Math. Struct. in Comp. Science (2007), *vol.* 17, *pp.* 99–127. © 2007 Cambridge University Press
doi:10.1017/S096012950600586X Printed in the United Kingdom

Implementing the cylindrical algebraic decomposition within the Coq system

ASSIA MAHBOUBI

INRIA Sophia Antipolis, 2004, routes des Lucioles – B.P. 93,
06902 Sophia Antipolis Cedex, France
Email: assia.mahboubi@sophia.inria.fr

Received 1 December 2005; revised 29 June 2006

The Coq system is a Curry–Howard based proof assistant. Therefore, it contains a full functional, strongly typed programming language, which can be used to enhance the system with powerful automation tools through the implementation of reflexive tactics. We present the implementation of a cylindrical algebraic decomposition algorithm within the Coq system, whose certification leads to a proof producing decision procedure for the first-order theory of real numbers.

1. Introduction

We present the development of a proof producing decision procedure for closed first-order formulas in the language of real numbers. This procedure is intended for the Coq system (The Coq Development Team 2004; Bertot and Casteran 2004), which is a proof assistant based on the Calculus of Inductive Constructions. The theory has been known to be decidable for classical real numbers since the early work of A. Tarski (Tarski 1948). However, most concrete problems remained beyond reach because of the complexity of the decision method he proposed (a tower of exponentials with length given by the number of variables involved).

An example of a problem that this kind of procedure can solve is if we are given a finite set of multivariate polynomials with n variables and a sign condition for each polynomial, does there exist a vector in $\mathbb{R}^n$ satisfying all these sign conditions simultaneously?

The potential uses for such a decision procedure are manifold in the formal verification of real analysis, where quantified inequalities are pervasive: $\epsilon - \delta$ cuttings, geometry with cartesian coordinates, constraints on constants for approximations and so on. For example, when reporting on their formalisation in the Isabelle proof assistant of the prime number theorem, the authors of Avigad *et al.* (2005) note the frustrating lack of automation in the theory of real numbers for non-linear problems, which imposes tedious work on steps that are usually left implicit in pen-and-paper proofs.

The following, which comes from a bunch of proof obligations for the termination of a rewriting order by polynomial interpretation, is an example of an irritating formula like this, and there is currently no tool for solving it automatically within the Coq system:

$$\forall x_1, v_0, v_1 \in \mathbb{R}, x_1 \geqslant v_1 + 1 \Rightarrow x_0 \geqslant v_0 \Rightarrow x_1 x_0 + x_1 > v_0 + v_1 v_0 + v_1.$$

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

Even an inefficient, but automatic, method for generating proofs would help formal methods developers focus on the mathematically relevant part of analysis proofs rather than tedious arithmetical lemmas.

The solution we propose consists of building a reflexive tactic, based on an implementation of the Cylindrical Algebraic Decomposition algorithm (CAD). This algorithm was introduced by G. E. Collins in 1974 (Collins 1975), was reprinted in Caviness and Johnson (1998), and has been studied extensively since then. It provides a dramatic improvement to the complexity compared with previously existing decision algorithms, which were just about as inefficient as Tarski's. With CAD, the complexity is polynomial in the maximum degree of the polynomials and doubly exponential in the number of variables. The algorithm has been studied extensively (Caviness and Johnson 1998; Collins and Hong 1991), and an optimised version is now a standard tool in computer algebra[†].

Unlike in a CAS, however, where one implements the algorithm in a standard programming language and accepts the results of the implementation without question, in a proof assistant we are obliged to prove the correctness of the algorithm. This is a rather heavy burden for an algorithm of this size and sophistication. Moreover, for technical reasons, some basic programming constructs such as exception handling and general recursion are missing from Coq's programming language. We thus chose to keep the implementation simple, basically following Collins' original plan, without the optimisations discovered in recent decades. This is perhaps not as serious a drawback as it may appear, since the problems one comes across in theorem proving are quite different from those for which one typically uses a CAS. In the former case, we are usually interested in automating steps of paper proofs, which are generally quite small (in terms of the number of symbols) but time-consuming to prove using purely linear methods. In the latter case, one is typically interested in large problems that go beyond the reach of reasonable paper calculation.

The implementation and certification of such an intricate algorithm within a proof assistant constitutes a large test-suite for evaluating the maturity of the proof assistant in terms of both computational efficiency and the proficiency for the mechanisation of computer algebra proofs. This is the first development of computer algebra of this scale developed in the Coq system, and it follows in the direction initiated by L. Théry in his formalisation of the Buchberger algorithm (Théry 2001).

While the design of the alogorithm was our principle goal, in doing so we had to provide several elementary units, such as a library of polynomial arithmetic, an encoding of algebraic numbers and, at the meta-level, an improved tool for managing equalities in ring structures (Grégoire and Mahboubi 2005). These parts of the development will enrich the system, and we hope that they may be reusable for other formal proofs.

This paper outlines the algorithms we have implemented for the decision procedure and reports on related formalisation issues. The second section introduces the theory of real closed fields and establishes the scope of the decision procedure. The third section describes the solution of univariate problems. The fourth section gives insights for the

[†] It is worth mentioning that due to the complexity of the algorithm, the CAD algorithm is not implemented in every computer algebra system. For instance, Mathematica provides an implementation of an optimised CAD algorithm, but the standard library of Maple deals only with univariate problems.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

general, multivariate case. Section 5 details a simple example of a CAD computation. Section 6 is a discussion of the constructive aspects of the method and the design of the reflexive tactic. Section 7 reports on the formalisation choices and the current state of the formal proofs. Section 8 summarises related work in computer algebra and formal proofs.

2. Real closed fields

2.1. *Real closed fields*

We work with a mathematical structure known as a *real closed field*.

Definition 2.1.1 (Real closed field). The following axioms define the theory of intuitionist real closed fields in the language $\mathcal{L} = \{+, \cdot, 0, 1, P\}$, where P is a unary predicate (interpreted as a positivity predicate):

1. The axioms for a commutative unit ring.
2. P(1).
3. $P(x) \vee P(-x) \Rightarrow x$ is invertible.
4. $\neg P(0)$.
5. $\neg P(-x) \wedge \neg P(x) \Rightarrow x = 0$.
6. $P(x) \wedge P(y) \Rightarrow P(x + y) \wedge P(x.y)$.
7. $P(x + y) \Rightarrow P(x) \vee P(y)$.
8. The axiom scheme of the intermediate value theorem for polynomials:

$$f(x) = y_0 x^n + y_1 x^{n-1} + \cdots + y_n$$
$$f(u) < 0 < f(v) \wedge u < v \Rightarrow \exists x, u < x < v \wedge f(x) = 0.$$

Above, and in the rest of the paper, we use $x < y$ to denote $P(y - x)$ and $x \leqslant y$ to denote $\neg P(x - y)$. We also introduce the symbol $>$ for convenience, with $x > y$ denoting $-x < -y$.

We extend this theory with the axiom stating that positivity is decidable, and this leads to the theory of discrete real closed fields.

9. $P(x) \vee \neg P(x)$.

E. Palmgren has shown (Palmgren 2002) that the addition of this latter axiom leads to the classical theory of real closed fields.

Theorem 2.1.1. The theory of discrete real closed fields is intuitionistically equivalent to the classical theory of real closed fields (that is, where the principle of excluded middle holds for every formula).

Constructive real numbers provide a model for the theory of real closed fields, but of course not for the discrete theory. We will discuss the use we make of the axiom of decidability of positivity in Section 6.

2.2. *Quantifier elimination*

Let R be a real closed field and D be a sub-ring of R. We will now define simultaneously the *formulas in the language of ordered fields with coefficients in D* and the set of *free variables* $Free(F)$ of a formula F.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

Definition 2.2.1 (Language of ordered fields with coefficients in D).

— An *atom* $P = 0$ or $P > 0$ or $P < 0$, where P is a polynomial in $D[X_1, \ldots, X_n]$ is a formula with free variables $\{X_1, \ldots, X_n\}$.
— If Φ_1 and Φ_2 are formulas, then $\Phi_1 \wedge \Phi_2$ and $\Phi_1 \vee \Phi_2$ are formulas with $\mathrm{Free}(\Phi_1 \wedge \Phi_2) = \mathrm{Free}(\Phi_1 \vee \Phi_2) = \mathrm{Free}(\Phi_1) \cup \mathrm{Free}(\Phi_2)$.
— If Φ is a formula, then $\neg\Phi$ is a formula and $\mathrm{Free}(\neg\Phi) = \mathrm{Free}(\Phi)$.
— If Φ is a formula and $X \in \mathrm{Free}(\Phi)$ then, $\exists X \Phi$ and $\forall X \Phi$ are formulas and $\mathrm{Free}(\exists x \Phi) = \mathrm{Free}(\forall x \Phi) = \mathrm{Free}(\Phi) \backslash \{X\}$.

We also need the following two definitions.

Definition 2.2.2 (Algebraic sets and semi-algebraic sets). Let $\mathcal{P}$ be a finite subset of $R[X_1, \ldots, X_n]$. The set of zeros of $\mathcal{P}$ is defined by $Z(\mathcal{P}) = \{x \in R^n | \bigwedge_{P \in \mathcal{P}} P(x) = 0\}$. Such a subset of R^k is called an *algebraic* subset of R^k.

The *semi-algebraic* subsets of R^k are the smallest family containing algebraic sets and sets defined by polynomial inequalities that is closed under complementation, finite union and finite intersection.

Definition 2.2.3 (Semi-algebraic functions). Let $S \subset R^k$ and $T \subset R^l$ be semi-algebraic sets. A function $f : S \to T$ is called a *semi-algebraic* function if its graph is a semi-algebraic subset of R^{k+l}.

The theory of discrete real closed fields admits quantifier elimination in the language of (discrete) ordered fields.

Theorem 2.2.1 (Quantifier elimination). Let $\Phi(Y)$ be a formula in the language of ordered fields with coefficients in an ordered ring D contained in the real closed field R. Then there exists a quantifier free formula $\Psi(Y)$ with coefficients in D such that for every $y \in R^k$, $\Psi(y)$ is true if and only if $\Phi(y)$ is true.

Proof. Quantifier elimination is a consequence of the stability of the family of algebraic subsets of R^k under projection. See, for example, Basu *et al.* (2003, page 60). □

Remark 2.2.1. In the rest of the paper, the (discrete) real closed field for which the decision procedure is built is assumed to be *archimedean*.

A decision procedure by cylindrical algebraic decomposition is also possible for the non-archimedean (discrete) real closed fields but the algorithms, using Thom encodings, are more intricate, and we are here mainly interested in providing a tool for the field of real numbers. Hence, from now on we will assume that this archimedean real closed field is the set of real numbers, denoted by $\mathbb{R}$.

To build a decision procedure from this theorem, we have to chose a ring D over which sign and equality are *computable*. In our work, the set of coefficients is $\mathbb{Q}$, the set of rational numbers.

3. Algorithm: univariate case

The algorithms we describe in the next two sections are extracted from Basu *et al.* (2003). We will only give an informal description of this architecture: see Basu *et al.* (2003) for details of the algorithm.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

3.1. *Sketch of the univariate decision procedure*

Given a list of polynomials $P_1, \ldots, P_k \in \mathbb{Q}[X]$, we consider the following system of inequalities:

$$P_1(x) \sharp_1 0 \wedge \cdots \wedge P_k(x) \sharp_k 0, \text{ where } \sharp_1, \ldots, \sharp_k \in \{<, =, >\}.$$

We want to know whether it is satisfiable or universally satisfied.

To answer this question, we will compute all the conjunctions of sign conditions that can be realised by the signs of the polynomials at each element of a complete sample of points. In fact, we will compute an *isolating list* for the family.

Definition 3.1.1 (Isolating list). Let $\mathcal{P} = P_1, \ldots, P_k$ be a finite list of polynomials in $\mathbb{Q}[X]$. Let $Roots_{\mathcal{P}}$ be the set of all the (real) roots of these polynomials:

$$Roots_{\mathcal{P}} := \{x \in \mathbb{R} \quad | \quad \exists i, P_i(x) = 0\}.$$

An *isolating list* for $\mathcal{P}$ is a list L of a subset of $\mathbb{R}$. Each element of L is either a singleton containing a rational point, or an open interval with rational endpoints. Each element of L contains exactly a single element of $Roots_{\mathcal{P}}$.

We will identify elements of L with the corresponding element of $Roots_{\mathcal{P}}$. For each element x of $Roots_{\mathcal{P}}$ (or equivalently of an isolating list), we define $IsNul_{\mathcal{P}}(x)$, the subset of $\mathcal{P}$ containing the polynomials that are null in x. The steps of the univariate decision procedure will be as follows:

1. Compute two rational points A and B, such that $Roots_{\mathcal{P}} \subset [A, B]$.
2. Compute an ordered isolating list for the family.
3. For each element x of the list, compute the sign of the polynomials in $\mathcal{P} \setminus IsNul_{\mathcal{P}}(x)$: if x is a point, evaluate the polynomials at that point, if x is an interval, evaluate the polynomials at the centre of the interval, which is a rational point.
4. Compute a rational point between every two consecutive elements of the isolating list. This is possible since the elements of the list are disjoint and have rational endpoints. Evaluate the polynomials at these rational points.

This produces vectors of sign conditions that constitute an exhaustive sign table for the family of polynomials, thanks to the intermediate value property and the correctness of the isolating list.

3.2. *Computation of isolating lists*

The first step of the computation is to restrict the problem to a finite interval of the real line. We use standard result on the bounds of the real roots of a polynomial, expressed as rational fractions of its coefficients.

Lemma 3.2.1 (Cauchy bounds on roots). Consider the polynomial

$$P = a_p X^p + \cdots + a_q X^q, \quad p > q, \quad a_p a_q \neq 0.$$

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

Let

$$C(P) = \sum_{q \leqslant i \leqslant p} \frac{|a_i|}{|a_p|} \quad \text{and} \quad c(P) = \sum_{q \leqslant i \leqslant p} \frac{|a_i|}{|a_q|}.$$

Then the absolute value of any non-zero root of P in $\mathbb{R}$ is smaller than $C(P)$ and bigger than $c(P)$.

Proof. See Lemmas 10.4 and 10.5 in Basu *et al.* (2003, page 323). □

We now have to compute information about the existence of roots for a polynomial on a given interval. This information is given by the number of sign changes in the coefficients of this polynomial in a special basis, known as the *Bernstein basis.*

Definition 3.2.1 (Bernstein polynomials). Let $c, d \in \mathbb{Q}$. The Bernstein polynomials of degree p for $c \neq d$ are

$$B_{p,i}(c,d) = \binom{p}{i} \frac{(X-c)^i (d-X)^{p-i}}{(d-c)^p} \quad 0 \leqslant i \leqslant p.$$

Theorem 3.2.1 (Bernstein basis). The Bernstein polynomials of degree p for c, d form a basis of the vector space of polynomials of degree $\leqslant p$.

Proof. Consider the polynomial

$$\frac{(X-c)^i (d-X)^{p-i}}{(d-c)^p}.$$

It is mapped to the polynomial X^i by the transformation

$$P(X) \mapsto P\left(\frac{X-c}{d-c}\right)$$

followed by

$$P(X) \mapsto X^p P(1/X).$$ □

The coefficients of a polynomial P is the Bernstein basis for (c, d), which gives us enough information to know when the interval contains exactly one root of P, thanks to the following property.

Theorem 3.2.2. Let $P \in \mathbb{Q}[X]$ of degree p, and $c < d \in \mathbb{Q}$. Let $V_b(P)$ denote the number of sign changes in the ordered sequence $b = (b_i)_{i=0\ldots p}$, where b_i is value of the $B_{p,i}(c,d)$-component of P in the Bernstein basis $B_p(c,d)$.

The following property holds:

— $V_b(P) = 0$ iff P has no root on $]c, d[$.
— If $V_b(P) = 1$, then $P(c)P(d) < 0$ iff P has exactly one root on $]c, d[$.
— Otherwise, anything can happen.

Proof. The proof is quite elementary if the domain interval is $]0, +\infty[$ and the basis is the canonical one $1, \ldots X^p$ rather than Bernstein. To get the Bernstein version, we use the transformation of Lemma 3.2.1, which maps the domain $]c, d[$ to $]0, +\infty[$. □

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

With Theorem 3.2.2, we can check when a candidate interval can be included in the isolating list for a single polynomial.

If the first case applies, we know that the interval we are looking at is not relevant.

If the second case applies, we can put this interval in the isolating list for the polynomial.

If the final case applies, we have to split the interval $]c, d[$ and work on each half, initiating a dichotomy process. We can ensure the termination of the dichotomy process thanks to the following property.

Theorem 3.2.3. If $P \in \mathbb{Q}[X]$ has either no root or exactly one simple root in $]c, d[$, and P has no complex root in $\mathcal{C}(c, d-c) \cup \mathcal{C}(d, d-c)$, where $\mathcal{C}(x, r)$ is the complex disk of centre $x \in \mathbb{R}$ and radius r:

— $V_b(P) = 0$ iff P has no root on $]c, d[$.
— $V_b(P) = 1$ iff $P(c)P(d) < 0$, in this case P has exactly one root on $]c, d[$.

Proof. See Theorem 10.39 in Basu *et al.* (2003, page 339), using Descartes' law of signs. □

Theorem 3.2.3 ensures that if we refine the initial interval enough, until the associated circle no longer contains any of the polynomial's complex roots, we reach a situation where $V_b(P)$ is 0 or 1.

Note that each time we refine an interval by splitting it into two parts, we need to recompute the coefficients of the polynomial in the new Bernstein basis associated with the new rational endpoints. There is an efficient algorithm for computing the two new lists of coefficients, one for each half, given the initial coefficients, the initial endpoints and the point at which we will split. It is linear in degree and bit-size of the coefficients: see Algorithm 10.31 in Basu *et al.* (2003, page 333).

3.3. *Description of the algorithm*

We will describe in detail the computation of an isolating list for a family of two polynomials P_1 and P_2. We use $Bern(P, a, c)$ to denote the call to the function computing the number of sign changes in the list of coefficients of P in the Bernstein basis of degree $deg(P)$ for a and b. In order to distinguish an interval contaning a common root for P_1 and P_2 from an interval containing one root for P_1 and one distinct root for P_2, which must be separated, we will need to use the gcd of P_1 and P_2.

Without loss of generality, we can suppose that P_1 and P_2 are square-free, since the square-free part of a polynomial P is easily computed as $gcd(P, P')$. We are indeed only interested in the existence and relative positions of roots, not in their multiplicity.

1 Compute the bounds for the roots of P_1 and P_2, and then take the extrema A and B to get bounds on the roots of the family.
2 Compute an isolating list L_{P_1} for P_1 on $]A, B[$ using the following algorithm.
To compute an isolating list L_{P_1} for P_1 on an open interval $]a, b[$:
— If $Bern(P_1, a, b) = 0$, then $L_{P_1} = \{\}$.
— If $Bern(P_1, a, b) = 1$, then $L_{P_1} = \{]a, b[\}$.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

— Otherwise:

(a) Compute L'_{P_1}, an isolating list of P_1 on $]a, \frac{a+b}{2}[$.

(b) Compute L''_{P_1}, an isolating list of P_1 on $]\frac{a+b}{2}, b[$.

(c) If $P_1(\frac{a+b}{2}) = 0$, then:

– $L_{P_1} = L'_{P_1} \cup L''_{P_1} \cup \{[\frac{a+b}{2}]\}$.

Otherwise:

– $L_{P_1} = L'_{P_1} \cup L''_{P_1}$

3 If $L_{P_1} = \{\}$:

— Compute an isolating list for P_2 on $]a, b[$.

Otherwise:

— For each element u of L_{P_1}, compute the sign of P_2 on u as follows:

– If u is a single rational point, evaluate P_2.

– If $u =]c, d[$, then:

• If $Bern(P_2, c, d) = 0$, evaluate P_2 at $\frac{c+d}{2}$.

• If $Bern(P_2, c, d) = 1$, we have to decide whether P_1 and P_2 have a common root between c and d or not. Let G be the greatest common divisor of P_1 and P_2.

- If $Bern(G, c, d) = 1$, the root is common.

- If $Bern(G, c, d) = 0$, refine u to get an interval (or a single point) u' such that P_2 has no root in u' by splitting u' up to the point at which there is no sign change in the sequence of Bernstein coefficients of P_2.

- If $Bern(G, c, d) > 0$, refine u to get an interval (or a single point) falling into one of the previous cases by splitting u and testing its middle and the two open halves.

We have now built a refined isolating list L'_{P_1} for P_1, such that each element of L'_{P_1} is a root of P_2 or an interval over which P_2 has a constant sign, which has been computed.

4 For each interval between two consecutive elements of L'_{P_1}, compute an isolating list for P_2 and the sign of P_1 on these intervals. Merge these lists with L'_{P_1} to get L_{P_1,P_2}, which is an isolating list for the family.

5 Compute the sign of P_1 and P_2 at A and B.

6 Compute the sign of P_1 and P_2 between two consecutive elements of L_{P_1,P_2}, by evaluating them at the middle of the endpoints. If $L_{P_1,P_2} = \emptyset$, evaluate them at 0.

Remark 3.3.1. This is just an outline of the method, and the implemented algorithm contains a few improvements for efficiency. For instance, we share the computations of signs and gcd's.

Remark 3.3.2. It is crucial that we work with an archimedean real closed field for these dichotomy processes to terminate.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

4. Algorithm: multivariate case

The multivariate case is treated using the Cylindrical Algebraic Decomposition (CAD) algorithm. This algorithm is too sophisticated to be described completely in this paper. We will only try to provide an informal account of its salient points and just give the details for a small example. Further information on CAD can be found in the abundant literature on the topic: see, for example, Caviness and Johnson (1998) and Basu *et al.* (2003). The algorithm we have implemented is the archimedean version found in Basu *et al.* (2003, Chapter 12).

4.1. *The multivariate problem*

Given a finite list $\mathcal{P}$ of polynomials $P_1, \ldots, P_k \in \mathbb{Q}[X_1, \ldots, X_n]$, we say that a finite partition of $\mathbb{R}$ is *adapted* to the family $\mathcal{P}$ if on every cell of this partition, the sign of each element of $\mathcal{P}$ is invariant.

The CAD algorithm computes such a partition, in fact, a sample set of points for an adapted partition, realising all the vectors of sign conditions that can be obtained by the evaluation of the sign of $P_1, \ldots, P_k$ at a point $\overline{x} \in \mathbb{R}^n$.

Organising the result as an unstructured list of sign conditions is not well-adapted for our purposes. For example, the formula $\exists x \forall y P(x, y) > 0$ holds if $P(X, Y) = XY$ and does not hold if $P(X, Y) = X + Y$, but the sign conditions realisable by these two polynomials are the same. We need to organise this list as a *tree*, reflecting the structure of the partition.

We proceed by successive elimination of the variables:

$$\mathbb{R}[X_1, \ldots, X_n] \qquad\qquad \mathbb{R}[X_1, \ldots, X_{n-1}]$$

$$\mathcal{P} = P_1, \ldots, P_s \xrightarrow[\text{phase}]{\text{elimination}} \mathcal{Q} = Q_1, \ldots, Q_t$$

$$\text{CAD and signs for } \mathcal{P} \xleftarrow[\text{phase}]{\text{lifting}} \text{CAD and signs for } \mathcal{Q}$$

$$\mathbb{R}^n \qquad\qquad \mathbb{R}^{n-1}$$

$\mathcal{Q}$ is a, possibly larger, new family of polynomials, where X_n has been eliminated. The *elimination phase* of the algorithm computes successively the families $C_i(\mathcal{P}) \subset \mathbb{Q}[X_1, \ldots, X_i]$ for $i = n \ldots 1$, starting from $C_n(\mathcal{P}) = \mathcal{P}$. The partition corresponding to $C_1(\mathcal{P})$ is an isolating list as presented in the previous section. We consider this list as the labelled leaves of a tree with depth 1.

4.2. *The elimination phase*

The elimination phase for X_n applies a *projection operator* to the family $\mathcal{P}$, and this outputs a new family $\mathcal{Q}$ of polynomials describing the expected partition on the $n-1$ first

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

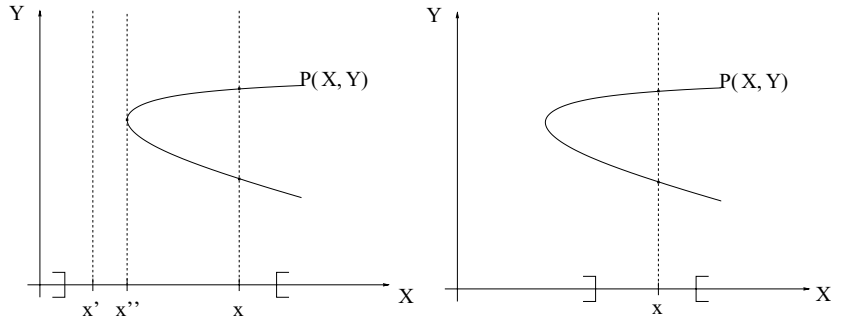


Fig. 1. On the left, the interval cannot be a cell of level one: $P(x, Y)$ has two roots, $P(x', Y)$ has no root, $P(x'', Y)$ has a single root. On the right it can be a cell.

coordinates. More precisely, if we are given a certain partition $\mathcal{S}_{n-1}$ of $\mathbb{R}^{n-1}$ adapted to the new family $\mathcal{Q}$, we should be able to deduce a partition $\mathcal{S}_n$ adapted to the initial family $\mathcal{P}$. A cell $C_{\mathcal{P}} \subset \mathbb{R}^n$ of the new partition will be obtained as follows:

1 Take a cell $D_{\mathcal{Q}} \subset \mathbb{R}^{n-1}$ from the decomposition adapted to $\mathcal{Q}$.
2 Build the cylinder $D_{\mathcal{Q}} \times \mathbb{R} \subset \mathbb{R}^n$.
3 Cut it in 'a polynomial way' with respect to the last coordinate, that is to say:
   — take the intersection of this cylinder with the graph of a semi-algebraic function (see Definition 2.2.3); or
   — take the semi-cylinder above (respectively, under) the graph of a semi-algebraic function; or
   — take a part of the cylinder delimited by the graph of two semi-algebraic functions f and g such that $f < g : D_{\mathcal{Q}} \to \mathbb{R}$.

The elements of $\mathcal{S}_n$ are called the cells of level n and the set of successive partitions $\mathcal{S}_n, \ldots, \mathcal{S}_1$ is called a *cylindrical decomposition* of $\mathbb{R}^n$.

In fact, pieces of cylinders will be delimited by intersections of graphs of algebraic functions defined implicitly by

$$P(x_1, \ldots, x_{n-1}, f(x_1, \ldots, x_{n-1})) = 0 \qquad (*)$$

where P is a polynomial in $\mathcal{P}$ and $x_1, \ldots, x_{n-1} \in \mathbb{R}^{n-1}$ is a vector belonging to $D_{\mathcal{Q}}$.

We should be able to infer the behaviour of any polynomial P in $\mathcal{P}$ by studying the behaviour of $P(a_1, \ldots, a_{n-1}, X)$ $(a_1, \ldots, a_{n-1})$, which is an arbitrary element of $D_{\mathcal{Q}}$. This sample point $(a_1, \ldots, a_{n-1}) \in D_{\mathcal{Q}}$ will be the one computed by the recursive application of the algorithm to $\mathcal{Q}$.

To validate this arbitrary choice of a point in $D_{\mathcal{Q}}$, the behaviour of all the polynomials in $\mathcal{P}$ should be uniform on every cell of level $n-1$ with respect to the $n-1$ first coordinates (see Figure 1). For example, a real line $\overline{x} \times \mathbb{R}$ should meet the algebraic curve defined by P at the same number k of points, whatever point $\overline{x}$ we choose in the same cell $D_{\mathcal{Q}}$. Then the implicit equation $(*)$ would lead to the definition of k semi-algebraic functions over $D_{\mathcal{Q}}$.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

The following are sufficient conditions to ensure this uniformity:

— Every polynomial in $\mathcal{P}$ should remain in the same shape, that is, leading coefficients do not vanish. This means, for example, that the same cell for $Elim_Y(\mathcal{P})$ should not contain both 0 and 1 if $\mathcal{P}$ contains $XY - 1$, otherwise, for this cell, the polynomial could either be $xY - 1$ or -1.

— Roots of polynomials in $\mathcal{P}$ should be defined over a cell of level $n-1$ as a *function* of the $n-1$ first coordinates, according to the implicit equation $(*)$. This phenomenon is the one illustrated on Figure 1.

These two elimination properties in $\mathcal{P}$ should, furthermore, be guaranteed only by the fact that polynomials in $\mathcal{Q}$ have a constant sign over the cell.

To ensure the first condition, we need to supplement the family $\mathcal{P}$ with the set of *truncations* of its polynomials, where non-constant leading coefficients are progressively removed. For instance, if $\mathcal{P}$ contains $X^2Y^3 + (X+1)Y^2 + Y + 1$, the polynomials $(X+1)Y^2 + Y + 1$ and $Y + 1$ should be added to the family, but not the constant polynomial 1 since the leading coefficient of $Y + 1$ cannot vanish. At the same time, the non-constant leading coefficients are added in $\mathcal{Q}$. In our example, these coefficients are X^2 and $X + 1$.

To ensure the second condition, all the available methods rely on successive partial derivatives of the polynomials, which is a computationally intensive process. We have chosen to implement the *subresultant elimination* (see Basu *et al.* (2003, page 422)). Subresultant coefficients of a pair of polynomials P, Q are minors of the Sylvester matrix of P and Q, hence they are computed as polynomials of the coefficients of P and Q.

Subresultant coefficients and polynomials have been extensively studied, see, for example, von zur Gathen and Lücking (2003), and there is a wide range of possible definitions, inducing various computation algorithms. For a detailed description of the algorithms we have implemented for the computation of subresultants, again refer to Basu *et al.* (2003, Algorithms 8.73 and 8.75, starting on page 279). The property we use here is that fixing the signs of the subresultant coefficients of P and Q will fix the degree of $gcd(P, Q)$.

4.3. *The lifting phase*

During the lifting phase, we focus on the polynomials of $\mathcal{P}$ *above* the elements of the sample set of points describing the cells of the partition adapted to $\mathcal{Q}$. These points are elements of $\hat{\mathbb{Q}}^{n-1}$, where $\hat{\mathbb{Q}}$ is the set of real algebraic numbers. On the real line, we encoded elements of $\hat{\mathbb{Q}}$ as couples of an open interval and a polynomial. Here this encoding generalises to *isolating parallelepipeds.*

Definition 4.3.1 (Isolating parallelepiped). A parallelepiped isolating

$$z = (z_1, \ldots, z_k) \in \mathbb{R}^k$$

is a list

$$(\mathcal{T}_1, I_1), \ldots (\mathcal{T}_k, I_k)$$

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

where:

— $\mathcal{T}_1 \in \mathbb{Q}[X_1], \ldots \mathcal{T}_k \in \mathbb{Q}[X_1, \ldots, X_k]$.
— I_1 is an open interval with rational endpoints or a rational singleton containing the root z_1 of $\mathcal{T}_1$ and no other root of $\mathcal{T}_1$.
— I_2 is an open interval with rational endpoints or a rational singleton containing the root z_2 of $\mathcal{T}_2(z_1, X_2)$ and no other root of $\mathcal{T}_2(z_1, X_2)$.
— ...
— I_k is an open interval with rational endpoints or a rational singleton containing the root z_k of $\mathcal{T}_k(z_1, \ldots, z_{k-1}, X_k)$ and no other root of $\mathcal{T}_k(z_1, \ldots, z_{k-1}, X_k)$.

The output tree for the family $\mathcal{Q}$ is a tree of depth $n-1$ whose leaves are labelled with isolating parallelepipeds of dimension $n-1$. For each of these leaves, labelled with an isolating parallelepiped p encoding the algebraic vector $\overline{x}$, the lifting phase from level $n-1$ to level n will focus on the univariate polynomials with algebraic coefficients $P(\overline{x}, X_n)$, for each $P \in \mathcal{P}$. This will produce an isolating list, and we will replace the leaf labelled with p by the list of isolating parallelepipeds obtained by gluing each element of the isolating list on top of p.

We still need to justify the fact that we are able to apply the univariate method to polynomials $P(\overline{x}, X)$, where $\overline{x}$ is an algebraic vector given by an isolating parallelepiped. The coefficients of such a $P(\overline{x}, X)$ are polynomials (with rational coefficients) in the coordinates of $\overline{x}$. The only requirement on the ring of coefficients for applying the isolation algorithm described in Section 3 is that we can compute the sign of any element of this ring. This means we should be able to compute the sign of a polynomial in $\mathbb{Q}[X_1, \ldots, X_{n-1}]$ evaluated at $\overline{x}$.

There are two arguments showing that we are able to do this. First, we know how to compute the sign of a univariate polynomial at a root of an other one (see Section 3.3). Then, we know how to compute this sign using only the *coefficients* of the first polynomial (see Section 3.2). It is hence possible to isolate the roots of a $P(\overline{x}, X_n)$ with $P \in C_n(\mathcal{P})$ using recursive calls to the univariate procedure.

The univariate procedure also involves gcd computations that should now become gcd computations between univariate polynomials with algebraic coefficients. Since all the polynomials we are handling are of the form $Q(\overline{x}, X_n)$, for a fixed $\overline{x}$, the problem is to compute gcds between elements of $\mathbb{Q}[X_1, \ldots, X_{n-1}][X_n]$. There is a well-known solution to the problem of polynomial gcd computation for polynomials with coefficients in a ring: it is again the subresultant algorithm.

5. Example

In this section we will illustrate the computations using a simple example. Here we consider the family with only one polynomial $\mathcal{P} := \{X^2 + Y^2 - 1\} \subset \mathbb{Q}[X, Y]$, whose set of roots is the unit circle in the plane.

First, we compute the successive families of polynomials. Here there are only two such families, as two variables are involved. Here the elimination phase computes one

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

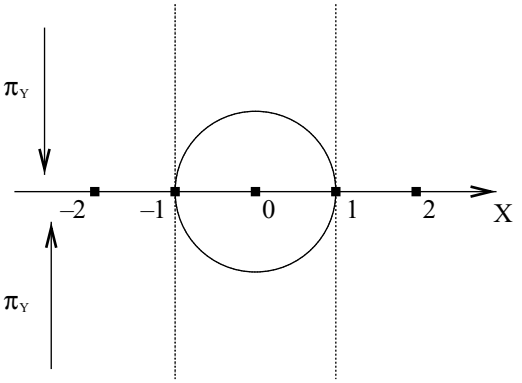


Fig. 2. Cells of level 1 for the unit circle

subresultant coefficient, coming from the subresultant chain from the polynomial and its partial derivative. No truncation is needed.

— $\mathcal{C}_2(\mathcal{P}) = \{X^2 + Y^2 - 1\}$
— $\mathcal{C}_1(\mathcal{P}) = \{X^2 - 1\}$.

Now we begin the lifting phase. The sample of points for cells of level 1 will consist of 5 points: the two roots of $X^2 - 1$ and one point in each interval they determine. Let us first suppose that the dichotomy process has allowed us to find the exact value of the roots -1 as a single point and isolated 1 by the interval $]\frac{1}{2}, \frac{3}{2}[$. This is not realistic given the symmetry of the problem, but it will enable us to study the two possible configurations.

The isolating list, at level 1, which is labelled with the sign of the polynomial in $C_1(\mathcal{P})$ can be

$$(\{-2\}, +),\ (\{-1\}, 0),\ (\{0\}, -),\ (]\ \frac{1}{2}, \frac{3}{2}\ [, X^2 - 1), 0),\ (\{2\}, +),$$

which can be represented as the following sign table:

	$-2 \in]-\infty, -1[$	-1	$0 \in]-1, 1[$	$]\frac{1}{2}, \frac{3}{4}[$ isolating 1	$2 \in]1, \infty[$
$X^2 - 1$	+	0	−	0	+

Note that these cells of level one are what we called parallelepipeds (of dimension one). Now we have to lift in order to build cells of level 2. We will consider the cells of level 1 one after the other, and for each one consider the polynomial in $C_2(\mathcal{P})$ on the real line above the sample point.

— Above -2: $P(-2, Y) = Y^2 + 3$ has a constant positive sign.
  A real line above any point of this cell will never meet the circle, and the half plane $\{(x, y) \mid x < -1, y \in \mathbb{R}\}$ is a single cell.
— Above 2.
  Similar considerations mean the half plane $\{(x, y) \mid x > 1, y \in \mathbb{R}\}$ is a single cell.
— Above -1: $P(-1, Y) = Y^2$ has a single root 0.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

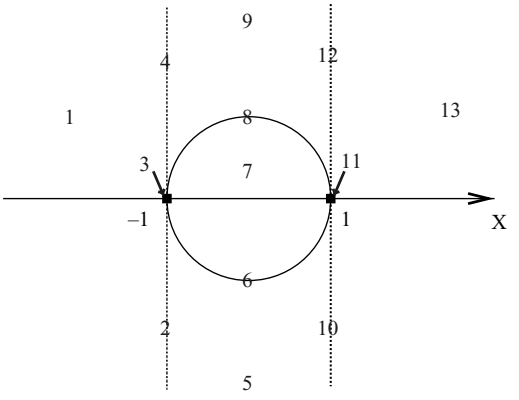


Fig. 3. Cells of level 2 for the unit circle

There are three cells of level 2 above this cell of level 1: two half-lines $\{(x, y) \mid x = -1, y < 0\}$ and $\{(x, y) \mid x = -1, y > 0\}$ and one reduced to a single point $\{ (-1, 0) \}$. Check this line meets the circle at a single point.

— Above 1.
As in the previous case but replacing -1 by 1. However, the computations are different, since the algorithm does not know the exact value of the root (which is, in fact, 1). We have to consider $P(z, Y) = Y^2 + (z^2 + 1)$ where z is the only root of $X^2 + 1$. For this we have to compute the coefficients of $P(z, Y)$ in various Bernstein bases, and then count the sign changes. But coefficients of $P(z, Y)$ will be polynomials in X evaluated at z. Computing the sign of such a coefficient will just be computing the sign of a polynomial at the isolated root z of $X^2 - 1$. This can be done using the univariate algorithm.

— Above 0: $P(0, Y) = Y^2 - 1$.
As in the case of the cell $\{-1\}$, we find five cells of level 2: $\{(x, y) \mid y < \sqrt{1 - x^2}\}$, $\{(x, y) \mid y > \sqrt{1 - x^2}\}$, $\{(x, y) \mid y = \sqrt{1 - x^2}\}$, $\{(x, y) \mid y = -\sqrt{1 - x^2}\}$.

In all, we obtain 13 cells, as described in Figure 3.

Remark 5.0.1. Notice that this adapted partition is not the smallest possible in terms of the number of cells (three cells, consisting of the inner disk, the circle and the area outside the circle, would have been sufficient). However, it is still the smallest possible cylindrical algebraic decomposition, though this is not always the case: the cylindrical algebraic decomposition computed by the algorithm may not be the most concise one. This can occur, for example, if the problem is not symmetric and the order of variable elimination is chosen badly.

A tree-like sample of points that can be output by the algorithm is represented in Figure 4. For the sake of readability, not all the leaves are represented, and inner nodes are labelled with the sign tables found during the intermediate computations.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

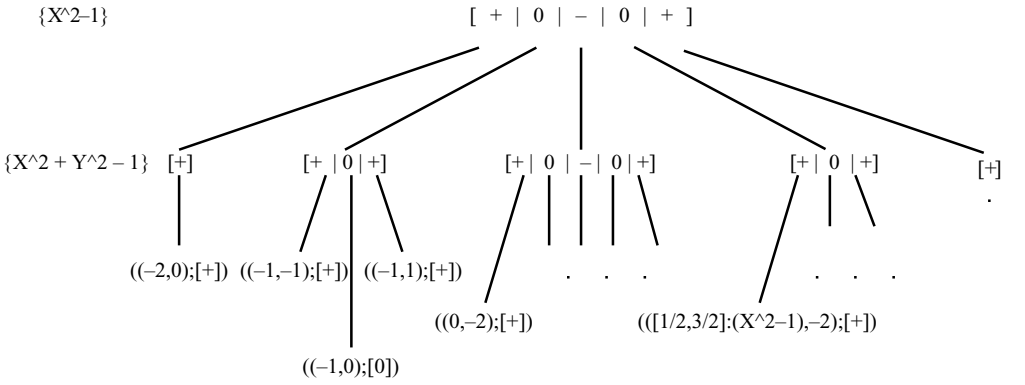


Fig. 4. Sign tree for the cad of $X^2 + Y^2 - 1$

6. Reasoning about constructive reals

In this section we discuss the constructive aspects of this decision procedure and present the architecture of a *reflexive* tactic based on the CAD algorithm.

6.1. *A 'semi-classical' decision procedure*

If we run the CAD-algorithm on a family $\mathcal{P}$ of polynomials in $\mathbb{Q}[X_1, \ldots, X_n]$, we will get a tree, each leaf of which is labelled with a real (algebraic) point, and a list of the sign conditions satisfied by the elements of $\mathcal{P}$ at that point.

We could describe this output tree by a (possibly huge) formula. For the example given in Section 5, with $P(X, Y) = X^2 + Y^2 - 1$, the formula would be

$$
\begin{aligned}
&\exists x_1, \exists x_2, \ \forall x, \\
&\qquad \{x < x_1 \ \Rightarrow \ \forall y \\
&\qquad\qquad P(x, y) > 0\} \\
&\wedge \quad \{x = x1 \ \Rightarrow \ \exists y_1 \forall y \\
&\qquad\qquad y < y_1 \Rightarrow \quad P(x, y) > 0 \\
&\qquad\qquad y = y_1 \Rightarrow \quad P(x, y) = 0 \\
&\qquad\qquad y > y_1 \Rightarrow \quad P(x, y) > 0\} \\
&\wedge \quad \ldots
\end{aligned}
$$

We will *not* build this in the system as a Coq lemma since the tactic is based on the two-levels approach described in Sections 6.2 and 6.3. However, this formula does help us understand the logical information output by the CAD algorithm. This information is already sufficient for proving constructively a subset of first-order formulas involving the elements of P: for example, valid existential formulas for which the decision procedure computes a witness.

However, to express the fact that the partition is exhaustive, we need the trichotomy axiom, without which no valid universally quantified formula can be deduced from such a sign table tree description: the trichotomy axiom cannot be deduced from the sign table of X.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

Here we note that the use of the classical axiom is confined to the interpretation of the description of the output tree. In fact, as the description formula is proved exhaustive in the classical theory, we can say that the first-order formulas of the theory of intuitionist real closed fields that are validated by this decision procedure are the ones that classically have no counter-example.

The Coq proof assistant (The Coq Development Team 2004) is based on the Calculus of Inductive Constructions, which is a type theory in which we can choose between two sorts for first-class citizens: *Set* and *Prop*. One characteristic of the *Prop* sort is that the typing rules will never allow the user to define an object lying in the *Set* sort by case analysis on an object in the *Prop* sort. The informal meaning of this is that the *Set* sort contains the computational part of a development, which can be extracted to a running functional program by forgetting about the logical part of the development, which is placed in the *Prop* sort.

It is possible to define a semi-classical theory of real closed fields, by putting all the axioms of Section 2.1 in *Prop*, including the trichotomy axiom. With these axioms, we can reason about the sign of a real, which can only take three values, but the type-checker (which is also the proof-assistant proof-checker) will prevent the user from trying pattern matching on the sign of a real number: objects in *Prop* can only be eliminated to build objects that are also in *Prop*. Hence the sign of an element of such a real closed field remains uncomputable in general. In this context, it is possible to build a tactic that will give a constructive proof of a given first-order formula of the theory if it is possible, just as it would have deduced it from the description formula, and if it fails, will use the *non-computational* trichotomy to give a proof in this theory.

6.2. *Scheme of the reflexive tactic*

There are several ways of building a tactic using this CAD-tree output to prove an arbitrary closed formula of the first-order theory of real numbers. A first possibility would be to generate automatically the formula describing the computed partition, which would be a Coq proposition, together with a proof of it. Then we would deduce the formula the user was interested in from this description lemma.

Our choice was to make the most of the computational abilities of the system to reduce the size of the proof object and speed up proof checking by using the so-called *refletive* (also called *two-level*) approach for building the tactic. Proofs by reflection, which were introduced by Allen *et al.* (1990), are designed according to the following scheme:

— Let $P : A \to Prop$ be a predicate over a set A.
— Suppose that we are able to write in the system a semi-decision procedure f such that f is computable and if f returns *true* on the entry x, then $P(x)$ is valid, that is to say

$$\mathsf{f_correct} : \forall x, f(x) = true \Rightarrow P(x).$$

If we want to prove $P(y)$ for a particular y, and if we know that $f(y)$ *reduces* to *true*, then we can simply apply the lemma f_correct to y and to a proof that $true = true$. Thanks to the following conversion rule, we can replace the type of a term with any type that is

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

```
Inductive Atom : Set :=
  | Pos  : Pol ⇒ Atom
  | Neg  : Pol ⇒ Atom
  | Zero : Pol ⇒ Atom.

Inductive Form : Set :=
  | Aatom : Atom ⇒ Form
  | Anot  : Form ⇒ Form
  | Aand  : Form ⇒ Form ⇒ Form
  | Aor   : Form ⇒ Form ⇒ Form
  | Aall  : Form ⇒ Form
  | Aex   : Form ⇒ Form.
```

Fig. 5. Inductive definition of abstract formulas

convertible modulo the reduction rules of the calculus:

$$\frac{\Gamma \vdash t : T \qquad T \equiv U}{\Gamma \vdash t : U}.$$

In Coq, the standard proof of $true = true$, is written `(refl_equal true)`. It is also implicitly a proof that $f(y) = true$ because $f(y)$ reduces to $true$, so $true = true$ is convertible with $f(y) = true$. Finally, the proof of $P(y)$ we have built is

```
f_correct y (refl_equal true).
```

The size of such a proof only depends on the size of the particular argument y and does not depend on the number of implicit reduction steps.

We would like this correctness lemma to apply to any goal that is a closed first-order formula in the theory of real numbers. In the Coq type system, these first-class citizens are in the *Prop* sort and pattern matching on these terms is not possible within the system: we cannot discriminate the formulas and apply a decision procedure to them.

Hence, we define *inductively* an abstract representation of first-order formulas in the theory of real numbers. This inductive type will describe the abstract syntax of the *concrete* formulas our tactic should decide, and pattern matching will be possible on elements of this type. Abstract formulas can be defined as in Figure 5.

We also define the intuitive interpretation $\phi : Form \to Prop$ of an abstract real closed formula to a concrete proposition (a Coq term in the sort *Prop*) by simple case analysis on the structure of an abstract real formula. Each abstract atom is mapped to its concrete version via the evaluation of polynomials at a list of real values, and abstract connectives and quantifiers are mapped to their images in *Prop*.

We would now like a decision procedure f satisfying

$$\mathsf{f_correct}\colon \forall \overline{F}, f(\overline{F}) = true \Rightarrow \phi(\overline{F}),$$

where $\overline{F}$ ranges over the type of abstract real formula.

When this decision procedure f is programmed, to prove an arbitrary *concrete* formula F of the real theory, we will guess an abstract version of it, namely an abstract formula $\overline{F}$ such as $\phi(\overline{F}) = F$. This step is called the *reification* step, and again it cannot be programmed within the theory because we would need pattern-matching on arbitrary Coq terms. However, it is easy to implement it in the meta-language of Coq, as an oracle.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

This reification only consists of reflecting in `Form` the syntax of a formula, and this mapping can be implemented in the top-level metalanguage Ltac (Delahaye 2002; Bertot and Casteran 2004) provided by the Coq system. We can prove by a straightforward induction that this interpretation ϕ is complete, but we do not need to prove formally any requirement on ϕ within the system, since its possible defects will only affect the completeness of the tactic, and not the proof of the correctness lemma.

6.3. *Decision and correctness*

We have still to explain how to implement the decision procedure f on abstract formulas and how f_correct will be proved.

If $\mathcal{P}$ is a finite subset of $\mathbb{Q}[X_1, \ldots, X_n]$, we use $\mathcal{P}$-*atom* to mean a formula of the form $P = 0$ or $P > 0$ or $P < 0$, where P belongs to $\mathcal{P}$. A $\mathcal{P}$-*formula* is then a first-order formula whose atoms are $\mathcal{P}$-atoms. These definitions can be transposed to their abstract counterpart. If F is the (concrete) formula we want to decide, the first step is to collect the smallest finite family $\mathcal{P}$ such that F is a $\mathcal{P}$-formula and to build its reification $\overline{F}$, which is a $\mathcal{P}$ abstract formula.

Suppose that $\overline{F}$ is a $\mathcal{P}$ abstract formula, and that we have already computed a CAD-tree for the family $\mathcal{P}$. For the bare decision process, let us temporarily consider for the sake of clarity a simplified output t from this algorithm. A leaf of t is labelled with the column of the signs respectively realised by the elements of $\mathcal{P}$ at the point of the sample represented by this leaf:

```
simpl_output := Leaf of sign_column | Node of list simpl_output.
```

Figure 6 shows the way to decide $\overline{F}$ using t by induction on the structure of $\overline{F}$, where we have replaced the constructors of Figure 5 by their logical notation for the sake of readability.

The failures in this algorithm correspond to inconsistencies between the CAD-tree and the formula. This inconsistency should never occur, since the depth of the tree t is the number of nested quantifiers in $\overline{F}$, and because we start with a closed formula.

To summarise, the f function applied to an abstract closed formula $\overline{F}$ will:

— Compute $\mathcal{P}$ such that $\overline{F}$ is a $\mathcal{P}$ abstract formula.
— Compute the CAD-tree t associated with $\mathcal{P}$.
— Apply the algorithm of Figure 6.

The core of the tactic will be to prove the correctness lemma associated with the decision procedure described above. This work is still in progress, and we present here our current approach to organising this proof. It is divided into two main parts.

The first step deals with the correctness of the computations we have performed. Rather than computing a sample set of points with sign conditions as in Figure 4, we need to remember conditions on the successive coordinates, and to share these conditions between the various branches.

In the `CAD_output` data-type depicted in Figure 7, the labels of inner nodes are position requirements, together with a witness, to ease existential proofs. The labels of the leaves

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

```
Cad_decision F t :=
Match F with:
— atom P♯0, (with ♯ ∈ {>, <, =}):
    if t = Leaf c then
     if c entails P♯0 then return true
     else return false
    else fail
— F1 ∧ F2 : return (Cad_decision F2 t) & (Cad_decision F1 t)
— F1 ∨ F2 : return (Cad_decision F2 t) or (Cad_decision F1 t)
— ¬F : return not (Cad_decision F t)
— ∀x, F :
    if t = Node t1 ... tn then
    return (Cad_decision F t1)& ... & (Cad_decision F tn)
    else fail
— ∃x, F :
    if t = Node t1 ... tn then
    return (Cad_decision F t1) or ... or (Cad_decision F tn)
    else fail
```

Fig. 6. Decision procedure for abstract formulas from CAD-trees

```
Inductive Condition : Set :=
  | Minfty  : Alg ⇒ Condition
  | Pinfty  : Alg ⇒ Condition
  | Root    : Alg ⇒ Condition
  | Interval : Alg ⇒ Alg ⇒ Condition.

Inductive CAD_output : Set :=
  | Leaf  : sign_column ⇒ CAD_output
  | Node  : Condition ⇒ R ⇒ list CAD_output ⇒ CAD_output.
```

Fig. 7. Representation in Coq of the CAD-tree output

are conclusions about the sign of the elements of the family. A path in the tree can hence be read as a list of conditions on the positions of the variables under which a $\mathcal{P}$-atom is valid.

Returning to our previous example, the output tree described in Figure 4 for the input family $\{X^2 + Y^2 - 1\}$ is in fact the one represented in Figure 8.
For example, the second path, starting from the left is described by the formula

$$\forall x, x = x_1 \Rightarrow \forall y, y < y1 \Rightarrow P(x, y) > 0.$$

We will need to prove that on every input (being a finite family of polynomials), the sign conditions labelling each leaf of the output tree is correct under the conditions specified along the path leading to this leaf. This first step of the proof of f_correct includes correctness proofs for all the algorithms we have implemented.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

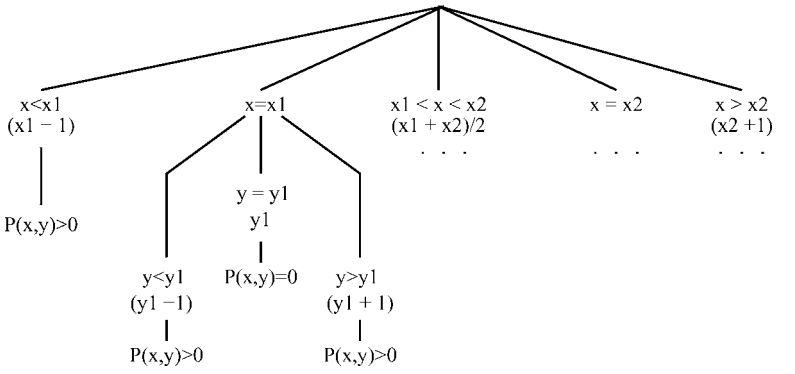


Fig. 8. Annotated tree for the CAD of $X^2 + Y^2 - 1$

The proof is an induction on the depth of the output tree (or equivalently on the number of variables) and uses two kinds of arguments.

— First, we need to guarantee that it is fair to choose one sample point in each interval computed along each coordinate, because the behaviour of polynomials in these intervals is uniform. At level one, this follows from the theorem of intermediate values. In higher dimensions, we will use the properties of the projection operator, which was designed for this purpose, and the continuity of roots: see Basu *et al.* (2003, pages 143–145).
— The second kind of proof is the correctness of the signs computed by the algorithm, which is obtained from variants of Descartes's law of signs (*c.f.* Theorem 3.2.2).

The second step of the proof of f_correct is to prove the correctness of the search algorithm. We need to establish that if the Cad_decision procedure computes the output true on an abstract formula $\overline{F}$, then we are able to construct a Coq proof term for $\phi(\overline{F})$. We again insist on the fact that, since we have chosen a reflexive method, we prove the correctness lemma once, and the proof of $\phi(\overline{F})$ is never constructed as such, but the tactic will only *reduce* (Cad_decision $\overline{F}$ t) to *true*.

This correctness proof is an induction on the structure of an abstract real formula, while deconstructing the associated CAD tree and remembering the path leading to the current CAD tree. We need first to prove that the depth of the annotated CAD tree is the depth of quantification of the closed abstract $\mathcal{P}$-formula, which is straightforward. Then the only non-trivial cases are the quantified ones. The property on the depth of the tree ensures that the CAD tree given in the input is not a leaf.

When we are dealing with an existential formula $\exists x\overline{F}$, we have to produce a witness x for which we can prove $\overline{F}$. But the induction hypothesis and the fact that the result of the computation is true means that one of the branches resulting from the current node is successful. Following this successful branch, the next child will be labelled with the witness we are looking for. In other words, for each existential quantifier we cross, we go down in the tree, collecting the witnesses for which the final atom will hold.

In the case where we are dealing with a universal formula $\forall x\overline{F}$, we need to use the exhaustiveness of our description. Here we need to apply subdivision lemmas generalising

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

trichotomy to ensure that gathering the proofs from the subtrees resulting from the current node is sufficient. More precisely, either the node has a single child, or it has $2n+1$ nodes (intervals and points of a n points partition), with $n > 0$, and we apply the lemma

$$\forall x_1, \ldots, x_n, \forall x, [x < x_1] \vee [x = x_1] \vee [x_1 < x < x_2] \vee \cdots \vee [x_n < x].$$

These lemmas (and their proofs) for an arbitrary n can be produced in a generic way. We can then reason by case analysis on the relative position of the universally quantified variable x with respect to $x_1 \ldots x_n$.

Finally, the provability of atoms is exactly what we proved in the first step, that is to say, the correctness of the CAD tree computation.

7. Formalisation issues

In addition to the usual trade-off between efficient implementations and tractable formal proofs, programming such an algorithm demands a special care in the implementation of the basic objects and operations: the complexity of the algorithm is already doubly exponential in the number of variables, even if it is 'only' polynomial in the number of polynomials and maximum degree.

This is all the more true when we are programming within a system like Coq, in which computations are made using the reduction machine of the system, where even machine integers are not available (we will discuss this point further in Subsection 7.3). In this section, we present our choices for representing the basic objects handled by the algorithm: polynomials and rational numbers.

7.1. *Implementation of multivariate polynomials*

The set $D[X_1, \ldots, X_n]$ of multivariate polynomials with coefficients in a set D is built recursively using univariate polynomials, thanks to the isomorphism $D[X_1, \ldots, X_n] \equiv D[X_1, \ldots, X_{n-1}][X_n]$.

Univariate polynomials over a set of coefficients D are represented in *sparse Horner form*.

Definition 7.1.1 (sparse Horner form). A polynomial in sparse Horner form is:

— a constant polynomial; or
— of the form $PX^i + c$, where $c \in D$ is a constant, $i \in \mathbb{N}^*$ is called the power index, and P is a polynomial in sparse Horner form.

Figure 9 shows how this inductive definition can be declared in the Coq system.

The set of polynomials with n (ordered) variables is represented as a dependent type (depending on the number n of variables), which is the application of `Pol1` to the set of polynomials with n variables.

Remark 7.1.1. The power index is used to avoid holes in the representation of polynomial when there are jumps in degree between monomials.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

```
Inductive Pol1 (D:Set) : Set :=
  | Pc  : D ⇒ Pol1 D (* Constant *)
  | PX  : Pol1 D ⇒ ℕ+ ⇒ D ⇒ Pol1 D. (* Non-constant *)
```

Fig. 9. Representation of polynomials

Yet another kind of hole arises with the recursive representation of multivariate polynomials: holes in variables. For example, the representation of X_1 as an element of $D[X_1]\dots[X_{100}]$ will be a term of depth 100, first nesting 99 constructors of constant polynomials `Pc`, then building `PX 1 1 0`. This could be avoided by dropping the idea of a type depending on the number of variables and adding an extra constructor to inject a polynomial in $D[X_1 \dots X_n]$ into any $D[X_1 \dots X_m]$ with $n < m$, *c.f.* Grégoire and Mahboubi (2005), but the proofs become more cumbersome, while the complexity of the CAD algorithm limits the number of variables.

In our data structure, there are several representations for the same polynomial. For a given polynomial, we define the normal form as the representation with no leading zero and maximal power indices. Normal forms are more compact than other representations and it is worthwhile making sure that polynomials are always in normal form.

Instead of using a normalisation function, which would introduce an extra cost to all the operations, we always assume that the arguments of the operations are in normal form and program operations so that they maintain this invariant.

To this end, we use a function `mkPX`, which can be seen as an intelligent constructor, that builds a polynomial $PX^i + c$ in normal form when given as its arguments P a polynomial in normal form, i a power index and c a constant. This intelligent constructor is used to perform cheap local normalisations when needed.

The set of coefficients that we use in our polynomials must be integral domains: rings with a partial division operation. In turn, we can provide a division operation for the univariate polynomials, thus making the set of polynomials an integral domain too.

7.2. *Formal proofs*

The development of a certified library of algorithms like the one we provide for polynomials is governed by (at least) two elementary general principles.

The first is self-evident: programming style imposes proof-style. It is not surprising that 'formal debugging' imposes a strong discipline on the programmer just to make their work tractable. Even though the primitive status of computation in Coq results in a functional typed programming language, certain natural constructions, like nested recursion, are harmful to the certification. Isolating and abstracting smaller pieces of code, which are easier to specify, before combining them always seems like a good strategy.

Another stumbling block is the treatment of partial functions, which remains a delicate issue in type theory. For example, we need to define a polynomial Euclidean division for coefficients in a ring. Our policy is to work systematically with total functions, and to postpone the definition of the domains in the specifications of the functions. Hence the Euclidean division on polynomials is defined everywhere, even if the divisor is zero or the

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

divisibility does not hold (this may happen when coefficients are in a ring). The default value is simply set at zero for inputs outside the domain of specification. This choice has also been made in similar developments in HOL-Light and Isabelle.

The second principle may have a stronger impact on the formal proofs: one should in general always try to prove naive algorithms on simple representations first, and then enrich the proofs with possible optimisations. This principle was applied at several steps of the development.

In particular, one of the most far-reaching choices for the proofs is the choice of the equality relation for our formalisation of polynomials. In this section the term (univariate) *polynomial* will be used to mean a finite list of ordered coefficients. Remember that in our formalisation, a polynomial may have different representations, so it does not make much sense to reason using the syntactic equality, which is also called Leibniz equality, on these objects.

This is an instance of the more general problem of reasoning modulo an equivalence relation, in this case where classes are the set of equivalent representations of a polynomial. We have considered three ways of addressing this problem.

The first possible solution is to choose a representative of each class, the normal form, and define equality of terms as the syntactic equality of these canonical representatives. This equality is easy to define. First we need to implement the normalisation function *norm* for polynomials, since up to now we have only used local normalisations. This normalisation function can be seen as a choice operator picking up a representative from each class. This is already unsatisfactory since we have carefully avoided the need to implement and use such a normalisation function. In addition, the proofs are more difficult and the propositions involving them become awkward, comprising numerous hypotheses stating that the polynomials involved are in normal form. So we first need to prove the correctness of the normalisation function: it should produce normal forms, that is to say polynomials in Horner form comprising only non-zero coefficients. We also need to ensure that it is compatible with the operations and to verify that the subset of polynomials in normal form satisfy the ring properties.

For example, proving the commutativity of addition finally leads to proving that $\forall A\ B, norm(A) + norm(B) = norm(B) + norm(A)$. This solution seems to be convenient when classes are finite and the choice operator is easy to define, but we believe that this solution was not appropriate to our formalisation.

Another possibility is to work within the quotient structure, and to reason modulo the equivalence relation using Coq's structure of setoids (Barthe *et al.* 2003). A setoid is a domain with a declared equivalence relation. Once operations on this domain have been proved compatible with the equivalence relation, Coq will automate and hide from the user the application of the appropriate compatibility lemma during rewriting, thereby facilitating proof development. Such compatible operations are called *morphisms*, and rewriting under this operations should be transparent to the user at the proof-script level. This is the solution we have adopted, and our motivation was the recent enhancement of setoid rewriting in Coq (Sacerdoti 2006). Because setoid reasoning in Coq is a rather new feature, this solution has some limitations. For example, the search for occurrences is rather inefficient, and it is difficult to rewrite just at a specific occurrence. Even in the

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

case of a single occurrence, the search sometimes fails, forcing the user to enumerate the compatibility and transitivity lemmas, which were supposed to be computed automatically by the setoid tactic. We hope these problems will be addressed in later versions of Coq.

The last idea would be to introduce an alternative representation of polynomials with naive ring operators and a syntactic equality. This alternative representation would be dedicated to proofs and it would be sufficient to prove that our optimised representation is an instance of the proof-oriented representation. This means that we need to prove that our optimisations do not change the represented mathematical object. After that, we would state and prove statements concerning the proof-oriented polynomials. This would be another illustration of our second principle.

A good choice for such a formalisation would be a list of coefficients, or even a stream of coefficients together with an integer representing the degree of the polynomial. With this last solution, normalisation is unnecessary This method was suggested by G. Gonthier but has not been thoroughly tested. It seems to us that the difficulty of the proofs may be comparable in the last two approaches. We think, however, that using a proof-dedicated representation may be easier and above all more satisfactory since it provides an abstraction that could be reused for proving any other implementation of polynomials.

At the time we are reporting on this work, the ring operations of the library, plus the specifications of some extra operations like powers, have been completely proved, using an abstract structure of coefficients and a setoid equality. This is a joint work with L. Rideau. The specifications of the Euclidean division are being proved by L. Pottier. Taking these specifications as parameters, we have formalised a large part of the correctness proof of the subresultant algorithm (Mahboubi 2006), which is a challenge in its own right.

We are working together with Y. Bertot to design the proof of properties of Bernstein polynomials, and we hope to have the complete set of correctness proofs for building a tactic for the univariate problems in the near future.

Most of the proofs for the univariate case will be reusable for the multivariate case: for example, the proofs for the polynomial arithmetic are parameterised by an abstracted ring of coefficients. Nonetheless, the correctness of the projection operator and the arguments based on continuity of roots will undoubtedly demand a significant amount of additional work to be properly stated and proved in Coq.

7.3. *Computing with rational numbers*

We have chosen to work with an abstract description of rational numbers for this development. We describe the extended signature needed for such an implementation of rational numbers, and let the users choose which one they want to compute with. Thus we were able to test the efficiency of the computations according to the choices made for the operations over rational numbers.

For example, if we choose to represent rational numbers as pairs, $\mathbb{Z} \times \mathbb{N}^*$, should we always normalise the fractions (this would lead to numerous gcd computations) or wait until the bit-size of the numerator or denominators reach a certain size?

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

Table 1. *Times in sec. on input* $\{(X-1)\dots(X-2n);(X-n)\dots(X-3n)\}$

Parameter n	**2**	**3**	**4**	**5**
Coq' s rational numbers	2	98	-	-
Efficient rationals	0	2	17	81

Rational numbers as fractions are implemented in Coq using binary integers. This means that computations with Coq's reduction machine are made using basis 2. This is far less efficient than machine integers, as used in computer algebra systems. The Coq system does not provide machine integers, but we have experimented with the work of L. Théry and B. Grégoire, who developed a functional modular library of arithmetic, including efficient algorithms for large integers in Coq (Grégoire and Théry 2005). B. Grégoire has extended this library to get an implementation of rational numbers (several, in fact, which use different normalisation strategies), which we plugged into our procedure. The performance improvement is significant and promising. In Table 1, we compare the execution of the algorithm computing the CAD for a univariate problem with two polynomials. Both polynomials have $2n$ roots, of which n are common roots. The size of the coefficients in the intermediate polynomials increases quickly during computations. We compare the execution using Coq's naive implementation and B. Grégoire's rational numbers, normalising systematically after each operation on rationals. For both implementations, our algorithm is much faster with this strategy.

Testing a non-systematic strategy is a difficult issue. If we never normalise, even with the efficient implementations, for the example in Table 1 with $n = 4$, computations take several minutes. We would like to test the alternative normalisation strategy 'if integers are too large', but it is difficult to find the relevant value for the authorised size and to estimate the consequences on complexity.

Another implementation that could be efficient would be rational numbers with lazy computations. In fact, each time we evaluate polynomials at some rational number, we are never interested in the exact value of the result, but only in its sign. This should be a typical case for the application of a lazy arithmetic of rational numbers, *c.f.* the Stern-Brocot trees implemented in Coq by M. Niqui, (Niqui and Bertot 2003).

We have not been able to test the lazy arithmetic described in Niqui and Bertot (2003) yet because one primitive function remains to be implemented in order for the rational signature to be completed, but we hope we will be able to test this approach very soon.

8. Related work and performances

There have been several experiments involving connecting computer algebra systems and proof assistants, for example, HOL and Maple (Harrison and Théry 1998) and Coq and Maple (Delahaye and Mayero 2005). Researchers from the computer algebra community have also investigated the possibility of building a checker on top of a computer algebra system: the Redlog system (Dolzmann and Sturm 1997) has been implemented on top of Reduce, and the Theorema system comes as an extension of Mathematica.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

However, the converse approach, consisting of importing the technology of computer algebra systems within proof assistants by programming (and proving) efficient algorithms within proof assistants has not been widely studied. Until recently, proving algorithm correctness with proof assistants was difficult. The first example of related work is probably the formalisation of the Buchberger algorithm, which was achieved by L. Théry (Théry 2001). In this development, the algorithm and its correctness proof are completely formalised in Coq and the code can be extracted and executed in Objective Caml.

With the introduction of a compiler (Grégoire and Leroy 2002), the Coq system has become a tool for efficient computation. Now it is worth trying a computational approach in formal proofs. In the case of the algorithm we are presenting, the use of a linked computer algebra system would be of little help: trusting a skillful oracle is interesting if result searching is difficult and result checking fast, but this is not the case for the present decision problem. On the other hand, it would be interesting to investigate what heuristics, which could use specialised and external tools, could be put on top of our tactic.

The CAD algorithm is both inefficient and complex. In fact, computer algebra systems do not always implement it. For example, Maple only provides a procedure allowing the user to isolate the roots of univariate polynomials, called `realroot`. A multivariate implementation has been developed in the Axiom system (Bronstein *et al.* 2004) by R. Rioboo, but the code is deprecated.

To our knowledge, quantifier elimination over real numbers by CAD has only been implemented in Mathematica (Strzebonski 2004), Redlog and Reduce (Dolzmann and Sturm 1997; Dolzmann *et al.* 1998). These systems integrate alternative methods for small problems and heuristics to simplify entries before starting the elimination process. Moreover, there is also an expert system dedicated to the CAD method, called QEPCAD(B) (Brown 2004; Brown 2003; Caviness and Johnson 1998). QEPCAD B offers numerous features enabling the exploitation of all the information given by a CAD computation, and can optimise both the computation and the size of the outputs.

It is not easy to compare the performance of our system with other implementations: for example, Mathematica implements a very powerful heuristic before its CAD procedure, which solves instantly all the examples we have tried. We have not been able to find a way to unplug the heuristic in order to access the bare CAD implementation.

It is also difficult to construct a relevant benchmark for such an algorithm, since its efficiency depends critically on the geometrical meaning of the input, rather than on more quantifiable factors. C. Brown claims in Brown (2003) that:

> 'Performance is an issue that has not been discussed in this paper. (...) there is no good characterisation of what kind of problems QEPCAD B solves quickly in practice. Experimental comparisons with other programs and methods of quantifier elimination or formula simplification would comprise another paper entirely.'

It seems to us that the efficiency is still reasonable enough to meet the demands of the user of a proof assistant. The example given in Avigad *et al.* (2005) is solved by our prototype within 12 minutes. However, a formula describing the Feuerbach Theorem, coming from the Theorema team, involving two variables, having a total degree of 8, and solved instantly by Mathematica (using its heuristic and not the CAD), was not solved after 48 hours of computations.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

As far as we know, the only proof-producing decision procedure currently available in another proof assistant is the one designed by S. McLaughlin and J. Harrison in the HOL-Light system (McLaughlin and Harrison 2005). A few years ago, we built another prototype of such a decision tactic for real numbers, using the same method as McLaughlin and Harrison. This work is described in Mahboubi and Pottier (2002). This algorithm is perhaps the most elementary in the literature, it uses only repeated applications of the intermediate value theorem, but its complexity is as bad as Tarski's. This Coq tool used an external oracle programmed in Objective Caml, and proved the generated traces of the computations. Each description formula of a partition that was computed was then proved on demand, and this was highly time-consuming, and we never produced a formal proof of the correctness of the the algorithm.

9. Conclusion

At the present time we have completely implemented the general, multivariate algorithm within the Coq system. At all times we have tried to program in an efficient way, in order to be able to test significant examples, but without including many optimisations. We think that it is already a significant result that it is possible to implement such an intricate computer algebra algorithm within a proof assistant, and to be able to run it on non-trivial examples.

Because of the numerous ingredients needed for this procedure, this already provides a Coq toy-computer algebra system for symbolic computations over polynomials. Designing the CAD algorithm with the Coq language (more than 4000 lines of instructions for the pure computational part) was not an easy task for many reasons. From a purely conceptual point of view, there are inherent difficulties, such as the computational complexity, relying on a recursion on the dimension of the problem, and manipulating algebraic numbers. Limitations of the Coq system provide yet another set of difficulties, including the peculiarities of programming in Coq, current limitations of the module system and idiomatic recursive programming.

Our implementation is modular, parameterised by the implementation of rational numbers. Thanks to this abstraction, we can test various possibilities and rely on powerful available rational arithmetics. The proof assistant's module system is appropriate for this purpose.

The building of the correctness proofs is ongoing, collaborative work. Most of the basic operators (ring operations and other basic manipulations with multivariate polynomials) have already been proved correct, and this already provides some automation of the proofs in later theories (automatic proving of ring identities over polynomials, rewriting modulo the equivalence relation on polynomial representation...). The real root isolation algorithm is being proved by Y. Bertot. The author is working with L. Théry and L. Rideau on the correctness proof of the subresultant algorithm. When these proofs are complete, we will have enough to prove the correctness of the univariate case.

The next challenge will be the formal proof of the projection operator needed for the correctness of the multivariate case.

The complexity of the algorithm is a limiting factor for the tactic efficiency, and it may be interesting to link such a tactic with some heuristics, trying first to decide first-order

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

formulas on real numbers using heuristics such as interval analysis, decomposition into sums of squares, and so on. Even as it stands, the computation times are already quite satisfactory for problems with a 'rather small' number of variables and 'rather small' degree. Indeed, the tactic computes much faster (a few minutes for many problems of degree and number of variables less than 4) than an expert user composing the formal proof by hand.

We think our implementation of CAD is a powerful tool for automating proofs in real analysis, and in addition serves as a large scale test-suite for studying how easily computer algebra techniques may be imported into the context of a proof assistant such as Coq.

Acknowledgments

I would like to thank the anonymous referees for their advice and many suggestions for improving this paper. I would also like to thank Yves Bertot for his careful proof-reading and advice.

References

Allen, S. F., Constable, R. L., Constable, L., Howe, D. J. and Aitken, W. (1990) The Semantics of Reflected Proof. *Proceedings of the 5th Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society Press.

Avigad, J., Donnelly, K., Gray, D. and Raff, P. (2005) A Formally Verified Proof of the Prime Number Theorem. To appear in the ACM Transactions on Computational Logic. (Available at `http://arxiv.org/abs/cs.AI/0509025`.)

Barthe, G., Capretta, V. and Pons, O. (2003) Setoids in Type Theory. *Journal of Functional Programming* **13** (2) 261–293.

Basu, S., Pollack, R. and Roy, M. (2003) Algorithms in Real Algebraic Geometry. *Algorithms and Computation in Mathematics* **10**, Springer-Verlag. (Draft of the second version available at `http://name.math.univ-rennes1.fr/marie-francoise.roy/bpr-posted1.html`.)

Bertot, Y. and Casteran, P. (2004) *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, an EATCS Series, Springer-Verlag.

Bronstein, M. *et al.* (2004) AXIOM – The Scientific Computation System. (Homepage: `http://axiom.axiom-developer.org/axiom-website/community.html`.)

Brown, C. W. (2004) QEPCAD. (Homepage: `http://www.cs.usna.edu/~qepcad/B/QEPCAD.html`.)

Brown, C. W. (2003) QEPCAD B: a Program for Computing with Semi-Algebraic Sets using CADs. *SIGSAM Bull.* **37** (4) 97–108.

Caviness, B. and J. Johnson (eds.) (1998) *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Texts and monographs in symbolic computation, Springer-Verlag.

Collins, G. E. (1975) Quantifier Elimination for the Elementary Theory of Real Closed Fields by Cylindrical Algebraic Decomposition. *Springer-Verlag Lecture Notes in Computer Science* **33** 134–183.

Collins, G. E. and Hong, H. (1991) Partial Cylindrical Algebraic Decomposition for Quantifier Elimination. *Journal of Symbolic Computation* **12** (3) 299–328.

Delahaye, D. (2002) A Proof Dedicated Meta-Language. In: Proceedings of Logical Frameworks and Meta-Languages (LFM), Copenhagen (Denmark). *Electronic Notes in Theoretical Computer Science* **70** (2).

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press

Delahaye, D. and Mayero, M. (2005) Quantifier Elimination over Algebraically Closed Fields in a Proof Assistant using a Computer Algebra System. In: *Proceedings of Calculemus 2005*.

Dolzmann, A., Seidl, A. and Sturm, T. (2004) Efficient Projection Orders for CAD. In: Gutierrez, J. (ed.) *Proceedings of the ISSAC 2004*, ACM Press 111–118.

Dolzmann, A. and Sturm, T. (1997) Redlog: Computer Algebra meets Computer Logic. *SIGSAM Bull* **31** (2).

Dolzmann, A., Sturm, T. and Weispfenning, V. (1998) A New Approach for Automatic Theorem Proving in Real Geometry. *Journal of Automated Reasoning* **21** 357–380.

Grégoire, B. and Leroy, X. (2002) A Compiled Implementation of Strong Reduction. In: *International Conference on Functional Programming 2002*, ACM Press 235–246.

Grégoire, B. and Mahboubi, A. (2005) Proving Equalities in a Commutative Ring Done Right in Coq. In: *Proceedings of the TPHOLs 2005*, Springer-Verlag 98–113.

Grégoire, B. and Théry, L. (2006) Certifying Large Prime Numbers: a purely functional library for modular arithmetic To appear in *Proceedings of IJCAR06*. (Available at `http://gforge.inria/projects/coqprime/.`)

Harrison, J. and Théry, L. (1998) A Skeptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning* **21** 279–294.

Hörmander, L. (2003) *The Analysis of Linear Partial Differential Operators (II)*, Grundlehren der Mathematischen Wissenschaften **257**, Springer-Verlag.

Mahboubi, A. (2006) Proving Formally the Implementation of an Efficient GCD Algorithm for Polynomials. To appear in *Proceedings of IJCAR06*.

Mahboubi, A. and Pottier, L. (2002) Elimination des Quantificateurs sur les Réels en Coq. In: *Journées Françaises des Languages Applicatifs*.

McLaughlin, S. and Harrison, J. (2005) A Proof-Producing Decision Procedure for Real Arithmetic. In: *CADE* 295–314.

Niqui, M. and Bertot, Y. (2003) Qarith: Coq Formalisation of Lazy Rational Arithmetic. In: TYPES 2003. *Springer-Verlag Lecture Notes in Computer Science* **3839** 309–323.

Palmgren, E. (2002) An Intuitionistic Axiomatisation of Real Closed Fields. *Mathematical Logic Quarterly* **48** (2) 297–299.

Sacerdoti, C. (2006) A Semi-reflexive Tactic for (Sub-)Equational Reasoning. In: TYPES 2004. *Springer-Verlag Lecture Notes in Computer Science* **3839** 98–114.

Strzebonski, A. (2004) Cylindrical Algebraic Decomposition. In: MathWorld – A Wolfram Web Resource, created by Eric W. Weisstein. (Available at `http://mathworld.wolfram.com/CylindricalAlgebraicDecomposition.html.`)

Tarski, A. (1948) *A Decision Method for Elementary Algebra and Geometry*, The RAND Corporation, Santa Monica, U.S. Air Force Project RAND, R-109.

The Coq Development Team (2004) *The Coq Proof Assistant Reference Manual Version 8.0*, Institut National de Recherche en Informatique et en Automatique. (Available at `http://coq.inria.fr/.`)

Théry, L. (2001) A Machine-Checked Implementation of Buchberger's Algorithm. *Journal of Automated Reasoning* **26** 107–137.

von zur Gathen, J. and Lücking, T. (2003) Subresultants Revisited. *Theoretical Computer Science* **297** 199–239.

https://doi.org/10.1017/S096012950600586X Published online by Cambridge University Press