Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Formally-Verified Decision Procedures for Univariate Polynomial Computation Based on Sturm's and Tarski's Theorems

Anthony Narkawicz, César Muñoz, and Aaron Dutle

Received: date / Accepted: date

Abstract Sturm's theorem is a well-known result in real algebraic geometry that provides a function that computes the number of roots of a univariate polynomial in a semi-open interval, not counting multiplicity. A generalization of Sturm's theorem is known as Tarski's theorem, which provides a linear relationship between functions known as Tarski queries and cardinalities of certain sets. The linear system that results from this relationship is in fact invertible and can be used to explicitly count the number of roots of a univariate polynomial on a set defined by a system of polynomial relations. This paper presents a formalization of these results in the PVS theorem prover, including formal proofs of Sturm's and Tarski's theorems. These theorems are at the basis of two decision procedures, which are implemented as computable functions in PVS. The first, based on Sturm's theorem, determines satisfiability of a single polynomial relation over an interval. The second, based on Tarski's theorem, determines the satisfiability of a system of polynomial relations over the real line. The soundness and completeness properties of these decision procedures are formally verified in PVS. The procedures and their correctness properties enable the implementation of PVS strategies for automatically proving existential and universal statements on polynomial systems. Since the decision procedures are formally verified in PVS, the soundness of the strategies depends solely on the internal logic of PVS rather than on an external oracle.

1 Introduction

Problems involving polynomial inequalities appear in applications such as air traffic conflict detection and resolution algorithms [24], floating point analysis [10], and uncertainty and reliability analysis of dynamic and control sys-

NASA Langley Research Center, Hampton, VA 23681, USA.
E-mail: {Anthony.Narkawicz ,Cesar.A.Munoz,Aaron.M.Dutle}@nasa.gov.

tems [6, 15]. Solving these problems in a rigorous way is fundamental to the logical correctness of these safety-critical systems.

Formal reasoning about polynomials and other nonlinear functions in an interactive theorem prover is challenging. Fortunately, significant advances have been made in this area in recent years. In addition to related work described in later sections, the authors developed formalizations in the Prototype Verification System (PVS) [36] of multivariate Bernstein polynomials [33] and a generic branch and bound algorithm [34], both of which are at the basis of well-known numerical approximation methods. These PVS developments include formally-verified semi-decision procedures for checking validity and satisfiability of nonlinear properties involving variables ranging over real intervals. The procedures are integrated into the PVS theorem prover as automated proof-producing strategies such as `bernstein`, which uses Bernstein polynomials, and `interval`, which uses interval arithmetic. To automatically verify a formula such as

$$x^{120} - 2x^{60} + 1.001 > 0, \tag{1}$$

whenever $x \in [0, 3]$, the user simply needs to invoke one of those strategies in PVS. In particular, the user does not need the insight that the polynomial in Formula (1) is equal to $(x^{60} - 1)^2 + 0.001$. While these strategies are powerful, they inherit the downsides of other numerical approximation methods. For instance, neither of these strategies succeeds in discharging Formula (1) when the variable x is unbounded even though the inequality still holds for any real number x. Moreover, in general, tools based on Bernstein polynomials and interval arithmetic can compute a tight bound for the range of a polynomial but not the exact range. Thus, the strategies `bernstein` and `interval` cannot prove that

$$x^{120} - 2x^{60} + 1 \geq 0, \tag{2}$$

whenever $x \in [0, 3]$, because 0 is the precise minimum of the polynomial on this interval.

This paper addresses these shortcomings of numerical approximation methods for the special case of univariate polynomials. In particular, this paper presents two decision procedures, which are formally verified in PVS, that can be used to check satisfiability and validity of univariate systems of polynomials, involving any of the relations $=, >, <, \neq, \geq, \leq$. The decision procedures presented in this paper are based on Tarski's theorem. In its basic form, Tarski's theorem can be used to explicitly calculate the difference

$$\begin{aligned} &\text{card}(\{x \in (a, b] : p(x) = 0 \wedge g(x) > 0\}) \\ &\quad - \text{card}(\{x \in (a, b] : p(x) = 0 \wedge g(x) < 0\}), \end{aligned}$$

where a and b are (extended) real numbers, p and g are univariate polynomials, and card indicates cardinality. This theorem is a generalization of Sturm's theorem, which allows one to explicitly compute the cardinality

$$\text{card}(\{x \in (a, b] : p(x) = 0\}).$$

The full version of Tarski's theorem is a statement about systems of polynomials. Let $\boldsymbol{g} = (g_0, \ldots, g_k)$ be a sequence of univariate polynomials. Tarski's theorem states that there is a linear relationship between cardinalities of the form

$$\text{card}(\{x \in \mathbb{R} : \; p(x) = 0 \; \wedge \; g_0(x) \, R_0 \, 0 \; \wedge \; \ldots \; \wedge \; \; g_k(x) \, R_k \, 0\}), \tag{3}$$

where $R_i \in \{=, >, <\}$ for $0 \leq i \leq k$, and a collection of so-called Tarski queries of p and polynomials derived from $\boldsymbol{g}$. Crucially, Tarski queries are computable using remainder sequences. This relationship allows for the explicit computation of cardinalities in Formula (3). In this paper, a slight generalization of this relationship is used to accommodate relations R_i in the larger set $\{=, >, <, \neq, \geq, \leq\}$.

This result is then used to define a function, in the PVS specification language, that explicitly computes whether any given system of polynomials is satisfiable. The input system to this function does not require that one of the polynomials play the role of p in Formula (3). That is, systems without an equality among the relations are permitted as well. The only restriction on the polynomial system is that all relations lie in the set $\{=, >, <, \neq, \geq, \leq\}$.

In addition, a second PVS function is defined, based on Sturm's theorem, that computes whether a polynomial relation $p(x) \, R \, 0$, where $R \in \{=, >, <, \neq, \geq, \leq\}$, is satisfiable on any interval. This problem could be decided with the PVS function for systems of relations described above by encoding the interval containment conditions as extra polynomial inequalities, but solving the problem this way is significantly less efficient. This is because the encoding requires computing several Tarski queries due to the additional constraints. Thus, it is beneficial to solve this special type of problem using an algorithm based on Sturm's theorem. This algorithm counts roots in an interval and uses a subdivision schema that continually subdivides the interval until each subinterval contains at most one root. This is a common approach to univariate satisfiability problems of the form $p(x) \, R \, 0$, where $R \in \{=, >, <, \neq, \geq, \leq\}$.

The PVS functions presented in the paper are actually decision procedures for determining satisfiability of formulas involving univariate polynomials. In particular, the correctness of the decision procedure based on Tarski's theorem proves the base case of a quantifier elimination algorithm for the closed field of real numbers. As noted above, the decision procedures are defined as functions in the PVS specification language and their correctness statements are specified as theorems regarding the output of these functions. These theorems are at the basis of proof producing strategies in PVS for automatically proving simply quantified formulas involving polynomial expressions.

The strategy `tarski` handles any univariate polynomial system, the strategy `sturm` handles any polynomial relation over any real interval, and the strategy `mono-poly` handles monotonicity properties of any univariate polynomial on any real interval. These strategies always terminate and their results are provably correct. The soundness of each strategy only relies on the PVS deduction engine. In particular, the strategies do not depend on any trusted external oracle. The core of the strategies is the invocation of the associated

correctness theorems of computable functions fully specified and verified in PVS.

The rest of this paper is organized as follows. Tarski's and Sturm's theorems are presented in Section 2. A key element in the formalization of these theorems is the computation of a sequence of polynomials reminders. This is the subject of Section 3. Decision procedures based on Sturm's and Tarski's theorems are described in Section 4 and Section 5, respectively. PVS strategies `sturm`, `mono-poly`, and `tarski` are defined in Section 6. Formalization issues and related work are discussed in Section 7 and Section 8, respectively. Finally, conclusions are presented in Section 9.

The formal development presented in this paper is electronically available in the contributions `Sturm`[1] and `Tarski`[2] in the NASA PVS Library.[3] All theorems presented in this paper are formally verified in PVS. For readability, standard mathematical notation is used throughout much of the paper. Some theorems in this paper correspond to multiple formal theorems in PVS. Appendix A contains a table that relates theorems in this paper to formal theorems in the PVS development. Furthermore, a PVS specification of the examples presented in this paper is listed in Appendix B. More examples are provided in the theories `Tarski@examples.pvs` and `Sturm@examples.pvs`, which are available as part of the NASA PVS Library.

2 Tarski's and Sturm's Theorems

A *univariate real polynomial* p is a formal expression

$$p \equiv \sum_{i=0}^{n} c_i x^i, \tag{4}$$

where x is an indeterminate, each c_i is a real constant, and $c_n \neq 0$. As usual, the natural number n is called the *degree* of p, and each c_i, with $0 \leq i \leq n$, is called a *coefficient* of p.[4] The coefficient c_n is called the *leading coefficient* of p. Such a polynomial can be identified with a *polynomial function* from $\mathbb{R}$ into $\mathbb{R}$ by evaluation. The polynomial function can be extended to a function from $\mathbb{R}^\infty = \mathbb{R} \cup \{-\infty, \infty\}$ into $\mathbb{R}^\infty$ by setting the evaluation of the polynomial at $\pm\infty$ to $\pm\infty$ depending on the degree of the polynomial and the sign of its leading coefficient. The reader is referred to Section 7.1 for a precise definition of $\mathbb{R}^\infty$ in PVS. It is well known that the identification of a polynomial to its polynomial function is unique for real polynomials and so, in subsequent sections, the distinction between a polynomial and its corresponding polynomial function is only made when it is material.

[1] `http://shemesh.larc.nasa.gov/people/cam/Sturm`.

[2] `http://shemesh.larc.nasa.gov/people/cam/Tarski`.

[3] `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library`.

[4] By convention, the zero polynomial is defined to have degree -1.

In the PVS formalization presented in this paper, a polynomial is represented by a nonempty list of numerical coefficients of type $T \subseteq \mathbb{R}$, i.e., the i-th element of the list represents the coefficient c_i. For instance, the polynomial $p \equiv 1 - 3x^2$ is represented by the list $(1, 0, -3)$. In general, the type T is the PVS native type `real`, since coefficients are real numbers. However, it can be instantiated with any subtype of real numbers to restrict the type of the coefficients of the polynomial. In particular, if each coefficient c_i is an integer for every $i \leq n$ then p will be called an *integer polynomial*. Similarly, if each such c_i is a rational number, then p will be called a *rational polynomial*. If c_i, with $0 \leq i \leq n$, are the coefficients of polynomial p, the *derivative* of p, denoted p', is the constant polynomial 0, if $n = 0$; otherwise, p' is the polynomial of degree $n - 1$ with coefficients $c'_i = (i + 1)\, c_{i+1}$.

Nothing in this paper fundamentally depends on the particular representation of polynomials as lists of coefficients. By abuse of notation, this paper uses italic lowercase letters, e.g., $f, g, \ldots, p, q, \ldots$ to denote polynomials, when they appear in the mathematical discourse, and polynomial representations, i.e., finite lists of real numbers, when they appear in a formal PVS context. Given a list of numerical constants $p \equiv (c_0, \ldots, c_n)$, a higher-order function `polylist` is defined in PVS to specify its corresponding polynomial function. That is, $\texttt{polylist}(p)$ is a function that maps a real number x to the real number $\sum_{i=0}^{n} c_i x^i$. Hence, the notation $p(x)$, when it appears in a formal PVS context, is used to denote the application of the function $\texttt{polylist}(p)$ to x, i.e., $p(x) \equiv \texttt{polylist}(p)(x)$. Furthermore, PVS functions that compute the degree, i-th coefficient, and leading coefficient of a polynomial p are defined as $\texttt{deg}(p) \equiv n$, $\texttt{coeff}(p, i) \equiv c_i$, and $\texttt{lc}(p) \equiv \texttt{coeff}(p, \texttt{deg}(p))$, respectively.

Let g and h be univariate polynomials, such that h is nonzero. Using the division algorithm, one can uniquely write $g = q \cdot h + r$ where q and r are polynomials, and r has degree strictly less than h. Let $\mathrm{rem}(g, h)$ be the polynomial r in this expression. Given univariate polynomials p and g, the *Sturm sequence* of p and g is a sequence S of polynomials

$$p_0,\ p_1,\ p_2,\ \ldots, p_m, \tag{5}$$

where

$$\begin{aligned} p_0 &= p, \\ p_1 &= g \cdot p', \\ \forall\, d > 1 \,:\, p_d &= -\mathrm{rem}(p_{d-2}, p_{d-1}), \\ p_m &= 0, \text{and} \\ p_{m-1} &\neq 0. \end{aligned} \tag{6}$$

Evaluating each of the polynomials in a Sturm sequence at some $x \in \mathbb{R}^\infty$ produces a sequence of extended real numbers $S(x)$. A function $\sigma_{p,g}$ is defined on $\mathbb{R}^\infty$ by setting $\sigma_{p,g}(x)$ to be equal to the number of sign changes in $S(x)$, where the number of sign changes in a sequence is defined as follows. Let $A = (a_0, a_1, \ldots, a_k)$ be a finite sequence of nonzero extended real numbers. The number of sign changes in A is defined as the number of indices i such

that a_i and a_{i+1} have different signs. Given a finite sequence A with extended real entries, define the sequence $\bar{A}$ to be the sequence A with all zero entries removed. The number of sign changes in A is defined as the number of sign changes in $\bar{A}$.

The condition that $p_d = -\text{rem}(p_{d-2}, p_{d-1})$ in Formula (6) can be relaxed to allow, for each remainder, multiplication by any positive real number c such that $p_d = -c \cdot \text{rem}(p_{d-2}, p_{d-1})$. A sequence of polynomials $\hat{S}$ that results from successively applying this transformation to the polynomials of the Sturm sequence S is called a *Sturm chain*. In practice, the value of each c will be chosen so that if p_0 and p_1 have integer coefficients, then each p_d, for $j > 1$, will have primitive integer coefficients; that is, the GCD of all of the coefficients will be 1. This is a simplification of a common approach examined by Knuth, Collins, Brown and Traub, and others [16].

It is easy to verify that for any real number x, the number of sign changes in a Sturm chain $\hat{S}(x)$ is equal to the number of sign changes in the Sturm sequence $S(x)$. Hence, Sturm chains can be used instead of Sturm sequences to compute $\sigma_{p,g}$. Choosing an appropriate Sturm chain can greatly improve the computation of $\sigma_{p,g}$. The precise definition of the Sturm chain used to compute $\sigma_{p,g}$ in the formalization presented in this paper is discussed in Section 3.

The general form of Tarski's theorem states that for $a, b \in \mathbb{R}^\infty$ with $a < b$, if neither a nor b is a root of both p and $p' \cdot g$, then

$$\begin{aligned}\sigma_{p,g}(a) - \sigma_{p,g}(b) = \text{card}(\{x \in (a,b] :\ & p(x) = 0 \wedge g(x) > 0\}) - \\ & \text{card}(\{x \in (a,b] :\ p(x) = 0 \wedge g(x) < 0\}).\end{aligned}$$

The case where g is the constant polynomial 1 is commonly known as Sturm's theorem [41]. This version of the result, which is stated by Theorem 1 below, has been formally proven in PVS (see Appendix A).

Theorem 1 *Let p, g be univariate polynomials. For $a, b \in \mathbb{R}^\infty$, with $a < b$, if neither a nor b is a root of both p and p', and if g is the constant polynomial 1, then $\sigma_{p,g}(a) - \sigma_{p,g}(b)$ is equal to*

$$\text{card}(\{x \in (a,b] :\ p(x) = 0\}).$$

Theorem 1 is the basis of the PVS function `sturm`, presented in Section 4, that specifies a decision procedure for determining the satisfiability of a single rational polynomial relation over an interval with rational or unbounded endpoints.

Fixing $a = -\infty$ and $b = \infty$ and letting p and g be considered parameters in the general version of Tarski's theorem motivates the definition of the *Tarski query*, TQ, which is a function with polynomials p and g as inputs.

$$\text{TQ}(p, g) \equiv \sigma_{p,g}(-\infty) - \sigma_{p,g}(\infty).$$

The following theorem involving Tarski queries has been formally proven in PVS (see Appendix A).

Theorem 2 *Let p, g be univariate polynomials. The Tarski query* $\mathrm{TQ}(p, g)$ *is equal to*

$$\mathrm{card}(\{x \in \mathbb{R} : \ p(x) = 0 \wedge g(x) > 0\}) \ - \ \mathrm{card}(\{x \in \mathbb{R} : \ p(x) = 0 \wedge g(x) < 0\}).$$

In Section 5, Theorem 2 is used to determine the number of roots of a polynomial (not counting multiplicities) that also satisfy a system of polynomial inequalities. This result becomes the basis of the PVS function `tarski` that specifies a decision procedure for determining the satisfiability of a system of rational polynomial relations.

Numerous well-written proofs of Sturm's and Tarski's theorems can be found in the literature [3,13,40]. The authors are content to avoid redundancy by duplicating such a proof. The PVS proofs of Theorem 1 and Theorem 2 most closely follow the approach presented in [40].

3 Remainder Sequences

An important element in the formalization presented in this paper is a function that explicitly computes the remainder after polynomial division. The remainder after division of a polynomial g by a nonzero polynomial h has coefficients that are expressions involving the coefficients of both g and h. These coefficients involve divisions by powers of the leading coefficient of h. In a sequence of remainders of integer polynomials, such as the Sturm sequence of two integer polynomials p and g (Formula (5) in Section 2), the coefficients of the higher numbered polynomials in the sequence can have large denominators. Computation with these large rationals is less efficient than computation with integers. Thus, it is desirable to have a division algorithm for integer polynomials that does not produce rational coefficients in the remainder sequence.

To avoid rational coefficients in the remainder sequence, the development presented in this paper uses an algorithm called *pseudo division* instead of the standard polynomial division algorithm. This division method does not involve division by coefficients of the polynomials, which means that if the coefficients of the original polynomials are integers, then the coefficients of their remainder after pseudo division will also be integers. The motivation for this is that the coefficients of the polynomials in the standard remainder sequence become quite complex as successive remainders are calculated. Computations involving these rationals with many digits in both their numerators and denominators are less efficient than computations on large integers, making the pseudo-remainder more efficient than the standard remainder. In PVS, pseudo division of polynomial g by nonzero polynomial h is defined by the following function, which is recursive on a parameter $i \in \mathbb{N}$, and returns a pair (q, r) of

polynomials.

$$
\begin{array}{l}
\texttt{pseudo\_div}(g,h,i) \equiv \\
\quad \texttt{if } i > \texttt{deg}(g) - \texttt{deg}(h) \texttt{ then } (0,g) \\
\quad \texttt{elsif deg}(h) = 0 \texttt{ then } (g,0) \\
\quad \texttt{else let } (q,r) \;=\; \texttt{if } i = \texttt{deg}(g) - \texttt{deg}(h) \texttt{ then } (0,g) \\
\qquad\qquad\qquad\qquad\quad \texttt{else pseudo\_div}(g,h,i+1) \\
\qquad\qquad\qquad\qquad\quad \texttt{endif in} \\
\qquad\quad (q \;\; \texttt{with}\,[i := \texttt{lc}(r)], f) \\
\quad \texttt{endif},
\end{array}
\tag{7}
$$

where f is the polynomial of degree less or equal than $\texttt{deg}(h) + i$, whose coefficients b_i, with $0 \leq i \leq \texttt{deg}(h) + i$, satisfy

$$
b_j = \begin{cases} \texttt{lc}(h) \cdot \texttt{coeff}(r,j) & \texttt{if } j < i, \\ \texttt{lc}(h) \cdot \texttt{coeff}(r,j) - \texttt{lc}(r)) \cdot \texttt{coeff}(h, j-i) & \texttt{if } i \leq j \leq \texttt{deg}(h) + i. \end{cases}
$$

The polynomial r computed by the pseudo division algorithm is called the *pseudo remainder*. The pseudo remainder is not equal to the remainder after standard division, but is a power of the leading coefficient of h multiplied by the standard remainder. This power can be either positive or negative. Therefore, a correctly signed pseudo remainder can be used in place of the standard remainder in the computation of a Sturm chain. However, the coefficients of the pseudo reminder may still be too large for the computation of $\sigma_{p,g}$. Hence, the pseudo remainder r is multiplied by the reciprocal of its *content* (GCD of all its coefficients), which still ensures that the output is an integer polynomial. This multiplication helps to mitigate coefficient growth and, since the resulting polynomial is still a multiple of the reminder, it can be used in the computation of a Sturm chain.

Let g and h be integer polynomials, such that h is nonzero. The PVS function $\texttt{adjusted\_reminder}(g,h)$, defined below, computes a negative multiple of the reminder of the division of g by h that is used in the formal definition of the function $\sigma_{p,g}$.

$$
\begin{array}{l}
\texttt{adjusted\_reminder}(g,h) \equiv \\
\quad \texttt{let } (q,r) = \texttt{pseudo\_div}(g,h,0), \\
\qquad d = \texttt{gcd\_coeff}(r) \texttt{ in} \\
\quad \texttt{if lc}(h) > 0 \;\vee \text{even}(\texttt{deg}(g) - \texttt{deg}(h) + 1) \\
\qquad \texttt{then } -r/d \\
\qquad \texttt{else } r/d \texttt{ endif}.
\end{array}
\tag{8}
$$

In this paper, the polynomial returned by `adjusted_reminder` is referred to as the *adjusted remainder*. In the PVS development, the function $\texttt{gcd\_coeff}(r)$ computes the greatest common divisor of the coefficients of the nonzero integer polynomial r.

It can be verified that since g and h both have integer coefficients, the polynomial $\texttt{adjusted\_reminder}(g, h)$ does as well. The key mathematical properties of $\texttt{adjusted\_reminder}(g, h)$ are that its degree is less than $\texttt{deg}(h)$ and that there exists a positive real number c, and a polynomial q such that

$$g = q \cdot h - c \cdot \texttt{adjusted\_reminder}(g, h).$$

This implies that $\texttt{adjusted\_reminder}(g, h)$ is a negative multiple of the standard remainder after division of g by h. A corollary of this fact that is instrumental in the proof of Theorem 2 is that if $h(x) = 0$ for some $x \in \mathbb{R}$, then g and $\texttt{adjusted\_reminder}(g, h)$ have opposite signs at x. Thus, if g, h, and $\texttt{adjusted\_reminder}(g, h)$ are consecutive terms in a sequence of remainders, and if $g(x) \neq 0$ and $h(x) = 0$, then there is exactly one sign change in this sequence, when evaluated at x between $g(x)$ and $\texttt{adjusted\_reminder}(g, h)$.

The function `compute_remainder_seq`, defined below, explicitly computes the remainder sequence for any two integer polynomials g and h such that the degree of h is less than the degree of g. It has a list ℓ of integer polynomials as an input, which is also used as an accumulator to store the sequence that is recursively computed by the function.

$$\begin{array}{l}
\texttt{compute_remainder_seq}(g, h, \ell) \equiv \\
\quad \texttt{if length}(\ell) = 0 \texttt{ then compute_remainder_seq}(g, h, \texttt{cons}(g, \ell)) \\
\quad \texttt{elsif deg}(\texttt{head}(\ell)) = 0 \texttt{ then } \ell \\
\quad \texttt{elsif length}(\ell) = 1 \texttt{ then compute_remainder_seq}(g, h, \texttt{cons}(h, \ell)) \\
\quad \texttt{else let } p = \texttt{adjusted_reminder}(\texttt{head}(\texttt{tail}(\ell)), \texttt{head}(\ell)) \texttt{ in} \\
\quad\quad \texttt{compute_remainder_seq}(g, h, \texttt{cons}(p, \ell)) \\
\quad \texttt{endif}.
\end{array} \tag{9}$$

The function `remainder_seq`, defined below, computes the remainder sequence of g and h, which in PVS is represented as list of polynomials.

$$\texttt{remainder\_seq}(g, h) \equiv \texttt{compute\_remainder\_seq}(g, h, ()), \tag{10}$$

where () refers to the empty list. The function `remainder_seq` computes a remainder list in reverse order with respect to the sequence $p_0, \ldots p_m$ given by Formula (5) in Section 2. It can be easily checked that the number of sign changes in ℓ and $\texttt{reverse}(\ell)$ are equal.

Let ℓ be a nonempty list of integer polynomials $(p_0, \ldots, p_m)$, e.g., a Sturm chain computed by the function `remainder_seq`, and x an extended real number, i.e., $x \in \mathbb{R}^{\infty}$. The function `sigma` above formalizes in PVS the function $\sigma_{p,g}$ presented in *Section* 2.

$$\texttt{sigma}(\ell, x) \equiv \texttt{let } a = \ell(x) \texttt{ in sign\_changes}(a, 1, 0, a_0), \tag{11}$$

where $\ell(x)$ denotes the list of real numbers $(p_0(x), \ldots p_m(x))$, with $0 \leq i \leq m$. The PVS function `sign_changes` is recursively defined as follows, where i is an

index between 1 to $m+1$, n counts the number of sign changes below index i, and ak is the value of the last value below index i that is nonzero.

$$\begin{array}{l}
\texttt{sign_changes}(a, i, n, ak) \equiv \\
\quad \texttt{if } i > \texttt{length}(a) \texttt{ then } n \\
\quad \texttt{elsif } a_i \neq 0 \ \wedge \ ak \neq 0 \ \wedge \ \text{sign}(a_i) \neq \text{sign}(ak) \texttt{ then} \\
\quad\quad \texttt{sign_changes}(a, i+1, n+1, a_i) \\
\quad \texttt{elsif } a_i \neq 0 \texttt{ then } \texttt{sign_changes}(a, i+1, n, a_i) \\
\quad \texttt{else } \texttt{sign_changes}(a, i+1, n, ak) \\
\quad \texttt{endif}.
\end{array} \tag{12}$$

4 A Decision Procedure Based on Sturm's Theorem

Sturm's theorem, proved in PVS as Theorem 1, provides a method for counting the number of roots of a polynomial p (not counting multiplicities) in a half-open interval $(a, b]$ under the condition that neither endpoint is a root of p of multiplicity greater than 1. This section details the use of this theorem to decide whether a rational polynomial inequality is always true on an arbitrary interval, provided the endpoints are either rational or infinite. The reader is referred to [35] for more technical information on how Sturm's theorem is formalized in PVS.

4.1 Decision Procedure for Integer Polynomials

This section presents a decision procedure for computing the *sign* of an integer polynomial p, i.e., its positivity, nonpositivity, negativity, nonnegativity, or nonzero property, on a nonempty interval, which may or may not have infinite endpoints. This decision procedure depends on a function that explicitly computes the number of roots of p in that interval. One immediately obvious problem with using the function σ from Sturm's theorem to define such function is that it will not work when either the lower bound or the upper bound of the interval is a root of multiplicity greater than one, i.e., when the polynomial and its derivative both have roots at that point.

The problem of having roots of multiplicity greater than 1 at the endpoints of an interval can be addressed by perturbing such bounds outward by a small amount so that the new interval contains exactly the same number of roots as the original but has endpoints that are not roots with multiplicity greater than 1. To see how such a perturbation can be computed, let r be a root of p, so $p(r) = 0$. A function can be defined with p and r as inputs that computes the width of a small interval around r that contains no other roots of p. The first step is to compute the degree of the first successive derivative of p that does not vanish at r, which is accomplished with the following recursive function.

$$\begin{array}{l}
\texttt{md}(p, r) \equiv \texttt{if } \texttt{deg}(p) = 0 \ \vee \ p(r) \neq 0 \texttt{ then } n \\
\qquad\qquad\qquad \texttt{else } \texttt{md}(p', r) \texttt{ endif}.
\end{array} \tag{13}$$

The function `md` is well defined since the degree of the polynomial that is passed as parameter strictly decreases at each recursive call.

By using Taylor's theorem, p is approximated in a small neighborhood of r by $p^{(n-\mathtt{md}(p,r))}(r) \cdot (x-r)^{n-\mathtt{md}(p,r)}$. The error is bounded by a constant times $(x-r)^{n-\mathtt{md}(p,r)+1}$, where $p^{(n-\mathtt{md}(p,r))}(r)$ is the $(n-\mathtt{md}(p,r))$-th derivative of p at r. The interval around r containing no other roots is determined by computing a neighborhood of r on which this derivative is always nonzero. Since it is always nonzero, it can be proved by induction and the mean value theorem that all lesser derivatives vanish only at r on this neighborhood. This neighborhood is computed as follows. First, the following function is defined that takes as inputs a polynomial p, a real number x, and a positive real number ϵ. It returns a positive real number δ with the property that for all $y \in \mathbb{R}$, if $|x-y| < \delta$ then $|p(x) - p(y)| < \epsilon$.

$$
\begin{aligned}
&\mathtt{pcc}(p, x, \epsilon) \equiv \\
&\quad \mathtt{let}\ n = \mathtt{deg}(p), \\
&\qquad c = \max_{i=0}^{n} |\mathtt{coeff}(p, i)| + 1\ \mathtt{in} \\
&\quad \mathtt{if}\ n = 0\ \mathtt{then}\ 1/2 \\
&\quad \mathtt{else}\ \min\left(\frac{\epsilon}{c}\left(1 + \sum_{i=1}^{n}\sum_{j=1}^{i}\binom{i}{j-1}|x|^{j-1}\right), \frac{1}{2}\right) \\
&\quad \mathtt{endif}.
\end{aligned}
\tag{14}
$$

Using the function `pcc`, it is now possible to define a radius around the point r in which the polynomial p has no other roots. This radius is computed with the function `root_rad`.

$$
\begin{aligned}
&\mathtt{root\_rad}(p, r) \equiv \mathtt{let}\ n = \mathtt{deg}(p)\ \mathtt{in} \\
&\qquad \mathtt{pcc}(p^{(n-\mathtt{md}(p,r))}, r, |p^{(n-\mathtt{md}(p,r))}(r)|).
\end{aligned}
\tag{15}
$$

Note that the function `root_rad` can be used to compute the number of roots in any interval, not simply an interval whose endpoints are not roots of multiplicity greater than 1. To see this by example, note that if $(a, b]$ is an interval of real numbers such that b is a root of p of multiplicity greater than 1, but a is not, then the interval $(a, b + \mathtt{root\_rad}(p, b)/2]$ contains the same number of roots as $(a, b]$ but neither endpoint is a root of multiplicity greater than 1.

The next step in the development is to define a function called `roots_cl_int` with two extended real numbers a and b as inputs, with $a < b$, along with an integer polynomial p. The function returns the number of roots in the closed interval $[a, b]$, not counting multiplicities. This means, for instance, that for the polynomial $p \equiv x^3$ on the interval $[-1, 1]$, it will count exactly one root. The other input to the function `roots_cl_int` is a list ℓ of polynomials. In practice ℓ is set to $\mathtt{sturm\_chain}(p) \equiv \mathtt{remainder\_seq}(p, p')$, where `remainder_seq` computes a Sturm chain as defined by Formula (10) in Section 3.

$$
\begin{array}{l}
\texttt{roots\_cl\_int}(p, a, b, \ell) \equiv \\
\quad \texttt{let } a^* = \texttt{if } a = -\infty \ \vee \ p(a) \neq 0 \ \vee \ p'(a) \neq 0 \texttt{ then } a \\
\qquad\qquad \texttt{else } a - \texttt{root\_rad}(p, a)/2 \texttt{ endif} \\
\qquad b^* = \texttt{if } b = \infty \ \vee \ p(b) \neq 0 \ \vee \ p'(b) \neq 0 \texttt{ then } b \\
\qquad\qquad \texttt{else } b + \texttt{root\_rad}(p, b)/2 \texttt{ endif} \\
\qquad c \ = \texttt{if } a \neq -\infty \ \wedge \ p(a) = 0 \ \wedge \ p'(a) \neq 0 \texttt{ then } 1 \\
\qquad\qquad \texttt{else } 0 \texttt{ endif} \\
\quad \texttt{in } \texttt{sigma}(\ell, a^*) - \texttt{sigma}(\ell, b^*) + c.
\end{array}
\tag{16}
$$

The definition of `roots_cl_int` uses the function `sigma`, defined by Formula (11) in Section 3. This function takes as inputs a finite list ℓ of polynomials and an extended real number r, and returns the number of sign changes in that list when each element is evaluated at r. The introduction of the number c in the definition of `roots_cl_int` addresses the limitation in Sturm's theorem, which only gives a way to count the number of roots in a half open interval that does not include its lower bound. The term c adjusts this number based on whether the lower bound is equal to the newly computed a^* and is also a root of p but not of its derivative.

The following theorem states the correctness of the function `roots_cl_int`. It has been formally proved in PVS (see Appendix A).

Theorem 3 *Let $a, b \in \mathbb{R}^\infty$, with $a < b$, p be an integer polynomial, and $S = \{r \in \mathbb{R} \mid a \leq r \leq b \text{ and } p(r) = 0\}$. It holds that*

$$\text{card}(S) = \textit{roots\_cl\_int}(p, a, b, \textit{sturm\_chain}(p)).$$

The next step in the PVS development is to use the function `roots_cl_int` to define a function `roots_interval` that computes the number of roots of an integer polynomial in any interval, whether it is closed, open, half open and half closed, unbounded, and whether or not the polynomial has a root of multiplicity greater than 1 at an endpoint of the interval. The precise definition of this function is omitted from this description, because it is straightforward to define it directly in terms of the function `roots_cl_int`. The reader is referred to the PVS development Sturm in the NASA PVS Library for the precise definition of this function. Basically, the number of roots in the closure of the given interval is computed using `roots_cl_int`, and this number is then adjusted upward or downward depending on whether the lower bound or the upper bound of the interval is a root of the polynomial, and whether the interval itself actually contains this bound. The following theorem has been proved in PVS (see Appendix A).

Theorem 4 *Let p be an integer polynomial, I an interval, whose bounds are extended reals, and $S = \{r \in \mathbb{R} \mid r \in I \text{ and } p(r) = 0\}$. It holds that*

$$\text{card}(S) = \textit{roots\_interval}(p, I, \textit{sturm\_chain}(p)).$$

It can now be noted that if a polynomial is always positive on a given interval, it is trivial to prove that this is true using the function `roots_interval`. All that must be checked is that the function `roots_interval` returns 0, so that the polynomial has no roots on the interval, and that it is positive at any fixed point in the interval, such as $(a+b)/2$ in the case where $a, b \in \mathbb{R}$. Thus, the difficulty when determining whether the polynomial p satisfies $p(r)\ R\ 0$ for all r in a given interval, when R is a relation in $\{>, <, \neq, \geq, \leq\}$ lies in the case when R is nonstrict, i.e., when $R \in \{\geq, \leq\}$. Moreover, if a decision procedure can be defined for when R is $\geq$, then it can be defined for when R is $\leq$ as well by just replacing p with $-p$. Thus, the next step is to define a specific decision procedure that determines whether an integer polynomial is always nonnegative on a bounded closed interval. The PVS function `nonneg_int` takes as inputs an integer polynomial p, real numbers x and y (not extended real numbers), and a list ℓ of polynomials, which in practice is set to `sturm_chain`(p). It returns a Boolean, which is equal to `true` if and only if $p(r) \geq 0$ for r in the closed, bounded interval $[x, y]$.

The function `nonneg_int` works by recursively subdividing the interval $[x, y]$ into left and right halves, until each subinterval is small enough that it contains at most one root of the polynomial p. Then, the polynomial p is evaluated at each endpoint of each subinterval. One important fact that is used in the verification of this function is that a continuous function with at most one root in a closed, bounded interval is always nonnegative on that interval if and only if it is nonnegative at its endpoints. This result follows directly from the intermediate value theorem. The recursive function `nonneg_int` is defined below.

$$
\begin{aligned}
&\texttt{nonneg\_int}(p, x, y, \ell) \equiv \\
&\quad \texttt{if } x > y \texttt{ then true} \\
&\quad \texttt{elsif } x = y \texttt{ then } p(x) \geq 0 \\
&\quad \texttt{elsif roots\_cl\_int}(p, x, y, \ell) \leq 1 \texttt{ then } p(x) \geq 0 \ \wedge\ p(y) \geq 0 \qquad (17) \\
&\quad \texttt{else nonneg\_int}(p, x, (x+y)/2, \ell) \wedge \\
&\qquad\quad \texttt{nonneg\_int}(p, (x+y)/2, y, \ell) \\
&\quad \texttt{endif}.
\end{aligned}
$$

The function `nonneg_int` is a decision procedure for nonnegativity on closed, bounded intervals. However, it should be noted that a polynomial is always nonnegative on any given interval if and only if it is nonnegative on the closure of that interval. For instance, p is always nonnegative on the open interval $(0, 1)$ if and only if it is nonnegative on the closed interval $[0, 1]$. Thus, the function `nonneg_int` can be used as a decision procedure on any bounded interval, even if it is open.

To extend this function to unbounded intervals, a number is computed that is guaranteed to bound all of the roots of a given polynomial. Such bounds are commonly referred to as *Cauchy bounds*, and can help reduce any unbounded interval to a bounded interval on which the polynomial is nonnegative if and

only if it is nonnegative on the original unbounded interval. While there are many possible definitions of such bounds, the formalization presented in this paper uses Knuth's bound as in [31]. The reader is referred to the development `reals` in the NASA PVS library, where the precise definition of the function `Knuth_poly_root_strict_bound`, which specifies Knuth's bound. The following theorem is proved in PVS (see Appendix A).

Theorem 5 *For any nonzero real polynomial p, any root of p lies in the open interval $(-k, k)$, where $k =$ `Knuth_poly_root_strict_bound`(p).*

The function `nonneg_int` can now be used to define a function `always_nonneg` that takes as inputs an integer polynomial p and two extended (so possibly unbounded) real numbers a and b with $a < b$. It returns `true` precisely when p is always nonnegative on (a, b), which, as noted above, is equivalent to p being nonnegative on the closure of this interval.

$$
\begin{array}{l}
\texttt{always\_nonneg}(p, a, b) \equiv \\
\quad \texttt{let } \ell = \texttt{sturm\_chain}(p), \\
\qquad M = \texttt{Knuth\_poly\_root\_strict\_bound}(p) \texttt{ in} \\
\quad \texttt{if } a \neq -\infty \ \wedge \ b \neq \infty \texttt{ then nonneg\_int}(p, a, b, \ell) \\
\quad \texttt{elsif } b \neq -\infty \texttt{ then nonneg\_int}(p, \min(-M, b-1), b, \ell) \\
\quad \texttt{elsif } a \neq \infty \texttt{ then nonneg\_int}(p, a, \max(M, a+1), \ell) \\
\quad \texttt{else nonneg\_int}(p, -M, M, \ell) \\
\quad \texttt{endif}.
\end{array}
\tag{18}
$$

The following theorem has been proved in PVS (see Appendix A).

Theorem 6 *If $a, b \in \mathbb{R}^{\infty}$, with $a < b$, and p is an integer polynomial, then `always_nonneg`$(p, a, b) =$ `true` if and only if every real number x, with $a \leq x \leq b$, satisfies $p(x) \geq 0$.*

Note again that the continuity of p implies that this theorem is true if either (or both) of the non-strict inequalities in the condition $a \leq x \leq b$ is replaced by strict inequality.

The decision procedure that determines whether $p(x)\ R\ 0$ for all x in a given interval I, with extended real bounds a, b, and where R is a relation in $\{>, <, \neq, \geq, \leq\}$, can now be defined as follows. If R is either $\geq$ or $\leq$, then the function `always_nonneg` is used for either p or $-p$ (respectively). Otherwise, it is simply checked that the function `roots_interval` returns 0 on the interval and that a given point r in the interval satisfies $p(r)\ R\ 0$. The correctness of the procedure then follows from the intermediate value theorem. The point r can be chosen in a number of ways. For this development, it is simply assumed that there is a function `choose`(I) that returns a real number in the interval I that is neither a nor b. For a precise definition of this function in PVS, the reader is referred to Section 7.1. The input p to the function `compute_poly_sat`, defined below, must be an integer polynomial.

$$\begin{array}{l}\texttt{compute\_poly\_sat}(p, I, R) \equiv \\ \quad \texttt{if } R = (\geq) \texttt{ then always\_nonneg}(p, a, b) \\ \quad \texttt{elsif } R = (\leq) \texttt{ then always\_nonneg}(-p, a, b) \\ \quad \texttt{else roots\_interval}(p, I, \texttt{sturm\_chain}(p)) \\ \qquad \wedge\ p(\texttt{choose}(p, I))\ R\ 0 \\ \quad \texttt{endif},\end{array} \tag{19}$$

where a and b are the extended real numbers denoting the upper and lower bounds of I, respectively. The following theorem has been formally proved in PVS (see Appendix A).

Theorem 7 *Let p be a nonzero integer polynomial, I be an interval whose bounds are extended real numbers, and R be a relation in $\{>, <, \neq, \geq, \leq\}$. It holds that*

$$\textit{compute\_poly\_sat}(p, I, R) = \textit{true}$$

if and only if $p(x)\ R\ 0$ for all real numbers $x \in I$.

Example 1 Consider the integer polynomial $p \equiv x^{120} - 2x^{60} + 1$, as in Formula (2) in Section 1, and the open interval $I \equiv (0, 3)$. It can be checked that

$$\begin{array}{l}\texttt{compute\_poly\_sat}(p, I, \geq) = \texttt{true}, \\ \texttt{compute\_poly\_sat}(p, I, >) = \texttt{false}.\end{array} \tag{20}$$

By Theorem 7, it holds that $\forall(x : I) : p(x) \geq 0$. Furthermore, it does not hold that $\forall(x : I) : p(x) > 0$. Therefore, $\exists(x : I) : p(x) = 0$. Indeed, p factors as $(x^{60} - 1)^2$.

4.2 Decision Procedure for Rational Polynomials

If p is rational polynomial, the relation $p(x)\ R\ 0$ can be decided on any given interval by multiplying the coefficients of p by the product of the denominators of its coefficients and then using the function `compute_poly_sat` on the resulting integer polynomial.

While this appears to be a simple process, the particular mechanics of PVS make this difficult. In PVS, rational numbers are represented by a primitive type. For example, numerical constants $\frac{1}{2}$, $\frac{2}{4}$, and 0.5 are indistinguishable. Hence, the definition of a PVS function that computes the numerator and denominator of a rational number is not straightforward. The solution to this problem, while interesting in the context of an interactive theorem prover as PVS, is not directly necessary for the explanations in this section. Interested readers are referred to Section 7.2 for further explanation. In this section, it is assumed that there is a function, namely `compute_pos_rat`, that takes a positive rational number r as input and returns a pair of natural numbers (a, b), relative primes, such that $r = a/b$.

The function `rat2poly` can then be defined in PVS which takes a rational polynomial and coverts it to an integer polynomial. This process works by recursively considering each coefficient of the polynomial. At each step, it multiplies the polynomial by the denominator of the coefficient in question, and it also stores the current greatest common divisor of all resulting integer coefficients that it has simplified so far in the recursion. At the end of the recursion, all of the coefficients are divided by this greatest common divisor to simplify the answer. It can be easily verified that $\texttt{rat2poly}(p)$ is an integer polynomial that is a positive multiple of p. The function `sturm` defined below specifies a decision procedure for rational polynomials.

$$\texttt{sturm}(p, I, R) \equiv \texttt{compute\_poly\_sat}(\texttt{rat2poly}(p), I, R). \tag{21}$$

The following theorem has been formally proved in PVS (see Appendix A).

Theorem 8 *Let p be a nonzero rational polynomial, I be an interval, whose bounds are extended real numbers, and R be a relation in $\{>, <, \neq, \geq, \leq\}$. It holds that*

$$\mathit{sturm}(p, I, R) = \mathit{true}$$

if and only if $p(x)\ R\ 0$ for all real numbers $x \in I$.

Example 2 Consider the rational polynomial $p \equiv x^{120} - \frac{2}{3}x^{60} + \frac{1}{9}$ and the open unbounded interval $I \equiv (-\infty, 3)$. It can be checked that

$$\begin{aligned} &\texttt{sturm}(p, -\infty, 3, \texttt{false}, \texttt{false}, \geq) = \texttt{true}, \\ &\texttt{sturm}(p, -\infty, 3, \texttt{false}, \texttt{false}, >) = \texttt{false}. \end{aligned} \tag{22}$$

By Theorem 8, it holds that $\forall(x : I) : p(x) \geq 0$. Furthermore, it does not hold that $\forall(x : I) : p(x) > 0$. Therefore, $\exists(x : I) : p(x) = 0$. Indeed, p factors as $(x^{60} - \frac{1}{3})^2$.

The formalization presented in this paper also includes a PVS function `mono` that decides if a polynomial function represented by p is increasing or decreasing in a given interval I. This function simply calls `sturm` to check the sign of the derivative of p on I. More precisely, `mono` has as parameters a representation of a rational polynomial, an interval representing the variable range, and two real order relations. It returns a Boolean value satisfying the following theorem, which is formally verified in PVS (see Appendix A).

Theorem 9 *Let p be a nonzero rational polynomial, I be an interval, whose bounds are extended real numbers, and R_1, R_2 be relations in $\{<, \leq, >, \geq, \neq\}$, $\mathit{mono}(p, I, R_1, R_2) = \mathit{true}$ if and only if $\forall(x, y\colon \mathbb{R}) : x, y \in I \wedge x\ R_1\ y \implies p(x)\ R_2\ p(y)$.*

The proof of Theorem 9 follows directly from Theorem 8.

5 A Decision Procedure Based on Tarski Queries

Theorem 2 in Section 2 asserts a relationship between a Tarski query $\mathrm{TQ}(p, q)$, which can be explicitly calculated using the methods presented in Section 3, and the cardinalities of two sets defined by the polynomials p and g. In Section 5.1, it is shown how this can be used to determine $\mathrm{card}(\{x \in \mathbb{R} : \ p(x) = 0 \ \wedge \ g_0(x)\, R_0\, 0 \ \wedge \ \ldots \ \wedge \ g_k(x)\, R_k\, 0\})$, where each g_i is a polynomial and each $R_i \in \{=, >, <, \neq, \geq, \leq\}$. In Section 5.2, this result is used to define an algorithm in PVS for determining the satisfiability of an arbitrary system of rational polynomial relations.

5.1 Generalizing Tarski Queries to Polynomial Systems

Theorem 2 states that the Tarski query $\mathrm{TQ}(p, g)$ is equal to

$$\mathrm{card}(\{x \in \mathbb{R} : \ p(x) = 0 \wedge g(x) > 0\}) \ - \ \mathrm{card}(\{x \in \mathbb{R} : \ p(x) = 0 \wedge g(x) < 0\}).$$

Instantiating the polynomial g with 1, g, and g^2, it can be seen that

$$\begin{bmatrix} \mathrm{TQ}(p,1) \\ \mathrm{TQ}(p,g) \\ \mathrm{TQ}(p,g^2) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \\ 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathrm{card}(\{x \ : \ p(x) = 0 \ \wedge \ g(x) = 0\}) \\ \mathrm{card}(\{x \ : \ p(x) = 0 \ \wedge \ g(x) > 0\}) \\ \mathrm{card}(\{x \ : \ p(x) = 0 \ \wedge \ g(x) < 0\}) \end{bmatrix}. \tag{23}$$

By inverting the 3×3 matrix above and calculating the three Tarski queries, the vector on the far right hand side can easily be computed.

Most treatments of the generalization of Tarski queries to systems of polynomials consider Formula (23) as the starting point, noting that for $R \in \{\neq, \geq, \leq\}$ the value of $\mathrm{card}(\{x \ : \ p(x) = 0 \ \wedge \ g(x)\, R\, 0\})$ can be calculated as the sum of two of the known cardinalities [3]. The present development includes the additional relations from the outset. Equation (23) and Theorem 2 imply the following equality of vectors.

$$\begin{bmatrix} \mathrm{TQ}(p,1) \\ \mathrm{TQ}(p,g) \\ \mathrm{TQ}(p,g^2) \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 1 & 0 & 0 \\ -1 & -1 & 0 & 0 & 1 & 0 \\ -1 & 0 & -1 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathrm{card}(S_=) \\ \mathrm{card}(S_>) \\ \mathrm{card}(S_<) \\ \mathrm{card}(S_\neq) \\ \mathrm{card}(S_\geq) \\ \mathrm{card}(S_\leq) \end{bmatrix}, \tag{24}$$

where $S_R = \{x \ : \ p(x) = 0 \ \wedge \ g(x) \ R \ 0\}$. The system (24) is used in lieu of the simpler system in Equation (23) because it simplifies the decision procedure. As noted above, if the 3×3 matrix were used in the following constructions and in the decision procedure presented later, a final step for calculating cardinalities of solution sets of systems involving multiple polynomials relations with $\leq, \geq$, and $\neq$ would require adding cardinalities of the sets with the simpler relations $=, >$, and $<$. This final step is exponential in the number of the relations with $\leq, \geq$, and $\neq$, whereas it is not even necessary if the 6×6 matrix $\mathbf{M}_6$ is used instead of the 3×3 matrix.

Throughout this paper, and in the PVS development, entries of matrices are expressed with indices starting at 0. That is, the top left entry of a matrix is its $(0,0)$-th entry, and the first entry of a vector is its 0-th entry. This greatly simplifies the expressions to follow, which refer to the entries of matrices and vectors using base-3 and base-6 representations of indices. The expression $A[i,j]$ denotes the (i,j) entry of a matrix A.

Let $\mathbf{M}_6$ denote the 6×6 matrix in Equation (24). Suppose that $\boldsymbol{g} = \{g_0, \ldots, g_k\}$ is a sequence of polynomials and define $\mathbf{TQ}_6(p, \boldsymbol{g})$ to be the vector with 6^{k+1} entries whose i-th entry is given as follows. Let $(i_0, \ldots, i_k)$ be the base-6 representation of i. If $i_d < 3$ for all $d \leq k$, then the i-th entry is given by

$$\mathrm{TQ}(p, \prod_{d=0}^{k} g_d^{i_d}).$$

Otherwise, the i-th entry is 0. Similarly, let $\mathbf{N}_6(p, \boldsymbol{g})$ be the vector with 6^{k+1} entries whose j-th entry is given by the cardinality of the set $\texttt{SolSet}(p, \boldsymbol{g}, j)$, defined by

$$\texttt{SolSet}(p, \boldsymbol{g}, j) = \{x \in \mathbb{R} : \ p(x) = 0 \ \wedge \ g_0(x) \ R_0 \ 0 \ \wedge \ \ldots \ \wedge \ \ g_k(x) \ R_k \ 0\},$$

where each relation R_d, with $0 \leq d \leq k$, is given by

$$R_d \equiv \begin{cases} = & \texttt{if } j_d = 0, \\ > & \texttt{if } j_d = 1, \\ < & \texttt{if } j_d = 2, \\ \neq & \texttt{if } j_d = 3, \\ \geq & \texttt{if } j_d = 4, \\ \leq & \texttt{if } j_d = 5, \end{cases}$$

and where $(j_0, \ldots, j_k)$ is the base-6 representation of j. The following theorem asserts a linear relation between $\mathbf{TQ}_6(p, \boldsymbol{g})$ and $\mathbf{N}_6(p, \boldsymbol{g})$. It has been formally verified in PVS (see Appendix A).

Theorem 10 *For a nonzero polynomial p and list of nonzero polynomials $\boldsymbol{g} = (g_0, \ldots, g_k)$, all with real coefficients,*

$$\mathbf{TQ}_6(p, \boldsymbol{g}) = \mathbf{M}_6^{\otimes(k+1)} \cdot \mathbf{N}_6(p, \boldsymbol{g}). \tag{25}$$

The matrix $\mathbf{M}_6^{\otimes(k+1)}$ in Formula (25) denotes the standard tensor power of $\mathbf{M}_6$. The formal proof of Theorem 10 mostly relies on basic mathematics and Theorem 2. An outline of the formal proof in PVS is given below.

Proof (Sketch) Choose $i < 6^{k+1}$, and let $(i_0, \ldots, i_k)$ be the base-6 representation of i. The i-th entry of $\mathbf{M}_6^{\otimes(k+1)} \cdot \mathbf{N}_6(p, \boldsymbol{g})$ is given by

$$\sum_{j=0}^{6^{k+1}-1} \left(\prod_{d=0}^{k} \mathbf{M}_6[i_d, j_d] \right) \cdot \mathrm{card}(\texttt{SolSet}(p, \boldsymbol{g}, j)). \tag{26}$$

Suppose first that $i_d < 3$ for all $d \leq k$. Then, Theorem 2 implies that the i-th entry of $\mathbf{TQ}_6(p, \boldsymbol{g})$ is given by

$$\begin{aligned}\mathrm{TQ}(p, \prod_{d=0}^{k} g_d^{i_d}) = \ & \mathrm{card}(\{x \in (a,b] \ : \ p(x) = 0 \ \wedge \ \prod_{d=0}^{k} g_d(x)^{i_d} > 0\}) \\ - \ & \mathrm{card}(\{x \in (a,b] \ : \ p(x) = 0 \ \wedge \ \prod_{d=0}^{k} g_d(x)^{i_d} < 0\}).\end{aligned}$$

Each coefficient $\prod_{d=0}^{k} \mathbf{M}_6[i_d, j_d]$ in the sum in Formula (26) is either 0, 1 or -1. It is relatively straightforward to show that the sets $\mathtt{SolSet}(p, \boldsymbol{g}, j)$ in Formula (26) for which this entry is -1 are disjoint and that their union is equal to

$$\{x \in (a,b] \ : \ p(x) = 0 \ \wedge \ \prod_{d=0}^{k} g_d(x)^{i_d} < 0\}.$$

Similarly, the sets for which this entry is 1 are disjoint and their union is equal to

$$\{x \in (a,b] \ : \ p(x) = 0 \ \wedge \ \prod_{d=0}^{k} g_d(x)^{i_d} > 0\},$$

which proves the result in this case.

Now suppose that $i_q \geq 3$ for some $q \leq k$. For example, assume that $q = 0$ and that $i_0 = 4$. The general case is similar, although it requires reordering the sum in Equation (26). Then

$$\begin{aligned}&\sum_{j=0}^{6^{k+1}-1} \left(\prod_{d=0}^{k} \mathbf{M}_6[i_d, j_d] \right) \cdot \mathrm{card}(\mathtt{SolSet}(p, \boldsymbol{g}, j)) \\ &\qquad = \sum_{t=0}^{6^k-1} \sum_{j=6t}^{6\cdot(t+1)-1} \left(\prod_{d=0}^{k} \mathbf{M}_6[i_d, j_d] \right) \cdot \mathrm{card}(\mathtt{SolSet}(p, \boldsymbol{g}, j)).\end{aligned}$$

However, for any $t < 6^k$,

$$\begin{aligned}
&\sum_{j=6t}^{6\cdot(t+1)-1} \left(\prod_{d=0}^{k} \mathbf{M}_6[i_d, j_d] \right) \cdot \mathrm{card}(\texttt{SolSet}(p, \boldsymbol{g}, j)) \\
&= (-1)\cdot \left(\prod_{d=1}^{k} \mathbf{M}_6[i_d, (6t)_d] \right) \cdot \mathrm{card}(\texttt{SolSet}(p, \boldsymbol{g}, 6t)) \\
&\quad +(-1)\cdot \left(\prod_{d=1}^{k} \mathbf{M}_6[i_d, (6t+1)_d] \right) \cdot \mathrm{card}(\texttt{SolSet}(p, \boldsymbol{g}, 6t+1)) \\
&\quad +0\cdot \left(\prod_{d=1}^{k} \mathbf{M}_6[i_d, (6t+2)_d] \right) \cdot \mathrm{card}(\texttt{SolSet}(p, \boldsymbol{g}, 6t+2)) \\
&\quad +0\cdot \left(\prod_{d=1}^{k} \mathbf{M}_6[i_d, (6t+3)_d] \right) \cdot \mathrm{card}(\texttt{SolSet}(p, \boldsymbol{g}, 6t+3)) \\
&\quad +1\cdot \left(\prod_{d=1}^{k} \mathbf{M}_6[i_d, (6t+4)_d] \right) \cdot \mathrm{card}(\texttt{SolSet}(p, \boldsymbol{g}, 6t+4)) \\
&\quad +0\cdot \left(\prod_{d=1}^{k} \mathbf{M}_6[i_d, (6t+5)_d] \right) \cdot \mathrm{card}(\texttt{SolSet}(p, \boldsymbol{g}, 6t+5)).
\end{aligned}$$

For any d such that $1 \leq d \leq k$, the numbers $(6t)_d$, $(6t+1)_d$, $(6t+2)_d$, $(6t+3)_d$, $(6t+4)_d$ and $(6t+5)_d$ are all equal, so the five products appearing in this sum are all equal. Also, the set $\texttt{SolSet}(p, \boldsymbol{g}, 6t+4)$ is equal to the (disjoint) union of $\texttt{SolSet}(p, \boldsymbol{g}, 6t)$ and $\texttt{SolSet}(p, \boldsymbol{g}, 6t+1)$. This completes the proof. □

The proof of Theorem 10 is mostly straightforward. However, its formal proof is long, due to the fact that the informal proof tends to gloss over details that must be made precise in the formal proof. In particular, neither base-6 nor base-3 representations are even mentioned in most references to this theorem in textbooks [3], while Theorem 10 involves vectors whose entries are actually defined using the base-6 or base-3 representations of the indices. It appears that the reason for this difference is that the theorem itself is usually viewed inductively, meaning that the point in most literature is to help the reader understand why it is true for k polynomials rather than how to compute with it directly. The formal PVS proof often proves equivalences of multidimensional constructions, such as vectors, by proving that individual entries are equal. Thus, much of the PVS development must refer to the entries of certain vectors, and these entries are computed by first converting the index to either its base-6 or base-3 representation. The PVS proof requires multiple translations between natural numbers and their and base-6 representations, which can be cumbersome at times and do not appear in the informal proofs. For this reason, the PVS proof, while not intellectually difficult, is more complex than the mathematical proof.

To determine if a system of polynomial inequalities has a solution, it is necessary to compute entries of the vector $\mathbf{N}_6(p, \boldsymbol{g})$ from Theorem 10. Fortunately, the matrix $\mathbf{M}_6^{\otimes(k+1)}$ is invertible and its inverse is given by

$$(\mathbf{M}_6^{\otimes(k+1)})^{-1} = (\mathbf{M}_6^{-1})^{\otimes(k+1)} = \begin{bmatrix} 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & \frac{1}{2} & -\frac{1}{2} & 0 & 1 & 0 \\ 1 & -\frac{1}{2} & -\frac{1}{2} & 0 & 0 & 1 \end{bmatrix}^{\otimes(k+1)}.$$

The formalization and implementation of the aspects of linear algebra used in the present development are discussed in Section 7.3. The following theorem has been verified in PVS (see Appendix A).

Theorem 11 *For a nonzero polynomial p and list of nonzero polynomials $\boldsymbol{g} = (g_0, \ldots, g_k)$, all with real coefficients,*

$$\mathbf{N}_6(p, \boldsymbol{g}) = (\mathbf{M}_6^{-1})^{\otimes(k+1)} \cdot \mathbf{TQ}_6(p, \boldsymbol{g}).$$

When using Theorem 11 to actually compute entries of $\mathbf{N}_6(p, \boldsymbol{g})$, it is worth noting that most of the entries of $\mathbf{TQ}_6(p, \boldsymbol{g})$ are zero. The following reduction simplifies this computation significantly.

Let $\mathbf{TQ}_3(p, \boldsymbol{g})$ be the vector of length 3^{k+1} whose i-th entry is given by

$$\mathrm{TQ}(p, \prod_{d=0}^{k} g_d^{i_d}),$$

where $(i_0, \ldots, i_k)$ is the base-3 representation of i. Let $\mathbf{A}_{63}$ denote the 6 by 3 matrix obtained from $\mathbf{M}_6^{-1}$ by simply deleting the last three columns, i.e.,

$$\mathbf{A}_{63} \equiv \begin{bmatrix} 1 & 0 & -1 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & \frac{1}{2} & -\frac{1}{2} \\ 0 & 0 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} \\ 1 & -\frac{1}{2} & -\frac{1}{2} \end{bmatrix}.$$

The following theorem has been verified in PVS (see Appendix A).

Theorem 12

$$\mathbf{N}_6(p, \boldsymbol{g}) = \mathbf{A}_{63}^{\otimes(k+1)} \cdot \mathbf{TQ}_3(p, \boldsymbol{g}).$$

Theorem 12 follows from the fact that any entry of $\mathbf{TQ}_6(p, \boldsymbol{g})$ is 0 if the base-6 representation of its index contains any number greater than 2. Again, the formal proof is straightforward, but tedious due to multiple conversions between natural numbers in standard, base-3, and base-6 representation.

5.2 Decision Procedure for Polynomial Systems

This section presents a formally defined decision procedure that computes whether a system of univariate polynomial relations has at least one solution in $\mathbb{R}$, where each polynomial has rational coefficients. Analogous to the formal development presented in Section 4, the procedure will be defined for systems with integer coefficients, and a preprocessing step of clearing denominators is used when the polynomials have rational coefficients.

The PVS functions for determining satisfiability of systems of polynomials encode the system with the sequence $\boldsymbol{g} = (g_0, \ldots, g_k)$ of polynomials, as well as a sequence $\boldsymbol{r} = (r_0, \ldots, r_k)$ of elements of the set $\mathbb{N}^5 \equiv \{0, 1, 2, 3, 4, 5\}$. Elements in $\mathbb{N}^5$ correspond to relations on the polynomials in $\boldsymbol{g}$ via the bijection R.

$$R(j) \equiv \begin{cases} = & \text{if } j = 0, \\ > & \text{if } j = 1, \\ < & \text{if } j = 2, \\ \neq & \text{if } j = 3, \\ \geq & \text{if } j = 4, \\ \leq & \text{if } j = 5. \end{cases}$$

The first step in defining the decision procedure is to define a function `comp_NSol` that takes as input an integer polynomial p, a sequence of integer polynomials $\boldsymbol{g}$ (of length k), and the sequence $\boldsymbol{r}$ (of length k), and returns the number

$$\text{card}(\{x \in \mathbb{R} : \; p(x) = 0 \; \wedge \; g_0(x) \; R(r_0) \; 0 \; \wedge \; \ldots \; \wedge \; g_k(x) \; R(r_k) \; 0\}).$$

This number is the j-th entry of the vector $\mathbf{N}_3(p, \boldsymbol{g})$, where

$$j \; = \sum_{d=0}^{k} r_d \cdot 6^k. \tag{27}$$

The function that performs this calculation, called `comp_NSol`, is defined as follows.

$$\texttt{comp\_NSol}(p, \boldsymbol{g}, \boldsymbol{r}) \equiv (\texttt{row}(\mathbf{A}_{63})(r_0) \otimes \cdots \otimes \texttt{row}(\mathbf{A}_{63})(r_k)) \cdot \mathbf{TQ}_3(p, \boldsymbol{g}).$$

It is worth noting that $(\texttt{row}(\mathbf{A}_{63})(r_0) \otimes \cdots \otimes \texttt{row}(\mathbf{A}_{63})(r_k))$ is a tensor product of $k+1$ rows of $\mathbf{A}_{63}$. This tensor product of rows is equal to the j-th row of $\mathbf{A}_{63}^{\otimes(k+1)}$, where j is given by Equation (27). The function is defined using this tensor product of rows to avoid computing the entire matrix $\mathbf{A}_{63}^{\otimes(k+1)}$ when a single row will suffice. In fact, the PVS development does not directly compute the dot product in the equation above. In many cases, a given entry of $(\texttt{row}(\mathbf{A}_{63})(r_0) \otimes \cdots \otimes \texttt{row}(\mathbf{A}_{63})(k))$ will be 0, making it unnecessary to compute the corresponding entry of $\mathbf{TQ}_3(p, \boldsymbol{g})$. Instead, the algorithm computes this dot product in a way that calculates an entry of $\mathbf{TQ}_3(p, \boldsymbol{g})$ only when the

corresponding entry of $(\texttt{row}(\mathbf{A}_{63})(r_0) \otimes \cdots \otimes \texttt{row}(\mathbf{A}_{63})(k))$ is nonzero. Specifying and and verifying the correctness of this computation requires some work in the PVS development, but helps to improve the efficiency of the decision procedure.

The next step in defining a decision procedure is to account for systems which may not include an equality relation. To accommodate such systems, the following approach is used. For any collection of polynomials $g_0, \ldots, g_k$ and relations $R_0, \ldots, R_k$, where $R_d \in \{=, >, <, \neq, \geq, \leq\}$ for $0 \leq d \leq k$, the system $S \equiv g_0(x)\ R_0\ 0 \ \wedge\ \ldots\ \wedge\ g_k(x)\ R_k\ 0$ is satisfiable if and only if one of the following conditions holds, where Q is the polynomial $\prod_{d=0}^{k} g_d$.

- S is satisfiable at $-\infty$.
- S is satisfiable at ∞.
- S and $Q = 0$ are satisfiable at a common point.
- S and $Q' = 0$ are satisfiable at a common point.

This result is not proven as a stand-alone theorem in PVS, but is instead embedded in the functions `compute_solvable_single` and `compute_solvable` to follow, which are proven to be correct.

Furthermore, it is convenient for the PVS proofs if none of the equations contains a $<$ or $\leq$ relation. This reduces the number of cases in the proof. Thus, the decision procedure converts every $<$ relation (respectively, $\leq$) to a $>$ relation (respectively, $\geq$), multiplying each of the corresponding polynomials by -1. This is accomplished through functions `greatify_rel` and `greatify_poly`. The function `greatify_rel` takes the sequence $\boldsymbol{r}$ as an input and returns a new sequence of elements in $\mathbb{N}^5$, the same length as $\boldsymbol{r}$, whose i-th entry is given by

$$\texttt{greatify\_rel}(\boldsymbol{r})_i \equiv \begin{cases} 1 & \text{if } r_i = 2, \\ 4 & \text{if } r_i = 5, \\ r_i & \text{otherwise.} \end{cases}$$

The function `greatify_poly` takes the sequences $\boldsymbol{g}$ and $\boldsymbol{r}$ as inputs and returns another sequence of polynomials whose i-th entry is given by

$$\texttt{greatify\_poly}(\boldsymbol{g}, \boldsymbol{r})_i \equiv \begin{cases} (-1) \cdot g_i & \text{if } r_i = 2 \ \vee\ r_i = 5, \\ g_i & \text{otherwise.} \end{cases}$$

Recall that the decision procedure for satisfiability takes an arbitrary number k of polynomials as input. Partly to make the proof more modular, a function is defined for when there is precisely one polynomial relation, i.e., $k = 0$. The function `compute_solvable_single` is defined below. It takes as input an element $i \in \mathbb{N}^5$ and a nonzero polynomial p, which has integer coefficients. It

returns a Boolean value.

$$
\begin{array}{l}
\texttt{compute\_solvable\_single}(i,p) \equiv \\
\quad \texttt{if } \texttt{deg}(p) = 0 \texttt{ then } \texttt{coeff}(p,0)\ R(i)\ 0 \\
\quad \texttt{elsif } \texttt{deg}(p) = 1\ \vee\ i = 3 \texttt{ then true} \\
\quad \texttt{elsif } i = 0 \texttt{ then } \texttt{comp\_NSol}(p,(1),(1)) \\
\quad \texttt{elsif } i \neq 1\ \wedge i \neq 2\ \wedge\ \texttt{comp\_NSol}(p,(1),(1)) \texttt{ then true} \\
\quad \texttt{else} \\
\quad\ \texttt{let}\ \ p^* = \texttt{if } (1\ R(i)\ 0) \texttt{ then } p \texttt{ else}(-1)\cdot p \texttt{ endif}\ \ \texttt{in} \\
\qquad \texttt{coeff}(p^*, \texttt{deg}(p)) > 0\ \vee \\
\qquad (\text{odd}(\texttt{deg}(p))\ \wedge\ \texttt{coeff}(p^*, \texttt{deg}(p)) < 0)\ \vee \\
\qquad \texttt{comp\_NSol}(p', (p^*), \texttt{greatify\_rel}((i))) \neq 0 \\
\quad \texttt{endif}.
\end{array}
\tag{28}
$$

The following theorem has been proved in PVS (see Appendix A).

Theorem 13 *Let p be an integer polynomial and $i \in \mathbb{N}^5$. There is a real number x such that $p(x)\ R(i)\ 0$ if and only if* `compute_solvable_single`$(i,p) =$ `true`.

The function for determining satisfiability of a system of polynomial relations, for integer polynomials, is named `compute_solvable`. It has as parameters a sequence $\boldsymbol{g} = (g_0, \ldots, g_k)$ of integer polynomials and a sequence $\boldsymbol{r} = (r_0, \ldots, r_k)$. The function returns a Boolean value. If any element of $\boldsymbol{r}$ is equal to 0, which means that one of the relations in the system is an equality, then the function `comp_NSol` can be directly used to determine satisfiability. The function `first_eq`, defined below, returns the first number j for which $r_j = 0$, and if there is no such j, then it returns $k+1$.

$$
\begin{array}{l}
\texttt{first\_eq}(\boldsymbol{r}) \equiv \texttt{if } \exists\ (j \leq k) :\ r_j = 0 \texttt{ then} \\
\qquad\qquad \min(\{j \leq k \mid r_j = 0\}) \\
\qquad\quad \texttt{else} \\
\qquad\quad\ k+1 \\
\qquad\quad \texttt{endif}.
\end{array}
$$

The PVS function `compute_solvable` is defined below.

$$
\begin{array}{l}
\texttt{compute\_solvable}(\boldsymbol{g},\boldsymbol{r}) \equiv \\
\quad \texttt{let} \\
\quad\quad \boldsymbol{g}^* = \texttt{greatify\_poly}(\boldsymbol{g},\boldsymbol{r}), \\
\quad\quad \boldsymbol{r}^* = \texttt{greatify\_rel}(\boldsymbol{r}), \\
\quad\quad e = \texttt{first\_eq}(\boldsymbol{r}), \\
\quad\quad \texttt{Qprod} = \displaystyle\prod_{i=0}^{k} g_i^*, \\
\quad\quad \texttt{G} = (g_0^*, \ldots, g_{e-1}^*, g_{e+1}^*, \ldots, g_k^*), \\
\quad\quad \texttt{R} = (r_0^*, \ldots, r_{e-1}^*, r_{e+1}^*, \ldots, r_k^*), \\
\quad \texttt{in} \\
\quad\quad \texttt{if } k = 0 \texttt{ then compute\_solvable\_single}(r_0, g_0) \\
\quad\quad \texttt{elsif } (\exists\ (i \le k) :\ r_i^* = 0) \texttt{ then comp\_NSol}(g_e^*, \texttt{G}, \texttt{R}) \\
\quad\quad \texttt{elsif deg}(\texttt{Qprod}) > 0\ \wedge\ \texttt{comp\_NSol}(\texttt{Qprod}, \boldsymbol{g}^*, \boldsymbol{r}^*) \texttt{ then true} \\
\quad\quad \texttt{elsif deg}(\texttt{Qprod}) < 2 \texttt{ then false} \\
\quad\quad \texttt{elsif } (\forall\ (j \le k) :\ \texttt{coeff}(g_j^*, \texttt{deg}(g_j^*))\ R(r_j^*)\ 0) \texttt{ then true} \\
\quad\quad \texttt{elsif } (\forall\ (j \le k) : \\
\quad\quad\quad\quad (\text{odd}(\texttt{deg}(g_j^*)) \ \Rightarrow\ -\texttt{coeff}(g_j^*, \texttt{deg}(g_j^*))\ R(r_j^*)\ 0) \vee \\
\quad\quad\quad\quad (\text{even}(\texttt{deg}(g_j^*)) \ \Rightarrow\ \texttt{coeff}(g_j^*, \texttt{deg}(g_j^*))\ R(r_j^*)\ 0)) \\
\quad\quad\quad \texttt{then true} \\
\quad\quad \texttt{else comp\_NSol}(\texttt{Qprod}', \boldsymbol{g}^*, \boldsymbol{r}^*) \\
\quad\quad \texttt{endif}.
\end{array}
\tag{29}
$$

The following theorem, which is the correctness statement for the decision procedure `compute_solvable` for integer polynomials, has been proved in PVS (see Appendix A).

Theorem 14 *If each polynomial in the sequence* $\boldsymbol{g} = (g_0, \ldots, g_k)$ *has integer coefficients and positive degree, then there exists a real number* x *such that* $g_j(x)\ R(\boldsymbol{r}(j))\ 0$*, for all* $j \le k$*, if and only if* $\texttt{compute\_solvable}(\boldsymbol{g},\boldsymbol{r}) = \texttt{true}$*.*

As in the development presented in Section 4, the decision procedure for rational polynomials is defined based on the decision procedure for integer polynomials, reusing the function `rat2poly`, which returns an integer polynomial that is a positive multiple of the input rational polynomial.

$$
\texttt{tarski}(\boldsymbol{q},\boldsymbol{r}) \equiv \texttt{compute\_solvable}((p_0, \ldots, p_k)), \boldsymbol{r}),
\tag{30}
$$

where $p_d = \texttt{rat2poly}(q_d)$, for $0 \le d \le k$. The following theorem has been formally proved in PVS (see Appendix A).

Theorem 15 *If each polynomial in the sequence* $\boldsymbol{q} = (q_0, \ldots, q_k)$ *has rational coefficients and positive degree, then there exists a real number* x *such that* $q_j(x)\ R(\boldsymbol{r}(j))\ 0$*, for all* $j \leq k$*, if and only if* $\texttt{tarski}(\boldsymbol{g}, \boldsymbol{r}) = \texttt{true}$*.*

6 Automated Strategies

The functions `sturm`, `mono`, and `tarski`, and their correctness properties, i.e., Theorem 8, Theorem 9, and Theorem 15, respectively, can be used to mechanically prove, in a theorem prover, properties involving univariate polynomials. The technical details of how this is done depend on the particular theorem prover where those functions and theorems are specified and verified. This section concerns the mechanical verification of polynomial inequalities in PVS, but a similar approach is directly applicable to other interactive theorem provers.

6.1 PVS Theorem Prover

In a nutshell, a PVS sequent is a logical judgement of the form $\Gamma \vdash \Delta$, where Γ, called the *antecedent*, and Δ, called the *consequent*, are lists of logical formulas. The intuitive meaning of a sequent is that from the conjunction of formulas in Γ, the disjunction of formulas in Δ can be deduced. PVS proof commands are logical rules that transform a sequent into a set of sequents, with the objective of producing sequents where the formula `false` appears in the antecedent or the formula `true` appears in the consequent. When all sequents generated by a proof command are of one of these forms, the initial sequent is *discharged*. Hence, a *proof* in PVS can be represented by a tree of proof commands that discharge an initial sequent.

The PVS type `real` is a primitive type, i.e., it is axiomatically defined in the PVS prelude. Furthermore, PVS supports subtyping. For example, the types `int` (integer numbers), `rat` (rational numbers), and `real` (real numbers) are axiomatically defined such that `nat` is a subtype of `int`, `int` is a subtype of `rat`, and `rat` is a subtype of `real`. Furthermore, all numerical constants are members of `rat`. Decimal notation is supported as syntactic sugar for rational numbers, e.g., the decimal number 0.52 represents the rational number $\frac{52}{100}$. In PVS, expressions such as $(x-1)^2$ and $x^2 - 2x + 1$, where x is a PVS variable of type `real`, are both real number expressions. The fact that they represent the same real number is not, apriori, known to PVS. Such an equality has to be verified in the theorem prover.

As with most modern interactive theorem provers, the set of proof commands of PVS can be conservatively extended by user-defined proof rules. In PVS, user-defined proof rules are called *strategies* and they are procedures written in the PVS strategy language that, when executed by the proof engine, produce PVS proofs. The PVS strategy language includes combinators for sequencing, branching, and backtracking of proof commands. The language also

provides mechanisms to inspect the internal representation of PVS syntactic elements within a proof context. PVS strategies, as tactics in the LCF-style of interactive theorem provers, preserve the logical consistency of the proof system. That is, any strategy within a proof can be expanded into a tree of proof commands that only includes basic PVS proof rules.

This section concerns the development of PVS strategies for reasoning, in an automated way, about relations on a class of real number expressions, called polynomial expressions, involving a real number variable x. Formally, *polynomial expressions* on a variable x of type `real` are formal PVS expressions of type `real` whose syntax is inductively defined as follows. Numerical constants and the real number variable x are polynomials expressions on x. Furthermore, if a and b are polynomial expressions on x, n is a natural number constant, and k is a nonzero numerical constant, then $a + b$, $-a$, $a - b$, $a\,b$, a^n, and $\frac{a}{k}$ are all polynomial expressions on x. Henceforth, the expression $\mathtt{p}\langle x\rangle$ denotes a polynomial expression on x. It is clear that any function that maps a real number x into a polynomial expression $\mathtt{p}\langle x\rangle$ is a polynomial function and can be represented by its list of coefficients as explained in Section 2.

6.2 Computational Reflection

The following example describes the use of Theorem 8 to discharge a PVS sequent involving polynomial expressions. Consider the sequent below displayed in a vertical layout. Formulas in a PVS sequent are numbered using the notation $\{n\}$, where $n < 0$ in the antecedent and $n > 0$ in the consequent. In this sequent, x is a free real variable (actually, a Skolem real constant in PVS terminology).

$$\begin{array}{l}\{-1\}\ x < 3\\ \vdash\\ \{\ 1\ \}\ x^{120} - \frac{2}{3}x^{60} + \frac{1}{9} \geq 0\end{array}$$

1. The first step in the PVS proof is to perform a proof command that instantiates Theorem 8 such that I is the interval $(-\infty, 3)$ and p is a list of rational numbers that is 0 everywhere except in the positions 0, 60, and 120, where it has the values $\frac{1}{9}$, $-\frac{2}{3}$, and 1, respectively. That proof command yields the following sequent.

$$\begin{array}{l}\{-1\}\ \mathtt{sturm}(p, I, \geq) \iff (\forall(x\colon \mathbb{R})\colon x \in I \implies p(x) \geq 0)\\ \{-2\}\ x < 3\\ \vdash\\ \{\ 1\ \}\ x^{120} - \frac{2}{3}x^{60} + \frac{1}{9} \geq 0\end{array}$$

2. Next, the Boolean value resulting from the evaluation of $\mathtt{sturm}(p, I, \geq)$ is found. This evaluation can be effectively performed, for example, by expanding the definition of `sturm` and simplifying the resulting expression.

The following sequent is obtained.

$$\begin{array}{l}\{-1\}\ \texttt{true} \iff (\forall(x\colon \mathbb{R})\colon x \in I \implies p(x) \geq 0)\\ \{-2\}\ x < 3\\ \vdash\\ \{\ 1\ \}\ x^{120} - \frac{2}{3}x^{60} + \frac{1}{9} \geq 0\end{array}$$

3. Sequent formula $\{-1\}$ can be reduced using propositional simplification to

$$\begin{array}{l}\{-1\}\ \forall(x\colon \mathbb{R})\colon x \in I \implies p(x) \geq 0\\ \{-2\}\ x < 3\\ \vdash\\ \{\ 1\ \}\ x^{120} - \frac{2}{3}x^{60} + \frac{1}{9} \geq 0\end{array}$$

4. Next, the quantified variable x in sequent formula $\{-1\}$ is instantiated with the Skolem constant x appearing in sequent formulas $\{-2\}$ and $\{1\}$. This instantiation yields the sequent.

$$\begin{array}{l}\{-1\}\ x \in I \implies p(x) \geq 0\\ \{-2\}\ x < 3\\ \vdash\\ \{\ 1\ \}\ x^{120} - \frac{2}{3}x^{60} + \frac{1}{9} \geq 0\end{array}$$

5. The elimination of the implication in sequent formula $\{-1\}$ yields two sequents.

$$\begin{array}{l}\{-1\}\ x < 3\\ \vdash\\ \{\ 1\ \}\ x \in I\\ \{\ 2\ \}\ x^{120} - \frac{2}{3}x^{60} + \frac{1}{9} \geq 0\end{array} \qquad \begin{array}{l}\{-1\}\ p(x) \geq 0\\ \{-2\}\ x < 3\\ \vdash\\ \{\ 1\ \}\ x^{120} - \frac{2}{3}x^{60} + \frac{1}{9} \geq 0\end{array}$$

6. The left-hand side and right-hand side sequents can be discharged by fully expanding the definitions in sequent formulas 1 and -1, respectively.

In the example above, the discharged sequent involves the polynomial expression $\mathtt{p}\langle x\rangle \equiv x^{120} - \frac{2}{3}x^{60} + \frac{1}{9}$, whereas the function `sturm` and its correctness theorem involve the polynomial representation $p \equiv (\frac{1}{9}, \ldots, -\frac{2}{3}, \ldots, 1)$, which is a list of real numbers. The equality $p(x) = \mathtt{p}\langle x\rangle$ has to be formally verified in PVS. This example illustrates a well-known technique in theorem proving known as *computational reflection* [19]. The key steps in a proof by computational reflection are a change of representation from objects of a theory of interest into a target theory and the use of trusted or verified algorithms in the target theory that decide particular types of properties. The correctness properties of these algorithms relate properties in the target theory to properties in the object theory. In the development presented in this paper, the theory of interest is that of polynomial expressions, the target theory are polynomials represented as lists of rational numbers, the properties of interest are real order relations, the verified algorithms are `sturm`, `mono`, and `tarski`, and their correctness properties are stated by Theorem 8, Theorem 9, and Theorem 15.

Computational reflection is particularly well-adapted for the development of automated proof strategies in interactive theorem provers. It produces proofs whose size is independent of the size of the initial sequent. In particular, the same proof method used in the example above can be used for discharging inequalities on any polynomial expression. Furthermore, proofs by computational reflection are small since the most involved logical steps are done once for all in the proof of theorems such as Theorem 8 and Theorem 15. Finally, from the point of view of the proof engine, proofs by computational reflection are efficient since they depend on computation rather than on deduction.

The two key steps of a computational reflection proof, i.e., changing the representation of the objects of interest and evaluating the verified algorithms, can be implemented in the PVS strategy language in several ways. Neither of these steps is particularly challenging from a logical point of view. However, implementing them in a way that is computationally efficient in PVS requires some additional formalization effort.

6.3 A Deep Embedding of Univariate Polynomials

In the PVS strategy language, it is possible to traverse the internal representation of a polynomial expression such as $x^{120} - \frac{2}{3}x^{60} + \frac{1}{9}$ and construct a concrete list p of rational numbers that represents it. The code that implements this construction does not have to be trusted since the equality $p(x) = x^{120} - \frac{2}{3}x^{60} + \frac{1}{9}$ is discharged by the strategy. Since $p(x)$ is equal to $\texttt{polylist}(p)(x)$, to discharge that equality it suffices to fully expand the definition of `polylist`. This approach was followed in a first implementation of the strategies presented in this paper. However, it has some drawbacks. To be useful, the code that constructs the polynomial representation should recognize any type of polynomial expressions, not only those that are written as sum of monomials, and construct their representations accordingly. This approach results in a large amount of strategy code. Furthermore, expanding the definition of `polylist`, which is a recursive function, may be inefficient. Since the PVS theorem prover does not provide enough control to expand a particular occurrence of a definition, expressions involving `polylist` tend to grow quickly, before they are simplified by the PVS strategies.

For the current version of the strategies presented below, an alternative approach to proving $p(x) = \mathtt{p}\langle x\rangle$ is implemented, which requires a deep embedding of univariate polynomials. In this deep embedding, a polynomial function is still represented by a list of rational numbers, but the concrete list is not computed by the strategies. Instead, operations to construct those lists are directly defined in PVS. For instance, the PVS function `pconst` has as parameter a rational number c and returns a list that contains c as its only element. List representations of monomials are constructed with the function `pconst`, which takes as parameters a rational number c and a natural number i and returns the list of length $i+1$ that has zeros everywhere except at its last positions where it has the value c. Similarly, functions `psum`, `pminus`, `pneg`, `pprod`,

`pscal`, `pdiv`, and `ppow` are defined that take as parameters list representations of polynomial functions and return list representations of their addition, subtraction, negation, product, scalar multiplication by a given rational number, division by a given rational number, and power to a given natural number. The following theorem, which shows the correctness of the embedding, has been formally proved in PVS (see Appendix A).

Theorem 16 *For all polynomial representations p and q, rational number c, natural number i, and real number x*

- $\mathit{polylist}(\mathit{pconst}(c))(x) = c$.
- $\mathit{polylist}(\mathit{pmonom}(c, i))(x) = c\ x^i$.
- $\mathit{polylist}(\mathit{psum}(p, q))(x) = p(x) + q(x)$.
- $\mathit{polylist}(\mathit{pneg}(p))(x) = -p(x)$.
- $\mathit{polylist}(\mathit{pminus}(p, q))(x) = p(x) - q(x)$.
- $\mathit{polylist}(\mathit{pprod}(p, q))(x) = p(x) \cdot q(x)$.
- $\mathit{polylist}(\mathit{pscal}(c, p))(x) = c\ p(x)$.
- $\mathit{polylist}(\mathit{pdiv}(p, c))(x) = \frac{p(x)}{c}$, *when* $c \neq 0$.
- $\mathit{polylist}(\mathit{ppow}(p, i))(x) = p(x)^i$.

In the new approach, the strategy code takes a polynomial expression such as $x^{120} - \frac{2}{3}x^{60} + \frac{1}{9}$ and constructs its representation using the deep embedding, i.e., $\mathtt{psum}(\mathtt{pminus}(\mathtt{pmonom}(1, 120), \mathtt{pmonom}(\frac{2}{3}, 60)), \mathtt{pconst}(\frac{1}{9}))$. The strategy code required to construct this object is considerably simpler than the code that constructs a concrete list representation of the polynomial expression. Indeed, it is a simple pretty-printer that reflects the syntactical structure of the input polynomial expression. The proof that $p(x) = \mathtt{p}\langle x \rangle$, where p is the deep embedding of the polynomial expression $\mathtt{p}\langle x \rangle$, proceeds by rewriting the left-hand side of this equality using Theorem 16. For simple polynomials, the new approach is slightly slower than the original one due to the overhead created by the deep embedding. However, since a polynomial expression and its deep embedding are syntactically isomorphic, rewriting is significantly faster than expanding, especially on large polynomial expressions.

6.4 Ground Evaluation

The computational approach used by the strategies presented in this paper requires the evaluation of algorithms, defined as PVS functions, on ground parameters. This evaluation can be performed by fully expanding the definitions involved in the algorithms and simplifying the resulting expressions using PVS's deductive rules. As explained in Section 6.3, the limited control offered by the PVS theorem prover to perform expansions makes this approach impractical. In the case of ground expressions, PVS provides, as part of its trusted code base, a ground evaluator [32]. The ground evaluator efficiently evaluates PVS expressions by translating them into Lisp objects and executing them using the PVS's Lisp engine. The ground evaluator supports a large set of the PVS language including higher order functions, literals of primitive

types, abstract datatypes, and bounded quantifications over integers [38]. It should be noted that while the soundness of the strategies presented in the following section depends on the correctness of the ground evaluator, the formal development presented in the previous sections does not. Furthermore, it is theoretically possible, although impractical, to replace every instance of the PVS ground evaluator in a proof by computational reflection by a strategy that only depends on symbolic evaluation such as PVS's `grind`.

6.5 PVS Strategies `sturm`, `mono-poly`, `tarski`

The formal development presented in this paper includes the PVS strategies `sturm`, `mono-poly`, and `tarski` that implement computational reflection methods for automatically proving properties involving polynomial inequalities.

The strategy `sturm` implements an enhanced version of the method described in Section 6.2. More generally, the strategy `sturm` automatically discharges sequents having one of the following forms

1. $X_1, \ldots, X_m, \Gamma \vdash \underline{\mathtt{p}_1\langle x\rangle\ R\ \mathtt{p}_2\langle x\rangle}, \Delta$
2. $X_1, \ldots, X_m, \underline{\mathtt{p}_1\langle x\rangle\ R'\ \mathtt{p}_2\langle x\rangle}, \Gamma \vdash \Delta$
3. $\Gamma \vdash \underline{\forall(x\colon T)\colon X_1 \wedge \ldots \wedge X_m \implies \mathtt{p}_1\langle x\rangle\ R\ \mathtt{p}_2\langle x\rangle}, \Delta$
4. $\underline{\forall(x\colon T)\colon X_1 \wedge \ldots \wedge X_m \implies \mathtt{p}_1\langle x\rangle\ R\ \mathtt{p}_2\langle x\rangle}, \Gamma \vdash \Delta$
5. $\Gamma \vdash \underline{\exists(x\colon T)\colon X_1 \wedge \ldots \wedge X_m \wedge \mathtt{p}_1\langle x\rangle\ R'\ \mathtt{p}_2\langle x\rangle}, \Delta$
6. $\underline{\exists(x\colon T)\colon X_1 \wedge \ldots \wedge X_m \wedge \mathtt{p}_1\langle x\rangle\ R'\ \mathtt{p}_2\langle x\rangle}, \Gamma \vdash \Delta$

where

- T is a subtype of $\mathbb{R}$,
- Γ and Δ are arbitrary lists of formulas, which are ignored by the strategy,
- $m \geq 0$ and for $1 \leq j \leq m$, X_m denotes a Boolean expression of one of the forms $a \prec x$, $x \prec a$, $|x| \prec a$, or $x \in I$, where $\prec \in \{<, \leq\}$, a is a numerical rational constant, and I is an interval, whose bounds are extended real numbers,
- $\mathtt{p}_1\langle x\rangle$ and $\mathtt{p}_2\langle x\rangle$ denote polynomial expressions on a real number variable x,
- R is a relation in $\{<, \leq, >, \geq, \neq\}$ and R' is a relation in $\{<, \leq, >, \geq, =\}$.

The strategy `sturm` works on a formula of interest, which is the underlined formula in the forms above. By default, the strategy assumes that the formula of interest is the first formula in the consequent, but the user can specify a different formula through an optional parameter in the strategy. As specified above, the formula of interest can appear in the antecedent or in the consequent, be nonquantified or quantified, and the quantifier can be existential or universal.

The strategy proceeds as follows. First, it examines the sequent and determines whether or not it has one of the supported forms. If this is not the case, the strategy prints an error message and does nothing else. If the sequent is supported, the strategy considers the formula of interest and constructs a PVS expression p that represents the deep embedding of the polynomial expression

$\mathtt{p}_1\langle x\rangle - \mathtt{p}_2\langle x\rangle$. Next, the strategy computes PVS data structures for instantiating the interval I in Theorem 8. These data structures are constructed by examining the relational formulas X_j, $1 \leq j \leq m$ and the type T. The correctness of the strategy is not compromised by the constructions of these objects. Indeed, the strategy formally verifies that the semantics of the original expressions and their data structure embeddings coincide.

The instantiation of the relation in Theorem 8 depends on the form of the sequent. In the case of forms 1, 3, and 4, the relation is instantiated with R. Otherwise, the relation is instantiated with $\neg R'$. From this point on, the strategy follows a computational reflection approach. Some minor modifications are needed for the cases where the formula of interest is quantified. Sequents of the forms 3 and 6 can be transformed into sequents of the forms 1 and 2 by introducing the quantified variable (this process is called Skolemization, in PVS terminology) and propositional simplification. Sequents of the forms 4 and 5 use the fact that Theorem 8 is a double implication. In this case, the evaluation of the function `sturm` on the given parameters should return `false` and the rest of the proof proceeds accordingly. Finally, if the sequent holds, the strategy succeeds and the sequent is discharged. If the sequent does not hold, the strategy prints a message stating that the sequent is not provable.

Example 3 Turan's inequality for Legendre's polynomials is a theorem that states that $p_n(x)^2 > p_{n-1}(x) \cdot p_{n+1}(x)$ for all x such that $-1 < x < 1$, where p_j is the j-th Legendre's polynomial (of degree j). Turan's inequality for $n = 9$ yields the following sequent

$$\begin{array}{l} \{-1\}\ |x| < 1 \\ \vdash \\ \{\ 1\ \}\ \mathtt{p}_9\langle x\rangle^2 < \mathtt{p}_8\langle x\rangle \cdot \mathtt{p}_{10}\langle x\rangle \end{array}$$

where

- $\mathtt{p}_8\langle x\rangle \equiv \frac{6435x^8-12012x^6+6930x^4-1260x^2+35}{128}$,
- $\mathtt{p}_9\langle x\rangle \equiv \frac{12155x^9-25740x^7+18018x^5-4620x^3+315x}{128}$, and
- $\mathtt{p}_{10}\langle x\rangle \equiv \frac{46189x^{10}-109395x^8+90090x^6-30030x^4+3465x^2-63}{256}$.

The strategy `sturm` automatically discharges this sequent in less than 5 seconds.[5]

The strategy `mono-poly` uses the PVS function `mono` and its correctness property (Theorem 9) to automatically prove properties regarding monotonicity of univariate polynomials in a given variable range. As in the case of the strategy `sturm`, the strategy `mono-poly` works on a formula of interest. By default, the strategy assumes that the formula of interest is the first formula in the consequent, but the user can specify a different formula through an optional parameter in the strategy. The strategy `mono-poly` discharges sequents having one of the following forms, where the formula of interest is underlined.

[5] All the examples in this paper are executed in a MacBook Pro 2.4 GHz Inter Core 2 Duo with 8 GB of memory.

1. $Y_1, \ldots, Y_m, \Gamma \vdash \underline{\mathtt{p}_1\langle x\rangle\ R_2\ \mathtt{p}_2\langle y\rangle}, \Delta$
2. $Y_1, \ldots, Y_m, \underline{\mathtt{p}_1\langle x\rangle\ R_2'\ \mathtt{p}_2\langle y\rangle}, \Gamma \vdash \Delta$
3. $\Gamma \vdash \underline{\forall(x, y\colon T)\colon Y_1 \wedge \ldots \wedge Y_m \implies \mathtt{p}_1\langle x\rangle\ R\ \mathtt{p}_2\langle y\rangle}, \Delta$
4. $\underline{\forall(x, y\colon T)\colon Y_1 \wedge \ldots \wedge Y_m \implies \mathtt{p}_1\langle x\rangle\ R\ \mathtt{p}_2\langle y\rangle}, \Gamma \vdash \Delta$
5. $\Gamma \vdash \underline{\exists(x, y\colon T)\colon Y_1 \wedge \ldots \wedge Y_m \wedge \mathtt{p}_1\langle x\rangle\ R'\ \mathtt{p}_2\langle y\rangle}, \Delta$
6. $\underline{\exists(x, y\colon T)\colon Y_1 \wedge \ldots \wedge Y_m \wedge \mathtt{p}_1\langle x\rangle\ R'\ \mathtt{p}_2\langle y\rangle}, \Gamma \vdash \Delta$

where

- T is a subtype of $\mathbb{R}$,
- Γ and Δ are arbitrary lists of formulas, which are ignored by the strategy,
- $m \geq 0$ and for $1 \leq j \leq m$, Y_m denotes a Boolean expression of one of the forms $a \prec v$, $v \prec a$, $|v| \prec a$, $v \in I$, or $x\ R_1\ y$, where $\prec \in \{<, \leq\}$, $R_1 \in \{<, \leq, >, \geq, \neq\}$, $v \in \{x, y\}$, a is a numerical rational constant, and I is an interval, whose bounds are extended real numbers,
- $\mathtt{p}_1\langle x\rangle$ and $\mathtt{p}_2\langle y\rangle$ denote polynomial expressions on real number variables x and y, respectively,
- R_2 are relations in $\{<, \leq, >, \geq, \neq\}$ and R_2' is a relation in $\{<, \leq, >, \geq, =\}$.

The strategy determines whether or not (a) the sequent has one of the supported forms, (b) variables x and y are defined on the same range, and (c) at least one of the formulas Y_m has the form $x\ R_1\ y$. If this is not the case, the strategy prints an error message and does nothing else. If the sequent is supported, the strategy constructs deep embeddings of the polynomial expressions $\mathtt{p}_1\langle x\rangle$ and $\mathtt{p}_2\langle y\rangle$ and ground evaluates them into lists of rational numbers. If the resulting lists are not equal, the strategy fails with a message and does nothing else. Otherwise, it proceeds by computational reflection applying the function `mono` and Theorem 9 to appropriate parameters according to the form of the sequent.

Example 4 The following sequent states that the 10-th Legendre's polynomial is decreasing in the range $[-0.75, -0.6]$.

$$\begin{array}{l}\{-1\}\ x \in [-0.75, -0.6]\\ \{-2\}\ y \in [-0.75, -0.6]\\ \{-3\}\ x < y\\ \vdash\\ \{\ 1\ \}\ \mathtt{p}_{10}\langle x\rangle > \mathtt{p}_{10}\langle y\rangle\end{array}$$

where the polynomial p_{10} is defined as in Example 3. The sequent is automatically discharged by the strategy `mono-poly` in less than 5 seconds.

Finally, the strategy `tarski` automatically discharges existential conjunctive formulas of polynomial inequalities or universal disjunctive formulas of polynomial inequalities. The strategy supports a superset of the sequents supported by the strategy `sturm`. On the sequents supported by both strategies, `sturm` is generally faster than `tarski` since the later encodes the conditions on the variable range as additional polynomial constraints. In contrast to the

strategies `sturm` and `mono-poly`, the strategy `tarski` allows for the specification of several formulas of interest. By default, the strategy assumes that the formulas of interests are all the formulas in the sequent, but the user can specify a different formula through an optional parameter in the strategy. The sequents supported by the strategy `tarski` have one the following forms, where the formulas of interest are underlined.

1. $\underline{P_1, \ldots, P_m}, \Gamma \vdash \underline{P_{m+1}, \ldots, P_n}, \Delta$,
2. $\Gamma \vdash \forall(x\colon T)\colon \underline{P_1 \wedge \ldots \wedge P_m \implies P_{m+1} \vee \ldots \vee P_n}, \Delta$
3. $\underline{\forall(x\colon T)\colon P_1 \wedge \ldots \wedge P_m \implies P_{m+1} \vee \ldots \vee P_n}, \Gamma \vdash \Delta$
4. $\Gamma \vdash \underline{\exists(x\colon T)\colon P_1 \wedge \ldots \wedge P_m}, \Delta$
5. $\underline{\exists(x\colon T)\colon P_1 \wedge \ldots \wedge P_m}, \Gamma \vdash \Delta$

where

- T is a subtype of $\mathbb{R}$,
- Γ and Δ are arbitrary lists of formulas, which are ignored by the strategy,
- $0 \leq m$ and for $1 \leq j \leq m$, P_j has one the forms $\mathtt{p}_j\langle x\rangle\ R_j\ \mathtt{q}_j\langle x\rangle$, $|\mathtt{p}_j\langle x\rangle| \prec_j \mathtt{q}_j\langle x\rangle$, or $\mathtt{p}_j\langle x\rangle \in I_j$, where $R_j \in \{<, \leq, >, \geq, \neq, =\}$, $\prec_j \in \{<, \leq\}$, $\mathtt{p}_j\langle x\rangle$ and $\mathtt{q}_j\langle x\rangle$ denote polynomial expressions on a variable x, and I is an interval, whose bounds are extended real numbers,
- $m \leq n$ and for $m < j \leq n$, P_j has one the forms $\mathtt{p}_j\langle x\rangle\ R_j\ \mathtt{q}_j\langle x\rangle$ or $|\mathtt{p}_j\langle x\rangle| \succ_j \mathtt{q}_j\langle x\rangle$, where $R_j \in \{<, \leq, >, \geq, \neq, =\}$, $\succ_j \in \{>, \geq\}$, and $\mathtt{p}_j\langle x\rangle$ and $\mathtt{q}_j\langle x\rangle$ denote polynomial expressions on a variable x.

As is the case of `sturm` and `mono-poly`, the strategy `tarski` first determines whether or not the sequent has one of the supported forms and reports an error if this is not the case. Then, it computes deep embeddings of polynomials representing $\mathtt{p}_j\langle x\rangle - \mathtt{q}_j\langle x\rangle$ and proceeds by computational reflection by applying the function `tarski` and Theorem 15 to appropriate parameters according to the form of the sequent.

Example 5 The following sequent states that at every point at least one of Legendre's polynomials p_i, with $2 \leq i \leq 6$, is nonnegative.

$$
\begin{array}{l}
\vdash \\
\{1\}\ \mathtt{p}_2\langle x\rangle \geq 0 \\
\{2\}\ \mathtt{p}_3\langle x\rangle \geq 0 \\
\{3\}\ \mathtt{p}_4\langle x\rangle \geq 0 \\
\{4\}\ \mathtt{p}_5\langle x\rangle \geq 0 \\
\{5\}\ \mathtt{p}_6\langle x\rangle \geq 0
\end{array}
$$

where

- $\mathtt{p}_2\langle x\rangle \equiv \frac{3x^2-1}{2}$,
- $\mathtt{p}_3\langle x\rangle \equiv \frac{5x^3-3x}{2}$,
- $\mathtt{p}_4\langle x\rangle \equiv \frac{35x^4-30x^2+3}{8}$,
- $\mathtt{p}_5\langle x\rangle \equiv \frac{63x^5-70x^3+15x}{8}$, and
- $\mathtt{p}_6\langle x\rangle \equiv \frac{231x^6-315x^4+105x^2-5}{16}$.

The strategy `tarski` automatically discharges this sequent in less than 10 seconds.

Example 6 In air traffic management [11], the predicate

$$\begin{aligned}\texttt{conflict?}(\mathbf{s},\mathbf{v},s_z,v_z,T) \equiv \exists\,(t\colon \mathbb{R})\colon\ & t \in [0,T] \wedge \\ & \mathbf{s}^2 + 2\,(\mathbf{s}\cdot\mathbf{v})\,t + \mathbf{v}^2\,t^2 < D^2 \wedge \\ & |s_z + t\,v_z| < H,\end{aligned} \tag{31}$$

where $\mathbf{s}, \mathbf{v}$ are 2-dimensional vectors, s_z, v_z are real numbers, and D, H, T are positive real numbers, specifies that two aircraft flying straight line trajectories will be in *a loss of separation* at some future time $t \leq T$. A loss of separation is a violation of a minimum horizontal distance D, typically 5 nautical miles (9260 m), and a minimum vertical distance H, typically 1000 feet (305 m). In Formula (31), $\mathbf{s}$ denotes the relative horizontal position of the aircraft at time 0, $\mathbf{v}$ denotes their relative horizontal velocity, s_z denotes their relative altitude, and v_z denotes their relative vertical speed.

Let $\mathbf{s}_o = (-37040, 0)$, $s_{oz} = 9144$, $\mathbf{s}_i = (37000, 0)$, and $s_{iz} = 9000$ be the horizontal and vertical positions (in meters) of two aircraft and $\mathbf{v}_o = (218, 218)$, $v_{oz} = 2.54$, $\mathbf{v}_i = (-205, 205)$, $v_{oz} = 2.5$ be their horizontal velocity and vertical speed (in meters per second). The following sequents state that the aircraft will be in conflict when T is 5 minutes, but they will be conflict free when T is 2.5 minutes.

$$\vdash\ \texttt{conflict?}(\mathbf{s}_o - \mathbf{s}_i, \mathbf{v}_o - \mathbf{v}_i, s_{oz} - s_{iz}, v_{oz} - v_{iz}, 300)$$

$$\texttt{conflict?}(\mathbf{s}_o - \mathbf{s}_i, \mathbf{v}_o - \mathbf{v}_i, s_{oz} - s_{iz}, v_{oz} - v_{iz}, 150) \vdash$$

Both sequents are discharged in less than 5 seconds by expanding the definition of `conflict?` and applying `tarski`.

7 Formalization Issues

Every result in the formalization presented in this paper has been formally verified in PVS. There are no axioms other than those defining the PVS logic. The developments `Sturm` and `Tarski`, in the NASA PVS Library, consist of 415 lemmas and 378 lemmas, respectively, which amount to 3119 lines of PVS specification. These numbers do not include lemmas added to other developments in the library, e.g., `reals` and `matrices`, which are needed in the present work. The strategies are encoded in 774 lines of PVS strategy code.

During the PVS specification and verification of properties and algorithms presented in this paper, numerous formalization issues arose. Most of these issues are due to the fact that in a formal development every technical detail in a proof has to be made explicit. Some of these technical details required the specification and development of nontrivial fundamental mathematical concepts. This section describes the main technical issues and the work done to overcome them. This section is primarily for the benefit of other PVS users who might find such solutions useful, and for users of other theorem-provers who may run into similar problems.

7.1 Extended Reals and Intervals

The formalization presented here uses extended real numbers $\mathbb{R}^\infty$ that extends the set of real numbers with infinite numbers $-\infty$ and ∞. There is no built-in support for extended reals in PVS. Hence, these numbers are presented by a pair (k, b), where k is a real number and b is a Boolean. When b is the Boolean constant `true`, (k, b) represents the real number k. In any other case, (k, b) represents an infinite number. The sign of that infinite number is the sign of the real number k. Using this data structure, the evaluation of a polynomial p at an extended real $r = (k, b)$ is defined as the extended real (k', b), where

$$k' \equiv \begin{cases} p(k) & \text{if } b = \texttt{true}, \\ \texttt{lc}(p) & \text{if } b = \texttt{false} \ \wedge \ \text{even}(\texttt{deg}(p)), \\ \text{sign}(k) \cdot \texttt{lc}(p) & \text{otherwise.} \end{cases}$$

In addition to polynomial evaluation on extended reals, the real order relations are extended to $\mathbb{R}^\infty$ such that

- $(k_1, \texttt{true}) < (k_2, b)$, when $k_1 < k_2$ or $b = \texttt{false}$.
- $(k_1, \texttt{false}) < (k_2, b)$, when $b = \texttt{true}$ or $\text{sign}(k_1) < \text{sign}(k_2)$.
- Infinite numbers are incomparable for equality.

No other operations are defined on infinite extended real numbers.

Extended intervals are represented by a 4-tuple (l, u, c_l, c_u), where l, u are extended real numbers and c_l, c_b are Boolean values. The extended real numbers l and u represent the lower and upper bounds of the interval, respectively. The Boolean values c_l and c_u indicate whether or not the lower and upper bounds, respectively, are included in the interval. It is assumed that intervals are nonempty and contain more than one element, i.e., $l < u$. Furthermore, to simplify the manipulation of intervals, they are constructed such that the following properties are satisfied.

- If l and u are both infinite numbers, then $l = (-1, \texttt{false})$, $u = (1, \texttt{false})$, and $c_l = c_u = \texttt{false}$.
- If $l = (k_l, \texttt{false})$ and $u = (k_u, \texttt{true})$, then $k_l = -|k_u - 1|$, and $c_l = \texttt{false}$.
- If $l = (k_l, \texttt{true})$ and $u = (k_u, \texttt{false})$, then $k_u = |k_u + 1|$, and $c_u = \texttt{false}$.

For instance, the PVS functions `choose`(I) that returns a real number k that is strictly included in I is defined as follows. If I is the interval (l, u, c_l, c_u), where $l = (k_l, b_l)$ and $u = (k_u, b_u)$, then $k = \frac{k_l + k_u}{2}$. It is easy to see that the extended, but finite, real number represented by the pair $(k, \texttt{true})$ satisfies the strict inequalities $l < (k, \texttt{true}) < u$.

7.2 Computing the Decomposition of a Rational Number

In Section 4.2, a function is described which takes a polynomial with rational coefficients and returns a multiple of the polynomial with integer coefficients

by multiplying by the product of the denominators of all of the coefficients. To do so, a function must be defined that computes the denominator of a given rational number. In PVS, where rational numbers are a primitive type, defining such a function is not as simple as it appears. This is mainly due to the fact that in the PVS specification language, different representations of the same rational number, e.g., $\frac{1}{2}$, $\frac{2}{4}$, and 0.5, are indistinguishable.

There are several ways in which a function that computes the numerator and denominator of a rational number can be defined in PVS. The PVS function that computes the rational decomposition is a recursive function called `compute_pos_rat`, which in essence computes the continued fraction representation of the rational number to recover the numerator and denominator. The function takes a positive rational number and returns a pair of positive integers, the first being the numerator and the second being the denominator.

$$\begin{aligned}
&\texttt{compute_pos_rat}(r) \equiv \\
&\quad \texttt{if } \texttt{floor}(r) = r \texttt{ then } (\texttt{floor}(r), 1) \\
&\quad \texttt{elsif } r > 1 \texttt{ then let } (a, b) = \texttt{compute_pos_rat}(r - \texttt{floor}(r)) \texttt{ in} \\
&\qquad (b \cdot \texttt{floor}(r) + a, b) \\
&\quad \texttt{else let } (a, b) = \texttt{compute_pos_rat}(1/r) \texttt{ in } (b, a) \texttt{ endif}.
\end{aligned} \tag{32}$$

The function `compute_pos_rat` is then be used to define two other functions, i.e., `numerator` and `denominator`, which are simply given by the first and second component of the output of `compute_pos_rat`. It has been verified in PVS that $r = \texttt{numerator}(r)/\texttt{denominator}(r)$, for all positive rational number r.

A challenging part of proving the correctness of `compute_pos_rat` is showing that the function terminates. In PVS, showing termination requires that the function be given a measure, which is a function on the inputs of the function that returns a natural number and strictly decreases in value every time the function is called recursively. In PVS, the measure function of the recursive function `compute_pos_rat` is defined by

$$\texttt{pos\_rat\_meas}(r) \;\equiv\; \texttt{if } r < 1 \texttt{ then } 10^g \texttt{ else } 10^g - 1 \texttt{ endif}, \tag{33}$$

where r is a rational number and g is the least integer such that there exists positive rational numbers a and b with $r = a/b$ and $g = a + b$.

7.3 Matrices

The PVS library for linear algebra presented in [22] does not include a computable formalization of invertibility of matrices. In particular, using that library, an inverse could not be computed, Gauss-Jordan elimination was not formalized, and the theorem that the determinant of a matrix is nonzero if and only if the matrix is invertible was not proved. All of these results were recently added to the NASA PVS Library in direct support of the formalization of Tarski's theorem. In fact, a new library called `matrices` was developed by the authors with a definition of matrices that is designed for computation.

In this new PVS development, matrices are defined with lists of lists of numbers, e.g., the $n \times m$-matrix

$$\begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix}$$

is represented by the list $((a_{11}, \dots, a_{1m}) \dots (a_{n1}, \dots, a_{nm}))$, of length n, of lists of length m.

The library includes formalization and proof of many of the main theorems related to matrices, determinants, and inverses. In addition, the particular application to Tarski's theorem required the specification and proof of many properties concerning the tensor product and tensor power of matrices. Chief among these were the fact that the tensor and inverse operations commute for nonsingular matrices, and the specification of a function that calculates an entry of a tensor power without calculating the entire tensor power. As alluded to in Section 5.1, this involved the manipulation of numbers in several different bases, which was at times tedious.

The development of the matrices library provided several examples of the subtleties of formalizing mathematics. To understand why this is the case, consider the following basic facts about matrices, which have omitted some of the obvious hypotheses that they require.

1. $\det(A \cdot B) = \det(A) \cdot \det(B)$.
2. $\det(A) \neq 0$ if and only if A is invertible.
3. A can be translated to upper triangular form by Gauss-Jordan elimination.
4. If $\det(A) \neq 0$, then A is a product of elementary matrices.
5. If $\det(A) \neq 0$, then the inverse of A can be computed by performing Gauss-Jordan elimination on the matrix $(A\ I)$.

When doing mathematics on a blackboard, it is relatively easy to prove each of these properties. In fact, one might even say that Gauss-Jordan elimination is equivalent to multiplying by a sequence of elementary matrices, and the first four properties follow directly from that. The final property might require a little more work to prove, but it would follow directly from the other four. Now, loosely speaking, the five properties above have almost identical proofs. In each case, there is a property P on matrices, and a specific matrix A is considered that can be written as the result of a finite number n of operations on the identity matrix. These operations are multiplications by certain elementary matrices, although in the proofs they may be expressed simply as Gauss-Jordan operations. The proof of each of these properties is by induction on n. Each time n increases by 1, this corresponds to multiplying by one more elementary matrix, and it is shown in the proof that this does not affect the property P.

One way to formalize each of these five properties is individually. This requires this induction to be carried out five times. If done in the most naive

way, this would be quite inefficient, since it would result in formalizing Gauss-Jordan elimination multiple times. One key motivation in formalizing mathematics is not doing more work than necessary. Thus, in the PVS formalization, these 5 properties are proved simultaneously, so that the induction step only happens once. What this means is that there is a single formalization of the Gauss-Jordan process that is computable. In addition, the five properties above are encoded into *types* of the objects that are returned by the functions performing the Gauss-Jordan process. The Gauss-Jordan process, and inverse computation, occurs in three steps, each of which is performed by a single PVS function:

1. Translate the original matrix A to upper triangular form with elementary row operations. Return the resulting upper triangular matrix, as well as the matrix representing this sequence of operations and its inverse.
2. Translate the upper triangular matrix to a diagonal matrix if its determinant is nonzero, leaving any column alone where the matrix has a 0 on the diagonal. Again, return the resulting matrix, as well as the matrix representing the operations and its inverse.
3. An function which takes the resulting diagonal matrix, when its determinant is nonzero, and computes the inverse of the diagonal matrix. Save the matrix representing these operations, as well as its inverse.

The inverse of the original matrix A can easily be computed using these three functions. All that one must do is return the product Q^* of the matrices returned by these three functions, which correspond to the Gauss-Jordan operations. When A has nonzero determinant, Q^* will then trivially have the property that $Q^* \cdot A = I$, and thus A has a left inverse. Since each of the three functions above also computes the inverse of the operations matrix, multiplying each of these in the correct order yields another matrix R^* such that R^* is the product of elementary matrices and $Q^* \cdot R^* = I$. Since this process has already showed that any matrix with a nonzero determinant has a left inverse, this means that Q^* has a left inverse. Thus, it is trivial to conclude that $A = R^*$, making A the product of elementary matrices.

For most of the specifics on how these functions are defined in PVS, the readers are referred to the PVS specifications. However, the first of these functions, which translates a matrix into upper triangular form through elementary row operations, is described below. This should help the reader to understand how the five matrix properties at the beginning of this section are encoded into the return types of the functions, requiring only one induction scheme in the correctness proof. The function `upper_triang` in PVS performs this upper triangulation process, and a slightly simplified version of it is described here. The actual version has flags that allow the matrices of operations to not be computed, making the function faster to evaluate. There are six inputs to `upper_triang`, which conditions on their *types*:

– A positive integer n.
– The original n by n matrix A.

- An n by n matrix Q which is the product of finitely many elementary matrices.
- An n by n matrix R which is the product of finitely many elementary matrices, such that $Q \cdot R = I$ and $R \cdot Q = I$.
- The pivot column j.
- The pivot row i.

The function `upper_triang` is recursive on the pivots j and i. It returns three matrices, and the types of matrices are specified by the return type of the function:

- An upper triangular n by n matrix A^* such that $\det(A^*) = \det(A)$.
- An n by n matrix Q^*, which is the product of finitely many elementary matrices, such that $Q^* \cdot A = A^*$.
- An n by n matrix R^*, which is the product of finitely many elementary matrices, such that $Q^* \cdot R^* = I$ and $R^* \cdot Q^* = I$.

Specifying these conditions on the outputs of the functions performing Gauss-Jordan elimination makes the final proofs of the main properties of determinants and invertibility quite easy, once the correctness of output types of these Gauss-Jordan functions has been verified.

8 Related Work

The authors are aware of the following bodies of work that are closely related to the formalization presented in this paper. Harrison proved Sturm's theorem in HOL and used it there for root isolation [20]. The intended application of that work was to guarantee error bounds of polynomial approximations to transcendental functions. Root isolation was used on the derivative of the error to find the places where an error may be maximized. In that work, Sturm's theorem was proved in the case where the polynomial is square free, which simplifies the proof. In that paper, Harrison writes "[...] we would prefer the polynomial to have no multiple real roots [...] Sturm's theorem is easier to prove for polynomials without multiple real roots - this is actually the only form we have proved in HOL." Harrison was not interested in proving statements like $p(x) \geq 0$, so a decision procedure for such problems was out of the scope of his paper. However, finding the square free part of p could be used as a preprocessing step, before root counting, since p and it square free part have the same roots. This would allow Harrison's proof to be used to verify a decision procedure for such statements.

Cohen and Mahboubi implemented a decision procedure in Coq for first order formulas over real closed fields that uses Tarski's theorem as a central result [4]. Their satisfiability theorem is more general than the one presented in this paper since it also covers the case of quantifier elimination for multivariate polynomials. However, the authors consider the developed procedure as primarily a theoretical result and thus make no attempt to automate the

procedure for use as a proof tactic. Indeed, the authors note that the "[...] procedure is formally proved correct and complete, but is totally ineffective for the time being." In contrast, one of the central goals of the current PVS implementation was to provide users a complete and correct, formally verified, but also *computationally feasible* automated procedure for deciding concrete instances of these problems in the real expressions of PVS.

The recent work by Eberl [12] is the most similar to this paper. It was apparently being completed concurrently with this work. Eberl completed a formal proof in Isabelle/HOL of Sturm's Theorem and used it to define proof methods `sturm` in Isabelle/HOL for solving polynomial relations similar to those solved by PVS strategies `sturm` and `mono-poly`. In the case of non-strict universally-quantified inequalities, Eberl's proof method relies on unverified ML code to generate the interval splitting as a witness. In practice, the use of ML code improves efficiency and does not compromise soundness. However, it may compromise completeness. In contrast, the PVS strategies presented in this paper rely on PVS algorithms that are proven to be correct and complete. Another distinction between the current paper and the work of Eberl is in the use of pseudo division instead of regular division. A point worth noting regarding the differences between the work in this paper and that by Eberl is that the proofs of Sturm's theorem are different. Eberl proves Sturm's theorem in the square free case, similar to Harrison's proof in HOL [20]. In the non-square free case, the proof proceeds by dividing each term in the remainder sequence by the greatest common divisor of the original polynomial and its derivative. The resulting sequence is not a Sturm sequence in the standard sense, but it maintains similar properties regarding root counting. The proof of Sturm's theorem in the PVS development presented in this paper follows a direct approach that considers the highest power of a linear divisor that divides each polynomial in the sequence and analyzes whether the polynomial swaps signs at the corresponding root. The distinctions between these proof methods primarily amount to user preference. Finally, the development presented in [12] does not consider the more general Tarski's theorem.

Sturm sequences and several speed enhancements such as the use of pseudo division are implemented in the SMT solver Z3 [31]. That implementation was the inspiration for the work presented in this paper. Z3 is a highly efficient tool. In some cases, for example when a formula is satisfiable, Z3 can produce models, which can be understood as proof certificates for existential formulas. However, in general, Z3 statements are not supported by formal proofs. Hence, in formal verifications efforts, Z3 is used as a trusted oracle.

Z3 is used as an external algebraic decision method (EADM) in Metitarski, a theorem prover for real numbers [1]. In a recent work, Denman and Muñoz [9] developed the PVS proof rule `metit` that integrates Metitarski as an external oracle into PVS theorem prover. In contrast to the work presented in this paper, `metit` is not implemented as a proof producing strategy. Nevertheless, the integration of Metitarski/Z3 in PVS, while unproven, is quite useful and has helped the authors to check results that were impossible with previous strategies in PVS.

There is a much larger collection of works on the general problem of reasoning about nonlinear arithmetic. Sophisticated implementations of the *Cylindrical Algebraic Decomposition* (CAD) [5] procedure are available in the Redlog system[6] and in the QEPCAD library.[7] The systems RealPaver [18] and dReal [14] integrate powerful methods based on interval constraint propagation [18]. MetiTarski [1] and RAHD (Real Algebra in High Dimensions) [37] are specialized theorem provers for the theory of real closed fields. MetiTarski is designed to prove universally quantified inequalities involving real-valued functions such as trascendental functions. RAHD combines several decision methods for the existential theory of real closed fields. Both systems use quantifier elimination procedures among many other proof strategies. As indicated by Section 2, Tarski's theorem is a generalization of Sturm's theorem, and Tarski's original proof was based on Sturm's theorem [42].

Proof tactics that implement Hörmander's quantifier elimination method are available in Coq and HOL Light [26,28]. These tactics are theoretically interesting and still practical for some type of problems. However, Hörmander's method is known to be more inefficient than CAD. An implementation of CAD that will eventually yield a proof producing tactic is available in Coq [25]. Another approach to solve multivariate polynomial inequalities in theorem provers is based on polynomial sum of square (SOS) decompositions through semidefinite programming. Such an approach has been implemented in HOL Light [21] and seems to be more promising than quantifier elimination for polynomials with many variables. Semidefinite programming is a somewhat complicated numerical procedure that is usually implemented with floating point numbers. Because of numerical approximation errors, it is difficult to integrate this method into theorem provers. Recent developments in SOS address this issue by producing rational polynomial decompositions [23,30]. Proof producing strategies for proving real-number properties based on interval arithmetic and branch and bound methods are available in PVS [7], Coq [29], HOL Light [39]. As stated in the introduction, the authors have developed semi-decision procedures for multivariate polynomials, based on Bernstein polynomials [33], and Boolean expressions involving real-valued functions, based on interval arithmetic [34]. Those algorithms are quite powerful and can prove tight bounds on complex polynomials with up to 16 variables and degree 4. However, techniques based on Bernstein polynomials and interval arithmetic cannot always prove exact bounds on a polynomial. That is, in general, they cannot prove that $p(x) \geq 0$ for x in a fixed, bounded interval unless it is also true that $p(x) > c$ for some positive c. This limitation is a key motivation to the authors' development presented in this paper.

The formal development presented in this paper includes a formalization in PVS of linear algebra concepts such as matrices and tensors that are needed for the definition of a decision procedure based on Tarski's theorem. As discussed in Section 7.3, this development improves over the work presented in [22] by

[6] `http://redlog.dolzmann.de`.

[7] `http://www.usna.edu/cs/~qepcad/B/QEPCAD.html`.

providing computable definitions of determinants and the Gauss-Jordan elimination algorithm. This work in PVS complements existing work on computable formalizations of linear algebra in other theorem provers [2, 8, 17, 27].

9 Conclusion

This paper presents formalizations of Sturm's and Tarksi's theorems in PVS. These theorems are used to develop decision procedures for deciding the satisfiability of univariate rational polynomial systems where the polynomial variable ranges over an interval, which can be any connected set of real numbers. The decision procedures, which are proven to be complete and correct in PVS, are used to implement several proof strategies for automatically discharging sequents involving polynomial expressions. The correctness of these strategies depend only on the PVS deduction engine, as opposed to an external oracle. The strategies are based on computational reflection, which is a theorem proving technique for building efficient strategies. Although the strategies employ data structures for representing and manipulating polynomials and infinite numbers, these data structures are invisible to the user. The strategies can be used to discharge proof sequents involving *native* PVS real number expressions.

The authors's main motivation for developing these tools for polynomial computation in PVS stems from NASA's ongoing verification efforts for aircraft separation assurance systems.[8] The PVS proofs of correctness of these systems are decidedly nontrivial and require considerable algebraic manipulation. In one particular case of an actual verification effort, a 176-line sequent involving a 16-variable polynomial was generated. That sequent was automatically checked using a PVS proof rule that integrates Metitarski and Z3 as trusted external oracles into the PVS theorem prover [9]. This example clearly illustrates the usefulness of techniques based on Sturm sequences and Tarski queries, which are implemented in Z3. Unfortunately, because the proof described above relies on external tools, it cannot be turned into a proof that relies solely on the internal logic of PVS. The ultimate goal of the work presented here is to provide such functionality directly in PVS with a formal proof of its correctness. The work with univariate polynomials presented in this paper is a first step in that direction. The next step is to define algorithms that not only handle multivariate polynomials, but also handle arbitrary Boolean expressions involving those polynomials. This work will be guided by the algorithms developed by de Moura in Z3, but the authors expect subtleties to arise, since they will have to be designed with formal verification in mind. That is, having to prove their correctness formally may change the way that these algorithms are specified in PVS.

Acknowledgements The authors are grateful to the anonymous reviewers for their insightful comments that greatly helped to improve this manuscript. Special thanks to Manuel

[8] http://shemesh.larc.nasa.gov/fm/fm-atm-cdr.html

Eberl for pointing out missprints in an earlier version of this document and for providing access to his formal development of Sturm's theorem in Isabelle/HOL. Last but not least, the authors would like to give credit to Leonardo de Moura for directing their work in the direction of Sturm's and Tarski's theorems and advising them in this effort.

References

1. Akbarpour, B., Paulson, L.C.: MetiTarski: An automatic theorem prover for real-valued special functions. Journal of Automated Reasoning 44(3), 175–205 (2010)
2. Aransay, J., Divasón, J.: Formalization and execution of linear algebra: from theorems to algorithms. In: Gupta, G., Peña, R. (eds.) Proceedings, 23rd International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2013, Madrid, Spain. Dpto. de Systemas Informáticos y Computation, Universidad Complutense de Madrid, TR-11-13 (Sep 2013)
3. Basu, S., Pollack, R., Roy, M.F.: Algorithms in Real Algebraic Geometry (Algorithms and Computation in Mathematics). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)
4. Cohen, C., Mahboubi, A.: Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. Logical Methods in Computer Science 8(1:02), 1–40 (Feb 2012), https://hal.inria.fr/inria-00593738
5. Collins, G.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Second GI Conference on Automata Theory and Formal Languages. Lecture Notes in Computer Science, vol. 33, pp. 134–183. Springer-Verlag, Kaiserslautern (1975)
6. Crespo, L.G., Muñoz, C.A., Narkawicz, A.J., Kenny, S.P., Giesy, D.P.: Uncertainty analysis via failure domain characterization: Polynomial requirement functions. In: Proceedings of European Safety and Reliability Conference. Troyes, France (September 2011)
7. Daumas, M., Lester, D., Muñoz, C.: Verified real number calculations: A library for interval arithmetic. IEEE Transactions on Computers 58(2), 1–12 (February 2009)
8. Dénès, M., Mörtberg, A., Siles, V.: A refinement-based approach to computational algebra in Coq. In: Beringer, L., Felty, A.P. (eds.) Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7406, pp. 83–98. Springer (2012), http://dx.doi.org/10.1007/978-3-642-32347-8
9. Denman, W., Muñoz, C.: Automated real proving in PVS via MetiTarski. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) Proceedings of the 19th International Symposium on Formal Methods (FM 2014). Lecture Notes in Computer Science, vol. 8442, pp. 194–199. Springer, Singapore (May 2014)
10. de Dinechin, F., Lauter, C., Melquiond, G.: Certifying the floating-point implementation of an elementary function using Gappa. IEEE Transactions on Computers 60(2), 242–253 (February 2011)
11. Dowek, G., Geser, A., Muñoz, C.: Tactical conflict detection and resolution in a 3-D airspace. In: Proceedings of the 4th USA/Europe Air Traffic Management R&D Seminar, ATM 2001. Santa Fe, New Mexico (2001), a long version appears as report NASA/CR-2001-210853 ICASE Report No. 2001-7
12. Eberl, M.: A decision procedure for univariate real polynomials in Isabelle/Hol. In: Proceedings of the 2015 Conference on Certified Programs and Proofs. pp. 75–83. CPP '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2676724.2693166
13. Eisermann, M.: The fundamental theorem of algebra made effective: An elementary real-algebraic proof via Sturm chains. The American Mathematical Monthly 119(9), 715–752 (November 2012)
14. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT solver for nonlinear theories over the reals. In: Bonacina, M.P. (ed.) Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7898, pp. 208–214. Springer (2013), http://dx.doi.org/10.1007/978-3-642-38574-2

15. Garloff, J.: Application of Bernstein expansion to the solution of control problems. Reliable Computing 6, 303–320 (2000)
16. von zur Gathen, J., Lücking, T.: Subresultants revisited. Theor. Comput. Sci. 297(1-3), 199–239 (Mar 2003), http://dx.doi.org/10.1016/S0304-3975(02)00639-4
17. Gonthier, G.: Point-free, set-free concrete linear algebra. In: van Eekelen, M.C.J.D., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) Interactive Theorem Proving - ITP 2011. vol. 6898, pp. 103–118. Radboud University of Nijmegen, Springer, Berg en Dal, Netherlands (Aug 2011), https://hal.inria.fr/hal-00805966
18. Granvilliers, L., Benhamou, F.: RealPaver: An interval solver using constraint satisfaction techniques. ACM Transactions on Mathematical Software 32(1), 138–156 (March 2006)
19. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK (1995)
20. Harrison, J.: Verifying the accuracy of polynomial approximations in HOL. In: Gunter, E.L., Felty, A. (eds.) Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs'97. Lecture Notes in Computer Science, vol. 1275, pp. 137–152. Springer-Verlag, Murray Hill, NJ (1997)
21. Harrison, J.: Verifying nonlinear real formulas via sums of squares. In: Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science, vol. 4732, pp. 102–118. Springer (2007)
22. Herencia-Zapana, H., Jobredeaux, R., Owre, S., Garoche, P.L., Feron, E., Perez, G., Ascariz, P.: PVS linear algebra libraries for verification of control software algorithms in C/ACSL. In: Goodloe, A., Person, S. (eds.) NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7226, pp. 147–161. Springer (2012), http://dx.doi.org/10.1007/978-3-642-28891-3
23. Kaltofen, E.L., Li, B., Yang, Z., Zhi, L.: Exact certification in global polynomial optimization via sums-of-squares of rational functions with rational coefficients. In: Robbiano, L., Abbott, J. (eds.) Approximate Commutative Algebra. Texts and Monographs in Symbolic Computation, Springer Vienna (2010)
24. Kuchar, J., Yang, L.: A review of conflict detection and resolution modeling methods. IEEE Transactions on Intelligent Transportation Systems 1(4), 179–189 (December 2000)
25. Mahboubi, A.: Implementing the cylindrical algebraic decomposition within the Coq system. Mathematical Structures in Computer Science 17(1), 99–127 (February 2007)
26. Mahboubi, A., Pottier, L.: Elimination des quantificateurs sur les réels en Coq. In: Journées Francophone des Langages Applicatifs (JFLA) (2002)
27. Mahmoud, M.Y., Aravantinos, V., Tahar, S.: Formalization of infinite dimension linear spaces with application to quantum theory. In: Brat, G., Rungta, N., Venet, A. (eds.) NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7871, pp. 413–427. Springer (2013), http://dx.doi.org/10.1007/978-3-642-38088-4
28. McLaughlin, S., Harrison, J.: A proof-producing decision procedure for real arithmetic. In: Nieuwenhuis, R. (ed.) Proceedings of the 20th International Conference on Automated Deduction, proceedings. Lecture Notes in Computer Science, vol. 3632, pp. 295–314 (2005)
29. Melquiond, G.: Proving bounds on real-valued functions with computations. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings. Lecture Notes in Computer Science, vol. 5195, pp. 2–17. Springer (2008), http://dx.doi.org/10.1007/978-3-540-71070-7_2
30. Monniaux, D., Corbineau, P.: On the generation of Positivstellensatz witnesses in degenerate cases. In: Proceedings of Interactive Theorem Proving (ITP). Lecture Notes in Computer Science (2011)
31. de Moura, L., Passmore, G.: Computation in real closed infinitesimal and transcendental extensions of the rationals. In: Automated Deduction - CADE-24, 24th International Conference on Automated Deduction, Lake Placid, New York, June 9-14, 2013, Proceedings (2013)

32. Muñoz, C.: Rapid prototyping in PVS. Contractor Report NASA/CR-2003-212418, NASA, Langley Research Center, Hampton VA 23681-2199, USA (May 2003)
33. Muñoz, C., Narkawicz, A.: Formalization of a representation of Bernstein polynomials and applications to global optimization. Journal of Automated Reasoning 51(2), 151–196 (August 2013), http://dx.doi.org/10.1007/s10817-012-9256-3
34. Narkawicz, A., Muñoz, C.: A formally verified generic branching algorithm for global optimization. In: Cohen, E., Rybalchenko, A. (eds.) Fifth Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE 2013). Lecture Notes in Computer Science, vol. 8164, pp. 326–343. Springer (2014)
35. Narkawicz, A.J., Muñoz, C.A.: A formally-verified decision procedure for univariate polynomial computation based on Sturm's theorem. Technical Memorandum NASA/TM-2014-218548, NASA, Langley Research Center, Hampton VA 23681-2199, USA (November 2014)
36. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) Proceeding of the 11th International Conference on Automated Deduction (CADE). Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Springer (June 1992)
37. Passmore, G.O., Jackson, P.B.: Combined decision techniques for the existential theory of the reals. In: Dixon, L. (ed.) Proceedings of Calculemus/Mathematical Knowledge Management. pp. 122–137. No. 5625 in LNAI, Springer-Verlag (2009)
38. Shankar, N.: Efficiently executing PVS. Tech. rep., Project Report, ComputerScience Laboratory, SRI International, Menlo Park (1999)
39. Solovyev, A., Hales, T.C.: Formal verification of nonlinear inequalities with Taylor interval approximations. In: Brat, G., Rungta, N., Venet, A. (eds.) Proceedings of the 5th International Symposium NASA Formal Methods. Lecture Notes in Computer Science, vol. 7871, pp. 383–397 (2013)
40. Sottile, F.: Chapter 2: Real solutions to univariate polynomials, http://www.math.tamu.edu/~sottile/teaching/10.S/Ch2.pdf, course Notes
41. Sturm, C.: Mémoire sur la résolution des équations numériques. In: Pont, J.C. (ed.) Collected Works of Charles François Sturm, pp. 345–390. Birkhäuser Basel (2009), http://dx.doi.org/10.1007/978-3-7643-7990-2_29
42. Tarski, A.: A decision method for elementary algebra and geometry. Bulletin of the American Mathematical Society 59 (1951)

A Map of Theorems to Formal Theorems in PVS

Theorem	PVS Theory	PVS Theorems
Theorem 1	Sturm@sturm	sturm, sturm_unbounded_left, sturm_unbounded_right, sturm_unbounded
Theorem 2	Tarski@sturmtarski	sturm_tarski_unbounded
Theorem 3	Sturm@compute_sturm	roots_closed_int_def
Theorem 4	Sturm@compute_sturm	number_roots_interval_def
Theorem 5	reals@more_polynomial_props	Knuth_poly_root_strict_bound
Theorem 6	Sturm@compute_sturm	always_nonnegative_def
Theorem 7	Sturm@compute_sturm	compute_poly_sat_def
Theorem 8	Sturm@poly_strategy	sturm_def
Theorem 9	Sturm@compute_sturm	mono_def
Theorem 10	Tarski@tarski_query_matrix	multi_sturm_tarski_6by6
Theorem 11	Tarski@tarski_query_matrix	multi_sturm_tarski_NSol
Theorem 12	Tarski@tarski_query_matrix	multi_sturm_tarski_NSol63
Theorem 13	Tarski@poly_system_strategy	compute_solvable_single_def
Theorem 14	Tarski@poly_system_strategy	compute_solvable_def
Theorem 15	Tarski@poly_system_strategy	tarski_def
Theorem 16	Sturm@polylist	polylist_const, polylist_monom, polylist_sum, polylist_minus, polylist_neg, polylist_prod, polylist_scal, polylist_div, polylist_pow

B PVS Examples

```
examples : THEORY
BEGIN

  IMPORTING Tarski@strategies

  x,y : VAR real

  example_1a : LEMMA
    x ## [|0,3|] IMPLIES x^120 -2*x^60 + 1 >= 0
%|- example_1a : PROOF
%|- (sturm)
%|- QED

  example_1b : LEMMA
    EXISTS (x:real): x ## [|0,3|] AND x^120 -2*x^60 + 1 = 0
%|- example_1b : PROOF
%|- (sturm)
%|- QED

  example_2a : LEMMA
    x ## [|-oo,open(3)|] IMPLIES
    x^120 - (2/3)*x^60 + 1/9 >= 0
```

```
%|- example_2a : PROOF
%|- (sturm)
%|- QED

  example_2b : LEMMA
    EXISTS (x:real) : x ## [|-oo,open(3)|] AND
                      x^120 - (2/3)*x^60 + 1/9 >= 0
%|- example_2b : PROOF
%|- (sturm)
%|- QED

  legendre_2(x:real) : MACRO real =
    (3*x^2-1)/2

  legendre_3(x:real) : MACRO real =
    (5*x^3-3*x)/2

  legendre_4(x:real) : MACRO real =
    (35*x^4-30*x^2+3)/8

  legendre_5(x:real) : MACRO real =
    (63*x^5-70*x^3+15*x)/8

  legendre_6(x:real) : MACRO real =
    (231*x^6-315*x^4+105*x^2-5)/16

  legendre_8(x:real) : MACRO real =
    (6435*x^8 - 12012*x^6 + 6930*x^4 - 1260*x^2 + 35)/128

  legendre_9(x:real) : MACRO real =
    (12155*x^9 - 25740*x^7 + 18018*x^5 - 4620*x^3 + 315*x)/128

  legendre_10(x:real) : MACRO real =
    (46189*x^10 -109395*x^8 + 90090*x^6 - 30030*x^4 + 3465*x^2 - 63)/256

  Turan_9: LEMMA
    abs(x) < 1 IMPLIES
    legendre_9(x)^2 > legendre_8(x)*legendre_10(x)
%|- Turan_9 : PROOF
%|- (sturm)
%|- QED

  Legendre_10 : LEMMA
    x ## [|-0.75,-0.6|] AND y ## [|-0.75,-0.6|] AND x < y IMPLIES
    legendre_10(x) > legendre_10(y)
%|- Legendre_10 : PROOF
%|- (mono-poly)
%|- QED

  Legendre_2_6 : LEMMA
    legendre_2(x) >= 0 OR legendre_3(x) >= 0 OR
    legendre_4(x) >= 0 OR legendre_5(x) >= 0 OR
    legendre_6(x) >= 0
%|- Legendre_2_6 : PROOF
%|- (tarski)
%|- QED

  IMPORTING vectors@vectors_2D
```

```
  so : Vect2 = (-37040,0) % [m]
  soz: real  = 9144       % [m]^2
  vo : Vect2 = (218,218)  % [m/s]^2
  voz: real  = 2.54       % [m/s]
  si : Vect2 = (37000,0)  % [m]^2
  siz: real  = 9000       % [m]
  vi : Vect2 = (-205,205) % [m/s]^2
  viz: real  = 2.5        % [m/s]

  s  : Vect2 = so-si
  sz : real  = soz-siz
  v  : Vect2 = vo-vi
  vz : real  = voz-viz

  D : real = 9260 % [m]
  H : real = 305  % [m]
  T : real = 300  % [s]

  conflict?(s,v:Vect2,T:real) : MACRO bool =
    EXISTS (t:real | t ## [|0,T|]) : (s*s)+(2*s*v)*t+(v*v)*t^2 < sq(D) AND
                                     abs(sz+t*vz) < H

  yes_conflict : LEMMA
    conflict?(so-si,vo-vi,300)
%|- yes_conflict : PROOF
%|- (tarski)
%|- QED

  no_conflict : LEMMA
    NOT conflict?(so-si,vo-vi,150)
%|- no_conflict : PROOF
%|- (tarski)
%|- QED

END examples
```