Solving Non-Linear Arithmetic

Dejan Jovanović[1] and Leonardo de Moura[2]

[1] New York University
[2] Microsoft Research

Abstract. We present a new algorithm for deciding satisfiability of non-linear arithmetic constraints. The algorithm performs a Conflict-Driven Clause Learning (CDCL)-style search for a feasible assignment, while using projection operators adapted from cylindrical algebraic decomposition to guide the search away from the conflicting states.

1 Introduction

From the early beginnings in Persian and Chinese mathematics [24,25,50] until the present day, polynomial constraints and the algorithmic ways of solving them have been one of the driving forces in the development of mathematics. Though studied for centuries due to the natural elegance they provide in modeling the real world, from resolving simple taxation arguments to modeling planes and hybrid systems, we are still lacking a practical algorithm for solving a system of polynomial constraints. Throughout the history of mathematics, many brilliant minds have studied and algorithmically solved many of the related problems, such as root finding [15,48,43] and factorization of polynomials [36,19,20]. But, it was not until Alfred Tarski [44,45,47] showed that the theory of real closed fields admits elimination of quantifiers that it became clear that a general decision procedure for solving polynomial constraints was possible. Granted a wonderful theoretical result of landmark importance, with its non-elementary complexity, Tarski's procedure was unfortunately totally impractical.

As one would expect, Tarski's procedure consequently has been much improved. Most notably, Collins [11] gave the first relatively effective method of quantifier elimination by cylindrical algebraic decomposition (CAD). The CAD procedure itself has gone through many revisions [31,22,12,5]. However, even with the improvements and various heuristics, its doubly-exponential worst-case behavior has remained as a serious impediment. The CAD algorithm works by decomposing $\mathbb{R}^k$ into connected components such that, in each cell, all of the polynomials from the problem are sign-invariant. To be able to perform such a particular decomposition, CAD first performs a *projection* of the polynomials from the initial problem. This projection includes many new polynomials, derived from the initial ones, and these polynomials carry enough information to ensure that the decomposition is indeed possible. Unfortunately, the size of these projections sets grows exponentially in the number of variables, causing the projection phase to be a key hurdle to CAD scalability.

We propose a new decision procedure for the existential theory of the reals that tries to alleviate the above problem. In the spirit of our recent decision procedure for linear integer arithmetic [26], the new procedure performs a backtracking search for a model in $\mathbb{R}$, where the backtracking is powered by a novel conflict resolution procedure. Our approach takes advantage of the fact that each conflict encountered during the search is based on the current assignment and generally involves only a few constraints, a *conflicting core*. When in conflict, we project only the polynomials from the conflicting core and explain the conflict in terms of the current model. This means that we use projection conservatively, only for the subsets of polynomials that are involved in the conflict, and even then we reduce it further. As another advantage, the conflict resolution provides the usual benefits of a Conflict-Driven Clause Learning (CDCL)-style [40,35] search engine, such as non-chronological backtracking and the ability to ignore irrelevant parts of the search space. The projection operators we use as part of the conflict resolution need not be CAD based and, in fact, one can easily adapt projections based on other algorithms (e.g [32,3]).

Due to the lack of space and the volume of algorithms and concepts involved, we concentrate on the details of the decision procedure in this paper and refer the reader to the existing literature for further information [8,9,10,27,3]. Acknowledging the importance that the details of a particular implementation play, we do include an appendix with the description of particular algorithms we chose for our implementation.

2 Preliminaries

As usual, we denote the ring of integers with $\mathbb{Z}$, the field of rational numbers with $\mathbb{Q}$, and the field of real numbers as $\mathbb{R}$. Unless stated otherwise, we assume all polynomials take integer coefficients, i.e. a polynomial $f \in \mathbb{Z}[\boldsymbol{y}, x]$ is of the form

$$f(\boldsymbol{y}, x) = a_m \cdot x^{d_m} + a_{m-1} \cdot x^{d_{m-1}} + \cdots + a_1 \cdot x^{d_1} + a_0 \enspace ,$$

where $0 < d_1 < \cdots < d_m$, and the coefficients a_i are in $\mathbb{Z}[\boldsymbol{y}]$ with $a_m \neq 0$. We call x the *top variable* and refer to $\boldsymbol{y}$ as variables of lower levels. The highest degree d_m is the *degree* of the polynomial f in variable x, and we denote it with $\mathsf{deg}(f, x)$. The set of coefficients of f is denoted as $\mathsf{coeff}(f, x)$. We call a_m the *leading coefficient* in variable x, and denote it with $\mathsf{lc}(f, x)$. If we exclude the first term of the polynomial f, we obtain the polynomial $\mathsf{R}(f, x) = a_{m-1}x^{d_{m-1}} + \cdots + a_0$, called the *reductum* of f. We denote the set of variables appearing in a polynomial f as $\mathsf{vars}(f)$ and call the polynomial *univariate* if $\mathsf{vars}(f) = \{x\}$ for some variable x. Otherwise the polynomial is *multivariate*, or a constant polynomial (if it contains no variables). Given a set of polynomials $A \subset \mathbb{Z}[x_1, \ldots x_n]$, we denote with A_k the subset of polynomials in A that belong to $\mathbb{Z}[x_1, \ldots, x_k]$, i.e. $A_k = A \cap \mathbb{Z}[x_1, \ldots, x_k]$.[3]

[3] We thus have $A_0 \subseteq A_1 \subseteq \cdots \subseteq A_n$, with A_0 being the constant polynomials of A, and $A_n = A$.

A number $\alpha \in \mathbb{R}$ is a *root of the polynomial* $p \in \mathbb{Z}[x]$ iff $f(\alpha) = 0$. We call a real number $\alpha \in \mathbb{R}$ *algebraic* iff it is a root of a univariate polynomial $f \in \mathbb{Z}[x]$, and we denote the field of all real algebraic numbers by $\mathbb{R}_{\mathsf{alg}}$. We can represent any algebraic number α as $(l, u)_f$, where α is a root of a polynomial f, and the only root in the interval (l, u), with $l, u \in \mathbb{Q}$.

Example 1. Consider the univariate polynomial $f_1 = 16x^3 - 8x^2 + x + 16$. This polynomial has only one root, the irrational number $\alpha_1 \approx -0.840661$ and we can represent it as $(-0.9, -0.8)_{f_1}$.

Given a set of variables $X = \{x_1, \ldots, x_n\}$, we call υ a *variable assignment* if it maps each variable x_k to a real algebraic number $\upsilon(x_k)$, the value of x_k under υ. We overload υ, as usual, to obtain the value of a polynomial $f \in \mathbb{Z}[x_1, \ldots, x_n]$ under υ and write it as $\upsilon(f)$. We say that a polynomial f *vanishes* under υ if $\upsilon(f) = 0$. We can update the assignment υ to map a variable x_k to the value α, and we denote this as $\upsilon[x_k \mapsto \alpha]$.

Under a variable assignment υ that interprets the variables $\boldsymbol{y}$, some coefficients of a polynomial $f(\boldsymbol{y}, x)$ may vanish. If a_k is the first non-vanishing coefficient of f, i.e., $\upsilon(a_k) \neq 0$, we write $\mathsf{R}(f, x, \upsilon) = a_k x^{d_k} + \cdots + a_0$ for the *reductum of f with respect to υ* (the non-vanishing part). Given any sequence of polynomials $\boldsymbol{f} = (f_1, \ldots, f_s)$ and a variable assignment υ we define the *vanishing signature* of $\boldsymbol{f}$ as the sequence $\mathsf{v\text{-}sig}(\boldsymbol{f}, \upsilon) = (f_1, \ldots, f_k)$, where $k \leq s$ is the minimal number such that $\upsilon(f_k) \neq 0$, or s if they all vanish. For the polynomial $f(\boldsymbol{y}, x)$ as above, we define the *vanishing coefficients signature* as $\mathsf{v\text{-}coeff}(f, x, \upsilon) = \mathsf{v\text{-}sig}(a_m, \ldots, a_0)$.

A *basic polynomial constraint* F is a constraint of the form $f \triangledown 0$ where f is a polynomial and $\triangledown \in \{<, \leq, =, \neq, \geq, >\}$. We denote the polynomial constraint that represents the *negation* of a constraint F with $\neg F$.[4] In order to identify the polynomial f of the constraint F, and the variables of F, we write $\mathsf{poly}(F)$ and $\mathsf{vars}(F)$, respectively. We normalize all constraints over constant polynomials to the dedicated constants true and false with the usual semantics. We write $\upsilon(F)$ to denote the evaluation of F under υ, which is the constraint $\upsilon(f) \triangledown 0$. If f does not evaluate to a constant under υ, then $\upsilon(F)$ evaluates to a new polynomial constraint F', where $\mathsf{poly}(F')$ can contain algebraic coefficients.

Borrowing from the extended Tarski language [4, Chapter 7], in addition to the basic constraints, we will also be working with *extended polynomial constraints*. An extended polynomial constraint F is of the form

$$x \triangledown_r \mathsf{root}(f, k) \quad , \tag{1}$$

where $\triangledown_r \in \{<_r, \leq_r, =_r, \neq_r, \geq_r, >_r\}$, f is a polynomial in $\mathbb{Z}[\boldsymbol{y}, \tilde{z}]$, with $x \notin \mathsf{vars}(f)$, and the natural number $k \leq \mathsf{deg}(f, x)$ is the *root index*. Variable $\tilde{z}$ is a distinguished free variable that cannot be used outside the root object. To be able to extract the polynomial of the constraint, we define $\mathsf{poly}(F) = f(\boldsymbol{y}, x)$. Note that, $\mathsf{poly}(F)$ replaces $\tilde{z}$ with x.

[4] For example $\neg(x^2 + 1 > 0) \equiv x^2 + 1 \leq 0$.

The semantics of the predicate (1) under a variable assignment υ is the following. If the polynomial $\upsilon(f)$ is univariate, and υ assigns x to α, the (Boolean) value of the constraint can be determined. If the univariate polynomial $\upsilon(f) \in \mathbb{R}_{\mathsf{alg}}[\tilde{z}]$ has the roots $\beta_1 < \cdots < \beta_n$, with $k \leq n$, and $\alpha \triangledown \beta_k$ holds, then (1) evaluates to true. Otherwise it evaluates to false. Note that the real roots of a polynomial in $\mathbb{R}_{\mathsf{alg}}[\tilde{z}]$ are still algebraic. We denote the number of real roots of a univariate polynomial f as $\mathsf{rootcount}(f)$. Naturally, if F is an extended polynomial constraint, so is the negation $\neg F$.[5]

Example 2. Take the bivariate polynomial $f = 2y\,\tilde{z}^3 - 8\,\tilde{z}^2 + \tilde{z} + 3y - 8$ and the variables assignment υ_1, with $\upsilon_1(x) = -1$ and $\upsilon_1(y) = 8$. Under this assignment we have that $p(y) = 16\,\tilde{z}^3 - 8\,\tilde{z}^2 + \tilde{z} + 16$ and, as in Ex. 1, it has only one root $\alpha = (-0.9, -0.8)_f$. Now consider the constraints

$$x <_r \mathsf{root}(f,1), \quad x \geq_r \mathsf{root}(f,1), \quad \neg(x <_r \mathsf{root}(f,1)), \quad \neg(x <_r \mathsf{root}(f,2)).$$

The value of the above constraints under υ_1 are true, false, false, true, correspondingly. The fourth constraint evaluates to true as f does not have 2 roots, and the predicate thus evaluates to false. Now consider the assignment $\upsilon_2 = \upsilon_1[y \mapsto 0]$. Under υ_2 we have that $p(y) = -8\,\tilde{z}^2 + \tilde{z} - 8$ which now does not have any real roots. The value of the constraints above under υ_2 is therefore false, false, true, true. Note that the second and third constraint might be mistaken to be equal, but are not – as can be seen, they have different semantics.

A *polynomial constraint* is either a basic or an extended one. Given a set of polynomial constraints $\mathcal{F}$, we say that the variable assignment υ satisfies $\mathcal{F}$ if it satisfies each constraint in $\mathcal{F}$. If there is such a variable assignment, we say that $\mathcal{F}$ is *satisfiable*, otherwise it is *unsatisfiable*. A *clause* of polynomial constraints is a disjunction $C = F_1 \vee \ldots \vee F_n$ of polynomial constraints. We use $\mathsf{literals}(C)$ to denote the set $\{F_1, \neg F_1, \ldots, F_n, \neg F_n\}$. We say that the clause C is satisfied under the assignment υ if some polynomial constraint $F_j \in C$ evaluates to true under υ. Finally, a *polynomial constraint problem* is a set of clauses $\mathcal{C}$, and it is satisfiable if there is a variable assignment υ that satisfies all the clauses in $\mathcal{C}$.

3 An Abstract Decision Procedure

We describe our procedure as an abstract transition system in the spirit of Abstract DPLL [37,29]. The crucial difference between the system we present is that we depart from viewing the Boolean search engine and the theory reasoning as two separate entities that communicate only through existing literals. Instead, we allow the model that the theory is trying to construct to be involved in the search and in explaining the conflicts, while allowing new literals to be introduced so as to support more complex conflict analyses. Additionally, our presentation makes the concept of relevancy inherent to procedure (e.g. [13]). The transition

[5] Note that, for example, $\neg(x <_r \mathsf{root}(f,k))$ is not necessarily equivalent to $x \geq_r \mathsf{root}(f,k)$.

system presented here applies to non-linear arithmetic, but it can in general be applied to other theories.

The states in the transition system are indexed pairs of the form $\langle M, \mathcal{C} \rangle_n$, where M is a sequence (usually called a *trail*) of *trail elements*, and $\mathcal{C}$ is a set of clauses. The index n denotes the current *level* of the state. Trail elements can be decided literals, propagated literals, or a variable assignment.

A *decided literal* is a polynomial constraint F that we assume to be true. On the other hand, a *propagated literal*, denoted as $E{\rightarrow}F$, marks a polynomial constraint $F \in E$ that is implied to be true in the current state by the clause E (the *explanation*). In both cases, we say that the constraint F appears in M, and write this as $F \in M$. We denote the set of polynomial constraints appearing in M with $\mathsf{constraints}(M)$. We say M is *non-redundant* if no polynomial constraint appears in M more than once. A *trail variable assignment*, written as $x{\mapsto}\alpha$, is an assignment of a single variable to a value $\alpha \in \mathbb{R}_{\mathsf{alg}}$. Given a trail M, containing variable assignments $x_{i_1} {\mapsto} \alpha_1, \ldots, x_{i_k} {\mapsto} \alpha_k$, in order, we can construct an assignment

$$v[M] = v_0[x_{i_1} \mapsto \alpha_1] \ \ldots \ [x_{i_k} \mapsto \alpha_k] \ ,$$

where v_0 is an empty assignment that does not assign any variables. We say that the sequence M is *increasing in level* when the sequence is of the form

$$M = [\![\overbrace{N_1, x_1{\mapsto}\alpha_1, \ldots, x_{k-1}{\mapsto}\alpha_{k-1}, N_k}^{M_k}, x_k{\mapsto}\alpha_k, \ldots, x_{n-1}{\mapsto}\alpha_{n-1}, N_n]\!] \ ,$$

where, for each level k, the sequence N_k does not contain any variable assignments, each constraint $F \in \mathsf{constraints}(N_k)$ contains the variable x_k, and (optionally) the variables $x_1, \ldots, x_{k-1}$ (and $\tilde{z}$). In such a sequence M, we denote with $\mathsf{level}(M) = n$ the *level* of the sequence, and identify the *subsequence of level* k by writing M_k, as depicted above. Note that M_k *does not* include the assignment of x_k, and in general M_n is different from M if M includes the assignment of x_n.

If a sequence M, of level n, is increasing in level, with $\mathcal{F} = \mathsf{constraints}(M)$, we say that it is *feasible*, when the set of univariate polynomial constraints $v[M_n](\mathcal{F})$ has a solution. We write $\mathsf{feasible}(M)$ to denote the feasible set of $v[M_n](\mathcal{F})$. Given an additional polynomial constraint $F \in \mathbb{Z}[x_1, \ldots, x_n]$, we say that F is *compatible* with the sequence M, when $\mathsf{feasible}([\![M, F]\!]) \neq \emptyset$ and denote this with a predicate $\mathsf{compatible}(F, M)$. In our actual implementation, we represent feasible sets using a set of intervals with real algebraic endpoints. The predicate $\mathsf{compatible}(F, M)$ is implemented using real root isolation and sign evaluation procedures. In the Appendix, we sketch the algorithms used to implement these procedures, and provide references to the relevant literature.

Definition 1 (Well-Formed State). *We say a state $\langle M, \mathcal{C} \rangle_n$ is well-formed when M is non-redundant, increasing in level,* $\mathsf{level}(M) = n$*, and all of the following hold.*

1. *Clauses up to level n are satisfied, i.e. we have that $v[M_n](\mathcal{C}_{n-1}) = \mathsf{true}$.*

2. *Literals up to level* n *are satisfied, i.e. for each* $F \in \mathsf{constraints}(M_{n-1})$ *we have that that* $\upsilon[M_n](F) = \mathsf{true}$.
3. *Literals of level* n *are consistent, i.e. we have that* $\mathsf{feasible}(M) \neq \emptyset$.
4. *Propagated literals* $E{\rightarrow}F$ *are implied, i.e. for all literals* $F' \neq F$ *in* E, $\upsilon[M_n](F') = \mathsf{false}$ *or* $\neg F' \in \mathsf{constraints}(M)$.

Intuitively, in a well-formed state we commit to the variable assignment of lower levels, and we make sure that the current level is still consistent. With this in mind, given a polynomial constraint F over variables $x_1, \ldots, x_n$, and a well-formed state M with $\mathsf{level}(M) = n$, we define the *state value* of F in M as

$$\mathsf{value}(F, M) = \begin{cases} \upsilon[M_n](F) & x_n \notin \mathsf{vars}(F) \ , \\ \mathsf{true} & F \in \mathsf{constraints}(M) \ , \\ \mathsf{false} & \neg F \in \mathsf{constraints}(M) \ , \\ \mathsf{undef} & \text{otherwise.} \end{cases}$$

Naturally, we overload value to also evaluate clauses of polynomial constraints, and sets of clauses, i.e. for a clause C we define $\mathsf{value}(C, M)$ to be true, if any of the literals evaluates to true, false if all literals evaluate to false, and undef otherwise.

We are now ready to define the transition system. We separate the transition rules into three groups: the search rules, the clause processing rules, and the conflict analysis rules. The *search rules* are the main driver of the procedure, with the responsibility for selecting clauses to process, creating the variable assignment while lifting the levels, and detecting Boolean conflicts. The search rules operate on well-formed states $\langle M, \mathcal{C}\rangle_n$. If the search rules select a clause C to process, we switch to a state $\langle M, \mathcal{C}\rangle_n \vDash C$, where we can apply the set of clause processing rules. The notation $\vDash C$ designates that we are performing semantic reasoning in order to assign a value to a literal of C. If the search rules detect that in the current state some clause $C \in \mathcal{C}$ is falsified, we switch to a state $\langle M, \mathcal{C}\rangle_n \vdash C$, where we can apply the *conflict analysis rules*. The notation $\vdash C$ denotes that we are trying to produce a proof of why C is inconsistent in the current state.

Finally, given a polynomial constraint problem $\mathcal{C}$, with $\mathsf{vars}(\mathcal{C}) = \{x_1, \ldots, x_n\}$, the overall goal of the procedure is, starting from an initial state $\langle [\![]\!], \mathcal{C}\rangle_1$, and applying the rules, to end up either in a state $\langle \upsilon, \mathsf{sat}\rangle$, indicating that the initial set of clauses $\mathcal{C}$ is satisfiable where the assignment υ is the witness, or derive unsat, which indicates that the set $\mathcal{C}$ unsatisfiable.

Search Rules. Fig 1 presents the set of search rules. The SELECT-CLAUSE rule selects one of the clauses of the current level, whose value is still undetermined, and transitions into the clause processing mode that will hopefully satisfy the clause. The CONFLICT rule detects if there is a clause of the current level that is inconsistent in the current state, and transitions into the conflict resolution mode that will explain the conflict and backtrack appropriately. On the other hand, if all the clauses of the current level are satisfied, we can either transition

to the next level, using the LIFT-LEVEL rule, or conclude that our problem is satisfiable, using the SAT rule. Since at this point the current level is consistent, in addition to formally introducing the new level, the LIFT-LEVEL rule selects a particular value for the current variable from the feasible set of the current level. Note that once we move to the next level, all the clauses of previous levels have values in the state, and can never be selected by the SELECT-CLAUSE or the CONFLICT rules. We conclude this set of rules with the FORGET rule that can be used to eliminate any learned clause (a clause added while analyzing conflicts) from the current set of clauses.

SELECT-CLAUSE				
$\langle M, \mathcal{C} \rangle_k$	$\longrightarrow$	$\langle M, \mathcal{C} \rangle_k \vDash C$	**if**	$C \in \mathcal{C}_k$ $\mathsf{value}(C, M) = \mathsf{undef}$
CONFLICT				
$\langle M, \mathcal{C} \rangle_k$	$\longrightarrow$	$\langle M, \mathcal{C} \rangle_k \vdash C$	**if**	$C \in \mathcal{C}_k$ $\mathsf{value}(C, M) = \mathsf{false}$
SAT				
$\langle M, \mathcal{C} \rangle_k$	$\longrightarrow$	$\langle \upsilon[M], \mathsf{sat} \rangle$	**if**	$x_k \notin \mathsf{vars}(\mathcal{C})$
LIFT-LEVEL				
$\langle M, \mathcal{C} \rangle_k$	$\longrightarrow$	$\langle [\![M, x_k \mapsto \alpha]\!], \mathcal{C} \rangle_{k+1}$	**if**	$x_k \in \mathsf{vars}(\mathcal{C})$ $\alpha \in \mathsf{feasible}(M)$ $\mathsf{value}(\mathcal{C}_k, M) = \mathsf{true}$
FORGET				
$\langle M, \mathcal{C} \rangle_k$	$\longrightarrow$	$\langle M, \mathcal{C} \setminus \{C\} \rangle_k$	**if**	$C \in \mathcal{C}$ C is a learned clause

Fig. 1. The search rules.

Clause Processing Rules. In this set of rules, presented in Fig 2, we are trying to assign a currently unassigned literal of the given clause C, hoping to satisfy the clause. When one of the clause processing rules is applied, we immediately switch back to the search rules. As usual in a CDCL-style procedure, the simplest way to satisfy the clause C is to perform the Boolean unit propagation, if applicable, by using the $\mathbb{B}$-PROPAGATE rule. We restrict the application of this rule so that adding the constraint to the state keeps it consistent, i.e., it is compatible with the current set of constraints. If this is the case, we add the constraint to the state together with the explanation (clause C itself). To allow more complex propagations, the ones that are valid in $\mathbb{R}$ modulo the current state, we provide the $\mathbb{R}$-PROPAGATE rule. This rule can propagate a constraint from the clause, if assuming the negation would be incompatible with the current state. The $\mathbb{R}$-PROPAGATE rule is equipped with an explanation function explain. The explain function, given a polynomial constraint F, and the trail M,

Decide-Literal

$\langle M, \mathcal{C}\rangle_k \vDash C$	$\longrightarrow$	$\langle [\![M, F_1]\!], \mathcal{C}\rangle_k$	**if**	$F_1, F_2 \in C$ $\forall i : \mathsf{value}(F_i, M) = \mathsf{undef}$ $\mathsf{compatible}(F_1, M)$

$\mathbb{B}$-Propagate

$\langle M, \mathcal{C}\rangle_k \vDash C$	$\longrightarrow$	$\langle [\![M, C{\to}F]\!], \mathcal{C}\rangle_k$	**if**	$C = F_1 \vee \ldots \vee F_m \vee F$ $\mathsf{value}(F, M) = \mathsf{undef}$ $\forall i : \mathsf{value}(F_i, M) = \mathsf{false}$ $\mathsf{compatible}(F, M)$

$\mathbb{R}$-Propagate

$\langle M, \mathcal{C}\rangle_k \vDash C$	$\longrightarrow$	$\langle [\![M, E{\to}F]\!], \mathcal{C}\rangle_k$	**if**	$F \in \mathsf{literals}(C)$ $\mathsf{value}(F, M) = \mathsf{undef}$ $\neg\,\mathsf{compatible}(\neg F, M)$ $E = \mathsf{explain}(F, M)$

Fig. 2. The clause satisfaction rules.

returns the explanation clause $E = \mathsf{explain}(F, M)$ that is valid in $\mathbb{R}$, and implies the constraint F under the current assignment (i.e., $F \in E$, all literals in E but F are false, and the $\mathbb{B}$-Propagate rule applies to E and F). The clause E may contain new literals that do not occur in $\mathcal{C}$, but they can only contain variables from lower levels. Now, it becomes clear the motivation for the definition of the state value function value. Given a new literal F_i from E, $\neg F_i \notin \mathsf{constraints}(M)$, but $\mathsf{value}(F_i, M) = \mathsf{false}$ because $v[M](F_i) = \mathsf{false}$. In $\mathbb{R}$-Propagate, the clause E is eagerly generated, this simplification clarifies the presentation, but in our actual implementation, we compute them only if they are needed during conflict resolution. Finally, if we cannot deduce the value of an unassigned literal, we can assume a value for such a literal using the Decide-Literal rule.

Conflict analysis rules. The conflict analysis rules start from an initial proper state $\langle M, \mathcal{C}\rangle_n \vdash C$, where $C \in \mathcal{C}$ is the conflicting clause. The conflict analysis is a standard Boolean conflict analysis [40] with a model-based twist. As the rules move the state backwards, the goal is to construct a new resolvent clause R, that will explain the conflict and ensure progress in the search. This means that, when we backtrack the sequence M just enough, the addition of R will ensure progress in the search by eliminating the inconsistent part from the state, and thus forcing the search rules to change some of the choices made. On the other hand, if the conflict analysis backtracks the state all the way into an empty state, this will be a signal that the original problem is unsatisfiable. Once the conflict analysis backtracks enough and deduces the resolvent R, then we pass it to the clause processing immediately.[6]

Termination. Our decision procedure consists of all three sets of rules described above. Any derivation will proceed by switching amongst the three distinct

[6] This is crucial in order to ensure termination.

Resolve-Propagation			
$\langle [\![M, E \to F]\!], \mathcal{C} \rangle_k \vdash C$	$\longrightarrow$ $\langle M, \mathcal{C} \rangle_k \vdash R$	**if**	$\neg F \in C$ $R = \mathsf{resolve}(C, E, F)$
▷ resolve returns the standard Boolean resolvent			
Resolve-Decision			
$\langle [\![M, F]\!], \mathcal{C} \rangle_k \vdash C$	$\longrightarrow$ $\langle M, \mathcal{C} \cup \{C\} \rangle_k \vDash C$	**if**	$\neg F \in C$
Consume			
$\langle [\![M, F]\!], \mathcal{C} \rangle_k \vdash C$	$\longrightarrow$ $\langle M, \mathcal{C} \rangle_k \vdash C$	**if**	$\neg F \notin C$
$\langle [\![M, E \to F]\!], \mathcal{C} \rangle_k \vdash C$	$\longrightarrow$ $\langle M, \mathcal{C} \rangle_k \vdash C$	**if**	$\neg F \notin C$
Drop-Level			
$\langle [\![M, x_{k+1} \mapsto \alpha]\!], \mathcal{C} \rangle_{k+1} \vdash C$	$\longrightarrow$ $\langle M, \mathcal{C} \rangle_k \vdash C$	**if**	$\mathsf{value}(C, M) = \mathsf{false}$
$\langle [\![M, x_{k+1} \mapsto \alpha]\!], \mathcal{C} \rangle_{k+1} \vdash C$	$\longrightarrow$ $\langle M, \mathcal{C} \cup \{C\} \rangle_k \vDash C$	**if**	$\mathsf{value}(C, M) = \mathsf{undef}$
Unsat			
$\langle [\![]\!], \mathcal{C} \rangle_1 \vdash C$	$\longrightarrow$ unsat		

Fig. 3. The conflict analysis rules.

modes. Proving termination in the basic CDCL(T) framework is usually a fairly straightforward task, as the new explanation and conflict clauses always contain only literals from the finite set of literals in the initial set of constraints. In our case, the main conundrum in proving termination is that we allow the explanations to contain fresh constraints, which, if we are not careful, could lead to non-termination. We therefore also require the set of new constraints to be finite.

We call an explanation function explain a *finite basis explanation function* with respect to a set of constraints $\mathcal{C}$, when there is a finite set of polynomial constraints $\mathcal{B}$ such that for any derivation of the proof rules, the clauses returned by applications of explain always contain only constraints from the basis $\mathcal{B}$. Having such an explanation function will therefore provide us with a termination argument, and we will provide one such explanation function for the theory of reals in the next section.

Theorem 1. *Given a set of polynomial constraints $\mathcal{C}$, and assuming a finite basis explanation function* explain*, any derivation starting from the initial state $\langle [\![]\!], \mathcal{C} \rangle_1$ will terminate either in a state $\langle v, \mathsf{sat} \rangle$, where the assignment v satisfies the constraints $\mathcal{C}$, or in the* unsat *state. In the later case, the set of constraints $\mathcal{C}$ is unsatisfiable in $\mathbb{R}$.*

Proof. Assume we have a set of polynomial constraints $\mathcal{C}_0$, over the variables $x_1, \ldots, x_n$, and a finite-basis explanation function explain. Starting from the initial state $\langle [\![]\!], \mathcal{C}_0 \rangle_1$, we claim that any derivation of the transition system (finite or infinite), satisfies the following properties

1. the derivation consists of only well-formed states;

2. the only possible "sink states" are the sat and the unsat states;
3. all $\vdash C$ clauses are implied by the initial constraints $\mathcal{C}_0$;
4. during conflict analysis the $\vdash C$ clause evaluates to false;

Assuming termination, the above properties the statement can be proven easily. Since sat and unsat are the only sink states, the derivation will terminate in one of these states. Since the LIFT-LEVEL rule considers the variables $x_1, \ldots, x_n$ in order, we can only enter the satisfiable state if it is of the form $\langle v, \mathsf{sat}\rangle_{n+1}$. Consequently, by the precondition of the LIFT-LEVEL rule, and the fact that we never remove the original constraints from $\mathcal{C}_0$, all the constraints in $\mathcal{C}_0$ are satisfied by v. Therefore if we terminate in a sat state, the original problem is indeed satisfiable. On the other hand, if we terminate in the unsat state, by above properties, the conflicting clause is implied by $\mathcal{C}_0$ and evaluates to false in the state $\langle [\![]\!], \mathcal{C}\rangle_1$. But, since there are no assertions in the trail, and variable assignment $v([\![]\!])$ does not assign any variables, it must be that the constraint is trivially false. Having that falsity implied by the original constraints, the initial constraints themselves must truly be unsatisfiable.

The first two properties in the list above are a fairly easy exercise in case analysis and induction, so we skip those and concentrate on the more interesting properties. Proving the properties of conflict analysis is also quite straightforward, via induction on the number of conflicts, and conflict analysis steps. Clearly, initially, we have that C evaluates to false (the precondition of the CONFLICT rule), and is implied by $\mathcal{C}$ by induction. Then, every new clause that we produce during conflict resolution is obtained by the Boolean resolve rule, which will produce a valid deduction. Additionally, since the clause we are resolving with is a proper explanation, it will have all literals except the one we are resolving evaluate to false. Therefore, the resolvent also evaluates to false. As we backtrack down the trail with the conflicting clause, by definition of value and the preconditions of the rules, the clause still remains false.

Now, let us prove that the system terminates. It is clear that both the clause processing rules (one step transitions) and the conflict analysis rules (always removing elements from the trail) always terminate in a finite number of steps, and return to the search rules (or the unsat state). For the sake of the argument, let us assume that there is a derivation that does not terminate, and therefore does not enter the unsat state. We can define a big-step transition relation $\longrightarrow_{\mathrm{bs}}$ that covers a transition from a search state, applying one or more transitions in the processing or analysis rules, and returns to a search state.

By assumption, we have a finite-basis explanation function explain, so we can assume a set of polynomial constraint literals $\mathcal{B}$ from which all the clauses that we can see during the search are constructed. In order to keep progress of the search, we first define a function search-level that, given the trail M, returns a pair (k, l), where k is the index of the next variable we are trying to assign, i.e. k is one more than the number of variable assignments in M, and l is the number of decided literals (applications of the DECIDE-LITERAL rule) in M. Note that the search-level of any state that we can encounter is always a pair (k, l) with $1 \leq k \leq n$ and $l \leq |\mathcal{B}|$. Given such a pair (k, l) we define the function

$\mathsf{search\text{-}subseq}(M,k,l)$ to be the largest prefix that contains at most k variable assignments and at most l decided literals, i.e. the largest prefix of M with $\mathsf{search\text{-}level}(M) \leq (k,l)$.

To define the measure of a state, we first define a series of weight functions ω_k that, given of a sequence M, returns

$$\omega_k(M) = \begin{cases} |\{\, F \in \mathcal{B}_k \mid \mathsf{value}(M_k, F) = \mathsf{undef} \,\}| & (k,0) \leq \mathsf{search\text{-}level}(M), \\ \infty & \text{otherwise.} \end{cases}$$

In other words, if we are trying to assign the variable x_k, where we already performed a number of literal decisions, this state is as heavy as the number of literals left in the basis containing only variables $x_1, \ldots, x_k$ that could still possibly be assigned.

In order to prove termination, we will track the progress of all levels simultaneously. We define the function Ω to map a sequence well-formed state $\langle M, \mathcal{C}\rangle_k$ into a $n(|\mathcal{B}|+1)+2$-tuple as

$$\begin{aligned} \Omega(M) = \langle & \omega_1(\mathsf{search\text{-}subseq}(M,1,0)), \ldots, \omega_1(\mathsf{search\text{-}subseq}(M,1,|\mathcal{B}|)), \\ & \omega_2(\mathsf{search\text{-}subseq}(M,2,0)), \ldots, \omega_2(\mathsf{search\text{-}subseq}(M,2,|\mathcal{B}|)), \\ & \vdots \\ & \omega_n(\mathsf{search\text{-}subseq}(M,n,0), \ldots, \omega_n(\mathsf{search\text{-}subseq}(M,n,|\mathcal{B}|)), \\ & \omega_{n+1}(M), |\mathcal{C}|\rangle \ . \end{aligned}$$

Given two well-formed states with trails M_1 and M_2, we write $M_1 \lessdot M_2$ if $\Omega(M_1) <_{\mathsf{lex}} \Omega(M_2)$, where $<_{\mathsf{lex}}$ is the natural lexicographical extension of the order $<$ on $\mathbb{N} \cup \{\infty\}$. Now consider a transition of the search $\langle M_1, \mathcal{C}_1\rangle_{k_1} \longrightarrow_{\mathrm{bs}} \langle M_2, \mathcal{C}_2\rangle_{k_2}$ and the following cases.

- If this transition was initiated by the Select-Clause rule, a new literal was assigned at the current literal decision level, or a new decision was introduced. In both cases $M_2 \lessdot M_1$ as either one element of the sequence decreased by 2 (literal and its negation were assigned), or the next element of the sequence decreased from ∞ to a finite value.
- If this transition was initiated by the Lift-Level rule, switching to level k, the first element of row k in $\Omega(M)$ decreased from ∞ to a finite value, so $M_2 \lessdot M_1$ again.
- If we went into conflict analysis mode via the Conflict rule, we will backtrack accordingly, learn a new clause, and then assign at least one new literal of the learned clause. Here note that if we used the Decide-Literal to assign this literal, it must be that we stepped out of conflict analysis with an application of the Drop-Level rule. If not, then we must have used the Resolve-Decision rule, which would force us to apply the $\mathbb{B}$-Propagate rule instead. Therefore, in such a case, we didn't remove any literal decisions at the level k we backtracked to, but have in fact introduced a new one, again decreasing an element of the measure from ∞ to a finite value. Otherwise,

if the value of the literal is assigned by one of the propagation rules, the measure decreases as in the first case. In both cases, it follows, we have that $M_2 \lessdot M_1$ again.
– If we applied the Forget rule, it is clear that only last element of the measure decreases, and hence also $M_2 \lessdot M_1$.

Since, we covered all cases, the function Ω is always decreasing, and termination of the system follows. □

Example 3. First, for the sake of this example, let us restrict ourselves to the case of linear constraints. When solving a set of linear constraints $\mathcal{C}$, one can use the Fourier-Motzkin elimination rule to define the explain function. As shown in [33,28], this will give a finite-basis $\mathcal{B}$ with respect to $\mathcal{C}$ that is obtained by closing $\mathcal{C}$ under the application of Fourier-Motzkin elimination step. It is fairly easy to show that the closure is a finite set, since we always produce constraints with one variable less.

We explain the search rules by applying them to the following set of linear polynomial constraints

$$\mathcal{C} = \{\ (\underbrace{x+1 \leq 0 \vee x-1 \geq 0}_{C_1}),\ \underbrace{x+y>0}_{C_2},\ \underbrace{x-y>0}_{C_3}\ \} \ .$$

During the search, we associate x with level 1, and y with level 2. Therefore the constraints of level 1 are $\mathcal{C}_1 = \{C_1\}$, and the constraints of level 2 are $\mathcal{C}_2 = \{C_2, C_3\}$. The following is a derivation of the transition system, starting from the initial state $\langle [\![]\!], \mathcal{C}\rangle$, applying the rules until we encounter a conflict.

$\langle [\![]\!], \mathcal{C}\rangle_1$

$\downarrow$ Select-Clause $(x+1 \leq 0) \vee (x-1 \geq 0)$, Decide-Literal $(x+1 \leq 0)$

$\langle [\![(x+1 \leq 0)]\!], \mathcal{C}\rangle_1$

$\downarrow$ Lift-Level

$\langle [\![(x+1 \leq 0), x \mapsto -1]\!], \mathcal{C}\rangle_2$

$\downarrow$ Select-Clause $(x+y>0)$, $\mathbb{B}$-Propagate $(x+y>0)$

$\langle [\![(x+1 \leq 0), x \mapsto -1, C_2 \rightarrow (x+y>0)]\!], \mathcal{C}\rangle_2$

$\downarrow$ Select-Clause $(x-y>0)$
$\mathbb{R}$-Propagate $(x-1 \leq 0)$ with $E_1 \equiv (x+y \leq 0) \vee (x-y \leq 0) \vee (x>0)$

$\langle [\![(x+1 \leq 0), x \mapsto -1, C_2 \rightarrow (x+y>0), E_1 \rightarrow (x-y \leq 0)]\!], \mathcal{C}\rangle_2$

$\downarrow$ Conflict

$\langle [\![(x+1 \leq 0), x \mapsto -1, C_2 \rightarrow (x+y>0), E_1 \rightarrow (x-y \leq 0)]\!], \mathcal{C}\rangle_2 \vdash (x-y>0)$

The derivation above assigns the first literal of C_1 to true, and then selects the value of -1 for x. With this value we go to the next level, and we then propagate C_2 to true by unit propagation. We continue by processing the clause $x-y>0$. But, under the assignment $v[M_2](x) = -1$ the constraints $x+y>0$ and $x-y>0$ evaluate to $y-1>0$ and $-y-1>0$, respectively, which taken together are inconsistent. This means that $x-y>0$ is incompatible with the current state,

and we use the $\mathbb{R}$-Propagate rule to propagate $\neg(x - y > 0) \equiv x - y \leq 0$. In order to explain the propagation of $x - y \leq 0$, we use a Fourier-Motzkin elimination step to obtain $E_1 \equiv (x + y > 0) \wedge (x - y > 0) \implies (x > 0)$ As soon as we propagate $x - y \leq 0$, we enter a conflict with the clause C_3, and we continue to conflict analysis mode.

Below is the continuation of the derivation that uses the conflict analysis rules to explain the conflict and backtrack.

$\langle [\![(x+1 \leq 0), x \mapsto -1, C_2 \rightarrow (x+y>0), E_1 \rightarrow (x-y \leq 0)]\!], \mathcal{C} \rangle_2 \vdash (x-y>0)$

$\downarrow$ Resolve-Propagation
$\mathsf{resolve}(x-y>0, E_1, (x-y \leq 0)) = (x+y \leq 0) \vee (x>0)$

$\langle [\![(x+1 \leq 0), x \mapsto -1, C_2 \rightarrow (x+y>0)]\!], \mathcal{C} \rangle_2 \vdash (x+y \leq 0) \vee (x>0)$

$\downarrow$ Resolve-Propagation
$\mathsf{resolve}((x+y \leq 0) \vee (x>0), (x+y>0), (x+y>0)) = (x>0)$

$\langle [\![(x+1 \leq 0), x \mapsto -1]\!], \mathcal{C} \rangle_2 \vdash (x>0)$

$\downarrow$ Drop-Level

$\langle [\![(x+1 \leq 0)]\!], \mathcal{C} \cup \{x>0\} \rangle_1 \vDash (x>0)$

$\downarrow$ $\mathbb{R}$-Propagate $(x \leq 0)$ with $E_2 \equiv (x+1>0) \vee (x \leq 0)$

$\langle [\![(x+1 \leq 0), E_2 \rightarrow (x \leq 0)]\!], \mathcal{C} \cup \{x>0\} \rangle_1$

$\downarrow$ Conflict

$\langle [\![(x+1 \leq 0), E_2 \rightarrow (x \leq 0)]\!], \mathcal{C} \cup \{x>0\} \rangle_1 \vdash (x>0)$

$\downarrow$ Resolve-Propagation
$\mathsf{resolve}((x>0), E_2, (x \leq 0)) = (x+1>0)$

$\langle [\![(x+1 \leq 0)]\!], \mathcal{C} \cup \{x>0\} \rangle_1 \vdash (x+1>0)$

$\downarrow$ Resolve-Decision $(x+1 \leq 0)$

$\langle [\![]\!], \mathcal{C} \cup \{x>0, x+1>0\} \rangle_1 \vDash (x+1>0)$

$\downarrow$ $\mathbb{B}$-Propagate $(x+1>0)$

$\langle [\![(x+1>0) \rightarrow (x+1>0)]\!], \mathcal{C} \cup \{x>0, x+1>0\} \rangle_1$

4 Producing Explanations

Given a polynomial constraint F s.t. $\mathsf{poly}(F) \in \mathbb{Z}[\boldsymbol{y}, x]$, and a trail M such that $\neg F$ is not compatible with M, the procedure $\mathsf{explain}(F, M)$ returns an explanation clause E that implies F under the current assignment. This clause is of the form $\mathcal{E} \wedge \mathcal{F} \implies F$, where $\mathcal{E}$ and $\mathcal{F}$ are sets of literals with $\mathsf{poly}(\mathcal{E}) \subset \mathbb{Z}[\boldsymbol{y}]$ and $\mathsf{poly}(\mathcal{F}) \subset \mathbb{Z}[\boldsymbol{y}, x]$. All literals in $\mathcal{F}$ occur in M, and all literals in $\mathcal{E}$ evaluate to true in the current assignment. Note that $\mathcal{E}$ may contain new literals, so we must ensure that the new literals in $\mathsf{poly}(\mathcal{E})$ are a subset of some finite basis.

In principle, for any theory that admits elimination of quantifiers, it is possible to construct an explanation function $\mathsf{explain}$. In this section, we describe how to produce an $\mathsf{explain}$ procedure based on cylindrical algebraic decomposition (CAD). Before that, we first make a short interlude into the world of CAD.

4.1 Cylindrical Algebraic Decomposition

Delineability plays a crucial role in the theory of CADs and in the construction of our $\mathsf{explain}$ procedure. Following the terminology used in CAD, we say a connected subset of $\mathbb{R}^k$ is a *region*. Given a region S, the *cylinder* Z over S is $S \times \mathbb{R}$. A θ-*section* of Z is a set of points $\langle \boldsymbol{\alpha}, \theta(\boldsymbol{\alpha}) \rangle$, where $\boldsymbol{\alpha}$ is in S and θ is a continuous function from S to $\mathbb{R}$. A (θ_1, θ_2)-*sector* of Z is the set of points $\langle \boldsymbol{\alpha}, \beta \rangle$, where $\boldsymbol{\alpha}$ is in S and $\theta_1(\boldsymbol{\alpha}) < \beta < \theta_2(\boldsymbol{\alpha})$ for continuous functions $\theta_1 < \theta_2$ from S to $\mathbb{R}$. Sections and sectors are also regions. Given a subset of S of $\mathbb{R}^k$, a *decomposition* of S is a finite collection of disjoint regions $S_1, \ldots, S_n$ such that $S_1 \cup \ldots \cup S_n = S$. Given a region S, and a set of continuous functions $\theta_1 < \ldots < \theta_n$ from S to $\mathbb{R}$, we can decompose the cylinder $S \times \mathbb{R}$ into the following regions:

- the θ_i-sections, for $1 \leq i \leq n$, and
- the (θ_i, θ_{i+1})-sectors, for $0 \leq i \leq n$,

where, with slight abuse of notation, we define θ_0 as the constant function that returns $-\infty$ and θ_{n+1} the constant function that returns ∞.

A set of polynomials $\{f_1, \ldots f_s\} \subset \mathbb{Z}[\boldsymbol{y}, x]$, $\boldsymbol{y} = (y_1, \ldots, y_n)$, is said to be *delineable* in a region $S \subset \mathbb{R}^n$ if the following conditions hold:

1. For every $1 \leq i \leq s$, the *total number of complex roots* of $f_i(\alpha, x)$ remains invariant for any α in S.
2. For every $1 \leq i \leq s$, the *number of distinct complex roots* of $f_i(\alpha, x)$ remains invariant for any α in S.
3. For every $1 \leq i < j \leq s$, the *number of common complex roots* of $f_i(\alpha, x)$ and $f_j(\alpha, x)$ remains invariant for any α in S.

Theorem 2 (Corollary 8.6.5 of [34]). *Let A be a set of polynomials in $\mathbb{Z}[\boldsymbol{y}, x]$, delineable in a region $S \subset \mathbb{R}^n$. Then, the real roots of A vary continuously over S, while maintaining their order.*

Example 4. Consider the polynomial $f = x^2 + y^2 + z^2 - 1$, with zeros of f depicted in Fig 4(a) together with two squiggly regions of $\mathbb{R}^2$. In the region S_1 that does not intersect the sphere, polynomial f is delineable, as the number of complex (and real) roots of $f(\alpha, x)$ is 2 for any α in S_1. In the region S_2 that intersects the sphere, f is not delineable, as the number of real roots of f varies from 0 (α's outside the unit circle), 1 (on the circle), and 2 (inside the unit circle).

We will call a *projection operator* any map P that, given a variable x and set of polynomials $A \subset \mathbb{Z}[\boldsymbol{y}, x]$, transforms A into a set of polynomials $\mathsf{P}(A, x) \subset \mathbb{Z}[\boldsymbol{y}]$. We call $\mathsf{P}(A, x)$ the *projection* of A under P with respect to variable x. In his seminal paper [11], Collins introduced a projection operator which we denote with P_c. In order to define the operator P_c, we first need to define some "advanced" operations on polynomials, and we refer the reader to [30,3,7] for a more detailed exposition.

Let $f, g \in \mathbb{Z}[\boldsymbol{y}, x]$ be two polynomials with $n = \min(\mathsf{deg}(f, x), \mathsf{deg}(g, x))$. For $k = 0, \ldots, n-1$, we denote with $\mathsf{S}_k(f, g, x)$ the k-th *subresultant* of f and g. The

k-th subresultant is defined as the determinant of the k-th Sylvester-Habicht matrix of f and g, and is a polynomial of degree $\leq k$ in x with coefficients in $\mathbb{Z}[\boldsymbol{y}]$. The matrix in question is a particular matrix containing as elements the coefficients of f and g. Additionally, we denote with $\mathsf{psc}_k(f,g,x)$ the k-th *principal subresultant coefficient* of f and g, which is the coefficient of x^k in the polynomial $\mathsf{S}_k(f,g,x)$, and define $\mathsf{psc}_n(f,g,x) = 1$. We denote the sequence of principle subresultant coefficients as $\mathsf{psc}(f,g,x) = (\mathsf{psc}_0(f,g,x), \ldots, \mathsf{psc}_n(f,g,x))$.

Theorem 3 (Theorem 2 in [11]). *Let $f, g \in \mathbb{Z}[\boldsymbol{y}, x]$ be non-zero polynomials. Then $\mathsf{deg}(\mathsf{gcd}(A,B),x) = k$ if and only if k is the least j such that $\mathsf{psc}_j(f,g) \neq 0$.*

Since the number of common complex roots of two polynomials corresponds to the degree of their gcd, the previous theorem provides us with a way to describe this number.

Definition 2 (Collins Projection). *Given a set of polynomials $A = \{f_1, \ldots, f_m\} \subset \mathbb{Z}[\boldsymbol{y}, x]$ the Collins projector operator $\mathsf{P}_c(A,x)$ is defined as*

$$\bigcup_{f \in A} \mathsf{coeff}(f,x) \cup \bigcup_{\substack{f \in A \\ g \in \mathsf{R}(f,x)}} \mathsf{psc}(g, g'_x, x) \cup \bigcup_{\substack{i<j \\ g_i \in \mathsf{R}(f_i,x) \\ g_j \in \mathsf{R}(f_j,x)}} \mathsf{psc}(g_i, g_j, x) \ ,$$

In order to denote the individual parts of the projection, in order, we designate them as $\mathsf{P}_c^1(A,x)$, $\mathsf{P}_c^2(A,x)$ and $\mathsf{P}_c^3(A,x)$.

Let $A = \{f_1, \ldots, f_m\} \subset \mathbb{Z}[\boldsymbol{y}]$ be a set of polynomials, where $\boldsymbol{y} = (y_1, \ldots, y_n)$, and S be a region of $\mathbb{R}^n$. If for any assignment υ such that $\upsilon(\boldsymbol{y}) = \boldsymbol{\alpha} \in S$, the polynomials in A have the same sign under υ, we say that A is *sign-invariant on* S.

Theorem 4 (Theorem 4 in [11]). *Given a finite set of polynomials $A \subset \mathbb{Z}[\boldsymbol{y}, x]$, where $\boldsymbol{y} = (y_1, \ldots, y_n)$, and let S be a region of $\mathbb{R}^n$. If $\mathsf{P}_c(A)$ is sign invariant on S, then A is delineable over S.*

The projection operator P_c guarantees delineability on any region S where the projection set $\mathsf{P}_c(A,x)$ is sign-invariant, due to the following:

1. The degree of $f_i(\alpha, x)$ (and the total number of complex roots) remains invariant for any α in S, by $\mathsf{P}_c^1(A,x)$ being sign-invariant.
2. The multiplicities of complex roots of $f_i(\alpha, x)$ remains invariant for any α in S, by $\mathsf{P}_c^2(A,x)$ being sign-invariant and Theorem 3
3. The number of common complex roots of $f_i(\alpha, x)$ and $f_j(\alpha, x)$ remain invariant for any α in S, by $\mathsf{P}_c^3(A,x)$ being sign-invariant and Theorem 3.

A *sign assignment* for a set of polynomials A is a mapping σ, from polynomials in A to $\{-1, 0, 1\}$. Given a set of polynomials $A \subset \mathbb{Z}[\boldsymbol{y}, x]$, we say a sign assignment σ is *realizable* with respect to some $\boldsymbol{\alpha}$ in $\mathbb{R}^n$, if there exists a $\beta \in \mathbb{R}$ such that every $f \in A$ takes the sign corresponding to its sign assignment, i.e., $\mathsf{sgn}(f(\boldsymbol{\alpha}, \beta)) = \sigma(f)$. The function sgn maps a real number to its sign $\{-1, 0, 1\}$. We use $\mathsf{signs}(A, \alpha)$ to denote the set of realizable sign assignments of A with respect to α.

Lemma 1. *If a set of polynomials $A \subset \mathbb{Z}[\boldsymbol{y}, x]$ is delineable over a region S, then $\mathsf{signs}(A, \alpha)$ is invariant over S.*

Proof. Since A is delineable over S, by Theorem 2, there are real functions θ_i, continuous on S and ordered, corresponding to roots of polynomials in A. We can therefore decompose the cylinder $S \times \mathbb{R}$ into θ_i-sections and (θ_i, θ_{i+1})-sectors, where each of these regions is connected and the signs of polynomials from A do not change. Let $\sigma_1 \in \mathsf{signs}(A, \boldsymbol{\alpha}_1)$ be a realizable sign assignment, with $\beta_1 \in \mathbb{R}$, such that at $(\boldsymbol{\alpha}_1, \beta_1)$ every polynomial $f \in A$ takes a sign corresponding to $\mathsf{signs}(A, \boldsymbol{\alpha}_1)$. Lets pick an arbitrary other other $\alpha_2 \in S$, and show that we realize σ_1 at $\boldsymbol{\alpha}_2$. We can pick an arbitrary point β_2 in the same sector (or section) R where β_1 came from. We claim that at $\langle \boldsymbol{\alpha}_2, \beta_2 \rangle$ the polynomials in A have the signs required by σ_1.

Assume the opposite, i.e. that there is a polynomial $f \in A$ with $\sigma_1(f) = \mathsf{sgn}(f(\boldsymbol{\alpha}_1, \beta_1)) \neq \mathsf{sgn}(f(\boldsymbol{\alpha}_2, \beta_2))$. Since R is connected we can connect $\langle \boldsymbol{\alpha}_1, \beta_1 \rangle$ and $\langle \boldsymbol{\alpha}_2, \beta_2 \rangle$ with a path π that does not leave R. Having that the sign of f is different at the endpoints of π, it must be that there is a point $\langle \boldsymbol{\alpha}_3, \beta_3 \rangle$ on the path, where the sign of f is 0, and at least another point where the sign of f is not 0. Now we distinguish the following cases

– If R is a (θ_i, θ_{i+1})-sector, then we have isolated a root of a polynomial in A that is between $\theta_i(\boldsymbol{\alpha}_3)$ and $\theta_{i+1}(\boldsymbol{\alpha}_3)$, which is impossible by the construction of the decomposition.
– If R is a θ_i-section, then the polynomial $f(\boldsymbol{\alpha}_3)$ has a root, and this root diverges from θ_i on R, which is impossible due to delineability.

4.2 Projection-Based Explanations

Suppose that we need to produce an explanation for propagating a polynomial constraint F, i.e. we are in a state such that $\neg\,\mathsf{compatible}(\neg F, M)$, with $\mathsf{poly}(F) \in \mathbb{Z}[\boldsymbol{y}, x]$, where $\boldsymbol{y} = (y_1, \ldots, y_n)$. To simplify the presentation, in the following, we write υ for $\upsilon[M]$. A model-based explanation procedure $\mathsf{explain}(F, M)$ consists of the following steps:

1. Find a minimal set $\mathcal{F}$ of literals in M, with $\mathsf{poly}(\mathcal{F}) \subset \mathbb{Z}[\boldsymbol{y}, x]$, such that $\upsilon(\mathcal{F})$ still does not allow a solution for x. We call this set (not necessarily unique) a *conflicting core*. Let A be the set of polynomials $\mathsf{poly}(\mathcal{F}) \cup \{\, \mathsf{poly}(F) \,\}$.
2. Construct a region S of $\mathbb{R}^n$ where A is delineable, and $\upsilon(\boldsymbol{y})$ is in S. Note that, $\neg F$ is incompatible with $\mathcal{F}$ for any other $\boldsymbol{\alpha}'$ in S. This follows from the fact that $\mathsf{signs}(A, \boldsymbol{\alpha})$ remains invariant for any $\boldsymbol{\alpha}$ in S.
3. Define S using extended polynomial constraints, obtaining a new set of constraints $\mathcal{E}$. Then, we define $\mathsf{explain}(F, M) = \mathcal{E} \wedge \mathcal{F} \implies F$.

We later explain how we obtain the minimal set $\mathcal{F}$. We now focus on the second step of the procedure. We first observe that our procedure just requires a connected subset S which contains the current assignment $\upsilon(\boldsymbol{y}) = \boldsymbol{\alpha}$. We therefore add the assignment υ as an additional argument to the projection

operator, and call such a projection operator *model-based*. Given a variable assignment υ, we denote the vanishing signature of a principle subresultant sequence as $\mathsf{v\text{-}psc}(f,g,x,\upsilon) = \mathsf{v\text{-}sig}(\mathsf{psc}_0(f,g,x),\ldots,\mathsf{psc}_n(f,g,x))$. Now, we define our model-based projection operator $\mathsf{P}_m(A,x,\upsilon)$ as follows.

Definition 3 (Model-Based Projection). *Given a set of polynomials $A = \{f_1,\ldots,f_m\} \subset \mathbb{Z}[\boldsymbol{y},x]$ and a variable assignment υ, the modified model-based Collins projector operator $\mathsf{P}_m(A,x,\upsilon)$ is defined as*

$$\bigcup_{f\in A} \mathsf{v\text{-}coeff}(f,\upsilon,x) \cup \bigcup_{\substack{f\in A\\ g=\mathsf{R}(f,x,\upsilon)}} \mathsf{v\text{-}psc}(g,g'_x,x,\upsilon) \cup \bigcup_{\substack{i<j\\ g_i=\mathsf{R}(f_i,x,\upsilon)\\ g_j=\mathsf{R}(f_j,x,\upsilon)}} \mathsf{v\text{-}psc}(g_i,g_j,x,\upsilon) \ .$$

We use the projection operator P_m to compute the region S which contains the current assignment $\upsilon(\boldsymbol{y})$, and show that A is delineable in S. Assume A is a set of polynomials in $\mathbb{Z}[y_1,\ldots,y_n,x]$. First, we will close the set of polynomials A under the application of a projection operator P_m. We compute this closure by computing sets of polynomials $\mathcal{P}^n,\ldots,\mathcal{P}^1$ iteratively, starting from $\mathcal{P}^n = \mathsf{P}_m(A,\upsilon,x)$, and then for $k=n,\ldots,2$, compute the subsequent ones as

$$\mathcal{P}^{k-1} = \mathsf{P}_m(\mathcal{P}^k,y_k,\upsilon) \cup (\mathcal{P}^k \cap \mathbb{Z}[y_1,\ldots,y_{k-1}]) \ .$$

Each set of polynomials $\mathcal{P}^k \subseteq \mathbb{Z}[y_1,\ldots,y_k]$ is obtained by projecting the previous set $\mathcal{P}^{k+1}$ and adding all the polynomials from $\mathcal{P}^{k+1}$ that do not involve the variable y_k.

With the projection closure of A computed, we can now start building the region S inductively, in a bottom up fashion, by constructing a sequence of regions $S^k \subset \mathbb{R}^k$ s.t. $\mathcal{P}^k$ is sign invariant in S^k, and $\mathcal{P}^{k+1}$ is delineable in S^k. For each $k=1,\ldots,n-1$, assume that S^{k-1}, and its defining constraints $\mathcal{E}^k$, have already been constructed. Let us now consider the set of root objects

$$R^k = \left\{ \ \mathsf{root}(f,i) \ \mid \ f\in\mathcal{P}^k, \ 1\le i\le \mathsf{rootcount}(\upsilon(f)) \ \right\} \ .$$

Under the assignment υ each of the root objects $\mathsf{root}(f,i)$ is defined and evaluates to some value $\omega_f^i \in \mathbb{R}_{\mathsf{alg}}$. Moreover, since the polynomials in $\mathcal{P}^k$ are delineable over S^{k-1}, for any other assignment υ' that maps $y_1,\ldots,y_{k-1}$ into S^{k-1}, the polynomials $f\in\mathcal{P}^k$ will have the same number of roots, and the same number of common roots. Therefore, the root objects in R^k will also be defined under any such υ', and will evaluate to values that are in the same exact order.

The values ω_f^i partition the real line into intervals where in each interval, the polynomials $f\in\mathcal{P}^k$ are sign invariant. We will pick the interval that contains $\upsilon(y_k)=\alpha_k$ to construct S^k by selecting one of the appropriate cases

$$\begin{aligned}
\alpha_k &\in (\omega_f^i,\omega_g^j) && \implies \mathcal{E}^k = \mathcal{E}^{k-1} \cup \{ \ y_k >_r \mathsf{root}(f,i), y_k <_r \mathsf{root}(g,j) \ \} \ , \\
\alpha_k &\in (-\infty,\omega_f^i) && \implies \mathcal{E}^k = \mathcal{E}^{k-1} \cup \{ \ y_k <_r \mathsf{root}(f,i) \ \} \ , \\
\alpha_k &\in (\omega_f^i,+\infty) && \implies \mathcal{E}^k = \mathcal{E}^{k-1} \cup \{ \ y_k >_r \mathsf{root}(f,i) \ \} \ , \\
\alpha_k &= \omega_f^i && \implies \mathcal{E}^k = \mathcal{E}^{k-1} \cup \{ \ y_k =_r \mathsf{root}(f,i) \ \} \ .
\end{aligned}$$

Finally, we guarantee that $\mathcal{P}^{k+1}$ is delineable in S^k because the set of polynomials $\mathcal{P}^* = \mathcal{P}^1 \cup \ldots \cup \mathcal{P}^k$ is sign invariant in S^k, and $\mathsf{P}_m(\mathcal{P}^{k+1}, \upsilon, y_{k+1})$ is a subset of $\mathcal{P}^*$. Now, it becomes clear why P_m is sufficient. P_m does not need to include all coefficients, reductums, and the whole psc chain, like P_c does, because the current assignment indicates which coefficients will (and will not) vanish in any element of $\mathcal{E}^k$. This follows from the fact that $(\upsilon(y_1) = \alpha_1, \ldots, \upsilon(y_k) = \alpha_k)$ is in $\mathcal{E}^k$, and all polynomial ins $\mathcal{P}^*$ are sign invariant in $\mathcal{E}^k$.

Once we have computed the regions $S^1, \ldots, S^n$, we can use the region $S = S^n$ and the corresponding constraints $\mathcal{E} = \mathcal{E}^n$ to explain why $\neg F$ is incompatible with $\mathcal{F}$. Thus, we set $\mathsf{explain}(F, M) \equiv \mathcal{E} \wedge \mathcal{F} \implies F$.

Theorem 5. *The explanation function* $\mathsf{explain}(F, M)$ *is a finite-basis explanation function for the existential theory of real closed fields.*

Proof. The key observation is that $\mathsf{P}_m(A, x, \upsilon) \subseteq \mathsf{P}_c(A, x)$, for any A, x and υ. Let $A_0 \subset \mathbb{Z}[x_1, \ldots, x_n]$ be the set of polynomials in the initial set of constraints $\mathcal{C}_0$. Using Collins projection operator $\mathsf{P}_c(A, x)$ we define the sets of polynomials $\mathcal{A}^n, \ldots, \mathcal{A}^1$ iteratively, starting from $\mathcal{A}^n = A_0$, and then for $k = n, \ldots, 2$,

$$\mathcal{A}^{k-1} = \mathsf{P}_c(\mathcal{A}^k, x_k) \cup (\mathcal{A}^k \cap \mathbb{Z}[x_1, \ldots, x_{k-1}])$$

Now, let $\mathcal{A}_c$ be the set $\mathcal{A}^n \cup \ldots \cup \mathcal{A}^1$. The set $\mathcal{A}_c$ is finite and for any $A \subseteq \mathcal{A}_c$ and variable x, we have $\mathsf{P}_c(A, x) \subseteq \mathcal{A}_c$. Consequently, for any $A \subseteq \mathcal{A}_c$, variable x, and assignment υ, $\mathsf{P}_m(A, x, \upsilon) \subseteq \mathcal{A}_c$.

Given a finite set of polynomials A, we have finitely many different polynomial constraints F s.t. $\mathsf{poly}(F) \in A$. This is clear for basic constraints, there are $6 \times |A|$ different basic constraints. For extended constraints $x \nabla_r \mathsf{root}(f, k)$, we recall that $k \leq \mathsf{deg}(f, x)$. Let $\mathcal{B}_c$ be the set $\{F \mid \mathsf{poly}(F) \in \mathcal{A}_c\}$. Thus, $\mathcal{B}_c$ is finite.

Now, it is clear that $\mathsf{explain}(F, M)$ is a finite basis explanation function. Given an initial set of constraints $\mathcal{C}_0$, for any application of $\mathsf{P}_m(A, x, \upsilon)$ in any application of $\mathsf{explain}(F, M)$ in any derivation of our procedure, we have that $\mathsf{P}_m(A, x, \upsilon) \subseteq \mathcal{A}_c$, and consequently $\mathcal{B}_c$ is a finite basis for $\mathsf{explain}(F, M)$.

Example 5. Consider the variable assignment υ, with $\upsilon(x) = 0$, and the set A containing two polynomials polynomials $f_2 = x^2 + y^2 - 1$ and $f_3 = -4xy - 4x + y - 1$. The projection operator P_m maps the set A into $\mathsf{P}_m(A, y, \upsilon)$

$$\{\ (\underbrace{16x^3 - 8x^2 + x + 16}_{f_1})x,\ -4x + 1,\ 4(x+1)(x-1),\ 2,\ 1\ \}\ , \qquad (2)$$

where f_1 is the polynomial from Ex. 1. The zeros of f_2 and f_3 are depicted in Fig. 4(b), together with a set of important points $\{-1, \alpha_1, 0, \frac{1}{4}, 1\}$, where α_1 is the algebraic number from Ex. 1. These are exactly the roots of the projection polynomials (2). It is easy to see from Fig. 4(b) that both f_2 and f_3 are delineable in the intervals defined by these points. Since, in this case, A is delineable on any region of $\mathbb{R}$ where the projection set is sign invariant, A is also delineable

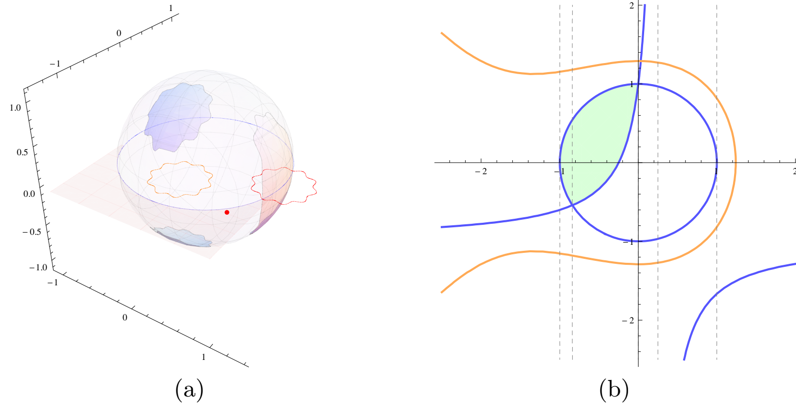

Fig. 4. (a) The sphere corresponding to the roots of $x^2+y^2+z^2-1$, and regions of Ex 4 and Ex 6. (b) Solutions of $f_2 = x^2+y^2-1 = 0$ and $f_3 = -4xy-4x+y-1 = 0$ in blue, solutions of $f_4 = x^3+2x^2+3y^2-5 = 0$ in orange. Solution set of $\{f_2 < 0, f_3 > 0, f_4 < 0$ in green. The dashed lines represent the zeros of the projection set (2).

.

in the region $[0, 0]$ containing $\upsilon(x) = 0$. But, considering another polynomial $f_4 = x^3 + 2x^2 + 3y^2 - 5$, we can see that it is not delineable on the interval $(1, +\infty)$.

Example 6. Consider the polynomial $f = x^2 + y^2 + z^2 - 1$, from Ex. 4, and the constraint $f < 0$ corresponding to the interior of the sphere in Fig. 4(a). Under an assignment υ with $\upsilon(x) = \frac{3}{4}$ and $\upsilon(y) = -\frac{3}{4}$ (the red point in Fig 4(a)) this constraint does not allow a solution for z (it evaluates to $z^2 < -\frac{1}{8}$). In order to explain it, we can compute the projection closure of $A = \{f\}$, using P_m, obtaining $\mathcal{P}_3 = A$ and

$$\mathcal{P}_2 = \mathsf{P}_m(\mathcal{P}_3, \upsilon, z) = \{\ 4x^2 + 4y^2 - 4,\ 2,\ 1\ \}\ ,$$
$$\mathcal{P}_1 = \mathsf{P}_m(\mathcal{P}_2, \upsilon, y) = \{\ 256x^2 - 256,\ 8,\ 4,\ 2,\ 1\ \}\ .$$

The sets of root objects under υ are then

$$R^2 = \{\ \mathsf{root}(\tilde{z}^2 + x^2 - 1, 1),\ \mathsf{root}(\tilde{z}^2 + x^2 - 1, 2)\ \}\ ,$$
$$R^1 = \{\ \mathsf{root}(\tilde{z}^2 - 1, 1),\ \mathsf{root}(\tilde{z}^2 - 1, 2)\ \}\ .$$

Since $\upsilon(x) = \frac{3}{4} = 0.75$ and the root objects of R_1 evaluate to -1 and 1, respectively, we need to fit x between the two of them, and so the constraints corresponding to the region S^1 are $(x > -1)$ and $(x < 1)$. The root objects of R_2 evaluate to $-\frac{\sqrt{7}}{4} \approx -0.6614$ and $\frac{\sqrt{7}}{4} \approx 0.6614$. Since $\upsilon(y) = -\frac{3}{4} = -0.75$, which is below the first root, the single constraint corresponding that we add to describe the region S^2 is $(y < \mathsf{root}(\tilde{z}^2 - x^2 - 1, 1))$. Having computed S^2, we

have obtained the region of delineability that contains the assignment υ, and we are ready to construct the explanation $\mathsf{explain}[\mathsf{P}_m](f < 0, \upsilon)$ as

$$(x \leq -1) \vee (x \geq 1) \vee \neg(y < \mathsf{root}(\tilde{z}^2 - x^2 - 1, 1)) \vee (f \geq 0) \ .$$

The explanation clause states that, in order to fix the conflict under the assignment υ, we must change υ so as to exit the region $-1 < x < 1$ below (in y) the unit circle. This is the region in Fig 4(a) containing $(x, y) = (\frac{3}{4}, -\frac{3}{4})$, colored red.

Isolating the conflicting core. Given a constraint F incompatible with a trail M, we now discuss how to compute a minimal set of constraints $\mathcal{F}$ from M that is not compatible with F. We start with an approximated method. It is based on the observation that every polynomial constraint F in M can be associated with a finite set of infeasible intervals $\mathsf{infset}(F, \upsilon)$ for its maximal variable. For example, assume the constraint $F = x_2^2 - x_1 < 0$ is in M, and x_1 is assigned to 2, then $\mathsf{infset}(F, \upsilon) = \{(-\infty, -\sqrt{2}), (\sqrt{2}, \infty)\}$. Additionally, for each variable x_k, we maintain a disjoint set of infeasible intervals $\mathsf{infset}(x_k)$, where each interval I is tagged with a constraint in M that implies I. Whenever a constraint F is added to M, we update $\mathsf{infset}(x_k)$. Let F be a a new constraint with $\mathsf{poly}(F) \in [x_1, \ldots, x_k]$. If $\mathsf{infset}(\neg F, \upsilon) \cup \mathsf{infset}(x_k)$ contains the whole real line, then we know F is incompatible with M, and the constraints tagging the intervals in $\mathsf{infset}(x_k)$ are a superset of the minimal set. We now can refine this approximation by trying to eliminate constraints from the check while checking whether the infeasible sets of each remaining constraints still cover the whole real line.

Example 7. Consider the set of polynomial constraints $\mathcal{C} = \{f_2 < 0, f_3 > 0, f_4 < 0\}$, where the polynomials f_2 and f_3 are from Ex. 5. The roots of these polynomials and the feasible region of $\mathcal{C}$ are depicted in Fig. 4(b). Our decision procedure could choose 0 as the first value for x, and end up in a state

$$\langle [\![x \mapsto 0, (f_2 < 0), (f_4 < 0), E \rightarrow (f_3 \leq 0)]\!], \mathcal{C} \rangle_2$$

We now need to compute the explanation E to explain the propagation. But, although the propagation was based on the inconsistency of $\mathcal{C}$ under M, we can pick the subset $\{f_2 < 0, f_3 > 0\}$ to produce the explanation. It is a smaller set, but sufficient, as it is also inconsistent with M. Doing so we reduced the number of polynomials we need to project, which, in CAD settings, is always an improvement.

5 Related Work and Experimental Results

In addition to CAD, a number of other procedures have been developed and implemented in working tools since the 1980s, including Weispfenning's method of virtual term substitution (VTS) [49] (as implemented in Reduce/Redlog), and the Harrison-McLaughlin proof producing version of the Cohen-Hörmander

method [32]. Abstract Partial Cylindrical Algebraic Decomposition [38] combines fast, sound but incomplete procedures with CAD. Tiwari [46] presents an approach using Gröbner bases and sign conditions to produce unsatisfiability witnesses for nonlinear constraints. Platzer, Quesel and Rümmer combine Gröbner bases with semidefinite programming [39] for the real Nullstellensatz.

In order to evaluate the new decision procedure we have implemented a new solver nlsat, the implementation being a clean translation of the decision procedure described in this paper. We compare the new solver to the following solvers that have been reported to perform reasonably well on fragments of nonlinear arithmetic: the z3 3.2 [14], cvc3 2.4.1 [2], and MiniSmt 0.3 [51] SMT solvers; the quantifier elimination based solvers Mathematica 8.0 [42,41], QEPCAD 1.65 [6], Redlog-CAD and Redlog-VTS [16]; and the interval based iSAT [17] solver.[7]

We ran all the solvers on several sets of benchmarks, where each benchmark set has particular characteristics that can be problematic for a non-linear solver. The meti-tarski benchmarks are proof obligations extracted from the Meti-Tarski project [1], where the constraints are of high degree and the polynomials represent approximations of the elementary real functions being analyzed. The keymaera benchmark set contains verification conditions from the Keymaera verification platform [39]. The zankl set of problems are the benchmarks from the QF_NRA category of the SMT-LIB library, with most problems originating from attempts to prove termination of term-rewrite systems [18]. We also have two crafted sets of benchmarks, the hong benchmarks, which are a parametrized generalization of the problem from [23], and the kissing problems that describe some classic kissing number problems, both sets containing instances of increasing dimensions.

Table 1. Experimental results.

	meti-tarski (1006)		keymaera (421)		zankl (166)		hong (20)		kissing (45)		all (1658)	
solver	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
nlsat	1002	343.48	420	5.73	89	234.57	10	170.33	13	95.62	1534	849.73
Mathematica	1006	796.90	420	171.96	50	366.10	9	208.04	6	29.01	1491	1572.01
QEPCAD	991	2616.94	368	1331.67	21	38.85	6	43.56	4	5.80	1390	4036.82
Redlog-VTS	847	28640.26	419	78.58	42	490.54	6	3.31	10	275.44	1324	29488.13
Redlog-CAD	848	21706.75	363	730.25	21	173.15	6	2.53	4	0.81	1242	22613.49
z3	266	83.18	379	1216.04	21	0.73	1	0.00	0	0.00	667	1299.95
iSAT	203	122.93	291	16.95	21	24.52	20	822.01	0	0.00	535	986.41
cvc3	150	13.52	361	5.45	12	3.11	0	0.00	0	0.00	523	22.08
MiniSmt	40	697.46	35	0.00	46	1370.14	0	0.00	18	44.67	139	2112.27

All tests were conducted on an Intel Pentium E2220 2.4 GHz processor, with individual runs limited to 2GB of memory and 900 seconds. The results of

[7] We ran the solvers with default settings, using the `Resolve` command of Mathematica, the `rlcad` command for Redlog-CAD, and the `rlqe` for Redlog-VTS.

our experimental evaluation are presented in Table 1. The rows are associated with the individual solvers, and columns separate the problem sets. For each problem set we write the number of problems that the solver managed to solve within the time limit, and the cumulative time for the solved problems. A plot of solver behavior with respect to solved problems is presented in Fig 5. All the benchmarks, with versions corresponding to the input languages of the solvers, the accompanying experimental data, are available from the authors website.[8]

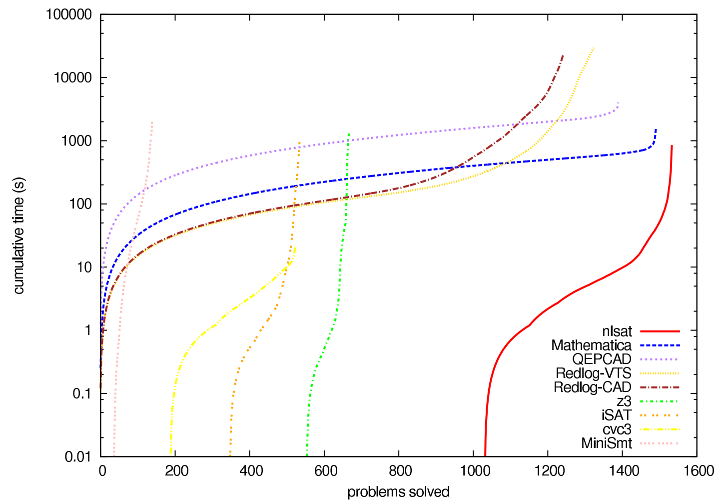

Fig. 5. Number of problems solved by each solver against the cumulative time of the solver (logarithmic time scale).

The results are both revealing and encouraging. On this set of benchmarks, except for nlsat and the quantifier elimination based solvers, all other solvers that we've tried have a niche problem set where they perform well (or reasonably well), whereas on others they perform poorly. The new nlsat solver, on the other hand, is consistently one of the best solvers for each problem set, with impressive running times, and, overall manages to solve the most problems, in much faster time.

6 Conclusion

We proposed a new procedure for solving systems of non-linear polynomial constraints. The new procedure performs a backtracking search for a model, where the backtracking is powered by a novel conflict resolution procedure. In our experiments, our first prototype was consistently one of the best solvers for each

[8] `http://cs.nyu.edu/~dejan/nonlinear/`

problem set we tried, and, overall manages to solve the most problems, in much faster time. We expect even better results after several missing optimizations in the core algorithms are implemented. For example, our implementation does yet support full factorization of multivariate polynomials, or algebraic number computations using extension fields.

We see many possible improvements and extensions to our procedure. We plan to design and experiment with different explain procedures. One possible idea is to try explain procedures that are more efficient, but do not guarantee termination. Heuristics for reordering variables and selecting a value from the feasible set should also be tried. Integrating our solver with a Simplex-based procedure is another promising possibility.

Acknowledgements. We would like to thank Grant Passmore for providing valuable feedback, the Meti-Tarski benchmark set, and so many interesting technical discussions. We also would like to thank Clark Barrett for all his support.

References

1. Behzad Akbarpour and Lawrence C. Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 44(3):175–205, 2010.
2. Clark Barrett and Cesare Tinelli. CVC3. In *Computer Aided Verification*, pages 298–302. Springer, 2007.
3. Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in real algebraic geometry*. Springer, 2006.
4. Christopher W. Brown. *Solution formula construction for truth invariant CAD's*. PhD thesis, University of Delaware, 1999.
5. Christopher W. Brown. Improved projection for cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 32(5):447–465, 2001.
6. Christopher W. Brown. QEPCAD B: a program for computing with semi-algebraic sets using CADs. *ACM SIGSAM Bulletin*, 37(4):97–108, 2003.
7. William S. Brown and Joseph F Traub. On Euclid's algorithm and the theory of subresultants. *Journal of the ACM*, 18(4):505–514, 1971.
8. Bruno Buchberger, George Edwin Collins, Rüdiger Loos, and Rudolf Albrecht, editors. *Computer algebra*. Symbolic and algebraic computation. Springer, 1982.
9. Bob F. Caviness and Jeremy R. Johnson, editors. *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Texts and Monographs in Symbolic Computation. Springer, 2004.
10. Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer Verlag, 1993.
11. George Edwin Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20–23, 1975*, pages 134–183. Springer, 1975.
12. George Edwin Collins and Hoon Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12(3):299–328, 1991.
13. Leonardo de Moura and Nikolaj Bjørner. Relevancy propagation. Technical Report MSR-TR-2007-140, Microsoft Research, 2007.

14. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.
15. René Descartes. *La Géométrie*. 1637.
16. Andreas Dolzmann and Thomas Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, 1997.
17. Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1(3-4):209–236, 2007.
18. Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. Sat solving for termination analysis with polynomial interpretations. *Theory and Applications of Satisfiability Testing*, pages 340–354, 2007.
19. Évariste Galois. Sur la théorie des nombres. *Bulletin des sciences mathématiques, physiques et chimiques*, 13:428–435, 1830.
20. Carl Friedrich Gauss. *Werke*, volume 2. Königlichen Gesellschaft der Wissenschaften, 1876.
21. Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for computer algebra*. Springer, 1992.
22. Hoon Hong. An improvement of the projection operator in cylindrical algebraic decomposition. In *Proceedings of the international symposium on Symbolic and algebraic computation*, pages 261–264. ACM, 1990.
23. Hoon Hong. Comparison of several decision algorithms for the existential theory of the reals. 1991.
24. Muḥammad ibn Mūsā al Khwārizmī. *Hisab al-jabr w'al-muqabala*. AD 820.
25. Qin Jiushao. *Mathemathical Treatise in Nine Chapters*. 1247.
26. Dejan Jovanović and Leonardo de Moura. Cutting to the chase: Solving linear integer arithmetic. In *Proceedings of the 23rd International Conference on Automated Deduction*, pages 338–353. Springer, 2011.
27. Donald Ervin Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison Wesley, third edition, 1997.
28. Konstantin Korovin, Nestan Tsiskaridze, and Andrei Voronkov. Conflict resolution. *Principles and Practice of Constraint Programming*, pages 509–523, 2009.
29. Sava Krstić and Amit Goel. Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. *Frontiers of Combining Systems*, pages 1–27, 2007.
30. Rüdiger Loos. Generalized polynomial remainder sequences. *Computer Algebra: Symbolic and Algebraic Computation*, pages 115–137, 1982.
31. Scott McCallum. *An improved projection operation for cylindrical algebraic decomposition*. PhD thesis, University of Wisconsin-Madison, 1984.
32. Sean McLaughlin and John Robert Harrison. A proof-producing decision procedure for real arithmetic. In *Proceedings of the 20th International Conference on Automated Deduction*, volume 3632, page 295. Springer, 2005.
33. Kenneth L. McMillan, Andreas Kuehlmann, and Mooly Sagiv. Generalizing DPLL to richer logics. In *Computer Aided Verification*, pages 462–476. Springer, 2009.
34. Bhubaneswar Mishra. *Algorithmic algebra*. Springer, 1993.
35. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.
36. Isaac Newton. *Arithmetica universalis: sive de compositione et resolutione arithmetica liber*. 1732.

37. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.
38. Grant O. Passmore. *Combined Decision Procedures for Nonlinear Arithmetics, Real and Complex*. PhD thesis, University of Edinburgh, 2011.
39. André Platzer, Jan-David Quesel, and Philipp Rümmer. Real world verification. In *Proceedings of the 22nd International Conference on Automated Deduction*, pages 485–501. Springer, 2009.
40. João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
41. Adam W. Strzeboński. Computing in the field of complex algebraic numbers. *Journal of Symbolic Computation*, 24(6):647–656, 1997.
42. Adam W. Strzeboński. Cylindrical algebraic decomposition using validated numerics. *Journal of Symbolic Computation*, 41(9):1021–1038, 2006.
43. Jacques Charles François Sturm. *Mémoire sur la résolution des équations numériques*. 1835.
44. Alfred Tarski. Über definierbare Mengen reeller Zahlen. *Annales de la Société Polonaise de Mathématique*, 9, 1930.
45. Alfred Tarski. A decision method for elementary algebra and geometry. Technical Report R-109, Rand Corporation, 1951.
46. Ashish Tiwari. An algebraic approach for the unsatisfiability of nonlinear constraints. In *Computer Science Logic*, pages 248–262. Springer, 2005.
47. Lou van den Dries. Alfred Tarski's elimination theory for real closed fields. *The Journal of Symbolic Logic*, 53(1):7–19, 1988.
48. Alexandre Joseph Hidulphe Vincent. Note sur la résolution des équations numériques. *Journal de Mathématiques Pures et Appliquées*, 1:341–372, 1836.
49. Volker Weispfenning. Quantifier elimination for real algebra - the quadratic case and beyond. *AAECC*, 8:85–101, 1993.
50. Wen-Tsun Wu. *Mathematics mechanization: Mechanical Geometry Theorem-Proving, Mechanical Geometry Problem-Solving, and Polynomial Equations-Solving*. Kluwer Academic Publishers, 2001.
51. Harald Zankl and Aart Middeldorp. Satisfiability of non-linear (ir)rational arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 481–500. Springer, 2010.

A Implementation Details

Acknowledging the importance that the details of a particular implementation play, in this section we describe which particular algorithms we used in our implementation, provide additional references, discuss alternatives, and analyze the impact of different optimizations we tried. Our procedure is based on several algorithms for manipulating polynomials and real algebraic numbers. Although most of the operations in these two modules have polynomial time complexity, they are the main bottleneck of our procedure. In our set of benchmarks, we identified two clear bottlenecks: the computation of *principal subresultant coefficients* (psc); and checking the sign of a multivariate polynomial in an irrational coordinate. In all benchmarks our prototype failed to solve, the computation was "stuck" in one of these two procedures.

A.1 Polynomials

We represent a multivariate polynomial using a sparse representation based on the sum-of-monomials normal form. Each monomial is basically a sorted vector of pairs: variable and degree. For example, the monomial $x_1^3 x_3^2 x_4$ is represented as $(\langle 1,3\rangle, \langle 3,2\rangle, \langle 4,1\rangle)$. A monomial also has a unique integer identifier that is used to implement mappings from monomials to values of some type T as vectors. We store monomials in a hash-table in order to have a unique reference for each monomial. A multivariate polynomial consists of two vectors: monomials and coefficients. The coefficients are arbitrary precision integers. The first monomial m in every non constant polynomial p always contain the maximal variable x in p, and $\mathsf{deg}(p, x) = \mathsf{deg}(m, x)$. Thus, the maximal variable of a polynomial and its degree can be quickly retrieved. Moreover, each polynomial has a unique integer identifier and a flag for marking whether the monomials are sorted using lexicographical order or not. Univariate polynomials are represented as a dense vector of coefficients. For example, the polynomial $2x^5 + 3x + 1$ is represented using the vector $(1, 3, 0, 0, 0, 2)$.

The polynomial arithmetic operations are implemented using the straightforward algorithms. Faster polynomial multiplication algorithms based on Fast-Fourier Transforms only outperforms the naïve algorithms for polynomials that are well beyond the current capabilities of our decision procedure. We use the standard polynomial pseudo-division algorithm ([10,27]). In many algorithms (GCD, resultant, psc), exact multivariate polynomial division is used. We say the division of a polynomial p by a polynomial q is exact when there is a polynomial h such that $p = qh$. We use the exact division algorithm described at [3] (Algorithm 8.6) . We implemented three different multivariate polynomial GCD algorithms: subresultant GCD (Chapter 3 [10]), Brown's modular GCD and Zippel's sparse modular probabilistic GCD (Chapter 7 [21]). Although the resultant of two polynomials is formally defined as the determinant of the Sylvester-Habicht Matrix, we used the algorithm based on polynomial pseudo-division, GCD and exact division described at [10] (Algorithm 3.3.7). We also implemented the principal subresultant coefficient algorithm in a similar fashion. The resultant of two polynomials can also be computed using modular techniques similar to the ones used to compute the GCD of two polynomials. However, we did not implement the modular resultant procedure yet. To the best of our knowledge, QEPCAD uses this modular algorithm to control the coefficient growth in the resultant computation.

Regarding polynomial factorization, we perform square-free factorization of a polynomial f using the GCD and its derivative with respect to some variable in f. The polynomial is then put into the form $\prod f_k^k$, where each f_k is the product of all factors of degree k. We extract content and primitive parts of a polynomial using the GCD and exact division. We use the standard approach for univariate factorization, where we first factor a square free polynomial in a finite field $GF(p)$ for some small prime p s.t. the factorization is also square free. Then, the factorization is lifted using Hensel's lemma, and finally we search for

factors in the result set of polynomials. Further details can be found in [10,27]. In the current prototype, we do not have support for full multivariate factorization.

We implemented two algorithms for root isolation of univariate polynomials with integer coefficients. One is based on Sturm sequences, and another on the Descartes' rule of signs [8]. In both cases, the computations are performed using *binary rational numbers* [9], also known as *dyadic rationals*. The ring $\mathbb{Z}[1/2]$ of binary rational numbers is the smallest subring of $\mathbb{Q}$ that contains $\mathbb{Z}$ and $1/2$. Binary rationals are rational numbers of the for $a/2^k$. $\mathbb{Z}[1/2]$ is not a field, but it is closed under division by 2. We represent binary rationals using an arbitrary precision integer for a, and a machine unsigned integer for k. The procedures for computing with binary rational numbers are more efficient than the equivalent ones for rational numbers.

A.2 Real algebraic numbers

In our implementation, a real algebraic number is a rational number or a square free polynomial f in $\mathbb{Z}[x]$ and an isolating interval of binary rational numbers. Moreover, zero is not a root of f, and the isolating interval does not contain zero. Several algorithms for manipulating algebraic numbers are greatly simplified when square free polynomials are used. Recall that a square free polynomial for f can be computed using $\mathsf{exactdiv}(f, \mathsf{gcd}(f, f'))$, where f' is the first derivative of f. The arithmetical operations $+$, $-$, $\times$, $/$ on algebraic numbers are implemented using resultants [10,34]. To evaluate the sign of a polynomial $p(x_1, \ldots, x_k)$ at $(\alpha_1, \ldots, \alpha_k)$, we first use interval arithmetic. If the result interval does not contain zero, we are done. Otherwise, we replace all rational α_i's, and try to use interval arithmetic again. We also refine the intervals of each irrational algebraic number until the result interval does not contain zero or the α_i's intervals have size less that $1/2^{32}$. If the result interval still contains zero, let us assume without loss of generality that none of the α_i's are rationals. Then, we compute

$$
\begin{aligned}
&R_1 = Res(y - p(x_1, \ldots, x_n), q_1, x_1) \\
&\ldots \\
&R_k = Res(R_{k-1}, q_k, x_k)
\end{aligned}
$$

where $Res(p, q, x)$ is the resultant of polynomials p and q with respect to variable x, and q_i is the defining polynomial for α_i. R_k is a polynomial in y, by resultant theory, $p(\alpha_1, \ldots, \alpha_k)$ is a root of R_k. Now, we compute a lower bound for the nonzero roots of R_k. This can be accomplished using the same algorithm used to compute a upper bound for the roots of a polynomial. We use the polynomial root upper bound algorithm described in [9]. Using this bound we can keep refining the α_i's intervals until the result interval for $p(\alpha_1, \ldots, \alpha_k)$ does not contain zero, or it is smaller than the lower bound for nonzero roots. In the second case, we have show that $p(\alpha_1, \ldots, \alpha_k)$ is zero.

For isolating the roots of $p(\alpha_1, \ldots, \alpha_k, y)$, we use a similar approach. We compute

$$\begin{aligned}
&R_1 = Res(p(x_1, \ldots, x_k, y), q_1, x_1) \\
&\ldots \\
&R_k = Res(R_{k-1}, q_k, x_k)
\end{aligned}$$

However, R_k vanishes if $p(x_1, \ldots, x_n, y)$ vanishes for some roots of $q_1, \ldots, q_k$. Example, $p(x_1, x_2, y) = x_1 y + x_2 y$, and $\alpha_1 = \alpha_2 = (x^2 - 2, (0, 2))$. That is, α_1 and α_2 are the $\sqrt{2}$. However, p vanishes for $p(\sqrt{2}, -\sqrt{2}, y)$. Thus, R_2 is the zero polynomial. To cope with this issue, we use a technique described in [41]. The basic idea is to use algebraic number arithmetic to evaluate the coefficients of p until we find one that does not vanish, or we prove that $p(\alpha_1, \ldots, \alpha_n, y)$ is the zero polynomial.

Finally, computation with algebraic numbers can be greatly improved if they are all elements of the extension field $\mathbb{Q}(\alpha)$, if we know the minimal polynomial for the algebraic number α. QEPCAD and Mathematica have support for extension fields. Moreover, given a set of algebraic numbers $\{\alpha_1, \ldots, \alpha_n\}$, there is a procedure for computing an algebraic number α s.t. $\alpha_1, \ldots, \alpha_n \in \mathbb{Q}(\alpha)$ [8,10]. Our prototype currently has no support for $\mathbb{Q}(\alpha)$.

A.3 Analysis

In this section, we analyze the impact of different algorithms and optimizations we tried. For that, we used an extended set of benchmarks containing 8928 problems. It was not computationally feasible to execute all other systems in this extended set. We remark that all benchmarks that our procedure could not solve or took more than one millisecond to solve are included in the results described in Section 5.

Benchmarks. The first observation is that most benchmarks can be solved with very few conflict resolution steps. Only 23 problems required more than 1000 conflict resolutions to be solved. The number of psc chain computations is also very small. Only 17 problems required more than 1000 psc computations. In our prototype, if possible we select a rational number in the rule Lift-Level. Therefore, many benchmarks can be solved without using any irrational algebraic number computation. Only 1826 benchmarks required irrational algebraic number computations to be solved.

Sparse modular GCD. The use of the sparse modular GCD algorithm instead of the subresultant GCD greatly improved the performance of our procedure. For 43 Meti-Tarski and Zankl benchmarks, we observed a two order of magnitude speedup.

Factorization. A standard technique used in CAD consists in factoring the polynomials obtained using the projection operator. If we disable factoring, 30 benchmarks from Meti-Tarski, Zankl and Hong families cannot be solved anymore, and another 18 benchmarks suffer from a two orders of magnitude slowdown. This suggests we may obtain even better performance results after we implement full multivariate polynomial factorization in our procedure.

Minimal polynomials. The minimal polynomial f of an algebraic number α is the unique irreducible polynomial of smallest degree with integer coefficients such that $f(\alpha) = 0$. Minimal polynomials are obtained using univariate polynomial factorization. Note that every minimal polynomial is square-free. By default, we use minimal polynomials for representing algebraic numbers. If we just use arbitrary square-free polynomials (that are not necessarily minimal) for encoding algebraic numbers, our procedure fails to solve 5 Meti-Tarski benchmarks, and suffers a two orders of magnitude slowdown in 12 other Meti-Tarski benchmarks.

Root isolation. By default, our procedure uses the Descartes' rule of signs procedure for isolating the roots of univariate polynomials. If we switch to a procedure based on Sturm sequences, the performance impact is negligible. Only one Meti-Tarski benchmark suffers from one order of magnitude slowdown.

Full dimensional. We say a problem is *full dimensional* if it contains only strict inequalities. A satisfiable full dimensional problem always has rational solutions. A standard optimization used in CAD-based procedures consists in ignoring sections when processing existential problems. This optimization is justified by the fact that in a full dimensional problem adding a constraint of the form $f \neq 0$, for some nonzero polynomial f, does not change the satisfiability of the problem. To the best of our knowledge, both QEPCAD and Mathematica use this optimization. We implement this approach in our procedure by simply using polynomial constraints of the form $y_k \leq_r \mathsf{root}(f, i)$ and $y_k \geq_r \mathsf{root}(g, j)$ instead of $y_k <_r \mathsf{root}(f, i)$ and $y_k <_r \mathsf{root}(g, j)$ when a problem is in the full dimensional fragment. With this optimization our prototype solved extra 12 problems.

Variable reordering. Variable order has a dramatic impact on CAD-based procedures. Mathematica uses heuristics for ordering variables, but we could not find any reference describing the actual heuristics used. We used a simple variable reordering heuristic, we compute the maximal degree maxdeg of each variable in the initial set of constraints. Then, before starting our procedure, we sort the variables using the total order $x_i \prec x_j$ iff $\mathsf{maxdeg}(x_i) > \mathsf{maxdeg}(x_j) \vee (\mathsf{maxdeg}(x_i) = \mathsf{maxdeg}(x_j) \wedge i < j)$. With this simple heuristic, our prototype can solve 54 (35 from the Meti-Tarski, and 15 from the Zankl set) problems that it could not solve. However, the heuristic also prevents our procedure from solving 3 (2 from Meti-Tarski, and 1 from the Zankl set) that could be solved without using it. These results suggest that further work should be invested in developing variable reordering techniques. Dynamically variable reordering during the search is also a promising possibility. However, to guarantee termination it should be eventually disabled.