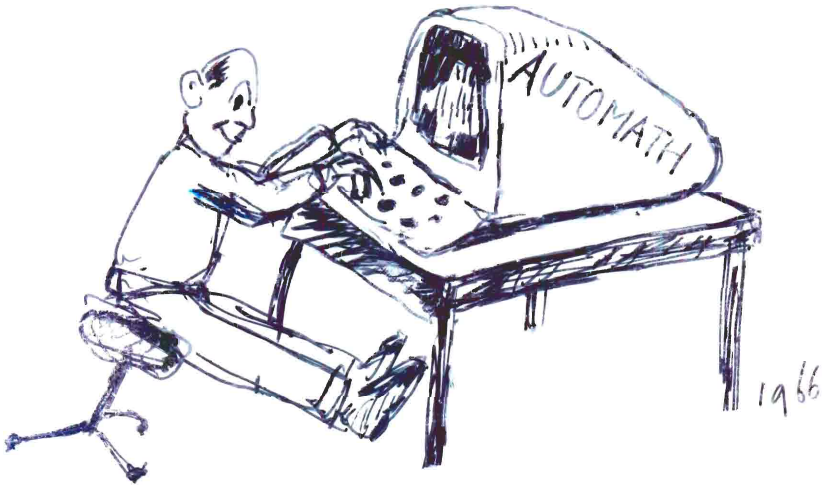
Two Tools for Formalizing Mathematical Proofs

Robert Y. Lewis

February 12, 2018

DREAM OF THE SIXTIES

INTERACTIVE PRODUCTION AND VERIFICATION OF MATHEMATICS



NOW BEGINNING TO COME TRUE

ALL BY THEMSELVES, THE CREATIVE POWER OF COMPUTERS IS LIMITED (JUST LIKE OURS)

A2.2

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Pure and Applied Logic
Department of Philosophy
Carnegie Mellon University
Pittsburgh, PA, USA

Dissertation committee:
Jeremy Avigad: Carnegie Mellon University, Department of Philosophy (committee chair)
Steve Awodey: Carnegie Mellon University, Department of Philosophy
Thomas Hales: University of Pittsburgh, Department of Mathematics
André Platzer: Carnegie Mellon University, Department of Computer Science

The images on the previous page and on page 80 are credited to N. G. de Bruijn, *Memories of the Automath project*, 2003.
http://www.win.tue.nl/automath/about_automath/mkm-compleet-klein.pdf

Acknowledgements

Thank you, Jeremy, for your years of advice, support, and patience. It's been a pleasure to be your student. I hope for many more years of collaboration in the future!

Thank you, Steve, Tom, and André, for all your guidance as part of my dissertation committee. Thanks also to all of those who have offered help and advice on various parts of this project: Jasmin, Minchao, José, Michael, James, Wilfried, Cody, and many others.

Thank you to the Lean development group, for designing a proof assistant good enough to make this dissertation possible. In particular, thank you Leo, for the incredible amount of energy that you've put into this project; it's hard to overstate how much your work is appreciated. Thanks also to the Lean group at CMU, past and present—Floris, Mario, Johannes, and everyone else—with whom countless hours of discussion have inspired the work in this document.

Thank you to my friends in Pittsburgh, who have gotten me through the last five years: for a very non–exhaustive list, Liam, Helga, Michal, Kasey, Aidan, Dan, Ruben, Mate. And thank you to my family, who have gotten me through many more than just the last five years.

Contents

Chapter 1

Introduction

Computers are ubiquitous in modern life. Their presence can be felt almost everywhere, including in the field of mathematics. Once restricted to using little more than a pencil and paper, mathematicians now use computers in many essential ways. Among other tasks, computers can help to visualize problems, test the plausibility of conjectures, search for examples and counterexamples, and answer arithmetic or computational questions. Even the language of mathematics has been digitized: researchers have developed software suites that will take a proof, written in the appropriate format, and check it for correctness according to some logical foundation. These software suites are often accompanied by tools for filling in parts of incomplete proofs automatically. One can imagine a future where mathematicians can state potential theorems to their computers, which will then search for proofs and counterexamples.

Not surprisingly, this introduction of mechanized methods to a resolutely human field has faced criticism. Both philosophers and mathematicians have questioned the importance of formalization projects, the epistemic gains of computer-aided proofs, and whether such proofs can be trusted at all. This skepticism is perhaps most intensely directed toward proofs that are *inherently* computational, in that they depend on results found by computers that cannot in practice be verified manually. Two (in)famous examples of this, Appel and Haken's 1976 proof of the four color theorem [6] and Hales' 1998 proof of the Kepler conjecture [73], raised concerns about the correctness of their computational results.

Verified computations—formal proof objects that certify the result of a computer's calculations—provide a way to satisfy some of these skeptics. Hales responded to the controversy surrounding his proof by launching a project to formally verify his proof. This project, named Flyspeck, was completed in 2014. Appel and Haken's result has also been formally verified. Such projects do not address all of the philosophical challenges to computers in mathematics, but they ensure the soundness of the results, resolving questions of trust.

Computational and formal mathematics, it seems, are here to stay, and while both paradigms differ from "traditional" mathematics, each has its own upsides. Given that there is desire and

need to verify mathematical arguments, it seems well advised to make this verification as easy and painless as possible. "Painless" is not an adjective that historically has been applied to formalization efforts. Most reports note the difficulty of formalizing even small components of arguments [13], and large scale projects like Flyspeck take years or decades to complete. Furthermore, much research in verification tools aims to solve problems in computer science, not in mathematics. While there is significant overlap, these tools do not always perform as well as one might hope in deeper, more heterogeneous settings.

This dissertation presents two tools that aim to ease the pain—at least slightly—of formalizing mathematics. By providing access to tools that mathematicians are accustomed to using, and making some "obvious" derivations obvious to proof checkers, we will be one step closer to practical formalization for the working mathematician. These tools are of use in problems from computer science and industry as well, just as tools aimed at those fields can be applied to mathematics.

There are more general themes that connect these projects. The first is *partial formalization*. Despite the best efforts of many researchers, the challenges of formalizing mathematics prevent most mathematicians from even trying. With a larger user base developing deeper and more varied libraries and tools, this barrier to entry will quickly lower. The question, then, is how to overcome the initial impasse. Perhaps due to cultural reasons within the field, most formalization work is currently gap-free: no unjustified steps or gaps in reasoning are allowed, no matter how small. This demand is in stark contrast to informal mathematics, and causes many of the challenges of formalization. One possible approach is to lessen the requirements of rigor, to allow gaps in proofs or even to trust external tools like computer algebra systems. Hales' formal abstracts project [72] goes so far as to omit formalized *proofs* entirely, focusing only on formalized definitions and theorem statements.

Both projects discussed in this dissertation separate the processes of *computation* from the processes of *verification*. Whether for theoretical reasons or for the sake of efficiency, users of these tools can opt to skip the verification steps and trust that the tools have done their jobs properly. Doing so does indeed reduce the level of certainty of the argument that is produced, and some might argue that doing so defeats the purpose of formalizing mathematics. But there are many benefits to formalization—among them, precisely specified claims and a searchable, parsable library—that do not depend on having gap-free proofs. In addition, if allowing partial formalizations increases the rate at which others can produce (full and partial) formalizations, it seems beneficial to the field as a whole to allow this concession at times.

The second theme that connects these projects is the formalization of mathematical *process*. Philosophers of mathematics have noted that there is more to mathematics than the raw output of its practitioners: the social environment, methods of reasoning and understanding, and metamathematical concepts are vital components of the field [80] [8] [98]. Nonetheless, formal mathematics libraries are, to a large extent, bare libraries of definitions, theorems, and proofs. Algorithms to

help with producing these proofs are implemented in different languages, often invisible to users, and can be difficult or impossible to extend to new theories.

The Lean *metaprogramming* framework allows methods for mathematical reasoning to be expressed in the same language and environment as the mathematics itself. Domain-specific proof search procedures can be developed in line with the theories that they act on. For one example, consider a development of basic undergraduate calculus. The heuristics that are taught for computing derivatives and integrals are simple examples of mathematical reasoning, that can be expressed as short metaprograms. Immediately after proving the product rule, chain rule, and so on, a Lean user can write a script for computing derivatives that applies these rules, and the script will exist alongside the definitions and theorems as a first-class member of the theory. The tools described in this dissertation, which are both implemented in Lean's metaprogramming framework, are designed to be adaptable to new formal developments. The rules and techniques needed to integrate these new developments exist as essential parts of the developments. This indicates a slight but important shift in the understanding of formalized mathematics: *a library is not a standalone tower of definitions, theorems, and proofs, but also includes metamathematical information, processes, and rules for how the mathematical objects interact with external systems.*

Outline. The following sections of this chapter provide general background about computation in, and verification of, mathematical proofs. Chapter 2 provides details about the Lean theorem prover, in which the tools described in the rest of this document are implemented. Chapter 3 describes a system for verifying systems of nonlinear inequalities over $\mathbb{R}$, and Chapter 4 describes an extensible link between Lean and the computer algebra system Mathematica. Chapter 5 presents some concluding thoughts.

1.1 Interactive theorem proving

To verify mathematical arguments by computer, the arguments must be be written so that machines can interpret them. We refer to the practice of writing such proofs as *interactive theorem proving*, and to the systems that check such proofs as *proof assistants* or sometimes *interactive theorem provers*.

A proof assistant must have a well-specified input language for mathematical content. That is, the user must be able to

- define mathematical objects, such as the set of natural numbers $\mathbb{N}$, the particular natural number 14, or the standard topological structure on $\mathbb{R}$;
- state properties of these objects, such as the fact that 14 is positive or that addition on $\mathbb{R}$ is continuous in each argument; and

- justify or prove these statements.

In order for the third point to have meaning, the definitions, theorems, and proofs must be connected by a *logic* which describes when a proof is correct. For the proof assistant to be usable, it should be possible—at least most of the time, in practice—to computationally check whether a proof is correct according to the given logic.

It is convenient, although not strictly necessary, for a proof assistant to automatically generate parts of proofs for the user. This can range from filling in implicit information to executing complex proof search programs. Existing proof assistants vary wildly in the level of automation that they provide, and pushing forward the limits of what can be automated is a very active area of research.

Two early proof assistants, Automath and Mizar, were developed in the late 1960s and early 1970s. Automath, whose genesis dates roughly to 1967, was a system based on dependent type theory developed by De Bruijn [1]. Its influence is clearly seen in more modern systems that use very similar foundations. Independently, Trybulec proposed (in 1973) and implemented (in 1975) Mizar, using Tarski–Groethendieck set theory as a foundation [101]. Mizar's language and capacities expanded over the following decades, and the system is still in use today. The Mizar Mathematical Library, a curated collection of Mizar proofs, is among the largest unified collection of formal proofs in existence [135].

Automath and Mizar are not only remarkable due to their age (or in the case of Mizar, its longevity). Both systems were designed by mathematicians for the express purpose of formalizing mathematics. De Bruijn made this motivation explicit [49]:

> AUTOMATH is a language intended for expressing detailed mathematical thoughts. It is *not* a programming language, although it has several features in common with existing programming languages. It is defined by a grammar, and every text written according to its rules is claimed to correspond to correct mathematics. It can be used to express a large part of mathematics, and admits many ways for laying the foundations.

A zoo of systems appeared in the following decades, and while many users were interested in these systems for their mathematical applications [3], the growing field of computer science exerted more and more influence. Some spiritual descendants of Automath are seen as programming languages first and verification tools second, and many primarily target the verification of software.

To describe this zoo of systems, it is helpful to classify them based on their logical foundations.

Higher order logic. The systems in the HOL family [67] are built on a foundation of simply-typed higher order logic. This family is a descendant of the Edinburgh LCF (logic for computable functions) program [66], extending the ideas found there to a richer theory. The language used in these systems is closely connected to their implementation language ML. Harrison's variant, HOL Light, is implemented in OCaml and is noted for the small size of its trusted kernel [77]. The HOL

Light standard library contains deep developments in analysis, and the system is used for industrial verification of hardware.

Isabelle, which also follows the LCF approach, is a generic proof assistant which can be instantiated to any number of logics [110]. Its most popular instantiation, Isabelle/HOL, uses a foundation very similar to that of the HOL family. With features including type classes [69], various tools for proof search [114], and a wide range of (co)datatypes [30], it supports the Archive of Formal Proofs, a large online database of formalization projects. While this archive delves into many fields of mathematics, it is heavily biased toward computer science [29].

The logic used by these systems is surprisingly expressive, given the simplicity of its foundations. It supports powerful and efficient automation for proofs, and is a natural environment in which to formalize topics like real analysis and number theory. However, the inherent restrictions on the types that it can express can lead to difficulties when reasoning with abstract algebraic structures [13] [24].

Intuitionistic type theory. Many of the ideas seen in Automath became clearer as the study of type theories progressed [108]. Martin-Löf's intuitionistic type theory [100] had an enormous influence on the next generation of proof assistants. The Calculus of Constructions [44] can be seen as a refinement of De Bruijn's and Martin-Löf's logics, and the Calculus of Inductive Constructions [45] extended this logic with a general schema for declaring datatypes. The CIC and its variants have proven to be extremely expressive and high-powered languages for both programming and formal verification.

Perhaps the most well-known of these systems is Coq [25], a project launched in 1984 by Coquand and Huet. While Coq does not have an "official" repository as extensive as Isabelle's AFP, there are a number of libraries, including the Mathematical Components library [96]. This library was the basis of the formal proofs of the Four Color Theorem [64] and the Odd Order Theorem [65].

While the foundations of Agda [111] are quite similar to those of Coq, its proof language is quite different: Agda proofs are declarative, while Coq proofs are largely based on tactics. Agda is often described equally as a programming language and a proof assistant. Idris [33] and Matita [7] also fall into this category. Some of these proof assistants support homotopy type theory [130], a variant of the CIC that is particularly convenient for formalizing algebraic topology [132].

The CIC is a more expressive language than the ones found in HOL-based systems, and it is well suited for reasoning with abstract structures. Since it is often implemented constructively, terms in dependent type theory often have computational interpretations, and this can be used to great effect. These features come at a cost, however. Due to the complexity of the logic, many algorithms for computation and proof search are inefficient or difficult to implement in a dependently typed setting [125].

The tools described in this dissertation are implemented in Lean, another proof assistant based

on intuitionistic type theory. Chapter 2 provides a more detailed description of the particular foundations of this system.

Set theory. Systems based on set theory often appeal to mathematicians who have been raised (perhaps only implicitly) in Zermelo–Fraenkel set theory with the axiom of choice. The simplicity and elegance of ZFC as a foundation for mathematics is undeniable, and a century of scrutiny has led all but the most skeptical to believe in its consistency. As a practical foundation for a proof assistant, however, set theory encounters some difficulties. Since terms do not contain information about "what they are" in their syntax—all terms are sets, and are elements of infinitely many other sets—such information must be tracked propositionally, and doing so can become quite intricate. Even Mizar supplements its set theoretic foundation with a system of weak types that makes it possible to manage the ensuing complications.

Metamath [104], developed in 1992 by Megill, is in fact logic-agnostic at its core, and can be thought of as a string rewriting system. The official and largest Metamath library implements ZFC.

Isabelle, another system with a core that is not specific to any logic, has been given multiple ZF(C) instantiations. The first, by Paulson, has been used primarily to verify proofs about set theory itself [113]. A second version, by Zhan, has attacked more general mathematical proofs [140]. Most developments in Zhan's Isabelle/ZFC use the Auto2 proof search tool [139]. With the exception of Auto2, automation in set theoretic systems has historically lagged behind that in HOL- and DTT-based systems [131].

Other foundations. PVS [126] is a system based on classical dependent type theory, used primarily in software verification. Its rich type system generates numerous proof obligations during typechecking, that are often dischargable with automation. ACL2 [91], also used widely in industry, is based on primitive recursive arithmetic. This is a rather weak logic, but as a consequence, most proofs are able to be automated; in fact, the primary mode of interaction with ACL2 is to simply assert theorems and ask the system to prove them.

Finally, there is a range of domain-specific provers that do not target general mathematics. One example, KeYmaera [117] [61], generates and checks proofs in differential dynamic logic [116]. It is used as a tool for verifying specifications of hybrid physical systems, which combine continuous and discrete behavior. KeYmaera depends on tools for solving problems in real arithmetic, as described in Section

1.2 Mathematical computation

To specify precisely what a "mathematical computation" is would be a major task. Even a general overview of computational techniques in mathematics would expand far beyond the confines of this

document. Here, we summarize some particular examples of mathematical computation that are relevant to the projects in Chapters 3 and 4, and to the general themes of this dissertation.

1.2.1 Inequalities over real closed fields

Chapter 3 describes a tool that symbolically verifies inequalities over the real numbers $\mathbb{R}$. This section describes the background motivating our approach and related methods in the literature.

Let $\mathbb{T}_{RCF}$ denote the first-order theory of $\mathbb{R}$ under the operations $+$ and $\cdot$, with constants 0 and 1 and relations $=$ and $<$. (Note that $-$, $\leq$, $>$, and $\geq$ are easily definable using first-order formulae.) This is known as the theory of *real closed fields*; examples of real closed fields include $\mathbb{R}$, the real algebraic numbers, and the hyperreal numbers. A number of equivalent necessary and sufficient conditions exist to establish that a given field F models $\mathbb{T}_{RCF}$ [38]. Among others:

- There is a total order on F such that each element $x \in F$ with $x > 0$ has a square root y such that $y \cdot y = x$, and every polynomial in $F[x]$ of odd degree has at least one root.
- There is a total order on F such that the intermediate value theorem holds for all polynomials in $F[x]$.
- F is not algebraically closed, but its field extension $F(\sqrt{-1})$ is algebraically closed.

Alfred Tarski discovered in the 1930s that $\mathbb{T}_{RCF}$ admits quantifier elimination, although he did not publish a proof until 1948 [129]. An immediate consequence of this observation is that $\mathbb{T}_{RCF}$ is decidable. His proof involves defining a succession of formulae that are used to transform a given sentence in the language of RCF into an equivalent quantifier-free sentence. Once this new sentence has been derived, one can apply a simple technique for deciding the truth of literals including only constants to decide the original sentence.

Tarski's procedure generalizes a technique from Sturm for counting roots of a polynomial in one variable. He describes how to construct formulae representing the nth derivative of a polynomial α in variable ξ and the statement that ξ is a root of α of order n. Combining these formulae, he obtains more complicated sentences $G^n_\xi(\alpha, \beta)$ that assert relations between the number and orders of roots of polynomials α and β. These sentences G are shown to have an equivalent quantifier-free form by a complex process that repeatedly divides α by β. Finally, arbitrary sentences are reduced to combinations of sentences of form G. Alongside an algorithm for deciding the satisfiability of quantifier-free sentences in this language, this process amounts to a decision procedure for $\mathbb{T}_{RCF}$.

In addition to being rather arcane, Tarski's proof is in effect a purely theoretical result. While he gives a (more or less) explicit algorithm for deciding $\mathbb{T}_{RCF}$, the algorithm can be shown [107] to have nonelementary complexity—that is, no tower function $2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}$ bounds the algorithm's run time for all n, where n is the number of quantifiers. In fact, Davenport and Heintz [48] show that the quantifier elimination problem is necessarily at least doubly exponential in n. Nonetheless, Tarski

is optimistic in his monograph [129] that his algorithm will be implemented by "machines," and refers frequently to the potential use of these machines in mathematical research. He further holds out hope that his procedure may be extended to include, for example, exponentiation.

Collins' cylindrical algebraic decomposition method [42] [21] realizes a doubly exponential bound on this decision problem. Given a set of polynomial inequalities $\mathcal{F} = \{f_i(\bar{x}) > 0 \mid f_i : \mathbb{R}^n \to \mathbb{R}\}$, one can define the notion of a *cell decomposition* for $\mathcal{F}$, a partition of $\mathbb{R}^n$ on each element of which the sign of f_i is constant. Collins develops a process for projecting $\mathcal{F}$ to lower dimensions, alongside a lifting technique for cells. When $n = 1$, the cell decomposition can easily be found, and then iteratively lifted up to $\mathbb{R}^n$. Checking the satisfiability of a quantified conjunction $\bigwedge \mathcal{F}$ is then reduced to a similarly quantified sentence about the n-dimensional cell structure, which is finite. While this technique significantly improves on Tarski's algorithm, and can in practice solve many problems, one can describe fairly simple problems on which its performance is impractically slow.

Various other techniques for deciding $\mathbb{T}_{RCF}$ have been discovered, including ones by Seidenberg [124], Cohen [40], Hörmander [84], and Ben-Or, Kozen, and Reif [23]. These algorithms have varying strengths and weaknesses, but all share a worst-case complexity that makes them impractical for large problems.

All of these methods decide problems over the full theory of real closed fields, including both universal and existential quantifiers, but similar simpler theories can be interesting on their own. Basu and Roy [21] give a singly exponential time algorithm to decide problems in the existential fragment of $\mathbb{T}_{RCF}$. Gröbner basis methods can be efficient for many problems in the universal fragment of arithmetic involving equalities, but do not extend well to inequalities [78]. And techniques of varying efficiency and completeness are known for the additive and multiplicative fragments of $\mathbb{T}_{RCF}$, arithmetic over the integers, and various other domains.

In [12], Avigad and Friedman investigate the practicality of combining decision procedures for the additive and multiplicative fragments of $\mathbb{T}_{RCF}$. Their investigation is partly motivated by the observation that the simplicity of these subtheories is somewhat incongruous with the complexity of $\mathbb{T}_{RCF}$. Since combining the subtheories does not produce the entirety of $\mathbb{T}_{RCF}$, one might hope that this combination has an easier decision problem. Avigad and Friedman show that the universal fragment of this combination is decidable (although the existential fragment of $\mathbb{T}[\mathbb{Q}]$ embeds Hilbert's 10th Problem over $\mathbb{Q}$, which is conjectured to be undecidable). Their argument reduces the decision procedure to that of $\mathbb{T}_{RCF}$, though, and so does not reduce complexity. They conclude by discussing "pragmatic" procedures that approximate their decidability results; this discussion motivates the project described in Chapter 3.

Related methods approach similar problems by using numerical techniques. These methods include those based on interval arithmetic [55] [105] and Taylor models [37]. Libraries exist for verified numeric computations over $\mathbb{R}$ [88], and have been used in formalization projects [87]. These methods differ from the project described in this dissertation in scope and applications, so we do

not describe them here. Nonetheless, they form an important part of the landscape of real number computation.

1.2.2 Computer algebra

Implementations of the algorithms described in the previous heading often fall under the general umbrella of *computer algebra* or *symbolic computation*. Computers were used for numerical computations since their inception, and the need to handle exact computation soon became apparent [63]. Experimental systems began to appear in the 60s, including ALTRAN [75], MATHLAB [58], and Reduce [20]. These systems provided tools that would look familiar to modern computer algebra users: they were able to manipulate polynomials and rational functions, heuristically derive and integrate, and (attempt to) display results using natural formatting, and many could be used in an interactive fashion. Such systems were claimed to perform "about as well as a good calculus student" [63, Section 1.3], and were adopted by physicists. Macsyma [99], released in 1971, quickly became one of the most powerful and popular of these early programs. Also appearing, alongside these general-purpose computation systems, were tools for specific calculations such as tensor component manipulation [60] and computing Poisson series [89].

The 1980s and 1990s saw a push toward efficiency and portability. Maple [39] was designed with a small, efficient kernel written in C, and a library around this kernel written in the interpreted Maple language. This approach was also seen in Wolfram's SMP [41] and Mathematica [137] systems, the latter of which was noteworthy for its graphical capabilities and user interface. Axiom, a successor to IBM's Scratchpad, was a strongly typed system able to compute over abstract algebraic structures [47]. Systems like GAP [62] and Cayley, which later became Magma [32], were designed for special-purpose computations in group theory.

Development on these systems and interfaces has continued into modern years, with increasing emphasis on ease of use [134]. Sage [128] packages many common open-source tools into a unified Python interface. Mathematica has expanded to include numerous tools for processing and visualizing data [138]. Tools for geometric arguments, including Geogebra [83], have been widely adopted in mathematics education.

The range of applications of computer algebra is vast, and it is difficult to describe these systems in general. For the most part, computer algebra systems and proof assistants contrast in how they approach *justification*. The former are tools for evaluating quantities, and while the algorithms they implement are based on sound mathematical theory, they do not typically explain or certify their results, beyond the explanations inferable from the correctness of the algorithms. They exist to inform the mathematician (or, commonly, the physicist or engineer) who queries them, and to aid in exploration, but do not help in the development of arguments. Proof assistants serve an almost dual purpose: they aid less in the discovery of concrete results, but more in the expression of abstract mathematical reasoning.

1.2.3 Verified computation

Hales' *Flyspeck* project [109] [112] [70] [71] provides an illustrative case study of automation in formalized mathematics. In 1998, Hales and Ferguson announced a proof of the Kepler conjecture, which states that the densest way to pack congruent three-dimensional spheres is the "obvious" face-centered packing. Their formidable proof relied on a large number of computations performed by an unverified C++ program, to check that certain nonlinear inequality constraints are unsatisfiable. After referees announced that they could not be 100% confident in the correctness of the proof, Hales started the Flyspeck project in 2003 to formally verify the theorem. Hales announced the completion of Flyspeck in August 2014. The project ultimately consisted of two parts. The first, corresponding to the "traditional" portion of Hales' original proof, was a formal proof in HOL Light that classified all tame graphs. The second, computational, part of the proof was verified in three pieces, two in HOL Light and one in Isabelle. The statement of the result formalized in Isabelle was assumed as an axiom in HOL Light, and the two parts were combined to create one "unified" proof.

Computation played an essential role in both the original proof of Kepler's conjecture and in the verification; confirming the inequality constraints by hand would be a hopeless task for any mathematician. Efficiently verifying the calculations performed by the computer proved to be extremely difficult, and the Isabelle formalization was completed much more quickly [74]. Traditional techniques for algorithmically deciding nonlinear inequalities were far too inefficient to be of use, and so the Flyspeck group had to develop specialized methods [127].

While rarely seen on the same scale as in the Flyspeck project, problems of verifying and refuting inequalities on $\mathbb{R}$ are found often across fields of mathematics. Automated methods to solve these problems that interact smoothly with proof assistants can be extremely desirable. McLaughlin and Harrison [103] describe a proof-producing implementation of Hörmander's technique, a less efficient but simpler analogue to CAD. As it is difficult to separate the "search" stage of this algorithm from the "proof" stage, almost every part of their implementation is forced to be proof-producing. Some general lemmas help to mitigate this somewhat, but their method ultimately is extremely expensive to run. Proving the sentence $(\forall x)(-1 \leq x \leq 1 \rightarrow -1 \leq 4x^3 - 3x \leq 1)$ takes over a minute on a standard desktop.

Verified computational proofs have appeared in other areas of math as well. Many SAT solvers will generate certificates of their computations, and these certificates can be verified in proof assistants [46] [81] [93]. SAT solvers have had success in solving problems in some areas of combinatorics [82], and despite the extreme size of the generated certificates, they can be checked.

Approaches to verified automation differ in the extent to which they compute "inside" the proof assistant. Some algorithms can be written in the object language of a proof assistant and proved to be correct; the algorithm can then be executed in the proof assistant kernel, or sometimes extracted to another language. The former provides a great deal of certainty, but can be extremely inefficient,

and the latter requires one to trust the code extraction process. Alternatively, as in McLaughin and Harrison's implementation of Hörmander's algorithm, the search procedure can build a proof term as it runs; this adds a lot of complexity to the process and can lead to extremely large proofs. A third approach, modeled by the verification of SAT certificates, is to offload the computation to external tools and to check or reconstruct results post hoc. Hammer systems [114] have used this approach to great success, particularly for proving smaller lemmas and subgoals in interactive proofs. These tools translate an ITP goal into first-order logic, along with a selection of relevant lemmas from the environment, and use efficient external solvers to search for a proof. If successful, they use information from that proof to reconstruct it using a less efficient, but proof-producing, internal solver.

None of these approaches is inherently better than the others; each is appropriate in some situations and inappropriate in others. The tools described in this dissertation largely follow the third approach, in that they keep the process of search separate from the process of verification. Doing so ties in well with the idea of partial formalization, in that users who trust the systems involved, or who aren't concerned about verification, may choose to skip the reconstruction steps for the sake of speed or flexibility.

Chapter 2

The Lean theorem prover

The tools described in this dissertation are implemented in and for the proof assistant Lean. Lean is principally developed by Leonardo de Moura at Microsoft Research [50]. The development of its standard library is led by Jeremy Avigad. Lean has been designed from the beginning to support strong automation. It aims to eventually straddle the line between an interactive theorem prover with powerful automation and an automated theorem prover with a verified code base and interactive mode. Its powerful *metaprogramming* framework [56] allows for complex programs that manipulate proofs and the proving environment to be written in the same language as the proofs themselves. This is achieved by exposing internal system datatypes and processes in the proof language; de Moura has described this approach as "whitebox automation."

In this chapter, we summarize the foundations and features of Lean, with emphasis on the components that will be used in the following chapters. More complete documentation of the system can be found in the system's publications [50] [56], documentation [9] [11], and website [2]. This chapter is not meant to stand as a first introduction to dependent type theory, of which there are many examples from many perspectives [108] [115] [130].

2.1 The Calculus of Inductive Constructions

Lean is based on the Calculus of Inductive Constructions (CIC) [44] [45], an extension of the typed lambda calculus. The CIC is dependently typed, with types given as terms of the language. In the following sections, we use the notation `T[a'/a]` to denote the substitution of `a'` for all occurrences of `a` in the term `T`. (Since bound variables will be indexed in practice, we assume no naming conflicts.) We write `t : T` to denote the judgment "term `t` has the type `T`," and read it "term `t` of type `T`."

There are two ways to define a type in the CIC. If `A` is a type and `T` is a type with variable `a : A` possibly occurring freely, then the pi type `Π a : A, T` is a type. A term of a pi type, `f : Π a : A, T` is a function which maps a term `a' : A` to a term of type `T[a'/a]`. The standard (non-dependent) function type, `A → T`, is a degenerate case of the pi type where `a` does not occur

in `T`. Types can also be defined inductively, by providing a list of their constructors. The empty type can be defined as an inductive type with no constructors; the unit type can be defined as an inductive type with one constructor, `unit.star : unit`; the natural numbers can be defined as an inductive type with two constructors, `nat.zero : nat` and `nat.succ : nat → nat`.

Every inductive type has an associated *recursor*, or *destructor*, a term which describes how to define a function mapping out of the type. For instance, to define `f : nat → int` using `nat.rec`, one must provide two arguments. The first, of type `nat`, is the value of `f` applied to `nat.zero`; the second, of type `nat → int → int`, is the value of `f` applied to `nat.succ n`, in terms of `n` and the value of `f` applied to `n`.

Terms of the CIC can take four forms:

- *Constants* are named, typed declarations. For instance, `nat.zero` is a constant with type `nat`. Every constant has a single unique type, up to convertibility. When one declares an inductive type, the constructors and the recursor can be thought of as constants.
- *Lambda abstractions* inhabit function types. If `t : T` is a term in which a variable `a : A` possibly appears, then `λ a : A, t` is a term with type `Π a : A, T`.
- *Applications* are terms formed by applying a term of function type to an argument. These are denoted by writing the function and the argument successively, as in `f a`.
- *Type universes* are the types of terms that appear on the right hand side of typing judgments. Intuitively, since types are terms of our language, and every term must have a type, every type must have a type as well; type universes are these "types of types." The structure of type universes varies between implementations of the calculus of constructions. To consistently include general inductive types, the CIC provides a hierarchy of type universes `Sort u`, `u` $\geq$ `0`, such that `Sort u : Sort (u+1)`. The rules for computing the universe level of inductively defined types are subtle and discussed in [45] and [50].

Terms of the CIC have a computational interpretation. There are *reduction rules* that (ideally) define an equivalence relation on the set of terms. The application of a pi or lambda expression to a term of the abstracted type reduces to a substitution: that is, if `T : B` and a' : A, then `(λ a : A, T) a'` reduces to `T[a'/a]`. Two terms that are equivalent under this notion of reduction are said to be *definitionally equal.* This sense of equality is distinct from the internal notion of equality defined below.

In addition, the application of the recursor for an inductive type `I` to a closed term `i : I` reduces to the application to `i` of the corresponding case of the recursor. If `f : nat → nat` is defined using `nat.rec`, as above, with arguments `k : nat` and `λ m v, t`, then `f nat.zero` reduces to `k` and `f (nat.succ n)` reduces to `t[n/m][(f n)/v]`.

Following the Curry–Howard correspondence [85], propositions can be expressed in the same language as datatypes. One can think of a term `P : Type` as a proposition, and a term `p : P` as a

```
meta inductive expr (elab : bool)
| var          : nat → expr
| sort         : level → expr
| const        : name → list level → expr
| mvar         : name → expr → expr
| local_const : name → name → binder_info → expr → expr
| app          : expr → expr → expr
| lam          : name → binder_info → expr → expr → expr
| pi           : name → binder_info → expr → expr → expr
| elet         : name → expr → expr → expr → expr
| macro        : macro_def → list expr → expr
```

Figure 2.1: Lean expression kinds

proof of `P`. The conjunction of two propositions, `P ∧ Q : Type`, is the same as the product `P × Q`, since the proof of a conjunction is a pair of proofs of each conjunct. The proof of an implication from `P` to `Q` is a function that takes a proof of `P` to a proof of `Q`, so implications are represented as function types. A proof of a universally quantified proposition `∀ x : A, P x` is a dependently typed function which maps `x : A` to a proof of `P a`. Using inductive types, it is possible to represent all propositional connectives with the traditional introduction and elimination rules of intuitionistic logic.

Equality can also be defined in the CIC as a family of inductive types.

```
inductive eq {α : Sort u} (a : α) : α → Prop
| refl : eq a
```

Intuitively, this says that the "only" way to prove two terms of a type `α` are equal is if they are the same term (where "same" means "definitionally equal"). The recursion principle for this type is exactly the substitution principle for equality.

```
eq.rec : ∏ {α : Sort u} {a : α} {C : α → Sort v}, C a → ∏ {b : α}, a = b → C b
```

2.1.1 Lean-specific foundations

Lean's implementation of the CIC uses a non-cumulative hierarchy of type universes `Sort u`, with the notation `Prop := Sort 0`, `Type := Sort 1`, and `Type u := Sort (u+1)`. Terms of type `Sort u` can be lifted to higher universes, but there is no subtype relation between universe levels.

The Lean expression grammar is presented (in Lean syntax) in Figure 2.1; the preceding keyword `meta` is described below in Section 2.3.

Each Lean expression exists in an environment, which contains the names, types, and definitions of previous declarations. The `const` kind accesses a previous declaration, instantiated to particular universe levels if the declaration is parametric. In addition to constants in its environment, an expression may refer to its local context, which contains variables and hypotheses of kind `local_const`. These expressions do not appear in closed terms.

The expression kinds `lam` and `pi` respectively represent lambda abstraction and the dependent function type. (Non-dependent function types are degenerate cases of pi types.) Each contains a name for the bound variable, the type of the variable, and the expression body. Bound variables of kind `var` are anonymous within the body, being represented by De Bruijn indices [102]. Application of one expression to another is represented by the `app` kind.

Type universes are implemented by the expression kind `sort`. Metavariables represent placeholders in partially constructed expressions; the `mvar` kind holds the name and type of the placeholder. Let expressions (`elet`) bind a named variable with a type and value within a body. Macro expressions are used during parsing and, occasionally, to encapsulate computationally expensive expressions. They can be eliminated by expansion and need not occur in fully elaborated terms.

Pre-expressions, or unelaborated expressions, share the same grammar as `expr`. These correspond roughly to user input text, and can be turned into elaborated expressions by inferring implicit information.

Lean's first type universe, `Sort 0` (more commonly referred to as `Prop`), is impredicative: it is possible to define a proposition by quantifying over other propositions. The following examples show the difference between `Prop` and the (predicative) universe `Type 0`. The `#check` command prints the type of the subsequent expression. In the first line, we see that the type level has raised by one, while in the second, it stays the same.

```
#check Π T : Type 0, T -- Type 1
#check Π T : Prop, T   -- Prop
```

The impredicativity of `Prop` is very powerful, as it allows all logical combinations of propositions to be expressed in the same universe. However, to preserve consistency with classical logic, it is necessary to restrict the recursors of (most) `Prop`-valued types to eliminate only into `Prop`. One can show that assuming impredicativity and excluded middle (for any `P : Prop`, `P ∨ ¬P` is inhabited) implies proof irrelevance (for any `P : Prop`, `h1 : P`, and `h2 : P`, `h1 = h2` is inhabited). But proof irrelevance is inconsistent with so-called "large" elimination: given an inductive `P : Prop` with two constructors `h1` and `h2`, one could define a function `f : P → bool` such that `f h1 = tt` and `f h2 = ff`, while proof irrelevance implies `h1 = h2`.

In practice, most concrete, non-`Prop` data types live in `Sort 1`, so we abbreviate this universe as `Type`.

In addition to being impredicative, `Prop` is definitionally proof-irrelevant. For any proposition `P : Prop`, any two terms `p1 : P` and `p2 : P` are considered equivalent by the kernel. The proof-irrelevance of `Prop` means that `Prop` is computationally inert. The existence of a term `p` of type `P : Prop` signifies a proof of `P`, but nothing can be said about that proof, and it cannot be used to generate data. This observation allows the virtual machine (Section 2.3) to ignore proofs of propositions entirely.

The core logic of Lean is constructive; it cannot be used to derive certain principles of clas-

sical mathematical reasoning. Lean declares an additional axiom, `axiom choice (α : Sort u) : nonempty α → α`, which "creates" data from the proposition `nonempty α`. This axiom is used to prove the law of excluded middle, along with a more traditionally stated choice principle. However, terms which depend on this axiom lose some of the convenient normalization properties of the core CIC, and cannot be used for computation in the VM.

Lean also adds quotient types to the core CIC, with the following interface:

```
constant quot {α : Sort u} (r : α → α → Prop) : Sort u
constant quot.mk {α : Sort u} (r : α → α → Prop) (a : α) : quot r
constant quot.lift {α : Sort u} {r : α → α → Prop} {β : Sort v} (f : α → β) :
  (∀ a b : α, r a b → eq (f a) (f b)) → quot r → β
constant quot.ind {α : Sort u} {r : α → α → Prop} {β : quot r → Prop} :
  (∀ a : α, β (quot.mk r a)) → ∀ q : quot r, β q
```

Given a relation `r` on a type `α`, these constructors create a type where `r`-related elements of `α` have been identified. Intuitively, this type is the quotient of `α` by the reflexive, transitive, and symmetric closure of `r`. The constant `quot.mk` creates a term of the quotient type from a term of the underlying type, while the constant `quot.lift` defines a function on the quotient type from a function on the underlying type (assuming this function respects `r`).

These additional constants do not break computation in the VM. In fact, the VM can safely ignore applications of `quot.mk` and `quot.lift`, treating such terms as the object of the underlying type.

2.2 Proving in Lean

Since there will be many examples in the following chapters, we briefly summarize the syntax and features of Lean proofs here.

Implicit arguments. Some arguments in Lean declarations are marked as implicit via the use of curly braces. In the following code, the argument `α` can be inferred from the argument `a`. Marking it as implicit allows the user to write, e.g., `id 0` instead of `id ℕ 0` when applying the declaration.

```
def id {α : Type} (a : α) : α := a
```

Other forms of implicit arguments include optional parameters, for which default values are provided, and parameters which are synthesized at application time by a tactic. In the following declaration `add_val`, the second argument `k` is optional, and assigned to the value `0` when it is not provided.

```
def add_val (n : ℕ) (k : ℕ := 0) : ℕ := n + k
#eval add_val 5    -- 5
#eval add_val 5 2  -- 7
```

In `double_pos`, the second argument `h` should prove that the first argument `n` is positive. This proof is not provided by the user, but found automatically using the `assumption` tactic. If this tactic fails to find a proof, a failure message will be displayed.

```
theorem double_pos (n : ℕ) (h : n > 0 . tactic.assumption) : n + n > 0 :=
add_pos h h

example (n : ℕ) (n > 0) : n + n > 0 := double_pos n
```

Lean's elaborator is tasked with inserting implicit arguments to turn pre-expressions into expressions. Type class instances, described below, are another example of implicit arguments.

Structures. Lean provides convenient syntax for dealing with structures, which are inductive types with one constructor. The following declarations define equivalent types, but the latter allows for named projections and "structure notation."

```
inductive pair
| mk : ℕ → ℕ → pair

structure pair' :=
(lhs : ℕ) (rhs : ℕ)

def pr : pair' := { lhs := 2, rhs := 3 }
```

Structures allow optional and default parameters, and structure notation allows updating and extending previously defined structures.

```
structure triple extends pair' :=
(middle : ℕ)

def tp : triple :=
{ pr with middle := 10 }
```

Type class inference. Families of structures can be declared to be *type classes*.

```
class has_add (α : Type u) := (add : α → α → α)
```

Declarations whose types are type classes can be declared as *instances*.

```
instance nat_has_add : has_add nat := { add := nat.add }
```

In subsequent declarations, arguments whose types are type classes can be marked as implicit type class parameters. These are inferred by Lean's elaborator, which recursively tries to find instances of the type class in the environment to synthesize the desired arguments. In the following, `plus 1 2` is definitionally equal to `nat.add 1 2`.

```
def plus {α : Type} [has_add α] (a b : α) : α := a + b
#check plus 1 2 -- ℕ
```

Type classes are used extensively in Lean, notably to build the algebraic hierarchy. Structures such as groups, rings, and fields are defined as type classes. Through type class inference and coercions, theorems proved about abstract rings (for example) can be applied to any concrete structure which has been shown to be an instance of a field.

Pattern matching. Lean declarations which decompose a term of an inductive type are often defined using pattern matching, which provides nicer notation than using the built-in recursor terms. Unlike systems like Coq, which support pattern matching in the kernel, Lean compiles such declarations to terms built using recursors; it does so in a way that is largely invisible to the user.

```
def double : ℕ → ℕ
| 0 := 0
| (n+1) := double n + 2
```

The equation compiler supports dependent types, wild card arguments, and many other features.

Tactic mode. In addition to the "declarative" style of proving and defining, where terms are written explicitly by the user, Lean proofs can also be synthesized using *tactics*. Intuitively, a tactic can be thought of a script that tells Lean how to construct a term. For an example, suppose we are trying to prove `succ (succ (succ (succ k))) ≥ k` for `k : ℕ`. This proof can be completed via a chain of applications of `le_trans` and `le_succ`. Rather than writing this chain manually, we can tell Lean to put it together for us, with the script `repeat {transitivity, apply le_succ}`. This script will continue to work if we change the number of occurrences of `succ` in our goal.

More formally, a proof obligation `T : Type u` occurs in a setting with an environment and a collection of local hypotheses. The tactic state at this setting can be thought of as the structure containing this environment, these local hypotheses, and the goal, which is represented as a metavariable `m : T`. Tactics are programs which manipulate this state, possibly adding new goals or assigning old ones to values. If a tactic assigns `m` to a term which contains no metavariables, the proof obligation is satisfied by that term.

Common tactics include `assumption`, which searches the local hypotheses for a term proving the goal, and `simp`, which rewrites the goal using rules found in the environment. The syntax `by tac` uses a single tactic `tac` to close the current proof obligation; a `begin ... end` block applies a sequence of tactics.

```
example (a : ℕ) (h : a > 0) : a > 0 := by assumption
example (a : ℕ) (h : a > 0) : a + 0 > 0 :=
begin
  simp, assumption
end
```

The following section describes how users can write custom tactics in the language of Lean.

2.3 Metaprogramming in Lean

Terms in the CIC have a computational interpretation. If it does not refer to any undefined constants or extra axioms, a closed term `t` of an inductive type `T` will always reduce to an application of a particular constructor of `T`. It is possible to use Lean's kernel to reduce terms to this normal form, a process known as kernel evaluation. This process can be computationally expensive: among other reasons, definitions in Lean are often chosen for ease of proving rather than evaluation complexity. Addition on the natural numbers is defined using unary arithmetic, and this extends to all other numerical computations. Since the kernel is as small and simple as possible to preserve trust, it does little to optimize evaluation.

However, it is not always necessary to evaluate in a trusted manner. Lean includes a virtual machine which uses a variety of techniques to evaluate terms more efficiently than the kernel. The VM discards proofs of propositions, as these are not relevant to evaluating type-correct terms, and replaces terms of some common types such as `nat` with efficient C++ representations [56]. It is expected that evaluating a term in the VM will produce the same result as evaluating it in the kernel, but the correctness of a proof in Lean never relies on this fact. The kernel and VM are entirely independent, and only the former is used for type checking. Any term used to prove a theorem, no matter its origin, will be passed through the kernel before being accepted.

The notion of computation in the VM is somewhat more general than that of the kernel. There is no theoretical concern about nonterminating VM evaluation. Lean allows declarations to be marked as "VM-only," or "untrusted," with the keyword `meta`. Termination checking is disabled for these declarations, as in the following:

```
meta def f : ℕ → ℕ
| n := if n=1 then 1
       else if n%2=0 then f (n/2)
       else f (3*n + 1)
```

This definition would be rejected by Lean's kernel, and it would be unsound to allow arbitrary recursive calls, as one could define a (nonterminating) proof of `false`. Nevertheless, the function `f` can be evaluated in the VM, failing to terminate if it is applied to `0`. Meta declarations can reference non-meta declarations, but not vice versa. The meta language is thus an extension of the core Lean language.

A major contribution of de Moura et al. [56] is to allow metaprograms to manipulate Lean expressions and environments. The metalanguage can thus be used to write tactics for the object language. Various underlying data types—including those of expressions, environments, and tactic states—and operations on them are reflected as meta constants. During VM evaluation, terms

of these types are replaced with objects of the underlying data type, and operations are similarly linked to their underlying implementations.

Consider, as a short example, the type `expr` (Figure 2.1), which reflects the grammar of Lean expressions. Lean's library contains the declaration `meta constant expr.has_var : expr → bool`, which exposes a function written in C++ that returns true when the input expression contains a free variable. To the kernel, this term is a constant with no computational behavior, but if one evaluates `expr.has_var e (expr.var 0)` in the VM, the result will be `tt`.

With the metaconstant `tactic_state`, which exposes the environment, local hypotheses, and goals of a proof in progress, this metaprogramming framework allows users to write complex procedures for constructing proofs. A term of type `tactic A` is a function `tactic_state → tactic_result A`, where a result is either a success (pairing a new `tactic_state` with a term of type `A`) or a failure. Proof obligations can be discharged by terms of type `tactic unit`; such a term is executed in the VM to transform the original `tactic_state` into one in which all goals have been solved. More generally, we can think of a term of type `tactic A` as a program that attempts to construct a term of type `A`, while optionally changing the tactic state.

It is important to emphasize again that these tactics do not lower the trustworthiness of the Lean system. While arbitrarily complex and potentially nonterminating tactics may be used to prove a theorem, they do so by constructing a proof term in the object language. The proof is not accepted until this term has been checked by the kernel. It is thus impossible for tactics with "unsound" behavior to introduce inconsistencies into Lean. One could define a tactic which replaces all goals with `true`, and solves them by applying `true.intro`, and while this tactic would execute successfully on any goal, the resulting proof term would be rejected.

When writing tactics, the command `do` enables Haskell-like monadic syntax. For example, the following tactic returns the number of goals in the current tactic state. The type of the meta constant `get_goals` is `tactic (list expr)`, where `list` is the standard (object-level) type defined in the Lean library.

```
meta def num_goals : tactic nat :=
do gs ← get_goals,
   return (length gs)
```

The tactic monad provides access to many low-level functions implemented in Lean's underlying language, including its elaborator and unifier. With these features exposed, it is possible to implement many of the tactics commonly found in proof assistants in the language of Lean itself, as they are generally built out of these atomic parts. The cores of some performance-critical tactics, including the simplifier, are implemented in C++, but the "user-facing" wrappers for these tactics are exposed in the tactic language.

Lean's metalanguage also provides an API for tagging declarations with *attributes*, and for retrieving the attributes associated with a particular declaration or the declarations associated

with a particular attribute. This allows metaprograms to be extensible. Lean's simplifier, for instance, will search the environment for declarations attributed as simp lemmas. Users can define their own attributes, with or without parameters and caching behavior.

Much of the power of Lean's metaprogramming framework comes from its integration with the object language. Theories and automation can be developed in-line in the same environment, ensuring a tight fit between the two. The tactics associated with a theory are thus an essential part of the theory itself.

2.3.1 Relation to the object language

The metalanguage of Lean can be looked at as an extension of the object language, where by the "object language" we mean the collection of permitted declarations without the `meta` prefix. Any declaration in the object language lifts without modification to one in the metalanguage. This extention is strict: not every metadefinition is valid in the object language. Similarly, while both languages can be interpreted in the VM, kernel reduction (and its ensuant level of trust) applies only to the object language.

Consider this simple example, which exploits the relaxed termination checking in the metalanguage:

```
meta def {u} inh_aux (α : Sort u) : unit → α
| () := inh_aux ()
meta def {u} inh (α : Sort u) : α := inh_aux α ()
```

It is clear that `inh` implies that every type, including the empty type, is inhabited, and can be instantiated to a (meta) proof of `false`. To allow `inh` to appear anywhere in a declaration in the object language would be unsound. We see that there is no consistent notion of a "metaproposition," and that only proofs written in the object language are trustworthy.

Moreover, one cannot use the object language of Lean to prove facts *about* metaprograms. The consequence for the projects described in this document is that it is impossible to state and prove formal soundness theorems in Lean. This should not reduce one's trust in any particular application, as the results provided by both tools are formally checked, but it leaves open the possibility of errors that are only caught at runtime.

Since the tactic monad, `expr` type, and various other building blocks are metadeclarations in Lean, it is not possible to implement these kinds of programs entirely in the object language. In principle, it would be possible to do so nearly entirely, wrapped in a minimal collection of metadeclarations. The object-level component could then be formally proved correct with respect to a given semantics in Lean. This would amount to proof-by-reflection, and would carry the attendant costs of kernel computation and a more difficult proving task. We believe that the post-hoc nature of these tools is a blessing, not a curse, for reasons discussed in Section 1.2, and are not concerned about the impossibility of formally proving correctness.

Chapter 3

A proof-producing method for verifying inequalities[1]

3.1 Introduction

The real numbers are a basic building block in many areas of mathematics, as well as in the modeling of physical processes. Beginning in primary school algebra, students learn how to compare real valued expressions and show that they have certain properties. No level of mathematics escapes these kinds of arguments: for example, Hales' proof of the Kepler Conjecture [73] involved the systematic verification of thousands of systems of real-valued inequalities. Given their omnipresence, it is no surprise that a variety of algorithms and tools have been developed to perform many types of symbolic computations over the reals.

Perhaps more surprising is the relative scarcity of *verified* algorithms focused on the real numbers. Certainly, general-purpose tools for rewriting, reasoning over rings or fields, and linear arithmetic apply perfectly well. But the real numbers, as a complete ordered field, are a common and unique structure, and verified reasoning with them in this context is computationally difficult. In Section 1.2 we describe some of the successes and challenges in verifying systems of inequalities over the real numbers. In summary, while $\mathbb{T}_{RCF}$ is decidable, known procedures are impractical on all but small problems. Furthermore, the most efficient techniques do not lend themselves to producing proofs; those that do produce proofs are impractically slow even on small problems. These procedures, both verified and unverified, are unsuited to producing proofs that fall even slightly outside of $\mathbb{T}_{RCF}$.

For mathematicians interested in interactive theorem proving, this is a particularly noticeable

[1] The prototype implementation of this project, written in Python, was described in the author's MS thesis [94] and in a paper published in the *Journal of Automated Reasoning* [14]. The work described in that paper was done in collaboration with Jeremy Avigad and Cody Roux. Later additions to the Python version, and the entire code base of the proof-producing version, are the work of the author.

problem. Small nonlinear inferences over $\mathbb{R}$ are often dismissed as "obvious" outside of educational settings, and while reasoning about these problems may not be entirely rote, it is usually done mindlessly. The relative lack of automation for this domain is a burden when working with a proof assistant.

Consider, for example, the implication

$$0 < x < y,\ u < v \ \Rightarrow\ 2u + \mathtt{exp}(1 + x + x^4) < 2v + \mathtt{exp}(1 + y + y^4)$$

The inference is not contained in linear arithmetic or even the theory of real closed fields. The inference is tight, so symbolic or numeric approximations to the exponential function are of no use. Backchaining using monotonicity properties of addition, multiplication, and exponentiation might suggest reducing the goal to subgoals $2u < 2v$ and $\mathtt{exp}(1 + x + x^4) < \mathtt{exp}(1 + y + y^4)$, but this introduces some unsettling nondeterminism. After all, one could just as well reduce the goal to

- $2u < \mathtt{exp}(1 + y + y^4)$ and $\mathtt{exp}(1 + x + x^4) < 2v$, or
- $2u + \mathtt{exp}(1 + x + x^4) < 2v$ and $0 < \mathtt{exp}(1 + y + y^4)$, or even
- $2u < 2v + 7$ and $\mathtt{exp}(1 + x + x^4) < \mathtt{exp}(1 + y + y^4) - 7$.

And yet, the inference is entirely straightforward. With the hypothesis $u < v$ in mind, it is easy to see that the terms $2u$ and $2v$ can be compared; similarly, the comparison between x and y leads to comparisons between x^4 and y^4, then $1 + x + x^4$ and $1 + y + y^4$, and so on.

The author's MS thesis [94] describes a method, developed with Jeremy Avigad and Cody Roux, that is based on such heuristically guided forward reasoning. While not a complete decision procedure for $\mathbb{T}_{RCF}$, this method is successful over a large class of problems; because of its modular structure, it can solve problems over expansions of $\mathbb{T}_{RCF}$. Furthermore, unlike many comparable tools, it can separate proof search from proof construction, and can thus be efficiently adapted to produce proof certificates. The project can be seen as a realization and extension of Avigad and Friedman's approximate decision procedure for $\mathbb{T}[\mathbb{Q}]$. It systematically applies "natural inferences"—canceling terms using addition and multiplication, instantiating axioms in "obvious" ways—to prove theorems in expansions of the language of real closed fields.

The method has been implemented in Python, and the code is available at

https://github.com/avigad/polya.

We have named the system "Polya," after George Pólya, in recognition of his work on inequalities [76] as well as his thoughtful studies of heuristic methods in mathematics [119].

The Python implementation of Polya answers questions about the satisfiability of heterogeneous systems, but it does so without justification. To trust its answer, one must assume there are no soundness bugs in the 15,000 lines of Python code. For more concrete certainty, Polya should produce rigorous proofs that can be verified by an independent system.

Depending on one's notion of rigor, this can be achieved in a variety of ways. Many SAT and SMT solvers produce proof certificates by defining atomic reasoning steps and outputting a trace, which describes the sequence of such steps needed to reach the claimed conclusion. There are only limited standards on how "large" an atomic step can be, however, and many steps (including the preprocessing of terms) are often left unjustified. One could follow a similar approach for a system like Polya, but this would be subject to the same reliability concerns. Furthermore, since no other system uses similar proof steps, a proof checker would be unique to Polya and would likely not be trustworthy itself.

For greater rigor, one can ask for Polya to produce Lean proof terms. These can be checked by the Lean kernel, which has been heavily tested and not found to be unsound. Integrated into Lean, Polya could be used to discharge arithmetic assumptions that arise in proofs. Rather than extending the Python version of Polya to do this, it is more sensible to redesign and reimplement the system in the meta language of Lean. This is one contribution of this dissertation.

We begin this chapter by summarizing the contents of the author's MS thesis on the unverified implementation of Polya [94]. We then describe the implementation in Lean, focusing on the tracing and production of proofs. We conclude with some examples and describe the behavior of the system on a suite of test problems from KeYmaera [117].

3.2 The Polya algorithm

For a full description of the original version of Polya, we refer the reader to [94] or [15]. Here we will summarize the architecture and the general idea of the algorithm. The details in this section describe the original Polya.

Polya's structure has been largely inspired by the satisfiability modulo theories (SMT) approach to automated theorem proving, which combines "theory solver" modules with a SAT solver core. In some sense, Polya represents a generalization of the SMT technique to theories with overlapping signatures. In other senses, Polya is much weaker. For one, it lacks the capacity for backtracking and conflict-driven clause learning often found in the SAT core of an SMT solver. More importantly, restricting the common language to equality gives the SMT search a useful finite basis property: for a fixed number of variables $x_1, \ldots, x_n$, there are only finitely many ways to assign literals $x_i = x_j$ or $x_i \neq x_j$ for pairs i, j. Our system lacks this property, since literals have the form $x_i \bowtie c \cdot x_j$ with infinitely many possibilities for c. Nonetheless, the comparison between the two frameworks is strong, and one can imagine fruitful improvements to Polya by thinking about SMT.

Polya consists of a complex information database (the Blackboard) that serves as a common state between various inferential modules. These modules read the information contained in the Blackboard and use it to derive new facts, which are themselves added to the Blackboard. The key idea is that, while individual modules can selectively use certain types of information and ignore others, many of the facts that the modules learn are in a shared language and can be used by

other modules. This allows them to exchange information and collectively reach conclusions that individual reasoning techniques would not be able to.

The core of the Polya algorithm involves modules for additive and multiplicative arithmetic reasoning. Considering the two domains separately avoids the complexity problems that arise in algorithms like cylindrical algebraic decomposition. By deriving facts in the shared language of comparisons $x_i \bowtie c \cdot x_j$, information learned in one domain can be shared with the other.

3.2.1 Term canonizer

We consider terms, such as $3(5x+3y+4xy)^2 f(u+v)^{-1}$, that are built up from variables and rational constants using addition, multiplication, rational powers, and function application. To account for the associativity of addition and multiplication, we view sums and products as multi-arity rather than binary operations. We account for commutativity by imposing an arbitrary ordering on terms and ordering the arguments accordingly.

Importantly, we would also like to easily identify the relationship between terms t and t' where $t = c \cdot t'$, for a nonzero rational constant c. For example, we would like to keep track of the fact that $4y + 2x$ is twice $x + 2y$. Towards that end, we distinguish between "terms" and "scaled terms." A scaled term is just an expression of the form $c \cdot t$, where t is a term and c is a rational constant. We refer to "scaled terms" as "s-terms" for brevity. This distinction is needed to make precise the following notion of canonization.

Definition 1. *We define the set of* terms $\mathcal{T}$ *and* s-terms $\mathcal{S}$ *by mutual recursion:*

$$\begin{aligned} t, t_i \in \mathcal{T} \quad &:= \quad 1 \mid x \mid \textstyle\sum_i s_i \mid \prod_i t_i^{n_i} \mid f(s_1, \ldots, s_n) \\ s, s_i \in \mathcal{S} \quad &:= \quad c \cdot t \; . \end{aligned}$$

Here x ranges over a set of variables*, f ranges over a set of* function symbols*, $c \in \mathbb{Q}$, and $n_i \in \mathbb{Z}$.*

Thus we view $3(5x + 3y + 4xy)^2 f(u + v)^{-1}$ as an s-term of the form $3 \cdot t$, where t is the product $t_1^2 t_2^{-1}$, t_1 is a sum of three s-terms, and t_2 is the result of applying f to the single s-term $1 \cdot (u + v)$.

Note that there is an ambiguity, in that we can also view the coefficient 3 as the s-term $3 \cdot 1$. This ambiguity is eliminated by a notion of *normal form* for terms, which clarifies when such coefficients must be considered as s-terms. The notion extends to s-terms: an s-term is in normal form when it is of the form $c \cdot t$, where t is a term in normal form. (In the special case where $c = 0$, we require t to be the term 1.) We also refer to terms in normal form as *canonical*, and similarly for s-terms. For details about this normal form, see [94].

3.2.2 Blackboard

The Blackboard serves as the system's information database, through which various modules communicate and share information. Metaphorically, we picture a number of mathematicians trained in different specialties working out problems on their own and writing results in a central location, where the results can be used as "black boxes" by the other mathematicians. The Blackboard does little computation itself, but tracks the information given to it in search of contradictions.

When the user asserts a comparison $s_1 \bowtie s_2$ to the Blackboard, s_1 and s_2 are first put in canonical form, and names $t_0, t_1, t_2, \ldots$ are introduced for each subterm. It is convenient to assume that t_0 denotes the canonical term 1. Given the example in the last section, the method could go on to define

$$t_1 := x, \quad t_2 := y, \quad t_3 := t_1 t_2, \quad t_4 := t_1 + (3/5) \cdot t_2 + (4/5) \cdot t_3,$$
$$t_5 := u, \quad t_6 := v, \quad t_7 := t_5 + t_6, \quad t_8 = f(t_7), \quad t_9 := t_4^2 t_8^{-1}$$

In that case, $75 \cdot t_9$ represents $3(5x + 3y + 4xy)^2 f(u + v)^{-1}$.

Any subterm common to more than one term is represented by the same name. Separating terms in this way ensures that each module can focus on only those definitions that are meaningful to it, and otherwise treat subterms as uninterpreted constants.

Any comparison $s \bowtie s'$ between canonical s-terms, where $\bowtie$ denotes any of $<, \leq, >, \geq, =$, and $\neq$, translates to a comparison $c_i t_i \bowtie c_j t_j$, where t_i and t_j name canonical terms. But this, in turn, can always be expressed in one of the following ways:

- $t_i \bowtie 0$ or $t_j \bowtie 0$, or
- $t_i \bowtie c \cdot t_j$, where $c \neq 0$ and $i < j$.

The blackboard therefore maintains the following data:

- a defining equation for each t_i, and
- comparisons between named terms, as above.

Note that this means that, *a priori*, modules can only look for and report comparisons between terms that have been "declared" to the blackboard. This is a central feature of our method: the search is deliberately constrained to focus on a small number of terms of interest. The architecture is flexible enough, however, that modules can heuristically expand that list of terms at any point in the search. For example, our addition and multiplication modules do not consider distributivity of multiplication over addition, beyond multiplication of rational scalars. But if a term $x(y + z)$ appears in the problem, a module could heuristically add the identity $x(y + z) = xy + xz$, adding names for the new terms as needed.

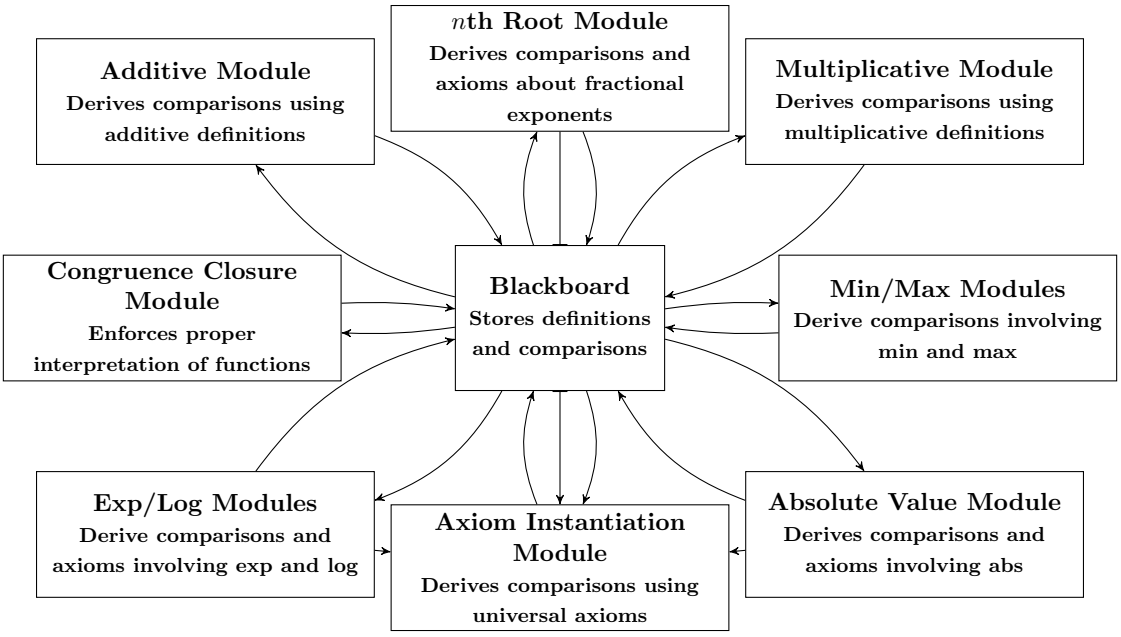


Figure 3.1: The inferential structure.

3.2.3 Modules

A Blackboard object is a static representation of known facts, akin to a chalkboard with notes but no note-writers. In order to produce a proof, someone or something must take these facts and use them to infer new ones. Our system incorporates a number of inferential modules to do this, illustrated in Figure 3.1.

We describe some of the key modules in more detail below; here, we simply note their common structure. Each module is equipped with an `update_blackboard` routine, which takes as an argument a Blackboard B. This routine selectively requests information from B, computes with it, and asserts new facts back to B. To the Blackboard, each module is a trusted black box. The modules are generally unaware of each other, with a few exceptions: the modules for interpreting absolute values, exponentials, and logarithms access the axiom instantiation module in order to axiomatize their respective functions.

Running Polya thus amounts to asserting hypotheses to a Blackboard, and then repeatedly updating the Blackboard using the various available modules. If at some point the Blackboard receives contradictory information, the process has proven the unsatisfiability of the hypotheses.

Arithmetic modules The heart of our procedure lies in proving arithmetical facts by separating the additive and multiplicative parts of terms. To this end, we have developed two modules that learn new comparisons about additive and multiplicative terms respectively. In fact, we have developed two versions of each of these modules. The first approach uses Fourier-Motzkin variable

elimination to deduce new facts; the second makes use of geometric insights to eliminate redundant computation, and depends on external software packages to run. To illustrate the tasks of the arithmetical modules, we will first describe the behavior of the additive routine. The multiplicative routine is largely analogous.

Given a collection of additive term definitions $\{t_i = \sum_{0 \leq j < i} c_{i,j} t_j\}_{1 \leq i \leq n}$ and a collection of atomic comparisons $\{t_i \bowtie c_{i,j} t_j\}$, for $\bowtie \in \{<, \leq, =, \geq, >\}$, our goal is to derive the "strongest" atomic comparisons between each pair t_i and t_j implied by the input. The notion of strength here is slightly subtle. For any t_i and t_j, two atomic comparisons $t_i \bowtie c t_j$ are necessary to have complete information about their relation.[2] For any set of three atomic comparisons, either one comparison is implied by the others, or the comparisons are unsatisfiable. This is easy to see by considering each comparison as a half-plane including the origin.

Our first approach to solving this problem uses the Fourier–Motzkin variable elimination algorithm [59] [136]. For each pair of variables t_i and t_j, we use this algorithm repeatedly to eliminate all other variables from the problem. It is easy to check that the remaining "reduced" inequalities are implied by the original set, and that any inequality between t_i and t_j implied by the original set is implied by the reduced set. In practice, this approach often works well, but the Fourier–Motzkin elimination algorithm has poor asymptotic behavior (doubly exponential in the number of variables) and this is visible on large problems.

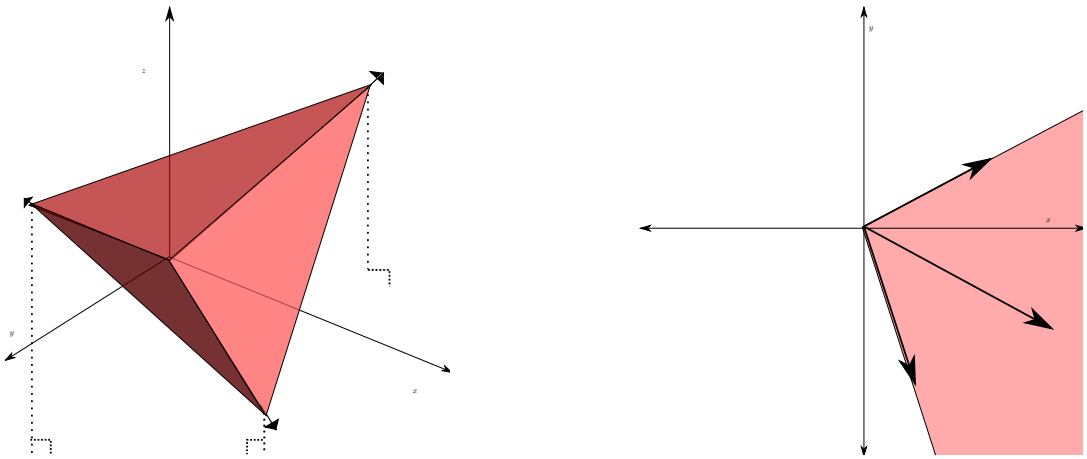
An alternative approach uses ideas from computational geometry. The set of solutions to the original inequalities can be seen as a polyhedral cone in $\mathbb{R}^n$; the feasible region for pairs (t_i, t_j) is the projection of this polyhedron to the $t_i t_j$ plane (Figure 3.2). By converting the inequality representation of this polyhedron to the corresponding vertex representation (c.f. [141, Section 1.1]), it is fairly easy to compute the inequalities that describe this feasible region. The halfplane-to-vertex conversion is somewhat more efficient than repeated applications of Fourier–Motzkin, but presents challenges for formalization.

To adapt these techniques for multiplicative inequalities, we note that the function e^x witnesses the isomorphism of ordered groups $\langle \mathbb{R}, 0, +, < \rangle \cong \langle \mathbb{R}^{>0}, 1, \cdot, < \rangle$.

Axiom module Users are allowed to define universally quantified axioms. A valid axiom has the form $(\forall x_1, \ldots, x_n)\varphi$, where the quantifier free formula φ consists of literal comparisons $c_i \cdot y_i \bowtie c_j \cdot y_j$ joined by logical connectives $\neg$, $\wedge$, $\vee$, and $\rightarrow$. Variables $\{y_i\}$ not among the bound variables $\{x_i\}$ are treated as constants. Upon definition, φ is immediately converted to conjunctive normal form, its literals are canonized, and its conjuncts are split into separate axioms; thus we may assume that each axiom has the clausal form $(\forall x_1, \ldots, x_n) \left(\bigvee y_i \bowtie c \cdot y_j\right)$. Instantiating an axiom of this form produces a clause that may be asserted to the Blackboard.

These uninstantiated axioms are stored in the axiom instantiation module. The main routine

[2] Here we ignore the minor but tedious complications introduced by equalities. Note also that comparisons with 0 ($t_i \bowtie 0 t_j$) are included in this count.



(a) A polyhedral cone in $\mathbb{R}^3$, defined by three half-spaces

(b) Projected to the xy plane, the polyhedron implies $x > 2y$ and $x > -\frac{1}{3}y$

Figure 3.2: Variable elimination by geometric projection

of this module observes the term definitions present in the Blackboard and searches for appropriate instantiations $\{x_i \mapsto c \cdot t_j\}$ for each axiom. To mediate between overaggressive (inefficient) and underaggressive (less effective) instantiation, a subset of terms in the axiom is unified with a set of terms defined in the Blackboard, with "term-matching" performed up to linear arithmetical equivalence.

Special function modules Polya has specialized modules for handling functions such as `exp`, `log`, `min`, and `floor`. These modules heuristically instantiate certain identities and comparisons for these functions that cannot be easily axiomatized in a general format. For instance, the exponential module will assert the equality between $\texttt{exp}(x + y)$ and $\texttt{exp}(x) \cdot \texttt{exp}(y)$.

3.3 Implementation in Lean: definitions and API

Different approaches can be used to transform an unverified algorithm into a proof-producing one. In some situations, algorithms are implemented in the object language of a proof assistant and verified directly; the certification of a particular computation is then a direct application of this verification theorem. This approach requires all computation to be done in the kernel of the proof assistant. A related approach, proof by reflection [97], is similarly dependent on kernel computation. Some algorithms produce results that can be certified post hoc, requiring little or no modification to the original algorithm; see Section 4.4.2 for some examples. The approach we use in the verified implementation of Polya most closely resembles this third option. During execution, we trace the sources of each inference step, and store the sources along with the derived fact. When the algorithm reaches a conclusion, we use this trace to produce a fully elaborated proof term.

Polya is well suited to this approach because its correctness does not depend on its search method; in other words, it naturally separates proof *search* from proof *verification*. This is, in large part, what allows the system to be modular. Changing the components or adding new ones may affect whether or not it finds a proof, but as long as the additions are sound, they will not affect the soundness of the system.

Proving by Polya is thus separated into two separate stages, *search* and *proof term reconstruction*. The search can be performed without reconstruction, and once a proof of falsity is found, it can be turned into a proof term. The Lean version of Polya is in this sense a direct generalization of the Python version. Users who are more concerned with speed than soundness can disable proof reconstruction and use the system much like the Python version.

In the current implementation, small proof reconstruction steps related to the normalization of terms are proved axiomatically. As work continues on this system and on Lean's internal automation, these gaps will be closed.

3.3.1 General architecture

The search process involves modules, currently an additive arithmetic module and a multiplicative arithmetic module, updating a shared Blackboard state. We can think of these update operations as happening within a `polya_state` monad, a standard state monad over a Blackboard object. Many operations on a Blackboard, including functions to add expressions and comparisons, retrieve comparison information, and check if a comparison is implied, are exposed in the `polya_state` monad. A module has type $\alpha \to$ `polya_state` α, where α is the type of cached information that the module stores. When implementing a module, one is expected to use the accessors and mutators in the `polya_state` monad.

Once a Blackboard has been found to be contradictory, it remains to reconstruct a proof of false. This reconstruction must happen within the tactic monad, for access to Lean's unifier and elaborator. It is possible to execute Polya outside of the tactic framework, but only if proof reconstruction is disabled. In practice, it is rare that one would want to run the algorithm outside of the tactic framework, and thus the primary user-facing interface is declared as

```
meta def polya_on_hyps (hys : list name) (rct : bool := tt) : tactic unit.
```

This tactic adds the provided hypotheses to a Blackboard and then cycles the additive and multiplicative modules. If both fail to learn any additional information in a given cycle, the tactic will fail. If a contradiction is found and `rct` is true, it will reconstruct the proof of contradiction and apply it.

We also define a tactic `tactic.interactive.polya`, which uses interactive-mode parsing (see Section 3.7) and applies `polya_on_hyps`. This allows us to write, e.g.,

```
example (h1 : u > 0) (h2 : u < 1*v) (h3 : z > 0) (h4 : 1*z + 1*1 < 1*w)
  (h5 : rat.pow (1*u + 1*v + 1*z) 3 ≥ 1* rat.pow (1*u + 1*v + 1*w + 1*1) 5) : false :=
```

```
inductive gen_comp
| le | lt | ge | gt | eq | ne

inductive comp
| le | lt | ge | gt

inductive spec_comp
| lt | le | eq
```

Figure 3.3: Comparison data types

```
/--
 An inequality (str, a, b) represents the halfplane counterclockwise
 from the vector (a, b). If str is true, it is strict
 (i.e., it doesn't include the line bx-ay=0).
-/
structure ineq :=
(strict : bool)
(x y : ℚ)

meta structure ineq_data (lhs rhs : expr) :=
(inq : ineq)
(prf : ineq_proof lhs rhs inq)
```

Figure 3.4: Inequality data types

```
by polya h1 h2 h3 h4 h5.
```

We describe a more truly interactive method of using Polya in Section 3.7.

3.3.2 Data types

We distinguish between *equalities*, *inequalities*, and *disequalities*. For each type of comparison, we define a `proof` datatype, which represents a justification for a particular comparison. A `proof` object can be elaborated into an actual Lean proof term. A `data` object pairs a comparison with a `proof` object, discussed below.

For any two expressions x and y, we may know inequalities, equalities, and/or disequalities of the form $x \bowtie c \cdot y$. Inequality information can be maximally represented by two such comparisons; any third inequality is either contradictory, redundant, or makes one of the previous two redundant. Alternatively, one equality of the form $x = c \cdot y$ makes any inequalities between x and y either contradictory or redundant. (These inequalities may imply sign information about x and y, which is stored elsewhere, but it is never necessary to store both an equality and an inequality between x and y.) We use the datatype `ineq_info` to store inequality and equality comparisons between two expressions.

We may also know data of the form $x \neq c \cdot y$ for any number of coefficients c. These are stored

```
meta structure eq_data (lhs rhs : expr) :=
(c   : ℚ)
(prf : eq_proof lhs rhs c)

meta structure diseq_data (lhs rhs : expr) :=
(c : ℚ)
(prf : diseq_proof lhs rhs c)

meta structure sign_data (e : expr) :=
(c : gen_comp)
(prf : sign_proof e c)
```

Figure 3.5: Equality, disequality, and sign data types

```
meta inductive ineq_info (lhs rhs : expr)
| no_comps {}  : ineq_info
| one_comp     : ineq_data lhs rhs → ineq_info
| two_comps    : ineq_data lhs rhs → ineq_data lhs rhs → ineq_info
| equal        : eq_data lhs rhs → ineq_info

meta def diseq_info (lhs rhs : expr) := rb_map ℚ (diseq_data lhs rhs)

meta def sign_info (e : expr) := option (sign_data e)
```

Figure 3.6: Information data types

in the `diseq_info` datatype. Finally, we may know at most one piece of sign information $x \bowtie 0$; this information is stored in the `sign_info` datatype.

Every expression entered into Polya is either a sum of expressions, a product of expressions, or an atom. Rather than naming subexpressions as in the Python version of Polya, we operate directly on terms of type `expr` themselves. Each input expression is paired with a term of type `expr_form`, which decomposes it into its compound structure. The types `sum_form` and `prod_form` are also used in the additive and multiplicative arithmetic modules, respectively.

A `blackboard` constitutes the basic state of the Polya process. It contains all of the information that as yet has been derived. In particular, it contains inequality, equality, and disequality information for any pair of expressions, and sign information for every expression. A Blackboard also contains a list of the expressions present, along with parsed versions of those expressions. If a Blackboard has been found to be contradictory, it also contains a `contrad` object tracing this proof.

To avoid duplicating symmetric information, comparisons are only stored between expressions `lhs` and `rhs` if `lhs` is greater than `rhs` using Lean's default expression comparison. Any piece of information can be inverted.

```
-- A sum is a set of summands, each with a rational coefficient.
meta def sum_form := rb_map expr ℚ

/- A product is a set of multiplicands, each with an integer exponent,
   and a leading rational coefficient. -/
meta structure prod_form :=
(coeff : ℚ)
(exps : rb_map expr ℤ)

meta inductive expr_form
| sum_f : sum_form → expr_form
| prod_f : prod_form → expr_form
| atom_f : expr → expr_form
```

Figure 3.7: Expression structure data types

```
meta structure blackboard :=
(ineqs : hash_map (expr×expr) (λ p, ineq_info p.1 p.2))
(diseqs : hash_map (expr×expr) (λ p, diseq_info p.1 p.2))
(signs : hash_map expr sign_info)
(exprs : rb_set (expr×expr_form))
(contr : contrad)
(changed : bool := ff)
```

Figure 3.8: The Blackboard data type

3.3.3 Proof trace data types

Every piece of information entered into a Blackboard must reference its source. That is, if a fact $x \bowtie c \cdot y$ is asserted, there must be a method to produce a Lean proof of this fact. The `ineq_proof`, `eq_proof`, `diseq_proof`, `sign_proof`, and `contrad` data types contain these proof traces. The `sum_form_proof` and `prod_form_proof` data types represent proof traces for the structures used in the arithmetic modules.

Each of these data types is an inductive type, whose constructors represent different ways of proving the corresponding kind of fact. For example, given a proof that $x > 0$ (sign information), one can derive a proof that $x > 0 * y$ (inequality information). Thus, there is a `ineq_proof` constructor `zero_comp_of_sign_proof`, which takes a `sign_proof` as an argument.

The interplay between these proof types is intricate. For example, an equality may be justified by two inequalities; an inequality may be justified by another inequality and sign information; sign information might be justified by an equality and a list of inequalities. Thus, the datatypes are mostly defined mutually. We avoid some unnecessary complexity by including an `adhoc` constructor for each proof type, which requires a reconstruction tactic to be provided when it is used. A finite list of constructors cannot account for all of the ways in which one could prove a certain kind of information, so the `adhoc` constructor serves to cover all of the remainign possibilities.

We will not discuss every constructor here, but it is worth noting why the dependency between `ineq_proof` and `sum_form_proof` is necessary. A common way to find a contradiction is to derive three mutually unsatisfiable inequalities between two variables. For instance, we may derive that $x > y$, $x < 2y$, and $x < (1/2)y$. It is easy to note that these facts are contradictory, but a general method for producing a proof of this fact involves the same linear arithmetic that is used in the additive module. It is thus helpful to allow inequality proofs to refer to linear arithmetic. We do not need an `of_prod_form_proof` constructor for `ineq_proof` or others, as this circularity is not present. Given a `prod_form_proof`, we can construct an `ineq_proof` or an `eq_proof` using the `adhoc` constructors.

3.3.4 Blackboard API

Polya is designed to be easily extensible; it should be simple for users to design new modules. Modules integrate into the broader system by processing and updating a Blackboard object. Thus, it is important that the Blackboard provide a clean interface for these modules. Note that many of the accessor and manipulator functions described here lift immediately to the `polya_state` monad.

A Blackboard provides basic tools for accessing the known information between expressions, visible in Figure 3.11.

It also provides low-level access to its data structures, allowing information to be inserted directly (Figure 3.12). These manipulator functions should generally be avoided, as they do not preserve dependencies between the structures. For instance, adding the sign fact that $x > 0$ necessitates adding the inequality fact $x > 0 * y$, but the low level manipulators will not do this. Instead, the compound `polya_state` functions in Figure 3.13 should be used. These functions will, for example, ensure that subexpressions are added to the list of expressions; use known disequalities to strengthen weak inequalities to strict inequalities; and, if contradictory information is asserted, raise the `is_contr` flag.

A number of relations are maintained between the Blackboard's data structures.

- Sign information is duplicated as inequality information: that is, $x < 0$ is also represented as $x < 0 \cdot y$, for each known expression y. This ensures that one can check whether an inequality between x and y is implied by looking only at an `ineq_info x y` object.

- Other redundant information is removed as soon as possible. Since $x > 0$ implies $x \neq -2$, the latter information will be deleted from the Blackboard when the former is asserted.

- In the `two_comps` case of an `ineq_info` object, we maintain that the defining ray of the first inequality is counterclockwise to that of the second. This simplifies the computations that determine whether a piece of information is known.

```
meta mutual inductive eq_proof, ineq_proof, sign_proof, sum_form_proof
with eq_proof : expr → expr → ℚ → Type
| hyp : Π lhs rhs c, expr → eq_proof lhs rhs c
| sym : Π {lhs rhs c}, Π (ep : eq_proof lhs rhs c), eq_proof rhs lhs (1/c)
| of_opp_ineqs : Π {lhs rhs i}, Π c,
  ineq_proof lhs rhs i → ineq_proof lhs rhs (i.reverse) → eq_proof lhs rhs c
| of_sum_form_proof : Π lhs rhs c {sf}, sum_form_proof ⟨sf, spec_comp.eq⟩ → eq_proof lhs rhs c
| adhoc : Π lhs rhs c, tactic expr → eq_proof lhs rhs c

with ineq_proof : expr → expr → ineq → Type
| hyp : Π lhs rhs i, expr → ineq_proof lhs rhs i
| sym : Π {lhs rhs i}, ineq_proof lhs rhs i → ineq_proof rhs lhs (i.reverse)
| of_ineq_proof_and_diseq : Π {lhs rhs i c},
    ineq_proof lhs rhs i → diseq_proof lhs rhs c → ineq_proof lhs rhs (i.strengthen)
| of_ineq_proof_and_sign_lhs : Π {lhs rhs i c},
    ineq_proof lhs rhs i → sign_proof lhs c → ineq_proof lhs rhs (i.strengthen)
| of_ineq_proof_and_sign_rhs : Π {lhs rhs i c},
    ineq_proof lhs rhs i → sign_proof rhs c → ineq_proof lhs rhs (i.strengthen)
| zero_comp_of_sign_proof : Π {lhs c} rhs i, sign_proof lhs c → ineq_proof lhs rhs i
| horiz_of_sign_proof : Π {rhs c} lhs i, sign_proof rhs c → ineq_proof lhs rhs i
| of_sum_form_proof : Π lhs rhs i {sfc}, sum_form_proof sfc → ineq_proof lhs rhs i
| adhoc : Π lhs rhs i, tactic expr → ineq_proof lhs rhs i

with sign_proof : expr → gen_comp → Type
| hyp  : Π e c, expr → sign_proof e c
| ineq_lhs : Π c, Π {lhs rhs iqp}, ineq_proof lhs rhs iqp → sign_proof lhs c
| ineq_rhs : Π c, Π {lhs rhs iqp}, ineq_proof lhs rhs iqp → sign_proof rhs c
| eq_of_two_eqs_lhs : Π {lhs rhs eqp1 eqp2},
    eq_proof lhs rhs eqp1 → eq_proof lhs rhs eqp2 → sign_proof lhs gen_comp.eq
| eq_of_two_eqs_rhs : Π {lhs rhs eqp1 eqp2},
    eq_proof lhs rhs eqp1 → eq_proof lhs rhs eqp2 → sign_proof rhs gen_comp.eq
| diseq_of_diseq_zero : Π {lhs rhs}, diseq_proof lhs rhs 0 → sign_proof lhs gen_comp.ne
| eq_of_eq_zero : Π {lhs rhs}, eq_proof lhs rhs 0 → sign_proof lhs gen_comp.eq
| ineq_of_eq_and_ineq_lhs : Π {lhs rhs i c}, Π c',
    eq_proof lhs rhs c → ineq_proof lhs rhs i →  sign_proof lhs c'
| ineq_of_eq_and_ineq_rhs : Π {lhs rhs i c}, Π c',
    eq_proof lhs rhs c → ineq_proof lhs rhs i →  sign_proof rhs c'
| ineq_of_ineq_and_eq_zero_rhs : Π {lhs rhs i}, Π c,
    ineq_proof lhs rhs i → sign_proof lhs gen_comp.eq → sign_proof rhs c
| diseq_of_strict_ineq : Π {e c}, sign_proof e c → sign_proof e gen_comp.ne
| of_sum_form_proof : Π e c {sfc}, sum_form_proof sfc → sign_proof e c
| adhoc : Π e c, tactic expr → sign_proof e c

with sum_form_proof : sum_form_comp → Type
| of_ineq_proof : Π {lhs rhs iq}, Π id : ineq_proof lhs rhs iq,
    sum_form_proof (sum_form_comp.of_ineq lhs rhs iq)
| of_eq_proof : Π {lhs rhs c}, Π id : eq_proof lhs rhs c,
    sum_form_proof (sum_form_comp.of_eq lhs rhs c)
| of_sign_proof : Π {e c}, Π id : sign_proof e c, sum_form_proof (sum_form_comp.of_sign e c)
| of_add_factor_same_comp : Π {lhs rhs c1 c2}, Π m : ℚ,
    sum_form_proof ⟨lhs, c1⟩ → sum_form_proof ⟨rhs, c2⟩ →
    sum_form_proof ⟨lhs.add_factor rhs m, spec_comp.strongest c1 c2⟩
| of_add_eq_factor_op_comp : Π {lhs rhs c1}, Π m : ℚ, sum_form_proof ⟨lhs, c1⟩ →
    sum_form_proof ⟨rhs, spec_comp.eq⟩ → sum_form_proof ⟨lhs.add_factor rhs m, c1⟩
| of_scale : Π {sfc}, Π m, sum_form_proof sfc → sum_form_proof (sfc.scale m)
| of_expr_def : Π (e : expr) (sf : sum_form),  sum_form_proof ⟨sf, spec_comp.eq⟩
| adhoc : Π sfc, tactic expr → sum_form_proof sfc
```

Figure 3.9: Proof objects (part 1)

```
meta inductive diseq_proof : expr → expr → ℚ → Type
| hyp : ∏ lhs rhs c, expr → diseq_proof lhs rhs c
| sym : ∏ {lhs rhs c}, ∏ (dp : diseq_proof lhs rhs c), diseq_proof rhs lhs (1/c)


meta inductive prod_form_proof : prod_form_comp → Type
| of_ineq_proof : ∏ {lhs rhs iq cl cr},
    ∏ (id : ineq_proof lhs rhs iq) (spl : sign_proof lhs cl) (spr : sign_proof rhs cr),
    prod_form_proof (prod_form_comp.of_ineq lhs rhs cl cr iq)
| of_eq_proof : ∏ {lhs rhs c}, ∏ (id : eq_proof lhs rhs c) (lhsne : sign_proof lhs
    gen_comp.ne),
    prod_form_proof (prod_form_comp.of_eq lhs rhs c)
| of_expr_def : ∏ (e : expr) (pf : prod_form), prod_form_proof ⟨pf, spec_comp.eq⟩
| of_pow : ∏ {pfc}, ∏ z, prod_form_proof pfc → prod_form_proof (pfc.pow z)
| of_mul : ∏ {lhs rhs c1 c2}, prod_form_proof ⟨lhs, c1⟩ → prod_form_proof ⟨rhs, c2⟩ →
    (list Σ e : expr, sign_proof e gen_comp.ne) →
    prod_form_proof ⟨lhs*rhs, spec_comp.strongest c1 c2⟩
| adhoc : ∏ pfc, tactic expr → prod_form_proof pfc
```

Figure 3.10: Proof objects (part 2)

```
get_ineqs (bb : blackboard) (lhs rhs : expr) : ineq_info lhs rhs
get_diseqs (bb : blackboard) (lhs rhs : expr) : diseq_info lhs rhs
get_sign_info (bb : blackboard) (e : expr) : sign_info e
get_sign_comp (bb : blackboard) (e : expr) : option gen_comp
has_sign_info (bb : blackboard) (e : expr) : bool
has_strict_sign_info (bb : blackboard) (e : expr) : bool
has_weak_sign_info (bb : blackboard) (e : expr) : bool
get_expr_list (bb : blackboard) : list expr
```

Figure 3.11: Blackboard accessors

```
add_expr (e : expr) (ef : expr_form) (bb : blackboard) : blackboard
insert_sign (e : expr) (si : sign_data e) (bb : blackboard) : blackboard
insert_diseq {lhs rhs} (dd : diseq_data lhs rhs) (bb : blackboard) : blackboard
insert_ineq_info {lhs rhs} (ii : ineq_info lhs rhs) (bb : blackboard) : blackboard
```

Figure 3.12: Low-level Blackboard manipulators

```
get_ineq_list : polya_state (list Σ lhs rhs, ineq_data lhs rhs)
get_eq_list : polya_state (list Σ lhs rhs, eq_data lhs rhs)
get_sign_list : polya_state (list Σ e, sign_data e)

get_add_defs : polya_state (list (expr × sum_form))
get_mul_defs : polya_state (list (expr × prod_form))

add_ineq {lhs rhs : expr} (id : ineq_data lhs rhs) : polya_state unit
add_eq {lhs rhs : expr} (ed : eq_data lhs rhs) : polya_state unit
add_diseq {lhs rhs : expr} (dd : diseq_data lhs rhs) : polya_state unit
add_sign {e : expr} (sd : sign_data e) : polya_state unit
```

Figure 3.13: High-level Blackboard functions

```
meta structure sum_form_comp :=
(sf : sum_form)
(c : spec_comp)

meta structure sum_form_comp_data :=
(sfc : sum_form_comp)
(prf : sum_form_proof sfc)
(elim_list : rb_set expr)
```

Figure 3.14: The datatypes used for additive Fourier–Motzkin elimination

3.4 Proof search

As described in Section 3.2, Polya's proof search proceeds by independent modules querying and asserting information from and to a shared Blackboard object. The two core modules learn arithmetic information, about additive and multiplicative terms respectively. The proof-producing implementations are similar to the Fourier–Motzkin modules in the Python implementation.

3.4.1 Additive module

A `sum_form_comp` object (Figure 3.14) represents a comparison of the form $\Sigma c_i \cdot x_i \bowtie 0$, where $\bowtie \in \{\leq, <, =\}$. Any piece of comparison data in the Blackboard that is not a disequality can be turned into a corresponding `sum_form_comp`. Additively defined expressions also induce `sum_form_comp` objects: for instance, the expression $3 \cdot x^2 + 4 \cdot y$ induces $(3 \cdot x^2 + 4 \cdot y) + -3 \cdot x^2 + -4 \cdot y = 0$. Conversely, any `sum_form_comp` with two or fewer components can be converted to a Blackboard-appropriate comparison.

Suppose an expression e appears in two `sum_form_comp` objects $s1$ and $s2$. If either $s1$ or $s2$ is an equality, or if the coefficient of e is positive in one and negative in the other, then $s1$ and $s2$ can be combined to eliminate e and produce a new `sum_form_comp`. There are corresponding constructors for `sum_form_proof`, so this elimination can be seen as an addition operator on `sum_form_comp_data`.

The elimination algorithm used in the proof-producing additive module is somewhat different from the unverified version, in order to facilitate caching the state after each round. Given a set S of `sum_form_comp_data` objects and a new `sum_form_comp` object s, we wish to extend S with all of the comparisons obtained by adding s to comparisons in S. These additions are restricted to those that do not reintroduce variables into a comparison from which they have already been eliminated; the `elim_list` field of `sum_form_comp_data` allows us to track this.

A function

```
meta def new_exprs_from_comp_data_set (sfcd : sum_form_comp_data)
    (cmps : rb_set sum_form_comp_data) : rb_set sum_form_comp_data
```

is defined to produce the set of new comparisons needed to extend S. We use this function to feed a queue of `sum_form_comp_data` objects into S.

```
meta def elim_list_into_set : rb_set sum_form_comp_data →
    list sum_form_comp_data → rb_set sum_form_comp_data
| cmps [] := cmps
| cmps (sfcd::new_cmps) :=
   if cmps.contains sfcd then elim_list_into_set cmps new_cmps else
   let new_gen := (new_exprs_from_comp_data_set sfcd cmps).keys in
   elim_list_into_set (cmps.insert sfcd) (new_cmps.append new_gen)
```

Seeded with the empty set and a list of comparisons l, this algorithm produces a set of comparisons equisatisfiable with the result obtained running the Python Fourier–Motzkin elimination procedure on l. By filtering out the comparisons with more than two components, we can convert the set into a set of `ineq_data` and `eq_data` objects.

The algorithm is designed to be incremental. That is, once a list of comparisons has been processed, it does not require any redundant computation to process one additional comparison. This allows us to store the state of the additive module once it has terminated, and resume from that state once more information has been learned; this is much more efficient than the former process, which restarted each time.

This algorithm is exposed via the function

```
meta def add_new_ineqs (start : rb_set sum_form_comp_data := mk_rb_set) :
    polya_state (rb_set sum_form_comp_data)
```

which extracts the relevant information from a Blackboard, computes the implied linear comparisons, and asserts them back to the Blackboard. An optional `start` argument allows it to begin from a saved state.

It takes only a few lines of code to rearrange these functions into a "standard" Fourier-Motzkin procedure, one which eliminates all variables from a set of comparisons in search of a proof of $0 < 0$. Since these functions are proof producing, this gives us a complete process for linear arithmetic for

```
meta structure prod_form_comp :=
(pf : prod_form)
(c : spec_comp)

meta structure prod_form_comp_data :=
(pfc : prod_form_comp)
(prf : prod_form_proof pfc)
(elim_list : rb_set expr)
```

Figure 3.15: The datatypes used for multiplicative Fourier–Motzkin elimination

free. (In fact, the Polya process is also complete for linear arithmetic, but less directly so than the standard method.)

3.4.2 Multiplicative module

The data types `prod_form`, `prod_form_comp`, and `prod_form_comp_data` (Figure 3.15) used in the multiplicative module are similar to their additive counterparts. A `prod_form_comp` represents a comparison of the form $1 \bowtie c \cdot \Pi x_i^{n_i}$, where $\bowtie \in \{\leq, <, =\}$. Call an inequality *redundant* if it is implied by the transitivity of sign information: for example, $x < y$ is redundant if $x < 0$ and $0 < y$. Any non-redundant, non-disequality Blackboard comparison can be turned into a `prod_form_comp`, provided the signs of its variables are known.

As described in Section 3.2, the multiplicative module operates only on terms for which the signs of all top-level variables are known. Access to this sign information is needed from within many functions, and occasionally comparisons with 1 are needed as well. It is convenient, then, to consider the collection of zero- and one-comparisons as a "state" and to work within the corresponding state monad (Figure 3.16).

The basic elimination algorithm is similar to that found in the previous section. One significant difference is that the multiplicative version is implemented within the `mul_state` monad, since the constructors for `prod_form_proof` (Figure 3.10) require proofs of sign information.

Another difference from the additive module is seen in the conversion of `prod_form_comp` objects into Blackboard-friendly data. Not every `prod_form_comp` corresponds to a comparison of the form $x \bowtie c \cdot y$. For example, in the absence of additional information, we cannot do anything with the `prod_form_comp` $1 \leq 1 \cdot (x^1 y^1)$. The `mul_state` monad includes information about how terms compare with 1 and -1, when this information is available, for use in these computations.

The first step in extracting Blackboard-friendly data from a `prod_form_comp` with two expressions is to balance the exponents of the expressions. Suppose we know that $1 \leq c \cdot (x^m y^n)$. We wish to find values k such that $1 \leq c \cdot (x^k y^{-k})$. Whether we can raise or lower the exponents m and n depends on the signs and relations to 1 of x and y, and the parities of m and n. We may be able to raise (or lower) the exponent by any integer or by only an even integer. Functions `can_raise`

```
meta def mul_state :=
state (hash_map expr sign_data × hash_map expr (λ e, ineq_info e rat_one))

meta instance mul_state.monad : monad mul_state := state.monad

-- returns the sign information of e
meta def si (e : expr) : mul_state (sign_data e) :=
λ hm, (hm.1.find' e, hm)

meta def oi (e : expr) : mul_state (ineq_info e rat_one) :=
λ hm, (hm.2.find' e, hm)

-- returns the known comparisons of e with 1 and -1
meta def oc (e : expr) : mul_state (option gen_comp × option gen_comp) :=
do ii ← oi e,
   return (find_comp ii 1, find_comp ii (-1))
```

Figure 3.16: The multiplicative state monad

and `can_lower` are implemented in Figure 3.17.

It may be possible to change m to be $-n$, to change n to be $-m$, to do both, or to do neither. The function `balance_coeffs (lhs rhs : expr) (el er : ℤ) : mul_state (list (ℤ × ℤ))` computes the possible exponent pairs. Once the exponents are balanced, we can find $\bowtie$ (depending on the sign of x or y) such that $x^k \bowtie c \cdot y^k$, and $c', \bowtie'$ such that $x \bowtie c' \cdot y$, where c' is either $\sqrt[k]{c}$ or a rational approximation in the correct direction.

There is also an analogue to the sign inference preprocessing done in the unverified version of Polya. This is run before the elimination, and also occurs within the `mul_state` monad; the state is reconstructed after processing to account for the consequences of the new sign information.

Computing the rational approximation. The standard nth root algorithm is not entirely suited to our purposes. We need to implement it in a way that

- if an exact root exists, we find it;
- we can bound the size of the denominator of the result; and,
- we can force an over- or under-approximation as needed.

Furthermore, since computing with rational numbers in the Lean VM is expensive, we would like to perform as much computation as possible at the integer level.

Our nth root algorithm takes a rational number (in reduced form, as required by Lean's `rat` datatype). By binary search on $\mathbb{Z}$, it checks if the numerator and the denominator are perfect nth powers. If not, it seeds a standard nth root algorithm with the quotient of the closest integer roots. When a root within the specified precision is reached, it rounds it to truncate the denominator, and

```
-- returns none if can never lower. some tt if can always lower. some ff if can only
    lower by even number
meta def can_lower (e : expr) (ei : ℤ) : mul_state (option bool) :=
do iplo ← is_pos_le_one e,
   if iplo then return $ some tt else  do
   ingno ← is_neg_ge_neg_one e,
   if ingno && (ei % 2 = 0) then return $ some ff else do
   ilno ← is_le_neg_one e,
   if ilno && (ei % 2 = 1) then return $ some ff
   else return none

meta def can_raise (e : expr) (ei : ℤ) : mul_state (option bool) :=
do igo ← is_ge_one e,
   if igo then return $ some tt else  do
   ilno ← is_le_neg_one e,
   if ilno && (ei % 2 = 0) then return $ some ff else do
   ingno ← is_neg_ge_neg_one e,
   if ingno && (ei % 2 = 1) then return $ some ff
   else return none
```

Figure 3.17: Test whether an exponent can be lowered or raised

```
meta inductive contrad
| none : contrad
| eq_diseq : Π {lhs rhs}, eq_data lhs rhs → diseq_data lhs rhs → contrad
| ineqs : Π {lhs rhs}, ineq_info lhs rhs → ineq_data lhs rhs → contrad
| sign : Π {e}, sign_data e → sign_data e → contrad
| strict_ineq_self : Π {e}, ineq_data e e → contrad
| sum_form : Π {sfc}, sum_form_proof sfc → contrad
```

Figure 3.18: The datatype for tracing contradictions

adds or subtracts the precision to correct the offset if needed. An extension of the `norm_num` tactic can be used during reconstruction to confirm that the nth power of the result relates correctly to the input value.

The Polya procedure cannot handle irrational exact roots, so in this situation, some precision is necessarily lost. In practice, this is rarely a problem.

3.5 Proof reconstruction

The process described so far is entirely "meta." Expression objects are only used, effectively, as indices: we perform no object-level arithmetic, application, or evaluation.

When Polya terminates with a contradiction, many Lean users are not willing to take its answer on faith. They would like a proof term, an expression in the object logic of type `false`.

Figures 3.9, 3.10, and 3.18 display the proof trace datatypes. A contradiction is represented by a

```
class comp_op (op : ℚ → ℚ → Prop) :=
(rev_op : ℚ → ℚ → Prop)
(rev_op_is_rev : ∀ {x y}, rev_op y x ↔ op x y)
(rel_of_sub_rel_zero : ∀ {x y : ℚ}, op (x-y) 0 ↔ op x y)
(op_mul : ∀ {x y c : ℚ}, c > 0 → op x y → op (c*x) (c*y))
(op_inv : ∀ {x y z : ℚ}, x > 0 → op z (x*y) → op ((rat.pow x (-1))*z) y)

class weak_comp_op (op) extends comp_op op :=
(strict_op : ℚ → ℚ → Prop)
(disj : ∀ {x y}, op x y ↔ x = y ∨ strict_op x y)
(ne_of_str : ∀ {x y}, strict_op x y → x ≠ y)
```

Figure 3.19: An algebraic structure for comparisons

`contrad` object. To turn a term of type `contrad` into a proof of false, we define `contrad.reconstruct`. Of course, we also need `ineq_proof.reconstruct`, `eq_proof.reconstruct`, and so on.

Reconstruction is relatively expensive. It is hard to avoid taxing Lean's elaborator and type class inference mechanism during the construction of large proofs like this. The process is designed to avoid as much definitional unfolding as possible. The `norm_num` tactic is frequently used to avoid kernel computation over rational numbers. But still, a large part of Polya's run time is spent in the reconstruction stage. It is for this reason that the algorithm is designed to strictly separate search and justification. The only proof terms that are reconstructed are those that appear in the final proof; search "tangents" that end up being unnecessary are ignored. Using proof trace data types, rather than full proof terms themselves, makes the process much more lightweight than it would otherwise be.

At the current stage, some small gaps exist in the proof reconstruction process. These are mainly related to algebraic normalization and cancellation, for example, reducing $2x + y - 2x$ to y. Lean's simplifier is soon to be able to do this kind of operation, and it was tangential to this dissertation to temporarily implement an alternative procedure. Proofs of this type are currently justified by `sorry`, a (trivially inconsistent) axiom used for placeholders in proof terms.

In general, the constructors for each type of proof trace are chosen to be fairly fine-grained. Justifying them should involve selecting and applying one or two lemmas from the proof reconstruction library. This library is designed to be as generic as possible over comparison operators (`<`, $\leq$, `=`, $\geq$, `>`, $\neq$). Many of the reconstruction theorems are stated in terms of `comp_op` and `weak_comp_op` (Figure 3.19).

The reconstruction process is similar for each type of proof trace object, so here we will focus on the `ineq_proof` and `sum_form_proof` examples. In fact, since many of the proof objects are mutually defined, their reconstruction functions must be as well. We gloss over this here, and assume that it only remains to define the functions in question.

```
meta def reconstruct : Π {lhs rhs : expr} {iq : ineq}, ineq_proof lhs rhs iq → tactic expr
| _ _ _ (hyp lhs rhs iq e) := reconstruct_hyp lhs rhs iq e
| _ _ _ (sym ip) := reconstruct_sym @reconstruct ip
| _ _ _ (of_ineq_proof_and_diseq ip dp) := reconstruct_ineq_diseq @reconstruct ip dp
| _ _ _ (zero_comp_of_sign_proof rhs iq sp) := reconstruct_zero_comp_of_sign rhs iq sp
| _ _ _ (horiz_of_sign_proof lhs iq sp) := reconstruct_horiz_of_sign lhs iq sp
| _ _ _ (of_sum_form_proof lhs rhs i sp) := reconstruct_of_sum_form_proof lhs rhs i sp
| _ _ _ (adhoc _ _ _ t) := t
```

Figure 3.20: The main function for reconstructing inequality proofs

3.5.1 Reconstructing inequality proofs

Inequality proofs are perhaps the most complex type of Blackboard-friendly comparison proof. The `ineq_proof` type has seven constructors, visible in the definition of `ineq_proof.reconstruct` in Figure 3.20.

Every piece of `ineq_data` has an associated normal form. The reconstruction of an `ineq_proof` should be a proof of this normal form expression. A term `iq : ineq` can be converted to a comparison `R` and a slope, which is either a rational `m` in reduced form or `slope.horiz`. In the former case, the normal form of an `ineq_data iq lhs rhs` object, where `lhs` and `rhs` are of type `expr` and `expr.lt rhs lhs`, is `lhs R c*rhs`. In the latter case, the normal form is `rhs R 0`.

Each auxiliary function begins by (perhaps recursively) reconstructing its proof object arguments. The auxiliary functions that require recursive access to `ineq_proof.reconstruct` receive it as an argument.

The `hyp` constructor. This constructor is used when a piece of `ineq_data` is derived directly from a hypothesis. The `reconstruct_hyp` function simply confirms that the `ineq_data` and the type of the hypothesis match.

The `sym` constructor. This constructor turns a proof of $x \bowtie c \cdot y$ into a proof of $y \bowtie' (1/c) \cdot x$, where $\bowtie'$ may be $\bowtie$ or the reverse of $\bowtie$ depending on the sign of c. Two lemmas are defined to handle each of these cases; `reconstruct_sym` detects the case and applies the appropriate lemma.

The `of_ineq_proof_and_diseq` constructor. A weak inequality $x \geq c \cdot y$ can be strengthened to $x > c \cdot y$ if a disequality $x \neq c \cdot y$ is known. The `reconstruct_ineq_diseq` function simply applies a lemma that proves this statement.

The `zero_comp_of_sign_proof` and `horiz_of_sign_proof` constructors. Sign information like $x > 0$ is also stored in the Blackboard as comparison information $x > 0 * y$ for each expression y. This redundancy simplifies the computation of implications, and makes it easier to modularize a number of computations. These constructors produce `ineq_proof` objects out of the corresponding

```
meta def reconstruct : Π {sfc}, sum_form_proof sfc → tactic expr
| _ (of_ineq_proof ip) := reconstruct_of_ineq_proof @reconstruct ip
| _ (of_eq_proof ep) := reconstruct_of_eq_proof @reconstruct ep
| _ (of_sign_proof sp) := reconstruct_of_sign_proof @reconstruct sp
| _ (of_add_factor_same_comp m sfpl sfpr) :=
      reconstruct_of_add_factor_same_comp @reconstruct m sfpl sfpr
| _ (of_add_eq_factor_op_comp m sfpl sfpr) :=
      reconstruct_of_add_eq_factor_op_comp @reconstruct m sfpl sfpr
| _ (of_scale m sfp) := reconstruct_of_scale @reconstruct m sfp
| _ (of_expr_def e sf) := reconstruct_of_expr_def e sf
```

Figure 3.21: The main function for reconstructing sum form proofs

`sign_proof` objects, given the new expression y. The former is used when the new variable appears on the right hand side of the comparison, and the latter when it appears on the left. Reconstructing these proofs is again straightforward.

The `of_sum_form_proof` constructor. The normal form of a `sum_form_comp` object is of the form $\Sigma c_i \cdot x_i \bowtie 0$, where $\bowtie \in \{<, \leq, =\}$. When there are exactly two summands, a proof of such a fact can be converted into an `ineq_proof` by moving one summand to the right hand side of the inequality and normalizing the left hand coefficient. Depending on the signs of the coefficients c_0 and c_1, the direction of the inequality may or may not change. The reconstruction function performs basic numeral arithmetic to do this rational normalization, and applies the correct one of two reconstruction lemmas.

The `adhoc` constructor. Users are able to add modules to the core Polya system, and these modules must have a way of adding information to the Blackboard. We cannot dynamically add constructors to the `ineq_proof` data type. Instead, an external module can create an `ineq_proof` object with `adhoc lhs rhs inq tac`, where `tac` is a tactic that constructs a proof of the normal form expression of the `ineq_data (lhs, rhs, inq)`. Reconstructing this proof simply involves executing `tac`.

3.5.2 Reconstructing sum form proofs

Many of the constructors for the `sum_form_proof` datatype (Figure 3.21) are similar to those seen above. In particular, `of_ineq_proof`, `of_eq_proof`, and `of_sign_proof` simply convert proofs to the normal form of sum forms. Here we discuss only the constructors that are specific to the elimination steps that occur in the additive module.

The `of_add_factor_same_comp` constructor. Two `sum_form_comp` objects $s_1 := \Sigma c_i x_i \bowtie 0$ and $s_1 := \Sigma d_i y_i \bowtie' 0$ can be added to produce a new `sum_form_comp` in the obvious way, where the

new comparison is the stronger of $\bowtie, \bowtie'$. Similarly, a `sum_form_comp` can be scaled by a positive coefficient m. This constructor justifies the comparison $s_1 + m \cdot s_2$ based on the justifications of m_1 and m_2. In the future, this justification will be handled by Lean's algebraic normalization module in the simplifier. Right now, the function `reconstruct_of_add_factor_same_comp` checks that the input proofs are as expected, and proves the result by `sorry`. The `of_add_eq_factor_op_comp` does similarly for negative values of m, applicable only when the second `sum_form_comp` is an equality.

The `of_scale` constructor. This constructor, and its reconstruction function, behave similarly to `of_add_factor_same_comp`, but only scale a single comparison rather than adding two.

The `of_expr_def` constructor. Given an expression $e := \Sigma_{i=1}^{n} c_i x_i$, we can create a `sum_form_comp` $\Sigma_{i=0}^{n} c_i x_i = 0$ by setting $c_0 = -1, x_0 = e$. The `sum_form_comp` objects of this form are necessary for the additive module to "decompose" additive definitions. Proofs of these equalities are by associativity and cancellation; they will be handled by Lean's algebraic normalizer in the future.

3.6 Examples and tests

We begin by discussing test cases from the Python implementation of Polya, to better describe the capabilities of the algorithm. Then, we provide test data for the proof-producing version. Finally, we discuss an integration of Polya into KeYmaera X, a system for verifying properties of hybrid systems.

3.6.1 Results from the unverified implementation

We aim to capture with our system a class of inferences that could be described as "natural" or "intuitive," that often come up in everyday proofs. For various reasons, Polya seems ill suited to attack large-scale problems in hundreds or thousands of variables, such as those found in industrial SMT applications. Smaller, heterogeneous problems make for more appropriate targets. These problems arise frequently in mathematics, both formal and informal, and are often surprisingly difficult for automated techniques to solve.

Here we highlight some of the noteworthy inferences that the Python implementation of Polya proves, and compare its performance with that of other automated provers. Not wishing to mislead, we also discuss some of the system's shortcomings. The results indicate that Polya fills a previously unfilled niche in the world of automated provers. We are able to prove many inferences, including a number found in real proofs and formalizations, that no other systems manage to solve. Given its significant shortcomings, we certainly do not expect Polya to replace these established systems, but it seems very promising as a tool to be used alongside them.

The examples seen here are a small selection of our test suite. Further examples can be found in the `examples` folder of the Polya distribution or in [94].

To start with, Polya handles inferences involving linear real inequalities, which are verified automatically by many interactive theorem proving systems. It can also handle purely multiplicative inequalities such as

$$0 < u < v < 1,\ 2 \le x \le y \ \Rightarrow\ 2u^2x < vy^2, \tag{3.1}$$

which are not often handled automatically. It can solve problems that combine the two, like these:

$$x > 1 \ \Rightarrow\ (1+y^2)x > 1+y^2 \tag{3.2}$$

$$0 < x < 1 \ \Rightarrow\ 1/(1-x) > 1/(1-x^2) \tag{3.3}$$

$$0 < u,\ u < v,\ 0 < z,\ z+1 < w \ \Rightarrow\ (u+v+z)^3 < (u+v+w)^5 \tag{3.4}$$

It also handles inferences that combine such reasoning with axiomatic properties of functions:

$$(\forall x.\ f(x) \le 1),\ u < v,\ 0 < w \ \Rightarrow\ u + w \cdot f(x) < v + w \tag{3.5}$$

$$(\forall x, y.\ x \le y \to f(x) \le f(y)),\ u < v,\ x < y \ \Rightarrow\ u + f(x) < v + f(y) \tag{3.6}$$

Isabelle's auto and Sledgehammer fail on all of these but (3.5) and (3.6), which are proved by resolution theorem provers. Sledgehammer can verify more complicated variants of (3.5) and (3.6) by sending them to Z3, but fails on only slightly altered examples.

Moreover, when handling nonlinear equations, Z3 "flattens" polynomials, which makes a problem like (3.4) extremely difficult. It takes Z3 a couple of minutes when the exponents 3 and 5 in that problem are replaced by 9 and 19, respectively. Polya verifies all of these problems in a fraction of a second, and is insensitive to the exponents in (3.4). It is also unfazed if any of the variables above are replaced by more complex terms.

Polya has built-in knowledge about many functions, such as `exp` and `log`, and verifies examples such as

$$z > \texttt{exp}(x), w > \texttt{exp}(y) \ \Rightarrow\ z^3 \cdot w^2 > \texttt{exp}(3x + 2y) \tag{3.7}$$

$$a > 1, c > 0, \texttt{log}(b^2) > 4, \texttt{log}(c) > 1, b \neq 0 \ \Rightarrow\ \texttt{log}(a \cdot b^2 \cdot c^3) > 7 \tag{3.8}$$

While Z3 sometimes succeeds on these types of examples, it needs to have the appropriate properties of `exp` or `log` described to it. It does not get either of the above.

Sledgehammer fails on an example that arose in connection with a formalization of the Prime Number Theorem, discussed in [10]:

$$0 \le n,\ n < (K/2)x,\ 0 < C,\ 0 < \varepsilon < 1 \ \Rightarrow\ \left(1 + \frac{\varepsilon}{3(C+3)}\right) \cdot n < Kx \tag{3.9}$$

Z3 verifies it when called directly. Sledgehammer also fails on these [12]:

$$0 < x < y \;\Rightarrow\; (1+x^2)/(2+y)^{17} < (1+y^2)/(2+x)^{10} \tag{3.10}$$

$$0 < x < y \;\Rightarrow\; (1+x^2)/(2+\mathtt{exp}(y)) \geq (2+y^2)/(1+\mathtt{exp}(x))\;. \tag{3.11}$$

Z3 gets (3.10) but not (3.11). Neither Sledgehammer nor Z3 get these:

$$(\forall x, y.\; f(x+y) = f(x)f(y)),\; a > 2,\; b > 2 \;\Rightarrow\; f(a+b) > 4 \tag{3.12}$$

$$(\forall x, y.\; f(x+y) = f(x)f(y)),\; a+b > 2,\; c+d > 2 \;\Rightarrow\; f(a+b+c+d) > 4 \tag{3.13}$$

Polya verifies all of the above easily.

The following problem was once raised on the Isabelle mailing list:

$$x > 0, y > 0, y < 1 \;\Rightarrow\; (x+y) > xy\;. \tag{3.14}$$

This inference is verified by Z3 as well as Sledgehammer, but both fail when x and y in the conclusion are replaced by x^{1500} and y^{1500}, respectively. Polya is insensitive to the exponent.

Let us consider two examples that have come up in recent Isabelle formalizations by Avigad [13]. Billingsley [26, page 334] shows that if f is any function from a measure space to the real numbers, the set of continuity points of f is Borel. Formalizing the proof involved verifying the following inequality:

$$\begin{aligned} i \geq 0,\; |f(y) - f(x)| < 1/(2(i+1)),& \\ |f(z) - f(y)| < 1/(2(i+1)) \;\Rightarrow\; |f(x) - f(y)| < 1/(i+1)\;.& \end{aligned} \tag{3.15}$$

Sledgehammer and Z3 fail on this, while Polya verifies it easily.

The second example involves the construction of a sequence $f(m)$ in an interval (a, b) with the property that for every $m > 0$, $f(m) < a + (b-a)/m$. The proof required showing that $f(m)$ approaches a from the right, in the sense that for every $x > a$, $f(m) < x$ for m sufficiently large. A little calculation shows that $m \geq (b-a)/(x-a)$ is sufficient. We can implicitly restrict the domain of f to the integers by considering only arguments $\lceil m \rceil$; thus the required inference is

$$\begin{aligned} (\forall m.\; m > 0 \rightarrow f(\lceil m \rceil) < a + (b-a)/\lceil m \rceil),& \\ a < b,\; x > a,\; m \geq (b-a)/(x-a) \;\Rightarrow\; f(\lceil m \rceil) < x\;.& \end{aligned} \tag{3.16}$$

Sledgehammer and Z3 do not capture this inference, and the Isabelle formalization was tedious. Polya verifies it immediately using only the information that $\lceil x \rceil \geq x$ for every x.

Of course, Polya fails on wide classes of problems where other methods succeed. To begin with,

it is much less efficient than the best linear solvers, and so should not be expected to scale to large industrial problems of this type. While there are many optimizations that could be made to Polya, we hold little hope of ever competing with established linear solvers; at its core, Polya is designed for nonlinear, heterogeneous problems.

Recall that the multiplicative module only takes advantage of equations where the signs of all terms are known. A preprocessing step to infer sign info helps with these sorts of problems. But this preprocessing is not robust, and Polya fails on some examples like the following:

$$x > 0,\ xyz < 0,\ xw > 0 \ \Rightarrow\ w > yz \tag{3.17}$$

However, problems of this sort are easily solved given a mechanism for splitting on the signs of w, y and z.

Another shortcoming, in contrast to methods which begin by flattening polynomials, is that Polya does not, *a priori*, make use of distributivity at all, beyond the distributivity of multiplication by a rational constant over addition. Any reasonable theorem prover for the theory of real closed fields can easily establish

$$x^2 + 2x + 1 \geq 0, \tag{3.18}$$

which can also be obtained simply by writing the left-hand side as $(x + 1)^2$. But, as pointed out by Avigad and Friedman [12], our method does not terminate on this example. Assuming the negation, Polya will learn that $x^2 \geq 0$, implying $2x + 1 \leq 0$ and thus $x \leq -1/2$. This implies $x^2 \geq 1/4$, beginning a cycle that will find progressively tighter bounds for x around -1.

Our method is known and intended to be incomplete, and there will always be problems on which it does not succeed. Many of these shortcomings are not worrisome—they are problems better left to other techniques. Nonetheless, there are improvements that could be made to Polya to help it handle difficult problems like these.

3.6.2 Results from the proof-producing implementation

In moving from an informal system to one that produces full proof terms, one generally expects a substantial slowdown. No matter what the domain, there is significant overhead in deriving and storing enough information for axiomatic justifications. The situation here is muddled, as both the implementation language and scope of the system have changed.

With proof production disabled, the Lean implementation of Polya solves the "smaller" arithmetic benchmarks in an amount of time comparable to the Python implementation. Enabling proof production does not make the system less effective—when the proof search terminates, the reconstruction does not time out—but in some cases, reconstruction takes noticeably more time than search. Some reasons for this are known, and are discussed in Section 3.9.2. Performance degrades on "large" problems, particularly those that involve computations with large rational numbers.

This is not unexpected, as again, the memory costs of producing proofs are quite high.

For a few examples, a proof trace of

$$u > 0,\ u < v < 1,\ 2 \leq x \leq y \ \Rightarrow\ u^2 x \leq (1/2) v y^2 \tag{3.19}$$

is found in .45 seconds, and reconstruction takes an additional .4 seconds. A trace of

$$0 < x < 3y,\ u < v < 0,\ 1 < v^2 < x \ \Rightarrow\ u(3y)^2 + 1 < vx^2 + x \tag{3.20}$$

is found in less than a tenth of a second, but reconstruction takes an additional tenth.

3.6.3 Integration into KeYmaera

KeYmaera is a verification tool for hybrid systems that combines deductive, real algebraic, and computer algebraic prover technologies [117] [118] [61]. Among other applications, it has been used to verify control systems for transportation systems. The current version of KeYmaera uses Z3 and Mathematica as a backend for solving the algebraic problems it generates. These algebraic problems are often well suited for Polya's approach.

We obtained a collection of 4442 problems generated by KeYmaera. With a 3 second timeout and case splitting disabled, the Python implementation of Polya was able to verify the unsatisfiability of 4252 (96%) in about six minutes. (With case splitting enabled, Polya solves an additional 15 problems, but runs for about ten minutes.)

To compare the Lean version of Polya on the same benchmarks, we have implemented a rudimentary SMTLIB parser in Lean. With the same timeout, we solve 72% of the benchmark problems. The total time spent is just under 90 minutes, but much of this can be attributed to the fixed cost of launching Lean for each problem. Using the link described in Chapter 4, we were also able to run the same benchmarks in Mathematica, which implements the CAD decision procedure for $\mathbb{T}_{RCF}$. Mathematica was able to solve 99% of the benchmark problems, but took nearly two hours (facing the same fixed cost from repeatedly launching Lean). This fixed cost is an implementation artifact, and we hope to significantly reduce it in the future; once this is done, the timing difference between Polya and Mathematica will be much more extreme.

3.7 Interacting with Polya

The unverified implementation of Polya functioned mainly as a push-button prover. In some sense, this belied the generality of the system; it can be informative for the user to inspect the prover's internal state, manually control the execution of modules, assert additional information during computation, or extract intermediate steps of the proof. Exposing an API for the internals of the system is consistent with the idea of "whitebox automation" expressed by de Moura [56].

```
meta structure polya_cache :=
(sum_cache : rb_set sum_form_comp_data)
(prod_cache : rb_set prod_form_comp_data)
(bb : blackboard)

meta def polya_tactic := state_t polya_cache tactic
```

Figure 3.22: Data structures for the interactive Polya mode

Lean supports an "interactive" tactic mode, where expressions appearing as arguments are auto-quoted to create a more natural proving environment. For instance, with a local expression `h : x < y`, users can write `apply le_of_lt h` to close a goal $x \leq y$. The argument `h` and application `le_of_lt h` are interpreted as pre-expressions, elaborated, and sent to the core `apply` tactic. The equivalent of this tactic in the non-interactive tactic mode would be `to_expr ``(le_of_lt h) >>= apply`.

This interactive mode can be extended beyond basic tactic proofs: for instance, Lean's SMT module includes an interactive mode. We have implemented a similar mode for Polya. The `polya_tactic` monad extends the normal tactic state with a `blackboard` and module caches (Figure 3.22). Functions of types `tactic α` and `polya_state α` lift to `polya_tactic α`, and module update functions, e.g. `sum_form.add_new_ineqs` and `prod_form.add_new_ineqs`, are also functions in this monad.

Using the Polya interactive mode (Figure 3.23), the user can, for example, choose when to add hypotheses to the blackboard; decide when to execute which modules; extract facts from the blackboard, and add them to the tactic state as local hypotheses; and trace the state of all or part of the blackboard at any given time. It is also straightforward for users to write custom Polya strategies.

3.8 Producing proof traces

Users interested in fully verified proof terms can enable proof reconstruction for the highest degree of certainty. Users who trust the system and are more interested in speed may wish to skip this step. It is also possible to produce an intermediate level of information: the proof trace objects can be assembled into "proof sketches," which outline the argument found by the system without fully verifying each step.

The arithmetic elimination steps done by the additive and multiplicative modules are easy to understand and difficult to display; these are good candidates to eliminate in a proof trace. Similarly, small steps such as normalization and reversing the order of a comparison can be skipped. We implement a rough heuristic for mapping proof trace objects to proof sketch objects, which are trees where each node is tagged with a fact and an explanation. The children of a node, stored in

```
-- equivalent to "by polya h1 h2 h3 h4 h5".
example (h1 : u > 0) (h2 : u < 1*v) (h3 : z > 0) (h4 : 1*z + 1*1 < 1*w)
  (h5 : rat.pow (1*u + 1*v + 1*z) 3 ≥ 1* rat.pow (1*u + 1*v + 1*w + 1*1) 5) : false :=
begin [polya_tactic]
  add_hypotheses h1 h2 h3 h4 h5,
  additive,
  multiplicative,
  reconstruct
end

-- adds terms "hzw1 : z < 1*w" and "hzw2 : z > 0*w" to the local context.
example (h1 : u > 0) (h2 : u < 1*v) (h3 : z > 0) (h4 : 1*z + 1*1 < 1*w)
  (h5 : rat.pow (1*u + 1*v + 1*z) 3 ≥ 1* rat.pow (1*u + 1*v + 1*w + 1*1) 5) : false :=
begin [polya_tactic]
  add_hypotheses h1 h2 h3 h4 h5,
  additive,
  extract_comparisons_between z w with hzw1 hzw2
end
```

Figure 3.23: The interactive Polya mode

a list, represent the facts that justify the parent.

```
meta inductive proof_sketch
| mk (fact : string) (reason : string) (justifications : list proof_sketch) :
    proof_sketch
```

The `adhoc` proof constructors present a minor obstacle to this sort of tracing, as they "flatten" the proof tree. To enable tracing, we extend the `adhoc` constructors with an additional argument of type `tactic proof_sketch`; to sketch an `adhoc` proof, one simply executes this tactic.

With a simple display function, one can see proof sketches as in Figures 3.24 and 3.25. Lean does not provide convenient tools for displaying information like this, but it is possible to format these sketches in a more pleasant (and graphical) way using the connection between Lean and Mathematica described in Chapter 4.

In principle, these proof sketches can serve more than just an informative purpose. With the proper tactics to fill in the gaps between steps, a proof sketch could be turned into an actual proof script, one which could replace the original call to Polya. Evaluating this proof script would be faster than running Polya, as most of the proof search stage will be eliminated. This behavior is similar to that of Isabelle's Sledgehammer tool [114], which replaces its own invocation with an appropriate proof script. The necessary tactics and formatting have not yet been implemented, and Lean's editor support for self-replacing tactics is limited, but this is a promising route for future work.

```
example  (h1 : x > 0) (h2 : x < 1*1)
  (h3 : (1*1 + (-1)*x)^(-1) ≤ 1*((1*1 + (-1)*x^2)^(-1))) : false

/-
false : contradictory inequalities
  1 ≤ 1*x^2 : by multiplicative arithmetic
    x^2 ≥ 1*x : by linear arithmetic
      1 * 1 + (-1) * x^2 ≤ 1*1 * 1 + (-1) * x : by multiplicative arithmetic
        (1 * 1 + (-1) * x)^-1 ≤ 1*(1 * 1 + (-1) * x^2)^-1 : hypothesis
        1 = 1 * ((1 * 1 + (-1) * x)^-1^-1 * (1 * 1 + (-1) * x)^-1) : by definition
        1 = 1 * ((1 * 1 + (-1) * x^2)^-1^-1 * (1 * 1 + (-1) * x^2)^-1) : by definition
    1 = 1 * (x^2^-1 * x^2) : by definition
  1 > 1*x^2 : by multiplicative arithmetic
    1 = 1 * (x^2^-1 * x^2) : by definition
    1 <1 * x^-1 : rearranging
      x < 1*1 : hypothesis
  x^2 > 0 : inferred from other sign data
-/
```

Figure 3.24: A formatted proof sketch

```
example (h1 : u > 0) (h2 : u < 1*v) (h3 : z > 0) (h4 : 1*z + 1*1 < 1*w)
  (h5 : (1*u + 1*v + 1*z)^3 ≥ 1*(1*u + 1*v + 1*w + 1*1)^5) : false

/-
false : contradictory inequalities
  1 * u + 1 * v + 1 * w + 1 * 1 ≤ 1*1 * u + 1 * v + 1 * z :
          by multiplicative arithmetic
    1 = 1 * ((1 * u + 1 * v + 1 * z)^3^-1 * (1 * u + 1 * v + 1 * z)^3) : by definition
    (1 * u + 1 * v + 1 * z)^3 ≥ 1*(1 * u + 1 * v + 1 * w + 1 * 1)^5 : hypothesis
    1 = 1 * ((1 * u + 1 * v + 1 * w + 1 * 1)^5^-1 * (1 * u + 1 * v + 1 * w + 1 * 1)^5)
    :
          by definition
  1 * u + 1 * v + 1 * z > 0 : by linear arithmetic
    u < 1*v : hypothesis
    u > 0 : hypothesis
    z > 0 : hypothesis
  1 * u + 1 * v + 1 * w + 1 * 1 > 1*1 * u + 1 * v + 1 * z : by linear arithmetic
    1 > 0 : hypothesis
    1 > 0 : hypothesis
    1 * z + 1 * 1 < 1*w : hypothesis
-/
```

Figure 3.25: A formatted proof sketch

3.9 Concluding thoughts

3.9.1 Related work

There are few systems that approach problems in the same domain as Polya, and fewer that do so in a certified way. Non-proof-producing systems with which we can make meaningful comparisons include Z3, an SMT solver developed at Microsoft Research [51]; Mathematica, a computer algebra system developed by Wolfram Research [137]; and MetiTarski, a resolution theorem prover by Paulson et al [5]. The comparisons below were performed on the collection of test problems collected for the Python implementation of Polya.

Z3 is a highly optimized SMT system that implements CAD as its nonlinear arithmetic theory solver. It performs successfully on a large class of problems, and has won numerous theorem-proving competitions. When restricted to problems involving linear arithmetic and axioms for function symbols, the behavior of Z3 and Polya is similar, although Z3 is vastly more efficient. As the examples above demonstrate, Polya's advantages show up in problems that combine multiplicative properties with either linear arithmetic or axioms. In particular, Z3 procedures for handling nonlinear problems do not incorporate axioms for function symbols. While CAD performs optimally in problems with two variables, the procedure's flattening and projection steps get bogged down in problems with three or more variables and larger exponents. It is on problems like this, such as Example 3.4 below, on which Polya's arithmetical capacity shines.

Mathematica's strengths and weaknesses are similar to those of Z3. It is able to handle many of the arithmetic examples in our test suite, but does so relatively slowly, and does not perform well on those involving function symbols.

MetiTarski also relies on a CAD elimination procedure, using the QEPCAD implementation in combination with a resolution prover. It targets theorems that involve specific real-valued operators, such as exp, log, and trigonometric functions, by using symbolic approximations. We found that, while MetiTarski is fairly successful on our purely arithmetical examples, it has similar weaknesses to Z3. It does not perform well on examples with interpreted functions, including the examples that involve tight inferences about exp.

Superposition, a proof search technique for first-order logic, can be parametrized by theory solvers for nonlinear arithmetic to attack problems involving transcendental constants [57]. It has been tested successfully on models of collision avoidance protocols, but does not seem to have been tested on problems like ours; in particular, the procedure seems tuned to notice satistiability rather than unsatisfiability.

Finally, ACL2 has support for nonlinear reasoning [86]. The method used there is locally somewhat similar to ours, although it lacks the same global guidance. ACL2 solves some, but not all, of the problems in our example suite; with manual configuration, it is able to solve additional problems. It does not seem to function well as a push-button prover on our examples, but a

knowledgeable user could be able to coax it into success.

McLaughlin and Harrison's proof-producing implementation of a decision procedure for $\mathbb{T}_{RCF}$ [103] is a theoretical comparison point for the Lean implementation of Polya. We have not run our examples using their tool in HOL Light, but based on the benchmarks provided in their paper, it seems unlikely to compare in efficiency, and does not handle examples involving function symbols.

We also verified a number of the following examples in Isabelle, trying to use Isabelle's automated tools as much as possible. These include "auto," an internal tableau theorem prover which also invokes a simplifier and arithmetic reasoning methods, and Sledgehammer [106] [27], which heuristically selects a body of facts from the local context and background library, and exports it to various provers. Sledgehammer successfully proved most of the same theorems as Z3 (which is not surprising, as Z3 is one of the provers it uses). The "auto" method only succeeded on the simplest examples.

3.9.2 Future work

There are two particularly pressing future tasks for the Polya project: the completion of proof reconstruction, and the development of a proof-producing axiom instantiation module.

Currently, proof reconstruction steps that involve algebraic normalization are done axiomatically. While Lean's simplifier is powerful and efficient, it does not have (very much) tooling for handling numerals, and is unable to, e.g., simplify `x + y + 2*x` to `3*x + y` or `2*x + y - x - x` to `y`. This type of simplification is needed for complete proof reconstruction in Polya, whether it is handled by Lean's built-in simplifier or by an independent module. The theory behind such a module is understood [52], but there are many intricacies in implementing it, and doing so fell outside the scope of this dissertation.

On the other hand, Lean provides support for heuristic lemma instantiation using its SMT module. With the ematching tooling provided there, we expect that a Lean implementation of the unverified axiom instantiation module will be fairly straightforward. As the SMT module matures to include other common components of SMT solvers, such as new theory provers and backtracking search guidance, we hope to take advantage of these features in Polya as well.

The Python version of Polya is somewhat more aggressive than the Lean version in its attempts to infer the signs of variables. It also contains basic support for case splitting on comparisons. Alongside an implementation of the axiom instantiation module, making these improvements should endow the Lean version with the same proving capabilities as the Python version.

Finally, there are many improvements that can be made to the efficiency of the Lean implementation. Some will be quite straightforward to implement. The work described in this dissertation is based on version 3.3.0 of Lean, but the latest development versions implement much more efficient versions of `rb_map` and `rb_set`. These data structures are used extensively in Polya, so we expect significant gains in efficiency by moving to a newer version of Lean.

Polya's largest efficiency problems involve computation with large rational numbers (during proof search) and an overreliance on Lean's elaborator (during proof reconstruction). The former problem is difficult to avoid, but can be somewhat ameliorated by bounding the size of denominators. The current approach fixes a bound once and for all, but it could be more efficient to start with a low bound and gradually raise it. The latter problem has a more obvious solution. At the moment, the library of lemmas that Polya uses for proof reconstruction makes extensive use of type classes, in order to state theorems generically over comparisons ($<, \leq, \geq, >$). The reconstruction process also forces some kernel computation of rational numbers, e.g. to confirm that $1/(-1)$ is definitionally equal to 1. Redesigning the library so that lemmas are stated more concretely, and so that equalities between coefficients are proved with the tactic `norm_num` rather than by definition, is likely to make a large dent in the time spent on proof reconstruction.

Chapter 4

A link between Lean and Mathematica[3]

4.1 Introduction

Many researchers have noted the disconnect between computer algebra and interactive theorem proving. In the former, one typically values speed and flexibility over absolute correctness. To be more efficient or user-friendly, a computer algebra system (CAS) may blur the distinction between polynomial objects and polynomial functions, assume that sufficiently small terms at the end of a series are zero, or resort to numerical approximations without warning. Such simplifying assumptions often make sense in the context of computer algebra; the capability and flexibility of these systems make them indispensable tools to many working mathematicians.

These assumptions, though, are antithetical to the goals of interactive theorem proving (ITP), where every inference must be justified by appeal to some logical principle. The strict logical requirements and lack of familiar algorithms discourage many mathematicians from using proof assistants. Conversely, the unreliability of many computer algebra systems, and their lack of proof languages and proof libraries, often makes them unsuitable for mathematical justification.

Integrating computer algebra and proof assistants is one way to reduce this barrier to entry to ITP and to strengthen the justificatory power of computer algebra. Bridges between the two types of systems have been built in a variety of ways. We contribute another such bridge, between the proof assistant Lean [50] and the computer algebra system Mathematica [138]. Since Mathematica is one of the most commonly used computer algebra systems, and a user with knowledge of the CAS can extend the capabilities of our link, we hope that the familiarity will lead to wider use.

Our connection is inspired by the architecture described by Harrison and Théry [79]. A number

[3] An early stage of the work described in this chapter appears in a paper published in Dubois and Paleo, eds., proceedings of Proof eXchange for Theorem Proving 2017 [95]. The ensuing work on connecting to Lean from within Mathematica was done in collaboration with Minchao Wu.

of features of our bridge distinguish it from earlier work. CAS results imported into the proof assistant can be trusted, verified, or used ephemerally; the translation can be extended in-line with library development, without modifying auxiliary dictionaries or source code; the link works bi-directionally using the same translation procedure, allowing Mathematica to access Lean's library and automation.

Our link separates the steps of communication, semantic interpretation, and verification: there is no a priori restriction on the type of information that can be shared between the systems. With the proof assistant in the "master" role, Lean expressions are exported to Mathematica, where they can be interpreted and manipulated. The results are then imported back into Lean and reinterpreted. One can then write scripts that verify properties of the translated results. This style of interaction, where verification happens on a per-case basis after the computation has ended, is called *ad hoc*.

By performing calculations in Mathematica and verifying the results in Lean, we relax neither the rigor of the proof assistant nor the efficiency of the CAS. Alternatively, we can trust the CAS as an oracle, or use it in a purely informative role, where its output does not appear in the final proof term. We provide comprehensive tactics to perform and verify certain classes of computations, such as factoring polynomials and matrices. But all the components of our procedure are implemented transparently in Lean's metaprogramming framework, and they can easily be extended or used for one-off computations from within the Lean environment.

This range of possibilities is intended to make our link attractive to multiple audiences. The working mathematician or mathematics student, who balks at the restrictions imposed by a proof assistant, may find that full access to a familiar CAS is worth the tradeoff in trust. Industrial users are often happy to trust both large-kernel proof assistants and computer algebra systems; the rigor of Lean with Mathematica as an oracle falls somewhere in between. And certifiable algorithms are still available to users who demand complete trust. The ease of metaprogramming in Lean is another draw: users do not need to learn a new programming or tactic language to write complicated translation rules or verification procedures.

The translation procedure used is symmetric and can be used for communication in the reverse direction as well. Mathematica has no built-in notion of proof, although it does have head symbols that express propositions. Rather than establishing an entire proof calculus for these symbols within Mathematica, we export theorem statements to Lean, where they can be verified in an environment designed for this purpose. The resulting proof terms are interpreted in the CAS and can be displayed or processed as needed. Alternatively, we can skip the verification step and display lemmas that are likely to be relevant to Mathematica's goal. In some sense, the link is allowing Mathematica to "borrow" Lean's semantics, proof language, and proof library.

The source for this project, and supplementary documentation, is available at
http://www.andrew.cmu.edu/user/rlewis1/leanmm/.

In this chapter, we use `Computer Modern` for Lean code and `TeX Gyre Cursor` for Mathematica code. We begin by describing salient features of the two systems. Section 4.3 discusses the translation of Lean expressions into semantically similar Mathematica expressions, and vice versa. Section 4.4 describes the interface for running Mathematica from Lean, and shows many examples of the link in action. Section 4.5 explains the reverse direction. We conclude with a discussion of related and future work.

4.2 Mathematica background

Mathematica is a popular symbolic computation system developed at Wolfram Research, implementing the Wolfram Language [137] [138]. First launched in 1988, Mathematica has gone through 11 release cycles. It now exists in an ecosystem of related projects, including Wolfram Alpha, a natural language frontend that provides easy access.

Mathematica provides comprehensive tools for rewriting and solving polynomial, trigonometric, and other classes of equations and inequalities; solving differential equations, both symbolically and numerically; computing derivatives and integrals of various types; manipulating matrices; performing statistical calculations, including fitting and hypothesis testing; visualizing and displaying plots and diagrams; and reasoning with classes of special functions. Along with support for a vast range of mathematical computations, Mathematica includes collections of data of various types and tools for manipulating this data.

This large library of functions is one reason to choose Mathematica for our linked CAS. Another reason is its ubiquity: Mathematica is frequently used in undergraduate mathematics and engineering curricula. Lean beginners who are accustomed to Mathematica do not need to learn a new CAS language for the advanced features of this link. Mathematica is sometimes regarded by mathematicians as a less "serious" CAS than some of its competitors, due to its relative focus on data, visualization, and interface over mathematical precision. For our purposes, though, these are not downsides. The design of our link means that we do not have to worry about potential soundness issues in the CAS; the visualization and interface tools make the ITP-from-CAS direction of our link easier and more interesting.

4.2.1 Mathematica syntax

The abstract syntax of the Wolfram Language is quite simple. An expression in Mathematica is either *atomic* or is an *application* of an expression to a list of expressions. There is no privileged notion of a binding expression, and expressions are untyped.

There are four types of atomic Mathematica expressions:

- *Integers* are arbitrary precision standard integers.

- *Reals* are floating point numbers.
- *Strings* are standard character strings. We write strings with double quotation marks, e.g. `"hello"`.
- *Symbols* are similar to strings, but serve a very different purpose in the language. We write symbols without quotation marks, e.g. `goodbye`.

Users accustomed to the Mathematica frontend may find this list surprisingly short. Many data types in Mathematica that appear atomic to the user are, in fact, compounds of these atomic types. The rational number `2/3`, for instance, is represented as an application of the symbol `Rational` to the integers `2` and `3`. Such applications are written using square brackets.

There is no restriction on the arity of applications. The expressions `Plus`, `Plus[]`, `Plus[x]`, and `Plus[x, y]` are all well formed. It is common to see a symbol (like `Plus`) applied to a list of expressions; in this situation, we refer to that symbol as the *head symbol* of the applications, and the following expressions as the *arguments*. We generally think of this pattern as function application, and will refer to it as such, but it is really purely structural: it is valid to write, e.g., `2[10, 3]`.

Mathematica supports common notation for many functions, such as `x + y + z + 2w`. It also supports postfix notation for unary application: `x^2 - 2x + 1 // Factor` is equivalent to `Factor[x^2 - 2x + 1]`. This syntax will appear frequently in this chapter.

There is no strong distinction between defined and undefined symbols. The user is free to introduce a new symbol and use it at will. The computational behavior of a head symbol can be fully or partially defined via pattern matching rules, such as `F[x_ ,y_] := x+y`; the underscores indicate that `x_` and `y_` are patterns. Mathematica evaluates expressions by repeatedly rewriting using these defined equalities. Performing computations in Mathematica amounts to evaluating expressions: to factor a polynomial, we evaluate `Factor[x^2 - 2x + 1]`. Evaluation is typically idempotent, with rare exceptions occurring when evaluation is controlled manually.

The Wolfram Language is untyped, so head symbols such as `Plus` and `Factor` can be applied to any argument or sequence of arguments. Evaluation is often restricted to certain patterns: `Plus[2, 3]` will evaluate to `5`, but `Plus[Factor, Plus]` will not reduce. Nevertheless, both are well formed Mathematica expressions.

4.2.2 Mathematica functions

Mathematica has a vast library of functions for various computations, mathematically or otherwise. Here, we give some details about functions that are useful for this project.

Algebraic functions. The head symbols `Plus` and `Times` represent the normal mixed-arity addition and multiplication operations on a commutative ring. `Plus` and `Times` are associative

and commutative, meaning that `Plus[Plus[x, y], z]` and `Plus[x, Plus[z, y]]` will both evaluate to `Plus[x, y, z]`. The behavior of the `Power` symbol is similar; applying `Power` to more than two arguments evaluates to an iterated binary application.

Evaluation control. The head symbol `Hold` does not have associated evaluation rules. Instead, its presence prevents the evaluation of its arguments: evaluating `Hold[Plus[2, 3]]` will return the same expression. Applying `ReleaseHold` recursively removes applications of `Hold`, so that `ReleaseHold[Hold[Plus[2, 3]]]` will evaluate to `5`. `Inactive`, an operator form of `Hold`, applies to head symbols. The expression `Inactive[Plus][2, 3]` will evaluate to itself, but `Inactive[Plus][Plus[1, 1], 3]` will evaluate to `Inactive[Plus][2, 3]`. Applying `Activate` removes applications of `Inactive` from its arguments. These functions are indispensable when treating Mathematica expressions as pieces of syntax rather than semantic objects, as otherwise Mathematica will eagerly evaluate all expressions.

Factor. The `Factor` function identifies the "variables" in an expression heuristically, by finding the largest non-algebraic parts, and factors the expression by treating it as a polynomial in these variables. This generality means that, if `a` and `b` are undefined symbols, Mathematica will factor `a[b]^2 - 2a[b] + 1` by treating `a[b]` as atomic.

FullSimplify. Basic evaluation rules, such as the associativity of `Plus`, are built into Mathematica by default. Others, particularly more expensive ones, can be triggered by applying `FullSimplify`. This head symbol intends to produce something semantically equal to its argument, by applying a large library of equalities and decision procedures. For instance, `Cos[x]^2 + Sin[x]^2` is fully evaluated, but `Cos[x]^2 + Sin[x]^2 // FullSimplify` evaluates to `1`. `FullSimplify` can be used to "prove" statements in trigonometry, calculus, algebra, and special function theory by reducing these statements to `True`.

FindInstance. Mathematica has many heuristics and decision procedures for solving systems of relations. `FindInstance` is a useful front end for these tools. Given a formula, a list of variables, and an optional domain, `FindInstance` will search for an assignment over the domain that satisfies the formula. For instance, evaluating

```
FindInstance[x^2 - 3 y^2 == 1 && 10 < x < 100, {x, y}]
```

will produce the solution `{x -> Sqrt[3073], y -> -32}`, and

```
FindInstance[x^2 - 3 y^2 == 1 && 10 < x < 100, {x, y}, Integers]
```

will produce `{x -> 26, y -> 15}`. Mathematica will attempt to detect the appropriate method do use, but the user may also specify a method. Among others, `FindInstance` can attempt to solve

problems in pure logic, linear and nonlinear integer, modular, real, complex, and mixed-domain arithmetic, and geometry.

4.3 The translation procedure

Our bridge can be used to import information from Mathematica into Lean, usually about some particular Lean expression. The logical foundations and semantics of the two systems are quite different, and we should not expect a perfect correspondence between the two. However, in many situations, an expression in Lean has a counterpart in Mathematica with a very similar intended meaning. We can exploit these similarities by ignoring the unsoundness of the translations in both directions and attempting to verify, post hoc, that the resulting expression has the intended properties.

As a running example, suppose we want to show in Lean that

```
x : real ⊢ x^2 - 2x + 1 ≥ 0.
```

Factoring the left-hand side of the inequality makes this a one-step proof (assuming we've proved that squares are nonnegative). It is nontrivial to write a reliable and efficient polynomial factoring algorithm, but luckily, one is implemented in Mathematica. Our tool allows us to do the following:

1. Transform the Lean representation of $x^2 - 2x + 1$ into Mathematica syntax.
2. Interpret this into the Mathematica representation of the same polynomial.
3. Use Mathematica's `Factor` function to factor the polynomial.
4. Transform this back into Lean syntax, and interpret it as a Lean polynomial.
5. Verify that the new expression is equal to the old.
6. Substitute this equality into the goal.

In Subsection 4.3.1 we describe steps 1, 2, and 4. Once we have a valid Mathematica expression, step 3 is trivial. We discuss steps 5 and 6 in Section 4.4; since checking that a polynomial has been factored correctly is much easier than factoring it in the first place, these are handled easily by simplification and rewriting.

It is worth emphasizing the modularity and extensibility of this approach. Both directions of translation are handled independently, and the translation rules can be extended or changed at will. Translation rules may be arbitrarily complex. Results from Mathematica are recieved with no *a priori* relation to the original Lean expression. (Indeed, there need not even be an original Lean expression—one could use Mathematica as a pseudorandom number generator, for example.) Users may choose to use alternate verification procedures, or to forgo the verification step entirely. The same translations are used for communication in the opposite direction, as described in Section 4.5.

4.3.1 Translating Lean to Mathematica

The Lean expression grammar is presented in Section 2.1.1. It is exposed in Lean's metaprogramming framework through the type `expr`. We can define functions to manipulate Lean expressions by recursion on this type; during execution, the virtual machine replaces terms of this type with objects of the kernel expression datatype.

To represent the expression grammar in Mathematica, we define

```
mathematica.form_of_expr : expr → string
```

by recursion over `expr`. We associate a Mathematica head symbol `LeanVar`, `LeanSort`, `LeanConst`, etc. to each constructor of `expr`. Names, levels, lists of levels, and binder information are also represented. (Binder information, which tracks whether a variable in a declaration is marked as implicit, explicit, or with some other properties, is very rarely relevant to the translation process.) The functions are displayed in Figure 4.1. Macro expressions are inherently internal to Lean, and it does not make sense to translate them to Mathematica; expressions containing macros should be unfolded before applying the translation.

Some of the information contained in a Lean expression has little plausible use in Mathematica, or is needlessly verbose: for example, it is hard to contrive a scenario in which the full structure of a Lean `name` is used in the CAS. Nonetheless, we do not strip any information at this stage, to preserve that an expression reflected into and immediately back from Mathematica should translate to the original expression without having to inject any additional information. For specific performance-critical applications, when this information is known to be unnecessary, users may wish to write specialized representation functions that do strip it away.

In our running example, we work on the expression $x^2 - 2x + 1$. The fully elaborated Lean expression and its Mathematica representation are too long to print here, but they can be viewed in the supplementary documentation. Instead, we consider the more concise example of $x + x$.

If we use strings to stand in for terms of type `name`, natural numbers in place of universe levels, and the string `"bi"` in place of the default `binder_info` argument, and we abbreviate

```
𝒳 := local_const "x" "x" "bi" (const "real" []),
```

the full form of $x + x$ is

```
app (app (app (app (const "add" [0]) (const "real" []))
                   (const "real.has_add" [])) 𝒳) 𝒳.
```

The corresponding Mathematica expressions are

```
X := LeanLocal["x", "x", "bi", LeanConst["real", {}]]

LeanApp[LeanApp[LeanApp[LeanApp[LeanConst["add", {0}],
            LeanConst["real", {}]], LeanConst["real.has_add", {}]], X], X].
```

```
meta def form_of_name : name → string
| anonymous         := "LeanNameAnonymous"
| (mk_string s nm)  := "LeanNameMkString[\"" ++ s ++ "\", " ++ form_of_name nm ++ "]"
| (mk_numeral i nm) := "LeanNameMkNum[" ++ to_string i ++ ", " ++ form_of_name nm ++
    "]"

meta def form_of_lvl : level → string
| zero         := "LeanZeroLevel"
| (succ l)     := "LeanLevelSucc[" ++ form_of_lvl l ++ "]"
| (max l1 l2)  := "LeanLevelMax[" ++ form_of_lvl l1 ++ "," ++ form_of_lvl l2 ++ "]"
| (imax l1 l2) := "LeanLevelIMax[" ++ form_of_lvl l1 ++ "," ++ form_of_lvl l2 ++ "]"
| (param nm)   := "LeanLevelParam[" ++ form_of_name nm ++ "]"
| (mvar nm)    := "LeanLevelMeta[" ++ form_of_name nm ++ "]"

meta def form_of_lvl_list : list level → string
| []       := "LeanLevelListNil"
| (h :: t) := "LeanLevelListCons[" ++ form_of_lvl h ++ ", " ++ form_of_lvl_list t ++
    "]"

meta def form_of_binder_info : binder_info → string
| binder_info.default := "BinderInfoDefault"
| implicit            := "BinderInfoImplicit"
| strict_implicit     := "BinderInfoStrictImplicit"
| inst_implicit       := "BinderInfoInstImplicit"
| other               := "BinderInfoOther"

meta def form_of_expr : expr → string
| (var i)                     := "LeanVar[" ++ (format.to_string (to_fmt i) options.mk)
                                   ++ "]"
| (sort lvl)                  := "LeanSort[" ++ form_of_lvl lvl ++ "]"
| (const nm lvls)             := "LeanConst[" ++ form_of_name nm ++ ", "
                                   ++ form_of_lvl_list lvls ++ "]"
| (mvar nm nm' tp)            := "LeanMetaVar[" ++ form_of_name nm ++ ", "
                                   ++ form_of_expr tp ++ "]"
| (local_const nm ppnm bi tp) := "LeanLocal[" ++ form_of_name nm ++ ", " ++
                                   form_of_name ppnm ++ ", " ++ form_of_binder_info bi
                                   ++ ", " ++ form_of_expr tp ++ "]"
| (app f e)                   := "LeanApp[" ++ form_of_expr f ++ ", " ++ form_of_expr e
                                   ++ "]"
| (lam nm bi tp bod)          := "LeanLambda[" ++ form_of_name nm ++ ", "
                                   ++ form_of_binder_info bi ++ ", "
                                   ++ form_of_expr tp ++ ", " ++ form_of_expr bod ++ "]"
| (pi nm bi tp bod)           := "LeanPi[" ++ form_of_name nm ++ ", "
                                   ++ form_of_binder_info bi ++ ", " ++ form_of_expr tp
                                   ++ ", " ++ form_of_expr bod ++ "]"
| (elet nm tp val bod)        := form_of_expr $ expand_let $ elet nm tp val bod
| (macro mdf mlst)            := "LeanMacro"
```

Figure 4.1: Representing Lean expression syntax in Mathematica.

Since the head symbols `LeanApp`, `LeanConst`, etc. are uninterpreted in Mathematica, this representation is not yet useful. We wish to exploit the fact that many Lean terms have semantically similar counterparts in Mathematica. For instance, the Lean constants `add` and `mul` behave similarly to the Mathematica head symbols `Plus` and `Times`; both systems have notions of application, although they handle the arity of applications differently; and Mathematica's concept of a "pure function" is analogous to lambda abstraction in Lean.

We thus define a translation function `LeanForm` in Mathematica that attempts to interpret the syntactic representation. Mathematica functions are typically defined using pattern matching. The `LeanForm` function, then, will look for familiar patterns (e.g. `add A h x y`, in Mathematica syntax) and rewrite them in translated form (e.g. `Plus[LeanForm[x], LeanForm[y]]`). Users can easily extend this translation function by asserting additional equations; a default collection of equations is loaded automatically.

For our factorization example, we want to convert Lean arithmetic to Mathematica arithmetic. Among other similar rules, we will need the following:

```
LeanForm[LeanApp[LeanApp[LeanApp[LeanApp[LeanConst["add",_],
  _], _], x_], y_]] := Inactive[Plus][LeanForm[x],LeanForm[y]]
```

Note that this pattern ignores the type argument and type class instance in the Lean term. These arguments are irrelevant to Mathematica and can be inferred again by Lean in the back-translation. We block Mathematica's computation with the `Inactive` head symbol; otherwise, Mathematica would eagerly simplify the translated expression, which can be undesirable. The function `Activate` strips these annotations, allowing reduction.

Numerals in Lean are type-parametric and are represented using the constants `zero`, `one`, `bit0`, and `bit1`. To illustrate, the type signature of the latter is

```
bit1 {u} : Π {A : Type u}, [has_add A] → [has_one A] → A → A
```

and the numeral 6 is represented as `bit0 (bit1 one)`; the type of this numeral is expected to be inferable from context. We can use rules similar to the above to transform Lean numerals into Mathematica integers:

```
LeanForm[LeanApp[LeanApp[LeanApp[LeanApp[LeanConst["bit1", _],
  _], _], _], t_]] := 2*LeanForm[t]+1.
```

Applying `LeanForm` will not necessarily remove all occurrences of the head symbols `LeanApp`, `LeanConst`, etc. This is not a problem: we only need to translate the "concepts" with equivalents in Mathematica. Unconverted subterms—for instance `X`, which contains applications of `LeanLocal` and `LeanConst`—will be treated as uninterpreted constants by Mathematica, and the back-translation described below will return them to their original Lean form.

```
inductive mmexpr
| sym   : string → mmexpr
| mstr  : string → mmexpr
| mint  : int → mmexpr
| app   : mmexpr → list mmexpr → mmexpr
| mreal : float → mmexpr
```

Figure 4.2: Mathematica expression kinds

In our running example (keeping the abbreviation `X`), applying the `LeanForm` and `Activate` functions produces the expression `Plus[1, Times[-2, X], Power[X, 2]]`. Applying `Factor` produces `Power[Plus[-1, X], 2]`.

4.3.2 Translating Mathematica to Lean

Mathematica expressions are composed of various atomic number types, strings, symbols, and applications, where one expression is applied to a list of expressions, as described in Section 4.2.1. We represent this structure in Lean with the data type `mmexpr` (Figure 4.2).

The result of a Mathematica computation is reflected into Lean as a term of type `mmexpr`. This is analogous to the original export of our Lean expression into Mathematica. It remains to interpret it as something meaningful.

A *pre-expression* in Lean is a term where universe levels and implicit arguments are omitted. It is not expected to type-check, but one can try to convert it into a type-correct term via elaboration. For instance, the pre-expression ``(add nat.one nat.one)` elaborates to `add.0 nat nat.has_add nat.one nat.one`. The notation ``(...)` instructs Lean's parser to interpret the quoted text as a term of type `pexpr`. Pre-expressions share the same structure as expressions.

Mathematica expressions are analogous to pre-expressions: they may be type-ambiguous and contain less information than their Lean counterparts. Thus we normally expect to interpret terms of type `mmexpr` as pre-expressions, and to use the Lean elaborator to turn them into full expressions. However, in rare cases an `mmexpr` may already correspond to a full expression: the unmodified representation of a Lean expression, sent back into Lean, should interpret as the original expression. We provide two extensible translation functions, `expr_of_mmexpr` and `pexpr_of_mmexpr`, to handle both of these cases. Since the implementations are similar, we focus on the latter here.

The function

```
pexpr_of_mmexpr : trans_env → mmexpr → tactic pexpr
```

takes a translation environment and an `mmexpr`, and, using the attribute manager, attempts to return a pre-expression. (Since the tactic monad includes failure, the process may also fail if no interpretation is found.) Interpreting strings as pre-expressions, or, indeed, as expressions, is straightforward. Since Mathematica integers may be used to represent numerals in many different

Lean types, expressions built with the `mint` constructor are interpreted as untyped numeral pre-expressions.

The `sym` and `app` cases are more complex: this part of the translation procedure is extensible by the user. We define three classes of translation rules:

- A sym-to-pexpr rule, of type `string × pexpr`, identifies a particular Mathematica symbol with a particular pre-expression. For example, the rule `("Real", ``(real))` instructs the translation to replace the Mathematica symbol `Real` with the Lean pre-expression `const "real"`.

- A keyed app-to-pexpr rule is of type `string × (trans_env → list mmexpr → tactic pexpr)`. When the procedure encounters an `mmexpr` of the form `app (sym head) args`—that is, the Mathematica head symbol `head` applied to a list of arguments `args`—it will try to apply all rules that are keyed to the string `head`. The rules for interpreting arithmetic expressions follow this pattern: a rule keyed to the string `"Plus"` will interpret `Plus[t₁, ..., tₙ]` by folding applications of `add` over the translations of `t₁` through `tₙ`.

- An unkeyed app-to-pexpr rule is of type `trans_env → mmexpr → list mmexpr → tactic pexpr`. If the head of the application is a compound expression, or if no keyed rules execute successfully, the translation procedure will try unkeyed rules. One such rule attempts to translate the head symbol and arguments independently, and fold application over these translations. Another removes instances of the symbol `Hold`, which blocks evaluation of sequences of expressions. The Lean translation of `Plus[Hold[x, y, z]]` should reduce to the translation of `Plus[x, y, z]`, but since `Hold[x, y, z]` translates to a sequence of expressions, this does not match either of the previous rule types.

Rules of these three types can be declared by the user and tagged with the corresponding attribute. The translation procedure uses Lean's caching attribute manager to collect relevant rules at runtime. The mechanism for extending the translation procedure is thus integrated into theory development; translation rules are first-class members of mathematical libraries, and any project importing a library will automatically have access to its translation rules.

Returning to our example, we have translated the expression `x^2 - 2x + 1` and factored the result, to produce `Power[Plus[-1, X], 2]`. This is reflected as the Lean `mmexpr`

```
app (sym "Power") [app (sym "Plus") [mint -1, X], mint 2],
```

where again

```
X := app (sym "LeanLocal") [str "17.27", str "x", str "bi",
                            app (sym "LeanConst") [str "real", []]].
```

Applying `pexpr_of_mmexpr` produces the pre-expression `pow_nat (add (neg one) x) (bit0 one)`, which elaborates to the expression

```
pow_nat real real_has_pow_nat (add real real_has_add (neg real real_has_neg (one real
    real_has_one) x) (bit0 nat nat_has_add one nat nat_has_one) : real.
```

Formatted with standard notation and implicit arguments hidden, we have constructed the term `x : real ⊢ (x + -1)^2 : real` as desired.

4.3.3 Translating binding expressions

Lean's expression structure uses anonymous bound variables to implement its `pi`, `lam`, and `elet` binder constructs. Mathematica, in contrast, has no privileged notion of a binder. The Lean pre-expression `λ x, x + x` is analogous to the Mathematica expression `Function[x, x+x]`, but the underlying representation of the latter is an application of the `Function` head symbol to two arguments, the symbol `x` and the application expression `Plus[x, x]`. Structurally it is no different from `List[x, x+x]`.

To properly interpret binder expressions, both translation routines need a notion of an environment. We extend the Mathematica function `LeanForm` with another argument, a list of symbols `env` tracking binder depth. When the translation routine encounters a binding expression, it creates a new symbol, prepends it to the `env`, and translates the binder body under this extended environment; a bound variable `LeanVar[i]` is interpreted as the ith entry in `env`.

In the opposite translation direction, a translation environment is a map from strings (names of symbols) to expressions, that is, `trans_env := rb_map string expr`. When translating a Mathematica expression such as `Function[x, x+x]`, the procedure extends the environment by mapping `x` to a placeholder variable, translates the body under this extended environment, and then abstracts over the placeholder. Unlike in Lean, where `pi`, `lam`, and `elet` expressions are the only expressions that encode binders, there are many Mathematica head symbols (e.g. `Function`, `Integrate`, `Sum`) that must be translated this way.

4.4 Querying Mathematica from Lean

4.4.1 Connection interface

Because of the cost of launching a new Mathematica kernel, it is undesirable to do so every time Mathematica is queried from Lean. Instead, we implement a simple server in Mathematica, which receives requests containing expressions and returns the results of evaluating these expressions. Lean communicates with this server by calling a Python client script. This short script is the only part of the link that is implemented neither in Lean nor in Mathematica.

This architecture ensures that a single Mathematica kernel will be used for as long as possible, across multiple tactic executions and possibly even multiple Lean projects. To preserve an illusion of "statelessness," each Mathematica evaluation occurs in a new context which is immediately

cleared. While this avoids accidental leaks of information, it is not a watertight seal, and users who consciously wish to preserve information between sessions can do so.

The translation procedure is exposed in Lean using the tactic framework via the declaration

```
meta def mathematica.execute : string → tactic mmexpr.
```

This tactic evaluates the input string in Mathematica, and returns a term with type `mmexpr` representing the result of the computation. From this basic tactic, it is easy to define variants such as

```
run_command_using : (string → string) → expr → string → tactic pexpr.
```

The first argument is a Mathematica command, including a placeholder bound variable, which is replaced by the Mathematica representation of the `expr` argument. The `string` argument is the path to a file which contains auxiliary definitions, usable in the command. This variant will apply the back-translation `pexpr_of_mmexpr` to produce a `pexpr`.

Another variant, `execute_global : string → tactic mmexpr`, evaluates its input in Mathematica's global context.

Going back to our running example from Section 4.3, assuming `e` is the unfactored expression, we would call

```
run_command_on (λ s, s ++ " // LeanConvert // Activate // Factor") e
```

to produce a pre-expression representing the factored form of `e`. (Recall that the Mathematica syntax `x // f` reduces to `f[x]`.) In fact, we can define

```
meta def factor (e : expr) : tactic pexpr :=
run_command_on (λ s, s ++ " // LeanConvert // Activate // Factor") e,
```

or a variant that elaborates the result into an `expr` with the same type as `e`.

4.4.2 Verification of results

So far we have described how to embed a Lean expression in Mathematica, manipulate it, and import the result back into Lean. At this point, the imported result is simply a new expression: no connection has been established between the original and the result. In our factoring example, we expect the two expressions to be equal; if we were computing an antiderivative, we would expect the derivative of the result to be equal to the original. More complex return types can lead to more complex relations. For example, an algorithm using Mathematica's linear arithmetic tools to verify the unsatisfiability of a system of equations may return a certificate that must be converted into a proof of falsity.

Users may simply decide to trust the translation and CAS computation, and assert without proof that the result has an expected property. An example using this approach is given at the end of this section. Of course, the level of trust needed to do this is unacceptably high for many

situations. We are often interested in performing *certifiable* calculations in Mathematica, and using this certificate to construct proofs in Lean.

It would be hopeless to expect one tool to verify all results. Rather, for each common computation, we will have a tactic script to (attempt to) prove the appropriate relation between input and output. "Uncommon" or one-off computations can be verified in-line by the user. This method of separating search (or computation) and verification is discussed at length by Harrison and Théry [79] and by many others. It turns out that a surprising number of algorithms are able to generate certificates to this end.

The tactics used in this section, along with more examples, are available in the supplementary information online. These examples are not meant to be exhaustive, but rather to illustrate the ease with which Mathematica can be accessed; with the possible exception of the linear arithmetic tactic, each is fairly simple to implement. The Lean library is still under development, and some types and functions used here are in fact axiomatized constants, but the implementation of these constants is not relevant to the behavior of our link.

Factoring. In our running example, we have used Mathematica to construct the Lean expression `(x + -1)^2 : real`. We expect to find a proof that `x^2 - 2*x + 1 = (x + -1)^2`. This type of proof is easy to automate with Lean's simplifier:

```
meta def eq_by_simp (l r : expr) : tactic expr :=
do gl ← mk_app `eq [l, r],
   mk_inhabitant_using gl simp <|> fail "unable to simplify"
```

Using this machinery, we can easily write a tactic `factor` that, given a polynomial expression, factors it and adds a constant to the local context asserting equality. (The theorem `sq_nonneg` proves that the square of a real number is nonnegative.)

```
example (x : ℝ) : x^2-2*x+1 ≥ 0 :=
by factor x^2-2*x+1 using q; rewrite q; apply sq_nonneg
```

We provide more examples of this tactic in action in the supplementary material, including one in which `x^10-y^10` factors into

```
(x + -1 * y) * (x + y) * (x^4 + -1 * x^3 * y + x^2 * y^2 + -1 * x * y^3 + y^4) * (x^4 +
    x^3 * y + x^2 * y^2 + x * y^3 + y^4).
```

In general, factoring problems are easily handled by this type of approach, since the results serve as their own certificates. Factoring integers is a simple example of this (to verify, simply multiply out the prime factors); dually, primality certificates can be checked as in Pratt [120].

Factoring matrices is slightly more complex. Mathematica implements a number of common matrix decomposition methods, whose computation can be verified in Lean by re-multiplying the factors. We can use these tools to, e.g., define a tactic `lu_decomp` which computes and verifies the LU decomposition of a matrix.

```
example : ∃ l u, is_lower_triangular l ∧ is_upper_triangular u
                 ∧ l ** u = [[1, 2, 3], [1, 4, 9], [1, 8, 27]] :=
by lu_decomp
```

Solving polynomials. Mathematica implements numerous decision procedures and heuristics for solving systems of equations. Many of these are bundled into its `Solve` function. Over some domains, it is possible to verify solutions in Lean using the simplifier, arithmetic normalizer, or other automation. Lean's `norm_num` tactic, which reduces arithmetic comparisons between numerals, is well suited to verifying solutions to systems of polynomial equations. The tactic `solve_polys` uses `Solve` and `norm_num` to prove theorems such as

```
example :  ∃ x y : ℝ, 99/20*y^2 - x^2*y + x*y = 0
                      ∧ 2*y^3 - 2*x^2*y^2 - 2*x^3 + 6381/4 = 0 :=
by solve_polys.
```

Users familiar with Mathematica may recall that `Solve` outputs a list of lists of applications of the `Rule` symbol, each mapping a variable to a value. Each list of rules represents one solution. A `Rule` has no general correspondent in Lean, and it would involve some contortion to translate this output and extract a single solution in the proof assistant. However, it is easy to perform this transformation within Mathematica, and processing the result of `Solve` *before* transporting it back to Lean makes the procedure much simpler to implement. This type of consideration appears often: some transformations are more easily achieved in one system or the other.

Linear arithmetic. Many proof assistants provide tools for automatically proving linear arithmetic goals, or equivalently for proving the unsatisfiability of a set of linear hypotheses. There are various techniques for doing this, including building proof terms incrementally using Fourier–Motzkin elimination [136]. Alternatively, linear programming can be used to generate certificates of unsatisfiability. In this setting, a certificate for the unsatisfiability of $\{p_i(\bar{x}) \leq 0 : 0 \leq i \leq n\}$ is a list of rational coefficients $\{c_i : 0 \leq i \leq n\}$ such that $\sum_{0 \leq i \leq n} c_i \cdot p_i = q > 0$ for some constant polynomial q; equivalently, this list serves as a witness for Farkas' lemma [122]. A slight generalization of Farkas' lemma that allows equalities and strict inequalities is known as the Motzkin transposition theorem.

Given a set of hypotheses in Lean that express linear inequalities, we can prove their unsatisfiability by generating a list of such coefficients (in Mathematica), automatically proving (in Lean) that these coefficients have the necessary properties, and applying a verified proof of the Motzkin transposition theorem.

While passing a list of inequalities to Mathematica may seem different from passing an expression such as `x^2 - 2*x + 1`, we are able to use the same translation procedure. The expression `x + 1 ≤ 2*y` has type `Prop`, which is to say it is a type living in the lowest universe level `Sort 0`. A

term of this type is a proof of the claim `x + 1 ≤ 2*y`. In our factorization example, we translated a term of type `real`, whereas here we translate the *type* of a hypothesis. But in dependent type theory, types are terms themselves, and we are able to represent any term in Mathematica. In Lean we define

```
le {u} : Π {A : Type u} [has_le A], A → A → Prop.
```

We reduce this in Mathematica using the rule

```
LeanForm[LeanApp[LeanApp[LeanApp[LeanApp[LeanConst["le", _],
         _], _], x_], y_]] :=
Inactive[LessEqual][x, y]
```

and define similar rules for $<$, $\geq$, $>$, and $=$.

Once the hypotheses have been translated to Mathematica, we must set up and solve the appropriate linear program. (Note that we are not trying to solve the hypotheses as given, but rather to find a certificate of their unsatisfiability.) A program provided in the supplementary materials to this paper shows how to use the Mathematica function `FindInstance` to produce the desired list of rational coefficients. This list is translated back to Lean, where it can be elaborated with type `list rat`. Once this list is confirmed to meet the requirements of Farkas' lemma, the lemma is applied to produce a proof of false.

```
example (x y : ℝ) (h1 : 2*x + 4*y ≤ 4) (h2 : -x ≤ 1) (h3 : -y ≤ -5) : false :=
by not_exists_of_linear_hyps h1 h2 h3
```

Sanity checking. Even non-certifiable computations can sometimes be useful for proof assistant users. Mathematica's `FindInstance` function, for example, can sometimes be used to check the plausibility of a goal. We define a tactic `sanity_check`, which fails if Mathematica is able to find a variable assignment that satisfies the local hypotheses and the negation of the current goal. This tactic is similar to a very lightweight version of Isabelle's Nitpick [28]. The first example below fails when Mathematica decides that the goal does not follow; the second succeeds.

```
example (x : ℝ) (h1 : sin x = 0) (h2 : cos x > 0) : x = 0 :=
by sanity_check; admit

example (x : ℝ) (h1 : sin x = 0) (h2 : cos x > 0)
        (h3 : -pi < x ∧ x < pi) : x = 0 :=
by sanity_check; admit
```

Axiomatized computations. Since it is possible to declare axioms from within the Lean tactic framework, we can axiomatize the results of Mathematica computations dynamically. This allows us to access a wealth of information within Mathematica, at least when we are not concerned about

complete verification. One interesting application is to query Mathematica for special function identities. While these identities may be difficult to formally prove, trusting Mathematica allows us to find some middle ground. The `prove_by_full_simp` tactic uses Mathematica's `FullSimplify` function to reduce the Bessel function expression on the left, and after checking that it is equal to the one on the right, adds this equality as an axiom in Lean:

```
example : ∀ x, x*BesselJ 2 x + x*BesselJ 0 x = 2*BesselJ 1 x :=
by prove_by_full_simp
```

We can also define a tactic that uses Mathematica to obtain numerical approximations of constants, and axiomatizes bounds on their accuracy:

```
approx (100 * BesselJ 2 (13 / 25)) (0.001 : ℝ)
```

declares an axiom stating that

```
75977 / 23000 < 100 * BesselJ 2 (13 / 25) ∧ BesselJ 2 (13 / 25) < 76023 / 23000.
```

4.5 Querying Lean from Mathematica

The scope of use of computer algebra in mathematics is largely limited to exploration and discovery. Finished proofs often avoid using these tools to justify claims, or even fail to mention them entirely. Outside of a few very specific domains, systems like Mathematica have no internal notion of proof or correctness. There are many documented instances of bugs and unexpected behavior in computer algebra systems [17], making concerns about this black-box nature more than just theoretical. Even the semantics for certain computations can be vague; reducing `(x^2 - 1)/(x - 1)` to `x + 1` is correct when considered as polynomial division, but computer algebra systems use this same notation to refer to a function of `x`.

Integrating a proof system into a mature CAS such as Mathematica is an enormous engineering task. A more realistic approach is to use a translation procedure to "borrow" a proof language and semantics from a proof assistant, on translatable domains. A proposition relating the input and output of a CAS evaluation can be exported to and proved in the proof assistant, which can return a proof term. This is morally similar to the ad hoc verification described in the previous section; while no general guarantee is claimed, individual computations can be checked.

4.5.1 Connection interface

We establish a connection to Lean from Mathematica using Mathematica's external command interface `RunProcess`. It is not necessary to replicate the server–client architecture of Section 4.4.1, since the cost of launching Lean is low. Lean already implements a server for communication with its editors, and using this interface would be particularly useful for applications requiring a

persistent, changing environment. However, this does not apply to the applications described here, so we have opted for the simpler approach.

We implement a function `ProveUsingLeanTactic[e_, tac_String]` which takes an arbitrary Mathematica expression `e`. It exports this expression to Lean and interprets it as a proposition. If successful, it attempts to prove the proposition using the tactic script `tac`, and returns the resulting proof term to Mathematica. While the tactic script may be arbitrarily complex, it will often be just a proof by `simp`, by `eblast`, or by other general-purpose automation.

A similar function, `ProveInteractively`, opens an interactive Lean session with the translation of `e` as the goal. When the session is terminated, the proof is checked and returned to Mathematica.

More generally, the function `RunLeanCommand[e_, cmd_String]` will evaluate a given Lean command on an expression `e` and return the result if successful. This is useful when the desired output is not a proof term.

4.5.2 Applications

Interpreting propositional proofs. Mathematica's built-in `TautologyQ` and `FullSimplify` functions serve as complete SAT solvers. However, both are black boxes: they produce no certificate or justification. Indeed, the system has no established proof language for propositional logic. On the other hand, Lean comes equipped with a number of proof-producing decision procedures for this domain. (For this example, we use `intuit`, as it produces proofs containing few extra constants.)

We define a minimal propositional proof calculus in Mathematica that mirrors the calculus in Lean. That is, we introduce head symbols `AndIntro`, `OrIntroLeft`, `FalseElim`, etc., and add `LeanForm` translation rules that map Lean's `and.intro`, `or.inl`, `false.elim`, etc. to their corresponding symbols. We can then state a propositional theorem in Mathematica, prove it in Lean, and interpret the resulting proof term in our calculus. While it would certainly be possible to implement the Lean proof search procedure in Mathematica directly, this approach ensures that the proof is correct, as it has been checked by Lean.

The resulting Mathematica proof object can be computed with in any number of ways. We implement a function which displays the proof as a natural deduction diagram, as in Figure 4.3. There is no fundamental reason why this approach cannot be extended to richer logics, such as first-order logic; the difficulties lie in representing a calculus for these logics in Mathematica, and generating proofs in Lean that can be translated to such a calculus. (Many proof tools in Lean use higher-order constructs that may be difficult to directly translate.)

Displaying significant proof steps. Interpreting arbitrary proofs in Mathematica may be too much to ask for, as the target language and translation rules may become arbitrarily complicated. An easier task is to interpret lemmas or relevant steps used to produce a proof. Lean's simplifier,

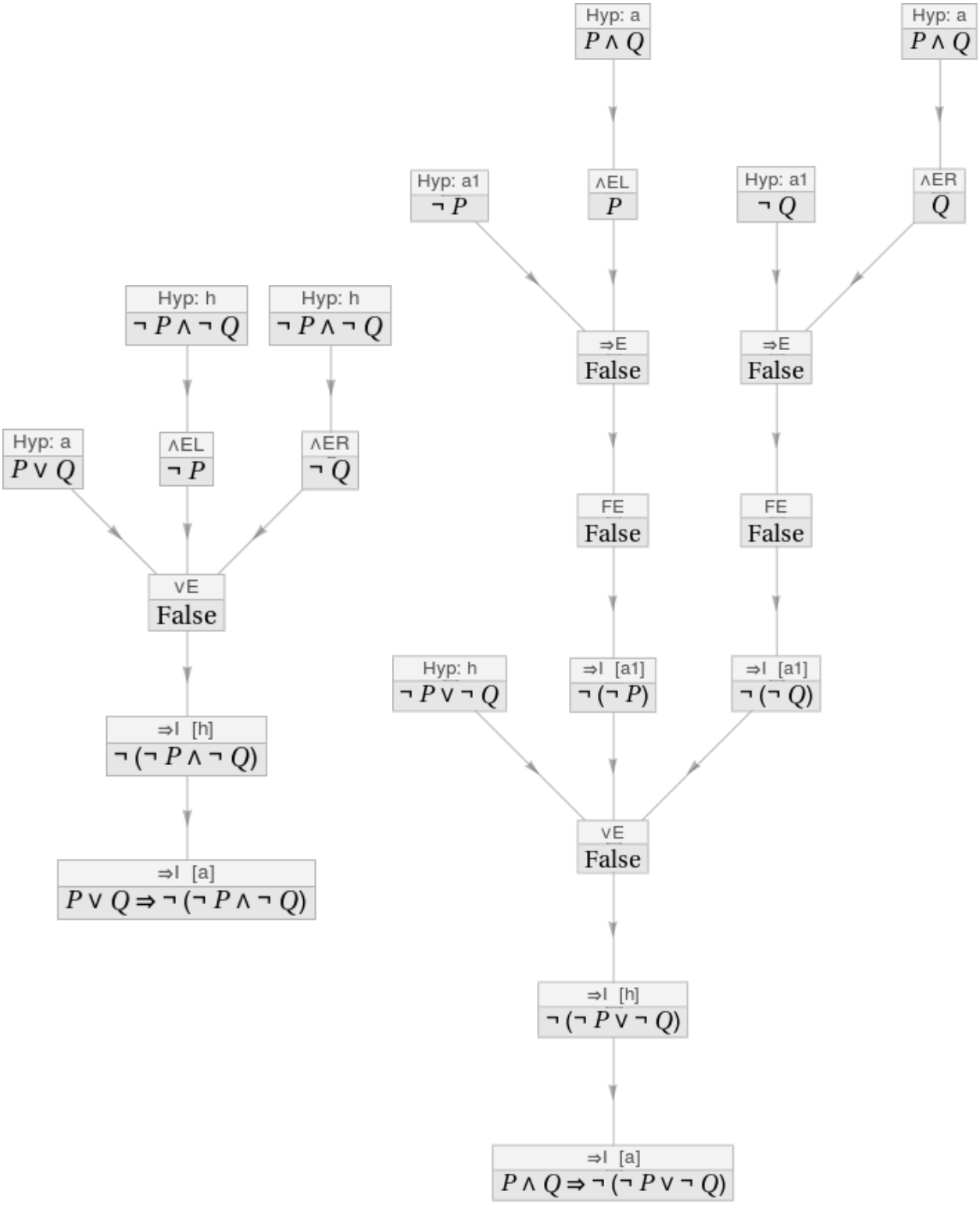


Figure 4.3: Natural deduction diagrams generated from Lean proof terms

heuristic quantifier instantiation procedure, and other general-purpose proof tactics search for lemmas in the Lean library to solve a goal. It is possible to inspect the proof terms generated by these tactics and extract theory lemmas, or in some cases, to implement versions of these tactics that produce a list of lemmas used. The types of the instances of these lemmas appearing in a proof term can be interpreted in Mathematica and displayed. Finding all and only the "interesting" lemmas is a difficult and poorly specified problem, but it is reasonable to implement a first-pass heuristic.

As an example, we do so in the context of set normalization. Mathematica has no built-in handling for arbitrary sets, but proofs of propositions such as $A \cap (B \cup \bar{A}) = A \cap B$ are easily found with Lean's simplifier. Noting that the relevant lemmas used by `simp` state that `A ∩ (B ∪ -A) = (A ∩ B) ∪ (A ∩ -A)`, `A ∩ -A = ∅`, and `(A ∩ B) ∪ ∅ = A ∩ B`, we can return these lemmas to Mathematica and display them as a "proof sketch." Note that there is no need to add translation rules for these lemmas themselves; alignments between the constants for union, intersection, complement, and equality are enough. This limits the need for a long list of translations and makes the procedure relatively robust to the introduction of new simplifier rules.

A similar application involves the use of a relevance filtering algorithm. Given a target expression, such an algorithm will return a list of declarations in the environment that, heuristically, may be useful to prove the target. Both symbolic and probabilistic relevance filters have been implemented in other systems, and are used for lemma selection for tools such as Isabelle's Sledgehammer [31]. We have implemented a rudimentary relevance filter in Lean. Using this tool, one can state a conjecture in Mathematica and receive a list of facts that may be of use to prove it, without depending on automation in Lean to actually find a proof.

A soft typing system for Mathematica. The Wolfram Language is untyped, and its notion of a well formed expression is very permissive. Unexpected evaluation, and the lack of expected evaluation, due to "ill-typed" expressions are common issues for Mathematica users, and can be difficult to diagnose. There is interest and preliminary work in establishing a soft typing system on some subset of Mathematica functions, particularly the mathematically oriented functions. Such a system would give a better notion of meaningful Mathematica expressions; by exploiting the Curry–Howard correspondence, a sufficiently strong type system could also provide an internal proof language.

In order to be usable in practice, such a type system requires algorithms for type checking and elaboration. If we construct a type system in Mathematica that is compatible with Lean's, we can avoid reimplementing these components by using those in the ITP. Indeed, a prototype presented at the 2016 Wolfram Technology Conference [43] implements a version of the calculus of constructions in Mathematica, along with a rudimentary type checker. We establish translation rules between this implementation and Lean, which allow us to elaborate Mathematica "pre-expressions" in the proof assistant and type-check the resulting terms in either system. The soft typing system is still at an early stage, and will certainly change in the future, but by maintaining this connection, we

will reduce the number of auxiliary tools needed to make the type system useful.

4.6 Concluding thoughts

4.6.1 Related work

The following discussion is not meant to be comprehensive, but rather to indicate the many ways in which one can approach connecting ITP and computer algebra.

Harrison and Théry [79] describe a "skeptical" link between HOL and Maple that follows a similar approach to our bridge. Computation is done in a standard, standalone version of the CAS, and sent to the proof assistant for certification. The running examples used are factorization of polynomials and antiderivation. The discussion is accompanied by an illuminating comparison of proof search to proof checking, and the relation to the class NP. Delahaye and Mayero [53] provide a similar link between Coq and Maple, specialized to proving field identities. Both projects tackle only the scenario in which the proof assistant drives the CAS.

Ballarin and Paulson [19] provide a connection between Isabelle and the computer algebra library Σ^{IT} [34] that is more trusting than the previous approach. They distinguish between sound and unsound algorithms in computer algebra: roughly, a sound algorithm is one whose correctness is provable, while an unsound algorithm may make unreasonable assumptions about the input data. Their link accepts sound algorithms in the CAS as oracles. A similarly trustful link between Isabelle and Maple, by Ballarin, Homann, and Calmet [18], allows the Isabelle user to introduce equalities derived in the CAS as rewrite rules. A third example by Seddiki, Dunchev, Khan-Afshar, and Tahar [123] connects HOL Light to Mathematica via OpenMath, introducing results from the CAS as HOL axioms.

A related, more skeptical, approach is to formally verify CAS algorithms and incorporate them into a proof assistant via reflection. This approach is taken by Dénès, Mörtberg, and Siles [54], whose CoqEAL library implements a number of algorithms in Coq.

Kerber, Kohlhase, and Sorge [92] describe how computer algebra can be used in proof assistants for the purpose of proof planning. They implement a minimal CAS, which is able to produce high-level sketch information. This sketch can be processed into a proof plan, which can be further expanded into a detailed proof.

Alternatively, one can build a CAS inside a proof assistant without reflection, such that proof terms are carried through the computation. Kaliszyk and Wiedijk [90] implement such a system in HOL Light, exhibiting techniques for simplification, numeric approximation, and antiderivation.

Going in the opposite direction, CAS users may want to access ATP or ITP systems. One example of a link in this direction is Adams et al. [4], who use PVS to verify side conditions generated in computations in Maple; Gottliebsen, Kelsey, and Martin [68] make use of similar ideas. Systems such as Analytica [22] and Theorema [35] provide ATP- or ITP-style behavior from within

Mathematica. Axiom [47] and its related projects provide a type system for computer algebra, which is claimed to be "almost" strong enough to make use of the Curry–Howard correspondence.

4.6.2 Future work

There is much room for an improved interface under the current ITP–CAS relationship. We imagine a link integrated with Lean's supported editors, where the user effectively has access to the Mathematica read–evaluate–print loop augmented by the current Lean environment. The REPL is a standard way of interacting with computer algebra systems, and contributes to their utility in exploration and discovery. Allowing this kind of interaction within Lean would greatly advance the goals of this project.

The server interface described in Section 4.4.1 only supports sequential evaluation of Mathematica commands. Both systems support parallel computation, and integrating the two could increase the utility of this link for large projects. Similarly, the physical connection between Mathematica and Lean can be strengthened by communicating with a Lean server. This would avoid the (small) cost of launching many instances of Lean, and would allow the possibility of maintaining state.

With the exception of the server running in Mathematica, the components of this link are generally adaptable to other computer algebra systems. More broadly, we see this project as part of a general trend. The various computer-based tools used in mathematical research, by and large, are independent of each other. It requires quite a lot of copying, pasting, and translating to (for example) compute an expression in Magma, verify its side conditions in Z3, visualize results in Mathematica, and export relevant formulas to LaTeX. Unified frameworks have been proposed and implemented [121], but are not widely used. Because they provide a strict logical foundation, precise semantics, and possibility of verification, proof assistants are strong candidates to center translation networks between systems.

Chapter 5

Conclusion

This dissertation has presented two tools that aim to support the formalization of mathematics. There is still a long way to go before the average mathematician is willing to pick up a proof assistant and start formalizing, but very slowly, the day is getting closer. The difficulties of formalization present themselves differently at various stages of proof, and for wider adoption, it is necessary to address all of these stages.

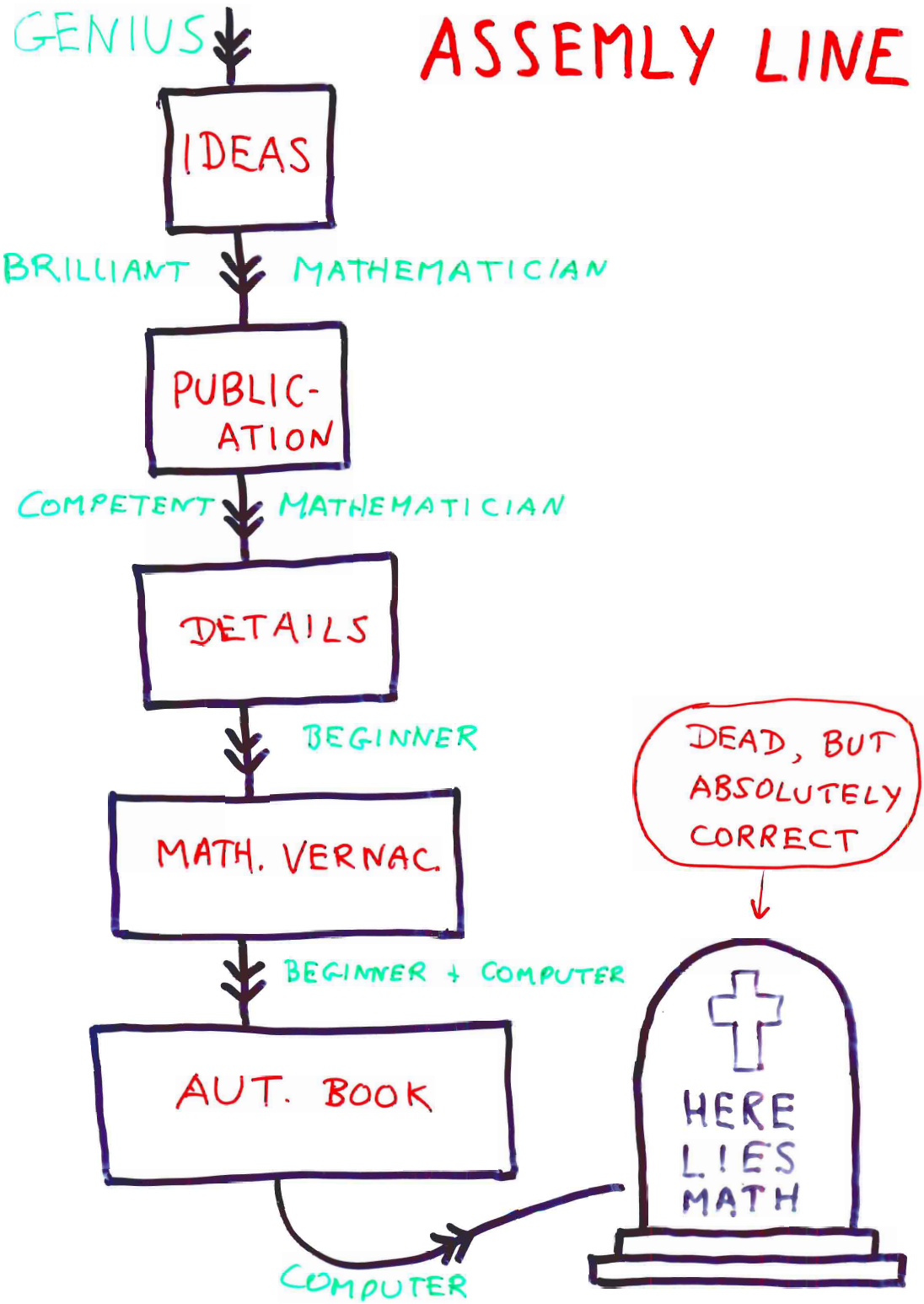
Simple proofs, the kind that mathematicians barely see as mathematical problems, are perhaps the most visible barrier. Outside of a few specific undergraduate lectures, nobody recognizes a difference between the rational number 3 and the real number 3, or between a vector of length $n + 1$ and one of length $1 + n$; these distinctions matter to a proof assistant, and sometimes create issues that are frustrating for beginners to diagnose. Similarly, converting between $\Sigma_{i=0}^{n} f(i)$ and $\Sigma_{i=1}^{n+1} f(i-1)$, or moving between representations of the rational numbers, can take more effort than intuition might suggest. The intricacies of dependent types can lead newcomers down paths of hopeless confusion. Many of these problems can be addressed with library design and straightforward automation, such as Lean's simplifier, and in some sense they appear more dire than they really are. Nevertheless, they look frightening enough to quickly scare away mathematical users. A proof assistant that is able to handle these situations truly transparently will be much more palatable.

Mid-level proofs, the type of problem attacked by the tools in this dissertation, are another challenge. There are many kinds of mathematical processes—solving inequalities, computing derivatives and integrals, rewriting combinatorial expressions—that fall between "trivial" and "hard." It is perhaps this level of difficulty at which proof assistants have the most to offer. Lean's metaprogramming framework allows simple proof methods to be integrated into its libraries, so that tools for approaching these problems are accessible and extensible. A proof step summarized in a paper as "integrating by parts" or "by induction and arithmetic" could be formalized in as many words, with the right tactics available. It is a substantial task to develop these tactics, and an interesting line of research to determine what sorts of processes will be most helpful for which fields of

mathematics. We hope that the contributions of this dissertation make some movement in this direction.

Perhaps the biggest difficulty in formalizing high-level proofs, proofs of the sort that appear in research mathematics, is in library development. In recent large-scale formalization efforts, much of the work involved has been in forming background theories that are robust enough to build up to the desired result. Of course, designing such a library requires one to address the issues above, and for mathematicians who wish to start using a proof assistant in their work, the lack of libraries is a major deterrent. In addition, high-level automated proof search is generally quite weak; while hammer tools try to automate lemma search and logical manipulations, they do not help much with the creative proof steps that are the bread and butter of human mathematics.

Progress is being made at all of these levels, and perhaps one day, it will reach a critical point. However, it was not just development in usability that convinced academics to write papers in LaTeX; it was the incredible benefit that the system provided over all alternatives. Alongside improvements in the areas described here, emphasizing the upsides of formalized proofs is a path toward wider adoption. Whether for educational purposes [16], for simplifying the refereeing process [72], or for avoiding (sometimes catastrophic) mistakes in reasoning [133], if mathematicians see sufficient value in the use of proof assistants, they will learn to formalize.


ASSEMLY LINE
GENIUS
IDEAS
BRILLIANT MATHEMATICIAN
PUBLIC-ATION
COMPETENT MATHEMATICIAN
DETAILS
BEGINNER
MATH. VERNAC.
BEGINNER + COMPUTER
AUT. BOOK
COMPUTER
DEAD, BUT ABSOLUTELY CORRECT
HERE LIES MATH

Bibliography

[1] Automath archive. http://www.win.tue.nl/automath/.

[2] The Lean Theorem Prover (website). https://leanprover.github.io/.

[3] The QED manifesto. In *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*, pages 238–251, 1994.

[4] A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '01, pages 27–42, London, UK, UK, 2001. Springer-Verlag.

[5] B. Akbarpour and L. C. Paulson. MetiTarski: An Automatic Prover for the Elementary Functions. In *AISC/MKM/Calculemus*, pages 217–231, 2008.

[6] K. Appel, W. Haken, et al. Every planar map is four colorable. part i: Discharging. *Illinois Journal of Mathematics*, 21(3):429–490, 1977.

[7] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. The Matita interactive theorem prover. In *CADE*, volume 6803, pages 64–69. Springer, 2011.

[8] J. Avigad. Mathematical method and proof. *Synthese*, 153(1):105–159, 2006.

[9] J. Avigad, L. de Moura, and S. Kong. Theorem proving in Lean. *https://leanprover.github.io/theorem_proving_in_lean*, 2016.

[10] J. Avigad, K. Donnelly, D. Gray, and P. Raff. A formally verified proof of the prime number theorem. *ACM Trans. Comput. Logic*, 9(1):2, 2007.

[11] J. Avigad, G. Ebner, and S. Ullrich. The Lean reference manual. *https://leanprover.github.io/reference/*, 2017.

[12] J. Avigad and H. Friedman. Combining decision procedures for the reals. *Log. Methods Comput. Sci.*, 2(4):4:4, 42, 2006.

[13] J. Avigad, J. Hölzl, and L. Serafin. A formally verified proof of the central limit theorem. *CoRR*, abs/1405.7012, 2014.

[14] J. Avigad, R. Y. Lewis, and C. Roux. A heuristic prover for real inequalities. In G. Klein and R. Gamboa, editors, *Interactive Theorem Proving*, volume 8558 of *Lecture Notes in Computer Science*, pages 61–76, 2014.

[15] J. Avigad, R. Y. Lewis, and C. Roux. A heuristic prover for real inequalities. *Journal of Automated Reasoning*, 2016.

[16] J. Avigad, R. Y. Lewis, and F. van Doorn. Logic and proof. *http://avigad.github.io/logic_-and_proof/*, 2017.

[17] D. H. Bailey, J. M. Borwein, V. Kapoor, and E. W. Weisstein. Ten problems in experimental mathematics. *American Mathematical Monthly*, 113(6):481–509, 2006.

[18] C. Ballarin, K. Homann, and J. Calmet. Theorems and algorithms: An interface between Isabelle and Maple. In *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation*, ISSAC '95, pages 150–157, New York, NY, USA, 1995. ACM.

[19] C. Ballarin and L. C. Paulson. A pragmatic approach to extending provers by computer algebra. *Fund. Inform.*, 39(1-2):1–20, 1999. Symbolic computation and related topics in artificial intelligence (Plattsburg, NY, 1998).

[20] D. Barton. A scheme for manipulative algebra on a computer. *The Computer Journal*, 9(4):340–344, 1967.

[21] S. Basu, R. Pollack, and M. Roy. *Algorithms in real algebraic geometry*, volume 10 of *Algorithms and Computation in Mathematics*. Springer-Verlag, Berlin, 2003.

[22] A. Bauer, E. Clarke, and X. Zhao. Analytica – an experiment in combining theorem proving and symbolic computation. *Journ. Autom. Reas.*, 21(3):295–325, 1998.

[23] M. Ben-Or, D. Kozen, and J. Reif. The complexity of elementary algebra and geometry. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 457–464. ACM, 1984.

[24] A. Bentkamp, J. C. Blanchette, and D. Klakow. A formal proof of the expressiveness of deep learning. In *International Conference on Interactive Theorem Proving*, pages 46–64. Springer, 2017.

[25] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: The calculus of inductive constructions*. Springer-Verlag, Berlin, 2004.

[26] P. Billingsley. *Probability and measure*. Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons Inc., New York, third edition, 1995. A Wiley-Interscience Publication.

[27] J. Blanchette, S. Böhme, and L. Paulson. Extending Sledgehammer with SMT solvers. *Automated Deduction–CADE-23*, pages 116–130, 2011.

[28] J. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. *Interactive Theorem Proving*, pages 131–146, 2010.

[29] J. C. Blanchette, M. Haslbeck, D. Matichuk, and T. Nipkow. Mining the archive of formal proofs. In *Conferences on Intelligent Computer Mathematics*, pages 3–17. Springer, 2015.

[30] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co) datatypes for Isabelle/HOL. In *International Conference on Interactive Theorem Proving*, pages 93–110. Springer, 2014.

[31] J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.

[32] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system I: The user language. *Journal of Symbolic Computation*, 24(3):235–265, 1997.

[33] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[34] M. Bronstein. σit—a strongly-typed embeddable computer algebra library. In *International Symposium on Design and Implementation of Symbolic Computation Systems*, pages 22–33. Springer, 1996.

[35] B. Buchberger, T. Jebelean, T. Kutsia, A. Maletzky, and W. Windsteiger. Theorema 2.0: Computer-assisted natural-style mathematics. *Journal of Formalized Reasoning*, 9(1):149–185, 2016.

[36] B. F. Caviness and J. R. Johnson, editors. *Quantifier elimination and cylindrical algebraic decomposition*, Texts and Monographs in Symbolic Computation, Vienna, 1998. Springer-Verlag.

[37] B. Chachuat and M. Villanueva. Bounding the solutions of parametric ODEs: When Taylor models meet differential inequalities. In I. D. L. Bogle and M. Fairweather, editors, *22nd European Symposium on Computer Aided Process Engineering*, volume 30 of *Computer Aided Chemical Engineering*, pages 1307 – 1311. Elsevier, 2012.

[38] C. C. Chang and H. J. Keisler. *Model theory*. North-Holland Publishing Co., Amsterdam-London; American Elsevier Publishing Co., Inc., New York, 1973. Studies in Logic and the Foundations of Mathematics, Vol. 73.

[39] B. W. Char, K. O. Geddes, W. M. Gentleman, and G. H. Gonnet. The design of Maple: A compact, portable, and powerful computer algebra system. In *European Conference on Computer Algebra*, pages 101–115. Springer, 1983.

[40] P. J. Cohen. Decision procedures for real and p-adic fields. *Communications on Pure and Applied Mathematics*, 22:131–151, 1969.

[41] C. A. Cole and S. Wolfram. Smp-a symbolic manipulation program. In *Proceedings of the fourth ACM symposium on Symbolic and algebraic computation*, pages 20–22. ACM, 1981.

[42] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata theory and formal languages (Second GI Conf., Kaiserslautern, 1975)*, pages 134–183. Lecture Notes in Comput. Sci., Vol. 33. Springer, Berlin, 1975. Reprinted in [36].

[43] L. Cong, Y. Dong, I. Ford, R. Y. Lewis, J. Martín-García, and J. Mulnix. Progress in pure mathematics. Wolfram Technology Conference, 2016.

[44] T. Coquand and G. Huet. The Calculus of Constructions. *Inform. and Comput.*, 76(2-3):95–120, 1988.

[45] T. Coquand and C. Paulin. Inductively defined types. In *COLOG-88 (Tallinn, 1988)*, volume 417 of *Lec. Notes in Comp. Sci.*, pages 50–66. Springer, Berlin, 1990.

[46] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt, M. Kaufmann, and P. Schneider-Kamp. Efficient certified rat verification. In L. de Moura, editor, *Automated Deduction – CADE 26*, pages 220–236, Cham, 2017. Springer International Publishing.

[47] T. Daly. *Axiom: The 30 year horizon*. Lulu Incorporated, 2005.

[48] J. H. Davenport and J. Heintz. Real quantifier elimination is doubly exponential. *J. Symbolic Comput.*, 5(1-2):29–35, 1988.

[49] N. G. de Bruijn. *AUTOMATH, a Language for Mathematics*, pages 159–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[50] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover. *http://leanprover.github.io/files/system.pdf*, 2014.

[51] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.

[52] D. Delahaye and M. Mayero. Field, une procédure de décision pour les nombres réels en Coq. In *JFLA*, pages 33–48, 2001.

[53] D. Delahaye and M. Mayero. Dealing with algebraic expressions over a field in Coq using Maple. *Journal of Symbolic Computation*, 39(5):569 – 592, 2005. Automated Reasoning and Computer Algebra Systems (AR-CA).

[54] M. Dénès, A. Mörtberg, and V. Siles. A refinement-based approach to computational algebra in Coq. In *Interactive theorem proving*, volume 7406 of *Lecture Notes in Comput. Sci.*, pages 83–98. Springer, Heidelberg, 2012.

[55] J. Duracz, A. Farjudian, M. Konečnỳ, and W. Taha. Function interval arithmetic. In *International Congress on Mathematical Software*, pages 677–684. Springer, 2014.

[56] G. Ebner, S. Ullrich, J. Roesch, J. Avigad, and L. de Moura. A metaprogramming framework for formal verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):34, 2017.

[57] A. Eggers, E. Kruglov, S. Kupferschmid, K. Scheibler, T. Teige, and C. Weidenbach. Superposition modulo non-linear arithmetic. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of Combining Systems*, pages 119–134, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[58] C. Engelman. MATHLAB: a program for on-line machine assistance in symbolic computations. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part II: computers: their impact on society*, pages 117–126. ACM, 1965.

[59] J. Fourier. Histoire de l'académie, partie mathématique. *Mémoires de l'Académie des sciences de l'Institut de France*, 7, 1827.

[60] I. Frick. The computer algebra system sheep, what it can and cannot do in general relativity. Technical report, 1977.

[61] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völp, and A. Platzer. Keymaera X: an axiomatic tactical theorem prover for hybrid systems. In *International Conference on Automated Deduction*, pages 527–538. Springer, 2015.

[62] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.8.8*, 2017.

[63] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for computer algebra*. Springer Science & Business Media, 1992.

[64] G. Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[65] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. O. Biha, et al. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179. Springer, 2013.

[66] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF: a mechanized logic of computation, volume 78 of lecture notes in computer science, 1979.

[67] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher-order logic*. Cambridge University Press, 1993.

[68] H. Gottliebsen, T. Kelsey, and U. Martin. Hidden verification for computational mathematics. *Journal of Symbolic Computation*, 39(5):539 – 567, 2005. Automated Reasoning and Computer Algebra Systems (AR-CA).

[69] F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In *International Workshop on Types for Proofs and Programs*, pages 160–174. Springer, 2006.

[70] T. Hales. *Dense Sphere Packings: A Blueprint for Formal Proofs*. Cambridge University Press, Cambridge, 2012.

[71] T. Hales, M. Adams, G. Bauer, T. D. Dang, J. Harrison, H. Le Truong, C. Kaliszyk, V. Magron, S. McLaughlin, T. T. Nguyen, et al. A formal proof of the Kepler conjecture. In *Forum of Mathematics, Pi*, volume 5. Cambridge University Press, 2017.

[72] T. Hales et al. Formal abstracts. https://github.com/formalabstracts/formalabstracts.

[73] T. C. Hales. A proof of the Kepler conjecture. *Ann. of Math. (2)*, 162(3):1065–1185, 2005.

[74] T. C. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete Comput. Geom.*, 44(1):1–34, 2010.

[75] A. D. Hall Jr. The Altran system for rational function manipulation—a survey. *Communications of the ACM*, 14(8):517–521, 1971.

[76] G. H. Hardy, J. E. Littlewood, and G. Pólya. *Inequalities*. Cambridge university press, 1952.

[77] J. Harrison. HOL light: a tutorial introduction. In M. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 265–269, 1996.

[78] J. Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, Cambridge, 2009.

[79] J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *J. Automat. Reason.*, 21(3):279–294, 1998.

[80] R. Hersh. *What is mathematics, really?* Oxford University Press, 1997.

[81] M. Heule, W. A. H. Jr., M. Kaufmann, and N. Wetzler. Efficient, verified checking of propositional proofs. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, pages 269–284, 2017.

[82] M. Heule, O. Kullmann, and V. W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 228–245, 2016.

[83] J. Hohenwarter, M. Hohenwarter, and Z. Lavicza. Introducing dynamic mathematics software to secondary school teachers: The case of GeoGebra. *Journal of Computers in Mathematics and Science Teaching*, 28(2):135–146, 2009.

[84] L. Hörmander. *The analysis of linear partial differential operators. II*, volume 257 of *Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]*. Springer-Verlag, Berlin, 1983. Differential operators with constant coefficients.

[85] W. A. Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[86] W. A. Hunt, R. B. Krug, and J. Moore. Linear and nonlinear arithmetic in ACL2. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods, Proceedings of CHARME 2003*, number 2860 in Lecture Notes in Computer Science, pages 319–333. Springer-Verlag, 2003.

[87] F. Immler. A verified enclosure for the Lorenz attractor (rough diamond). In *International Conference on Interactive Theorem Proving*, pages 221–226. Springer, 2015.

[88] F. Immler and J. Hölzl. Numerical analysis of ordinary differential equations in Isabelle/HOL. In *International Conference on Interactive Theorem Proving*, pages 377–392. Springer, 2012.

[89] W. Jeffreys. Trigman. *ACM SIGSAM Bulletin*, (24):20–21, 1972.

[90] C. Kaliszyk and F. Wiedijk. Certified computer algebra on top of an interactive theorem prover. In *Proceedings of the 14th Symposium on Towards Mechanized Mathematical Assistants: 6th International Conference*, Calculemus '07 / MKM '07, pages 94–105, Berlin, Heidelberg, 2007. Springer-Verlag.

[91] M. Kaufmann and J. S. Moore. ACL2: An industrial strength version of Nqthm. In *Computer Assurance, 1996. COMPASS'96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on*, pages 23–34. IEEE, 1996.

[92] M. Kerber, M. Kohlhase, and V. Sorge. Integrating computer algebra into proof planning. *J. Automat. Reason.*, 21(3):327–355, 1998.

[93] P. Lammich. Efficient verified (UN)SAT certificate checking. In L. de Moura, editor, *Automated Deduction – CADE 26*, pages 237–254, Cham, 2017. Springer International Publishing.

[94] R. Y. Lewis. Polya: a heuristic procedure for reasoning with real inequalities. *MS Thesis, Dept. of Philosophy, Carnegie Mellon University*, 2014.

[95] R. Y. Lewis. An extensible ad hoc interface between Lean and Mathematica. In *Proof eXchange for Theorem Proving*, 2017.

[96] A. Mahboubi and E. Tassi. *Mathematical Components.*

[97] G. Malecha, A. Chlipala, and T. Braibant. Compositional computational reflection. In *International Conference on Interactive Theorem Proving*, pages 374–389. Springer, 2014.

[98] P. Mancosu. *The philosophy of mathematical practice.* Oxford University Press on Demand, 2008.

[99] W. A. Martin and R. J. Fateman. The MACSYMA system. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 59–75. ACM, 1971.

[100] P. Martin-Löf and G. Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Napoli, 1984.

[101] R. Matuszewski and P. Rudnicki. Mizar: the first 30 years. *Mechanized Ma*, 4(1):3–24, March 2005.

[102] C. McBride and J. McKinna. Functional pearl: I am not a number–I am a free variable. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 1–9, New York, NY, USA, 2004. ACM.

[103] S. McLaughlin and J. Harrison. A proof producing decision procedure for real arithmetic. In R. Nieuwenhuis, editor, *Automated Deduction – CADE-20. 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings*, volume 3632 of *Lecture Notes in Artificial Intelligence*, pages 295–314, 2005.

[104] N. Megill. Metamath. *The Seventeen Provers of the World*, pages 88–95, 2006.

[105] G. Melquiond. Coq-interval, 2011.

[106] J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic*, 7(1):41–57, 2009.

[107] B. Mishra. Computational real algebraic geometry. In *Handbook of discrete and computational geometry*, CRC Press Ser. Discrete Math. Appl., pages 537–556. CRC, Boca Raton, FL, 1997.

[108] R. Nederpelt and H. Geuvers. *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, 2014.

[109] T. Nipkow, G. Bauer, and P. Schultz. Flyspeck I: Tame Graphs. In *Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Computer Science*, pages 21–35. Springer, 2006.

[110] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL. A proof assistant for higher-order logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 2002.

[111] U. Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Chalmers University of Technology, 2007.

[112] S. Obua and T. Nipkow. Flyspeck II: the basic linear programs. *Ann. Math. Artif. Intell.*, 56(3-4):245–272, 2009.

[113] L. C. Paulson. Set theory for verification: I. from foundations to functions. *Journal of Automated Reasoning*, 11(3):353–389, Oct 1993.

[114] L. C. Paulson and J. C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In *PAAR@ IJCAR*, pages 1–10, 2010.

[115] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[116] A. Platzer. The complete proof theory of hybrid systems. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, pages 541–550. IEEE Computer Society, 2012.

[117] A. Platzer and J.-D. Quesel. Keymaera: A hybrid theorem prover for hybrid systems. In Armando, Baumgartner, and Dowek, editors, *IJCAR 2008*, pages 171–178. Springer, Heidelberg, 2008.

[118] A. Platzer, J.-D. Quesel, and P. Rümmer. Real world verification. In *Automated deduction—CADE-22*, volume 5663 of *Lecture Notes in Comput. Sci.*, pages 485–501. Springer, Berlin, 2009.

[119] G. Polya. *How to solve it*. Princeton Science Library. Princeton University Press, Princeton, NJ, 2004. A new aspect of mathematical method, Expanded version of the 1988 edition, with a new foreword by John H. Conway.

[120] V. R. Pratt. Every prime has a succinct certificate. *SIAM Journal on Computing*, 4(3):214–220, 1975.

[121] F. Rabe. The MMT API: a generic MKM system. In *International Conference on Intelligent Computer Mathematics*, pages 339–343. Springer, 2013.

[122] A. Schrijver. *Theory of linear and integer programming*. Wiley-Interscience Series in Discrete Mathematics. John Wiley & Sons Ltd., Chichester, 1986. A Wiley-Interscience Publication.

[123] O. Seddiki, C. Dunchev, S. Khan-Afshar, and S. Tahar. *Enabling Symbolic and Numerical Computations in HOL Light*, pages 353–358. Springer International Publishing, Cham, 2015.

[124] A. Seidenberg. A new decision method for elementary algebra. *Ann. of Math. (2)*, 60:365–374, 1954.

[125] D. Selsam and L. de Moura. Congruence closure in intensional type theory. In *International Joint Conference on Automated Reasoning*, pages 99–115. Springer, 2016.

[126] N. Shankar, S. Owre, J. M. Rushby, and D. W. Stringer-Calvert. Pvs prover guide. *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1:11–12, 2001.

[127] A. Solovyev. *Formal Computation and Methods*. PhD thesis, University of Pittsburgh, 2012.

[128] W. Stein and D. Joyner. Sage: System for algebra and geometry experimentation. *ACM SIGSAM Bulletin*, 39(2):61–64, 2005.

[129] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 2nd edition, 1951.

[130] T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[131] J. Urban, P. Rudnicki, and G. Sutcliffe. ATP and presentation service for Mizar formalizations. *J. Automat. Reason.*, 50(2):229–241, 2013.

[132] F. van Doorn, J. von Raumer, and U. Buchholtz. *Homotopy Type Theory in Lean*, pages 479–495. Springer International Publishing, Cham, 2017.

[133] V. Voevodsky. The origins and motivations of univalent foundations. *The Institute Letter*, pages 8–9, 2014.

[134] J. Von Zur Gathen and J. Gerhard. *Modern computer algebra*. Cambridge University Press, 2013.

[135] F. Wiedijk. Formalizing Arrow's theorem. *Sādhanā*, 34(1):193–220, 2009.

[136] H. Williams. Fourier's method of linear programming and its dual. *The American Mathematical Monthly*, 93(9):681–695, 1986.

[137] S. Wolfram. *The Mathematica Book*. Number v. 1. Wolfram Media, 2003.

[138] S. Wolfram. *An Elementary Introduction to the Wolfram Language*. Wolfram Media, Incorporated, 2015.

[139] B. Zhan. Auto2, a saturation-based heuristic prover for higher-order logic. In *International Conference on Interactive Theorem Proving*, pages 441–456. Springer, 2016.

[140] B. Zhan. Formalization of the fundamental group in untyped set theory using auto2. In *International Conference on Interactive Theorem Proving*, pages 514–530. Springer, 2017.

[141] G. M. Ziegler. *Lectures on polytopes*, volume 152 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1995.