Polya: A Heuristic Procedure for Reasoning with Real Inequalities

Robert Y. Lewis

December 11, 2014

Contents

Acknowledgements

There are a number of people I must thank for their contributions, direct or indirect, to this project.

Thanks to my family and friends for their emotional support the past many years. This thesis would not exist without them.

Many have contributed helpful comments as this project has developed; I have no hope of listing them all here. Thanks to everyone who has given feedback during practice presentations, official presentations, and informal discussions concerning this work.

The comments of my thesis committee members Leonardo de Moura and Wilfried Sieg have been invaluable in shaping and perfecting this document. I can't overstate the value of Leo's advice during Polya's development and the writing of the article on which this thesis is based. I must also thank Cody Roux for his contributions to that article and to the Polya source code.

Finally, my advisor Jeremy Avigad has spent countless hours on this project with me: explaining theory, developing code, coming up with good ideas, and shooting down some bad ones. Thank you, Jeremy, for your time, patience, and guidance – I look forward to many more years of collaboration!

Chapter 1

Introduction

The complexity of mathematical arguments and the difficulty in verifying their correctness have grown substantially over the last decades. For this reason, mathematicians have been paying increasing interest to the field of automated formal verification. Under this paradigm, the mathematician produces a proof that resembles computer code. A verification program executes this code and confirms that it can be used to construct a gapless formal deduction. The epistemic merits of these interactive proofs have been much debated. Nevertheless, as the systems have become more and more sophisticated, their ability to deliver a high degree of confidence in the correctness of the results has begun to attract the notice of mathematicians.

In the process of formally certifying the correctness of arguments, many interactive proof environments incorporate calls to automated reasoning procedures, or *tactics*, that prove certain auxiliary claims. For example, rather than manually solving a long system of linear equations, the user can present the system and obtain the solution via a decision procedure for linear arithmetic. A proof assistant, ideally able to manage much of the computation automatically, frees the mathematician to focus on the higher structure of the proof.

Automated procedures of this sort exist for many domains and are often quite effective. However, there are domains for which the decision procedures are in practice intractable, and many others for which no procedure exists at all. These methods are generally effective on domains that are in some sense homogeneous; domains which mix operations or sorts of terms tend to be more difficult to decide. Of course, the interesting problems in mathematics are often the harder ones to solve, and problems over these heterogeneous domains show up frequently in interactive (and standard) proofs.

Even when a theory is decidable, the decision procedure can leave something to be desired. These algorithms may be inefficient, ill-suited to producing proof terms, or difficult or impossible to extend to larger theories. All of these flaws, to some extent or another, constrain the use of automated techniques in interactive theorem proving. The ideal tactic is quick to run, easily produces a "natural" proof that the proof assistant can interpret, and will run in a variety of settings.

This thesis describes an automated technique for real arithmetic, developed jointly with Jeremy Avigad and Cody Roux. Our system has the potential to overcome these flaws, at the cost of decidability. Once implemented in a proof-producing manner, we hope that it will be an important and useful tool in the interactive theorem proving toolkit.

The following sections provide background information about proof verification and decision procedures, and motivate our technique. In Chapter 2, we describe the class structure and architecture of our system; in Chapters 3 and 4, we explain its computational behavior. Chapter 5 explores the system's accomplishments and shortcomings, and compares it to other proof techniques in use. Finally, Chapter 6 discusses ways in which our system might be improved.

The description here aims to detail our system's theory, architecture, and computational abilities. It is suited for those who wish to understand the nuts and bolts, in order to use its advanced features or extend its abilities. Casual readers may wish to see [6], which provides a shorter overview of the system and its capacities. Sections of this thesis are taken directly from this paper.

1.1 Automated Reasoning and Proof Verification

The first mechanized proof assistant, Mizar, was described by Andrzej Trybulec in 1973; it wasn't until 1975 that a working proof-checker was produced ([41]). Mizar's language and capacities expanded over the following decades, and the Mizar Mathematical Library – a carefully maintained collection of Mizar proofs – is currently the largest unified collection of formal proofs in existence ([61]). Until recently ([60]), Mizar had very little support for automated methods, requiring users to make calculations explicit.

Since the introduction of Mizar, many systems have been developed based on many different underlying logics. Perhaps the most popular systems under current development are Coq ([11]), a powerful system based on constructive dependent type theory; the HOL family ([32]), a collection of assistants built around a small higher-order logic kernel; and Isabelle ([50]), a generic prover that can implement a number of underlying logics. In contrast to Mizar, these systems all integrate substantial automation, and interact in varying ways with other programs to verify complicated computational arguments. Lean, a forthcoming proof assistant discussed in Section 6.1, will provide even more options for automation.

Thomas Hales' *Flyspeck* project ([49], [51], [33]) 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 began 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 Isabelle that classified all tame graphs. The second, computational, part of the proof was done by a verified program in HOL Light. The Isabelle portion was mechanically translated into 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 faster ([34]). Traditional techniques for algorithmically deciding nonlinear inequalities were far too inefficient to be of use, and so the Flyspeck group had to develop methods specialized to their purposes ([57]).

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. In the following section we look at a number of procedures for related problems, some of which have been implemented in proof-producing ways.

1.2 Decision Procedures for the Reals

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 $\mathbb{T}_{RCF}$ ([19]). 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 (although did not publish a proof until 1948) that $\mathbb{T}_{RCF}$ admits quantifier elimination ([59]). As an immediate consequence of this observation, one sees 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. Along with 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 ([44]) to have NONELEMENTARY complexity – that is, no tower function $2^{2^{\cdot^{\cdot^{\cdot^{n}}}}}$ bounds the algorithm's run time for all n, where n is the number of quantifiers. In fact, Davenport and Heintz ([22] show that the quantifier elimination problem is necessarily at least doubly exponential in n. Nonetheless, Tarski is optimistic in his monograph 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 ([21], [10]) realizes the doubly exponential bound on this decision problem. Given a set of polynomial inequalities $\mathcal{F} = \{f_i(\bar{x}) > 0 | f_i : \mathbb{R}^n \to \mathbb{R}\}$, 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 it fails. (We investigate the implementation of CAD in the system Z3 below, in Section 5.1.)

Various other techniques for deciding $\mathbb{T}_{RCF}$ have been discovered, including ones by Seidenberg [1954], Cohen [1969], and Hörmander [1983] ([56], [20], [38]). McLaughlin and Harrison ([42]) 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 $(\forall x)(-1 \leq x \leq 1 \to -1 \leq 4x^3 - 3x \leq 1)$ takes over a minute on a standard desktop.

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 ([10]) give a singly-exponential time algorithm to decide problems in the existential fragment of $\mathbb{T}_{RCF}$. Gröbner basis methods are efficient for problems in the universal fragment of arithmetic involving equalities, but do not extend well to inequalities ([37]). 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.

1.3 Combining Decision Procedures

In [4], 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.

Formally, let $\mathbb{T}_{add}[\mathbb{Q}]$ denote the theory of the real numbers under the language consisting of symbols

$$0, 1, +, -, <, \dots, f_a, \dots,$$

where for each $a \in \mathbb{Q}$, the function symbol f_a denotes the mapping $x \mapsto ax$. We can similarly define $\mathbb{T}_{mul}[\mathbb{Q}]$, where the symbols $+$ and $-$ are replaced by $\times$ and $\div$ respectively. Denote $\mathbb{T}_{add}[\mathbb{Q}] \cap \mathbb{T}_{mul}[\mathbb{Q}]$ by $\mathbb{T}_{comm}[\mathbb{Q}]$, and $\mathbb{T}_{add}[\mathbb{Q}] \cup \mathbb{T}_{mul}[\mathbb{Q}]$ (a theory over the combined language) by $\mathbb{T}[\mathbb{Q}]$. Since interactions between addition and multiplication are not axiomatized, $\mathbb{T}[\mathbb{Q}]$ is weaker than $\mathbb{T}_{RCF}$: it does not contain the rule for distribution.

Each of $\mathbb{T}_{add}[\mathbb{Q}]$, $\mathbb{T}_{mul}[\mathbb{Q}]$, and $\mathbb{T}_{comm}[\mathbb{Q}]$ supports quantifier elimination via the Fourier-Motzkin algorithm.[1] Thus, each is complete and decidable. Avigad and Friedman go on to show, via a reduction to the decision procedure for $\mathbb{T}_{RCF}$, that the universal fragment of $\mathbb{T}[\mathbb{Q}]$ is decidable as well. (However, the existential fragment of $\mathbb{T}[\mathbb{Q}]$ embeds Hilbert's 10th Problem over $\mathbb{Q}$, which is conjectured to be undecidable.)

This conclusion is in some sense disappointing, as it seems to show that the combination of $\mathbb{T}_{add}[\mathbb{Q}]$ and $\mathbb{T}_{mul}[\mathbb{Q}]$ is no less complex (and in fact more unwieldy) than $\mathbb{T}_{RCF}$. However, at the end of the paper, Avigad and Friedman discuss "pragmatic" procedures to approximate their decidability results. They liken their approximations to "heuristic procedures that traverse the search space by applying a battery of natural inferences in a systematic way." While full decidability – even over the restricted theory $\mathbb{T}[\mathbb{Q}]$ – may be infeasible, these heuristic approximations are practical and often successful.

1.4 Polya: A Motivation

The various decision procedures described in Section 1.2 are quite powerful, and many are applicable in a wide variety of situations. But, frustratingly, one need not look very far to find situations in which most or all of these procedures fail. Consider as an example the following valid implication:

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

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 $\texttt{exp}(1 + x + x^4) <$

[1]See Section 3.2 of this thesis for a description of this elimination method.

$\mathtt{exp}(1 + y + y^4)$, 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 following chapters of this thesis propose a method 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 a preliminary release of 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 as well as his thoughtful studies of heuristic methods in mathematics. Henceforth, we use "Polya" to refer to the implementation, rather than the system in abstract.

Chapter 2

Architecture

In this chapter, we will take a look at the foundational architecture of the system. Polya consists of a complex information database (the Blackboard, Section 2.2) that serves as an interface between various inferential modules (discussed in Chapters 3 and 4). Users are able to interact with the system in a variety of ways (see Section 2.4), which allow differing degrees of control over the system's behavior.

Polya's structure has been largely inspired by the Satisfiability Modulo Theories (SMT) approach to automated theorem proving. SMT solvers typically consist of a boolean SAT solver that communicates with various specialized theory solvers. Formulae are given as boolean combinations of theory literals, which the SAT solver treats as atomic; it searches for satisfying instances of the boolean problem, and consults with the theory solvers whether these assignments are feasible. This approach (based on the theory-combination algorithm of Nelson and Oppen [48]) is analogous to the combination procedure discussed in Section 1.3, with theories whose languages share only the symbol for equality.

In some sense, then, Polya represents a generalization of the SMT technique to theories with overlapping alphabets. In other senses, of course, 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.

2.1 Term Structure and Normal Forms

We wish to 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.

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

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

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 will be eliminated when we define a notion of *normal form* for 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.

To define the notion of normal form for terms, we fix an ordering $\prec$ on variables and function symbols, and extend that to an ordering on terms and s-terms. For example, we can arbitrarily set the term 1 to be minimal in the ordering, then variables, then products, then sums, and finally function applications, recursively using lexicographic ordering on the list of arguments (and the function symbol) within the latter three categories. The set of terms in normal form is then defined inductively as follows:

- $1, x, y, z, \ldots$ are terms in normal form.
- $\sum_{i=1\ldots n} c_i \cdot t_i$ is in normal form provided $c_1 = 1$, each t_i is in normal form, and $t_1 \prec t_2 \prec \ldots \prec t_n$.
- $\prod_i t_i^{n_i}$ is in normal form provided each t_i is in normal form, and $1 \neq t_1 \prec t_2 \prec \ldots \prec t_n$.
- $f(s_1, \ldots, s_n)$ is in normal form if each s_i is.

The details are spelled out in Avigad and Friedman ([4]). That paper provides an explicit first-order theory, T, expressing commutativity and associativity of addition and multiplication, distributivity of constants over sums, and so on, such that the following two properties hold:

1. For every term t, there is a unique s-term s in canonical form, such that T proves $t = s$.

2. Two terms t_1 and t_2 have the same canonical normal form if and only if T proves $t_1 = t_2$.

For example, the term $3(5x+3y+4xy)^2 f(u+v)^{-1}$ is expressed canonically as $75 \cdot (x + (3/5) \cdot y + (4/5) \cdot xy)^2 f(u+v)^{-1}$, where the constant in the additive term $5x + 3y + 4xy$ has been factored so that the result is in normal form.

The two clauses above provide an axiomatic characterization of what it means for terms to have the same canonical form.

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 $<, \leq, >, \geq, =$, or $\neq$, 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.

2.2.1 Hierarchy of comparison types

Eight types of information are stored in the Blackboard class in Polya:

- Term definitions. The Blackboard is responsible for canonizing terms as described in Section 2.1, and for isolating and naming terms of different types. As a problem is described to the Blackboard, relevant subterms are associated with indices from 1 to n. We refer to these subterms as *problem terms*; the `IVar` t_k is a variable defined to be equal to the kth problem term. These definitions are stored in an array in the Blackboard. Recall that we follow the convention $t_0 := 1$.

  Modules cannot explicitly assert term definitions to the Blackboard. However, modules are able to request from the Blackboard a name for a given term s. The Blackboard will return an index k such that the `IVar` t_k is equal to s, if such an index is available; if not, the Blackboard optionally can create a new term defined to be the canonization of s, along with any subterms needed.

- Equalities between `IVars`. Two syntactically equal terms will be canonized to the same form, and will thus be identified in the Blackboard by the same `IVar` index. However, terms may be found to be equal (perhaps up to a constant multiple) in other ways, either by hypothesis, by arithmetic, or by computation from other modules. The Blackboard stores equalities between `IVars` t_i and t_j in a dictionary mapping (i, j) to a nonzero rational constant c, representing that $t_i = c \cdot t_j$. The case $t_i = 0$ is special, as it implies $t_i = 0 \cdot t_j$ for all $0 \leq j \leq n$; it is thus handled separately.

- Equalities with 0. There are again various ways in which the Blackboard could learn that an `IVar` is equal to 0. The Blackboard stores a list of indices i such that $t_i = 0$. This information supersedes equality with other `IVars`: from $t_i = 0$ the Blackboard can deduce all relevant comparisons between t_i and t_j, regardless of how much information it has about t_j.

- Disequalities between `IVars`. The Blackboard tracks information of the form $t_i \neq c \cdot t_j$ by maintaining a dictionary that maps (i, j) to a list of rational constants $[c_1, \ldots, c_k]$. This information is superseded[1] by equality information between t_j and t_j, and can be superseded by inequality information.

[1]Mostly. If $t_i = c_1 \cdot t_j$, the fact that $t_i \neq c_2 \cdot t_j$ is not irrelevant, as it implies that $t_i, t_j \neq 0$.

- Disequalities with 0. Similarly, the Blackboard tracks information of the form $t_i \neq 0$ by maintaining a list of indices.

- Inequalities between `IVars`. For two `IVars` t_i and t_j, the Blackboard may learn that $t_i \bowtie c \cdot t_j$ for $\bowtie \in \{<, \leq, \geq, >\}$. It is easy to see geometrically that absent any equality information, exactly two noncollinear comparisons of this form completely describe the known relationship between t_i and t_j: any third comparison of this form will either be implied by the others, contradict them, or replace one. This is described in more detail in Section 2.2.2 below, along with a description of the Halfplane data structure for tracking inequalities.

- Inequalities with 0. The Blackboard stores sign information for `IVars`. This information can often be computed from comparisons between `IVars`. However, since it is often of vital importance for modules to access signs of variables, the information is duplicated in a more easily accessible structure. The Blackboard maintains a dictionary that maps i to one of $\{<, \leq, \geq, >\}$.

- Clauses. Finally, the Blackboard stores disjunctive information. The Blackboard may be told that one of a list of comparisons must be true, without being told which one; it stores these lists and updates them as literals are confirmed or falsified. This structure is explained in more detail in Section 2.2.6.

The data hierarchy tries to minimize redundant information by storing only the strongest available comparisons. If an equality is known between t_i and t_j, it is unnecessary to store any inequality or disequality information between those variables; similarly, if two inequalities are known, this implies infinitely many disequalities that do not need to be stored.

Minimizing the storage of redundant information requires care in maintaining each data structure. Sign information about t_i and t_j can affect whether an inequality $t_i < c \cdot t_j$ is redundant or not, so the inequalities structure must take this information into account. Methods for adding information to one of these data structures must be able to update the others as well: adding $t_i > 0$ to the list of inequalities with 0 requires removing $t_i \neq c \cdot 1$ from the list of disequalities for all negative c.

2.2.2 Halfplane representation of inequalities

We claim above that to understand the relationship between two variables t_i and t_j absent equality information, it is sufficient to know two comparisons of the form $t_i \bowtie c \cdot t_j$, where $\bowtie \in \{<, \leq, \geq, >\}$. Here we explain this more thoroughly, and detail the data structure for storing such inequalities. Comparisons of the form $t_i, t_j \bowtie 0$ complicate this discussion somewhat. It is convenient to allow c to take the values 0 and ∞; as the following description is largely geometric, we think of these simply as horizontal and vertical lines. The translation of these ideas back to their algebraic descriptions, and their implementations in Polya, are conceptually trivial but tedious to describe.

An equality $t_i = c \cdot t_j$ corresponds to a line through the origin in the $t_i t_j$ coordinate space. If the equality is changed to an inequality, $t_i \bowtie c \cdot t_j$ corresponds to the halfplane on one side of this line. The halfplane contains the origin (and its defining boundary line) exactly when $\bowtie$ is a weak inequality $\leq$ or $\geq$. Under this representation, a conjunction of two inequalities is represented by an intersection of their corresponding halfplanes; unless the halfplanes have identical boundaries, their intersection will be a nonempty segment of the plane that is a proper subset of both halfplanes. This shows that, disregarding degenerate cases, knowing one inequality between two variables is insufficient to fully describe their relationship.

Furthermore, any satisfiable conjunction of three inequalities contains redundant information; one of its conjuncts may be discarded. (In other words, knowing two inequalities between two variables is always sufficient to fully describe their relationship.) Let H_1, H_2, and H_3 denote the corresponding halfplanes. We assume that none of these halfplanes have identical boundaries, as such edge cases are easy to eliminate. There are three cases to consider geometrically.

1. $H_1 \cap H_2 \cap H_3 = \emptyset$. If the intersection is empty, the conjunction of the inequalities is unsatisfiable.

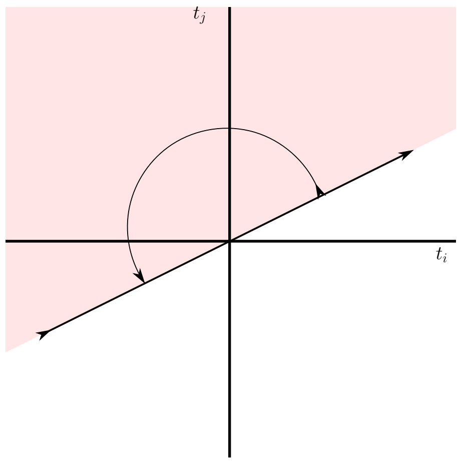
2. $H_1 \cap H_2 \subset H_3$. In this case, the conjunction of the first two inequalities implies the third, which may be discarded.

3. The boundary line $\partial H3$ intersects $H_1 \cap H_2$. In this case, either $H_1 \cap H_3 \subset H_1 \cap H_2$ or $H_2 \cap H_3 \subset H_1 \cap H_2$. If the former, the second inequality may be discarded; if the latter, the third may be discarded.

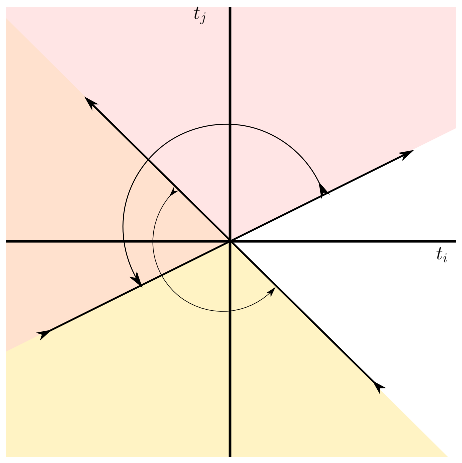
It is in this sense that we say two inequalities represent "complete" information between t_i and t_j.

These considerations also suggest halfplanes as a natural way of representing inequalities. We use this technique to store inequality information in the Blackboard. A Halfplane object is defined by a vector (a, b) and a boolean `strong`, representing the halfplane counterclockwise from (a, b) and containing the line connecting (a, b) to the origin iff `strong` is false. Comparisons between t_i and t_j are tracked by mapping (i, j) to a list of between zero and two Halfplanes, representing the conjunction of their corresponding inequalities. When two inequalities are known, we maintain that the defining ray of the first Halfplane is counterclockwise of that of the second. Note again that we may assume these halfplanes have distinct boundaries, since it is simple to reduce cases otherwise. If (a_1, b_1) and (a_2, b_2) are coincident, then either the Halfplanes are equal or exactly one is `strong`, in which case the other is redundant. If (a_1, b_1) and $(-a_2, -b_2)$ are coincident, then either both are not `strong` and the comparisons imply an equality, or at least one is `strong` and the comparisons are contradictory.

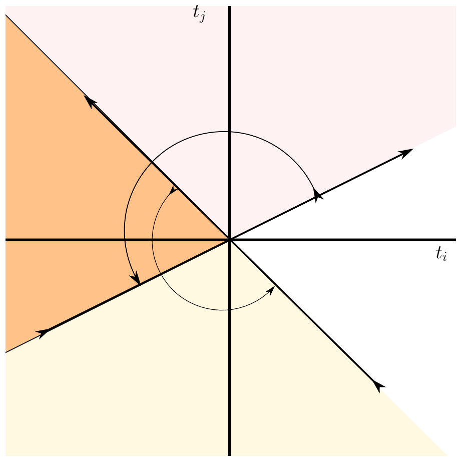
Given two inequalities H_1 and H_2, we often face the problem of determining whether a third inequality represented by H_3 is redundant, contradictory, or contains new information. With our representation, this is easily accomplished with simple arithmetic. If the defining vectors of H_3 lies between those of H_1 and H_2 (that is to say, H_1 is counterclockwise of H_3



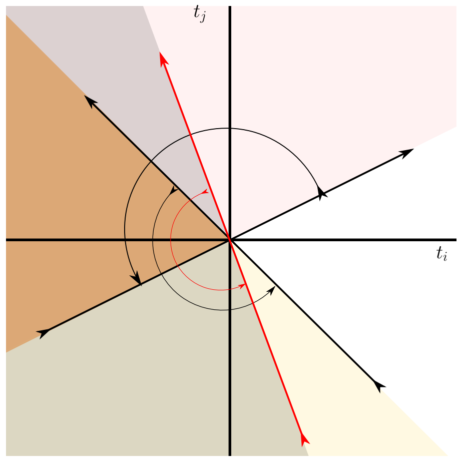
(a) The halfplane representing $t_i \leq 2 \cdot t_j$.



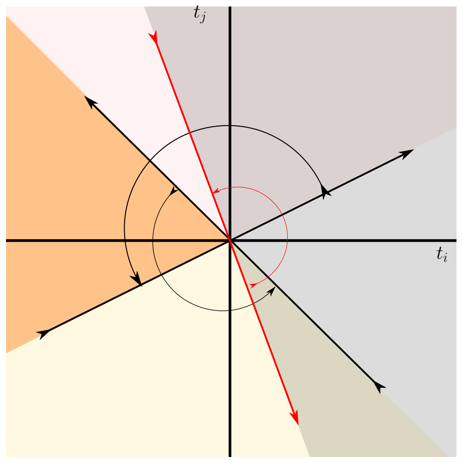
(b) Overlayed with $t_i \leq -t_j$.



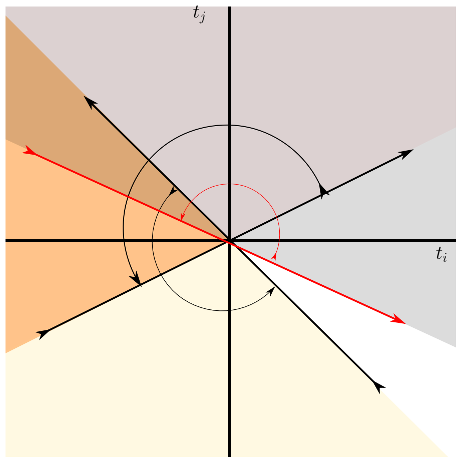
(c) Intersection represents their conjunction.



(d) $t_i \leq -1/2 \cdot t_j$ is implied.



(e) $t_i \geq -1/2 \cdot t_j$ is contradictory.



(f) $t_i \geq -2 \cdot t_j$ is new information.

Figure 2.1: Representation of inequalities by halfplanes

which is counterclockwise of H_1), then H_3 is redundant. If the negation of H_3 (obtained by reflecting its defining vector through the origin) is redundant, than H_3 is contradictory. Otherwise H_3 contains new information. Either H_3 is counterclockwise of both H_1 and H_2, in which case it replaces H_1, or it is clockwise of both, in which case it replaces H_2.

We note that (a, b) is clockwise of (c, d) exactly when $a \cdot d - b \cdot c > 0$, a formula derived from the right-hand cross product.

Figure 2.1 illustrates the Halfplane representation of inequalities. In Figure 2.1a, the vector $(2, 1)$ stands in for the inequality $t_i \leq 2 \cdot t_j$; the halfplane counterclockwise of $(2, 1)$ is shaded. (We have included the reflection of $(2, 1)$ through the origin for the sake of clarifying the picture. It is not part of the Halfplane representation.) In Figure 2.1b, the vector $(-1, 1)$ has been added, representing $t_i \leq -t_j$. Their intersection, visible as the darkly shaded region in Figure 2.1c, represents the conjunction of the two inequalities. In Figure 2.1d, we see that this conjunction implies $t_i \leq -1/2 \cdot t_j$, since the intersection lies inside the new halfplane. Figure 2.1e similarly shows that the conjunction contradicts $t_i \geq -1/2 \cdot t_j$. Finally, 2.1f shows that the inequality $t_i \geq -2 \cdot t_j$ contains new information, since its halfplane intersects the feasible region; to be precise, the conjunction of this with $t_i \leq -t_j$ implies the original inequality $t_i \leq 2 \cdot t_j$.

2.2.3 The `implies` routine

It is often necessary to determine whether the information stored in the Blackboard implies a given comparison $t_i \bowtie c \cdot t_j$. Since different types of comparisons are stored in different data structures, and certain types of comparisons imply comparisons of other types (e.g., an equality can imply an inequality), this routine is fairly complex. Its behavior is documented here in greater detail.

Since the Blackboard stores comparisons with 0 differently than comparisons between terms, the `implies` routine must handle this as a special case. When the routine is asked whether a comparison $t_i \bowtie 0$ is implied, its behavior depends on the type of the comparison $\bowtie$.

- $\bowtie$ is $=$. Equality information is not superseded by any other type of information. $t_i = 0$ is implied only when i is in the list of zero equality indices.

- $\bowtie$ is $\neq$. This is implied when i is in the list of zero disequality indices, or when one of $t_i > 0$ or $t_i < 0$ is known.

- $\bowtie$ is $<$ or $>$. This is implied when $\bowtie$ exactly matches the value for i in the zero inequality dictionary.

- $\bowtie$ is $\leq$ or $\geq$. This is implied when the direction of $\bowtie$ matches the direction of the value for i in the zero inequality dictionary, or when i is in the list of zero equality indices.

Similarly, the process to check whether $t_i \bowtie c \cdot t_j$ again depends on the type of $\bowtie$.

- $\bowtie$ is $=$. This is implied when (i, j) maps to c in the term equality dictionary, or when both t_i and t_j are equal to 0.
- $\bowtie$ is $\neq$. This is implied when (i, j) maps to a list containing c in the term disequality dictionary, or when the Blackboard implies either $t_i > c \cdot t_j$ or $t_i < c \cdot t_j$.
- $\bowtie$ is $<, \leq, \geq$, or $>$. This is the most difficult case, and splits into four subcases.
  - If either t_i or t_j are known to be zero, the comparison reduces to the previous algorithm.
  - If an equality $t_i = d \cdot t_j$ is known, use available sign information to check if the inequality must be satisfied. At least one of the points $(d, 1)$ and $(-d, -1)$ must cohere with the available sign information for t_i and t_j; otherwise, the equality and sign information would already have been seen to be unsatisfiable. If all such coherent points satisfy $t_i \bowtie c \cdot t_j$, then that inequality is implied.
  - If the coefficient c appears in an inequality between t_i and t_j already stored in the term inequalities dictionary, check if the old inequality is stronger than the new.
  - Otherwise, the situation reduces to a small number of cross product computations as described in the previous section to determine if the new inequality is subsumed by the others.

While the `implies` routine has many cases that make it difficult to describe, it is computationally cheap to perform. As such, both the comparison assertion routine (Section 2.2.5) and the clause manager (Section 2.2.6) are able to use it extensively without slowdown.

2.2.4 Implied coefficient ranges

Given `IVars` t_i and t_j and $\bowtie \in \{<, \leq, \geq, >\}$, it can be useful to find a range of coefficients $[a, b]$ such that the Blackboard implies $t_i \bowtie c \cdot t_j$ for all $c \in [a, b]$.[2] It is not always possible to find such a range – the comparison may be implied only for a single c, or for no c at all. Alternatively, the interval may be unbounded – a could be $-\infty$, or b could be ∞. (In these cases, the left or right sides of the interval would be open.)

This information is not difficult to compute with the Halfplane representation of comparisons. Geometrically, we can equate this interval with a range of angles, such that for each angle in this range, the halfplane defined by a vector at this angle contains the feasible sector for t_i and t_j. Depending on the direction of $\bowtie$, this range of angles will fall either in the first and second or the third and fourth quadrants of the plane.

We describe the process when $\bowtie$ is $>$ or $\geq$. Given the interval $[a, b]$, there are a number of possible combinations of these strengths: we may know either a strict or a nonstrict inequality at either endpoint and in the interior. These possibilities are not independent.

[2]At the time of writing, the module that interprets the `minimum` function relies heavily on this routine.

If a strict inequality is known at either endpoint, then a strict inequality is known in the interior as well. Thus, the possible situations are:

$$\begin{array}{lll} t_i \geq a \cdot t_j; & t_i \geq b \cdot t_j; & t_i \geq c \cdot t_j \text{ for } c \in (a, b) \\ t_i > a \cdot t_j; & t_i \geq b \cdot t_j; & t_i > c \cdot t_j \text{ for } c \in (a, b) \\ t_i \geq a \cdot t_j; & t_i > b \cdot t_j; & t_i > c \cdot t_j \text{ for } c \in (a, b) \\ t_i > a \cdot t_j; & t_i > b \cdot t_j; & t_i > c \cdot t_j \text{ for } c \in (a, b) \end{array}$$

Here we omit the details of distinguishing these cases. Readers interested in these details should see the `get_ge_range` method in `blackboard.py` in the Polya source code.

Let i and j be given; we wish to determine from the Blackboard the range $[a, b]$ of c such that $t_i \geq c \cdot t_j$. We consider a number of cases. If no inequalities and no equalities are known between i and j, of course, this range is empty. Suppose that an equality $t_i = d \cdot t_j$ is known. If no sign information is available for either t_i or t_j, then the only provable inequality is $t_i \geq d \cdot t_j$, and so $a = b = d$. Knowing the sign of either t_i or t_j, along with this equality, is enough to derive the sign of the other. With this information, the range $[a, b]$ can be found quickly: in the case where t_i, t_j, and d are all positive, then $t_i \geq c \cdot t_j$ for $c \in (-\infty, d]$, and analogously for other combinations of signs. Note that knowing an equality and a noncollinear inequality implies sign information for both variables.

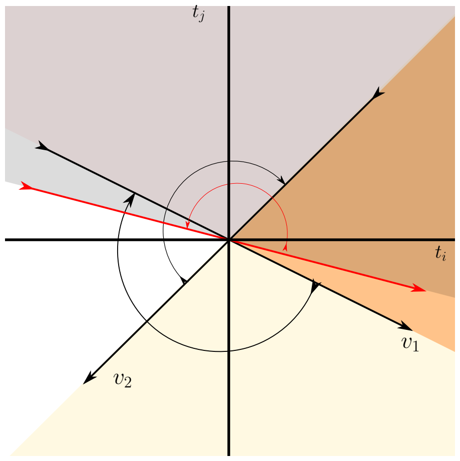
If no equality is known, and only one inequality $t_i \bowtie d \cdot t_j$ is known, then again we see $a = b = d$ exactly when $\bowtie \in \{\geq, >\}$. The difficult case comes when two inequalities $t_i \bowtie d \cdot t_j$ and $t_i \bowtie d \cdot t_j$ are known. Recall that comparisons can be represented by halfplanes, which can in turn be represented as vectors (with the convention that (x, y) stands for the halfplane counterclockwise of itself). A comparison of the form $t_i \geq c \cdot t_j$ then corresponds to a vector in the third or fourth quadrant of the plane. (Otherwise, the comparison would be $\leq$.) Let v_1 and v_2 represent the vectors corresponding to the given inequalities, such that v_2 is clockwise of v_1.

The vector $(1, 0)$ corresponds to the comparison "$t_i > -\infty \cdot t_j$" (i.e., $t_j > 0$) and the vector $(-1, 0)$ to ""$t_i > \infty \cdot t_j$" (i.e., $t_j < 0$). Our search for the interval $[a, b]$ can be pictured as a vector starting at $(1, 0)$ and "sweeping" clockwise through the fourth and third quadrants to $(-1, 0)$. If, during this sweep, the vector crosses v_1, then v_1 represents the beginning of the interval; otherwise, $(1, 0)$ does. Similarly, if the sweeping vector crosses v_2, then v_2 represents the end of the interval; otherwise, $(-1, 0)$ does. (A vector is transformed to a value for a or b by mapping $(x, y) \mapsto x/y$.) Equivalently, this process finds the intersection of the set of vectors between v_1 and v_2 with the third and fourth quadrants.

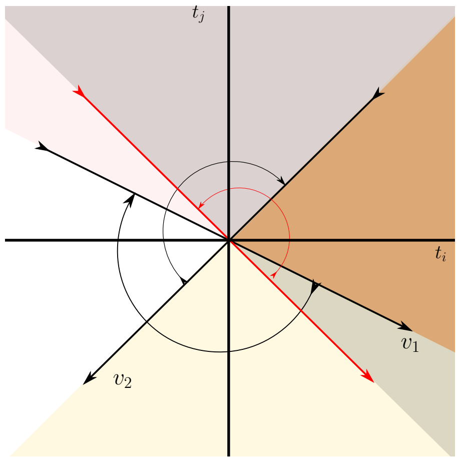
When $\bowtie$ is $\leq$, we picture the vector sweeping through the upper half-plane rather than the lower. All computations are analogous.

2.2.5 Asserting comparisons

In addition to requesting definitions and information about comparisons, modules often assert new information to the Blackboard. The Blackboard must categorize this information and, if necessary, store it appropriately.



(a) As the lighter vector sweeps from $(1, 0)$ to $(-1, 0)$, it crosses v_1 and then v_2.



(b) When it is between these two vectors, the inequality it represents is implied; the slopes of v_1 and v_2 indicate the range (a, b).

Figure 2.2: Illustrating the coefficient range search.

With the machinery defined so far, it is easy to identify redundant and contradictory information. A comparison $t_i \bowtie c \cdot t_j$ is redundant if it is implied by the Blackboard, and contradictory if its negation is implied. Otherwise, the information is new and must be recorded.

While the recording process is largely straightforward, there are a number of dependencies to track. An inequality $t_i \bowtie 0$ may affect comparisons between t_i and t_j for some or all j. Similarly, a disequality $t_i \neq c \cdot t_j$ could force a weak inequality $t_i \leq c \cdot t_j$ to become strict. The details to maintain the hierarchy of comparisons are tedious and unnecessary to list here in full, but can be seen in the Polya source code.

2.2.6 Clauses

Some routines – axiom instantiation and case-splitting, in particular – will not always learn comparisons directly, but will instead derive disjunctive (or conditional) information. It is natural to store a piece of information of this type in the Blackboard, and to update it as new comparisons are learned.

Call a disjunction of comparisons $\bigvee_{k=1}^{m} t_{1,k} \bowtie_k c_k \cdot t_{2,k}$ a *clause*. If any comparison in the clause (called a *literal*) is implied by the Blackboard, the entire clause is as well; if the negation of every literal is implied, then the clause is false. The Blackboard stores a list of clauses, maintaining that each literal is provable neither true nor false. As new information is asserted, the Blackboard updates these clauses, removing satisfied clauses and falsified literals. If a clause is reduced to a single literal, the Blackboard adds the comparison to its database; if a clause is reduced to zero literals, it is shown to be false and a contradiction has been derived.

2.3 Inference

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 2.3.

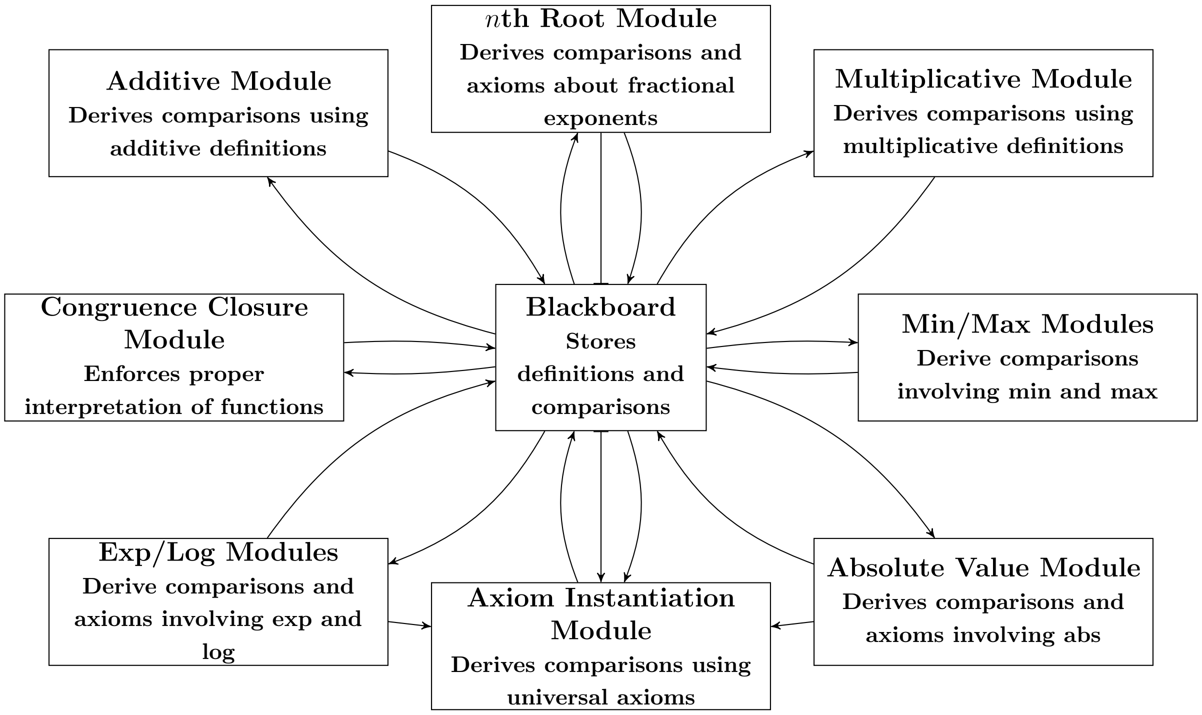

nth Root Module
Derives comparisons and axioms about fractional exponents
Additive Module
Derives comparisons using additive definitions
Multiplicative Module
Derives comparisons using multiplicative definitions
Congruence Closure Module
Enforces proper interpretation of functions
Blackboard
Stores definitions and comparisons
Min/Max Modules
Derive comparisons involving min and max
Exp/Log Modules
Derive comparisons and axioms involving exp and log
Axiom Instantiation Module
Derives comparisons using universal axioms
Absolute Value Module
Derives comparisons and axioms involving abs

Figure 2.3: The inferential structure.

Each individual module is described in detail in the following chapters. 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.

Modules are expected to *saturate* the Blackboard each time they are run: that is, running a module twice in a row should produce no new inferences the second time. Furthermore, modules should satisfy a *monotonicity* condition: if a module could infer comparisons C from a Blackboard B, and Blackboard B' is such that every comparison in B is implied by B', then the module should infer comparisons C' from B' such that $B' \cup C'$ implies each comparison in C. Note that information asserted to a Blackboard is never lost; thus, successively running modules on a Blackboard B will produce a sequence B', B'', ... of Blackboards with monotonically increasing information. Polya will accommodate modules that do not meet these conditions, but this may lead to curious or seemingly unpredictable behavior.

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. From the saturation condition, we see that if the Blackboard reaches a state where no module can learn any new information, the process terminates with no answer. Further, from the monotonicity condition, the order in which the modules are run does not affect the final answer as long as each is tried before terminating. (Of course, the order can affect the speed at which an answer is found.) Unfortunately, as long as this requirement is met, there are examples for which the process is nonterminating: see Section 5.3 for a discussion of this possibility.

Unless the user specifies otherwise, the built-in methods to compute with Polya will cycle through its modules in a fixed order. We foresee that users may want more control over the computational process, though, or may want to add additional modules (to interpret additional function symbols, say). This capacity is available, and the various interfaces are described in the following section.

2.4 User Interface

Polya is implemented in Python, and can be imported into any Python script as a library. Importing Polya gives the user access to the following classes and ways of interacting with the system.

First, users can declare variables and function objects:

```
x = Var('x')
y, z, w = Vars('y z w')
f = Func('f')
```

Note that variables always represent real numbers. Optionally, users may define a specialized canonization procedure when constructing a function object; this is described in the Polya API, available with the source code.

More complex terms can be created by combining variables, and these terms can be combined into comparison objects:

```
t1 = x + 2*y + f(z*w)
t2 = t1**2 * (4 + z)

c = (t1 <= 4*t2)
d = (x != f(y, w))
```

The user can write axioms describing the behavior of functions:

```
ax1 = Forall([x, y], Implies(x <= y, f(x) <= f(y)))
ax2 = Forall([x], And(Or(f(x)<0, x<0), Not(f(x)>10)))
```

The most immediate way to have Polya attempt to prove a theorem is to use the `solve` command.

```
solve(x > 0, y < 0, x*z < y*z)
```

This command will run Polya's default modules (see Chapters 3 and 4) in succession until either a contradiction is found, or no new information can be derived. It will return True in the former case, otherwise False.

The `solve` command can be overly simplistic for many cases; in particular, it does not allow the user to assert axioms about functions. We have implemented a `Solver` class, that will store information and run on demand:

```
s = Solver()
s.add(x > y, f(x) < f(y))
s.add(Forall([x, y], Implies(x <= y, f(x) <= f(y))))
s.check()
```

The `Solver` constructor takes optional arguments `assertions`, a list of comparisons to be added immediately; `axioms`, a list of axioms to be handled similarly; and `modules`, a list of modules to run. The latter argument will be used only in rare circumstances where the user wants to run a customized module or to avoid a default module; if the argument is not provided, the Solver will run all of the default modules in (cyclic) sequential order. At any point, the user may add an additional module to the end of the cycle, or define a new list of modules:

```
s = Solver()
m = MyNewModule()
s.append_module(m)
s.set_modules([FMAdditionModule(), m])
```

If the necessary external packages are present, the solver will default to using polyhedron arithmetic methods (Section 3.4), and otherwise default to Fourier-Motzkin methods (Section 3.2). Users may force both the `solve` method and the `Solver` class to use one method or the other by the following commands:

```
set_solver_type('poly')
s = Solver()
s.set_solver_type('fm')
```

Some problems may only be solvable by case-splitting on the directions of one or more comparisons. Most commonly, these splits occur on $x \bowtie 0$. Case splitting may be enabled in general by setting a default split depth and breadth, or by instantiating a `Solver` object with these parameters:

```
set_split_defaults(3, 10)  # depth, breadth
s = Solver(split_depth=3, split_breadth=10)
```

Finally, there are conceivable situations where the user may want more control over the Blackboard and the modules' computational order than the `Solver` class offers. Polya allows users to create and manipulate these objects directly.

```
b = Blackboard()
b.assert_comparison(x > y, x < 3*y)
b.get_le_range(1, 2)

b.assert_comparison(f(x) < f(y))

am, mm, fm = FMAdditionModule(), FMMultiplicationModule(), AxiomModule()
fm.add_axiom(Forall([x, y], Implies(x <= y, f(x) <= f(y))))

fm.update_blackboard(b)
am.update_blackboard(b)
```

Polya supports different levels of feedback via its `messages` system. At any point, the user may change this level to any of `quiet`, `modules`, `low`, `normal`, or `debug`:

```
messages.set_verbosity(messages.quiet)
```

A more complete API and example Python files are available on the project website. Most users will find the `Solver` class sufficient for their purposes.

Chapter 3

Arithmetical 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.

3.1 Problem Description

To illustrate the tasks of the arithmetical modules, we will first describe the behavior of the additive routine. The multiplicative routine is largely analogous; the small differences are noted below.

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 $t_i \bowtie ct_j$ are necessary to have complete information about their relation.[1] 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, as in the following picture.

The strongest available information is thus either the set of two derivable comparisons producing the smallest sector, or (if only one comparison is derivable) one single comparison. Of course, the strongest information between t_i and t_j implied by a collection of additive definitions and comparisons is not, necessarily, the strongest information available in the problem; perhaps the multiplicative definitions imply something stronger. While we show below that the arithmetical routines find the strongest information available to them, we must note that they do not necessarily identify which information is the strongest. (For

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

example, the Fourier-Motzkin modules may produce many comparisons between t_i and t_j, only two of which are relevant.) The database management routines in the Blackboard identify which assertions are relevant, and discard the others. We can thus picture the arithmetical modules as mindless "asserters" – they state as many true facts as they can, without knowing what will be useful.

3.2 Additive Fourier-Motzkin Module

The Fourier-Motzkin algorithm ([28], [62]) is a quantifier-elimination procedure for the theory of the structure $\langle \mathbb{R}, 0, +, < \rangle$, that is, the real numbers as an additive ordered group. Nothing changes essentially if we add to the language of that theory the constant 1 and scalar multiplication by c, for each rational c. This procedure is frequently used to decide problems in linear arithmetic by reducing sentences to a quantifier-free form. Our use of the algorithm here is slightly different: for every pair of variables t_i and t_j, we will use Fourier-Motzkin to eliminate all other variables, leaving us with a set of atomic comparisons $\{t_i \bowtie ct_j\}$. We will argue that this procedure produces the strongest available information about t_i and t_j.

While the implementation of this algorithm in Polya has small modifications for the sake of efficiency, it is clearest to describe in its basic form. Let S be the union of the given collections of additive term definitions and atomic comparisons, and suppose that we are searching for comparisons between t_i and t_j. Let $V = \{0, \ldots, n\}$ denote the indices of variables that have not yet been eliminated. The algorithm proceeds as follows:

- Rearrange all comparisons in S to have the form $\sum_{k \in V} c_k t_k \bowtie 0$, where $\bowtie \in \{=, \geq, >\}$. Note that this eliminates the distinction between definitional and learned comparisons.

- Use equalities in S to eliminate variables by substitution. For any equality $e := \sum_{k \in V} c_k t_k = 0 \in S$ with some $c_l \neq 0$, $l \notin \{i, j\}$, remove e from S and replace every occurrence of t_l in S by $\sum_{k \in V \setminus \{l\}} -c_k t_k$. Remove l from V. This strictly decreases the number of equalities in S.

- Use Fourier-Motzkin elimination to reduce the number of variables present. Pick $l \notin \{i, j\}$ and partition S into S_+, S_-, and S_0, where S_+ contains the comparisons in S in which t_l has a positive coefficient, S_- those in which t_l has a negative coefficient, and S_0 those in which t_l does not appear. For every pair of comparisons $e_+ \in S_+$, $e_- \in S_-$, find the scalar c such that t_l is eliminated from $e_+ + ce_-$, and add this combination to S_0. Set $S = S_0$ and repeat until all variables besides t_i and t_j have been eliminated.

- We are left with $S = \{c_i t_i + c_j t_j \bowtie 0\}$, a collection of (perhaps many) comparisons between only t_i and t_j. Rearrange these again to have the form of atomic comparisons, and assert each to the Blackboard.

Unfortunately, Fourier-Motzkin elimination is quite inefficient. Notice that in the third step, we partition the set S into three subsets, and consider all pairs of elements from the first and second set. If $|S_-| \approx |S_+|$ and S_0 is small, the size of the resulting set is on the order of $|S|^2$. In a problem with k inequalities in n variables, eliminating all but two of those variables results in $\mathcal{O}(k^{2^{n-2}})$ inequalities. To make matters worse, this elimination must be performed $\binom{n}{2} = \frac{n^2-n}{2}$ times for each round of the module.

Some unnecessary work can be avoided with a bit of cleverness. For instance, it is not necessary to begin with the set S containing all inequalities in all variables for each i, j pair. By choosing such i, j pairs in a diagonal order, the work done to eliminate variables between t_i and t_j need not be repeated for $i, j + 1$. Similarly, once a variable t_i has been compared to each other variable in the problem, it can be eliminated entirely. This cleverness does not reduce the computational complexity of the routine, but in practice yields noticeable improvement.

The Fourier-Motzkin routine is least efficient when a pivot variable x appears frequently in S with both positive and negative coefficients – it is in this situation that we see quadratic increase in the size of S after eliminating x. Often, though, many of the new inequalities produced are redundant. Syntactically identifying and filtering out identical inequalities is a simple task; taking a step further, it is possible to coarsely filter redundant comparisons using the simplex algorithm. As the simplex algorithm is quite efficient, doing so could at times improve the performance of the Fourier-Motzkin algorithm at little cost. Polya does not currently implement this technique, since the theoretical disadvantages of Fourier-Motzkin rarely pose a problem in practice. However, doing so would be an important step in adapting Polya to handle larger problems.

3.2.1 Proof of completeness

Here, we prove the claim that the algorithm described above produces the strongest derivable comparisons between two variables.

Theorem 1. *Let $\Gamma = \{s_k \bowtie_k t_k\}$ be a set of comparisons with $\bowtie \in \{<, \leq, =, \geq, >\}$. Then $\Gamma \models t_i \bowtie c \cdot t_j$ if and only if the Fourier-Motzkin elimination procedure on Γ produces a set of comparisons that jointly imply $t_i \bowtie c \cdot t_j$.*

Proof. We rely on the fact that the Fourier-Motzkin procedure is sound and complete: that is, if Γ is transformed to Γ' by a set of Fourier-Motzkin steps, then Γ is satisfiable if and only iff Γ' is satisfiable.

By definition, $\Gamma \models t_i \bowtie c \cdot t_j$ iff $\Gamma \cup \{t_i \not\bowtie c \cdot t_j\}$ is unsatisfiable. Let Γ' denote the result of eliminating all variables besides t_i and t_j from Γ using FM steps. Clearly, $(\Gamma \cup \{t_i \not\bowtie c \cdot t_j\})' = \Gamma' \cup \{t_i \not\bowtie c \cdot t_j\}$, and so $\Gamma \cup \{t_i \not\bowtie c \cdot t_j\}$ is unsatisfiable iff $\Gamma' \cup \{t_i \not\bowtie c \cdot t_j\}$ is unsatisfiable. By definition again, this is the case iff $\Gamma' \models t_i \bowtie c \cdot t_j$. □

3.3 Multiplicative Fourier-Motzkin Module

The Fourier-Motzkin multiplication module works analogously to the additive module. Given comparisons $t_i \bowtie c \cdot t_j$ or $t_i \bowtie 0$ and definitions of the form $t_i = \prod_j t_{k_j}^{n_j}$, the module aims to learn comparisons of the first two forms. The use of Fourier-Motzkin here is based on the observation that the structure $\langle \mathbb{R}^+, 1, \times, < \rangle$ is isomorphic to the structure $\langle \mathbb{R}, 0, +, < \rangle$ under the map $x \mapsto \log(x)$. With some translation, the usual procedure works to eliminate variables in the multiplicative setting as well. In the multiplicative setting, however, several new issues arise.

For one, the multiplicative module only makes use of terms t_i which are known to be strictly positive or strictly negative. In translating the Fourier-Motzkin routine to the multiplicative setting, it is necessary to (implicitly) take logarithms of terms, and of course the logarithm function is only defined for positive reals. With a bit of bookkeeping, it is easy to assume that all variables are positive and adjust the direction of the learned inequalities at the end; however, doing so requires knowing whether the variables are positive or negative. For this reason, the multiplicative routine is more effective in problems in which the signs of many variables are known.

To ameliorate this problem, the multiplicative module executes a preprocessing stage which tries to infer new sign information from the available data. For example, given the definition $t_4 = t_7^3 t_9 t_{11}^2$ and the sign information $t_4 > 0$ and $t_9 < 0$, one can infer $t_7 < 0$. This preprocessing does not perform any variable elimination itself, but increases the power of the main Fourier-Motzkin step.

A second issue that arises is that the inequalities that are handled by the multiplicative module are different from those handled by the additive module: terms can have a rational coefficient. For example, we may have an inequality $3t_2^2 t_5 > 1$; here, the multiplicative constant 3 would correspond to an additive term of $\log 3$ in the additive procedure. This difference makes it difficult to share code between the additive and multiplicative modules, but the rational coefficients are easy to handle.

Finally, the multiplicative elimination may produce information that cannot be asserted directly to the blackboard, such as a comparison $t_i^2 < 3t_j^2$ or $t_i^3 < 2t_j^2$. In that case, we have to pay careful attention to the signs of t_i and t_j and their relation to ± 1 to determine which facts of the form $t_i \bowtie c \cdot t_j$ can be inferred. We compute exact roots of rational numbers when possible, so a comparison $t_i^2 < 9t_j^2$ translates to $t_i < 3t_j$ when t_i and t_j are known to be positive. As a last resort, faced with a comparison like $t_i^2 < 2t_j^2$, we use a rational approximation of $\sqrt{2}$ to try to salvage useful information.

3.4 Additive Geometric Module

We have seen above that, while the Fourier-Motzkin algorithm performs well in practice on small problems, these modules are unlikely to scale well to larger problems. This inefficiency is due, in large part, to the generation of many redundant comparisons in each step and to the unavoidable repetition of many eliminations. While clever ordering and simplex filtering can reduce this somewhat, they may not always provide sufficient improvement.

A more direct approach to overcoming these issues can be seen by interpreting the problem geometrically. A linear inequality $c \leq \sum_{i=1}^{k} c_i \cdot t_i$ determines a half-space in $\mathbb{R}^{k+1}$; when $c = 0$, as in the homogenized inequalities in our current problem, the defining hyperplane of the half-space contains the origin. A set of n homogeneous inequalities determines an unbounded pyramidal polyhedron in $\mathbb{R}^k$ with vertex at the origin, called a "polyhedral cone." (See, e.g., Figure 3.1a.) Equalities, represented as $(k-1)$-dimensional hyperplanes, simply reduce the dimension of the polyhedron. The points inside this polyhedron represent solutions to the inequalities.

We will make use of the following well-known theorem of computational geometry (see [63, Section 1.1]):

Theorem 2. *A set $C \subseteq \mathbb{R}^d$ is a finite intersection of closed linear halfspaces (an $\mathcal{H}$-*polyhedron*) if and only if it is a finitely generated conical combination of vectors (a $\mathcal{V}$-*polyhedron*).*

Proof. We sketch the proof given by Ziegler; readers interested in the computational details are referred to his text. Let $P(A, \mathbf{z}) = \{\mathbf{x} \in \mathbb{R}^d : A\mathbf{x} \leq \mathbf{z}\}$ for $A \in \mathbb{R}^{m \times d}$, $\mathbf{z} \in \mathbb{R}^m$. We read $P(A, 0)$ as the $\mathcal{H}$-polyhedron generated by the homogeneous system of inequalities $A\mathbf{x} \leq 0$. Let $\text{cone}(Y) = \{t_1\mathbf{y}_1 + \ldots + t_n\mathbf{y}_n : t_i \geq 0\}$ for $Y = \{\mathbf{y}_1, \ldots, \mathbf{y}_n\} \subseteq \mathbb{R}^d$, the $\mathcal{V}$-polyhedron described by the set of vectors Y.

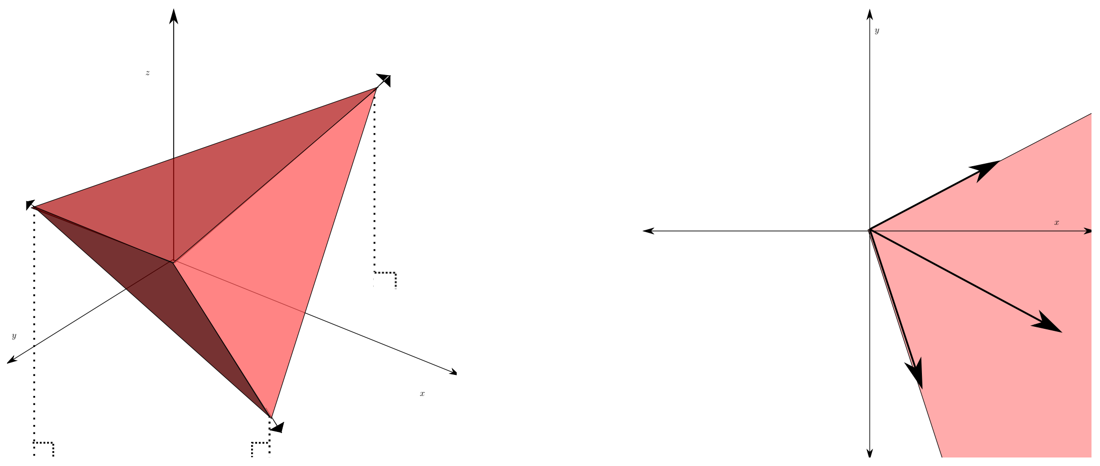
For the forward direction, let $C = P(A, 0) \subseteq \mathbb{R}^d$ be a given $\mathcal{H}$-polyhedron. By definition we have $C = \{\mathbf{x} \in \mathbb{R}^d : A\mathbf{x} \leq 0\} \cong \{(\mathbf{x}, \mathbf{w}) \in \mathbb{R}^{d+m} : A\mathbf{x} \leq \mathbf{w}\} \cap \{(\mathbf{x}, \mathbf{w}) \in \mathbb{R}^{d+m} : \mathbf{w} = 0\}$. The left-hand side of the intersection is a $\mathcal{V}$-polyhedron, and the intersection can be achieved by successively intersecting hyperplanes of the form $H_k = \{\mathbf{y} \in \mathbb{R}^{d+m} : y_k = 0\}$. It thus suffices to prove that if C is a $\mathcal{V}$-polyhedron, then so is $C \cap H_k$. This can be shown by a short (albeit finicky) algebraic computation.

For the backward direction, let $C = \text{cone}(Y) \subseteq \mathbb{R}^d$ be a given $\mathcal{V}$-polyhedron. Rewording the definition, we see $C = \{\mathbf{x} \in \mathbb{R}^d : \exists \mathbf{t} \in \mathbb{R}^n \text{ s.t. } \mathbf{t} \geq 0, \mathbf{x} = Y\mathbf{t}\}$. It is clear that $\{(\mathbf{x}, \mathbf{t}) \in \mathbb{R}^{d+n} : \mathbf{t} \geq 0, \mathbf{x} = Y\mathbf{t}\}$ is an $\mathcal{H}$-polyhedron. Furthermore, C is the projection of this polyhedron to the subspace $\{(\mathbf{x}, \mathbf{t}) \in \mathbb{R}^{d+n} : \mathbf{t} = 0\}$. Another short computation shows that projections of this sort preserve $\mathcal{H}$-polyhedrons. $\square$

This theorem can be made more general, to handle nonhomogeneous inequalities; however, we need only this specific result. We will refer to the description as an intersection of closed half-spaces as the $\mathcal{H}$-representation of a polyhedron, and similarly to the conical representation as the $\mathcal{V}$-representation.

The problem of determining the strongest comparisons between t_i and t_j reduces to finding the "extremal" ratios of the i-th and j-th coordinates of points inside the polyhedron. The notion of extremality here is subtle – one must find two maximal and/or minimal ratios relative to the signs of the ratios. (This closely parallels the discussion in Section 5.1, where the two relevant comparisons between t_i and t_j might both be $\geq$, both be $\leq$, or be one of each.) Luckily, understanding this extremality and finding these extremal ratios is fairly straightforward to do, by projecting the $\mathcal{V}$-representation of the polyhedron to the t_it_j plane. Given a nonempty list V of $k+1$-dimensional points describing the $\mathcal{V}$-polyhedron, we proceed as follows:

- For each $0 \leq i < j \leq k$, let $V' = \{(v_i, v_j) : \bar{v} = (v_0, \ldots, v_k) \in V\}$.



(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 3.1: Variable elimination by geometric projection

- If $V' = \{(0,0)\}$, then each point in the polyhedron has $t_i = t_j = 0$. Assert this to the Blackboard.

- If all points in V' are collinear, then for each point in the polyhedron, we have $t_i = ct_j$ for some scalar c. Assert this to the Blackboard.

- If all points in V' fall on the same side of some line $t_i = ct_j$, then the entire polyhedron must be contained in the half-spaces $t_i \bowtie c_1t_j, t_i \bowtie c_2t_j$, for $\bowtie \in \{\leq, \geq\}$. The scalars c_1 and c_2 can be determined by finding the points in V' with the largest angle between them, as in 3.1b. Assert these comparisons to the Blackboard.

- Otherwise, the polyhedron is not contained in any half-space determined by t_i and t_j. There is nothing to be learned about these two variables.

Intuitively, this algorithm observes the "shadow" of the polyhedron on the t_it_j plane (Figure 3.1b). If the shadow covers the entire plane (case 4), there is no useful information; otherwise, we see how to describe the 0-, 1-, or 2-dimensional shadow in terms of bounds on t_i and t_j.

The computational difficulty, of course, is found in converting the $\mathcal{H}$-polyhedron to its $\mathcal{V}$-representation. There are many ([8]) vertex enumeration algorithms to perform this conversion, optimized for different situations. (Some handle degeneracies more gracefully than others; some are sensitive to the size of the output rather than the size of the input.) Many (at least implicitly) use Fourier-Motzkin elimination in a manner similar to our modules. As this sort of geometric computation is common and well-studied, we have the luxury of relying on highly-optimized implementations of these algorithms that avoid many of the potential exponential increases in our Fourier-Motzkin modules. Additionally, since the conversion need only be performed once per round (rather than for each pair of variables), our concerns about repeating calculations vanish.

We looked into a number of vertex-enumeration methods when designing this module. Avis' *lrs* package ([7]) and Fukuda's *cdd* package ([29]) were promising candidates; other techniques (e.g. [40]) were ruled out based on their theoretical efficiencies and sensitivities. The reverse search algorithm for vertex enumeration is analogous to the well-known simplex algorithm for solving linear programs. It reverses the standard pivoting operation in order to trace paths through all vertices of the polyhedron. *lrs* implements this algorithm based on a specific lexicographic pivoting rule. Alternatively, the double-description method of Motzkin uses simple techniques to iteratively build the vertex set, adding one comparison at a time. This technique, implemented by *cdd*, handles degeneracies more naturally than the standard reverse search algorithm. While the two techniques showed similar performance on many problems, *lrs* proved to be somewhat faster. It is certainly possible that reimplementing one of these techniques with our specific purpose in mind could lead to an increase in speed; in fact, to make the procedure proof-producing, such a reimplementation will likely be necessary.

Vertex enumeration algorithms typically assume convexity of the polyhedron: that is, all inequalities are taken to be weak. As it is essential for us to distinguish between $>$ and $\geq$, we use a trick taken from Dutertre and de Moura [27, Section 5]. Namely, given a set of strict inequalities $\{0 < \sum_{i=1}^{k} c_i^m \cdot t_i : 0 \leq m \leq n\}$, we introduce a new variable δ with constraints $0 \leq \delta$ and $\{\delta \leq \sum_{i=1}^{k} c_i^m \cdot t_i : 0 \leq m \leq n\}$, and generate the corresponding polyhedron. If, in the vertex representation, every vertex has a zero δ-coordinate, then the inequalities are only satisfiable when $\delta = 0$, which implies that the system with strict inequalities is unsatisfiable. Otherwise, a comparison $t_i \bowtie c \cdot t_j$ is strict if and only if every vertex on the hyperplane $t_i = c \cdot t_j$ has a zero δ coordinate, and weak otherwise.

The geometric addition module has exactly the same output as the Fourier-Motzkin version; they only differ in their efficiency. Both run in similar (near-instantaneous) time on most of the problems in our test suite. In fact, perhaps due to the increased overhead of loading external software, the Fourier-Motzkin module is often a few milliseconds faster. They begin to diverge on larger problems, particularly when the information provided to the Blackboard is consistent. (See, for instance, Example 5.28 below. Both versions time out on this example, but the geometric module makes it through more iterations.)

3.5 Multiplicative Geometric Module

As with the Fourier-Motzkin method, multiplicative comparisons $1 \leq \prod_{i=1}^{k} t_i^{e_i}$ can be handled in a similar manner, by restricting to terms with known sign information and taking logarithms. Once again, there is a crucial difference from the additive setting: taking the logarithm of a comparison $c \cdot t_i \cdot t_j^{-1} \bowtie 1$ with $c \neq 1$, one is left with an irrational constant $\log c$, and the standard computational methods for vertex enumerations cannot perform exact computations with these terms.

To handle this situation we introduce new variables to represent the logarithms of the prime numbers occurring in these constant terms. Let $p_1, \ldots, p_l$ represent the prime factors of all constant coefficients in such a problem, and for each $1 \leq i \leq l$, let q_i be a variable representing $\log p_i$. We can then rewrite each $c \cdot t_i \cdot t_j^{-1} \bowtie 1$ as $p_1^{d_0} \cdot \ldots \cdot p_l^{d_l} \cdot t_i \cdot t_j^{-1} \bowtie 1$. Taking logarithms of all such inequalities produces a set of additive inequalities in $k + l$ variables.

In order to find the strongest comparisons between t_i and t_j, we can no longer project to the $t_i t_j$ plane, but instead look at the $t_i t_j q_1 \ldots q_l$ hyperplane. The simple arithmetical comparisons to find the two strongest comparisons are no longer applicable; we face the harder problem of converting the vertex representation of a polyhedron to a half-space representation. This problem is dual to the conversion in the opposite direction, and the same computational packages are equipped to solve it. Experimentally, we have found *cdd* to be faster than *lrs* for this procedure.

Unfortunately, this technique makes the efficiency of the multiplicative module sensitive to the size and number of factors of the constant coefficients involved. A problem with many unrelated large numbers can have significantly more prime number variables than `IVars`. While we have not in practice seen this lead to a slowdown compared to the Fourier-Moztkin routine, it does limit the module's scalability. In principle, the vertex enumeration algorithms do not require coefficients to be rational, and implementing these algorithms symbolically would allow us to compute without this translation.

Chapter 4

Additional Modules

The arithmetical modules form Polya's computational core, and solve problems in the language of RCF. However, Polya's term structure and modular nature allow for more expressivity than this: most importantly, we can define and reason with symbols denoting arbitrary functions.

No interpretations are initially given to function symbols; the functions are simply assumed to take values in $\mathbb{R}$. (In fact, function symbols need not even have a fixed arity. The `minm` function takes an arbitrary number of arguments, for example.) Users are allowed (but not required) to introduce custom canonical forms for particular function symbols, to overwrite the default form described in Section 2.1. They can additionally assert universal axioms that restrict the interpretation of a given function symbol, as described in the following section.

Polya has built-in interpretations for a number of function symbols. Exponentials and logarithms, minima and maxima, and absolute values are all handled natively. Individual modules are equipped to reason specifically with each function symbol.

4.1 Axiom Instantiation Module

Users are allowed to define universally instantiated axioms. A valid axiom has the form $(\forall x_1, \ldots, x_n)\varphi$. 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 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. The notion of an "appropriate" instantiation is rather difficult to pin down. Naively, one might try all possible mappings of quantified variables to problem terms. It is easy to see, though, that this approach is too eager. Consider a problem with one variable a, and the axiom $(\forall x)(f(x) > 0)$. Iterated applications of the naive approach will assert $f(a) > 0$, $f(f(a)) > 0$, $f(f(f(a))) > 0$, and so on, which will help find a proof only in exceedingly rare situations. In fact, this approach can also miss useful instantiations. Suppose $f(a) < 1/2a$ with the axiom $(\forall x)(f(2x) > x)$. The module will learn that $f(2a) > a$, $f(2f(2a)) > f(2a)$, etc, but will never try the assignment $x \mapsto 1/2a$.

To avoid this, one might try to instantiate an axiom only with substitutions that map terms in the axiom to terms defined in the Blackboard. But this approach is too cautious. Suppose $a < b$ and $f(a) > f(b)$, with the axiom that $(\forall x, y)(x < y \to f(x) - f(y) < 0)$. Since the term $f(a) - f(b)$ is not present in the Blackboard, the instantiation $\{x \mapsto a, y \mapsto b\}$ will fail; adding this term, though, would allow the arithmetical modules to derive a contradiction. The "appropriate" approach to selecting instantiations must lie somewhere in between.

We find this middle ground by specifying for each axiom a set of *trigger terms* ([25]). An assignment $\{x_i \mapsto c \cdot t_j\}$ is appropriate when it maps each trigger term to a (constant multiple of a) problem term. By default, the trigger terms are chosen to be the functional subterms found in the axiom; however, for some axioms the user may wish to manually define the trigger terms. The capacity to do so is present but limited in the current implementation, and strengthening it is a potential improvement. Here we assume that the trigger terms are chosen by the default rule.

We further require that each quantified variable occurs alone (up to a constant multiple) as an argument to some function in the axiom. That is, neither $(\forall x, y)(f(x) < y)$ nor $(\forall x, y)(f(xy) > 0)$ would be acceptable, but $(\forall x, y)(f(x, y) > 0)$ would be. This requirement is not strictly necessary, but it drastically simplifies the unification procedure during instantiation, and most reasonable axioms are expressible in this form. (Note that the first example given is clearly inconsistent, and the second is more clearly written as $(\forall x)(f(x) > 0)$.)

Given an axiom $(\forall x_1, \ldots, x_n)\varphi$ in CNF meeting these constraints, and a Blackboard B, the axiom instantiation module updates B in a two-step process:

1. *Unify* the set of trigger terms of the axiom with the term definitions in B. If successful, this process produces a list of assignments mapping $x_1, \ldots, x_n$ to scalar multiples of terms in B.

2. For each assignment, *substitute* terms accordingly into φ and *reduce* φ to a clause of the form $\bigvee t_i \bowtie c_{i,j} \cdot t_j$. This may involve introducing new terms to B.

The unification routine depends heavily on a *term-matching* subroutine `match`, which we describe first. Relative to a Blackboard B, `match` takes as input a term t whose atoms are `IVars`, and returns a rational c and integer i such that B implies $t = c \cdot t_i$. We assume here that t had a coefficient of 1, since any other coefficient could be factored into c.

- The simple case occurs when t is syntactically equal to a term already defined in B. In this case, `match` returns the index of this term, along with the appropriate coefficient.

- Otherwise, `match` splits based on the type of t. If t is a function term, it can only match other function terms with the same name and arity.

  - Recursively call `match` on each argument of t, to get $t = f(c_1 \cdot t_{i_1}, \ldots, c_k \cdot t_{i_k})$. If any of these calls to `match` fail, then fail.
  - Otherwise, look at each term $s = f(d_1 \cdot t_{j_1}, \ldots, d_k \cdot t_{j_k})$ defined in B. If B implies that $c_m \cdot t_{i_m} = d_m \cdot t_{j_m}$ for each $1 \leq m \leq k$, then $t = s$; return the index of s with the appropriate coefficient. Otherwise, fail.

- If t is an additive term, `match` must try to reduce t to a single `IVar` by substituting equalities known in B. For an example, let $t = 3t_1 + 2t_2 + 4t_3$, and suppose this is not syntactically equal to any t_i defined in B. If B contains the equalities $t_2 = 10t_1$ and $t_3 = -t_2$, `match` should recognize that $t = -17t_1$. This amounts to performing, for each i, a sequence of Gaussian elimination steps on the set of additive equalities in B, with the goal of finding $t = c \cdot t_i$. If this elimination succeeds for some i, return i along with the appropriate coefficient. Otherwise, fail.

- If t is a multiplicative term, proceed analogously to the additive case: Gaussian elimination can be adapted to multiplication. Here we face a similar restriction to what we saw in the multiplicative modules, namely that we can only perform this elimination on variables known to be nonzero.

The `unify` routine takes as arguments a Blackboard B and a set of terms T. Let the word `UVar` denote a free variable x_i in T. `unify` returns a set of (partial) assignments $\{s_1, \ldots, s_n\}$, where each s_i maps $x_i \mapsto c_i \cdot t_{j_i}$ for some collection of `UVars` $\{x_i\}$.

- Pick a `UVar` u that occurs in T as the jth argument to some function term f: that is, the term $f(\ldots, c \cdot u, \ldots)$ appears in T.

- Let $\mathcal{F}$ be the set of all function terms defined in B with the same name and arity as f. For $g \in \mathcal{F}$, let c_g be a rational and t_g be a problem term, such that $c \cdot c_g \cdot t_g$ occurs as an argument to g in the same place that $c \cdot u$ occurs as an argument to f.

- Let S denote the (currently empty) list of assignments to return.
  - For each $g \in \mathcal{F}$, $c_g \cdot t_g$ represents a potential assignment for u. Let $\mathcal{O}$ be the set of terms in $\{t[c_g \cdot t_g/u] : t \in T\}$ that have remaining `UVars`, and $\mathcal{C}$ be the set of those for which all `UVars` have been eliminated.
  - If `match` fails for some term in $\mathcal{C}$, this assignment is not feasible; continue to the next $g \in \mathcal{F}$. Otherwise, each term in $\mathcal{C}$ matches some problem term in B, and this assignment is a good candidate.
  - The unification process has ended if $\mathcal{O}$ is empty. In this case, add the assignment $u \mapsto c_g \cdot t_g$ to S. Otherwise, recursively call `unify` on $\mathcal{O}$, and add $u \mapsto c_g \cdot t_g$ to each of the resulting assignments. Add these assignments to S.

- If S is empty, there is no variable assignment that unifies T with the terms in B; `unify` has failed. Otherwise, return the set of plausible assignments S.

Finally, the `instantiate` routine takes an axiom $A = (\forall x_1, \ldots, x_n)\varphi$ with φ in CNF and a Blackboard B, and asserts a number of clauses to B.

- Let S be the set of assignments obtained by unifying the trigger terms of A with B. If this unification fails, there are no plausible instantiations of A.
- For each variable assignment in S:
  - Perform the substitution on all literals of φ.
  - For each literal $s \bowtie c \cdot t$, try to `match` s and t to terms defined in B. If `match` fails, add the term to B as a new term definition. The literal can now be put in the form $t_{j_1} \bowtie c \cdot t_{j_2}$.
  - Now, φ is of the form $\bigvee_{i=1}^{k} t_{j_i} \bowtie c \cdot t_{j_i}$. This is precisely a clause (Section 2.2.6) that can be asserted to B.

4.2 Congruence closure module

Polya's blackboard does not enforce that a function must have the same output given equal inputs. This property is known as *congruence closure*. The well-known union-find data structure and its variations (e.g. [23], [47]) provides an efficient way to maintain these equalities in a database. This maintenance is not a bottleneck for our algorithm, though, and a more naive approach works seamlessly. Polya runs a congruence closure module that searches for pairs of problem terms with the same function symbol and arity. If the Blackboard implies that each corresponding pair of arguments are equal, the module asserts that the terms are themselves equal. The runtime of this module is negligible compared to that of the arithmetical modules, so implementing a more structured method is not a priority.

4.3 nth Root Module

Handling terms such as $x^{1/2}$ can be difficult, as this expression is undefined when $x < 0$. The canonization method must take care to avoid unsound simplifications such as $x^{1/2} \cdot x^{1/2} \to x$. Nevertheless, when $x > 0$ is known, many inferences can be made about $x^{1/2}$, and these inferences can be vital for completing a proof. It is convenient to treat fractional exponents $(\cdot)^{1/n}$ as instances of a function term $\texttt{root}(n, \cdot)$. Reasoning with these functions can largely be handled by the axiom instantiation module, such that for even n, inferences about $\texttt{root}(n, t)$ will be made only if t is known to be positive.

The nth root module guarantees that the proper axioms for a given problem have been added to the Blackboard. The module finds a list of n such that $\texttt{root}(n, s)$ appears as some problem term, and axiomatizes the behavior of $\texttt{root}(n, \cdot)$ appropriately for each n. If n is even, the axioms

$$(\forall x)\,(x \geq 0 \to \texttt{root}(n, x)^n = x)$$
$$(\forall x)\,(x \geq 0 \to \texttt{root}(n, x) \geq 0)$$

are added to the instantiation module. If n is odd, the axiom

$$(\forall x)\,(\texttt{root}(n,x)^n = x)$$

is added. These conditional identities provide a sound way of reasoning with fractional exponents.

4.4 Exponential and Logarithm Module

Without computing any exact or approximate values, we can describe the exponential function $\texttt{exp}(x) = e^x$ as a positive, strictly increasing function defined on all of $\mathbb{R}$. The module which interprets this function thus adds the following axioms to the instantiation module:

$$\begin{aligned}
(\forall x)\,&(\texttt{exp}(x) > 0)\\
(\forall x)\,&(\texttt{exp}(x) > x)\\
(\forall x)\,&(x \geq 0 \to \texttt{exp}(x) \geq 1)\\
(\forall x)\,&(x > 0 \to \texttt{exp}(x) > 1)\\
(\forall x,y)\,&(x \leq y \to \texttt{exp}(x) \leq \texttt{exp}(y))\\
(\forall x,y)\,&(x < y \to \texttt{exp}(x) < \texttt{exp}(y))\\
(\forall x,y)\,&(x \neq y \to \texttt{exp}(x) \neq \texttt{exp}(y))
\end{aligned}$$

Additionally, the exponential function satisfies the identities

$$\begin{aligned}
\texttt{exp}(c \cdot x) &= \texttt{exp}(x)^c\\
\texttt{exp}(x_1 + \ldots + x_n) &= \texttt{exp}(x_1) \cdot \ldots \cdot \texttt{exp}(x_n)
\end{aligned}$$

for scalar c. These cannot be axiomatized in a way that the instantiation module will recognize, so the exponential module must search for and add these identities itself.

The natural logarithm function $\texttt{log}$ is dual to $\texttt{exp}$ and is axiomatized similarly. However, since $\texttt{log}$ is only defined on the positive reals, we must condition our axioms as we did in the *nth* root module.

$$\begin{aligned}
(\forall x)\,&(x \geq 1 \to \texttt{log}(x) \geq 0)\\
(\forall x)\,&(x > 1 \to \texttt{log}(x) > 0)\\
(\forall x)\,&(x > 0 \to \texttt{log}(x) < x)\\
(\forall x,y)\,&(0 < x \wedge x < y \to \texttt{log}(x) < \texttt{log}(y))\\
(\forall x,y)\,&(0 < x \wedge x \leq y \to \texttt{log}(x) \leq \texttt{log}(y))\\
(\forall x,y)\,&(0 < x \wedge 0 \leq y \wedge x \neq y \to \texttt{log}(x) \neq \texttt{log}(y))
\end{aligned}$$

We also have the identities

$$\begin{aligned}
\texttt{log}(x^c) &= c \cdot \texttt{log}(x)\\
\texttt{log}(x_1 \cdot \ldots \cdot x_n) &= \texttt{log}(x_1) + \ldots + \texttt{log}(x_n)
\end{aligned}$$

for scalar c, provided $x, x_1, \ldots, x_n$ are all positive.

4.5 Minimum and Maximum Module

The minimum function $\texttt{minm}(x_1, \ldots, x_k)$ is interpreted in the standard way: it returns the value of one of its arguments x_i such that $x_i \leq x_j$ for all $1 \leq j \leq k$. This function presents an interesting challenge to Polya, since its arity is not fixed; this makes it nearly impossible to describe to the general axiom instantiation module. A specialized module allows us to interpret it.

The minimum module searches a Blackboard B for terms $t := \texttt{minm}(c_1 \cdot t_1, \ldots, c_k \cdot t_k)$. It immediately asserts that $t \leq c_i \cdot t_i$ for $1 \leq i \leq k$. As it is useful to find as much sign information as possible for the multiplicative module, the minimum module also checks for $\bowtie \in \{<, \leq, \geq, >\}$ if $c_i \bowtie 0$ for all i; if so, it asserts that $t \bowtie 0$ as well.

The module must also account for the fact that $t = \texttt{minm}(c_1 \cdot t_1, \ldots, c_k \cdot t_k)$ is the *smallest* number less than or equal to all of its arguments. If for some constant d and problem term s we have $s \leq d \cdot c_j \cdot t_j$ for all $1 \leq j \leq k$, then we also know that $s \leq d \cdot t$. The minimum module uses the Blackboard's methods for finding implied coefficient ranges (Section 2.2.4) to find, for each problem term s, an interval $[a, b]$ for which $b \in [a, b]$ implies $s \leq d \cdot c_j \cdot t_j$ holds for all $1 \leq j \leq k$. If such an interval exists, the module asserts that $s \leq a \cdot t$ and $s \leq b \cdot t$. (In fact, the module may also determine that this inequality should be strict.)

Conveniently, we do not need a separate module to handle maxima. Polya defines the `maxm` function to be the dual of `minm`: $\texttt{maxm}(c_1 \cdot t_1, \ldots, c_k \cdot t_k) = -\texttt{minm}(-c_1 \cdot t_1, \ldots, -c_k \cdot t_k)$.

4.6 Absolute Value Module

Finally, Polya has a specialized module for interpreting the absolute value function. Basic properties of `abs` are handled by asserting the following axioms to the function module:

$$\begin{aligned}
&(\forall x)(\texttt{abs}(x) \geq 0)\\
&(\forall x)(\texttt{abs}(x) \geq x)\\
&(\forall x)(\texttt{abs}(x) \geq -x)\\
&(\forall x)(x \geq 0 \to \texttt{abs}(x) = x)\\
&(\forall x)(x \leq 0 \to \texttt{abs}(x) = -x).
\end{aligned}$$

It is not possible to axiomatize the triangle inequality in full generality, though, as the form of such an axiom would not meet the restrictions on the instantiation module. The purpose of the absolute value module is to find and perform promising instantiations of the triangle inequality. Specifically, the module adds comparisons of the forms

$$\begin{aligned}
&\texttt{abs}(c_1t_1 + \ldots + c_kt_k) \leq \texttt{abs}(c_1t_1) + \ldots + \texttt{abs}(c_kt_k)\\
&\texttt{abs}(c_1t_1 + \ldots + c_kt_k) \geq \texttt{abs}(c_jt_j) - (\texttt{abs}(c_1t_1) + \ldots + \texttt{abs}(c_kt_k))\,.
\end{aligned}$$

It would be unproductive to add these comparisons indiscriminately. Doing so would necessitate creating new problem terms $\texttt{abs}(c_jt_j)$ for each argument, if these terms were not already present. The absolute value module takes a more subtle approach, only learning these comparisons if for each j, either $\texttt{abs}(c_j t_j)$ is already a problem term, or the sign of j is known (in which case $\texttt{abs}(c_j t_j)$ is replaced with $\pm c_j t_j$ as appropriate). This approach does not seem to miss any inferences that the indiscriminate approach would capture, since the comparisons learned will only be useful if something is known about each absolute value.

Chapter 5

Performance and Examples

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, such as the example used for motivation in Section 1.4, 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.

In the following sections, we highlight some of the noteworthy inferences that 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.

5.1 Comparisons With Other Systems

Before seeing how Polya performs in comparison to other automated provers, it is important to identify the competition. Automated arithmetic outside of $\mathbb{T}_{RCF}$ (and even inside it – see the discussion of cylindrical algebraic decomposition in Section 1.2) is a difficult domain, and there are few systems that are strong in this area. A few promising candidates are Z3 ([24]), an SMT solver developed at Microsoft Research, and MetiTarski ([1]), a resolution theorem prover by Lawrence Paulson.

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 below show, 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 5.4 below, on which Polya's arithmetical capacity shines.

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 was fairly successful on our purely arithmetical examples, it had similar weaknesses to Z3. It did not perform well on examples with interpreted functions, including the examples that involve tight inferences about exp.

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 ([43], [13]), 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.

Finally, ACL2 has support for nonlinear reasoning (see e.g. [39]). The method used there is locally somewhat similar to ours, although it lacks the same global guidance. Preliminarily, ACL2 appears to solve some, but not all, of the problems in our example suite, and requires somewhat more input to do so. We hope to have a more detailed comparison between Polya and ACL2 in the future.

5.2 Polya's Successes

Here we walk through a number of examples that we have chosen to illustrate Polya's strengths. We indicate how Z3 and Isabelle's methods perform in comparison.

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

$$0 < u < v < 1,\ 2 \le x \le y \ \Rightarrow\ 2u^2x < vy^2, \tag{5.1}$$

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

$$x > 1 \ \Rightarrow\ (1+y^2)x > 1+y^2 \tag{5.2}$$

$$0 < x < 1 \ \Rightarrow\ 1/(1-x) > 1/(1-x^2) \tag{5.3}$$

$$0 < u,\ u < v,\ 0 < z,\ z+1 < w \ \Rightarrow\ (u+v+z)^3 < (u+v+w)^5 \tag{5.4}$$

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

$$(\forall x.\ f(x) \leq 1),\ u < v,\ 0 < w \ \Rightarrow\ u + w \cdot f(x) < v + w \tag{5.5}$$

$$(\forall x, y.\ x \leq y \to f(x) \leq f(y)),\ u < v,\ x < y \ \Rightarrow\ u + f(x) < v + f(y) \tag{5.6}$$

Isabelle's auto and Sledgehammer fail on all of these but (5.5) and (5.6), which are proved by resolution theorem provers. Sledgehammer can verify more complicated variants of (5.5) and (5.6) by sending them to Z3, but fails on only slightly altered examples, such as:

$$(\forall x.\ f(x) \leq 2),\ u < v,\ 0 < w \ \Rightarrow\ u + w \cdot (f(x) - 1) < v + w \tag{5.7}$$

$$\begin{aligned}(\forall x, y.\ x \leq y \to f(x) \leq f(y)),\ u < v,\ 1 < v,\ x \leq y \ \Rightarrow& \\ u + f(x) \leq v^2 + f(y)&\end{aligned} \tag{5.8}$$

$$\begin{aligned}(\forall x, y.\ x \leq y \to f(x) \leq f(y)),\ u < v,\ 1 < w,\ 2 < s,& \\ (w + s)/3 < v,\ x \leq y \ \Rightarrow\ u + f(x) \leq v^2 + f(y)&\end{aligned} \tag{5.9}$$

Z3 gets most of these when called directly, but also fails on (5.8) and (5.9). Moreover, when handling nonlinear equations, Z3 "flattens" polynomials, which makes a problem like (5.4) 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 (5.4). 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

$$z > \texttt{exp}(x), w > \texttt{exp}(y) \ \Rightarrow\ z^3 \cdot w^2 > \texttt{exp}(3x + 2y) \tag{5.10}$$

$$a > 1, c > 0, \texttt{log}(b^2) > 4, \texttt{log}(c) > 1, b \neq 0 \ \Rightarrow\ \texttt{log}(a \cdot b^2 \cdot c^3) > 7 \tag{5.11}$$

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.

Polya has no problem with examples such as

$$0 < x < y,\ u < v \ \Rightarrow\ 2u + \texttt{exp}(1 + x + x^4) < 2v + \texttt{exp}(1 + y + y^4), \tag{5.12}$$

mentioned in the introduction. Sledgehammer verifies this using resolution, and slightly more complicated examples by calling Z3 with the monotonicity of `exp`. Sledgehammer restricts Z3 to linear arithmetic so that it can reconstruct proofs in Isabelle, so to verify (5.12) it provides Z3 with the monotonicity of the power function as well. When called directly on this problem with this same information, however, Z3 resorts to nonlinear mode, and fails.

Sledgehammer fails on an example that arose in connection with a formalization of the Prime Number Theorem, discussed in [3]:

$$0 \leq n,\ n < (K/2)x,\ 0 < C,\ 0 < \varepsilon < 1 \ \Rightarrow\ \left(1 + \frac{\varepsilon}{3(C + 3)}\right) \cdot n < Kx \tag{5.13}$$

Z3 verifies it when called directly. Sledgehammer also fails on these [4]:

$$0 < x < y \;\Rightarrow\; (1+x^2)/(2+y)^{17} < (1+y^2)/(2+x)^{10} \tag{5.14}$$

$$0 < x < y \;\Rightarrow\; (1+x^2)/(2+\texttt{exp}(y)) \geq (2+y^2)/(1+\texttt{exp}(x))\ . \tag{5.15}$$

Z3 gets (5.14) but not (5.15). Neither Sledgehammer nor Z3 get these:

$$(\forall x,y.\ f(x+y) = f(x)f(y)),\ a > 2,\ b > 2 \;\Rightarrow\; f(a+b) > 4 \tag{5.16}$$

$$(\forall x,y.\ f(x+y) = f(x)f(y)),\ a+b > 2,\ c+d > 2 \;\Rightarrow\; f(a+b+c+d) > 4 \tag{5.17}$$

Polya verifies all of the above easily.

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

$$x > 0, y > 0, y < 1 \;\Rightarrow\; (x+y) > xy\ . \tag{5.18}$$

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 ([5]). Billingsley [12, 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:

$$\begin{aligned} &i \geq 0,\ |f(y) - f(x)| < 1/(2(i+1)),\\ &\qquad |f(z) - f(y)| < 1/(2(i+1)) \;\Rightarrow\; |f(x) - f(y)| < 1/(i+1)\ . \end{aligned} \tag{5.19}$$

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

$$\begin{aligned} &(\forall m.\ m > 0 \to f(\lceil m \rceil) < a + (b-a)/\lceil m \rceil),\\ &\qquad a < b,\ x > a,\ m \geq (b-a)/(x-a) \;\Rightarrow\; f(\lceil m \rceil) < x\ . \end{aligned} \tag{5.20}$$

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.

Finally, Polya succeeds in verifying examples that involve combinations of interpreted functions:

$$\texttt{minm}(\texttt{exp}(3x), \texttt{exp}(9x^2 - 2), \texttt{log}(x)) > 1, x > 0 \;\Rightarrow\; x > 1 \tag{5.21}$$

$$y > \texttt{maxm}(2, 3x), x > 0 \;\Rightarrow\; \texttt{exp}(4y - 3x) > \texttt{exp}(6) \tag{5.22}$$

$$x \neq 0, y > 0, \texttt{log}(|x| + 2|y|) > 5, \texttt{log}(|y|) < \sqrt{2} \;\Rightarrow\; \texttt{log}(|x|) > 2 \tag{5.23}$$

$$x \neq 0, y > 0, \texttt{log}(|x| + 2|y|) > 5, \texttt{log}(|y|) < \sqrt{2} \;\Rightarrow\; \texttt{log}(\texttt{exp}(x)) > \texttt{exp}(-2) \tag{5.24}$$

5.3 Polya's Shortcomings

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. When called directly, the module fails to make the trivial inference

$$x > 0,\ y < z \ \Rightarrow\ xy < xz\ . \tag{5.25}$$

The preprocessing step described in Section 3.3 enables Polya to prove this inference. But this preprocessing is not robust, and minor adjustments cause Polya to fail:

$$x > 0,\ xyz < 0,\ xw > 0 \ \Rightarrow\ w > yz \tag{5.26}$$

Problems of this sort are easily solved given a mechanism for splitting on the signs of w, y and z. See Section 6.3 below for a discussion of this.

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

$$x^2 + 2x + 1 \geq 0, \tag{5.27}$$

which can also be obtained simply by writing the left-hand side as $(x+1)^2$. But, as pointed out by Avigad and Friedman [4], the method implemented by Polya is, in fact, nonterminating 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.

Finally, there are examples on which the arithmetical modules reach their computational limits. We have seen this happen particularly on large, satisfiable problems and problems involving complicated rationals, but the unfortunate behavior is not limited to these cases. Consider the following, taken from Solovyev's work on the Flyspeck project ([57]):

$$\begin{aligned} &4 \leq x_i \leq 6.3504 \ \Rightarrow \\ &\qquad x_1x_4(-x_1 + x_2 + x_3 - x_4 + x_5 + x_6) + x_2x_5(x_1 - x_2 + x_3 + x_4 - x_5 + x_6) \\ &\qquad + x_3x_6(x_1 + x_2 - x_3 + x_4 + x_5 + -x_6) - x_2x_3x_4 - x_1x_3x_5 - x_1x_2x_6 - x_4x_5x_6 > 0 \end{aligned} \tag{5.28}$$

As the arithmetical modules search for a contradiction, they derive more and more comparisons with large (fractional) coefficients. Both the Fourier-Motzkin and the geometric elimination routines are prone to timing out on these problems; this appears to be the system's main bottleneck.

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. We look into some of these improvements in the final chapter.

Chapter 6

Conclusions and Future Work

As we have seen in the last chapter, Polya is currently able to prove many interesting theorems. Of course, there is always room to improve. Most modifications to our system can be put into one of two categories: changes that improve *efficiency*, and changes that increase the *scope* of problems that we can solve. We discuss both of these below.

6.1 Producing Proofs

Perhaps the most important improvement, though, is to extend Polya to produce proof certificates; this does not fit cleanly into either category. (In fact, a proof-producing version would certainly be slower than the current implementation.) Producing certificates is extremely desirable for integration into interactive theorem provers. What's more, doing so would help ensure the correctness of our implementation. We have found and fixed countless bugs over the course of development, and there are doubtlessly many more lurking; with certified results, our code becomes much more trustworthy.

Because of the simple nature of Polya's inferences, doing so should not require huge amounts of extra overhead. It is an easy matter to trace the history of any comparison asserted to a Blackboard. The comparison must come either from the user – as a hypothesis to the problem – or from a computational module. The arithmetic modules create comparisons by taking linear combinations of others, and so the source of a new comparison can be represented by its two predecessors and a coefficient (exponent) of combination. The other modules behave similarly; for instance, comparisons produced by the axiom instantiation module can be traced to an axiom (assumed or provable) and a specific instantiation.

Lean, a new proof assistant under development by Leonardo de Moura, is a promising environment in which to integrate Polya. Instead of having an "interactive core" with mechanized tactics added on, Lean integrates powerful automated techniques from the bottom up. It can be seen as an interactive proof assistant with many automated tactics, or as an automated tool that produces proofs based on a verified library. Using a scripting language called Lua, Lean users will be able to design new tactics of their own on top of the built-in capabilities.

We envisage Polya running as normal, tracking this historical structure for every assertion. If and when the system derives a contradiction, identifying its source is just a matter of looking down all branches of the tree; all the necessary information to construct a formal proof is present. This trace can be given to a proof-producing algorithm – say, a Lua tactic for Lean – that will reconstruct this proof. When Lean and its libraries are sufficiently robust, we plan to implement exactly this process.

6.2 Improving Efficiency

The current Python implementation of Polya is a prototype, and much of the code could be further optimized. We would also expect to see great increases in efficiency by implementing the system in another programming language such as OCaml or C++. These optimizations are not of theoretical interest, though, and so we focus here on more substantial improvements.

In their current designs, the arithmetical modules start from scratch each time they are run. One can imagine incremental ways of running both the Fourier-Motzkin and geometric elimination routines, so that little work is repeated. In the Fourier-Motzkin modules, this would involve storing the sets of inequalities generated at each stage and extending the technique to begin partway through. In the geometric modules, this would involve generating an initial polyhedron and intersecting it with new half-planes as new information is added. As doing this does not seem to be a feature of *cdd* or *lrs*, adding this capability to Polya would likely involve implementing our own vertex enumeration method.

An original implementation of a vertex enumeration algorithm would allow for other improvements as well. The "hacks" we use to account for strict inequalities and irrational coefficients (see Sections 3.4 and 3.5) could be avoided by handling these situations naturally. Furthermore, the problems Polya sends to its geometric tools are purely homogeneous, yet *cdd* and *lrs* are designed to handle problems with constants; it is possible that restricting the algorithms to this class of problems could result in greater efficiency.

We saw the arithmetical modules reach their computational limits in Example 5.28 above. Improving the modules as described might enable them to finish this example, but one could always construct larger examples with more variables and more complicated coefficients. Notably, the kind of slowdown we see in this example happens most often on "satisfiable" problems – that is, when no amount of computation would allow Polya to derive a contradiction.[1] This suggests running Polya in parallel with a model-search technique. On satisfiable problems, successful completion of the model search would terminate Polya before these slowdowns could occur.

Like the arithmetical modules, the hierarchical data storage in the Blackboard is also highly nonincremental. In order to properly and efficiently implement case-splitting and backtracking, it will be necessary to better track the cascading effects of asserting a new

[1]We point out a subtlety in this notion of satisfiability: a problem unsatisfiable over $\mathbb{R}$ may not appear contradictory to Polya, if deriving a contradiction necessarily involves applying distributivity. Polya searches for unsatisfiability over $\mathbb{T}[\mathbb{Q}]$.

comparison. (Currently, the only way to "undo" an assertion is to store a copy of the Blackboard before the assertion is made, and revert to that.) And despite maintaining a hierarchical structure, the Blackboard still stores plenty of redundant information, leading to inefficiencies in both storage space and computation time.

Ultimately, though, the current and unoptimized version of Polya runs quite well on realistic examples. While we naturally expect computation times to increase in a proof-producing version, improving efficiency seems less of an imminent need than widening the class of problems Polya can solve.

6.3 Solving More Problems

Recall that on problems like Example 5.26 above, the multiplicative arithmetic modules fail due to a lack of sign information. We can get around this limitation to some extent by preprocessing multiplicative terms. But in general, the principled way to handle these situations is with case splits. If Polya is able to derive a contradiction assuming each of $y > 0$, $y = 0$, and $y < 0$, then it can conclude that the problem is unsatisfiable. And assuming any one of these will allow the multiplicative module to do its job on Example 5.26. Case splits on the signs of terms seem especially fruitful in our framework, given the multiplicative modules' limitations. But in other cases it may be helpful to split on $x < c \cdot y$, $x = c \cdot y$, and $x > c \cdot y$ for arbitrary terms x and y and coefficient c – perhaps one of these assumptions might satisfy the hypothesis of an axiom, say.

The current implementation of Polya is able perform very rudimentary case splitting when the modules cannot derive any new comparisons. Variables are ranked based on the frequency with which they appear in multiplicative terms, and case splits are made in this order up to a fixed depth d. If the system is still unable to derive a contradiction after making d assumptions, it will revert to a previous state and try to split on different variables.

Without efficient methods to incrementally update and backtrack in the Blackboard, this method of case splitting is extremely costly. It will often be the case that assuming $y > 0$ will result in the same conclusions regardless of whether $x < 0$ or $x > 0$ was assumed before, but our current implementation cannot recognize this or avoid recalculating these conclusions.

More generally, an efficient case-splitting routine would involve conflict-driven clause learning (CDCL). This technique, used extensively by DPLL-based SAT solvers, involves finding "core conflicts" when a case split leads to a contradiction and using this information in future assignments. For example, a case split that chooses to assign $w = 0$, $x > 0$, $y > 0$, and $z < 0$ may derive a contradiction that depends only on the assumptions about w and z; knowing this restricts the space of feasible assignments that the system must check. Implementing CDCL is, of course, closely related to the incrementality concerns discussed above.

We have also considered improvements to Polya that are unrelated to case-splitting. One glaring weakness of the system is its inability to understand distributivity, as illustrated in Example 5.27. This limitation comes from the interplay between the additive and multiplicative arithmetical modules, but nothing prevents another module from stepping in: we imagine a "distribution" module, which heuristically distributes or factors terms and adds identities as appropriate.

Finally, mathematicians frequently need to prove inequalities involving higher-order operators: combinatorialists may seek to bound finite sums, or analysts integrals. The heuristic techniques of Polya could be extended to reason with these types of terms as well. For instance, if f is known to be positive and $n < m$ are integers, then $\sum_{i=1}^{n} f(i) < \sum_{i=1}^{m} f(i)$, and $\int_{n}^{m} f(x)dx > 0$. Handling these sorts of operators could be very fruitful.

6.4 Concluding Remarks

Incomplete heuristic procedures for automated theorem proving are often shunned in favor of complete but inefficient tactics. By describing a heuristic procedure that can outperform established methods on natural examples, we hope to convince some readers to do otherwise. Looking at various sources – the Flyspeck project, formalizations presented at conferences on interactive theorem proving, and automated theorem proving competitions, to name just a few – it is clear that on complicated domains such as $\mathbb{T}_{RCF}$, there is no single "correct" automated tool. Heuristic methods may be more modular, take fewer resources to run, cover broader domains of problems, or more easily produce (humanly comprehensible) proofs. For this they deserve to stand alongside decision procedures in the lineup of tools available to mathematicians.

Polya has a number of features that compare favorably to those of other procedures. Its modular structure allows for enormous flexibility and extensibility. Modularity, while often venerated in computer science, is perhaps neglected in mathematics. Proofs that are "pure" to one discipline, or that cleanly separate arguments using technologies from different subfields, have both mathematical and philosophical upsides ([2], [26]). Polya explicitly enforces this kind of separation. Improvements in any one "subfield" – say, developing a more efficient routine for linear arithmetic – can be seamlessly incorporated without affecting any other parts. And similarly, extending its capabilities to capture new types of problems does not weaken its performance on old types.

The virtue of producing comprehensible proofs is particularly intriguing. Some mathematicians and philosophers of math have argued against formalized mathematics, on the grounds that the real purpose of mathematical proof is to increase mathematical *understanding.* Formalizing already-known proofs, they argue, does not produce extra insight, and often even obscures what was in the original proof. Reliance on automated techniques can reinforce these claims: knowing by a complicated CAD computation that a certain statement holds in $\mathbb{T}_{RCF}$ does not explain *why* the statement is true. In fact, this line of thought is partly why the Mizar system supports very little automation.

Of course, one can argue in the other direction as well. A formal proof has virtues other than explanatory power; for one, it instills a high degree of confidence in the correctness of its result. And the formalization process can lead to insights and subtleties that were missed in the traditional proof, such as happened in the Flyspeck project. Perhaps most compellingly, the mathematical work involved in designing these formalization tools is noteworthy in its own right. The desire for proof assistants and automated techniques has spurred significant developments in algorithms, logic, and other areas of math and computer science, and investigating when and why these techniques fail can provide great insight into the questions asked.

Still, though, the naysayers' critique has some weight to it: heavy automation can sometimes obscure the more general reasoning behind a proof, and this can make it difficult to generalize results or understand why certain cases fail. Techniques that produce natural, comprehensible derivations may lessen these worries. By outputting a proof that a human can read and understand, perhaps these techniques increase the explanatory power of their results and of the proofs that rely on them. In the case of $\mathbb{T}_{RCF}$, proofs via CAD and similar methods are complicated, indirect, and have many extraneous steps; in comparison, Polya's proofs can be streamlined and simple. One can see directly the connections between the hypotheses and the conclusion, and try to infer from them generalizations or interesting special cases. A failure of Polya to find a proof can explain as well: maybe the problem depends inherently on the distributivity law, or on a property of a function that was not axiomatized. These musings indicate that there are philosophical virtues to these proofs beyond simply their correctness, and that these epistemic concerns ought to be explored.

Bibliography

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

[2] A. Arana. Methodological purity as mathematical ideal. Preprint, November 2001.

[3] 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.

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

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

[6] 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.

[7] D. Avis. Living with lrs. In *Discrete and computational geometry (Tokyo, 1998)*, volume 1763 of *Lecture Notes in Comput. Sci.*, pages 47–56. Springer, Berlin, 2000.

[8] D. Avis, D. Bremner, and R. Seidel. How good are convex hull algorithms? *Comput. Geom.*, 7(5-6):265–301, 1997. 11th ACM Symposium on Computational Geometry (Vancouver, BC, 1995).

[9] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In H. v. M. Armin Biere, Marijn Heule and T. Walsch, editors, *Handbook of Satisfiability*. IOS Press, 2008.

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

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

[12] 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.

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

[14] E. Boros, K. Elbassioni, V. Gurvich, and K. Makino. Generating vertices of polyhedra and related problems of monotone generation. In *Polyhedral computation*, volume 48 of *CRM Proc. Lecture Notes*, pages 15–43. Amer. Math. Soc., Providence, RI, 2009.

[15] J. M. Borwein and A. S. Lewis. *Convex analysis and nonlinear optimization.* CMS Books in Mathematics/Ouvrages de Mathématiques de la SMC, 3. Springer, New York, second edition, 2006. Theory and examples.

[16] S. Boyd and L. Vandenberghe. *Convex optimization.* Cambridge University Press, Cambridge, 2004.

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

[18] A. Chaieb and T. Nipkow. Proof synthesis and reflection for linear arithmetic. *J. Autom. Reasoning*, 41(1):33–59, 2008.

[19] 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.

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

[21] 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 [17].

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

[23] L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *Automated deduction—CADE-21*, volume 4603 of *Lecture Notes in Comput. Sci.*, pages 183–198. Springer, Berlin, 2007.

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

[25] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[26] M. Detlefsen and A. Arana. Purity of methods. *Philosopher's Imprint*, 11(2):1–20, 2011.

[27] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for dpll(t). In *CAV*, pages 81–94, 2006.

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

[29] K. Fukuda and A. Prodon. Double description method revisited. In *Combinatorics and computer science (Brest, 1995)*, volume 1120 of *Lecture Notes in Comput. Sci.*, pages 91–111. Springer, Berlin, 1996.

[30] S. Gao, J. Avigad, and E. M. Clarke. Delta-complete decision procedures for satisfiability over the reals. In *IJCAR*, pages 286–300, 2012.

[31] D. J. H. Garling. *Inequalities: a journey into linear analysis.* Cambridge University Press, Cambridge, 2007.

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

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

[34] 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.

[35] G. H. Hardy, J. E. Littlewood, and G. Pólya. *Inequalities.* Cambridge Mathematical Library. Cambridge University Press, Cambridge, 1988. Reprint of the 1952 edition.

[36] 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.

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

[38] 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.

[39] 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.

[40] C. N. Jones, E. C. Kerrigan, and J. M. Maciejowski. Equality set projection: A new algorithm for the projection of polytopes in halfspace representation. Technical report, Department of Engineering, University of Cambridge, March 2004. CUED/F-INFENG/TR.463.

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

[42] 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.

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

[44] 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.

[45] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to interval analysis.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2009.

[46] J. Moses. Algebraic simplification: A guide for the perplexed. *Commun. ACM*, 14(8):527–537, 1971.

[47] M. Moskal, J. Łopuszański, and J. R. Kiniry. E-matching for fun and profit. In *Proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007)*, volume 198 of *Electron. Notes Theor. Comput. Sci.*, pages 19–35. Elsevier Sci. B. V., Amsterdam, 2008.

[48] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions of Programming Languages and Systems*, 1:245–257, 1979.

[49] 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.

[50] 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.

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

[52] 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.

[53] V. Prevosto and U. Waldmann. Spass+T. In G. Sutcliffe, R. Schmidt, and S. Schulz, editors, *ESCoR: Empirically Successful Computerized Reasoning 2006*, volume 126, pages 18–33. CEUR Workshop Proceedings, 2006.

[54] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:4–13, 1992.

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

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

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

[58] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. The tptp typed first-order form with arithmetic. In A. V. Nikolaj Bjørner, editor, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume Volume 7180 of *Lecture Notes in Computer Science.* Springer, 2012.

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

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

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

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

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