (Atom (Leq B))
        (Or
          (Atom (Less A))
          (Atom (Less (A^2-B^2*c)))))
)" |
  "quadratic_sub var a b c d (Leq p) = (
    let (p1::real mpoly) = quadratic_part_1 var a b d (Leq p) in
    let (p2::real mpoly) = quadratic_part_2 var c p1 in
    let (A::real mpoly) = isolate_variable_sparse p2 var 0 in
    let (B::real mpoly) = isolate_variable_sparse p2 var 1 in
    Or
      (And
        (Atom (Leq(A)))
        (Atom (Leq(B^2*c-A^2))))
      (And
        (Atom (Leq B))
        (Atom (Leq (A^2-B^2*c))))
)" |
  "quadratic_sub var a b c d (Neq p) = (
    let (p1::real mpoly) = quadratic_part_1 var a b d (Neq p) in
    let (p2::real mpoly) = quadratic_part_2 var c p1 in
    let (A::real mpoly) = isolate_variable_sparse p2 var 0 in
    let (B::real mpoly) = isolate_variable_sparse p2 var 1 in
    Or
      (Atom (Less(-A*B)))
      (Atom (Neq(A^2-B^2*c)))
)"
```

Intuitively, in each case we are splitting the quadratic substitution into two main steps (the fractional step and the square root step). Essentially, we want to substitute $\mathit{var} = (a + by)/d$ for a meta-variable y, and then substitute $y = \sqrt{c}$.

The helper function *quadratic_part_1* formalizes the fractional step, which is the normalization of denominators discussed in Sect. 2.2.2. It behaves like *linear_substitution* (discussed in Appendix A.1.1), except that the fractional polynomial that *quadratic_part_1* is substituting into variable *var* is *(a+b*(Var var))/d*.

Remark 7. *Notice that in the above, we are actually substituting a polynomial mentioning* var *for the variable* var*. This situation arises as a consequence of our multiple-step substitution process for quadratic roots: If we associate the polynomial variable* var *with the meta-variable* x *and create a second meta-variable* y*, then we want to perform the substitutions* $x = (a+by)/d$ *and* $y = \sqrt{c}$*. After performing the first substitution for meta-variable* x*, the polynomial variable* var *representing the meta-variable* x *gets eliminated. As such, it is okay for us to repurpose* *the same polynomial variable `var` to represent our new meta-variable y when performing the substitution $y = \sqrt{c}$. Further, reusing this specific variable `var` allows us to avoid potential conflicts with other variables within the polynomial.*

The helper function *quadratic_part_2* handles the parity check on the normalization factor (a key part of the square root substitution step), as described in Sect. 2.2.2. Here, we simply collect the summation of all the even-degree monomials (in our variable of interest `var`) into one polynomial A and all the odd-degree monomials into another polynomial B, so that ultimately our polynomial is of the form $A + B\sqrt{c}$ where A and B no longer involve square root terms.

```
fun quadratic_part_2 ::
    "nat ⇒ real mpoly ⇒ real mpoly ⇒ real mpoly" where
  "quadratic_part_2 var c p = (
  let deg = MPoly_Type.degree p var in
  ∑i∈{0..<deg+1}.
    (isolate_variable_sparse p var i)*(c^(i div 2)) *
        (Const(of_nat(i mod 2))) * (Var var)
   +(isolate_variable_sparse p var i)*(c^(i div 2)) *
        Const(1 - of_nat(i mod 2)))"
```

In this, `of_nat` casts natural numbers to real numbers. For each `0` $\leq$ `i` $<$ `deg + 1`, `Const(of_nat(i mod 2))` is 0 when `i` is even and 1 when `i` is odd. Similarly, `Const(1 - of_nat(i mod 2))` is 1 when `i` is even and 0 when `i` is odd. So,

```
(isolate_variable_sparse p var i)*(c^(i div 2)) *
         (Const(of_nat(i mod 2)))*(Var var)
```

collects the summation of the odd-degree monomials (in `var`) in the form $a_i c^{\lfloor i/2 \rfloor}\sqrt{c}$, for some coefficient a_i, and where `Var var` represents $\sqrt{c}$, while

```
(isolate_variable_sparse p var i)*(c^(i div 2)) *
         (Const(1 - of_nat(i mod 2)))
```

collects the summation of the even-degree monomials in `var` in the form $a_i c^{i/2}$. Notice that in the even cases, we are able to completely eliminate the square root, whereas the odd cases leave us with a variable of degree one, for which we will need to substitute $\sqrt{c}$.

Note that here we again exploit the technique explained in Remark 7 of reusing the variable index `var`. This is because we have split the square root substitution step (that is, the substitution of $y = \sqrt{c}$) into two substeps: the parity check performed by *quadratic_part_2* which collects terms into A and B, and the final step performed in *quadratic_sub*. Since A and B are both free in `var`, it is safe for us to use reuse index `var` to have *quadratic_part_2* return something of the form $A + B \cdot$ `var`. Then *quadratic_sub* pieces everything together and does the final substitution of $\sqrt{c}$ for `var`.

A.2 Top-Level Algorithms

We develop (and export to SML for experimentation) several top-level algorithms that use various combinations of VS procedures and optimizations; these are briefly detailed here.

The *VSLucky* algorithm recursively searches through the available atom conjunct list, searching for a quadratic equality atom that features a constant coefficient with respect to the variable x that we are eliminating. If it finds some $ax^2 + bx + c$ where a, b, or c is a nonzero constant, we are guaranteed that $ax^2 + bx + c$ is not the zero polynomial, and thus we have the following full elimination of the quantifier on x, without the remaining zero case (cross-reference Sect. 2.2.2):

$$\begin{aligned}
&\big(\exists x.(ax^2 + bx + c = 0 \land F)\big) \longleftrightarrow \\
&\Big((a = 0 \land b \neq 0 \land F_x^{-c/b}) \lor \\
&(a \neq 0 \land b^2 - 4ac \geq 0 \land (F_x^{(-b+\sqrt{b^2-4ac})/(2a)} \lor F_x^{(-b-\sqrt{b^2-4ac})/(2a)}))\Big).
\end{aligned}$$

Then we can use VS to expand the RHS of the equivalence. So, *VSLucky* is indeed lucky, because it fully removes the quantifier on the variable in question.

Next, the *VSEquality* algorithm performs the equality version of VS (Sect. 2.2.2) iteratively for all equality atoms of at most quadratic degree, and the *VSGeneral* algorithm performs the general version of VS (Sect. 2.2.3).[1]

Finally, we have *VSLEG* which performs all three of *VSLucky*, *VSEquality*, *VSGeneral* (in that order, as the first is the quickest while the last is the most general); this algorithm was the most competitive one on our benchmarks. We now discuss the correctness theorem for these algorithms.

A.2.1 Correctness of Top-Level Algorithms

The correctness of our top-level algorithms is established by the following theorem, which expresses that any initial formula φ is logically equivalent to the formula obtained by running the algorithm on φ; in other words, the two formulas (initial and final) have the same truth-value in every state, i.e. for every valuation of free variables.

```
theorem TopLevelSoundessTheorems:
  "∀ν. (eval (VSEquality φ) ν = eval φ ν)"
  "∀ν. (eval (VSGeneral φ) ν = eval φ ν)"
  "∀ν. (eval (VSLucky φ) ν = eval φ ν)"
  "∀ν. (eval (VSLEG φ) ν = eval φ ν)"
```

In this theorem, our `eval` function captures the semantics of substituting a valuation into a formula: For a valuation characterized by a list ν of real numbers and a formula φ, `eval φ ν` is true whenever φ is true at ν, i.e. when the ith entry of ν is plugged in for the ith free variable of φ (for all i). More precisely, the exact definition of `eval` is as follows:

```
fun eval :: "atom fm ⇒ real list ⇒ bool" where
  "eval (Atom a) ν = aEval a ν" |
  "eval (TrueF) _ = True" |
  "eval (FalseF) _ = False" |
```

[1]Additionally, we have *VSEquality_3_times*, which performs the equality algorithm three times, and *VSGeneral_3_times*, which performs the general algorithm three times. Although these account for the kind of potential edge cases discussed in Sect. 2.3.2, they did not realize significant experimental benefits over *VSEquality* and *VSGeneral*.

```
  "eval (And φ ψ) ν = ((eval φ ν) ∧ (eval ψ ν))" |
  "eval (Or φ ψ) ν = ((eval φ ν) ∨ (eval ψ ν))" |
  "eval (Neg φ) ν = (¬ (eval φ ν))" |
  "eval (ExQ φ) ν = (∃x. (eval φ (x#ν)))" |
  "eval (AllQ φ) ν = (∀x. (eval φ (x#ν)))" |
  "eval (AllN i φ) ν = (∀γ. length γ = i ⟶ (eval φ (γ @ ν)))" |
  "eval (ExN i φ) ν = (∃γ. length γ = i ∧ (eval φ (γ @ ν)))"
```

This is the canonical semantics for evaluating atom formulas. The interesting cases are the *Atom*, *ExQ*, and *AllQ* cases. We discuss each.

In the *Atom* case, the *aEval* function is as follows:

```
fun aEval :: "atom ⇒ real list ⇒ bool" where
  "aEval (Eq p) ν = (insertion (nth_default 0 ν) p = 0)" |
  "aEval (Less p) ν = (insertion (nth_default 0 ν) p < 0)" |
  "aEval (Leq p) ν = (insertion (nth_default 0 ν) p ≤ 0)" |
  "aEval (Neq p) ν = (insertion (nth_default 0 ν) p ≠ 0)"
```

Here, we are using the *insertion* function from the multivariate polynomials library [80] to insert the n values from list ν for variables $0, \dots, n-1$. This captures evaluation by substituting values for the free variables in the formula.

Because the first argument of the *insertion* function has type *(nat ⇒ 'a)*, where in our case *'a* has type *real*, we use Isabelle/HOL's standard library function *nth_default* to expand ν into a mapping from nats to reals. That is, *(nth_default 0 ν) i* is the ith element of ν when i is less than the length of ν (lists are 0-indexed in Isabelle/HOL), and 0 otherwise. Note that this means that if our list ν is not long enough to cover the valuations for all the free variables in the polynomial, the *nth_default* function will assign it a value of 0. This is merely a convenience; our correctness lemmas are unaffected by this, as they quantify over all possible valuations ν (and thus over all valuations of the correct length).

The *ExQ φ* case of a new existential quantifier is handled by embedding into Isabelle's built-in notion of existential quantification over the real numbers. More precisely, in *∃x. (eval φ (x#ν))*, Isabelle treats the quantified variable x as a new free variable, which is added to the front of the valuation ν, written *x#ν*. The *AllQ* case is similar.

As an example, consider evaluating

$$\exists x. \forall y. \, (x + y \cdot y + z > 0)$$

in a state ν where $\nu(z) = 1$. In our framework in Isabelle/HOL, this translates as:

```
eval (ExQ (AllQ (Var 1 + Var 0*Var 0 + Var 2 > 0)) 1
```

Here we have two quantified variables, Var 0 and Var 1, and one free variable, Var 2 (this is because we are using de Bruijn indices). Var 0 matches the *AllQ* quantifier and Var 1 matches the *ExQ* quantifier. We are considering the valuation where Var 2 is set to 1.

In the first step, we expand to:

```
Exists x.
    (eval (AllQ (Var 1 + Var 0*Var 0 + Var 2 > 0)) (x#1)),
```

and in the second step, we achieve:

```
Exists x. Forall y.
(eval (Var 1 + Var 0*Var 0 + Var 2 > 0) (y#(x#1))).
```
This asks whether there exists an x such that for all y, $x + y^2 + 1 > 0$, which matches the desired semantics.

We additionally include *ExN* and *AllN*, which take in two inputs i and φ and are equivalent to the single quantifier forms *ExQ* and *AllQ* repeated i times on φ (which, thanks to de Bruijn indices, are quantifiers for i different variables). In the *eval* function, these are represented via a quantified list γ of real number valuations of length i. Having these *ExN* and *AllN* in the representation allows for specialized algorithms like our block quantifer heuristics (see A.3.3).

Now that we have established that our correctness theorem is correctly stated, we discuss its proof.

A.2.2 Proving Correctness: Extensibility

It would be very tedious to independently prove the correctness theorems for all of our top-level algorithms. Instead, our framework is designed so that any optimization *opt* can easily be incorporated into our framework, as long as *opt* does not change the truth-value of any formula; i.e. our framework is designed to be ***extensible***. This allows us to cleanly substitute different combinations of optimizations into our top-level QE algorithms without incurring the burden of significant reproving.

This extensibility can be seen in the following correctness lemma for our *QE_dnf* function, which lifts QE algorithms defined for a single quantifier to apply to all quantifiers using the modified DNF transformation discussed in Sect. 2.3.2:

theorem *QE_dnf_eval:*
  assumes *steph : "⋀var amount L F ν. amount≤var+1 ⟹*
    eval (ExN (var+1) (list_conj (map fm.Atom L @ F))) ν =
    eval (ExN (var+1) (step amount var L F)) ν"
  assumes *opth : "⋀ν F . eval (opt F) ν = eval F ν"*
  shows *"eval (QE_dnf opt step φ) ν = eval φ ν"*

Here, the *step* function is intended to be a function that performs virtual substitution. Its behavior is governed by the *steph* hypothesis. The *opt* function is intended to be a function that performs various optimizations, and its behavior is governed by the *opth* hypothesis. Intuitively, *QE_dnf_eval* proves that any function that obeys the *steph* hypothesis (as we prove that our VS procedures do) and any functions that obey the *opth* hypothesis (as we prove that our optimizations do) can be combined to create an overall top-level QE procedure. We now discuss the characteristics of *opt* and *step* further.

The *opth* hypothesis assumes that we have a procedure *opt* which preserves the truth value of the *eval* function for every valuation on every formula. In our QE framework, this function is called before performing the DNF transformation of Sect. 2.3.2, and then *step* is called after performing the DNF transformation.

As specified by the *steph* hypothesis, the *step* function receives as input two natural numbers *amount* and *var*, a list of atoms *L*, and a list of atom formulas *F*. Here, *amount* is designed to track for how many of the *var+1* existential quantifiers in the prefix QE has not yet been attempted. Since the modified DNF transformation (see Sect. 2.3.2) is performed before *step*, this can move some quantifiers for which we have already attempted VS to the top-level of the formula. In such cases, tracking `amount` allows us to stop computation early, rather than re-attempting VS on quantifiers where it previously did not apply.

The left-hand side of the equality in `steph` represents the form that our formula takes in one of the disjuncts of the DNF transformation (cross-reference Sect. 2.3.2): there are `var+1` existential quantifiers in this disjunct (because we use zero-indexing for variables), and everything in `L` and `F` is conjuncted. Intuitively, this is what we pass into virtual substitution. The right-hand side of the equality in `steph` represents the result of calling the `step` function. The equality captures that the original formula and the formula we obtain by applying `step` have the same truth value in any valuation ν. So, overall, the `steph` equality captures that `step` can perform any arbitrary manipulation of the inputs as long as it preserves logical equivalence.

The `QE_dnf_eval` lemma showcases the extensible nature of our framework and allows us to determine the soundness of the top level algorithms described in Appendix A.2, but it makes no claim about whether VS is actually simplifying our formula. We are unable to make such claims for this top-level procedure, as we must allow for our VS algorithms to fail to make progress in cases where VS does not apply (e.g. in the presence of high degree polynomials), but we now discuss the specific cases where our VS `step` procedures successfully eliminate quantifiers, as well as the important correctness lemmas for these procedures.

A.2.3 Proving Correctness: VS

Our `elimVar` function is the multivariate analog of the univariate general VS procedure discussed in Sect. 2.2.3) (and it closely resembles this procedure). More specifically, `elimVar` is an overarching proof procedure that analyzes the roots of a polynomial and applies the appropriate VS algorithms for both the equality and the off-root cases. It checks whether we wish to substitute the exact root (for $=$ and $\leq$ atoms) or the off-roots (for $<$ and $\neq$ atoms).

As input, `elimVar` receives a variable `var` we are eliminating on, a list of atoms `L` and atom formulas `F` which are joined by conjunction, and the atom `At` that we wish to substitute, which is guaranteed to be at most quadratic (cross-reference Sect. 2.3.2). Each of the top-level VS algorithms utilizes `elimVar` as a helper function to substitute particular atoms. Significantly, `elimVar` has the property that it removes the variable it is substituting. This is expressed in the following lemma, where our `variableIsRemoved` function is true when the input variable `var` is not present in a formula:

```
lemma elimVar_removes_variable: "variableIsRemoved var
  (elimVar var L F At)"
```

If we assume that the atom `At` that the `elimVar` function takes as input is an equality atom with a quadratic or linear polynomial with respect to the variable being eliminated, we can prove the following important lemma for the equality case of VS:

```
lemma elimVar_eq:
  assumes hlength: "length (ν::real list) = var"
  assumes noVariable: "var ∉ vars a" "var ∉ vars b" "var ∉ vars c"
  assumes inList: "Eq (a*(Var var)^2+b*(Var var)+c)  ∈ set(L)"
  assumes nonzero:
    "insertion' a (ν @ x # ν') ≠ 0 ∨ insertion' b (ν @ x # ν') ≠ 0"
  shows "(∃x. eval (list_conj (map fm.Atom L @ F)) (ν @ x # ν')) =
(∃x. eval(elimVar var L F (Eq(a*(Var var)^2+b*(Var var)+c)))(ν@x#ν'))"
```

Here, we are assuming that we have the equality atom `a*(Var var)^2+b*(Var var)+c` within our list of atoms `L`, where the variable `var` does not occur in the coefficients `a`, `b`, and `c`; these conditions are captured by the `inList` and `noVariable` hypotheses, respectively. We also assume that this polynomial is quadratic or linear in the particular variable of interest (in the `nonzero` hypothesis). Performing `elimVar` on this atom allows us to remove the quantifier completely: in the linear case, when `Eq(a*(Var var)^2 + b*(Var var) + c)` is `Eq(b*(Var var) + c)`, `elimVar` virtually substitutes `-c/b` for `var` into every formula; in the quadratic case, `elimVar` produces the disjunct of virtually substituting the two possible quadratic roots into every formula. This lemma establishes the correctness of the Equality VS algorithm as described in Section 2.2.2. Note that we could strengthen the $\exists$ quantifier on the RHS of the equality to a $\forall$ by using `elimVar_removes_variable`.

Additionally, we use `elimVar` to formulate the lemma for the general VS case:

```
lemma gen_qe_eval:
  assumes "F = list_conj (map Atom L)"
  assumes "all_degree_2 var L"
  assumes "length ν > var"
  shows "(∃x. (eval F (ν[var:=x]))) = (∀x. (eval
      (list_disj (mapNegInfinity var L # mapElimVar var L))
    (ν[var:=x])))"
```

This lemma regards the executable multivariate procedure that corresponds to the univariate lemma explained in Section 2.2.3. Here, we assume that our formula `F` is a conjunction of atoms `L` as expressed in the first hypothesis. Additionally, the second hypothesis expresses that `F` has the requisite shape for general VS to apply; this means that every polynomial in `L` has at most degree two with respect to the variable we are eliminating, `var`. The third assumption states that our valuation ν is long enough to cover the variable in question (which makes it a valid valuation).

Under these assumptions, we show that there exists an x that can be substituted for `var` to make formula `F` true iff substituting one of the sample points prescribed by virtual substitution (which are negative infinity, the roots of the $=$ and $\leq$ atoms in `L` with respect to `var`, and the off-roots of the $\neq$ and $<$ atoms in `L` with respect to `var`) makes `F` hold. Notice that the $\forall$ quantification on the x on the RHS of the equality is permissible because we have actually eliminated the variable x from the formula after applying virtual substitution (this is established by the `elimVar_removes_variable` lemma).

A.2.4 More Polynomial Library Contributions

In order to prove correctness of VS, we needed to formalize a large number of additions to the Isabelle/HOL multivariate polynomials library. For example, we added a partial derivative function (which is needed to formalize VS for infinitesimals, as discussed in Sect. 2.2.3); and prove its correctness for polynomials of degree at most two (more is not needed for quadratic virtual substitution).

For execution, we also needed to extend code theorems for the generically defined `insertion` function, which inserts a valuation into a polynomial, in order to be able to compute the result of valuations and export our program. This required a number of lemmas that rely on the `monomials` function, which separates a polynomial into a sum of monomials, and the `degree` function, which computes the degree of a multivariate polynomial with respect to a single variable.

To make these new functions usable in proofs, we formulated a large collection of lemmas for polynomials with real-valued coefficients. These include a variety of simplification lemmas for the interaction of the `isolate_variable_sparse` function discussed in Sect. 2.3.4 and the `insertion`, and `degree` functions across summations and products of polynomials.

To utilize the variables within polynomials as de Bruijn indices, we implemented various lifting and substitution operations. Our `liftPoly` function takes in a lower limit d, a lifting amount a, and a polynomial p and returns a polynomial which reindexes variables within p such that every variable greater or equal to d is increased by a. This is commonly denoted by $p\uparrow_d^a$. This `liftPoly` is needed in cases where we want to reshuffle formulas to increase the number of quantifiers surrounding a formula. For example, $\forall.((\exists A) \wedge B)$ is equivalent to $\forall.\exists.(A \wedge (B\uparrow_1^1))$.

We also need an inverse function to `liftPoly`, which we call `lowerPoly`. Whenever we have eliminated a quantified variable with our QE procedure, we can drop that quantifier and use `lowerPoly` to reindex all the other variables accordingly. Our `clearQuantifiers` procedure implements this.

A.3 Optimizations

We formalize a number of optimizations for VS, as optimizations are critical for achieving reasonable performance. It will take some time to catch up to the highly optimized performance of tools like Wolfram Engine and Redlog.

This is the benefit of an extensible framework: future optimizations can be easily integrated. From Appendix A.2.2, we see that any optimization function `opt` that satisfies the truth-preservation lemma `lemma "eval (opt F) xs = eval F xs"` can be cleanly integrated into our algorithm (using the `QE_dnf_eval` theorem), and the composition of several of these optimization functions directly preserves this property. We discuss our current optimizations in this appendix.

A.3.1 Unpower

As our algorithm performs VS on quadratic and linear polynomials, it is critical to reduce the degree of polynomials whenever possible. The most natural simplification we can perform is to factor out a common x^n from every monomial:

$$\sum a_i x^{n+i} = x^n \sum a_i x^i = px^n$$

From here, we can split the atom $px^n \sim 0$, where $\sim \in \{=, < . \leq, \neq\}$, into lower-degree atoms that involve p and x^n separately.

In the equality case, $px^n = 0$ reduces to $x = 0 \vee p = 0$, as the product of the components is zero if and only if at least one of the components is zero. For inequalities, we case on the parity of n. If we have an even exponent, we can reduce $px^n < 0$ into $p < 0 \wedge x \neq 0$, since x^n is nonnegative. Otherwise, we must assert that the sign values differ: $(p < 0 \wedge x > 0) \vee (p > 0 \wedge x < 0)$.

All $\neq$ are treated as negated equality atoms. For $\leq$ atoms, we follow a similar structure to $<$: when n is even, the result is $p \leq 0 \vee x = 0$, and when n is odd the result is $p = 0 \vee (p < 0 \wedge x \geq 0) \vee (p > 0 \wedge x \leq 0)$.

A.3.2 Simplifying Constants

It is clear that the atom $5 = 0$ is always false and can be replaced by the `FalseT` formula. Our `simpfm` function replaces constant polynomial atoms with their respective `TrueT` or `FalseT` evaluations and performs shortcut optimizations for the $\vee, \wedge, \neg$ connectives. This is especially useful when our QE algorithm is successful on closed formulas: since no more variables are present, the formula becomes a collection of constant polynomial atoms joined by connectives, which means the whole formula can be reduced to either `TrueF` or `FalseF` by `simpfm`.

This constant identification and constant folding is also crucial for effiency within the QE procedure to cut down the expansion of the formula. Recall that we transform QE problems to have the form

$$\bigvee \exists z_0. \cdots \exists z_n. \exists x. \left(\left(\bigwedge A_i \right) \wedge \left(\bigwedge \forall y. F_j \right) \right), \tag{A.1}$$

where the A_i are atoms and the F_i are formulas (see Sect. 2.3.2). Now consider a QE problem of the form $\exists x.(ax^2 + bx + c = 0 \wedge F)$. In the event that at least one of a, b, or c is a nonzero constant polynomial, we can immediately determine the *only* possible values of x in the whole formula, which are just the roots of this specific polynomial. As such, in both our general QE and equality QE algorithms, for each disjunct in a formula of the form of (A.1), our algorithm performs a linear scan for these "lucky" atoms within the conjunct list of atoms before proceeding with the VS algorithm. If a lucky atom is found, we immediately eliminate its associated quantifier and then proceed with other steps of the algorithm. This optimization demonstrates significant experimental benefits, as it eliminates quantifiers without greatly increasing the size of the formula. It also utilizes the DNF form optimizations discussed in Section 2.3.2 to reach underneath existential quantifiers to find more "lucky" atoms. It could be further adapted in the future to, for example, allow for simplifications underneath universal quantifiers.

Even better than the "lucky" atoms, where a single coefficient is constant, are "luckiest" atoms, in which all coefficients are constants: with luckiest atoms, we are substituting real numbers rather than polynomials. We found that "luckiest" atoms are empirically very significant (identifying and cleverly utilizing them yields significant speedup), so our algorithms preprocess input formulas for these kinds of atoms and perform virtual substitution on them first.

A.3.3 Variable Ordering Heuristics

Variable ordering heuristics are of great practical significance in QE [60]. We proved that one can freely swap the ordering of quantifiers in a homogeneous *block* (a number of existential or of universal quantifiers that occur in a row) without affecting a formula's truth value. To capitalize on this, our `groupQuantifiers` function locates instances of homogeneous blocks in a formula and converts them into an equivalent block quantifier representation; for example, `ExQ (ExQ F)` is converted into `ExN 2 F` (see Sect. 2.3.1). This allows later algorithms to identify blocks by pattern matching. When the VS algorithm reaches a block `ExN n F`, it uses the modified DNF algorithm on `F` and focuses on a single disjunct, at which point we invoke a heuristic function to choose which of the quantifiers in the innermost block to eliminate first.

Since DNF yields a disjunct list of conjuncts, the input information to the heuristic function (at each disjunct) is a conjunct list of atoms `L` and formulas `F`. As such, a heuristic function `H` is of the type `H :: nat ⇒ atom list ⇒ atom fm list ⇒ nat`, where `H (N-1) L F` analyses the input conjuncts `L` and `F` and determines the optimal variable ranging from `0` to `N-1` to eliminate on (remember that we use 0-indexing for variables). After selecting a variable `var` to eliminate, we swap `var` with variable `N-1`, perform VS on the newly reindexed `var`, optimize the result, and then run DNF again[2]. We then recursively proceed on each new disjunct until we have attempted to eliminate every quantifier.

The only property that the heuristic function must satisfy for correctness purposes is that it must suggest a variable within the block of interest; this allows users to create their own heuristics without incurring significant proof burden. Verifying this property is trivial for heuristics that check their result and explicitly disallow results outside of the desired range.

We implement three heuristic functions: one of our own design, one based on the literature, and the identity heuristic. The identity heuristic always returns the innermost variable, which yields experimental results comparable to an earlier version of the framework which did not implement block quantifiers; this demonstrates that we do not incur significant overhead by supporting block quantifiers. We also implement a heuristic based on Brown's heuristic for quantifier ordering for CAD, as presented in [37]; this yielded promising experimental results. Lastly, the heuristic that we designed both chooses which variable to eliminate first and also chooses which VS algorithm to use at each step; this heuristic was the most experimentally successful of the three. Overall, our results indicate that block quantifiers do not introduce significant overhead and confirm that variable ordering heuristics are of practical significance in virtual substitution.

[2]As an implementation detail, although our VS construction typically quantifiers inside-out, so that the innermost quantifier is eliminated first, within blocks we eliminate outside-in, so in a block of length `N`, the variable with index `N-1` is eliminated first.

Bibliography

[1] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry*. Springer, Berlin, Heidelberg, second edition, 2006. doi: 10.1007/3-540-33099-2. 1.2, 3.1.1, 3.1.3, 3.1.3, (3), 4, 4.1.2

[2] Michael Ben-Or, Dexter Kozen, and John H. Reif. The complexity of elementary algebra and geometry. *J. Comput. Syst. Sci.*, 32(2):251–264, 1986. doi: 10.1016/0022-0000(86)90029-2. 1.1, 1.2, 3, 3.1, 3.3, 2, 4, 4.1.1, 2, 4.1.2, 8, 4.2.1, 4.2.2, 4.4

[3] Rose Bohrer, Vincent Rahli, Ivana Vukotic, Marcus Völp, and André Platzer. Formally verified differential dynamic logic. In Yves Bertot and Viktor Vafeiadis, editors, *CPP*, pages 208–221, New York, 2017. ACM. ISBN 978-1-4503-4705-1. doi: 10.1145/3018610.3018616. 1

[4] Rose Bohrer, Yong Kiam Tan, Stefan Mitsch, Andrew Sogokon, and André Platzer. A formal safety net for waypoint-following in ground robots. *IEEE Robotics Autom. Lett.*, 4(3):2910–2917, 2019. doi: 10.1109/LRA.2019.2923099. 1

[5] Russell J. Bradford, James H. Davenport, Matthew England, AmirHosein Sadeghimanesh, and Ali Kemal Uncu. The DEWCAD project: pushing back the doubly exponential wall of cylindrical algebraic decomposition. *ACM Commun. Comput. Algebra*, 55(3):107–111, 2021. doi: 10.1145/3511528.3511538. 1.2

[6] Christopher W. Brown. Improved projection for cylindrical algebraic decomposition. *J. Symb. Comput.*, 32(5):447–465, 2001. doi: 10.1006/jsco.2001.0463. 1, 1.2

[7] Christopher W. Brown. QEPCAD B: a program for computing with semi-algebraic sets using cads. *SIGSAM Bull.*, 37(4):97–108, 2003. doi: 10.1145/968708.968710. URL `https://doi.org/10.1145/968708.968710`. 1, 1.2

[8] Christopher W. Brown and Scott McCallum. Enhancements to lazard's method for cylindrical algebraic decomposition. In François Boulier, Matthew England, Timur M. Sadykov, and Evgenii V. Vorozhtsov, editors, *CASC*, volume 12291 of *LNCS*, pages 129–149. Springer, 2020. doi: 10.1007/978-3-030-60026-6_8. 1.2

[9] John F. Canny. Some algebraic and geometric computations in PSPACE. In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 460–467. ACM, 1988. doi: 10.1145/62212.62257. 1.1

[10] John F. Canny. Improved algorithms for sign determination and existential quantifier elimination. *Comput. J.*, 36(5):409–418, 1993. doi: 10.1093/comjnl/36.5.409. 1.1, 1.2, 3, 4.4

[11] Amine Chaieb. *Automated methods for formal proofs in simple arithmetics and algebra*. PhD thesis, Technische Universität München, 2008. URL `https://mediatum.ub.tum.de/doc/649541/649541.pdf`. 1.2, 1.2, 2.1

[12] Rachel Cleaveland, Stefan Mitsch, and André Platzer. Formally verified next-generation airborne collision avoidance games in ACAS X. *ACM Trans. Embed. Comput. Syst.*, 22(1): 10:1–10:30, 2023. doi: 10.1145/3544970. 1

[13] Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory*. PhD thesis, École polytechnique, Nov 2012. URL `https://perso.crans.org/cohen/papers/thesis.pdf`. 1.2, 3.1.3, 4.3

[14] Cyril Cohen. Formalization of a sign determination algorithm in real algebraic geometry. Preprint on webpage at https://hal.inria.fr/hal-03274013/document, 2021. 9, 4.3

[15] Cyril Cohen and Assia Mahboubi. Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. *Log. Methods Comput. Sci.*, 8(1), 2012. doi: 10.2168/LMCS-8(1:2)2012. 1.2, 1.2, 3.1.3, 3.3, 4.3

[16] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H. Barkhage, editor, *Automata Theory and Formal Languages*, volume 33 of *LNCS*, pages 134–183. Springer, 1975. doi: 10.1007/3-540-07407-4_17. 1, 1.2

[17] George E. Collins and H. Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *J. Symb. Comput.*, 12(3):299–328, 1991. doi: 10.1016/S0747-7171(08)80152-6. 1, 1.2

[18] Katherine Cordwell, César Muñoz, and Aaron Dutle. Improving automated strategies for univariate quantifier elimination. Technical Memorandum NASA/TM-20205010644, NASA, Langley Research Center, Hampton VA 23681-2199, USA, January 2021. URL `https://shemesh.larc.nasa.gov/fm/papers/NASA-TM-20205010644.pdf`. 1.2, 1.2, 3.3

[19] Katherine Cordwell, Yong Kiam Tan, and André Platzer. A verified decision procedure for univariate real arithmetic with the BKR algorithm. In Liron Cohen and Cezary Kaliszyk, editors, *ITP*, volume 193 of *LIPIcs*, pages 14:1–14:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPIcs.ITP.2021.14. 3, 4.1.1, 4.1.2, 8, 4.2, 4.2.2, 4.3, 4.4

[20] Katherine Cordwell, Yong Kiam Tan, and André Platzer. The BKR decision procedure for univariate real arithmetic. *Archive of Formal Proofs*, April 2021. `https://www.isa-afp.org/entries/BenOr_Kozen_Reif.html`, Formal proof development. 3, 3, 8, 4.1.2

[21] 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 Marijn Heule and Sean A. Weaver, editors, *SAT*, volume 9340 of *LNCS*, pages 360–368. Springer, 2015. doi: 10.1007/978-3-319-24318-4_26. 2, *c*)

[22] Felipe Cucker, Hervé Lanneau, Bud Mishra, Paul Pedersen, and Marie-Françoise Roy. NC algorithms for real algebraic numbers. *Appl. Algebra Eng. Commun. Comput.*, 3:

79–98, 1992. doi: 10.1007/BF01387193. URL `https://doi.org/10.1007/BF01387193`. 1.2, 4.4

[23] James H. Davenport. Varieties of doubly-exponential behaviour in cylindrical algebraic decomposition. *CEUR Workshop Proceedings*, 3273:31–40, August 2022. ISSN 1613-0073. 1

[24] James H. Davenport and Joos Heintz. Real quantifier elimination is doubly exponential. *J. Symb. Comput.*, 5(1/2):29–35, 1988. doi: 10.1016/S0747-7171(88)80004-X. 1

[25] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24. 2, *d*)

[26] Leonardo Mendonça de Moura and Grant Olney Passmore. Computation in real closed infinitesimal and transcendental extensions of the rationals. In Maria Paola Bonacina, editor, *CADE*, volume 7898 of *LNCS*, pages 178–192. Springer, 2013. doi: 10.1007/978-3-642-38574-2_12. 6

[27] Jose Divasón, Sebastiaan J. C. Joosten, René Thiemann, and Akihisa Yamada. A formalization of the Berlekamp-Zassenhaus factorization algorithm. In Yves Bertot and Viktor Vafeiadis, editors, *CPP*, pages 17–29. ACM, 2017. doi: 10.1145/3018610.3018617. (1)

[28] Jose Divasón and Jesús Aransay. Rank-nullity theorem in linear algebra. *Archive of Formal Proofs*, January 2013. ISSN 2150-914x. `https://isa-afp.org/entries/Rank_Nullity_Theorem.html`, Formal proof development. 3.2.2

[29] Jose Divasón and Jesús Aransay. Gauss-Jordan algorithm and its applications. *Archive of Formal Proofs*, September 2014. ISSN 2150-914x. `https://isa-afp.org/entries/Gauss_Jordan.html`, Formal proof development. 5

[30] Andreas Dolzmann and Thomas Sturm. REDLOG: computer algebra meets computer logic. *SIGSAM Bull.*, 31(2):2–9, 1997. doi: 10.1145/261320.261324. 1.1, 2, *b*)

[31] Andreas Dolzmann and Thomas Sturm. Simplification of quantifier-free formulae over ordered fields. *J. Symb. Comput.*, 24(2):209–231, 1997. doi: 10.1006/jsco.1997.0123. 2.5

[32] Andreas Dolzmann, Andreas Seidl, and Thomas Sturm. Efficient projection orders for CAD. In Jaime Gutierrez, editor, *ISSAC*, pages 111–118. ACM, 2004. doi: 10.1145/1005285.1005303. URL `https://doi.org/10.1145/1005285.1005303`. 1.2

[33] Lionel Ducos. Optimizations of the subresultant algorithm. *J. Pure Appl. Algebra*, 145(2):149–163, 2000. doi: 10.1016/S0022-4049(98)00081-4. 3.2.3

[34] Antonio J. Durán, Mario Pérez, and Juan L. Varona. The misfortunes of a trio of mathematicians using computer algebra systems. can we trust in them? *Notices of the AMS*, 61 (10):1249–1252, 2014. doi: 10.1090/noti1173. 1

[35] Manuel Eberl and René Thiemann. Factorization of polynomials with algebraic coefficients. *Archive of Formal Proofs*, November 2021. ISSN 2150-914x. `https://isa-afp.org/entries/Factor_Algebraic_Polynomial.html`, Formal proof development. 4.2.3

[36] Matthew England and Dorian Florescu. Comparing machine learning models to choose the

variable ordering for cylindrical algebraic decomposition. In Cezary Kaliszyk, Edwin C. Brady, Andrea Kohlhase, and Claudio Sacerdoti Coen, editors, *CICM*, volume 11617 of *LNCS*, pages 93–108. Springer, 2019. doi: 10.1007/978-3-030-23250-4_7. 1.2

[37] Dorian Florescu and Matthew England. Algorithmically generating new algebraic features of polynomial systems for machine learning. *CoRR*, abs/1906.01455, 2019. URL `http://arxiv.org/abs/1906.01455`. A.3.3

[38] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In Amy P. Felty and Aart Middeldorp, editors, *CADE*, volume 9195 of *LNCS*, pages 527–538. Springer, 2015. doi: 10.1007/978-3-319-21401-6_36. 1

[39] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *CADE*, volume 7898 of *LNCS*, pages 208–214. Springer, 2013. doi: 10.1007/978-3-642-38574-2_14. 2.1

[40] John Harrison. Verifying nonlinear real formulas via sums of squares. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *LNCS*, pages 102–118. Springer, 2007. doi: 10.1007/978-3-540-74591-4_9. 1.2, 1.2

[41] Joos Heintz, Marie-Françoise Roy, and Pablo Solernó. On the theoretical and practical complexity of the existential theory of reals. *Comput. J.*, 36(5):427–431, 1993. doi: 10.1093/comjnl/36.5.427. 5, 4.2.4

[42] Hoon Hong. Comparison of several decision algorithms for the existential theory of the reals. Technical report, RISC, 1991. URL `https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.8707`. 5, 4.2.4

[43] Galen B. Huntington. *Towards an efficient decision procedure for the existential theory of the reals*. PhD thesis, 2008. URL `https://www.proquest.com/dissertations-theses/towards-efficient-decision-procedure-existential/docview/304695909/se-2`. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-02-23. 5

[44] Lars Hupel and Tobias Nipkow. A verified compiler from Isabelle/HOL to CakeML. In Amal Ahmed, editor, *ESOP*, volume 10801 of *LNCS*, pages 999–1026. Springer, 2018. doi: 10.1007/978-3-319-89884-1_35. 2, 2.2.4, 9

[45] Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Aurora C. Schmidt, Ryan W. Gardner, Stefan Mitsch, and André Platzer. A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system. *Int. J. Softw. Tools Technol. Transf.*, 19(6):717–741, 2017. doi: 10.1007/s10009-016-0434-1. 1

[46] Dejan Jovanovic and Leonardo Mendonça de Moura. Solving non-linear arithmetic. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR*, volume 7364 of *LNCS*, pages 339–354. Springer, 2012. doi: 10.1007/978-3-642-31365-3_27. 2.1

[47] Aditi Kabra, Stefan Mitsch, and André Platzer. Verified train controllers for the federal railroad administration train kinematics model: Balancing competing brake and track forces.

IEEE Trans. Comput. Aided Des. Integr. Circuits Syst., 41(11):4409–4420, 2022. doi: 10.1109/TCAD.2022.3197690. 1

[48] Katherine Kosaian, Yong Kiam Tan, and André Platzer. A first complete algorithm for real quantifier elimination in Isabelle/HOL. *Archive of Formal Proofs*, December 2022. `https://www.isa-afp.org/entries/Quantifier_Elimination_Hybrid.html`, Formal proof development. 4

[49] Katherine Kosaian, Yong Kiam Tan, and André Platzer. A first complete algorithm for real quantifier elimination in Isabelle/HOL. In Brigitte Pientka and Steve Zdancewic, editors, *CPP*, pages 211–224, New York, 2023. ACM. ISBN 9798400700262. doi: 10.1145/3573105.3575672. 4

[50] Marek Košta. *New concepts for real quantifier elimination by virtual substitution*. PhD thesis, Universität des Saarlandes, 2016. 2

[51] Wenda Li. The Sturm-Tarski theorem. *Archive of Formal Proofs*, September 2014. ISSN 2150-914x. `https://isa-afp.org/entries/Sturm_Tarski.html`, Formal proof development. 3.1.3, 3.2, 4.1.2, 4.1.2, 4.3

[52] Wenda Li and Lawrence C. Paulson. A modular, efficient formalisation of real algebraic numbers. In Jeremy Avigad and Adam Chlipala, editors, *CPP*, pages 66–75. ACM, 2016. doi: 10.1145/2854065.2854074. 3.3, 4.1.2

[53] Wenda Li and Lawrence C. Paulson. Counting polynomial roots in Isabelle/HOL: a formal proof of the Budan-Fourier theorem. In Assia Mahboubi and Magnus O. Myreen, editors, *CPP*, pages 52–64. ACM, 2019. doi: 10.1145/3293880.3294092. 3.2.3

[54] Wenda Li, Grant Olney Passmore, and Lawrence C. Paulson. Deciding univariate polynomial problems using untrusted certificates in Isabelle/HOL. *J. Autom. Reason.*, 62(1): 69–91, 2019. doi: 10.1007/s10817-017-9424-6. (document), 1.2, 1.2, 1.2, 3.2.3, **??**, 3.1, 3.3, 4.1.2, 6, 4.1.3, 4.2, 4.3

[55] Assia Mahboubi. Implementing the cylindrical algebraic decomposition within the coq system. *Math. Struct. Comput. Sci.*, 17(1):99–127, 2007. doi: 10.1017/S096012950600586X. 1.2, 1.2

[56] Scott McCallum. An improved projection operation for cylindrical algebraic decomposition. In B. F. Caviness, editor, *EUROCAL*, volume 204 of *LNCS*, pages 277–278. Springer, 1985. doi: 10.1007/3-540-15984-3_277. 1, 1.2

[57] Scott McCallum. Solving polynomial strict inequalities using cylindrical algebraic decomposition. *Comput. J.*, 36(5):432–438, 1993. doi: 10.1093/comjnl/36.5.432. 3.3

[58] Sean McLaughlin and John Harrison. A proof-producing decision procedure for real arithmetic. In Robert Nieuwenhuis, editor, *CADE*, volume 3632 of *LNCS*, pages 295–314. Springer, 2005. doi: 10.1007/11532231_22. 1.2, 1.2, 1.2, 4

[59] Bhubaneswar Mishra and Paul Pedersen. Arithmetic with real algebraic numbers is in NC. In Shunro Watanabe and Morio Nagata, editors, *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC '90, Tokyo, Japan, August 20-24, 1990*, pages 120–126. ACM, 1990. doi: 10.1145/96877.96909. 1.1

[60] Casey B. Mulligan, Russell J. Bradford, James H. Davenport, Matthew England, and Zak Tonks. Quantifier elimination for reasoning in economics. *CoRR*, abs/1804.10037, 2018. URL `http://arxiv.org/abs/1804.10037`. 2.4, A.3.3

[61] César A. Muñoz, Anthony J. Narkawicz, and Aaron Dutle. A decision procedure for univariate polynomial systems based on root counting and interval subdivision. *J. Formaliz. Reason.*, 11(1):19–41, 2018. doi: 10.6092/issn.1972-5787/8212. 1.2, 1.2, 1.2

[62] Jasper Nalbach, Erika Ábrahám, Philippe Specht, Christopher W. Brown, James H. Davenport, and Matthew England. Levelwise construction of a single cylindrical algebraic cell. *J. Symb. Comput.*, 2023. ISSN 0747-7171. doi: https://doi.org/10.1016/j.jsc.2023.02.007. 1.2

[63] Anthony Narkawicz, César A. Muñoz, and Aaron Dutle. Formally-verified decision procedures for univariate polynomial computation based on Sturm's and Tarski's theorems. *J. Autom. Reason.*, 54(4):285–326, 2015. doi: 10.1007/s10817-015-9320-x. 1.2, 1.2, 3.2.3, 3.3

[64] Tobias Nipkow. Linear quantifier elimination. *J. Autom. Reason.*, 45(2):189–212, 2010. doi: 10.1007/s10817-010-9183-0. 1.2, 1.2, 2.1, 2.3.1, 2.3.2, 4.1.3

[65] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. ISBN 3-540-43376-7. doi: 10.1007/3-540-45949-9. 1.1, 3.1

[66] Grant Olney Passmore. *Combined Decision Procedures for Nonlinear Arithmetics, Real and Complex*. PhD thesis, School of Informatics, University of Edinburgh, 2011. 1, 1.1, 2

[67] Lawrence C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5 (3):363–397, 1989. doi: 10.1007/BF00248324. 1.1

[68] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *IWIL*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010. 1.1, 3.2, 4.2

[69] André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010. ISBN 978-3-642-14508-7. doi: 10.1007/978-3-642-14509-4. 1

[70] André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, Cham, 2018. ISBN 978-3-319-63587-3. doi: 10.1007/978-3-319-63588-0. 1, 2.2.3

[71] André Platzer and Yong Kiam Tan. Differential equation invariance axiomatization. *J. ACM*, 67(1):6:1–6:66, 2020. doi: 10.1145/3380825. 1

[72] André Platzer, Jan-David Quesel, and Philipp Rümmer. Real world verification. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *LNCS*, pages 485–501, Berlin, 2009. Springer. ISBN 978-3-642-02958-5. doi: 10.1007/978-3-642-02959-2_35. 1.2, 1.2, 3, 4, 1.2, 2.4

[73] T.V.H. Prathamesh. Tensor product of matrices. *Archive of Formal Proofs*, January 2016. ISSN 2150-914x. `https://isa-afp.org/entries/Matrix_Tensor.html`,

Formal proof development. 4

[74] Stefan Ratschan and Jan-Georg Smaus. Verification-integrated falsification of non-deterministic hybrid systems. In Christos G. Cassandras, Alessandro Giua, Carla Seatzu, and Janan Zaytoon, editors, *ADHS*, volume 39 of *IFAC Proceedings Volumes*, pages 371–376. Elsevier, 2006. doi: 10.3182/20060607-3-IT-3902.00068. 2.1

[75] James Renegar. On the computational complexity and geometry of the first-order theory of the reals, part III: Quantifier elimination. *J. Symb. Comput.*, 13(3):329–352, 1992. doi: 10.1016/S0747-7171(10)80005-7. 1.1, 1.2, 5, 3, 3.1, 3, 2, 3, 4, 4.1.1, 2, 4.1.2, 8, 4.2.1, 4.2.2, 4.4

[76] Matias Scharager, Katherine Cordwell, Stefan Mitsch, and André Platzer. Verified quadratic virtual substitution for real arithmetic. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *FM*, volume 13047 of *LNCS*, pages 200–217. Springer, 2021. doi: 10.1007/978-3-030-90870-6_11. 1.2, 1.2, 2, 4.1.3, 4.2, 4.2.3, 4.2.4, 4.3, 4.4

[77] Matias Scharager, Katherine Cordwell, Stefan Mitsch, and André Platzer. Verified quadratic virtual substitution for real arithmetic: Benchmark examples and scripts. Zenodo, 2021. 2.4

[78] Matias Scharager, Katherine Cordwell, Stefan Mitsch, and André Platzer. Verified quadratic virtual substitution for real arithmetic. *Archive of Formal Proofs*, August 2021. `https://www.isa-afp.org/entries/Virtual_Substitution.html`, Formal proof development. 2, 4.2.3

[79] Andrew Sogokon, Stefan Mitsch, Yong Kiam Tan, Katherine Cordwell, and André Platzer. Pegasus: Sound continuous invariant generation. *Form. Methods Syst. Des.*, 2021. ISSN 0925-9856. doi: 10.1007/s10703-020-00355-z. Special issue for selected papers from FM'19. 1

[80] Christian Sternagel and René Thiemann. Executable multivariate polynomials. *Archive of Formal Proofs*, August 2010. `https://www.isa-afp.org/entries/Polynomials.html`, Formal proof development. 2.3.1, 2.3.4, A.2.1

[81] Christian Sternagel and René Thiemann. Executable matrix operations on matrices of arbitrary dimensions. *Archive of Formal Proofs*, June 2010. ISSN 2150-914x. `https://isa-afp.org/entries/Matrix.html`, Formal proof development. 3.2.2

[82] Adam Strzeboński. Solving algebraic inequalities. *The Mathematica Journal*, 7(4): 525–541, 2000. 1, 1.2

[83] Thomas Sturm. A survey of some methods for real quantifier elimination, decision, and satisfiability and their applications. *Math. Comput. Sci.*, 11(3-4):483–502, 2017. doi: 10.1007/s11786-017-0319-z. 1

[84] Thomas Sturm. Thirty years of virtual substitution: Foundations, techniques, applications. In Manuel Kauers, Alexey Ovchinnikov, and Éric Schost, editors, *ISSAC*, pages 11–16. ACM, 2018. doi: 10.1145/3208976.3209030. 2

[85] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. RAND Corporation, Santa Monica, CA, 1951. 1, 1.1, 1.2, 3.1.3, 4.4

[86] René Thiemann and Akihisa Yamada. Formalizing Jordan normal forms in Isabelle/HOL.

In Jeremy Avigad and Adam Chlipala, editors, *CPP*, pages 88–99. ACM, 2016. doi: 10.1145/2854065.2854073. 3.2.2

[87] René Thiemann and Akihisa Yamada. Matrices, Jordan normal forms, and spectral radius theory. *Archive of Formal Proofs*, August 2015. ISSN 2150-914x. `https://isa-afp.org/entries/Jordan_Normal_Form.html`, Formal proof development. 3.2, 3.2.2, 4, 3.2.2, 5

[88] René Thiemann, Akihisa Yamada, and Sebastiaan Joosten. Algebraic numbers in Isabelle/HOL. *Archive of Formal Proofs*, December 2015. ISSN 2150-914x. `https://isa-afp.org/entries/Algebraic_Numbers.html`, Formal proof development. 3.2, (3)

[89] Zak Tonks. A poly-algorithmic quantifier elimination package in maple. In Jürgen Gerhard and Ilias S. Kotsireas, editors, *Maple in Mathematics Education and Research - Third Maple Conference, MC 2019, Waterloo, Ontario, Canada, October 15-17, 2019, Proceedings*, volume 1125 of *Communications in Computer and Information Science*, pages 171–186. Springer, 2019. doi: 10.1007/978-3-030-41258-6_13. 1.1

[90] Joachim von zur Gathen. Parallel algorithms for algebraic problems. *SIAM J. Comput.*, 13 (4):802–824, 1984. doi: 10.1137/0213050. 5

[91] Volker Weispfenning. The complexity of linear problems in fields. *J. Symb. Comput.*, 5 (1-2):3–27, 1988. doi: 10.1016/S0747-7171(88)80003-8. 1, 1.1, 2

[92] Volker Weispfenning. Quantifier elimination for real algebra - the cubic case. In Malcolm A. H. MacCallum, editor, *ISSAC*, pages 258–263. ACM, 1994. doi: 10.1145/190347.190425. 2

[93] Volker Weispfenning. Quantifier elimination for real algebra - the quadratic case and beyond. *Appl. Algebra Eng. Commun. Comput.*, 8(2):85–101, 1997. doi: 10.1007/s002000050055. 1.1, 2

[94] Makarius Wenzel. Structured induction proofs in Isabelle/Isar. In Jonathan M. Borwein and William M. Farmer, editors, *MKM*, volume 4108 of *LNCS*, pages 17–30. Springer, 2006. doi: 10.1007/11812289_3. 4.2.2

[95] Wolfram. Real polynomial systems. Wolfram Language & System Documentation Center, 2023. https://reference.wolfram.com/language/tutorial/RealPolynomialSystems.html. 1.1

[96] Hitoshi Yanami and Hirokazu Anai. Synrac: a maple toolbox for solving real algebraic constraints. *ACM Commun. Comput. Algebra*, 41(3):112–113, 2007. doi: 10.1145/1358190.1358205. 1.1