A Modular, Efficient Formalisation of Real Algebraic Numbers

Wenda Li Lawrence C. Paulson
Computer Laboratory, University of Cambridge
{wl302,lp15}@cam.ac.uk

Abstract

This paper presents a construction of the real algebraic numbers with executable arithmetic operations in Isabelle/HOL. Instead of verified resultants, arithmetic operations on real algebraic numbers are based on a decision procedure to decide the sign of a bivariate polynomial (with rational coefficients) at a real algebraic point. The modular design allows the safe use of fast external code. This work can be the basis for decision procedures that rely on real algebraic numbers.

Categories and Subject Descriptors I.1.1 [*Symbolic and Algebraic Manipulation*]: Expressions and Their Representation—Representations (general and polynomial); D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs, formal methods

Keywords Real algebraic geometry, Isabelle/HOL, Decision procedure

1. Introduction

Real algebraic numbers (e.g. $\sqrt{2}$ or $3+2\sqrt{5}$) are real numbers that are defined as particular roots of non-zero polynomials with rational (or integer) coefficients. They are important in computer algebra as each one can be encoded precisely (unlike real numbers), and their arithmetic and comparison operations are decidable. Formalizing them in Isabelle/HOL [13], an interactive theorem prover, opens the way to numerous decision procedures in computer algebra. For example, consider this real closed problem in existential form:

$$\exists x\ y \in \mathbb{R}.\ x^2 - 2 = 0 \wedge xy > 1$$

With our formalization of executable arithmetic and comparison operations, we can prove it computationally: we search for a real algebraic point (e.g. through cylindrical algebraic decomposition [1]) that satisfies the quantifier-free part of the formula. One such solution is $(x = \sqrt{2},\ y = 1)$.

Our formalization[1] follows Isabelle's tradition of separation of abstraction and implementation. That is,

- **abstraction**: we first formalize real algebraic numbers on an abstract level without considering executability (see Section 2). More specifically, we formalize real algebraic numbers as a subtype of real numbers, and show them to form an ordered field using classic proofs in abstract algebra.
- **implementation (restoring executability)**: we then restore executability on real algebraic numbers (see Section 3). More specifically, we define a pseudo constructor for algebraic real numbers and prove code equations for algebraic arithmetic and comparison on this constructor. Some of the code equations for algebraic arithmetic are based on a verified decision procedure to decide the sign of a bivariate polynomial with rational coefficients at real algebraic points.

The whole project is intended for practical applications. We aim to build decision procedures related to real algebraic numbers on the top of our current formalization.

The paper continues as follows. The first component of our modular design is the abstract specification of real algebraic numbers (§2), which is then followed by an implementation (§3) in the form of Isabelle/HOL code equations. In particular, the implementation is concerned with deciding the signs of polynomials at a given real algebraic point. Related work (§5) is then described along with a discussion of various limitations of and extensions to the work (6), followed by conclusions (§7).

2. Construction on an Abstract Level

This section presents our formalization of real algebraic numbers as an abstract data type. Definitions on this level will be as abstract as possible without considering executability.

Mathematically, a real algebraic number α is a real number for which there exists a non-zero univariate polynomial $P(x)$ with integer (or rational) coefficients, such that $P(x) = 0$ when $x = \alpha$.

It is then straightforward to define the real algebraic numbers as a subset of the real numbers. We formalise this construction by defining type alg as a subtype[2] of type real :

```
typedef alg = "{x::real. ∃p::int poly. p≠0
                ∧ poly (of_int_poly p) x = 0}"
```

where `of_int_poly` converts coefficients of a polynomial from `int` to `real`, and `poly p x` means evaluating polynomial p at x.

To prove non-trivial properties about real algebraic numbers, we need at least to prove that they are closed under the basic arithmetic operations and hence form a field. For example, to show that real algebraic numbers are closed under addition, suppose we have two real algebraic numbers α and β, given by polynomials P and Q:

$$\alpha \in \mathbb{R}, P \in \mathbb{Z}[x] \qquad P \neq 0 \wedge P(\alpha) = 0$$

$$\beta \in \mathbb{R}, Q \in \mathbb{Z}[x] \qquad Q \neq 0 \wedge Q(\beta) = 0$$

[1] Source is available from `https://bitbucket.org/liwenda1990/real_algebraic_numbers`

[2] A description of Isabelle/HOL subtype definitions can be found in the Tutorial [13, §8.5.2]

CPP'16, January 18–19, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-4127-1/16/01...$15.00
http://dx.doi.org/10.1145/2854065.2854074

Then we have to show that

$$\exists R \in \mathbb{Z}[x].\, R \neq 0 \wedge R(\alpha+\beta) = 0. \quad (1)$$

One way to show this is to compute R constructively using resultants as in Cyril Cohen's proof in Coq [2]. However, as we are working on an abstract level and not concerned with executability, a non-constructive but usually simpler proof (to show the mere existence of such a polynomial) seems more appealing. Therefore, we decided to follow a classic proof in abstract algebra.

Definition 1 (vector space)**.** *A vector space V over a field F is an abelian group associated with scalar multiplication αv for all $\alpha \in F$ and $v \in V$, satisfying the standard additivity and identity axioms.*

In the Multivariate_Analysis library in Isabelle/HOL, the notion of a vector space is formalized using a `locale`:

```
locale vector_space =
  fixes scale :: "'a::field ⇒ 'b::ab_group_add ⇒ 'b"
  assumes "scale a (x + y) = scale a x + scale a y"
  and "scale (a + b) x = scale a x + scale b x"
  and "scale a (scale b x) = scale (a * b) x"
  and "scale 1 x = x"
```

where `scale::'a ⇒'b ⇒'b` denotes scalar multiplication for `'a` a field and `'b` an abelian group.

The standard library development of vector spaces has been extended by Jose Divasón and Jesús Aransay in their formalization of the Rank-Nullity Theorem in Linear Algebra, including definitions of *span* and of *linearly dependent* [4].

Definition 2 (Span)**.** *Let $S = \{v_1, v_2, ..., v_n\}$ be a set of vectors in a vector space, then* $\mathrm{span}(S)$ *is defined as*

$$\{w \mid w = a_1 v_1 + a_2 v_2 + ... + a_n v_n, \textit{ and } a_1,...,a_n \textit{ are scalars}\}$$

Divasón and Aransay [4] formalize `span` slightly differently, but the following lemma can be considered as an alternative definition that matches standard mathematical definitions:

```
lemma span_explicit:
  "span P = {y. ∃S u. finite S ∧ S ⊆ P
            ∧ setsum (λv. scale (u v) v) S = y}"
```

where `u` of type `'b ⇒'a` maps each vector in `S` to the corresponding scalar. And `setsum (λv. scale (u v) v) S` maps each element in `S` using `(λv. scale (u v) v)` and sums the results.

Definition 3 (linearly dependent)**.** *Let $S = \{v_1, v_2, ..., v_n\}$ be a set of vectors in a vector space, we say S is* linearly dependent *if there exist scalars $a_1, a_2, ..., a_n$, at least one of which is non-zero, such that*

$$a_1 v_1 + a_2 v_2 + ... + a_n v_n = \mathbf{0}$$

Divasón and Aransay [4] formalize `dependent` as

```
definition "dependent S ⟷ (∃a ∈ S. a ∈ span (S - {a}))"
```

since

$$a_1 v_1 + a_2 v_2 + ... + a_n v_n = \mathbf{0}$$
$$\iff v_n = \frac{a_1}{-a_n} v_1 + \frac{a_2}{-a_n} v_2 + ... + \frac{a_{n-1}}{-a_n} v_{n-1}$$

assuming a_n is the non-zero scalar.

Now, back to the problem of showing the formula (1), we can consider the vector space of reals with rational scalars:

```
interpretation rat: vector_space
  "(λx y. (of_rat x * y))::rat ⇒ real ⇒ real"
```

where `of_rat :: rat ⇒ real` embeds `rat` into `real` and the `scale` function in `vector_space` is instantiated as

```
(λx y. (of_rat x * y))::rat ⇒ real ⇒ real
```

After the interpretation, we have new constants, such as

```
rat.span :: real set ⇒ real set  and
rat.dependent :: real set ⇒ bool
```

that instantiate constants such as `vector_space.span` and `vector_space.dependent`, and inherit all associated lemmas from `vector_space`.

If we can show that $\{1, x, x^2, ..., x^n\}$ is linearly dependent, then (by the definition of linear dependence) it is not hard to see that there exists a non-zero polynomial with rational coefficients and degree at most n that vanishes at x:

```
lemma dependent_imp_poly:
  fixes x::real and n::nat
  assumes "rat.dependent {x ^ k |k. k ≤ n}"
  shows "∃p::rat poly. p ≠ 0 ∧ degree p ≤ n ∧
                       poly p x = 0"
```

Now the problem becomes, how can we deduce the linear dependence of a set of vectors? The solution is based on a lemma: if m vectors live in the span of n vectors with $m > n$, then these m vectors are linearly dependent.

```
lemma (in vector_space) span_card_imp_dependent:
  fixes S B::"'b set"
  assumes "S ⊆ span B" and "finite B"
      and "card S > card B"
  shows "dependent S"
```

Moreover, we can also show for all $n \in \mathbb{N}$

$$(\alpha+\beta)^n \in \mathrm{span}\{\alpha^i \beta^j \mid i\, j \in \mathbb{N}. i \leq \deg(P) \wedge j \leq \deg(Q)\}$$

which can be derived by the following lemma in Isabelle:

```
lemma bpoly_in_rat_span:
  fixes p q::"rat poly" and x y::real
    and bp::"rat poly poly"
  assumes "poly p x = 0" and "p ≠ 0"
  assumes "poly q y = 0" and "q ≠ 0"
  shows "bpoly bp x y ∈ rat.span {x ^ k1 * y ^ k2
     |k1 k2. k1 ≤ degree p ∧ k2 ≤ degree q}"
```

Above, `bp::"rat poly poly"` means a bivariate polynomial with rational coefficients and `bpoly bp x y` evaluates `bp` at (x, y). It follows that

$$1, (\alpha+\beta), ..., (\alpha+\beta)^{(\deg(P)+1)(\deg(Q)+1)}$$

are linearly dependent by applying Lemma `span_card_imp_dependent`,[3] since

$$\begin{aligned}(\deg(P)+1)(\deg(Q)+1)+1 &> \mathrm{card}\{\alpha^i\beta^j \mid i\, j \in \mathbb{N}. i \leq \deg(P) \wedge j \leq \deg(Q)\} \\ &= (\deg(P)+1)(\deg(Q)+1)\end{aligned}$$

hence there exists a non-zero polynomial with integer coefficients[4] vanishing at $\alpha+\beta$. Similarly, there exist such polynomials for the difference $\alpha-\beta$ and the product $\alpha\beta$:

```
lemma root_exist:
  fixes x y::real and p q::"rat poly"
  assumes "poly p x = 0" and "p≠0"
  assumes "poly q y = 0" and "q≠0"
  defines "rt≡(λz::real. ∃r::int poly. r≠0
    ∧ poly r z = 0)"
  shows "rt (x+y)" and "rt (x-y)" and "rt (x*y)"
```

[3] In fact, there are corner cases when $\alpha+\beta = -1, 0, 1$, but all of them can be satisfied, so the conclusion holds.

[4] We have a lemma to convert a polynomial with rational coefficients into one with integer coefficients, multiplying out the denominators.

Every rational number r is real algebraic, given by the root of the first degree polynomial $x - r$. Therefore $0 - \alpha$ is real algebraic, covering the case of $-\alpha$.

As for the multiplicative inverse, let

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_0.$$

Then clearly

$$P(\alpha) = 0 \wedge \alpha \neq 0 \implies a_0 \left(\frac{1}{\alpha}\right)^n + a_{n-1} \left(\frac{1}{\alpha}\right)^{n-1} + ... + a_n = 0 \quad (2)$$

and hence we get vanishing polynomials for $1/\alpha$:

```
lemma inverse_root_exist:
  fixes x::real and p::"rat poly"
  assumes "poly p x = 0" and "p≠0"
  shows "∃q::int poly. q≠0 ∧ poly q (inverse x) = 0"
```

as well as α/β (treated as $\alpha \times (1/\beta)$).

Finally, to define arithmetic operations on `alg`, we lift the corresponding operations from its underlying type, `real`. For example, addition on `alg` is defined as

```
lift_definition plus_alg :: "alg ⇒ alg ⇒ alg"
  is "plus::real ⇒ real ⇒ real"
```

which leaves us a goal to show that the invariant condition on `alg` is maintained (that `alg` is closed under addition):

```
⋀r1 r2 ::real.
     ∃p. p ≠ 0 ∧ poly (of_int_poly p) r1 = 0 ⟹
     ∃p. p ≠ 0 ∧ poly (of_int_poly p) r2 = 0 ⟹
     ∃p. p ≠ 0 ∧ poly (of_int_poly p) (r1 + r2) = 0
```

and this goal can be discharged by our previous Lemma `root_exist`. Similarly, we obtain `0::alg` and `1::alg`, and the ordering operations are lifted from `real` as well:

```
lift_definition zero_alg::alg is "0::real"

lift_definition one_alg::alg is "1::real"

lift_definition less_alg::"alg ⇒ alg ⇒ bool"
  is "less::real ⇒ real ⇒ bool"
```

The command **lift_definition** is part of Isabelle's Lifting and Transfer package [9].

With zero, one, arithmetic and ordering operations defined, it follows that `alg` forms an ordered field:

```
instantiation alg :: linordered_field
```

Because `alg` is essentially a subtype of `real`, all the instance proofs of the **instantiation** above are one-liners, again thanks to the Lifting and Transfer package [9]. For example, the associativity of `alg` multiplication is proved by the following tactic:

```
  show "(a * b) * c = a * (b * c)"
    by transfer auto
```

And so, we have constructed the real algebraic numbers on an abstract level and proved that they form an ordered field. But now it is time to consider the question of executability.

3. Implementation

Executability is a key property of real algebraic numbers. They are a countable subset of the real numbers and can be represented exactly in computers. This section will demonstrate how we have implemented algebraic real numbers and achieved executability on their arithmetic operations through verified bivariate sign tests.

3.1 More Pseudo-Constructors on Real Numbers

Recall that our `alg` is actually a subtype of `real`, hence executability on `real` operations can be reflected in `alg`. Therefore, our following focus is to extend executability on type `real`.

The set of real numbers, as we know, is uncountable, hence not every real number can be encoded finitely. That is, arithmetic operations can only be executable on a strict subset of the real numbers. Prior to our work, arithmetic operations on type `real` in Isabelle were only executable on rational numbers embedded into the reals (rational reals). For example, the following expression could be evaluated to be true:

```
value "Ratreal (3/4) * Ratreal 2 > (0::real)"
```

where `Ratreal` of type `rat ⇒ real` is a pseudo-constructor [6] that constructs a `real` from a `rat`. With `Ratreal`, code equations such as

```
lemma real_plus_code [code]:
   "Ratreal x + Ratreal y = Ratreal (x + y)"
```

restore executability on rational reals by mapping arithmetic operations on rational reals to rational numbers.

Our formalization follows a similar approach. We want to have a constructor `Alg` of type `_ ⇒ real` to construct (algebraic) reals from some encodings of real algebraic numbers.

An encoding (of a real algebraic number) is essentially a polynomial (with integer or rational coefficients) and a root selection strategy to distinguish a particular real root of the polynomial from any others. There are several such strategies, such as using a rational interval that only includes the target root, a natural number to indicate the index of the root and Thom encoding [1, p. 42]. We have decided to use the interval strategy, which is straightforward to implement. Therefore, `Alg` is of type `int poly ⇒ rat ⇒ rat ⇒ real`, where the two `rat` arguments represent an interval.

As each `real` number in Isabelle is presented as a Cauchy sequence of type `nat ⇒ rat`, we explicitly construct such a sequence using a suitable encoding:[5]

```
fun to_cauchy:: "rat poly × rat × rat ⇒ nat ⇒ rat"
where
  "to_cauchy (_, lb, ub) 0 = (lb+ub)/2"
| "to_cauchy (p, lb, ub) (Suc n) = (
    let c = (lb+ub)/2
    in if poly p lb * poly p c ≤ 0
       then to_cauchy (p, lb, c) n
       else to_cauchy (p, c, ub) n)"
```

The idea is to bisect the given interval at each stage. The midpoint `c` is determined as the average of the lower and upper bounds. Recall that the polynomial has exactly one root in this interval. If this root lies in the first half (indicated by a change of sign in the polynomial) then this half is chosen, and otherwise the opposite half.

We can now define the constructor `Alg`:

```
definition Alg:: "int poly ⇒ rat ⇒ rat ⇒ real" where
  "Alg p lb ub = (if valid_alg p lb ub
                  then Real (to_cauchy (p,lb,ub))
                  else undefined)"
```

where `valid_alg p lb ub` checks several validity conditions,

- `lb < ub`,
- `poly p lb * poly p ub <0`
- `p` contains exactly one real root[6] within the interval `(lb, ub)`

[5] An `int poly` can trivially be mapped to a `rat poly`.

[6] This can be computationally checked using Sturm's theorem, which is a special case of our previously formalized Sturm-Tarski theorem (see Theorem 1 in the next section).

- the interval `(lb,ub)` excludes `0` unless `0` is a root of `p`

The function `Real`, of type `(nat ⇒ rat) ⇒ real`, is the abstraction function that constructs a `real` number from its representation as a Cauchy sequence.

Some useful lemmas regarding reals constructed by `Alg` can be derived. For example, provided `valid_alg p lb ub`, we can show that `Alg p lb ub` lies within the interval (lb, ub) and is a root of the polynomial p:

```
lemma alg_bound_and_root:
  fixes p::"int poly" and lb ub::rat
  assumes "valid_alg p lb ub"
  shows "lb < Alg p lb ub" and "Alg p lb ub < ub"
      and "poly p (Alg p lb ub) = 0"
```

Note that we have described the constructor `Alg` previously [11] and repeat it here for completeness.

3.2 Deciding the Sign of a Bivariate Polynomial at a Real Algebraic Point

We have previously formalized the Sturm-Tarski theorem, and used it to decide the sign of any *univariate* polynomial with rational coefficients at a real algebraic point [10, 11]. In this section, we shall formalize a sign determination algorithm for *bivariate* polynomials. Note that $\overline{\mathbb{R}}$ denotes $\mathbb{R} \cup \{-\infty, \infty\}$, the reals extended with infinity.

Definition 4 (Tarski Query)**.** *The Tarski query* $\mathrm{TaQ}(Q, P, a, b)$ *is*

$$\mathrm{TaQ}(Q, P, a, b) = \sum_{x \in (a,b), P(x)=0} \mathrm{sgn}(Q(x))$$

where $a, b \in \overline{\mathbb{R}}$*,* $P, Q \in \mathbb{R}[X]$*,* $P \neq 0$ *and* $\mathrm{sgn} : \mathbb{R} \to \{-1, 0, 1\}$ *is the sign function.*

Essentially, the Sturm-Tarski theorem (sometimes known simply as Tarski's theorem [1]) provides a way to compute Tarski Queries using some remainder sequences:

Theorem 1 (Sturm-Tarski)**.** *Every Tarski query satisfies*

$$\mathrm{TaQ}(Q, P, a, b) = \mathrm{Var}(\mathrm{SPRemS}(P, P'Q); a, b),$$

where $P \neq 0$*,* $P, Q \in \mathbb{R}[X]$*,* P' *is the first derivative of* P*,* $a, b \in \overline{\mathbb{R}}$*,* $a < b$ *and are not roots of* P*. Moreover,* SPRemS *is the signed pseudo remainder sequence and*

$$\mathrm{Var}([p_0, p_1, ..., p_n]; a, b) = \mathrm{Var}([p_0(a), p_1(a), ..., p_n(a)]) - \mathrm{Var}([p_0(b), p_1(b), ..., p_n(b)])$$

is the difference in the number of sign variations (after removing zeroes) in the polynomial sequence $[p_0, p_1, ..., p_n]$ *evaluated at* a *and* b*.*

Note that previously [10, 11] we have used the signed remainder sequence SRemS, in which the remainder operation (mod) is from Euclidean division on polynomials:

$$P = (P \operatorname{div} Q)\, Q + (P \bmod Q)$$
$$\text{and if } Q \neq 0 \text{ then } \deg(P \bmod Q) < \deg(Q).$$

Here, the signed *pseudo* remainder sequence SPRemS is based on polynomial pseudo-divisions (pmod and pdiv):

$$\mathrm{lc}(Q)^{1+\deg(P)-\deg(Q)} P = (P \operatorname{pdiv} Q)\, Q + (P \operatorname{pmod} Q)$$
$$\text{and if } Q \neq 0 \text{ then } \deg(P \operatorname{pmod} Q) < \deg(Q)$$
$$\text{and } \mathrm{lc}(Q) \text{ is the leading coefficient of } Q \quad (3)$$

The key difference between Euclidean divisions and pseudo-divisions is that Euclidean divisions can only be carried out on polynomials over a field while pseudo-divisions are suitable for polynomials over an integral domain.

The signed pseudo remainder sequence SPRemS is implemented as follows:

```
function spmods :: "'a::idom poly ⇒ 'a poly
    ⇒ ('a poly) list" where
  "spmods p q= (if p=0 then [] else
      let
        m=(if even(degree p+1-degree q)
            then -1
            else -lead_coeff q)
      in
        Cons p (spmods q (smult m (p pmod q))))"
```

where `lead_coeff p` is the leading coefficient of the polynomial p, `pmod` is the pseudo remainder operation satisfying Equation (3), and `smult` of type `'a ⇒'a poly ⇒'a poly` is scalar multiplication that multiplies a polynomial by a constant.

Here is an example to decide the sign of $\alpha - 1$ at $\alpha = \sqrt{2} = (x^2 - 2, 1, 2)$:

$$\begin{aligned}\mathrm{TaQ}(x-1, x^2-2, 1, 2) &= \mathrm{Var}(\mathrm{SPRemS}(x^2-2, (2x)(x-1)); 1, 2)\\ &= \mathrm{Var}([x^2-2, 2x^2-2x, 8-4x, -64]; 1, 2)\\ &= 2 - 1 = 1\end{aligned}$$

As $\mathrm{TaQ}(x-1, x^2-2, 1, 2) = 1$ and $\sqrt{2}$ is the only root of $x^2 - 2$ within $(1, 2)$, we can say that $\alpha - 1$ is positive at $\alpha = \sqrt{2}$.

To illustrate our idea for a bivariate sign determination procedure, suppose we want to decide the sign of $Q(x, y) \in \mathbb{Q}[x, y]$ at (α, β) with $\alpha = (P_1, a_1, b_1)$ and $\beta = (P_2, a_2, b_2)$. By substituting y by β, we have $Q(x, \beta)$ as a univariate polynomial in $\mathbb{Q}(\beta)[x]$, where $\mathbb{Q}(\beta)$ is the field $\mathbb{Q}$ extended by β. Pretending to have arithmetic of real algebraic numbers, we can still use the univariate sign determination procedure:

$$\begin{aligned}&\mathrm{TaQ}(Q(x, \beta), P_1(x), a_1, b_1)\\ &\quad = \mathrm{Var}(\mathrm{SPRemS}(P_1(x), P_1(x)'Q(x, \beta)); a_1, b_1) \quad (4)\end{aligned}$$

To proceed from (4), we need to somehow eliminate algebraic arithmetic in the operation pmod inside SPRemS. A key lemma is

```
lemma poly_y_dist_pmod:
  fixes p::"'a::idom poly poly" and y::'a
  assumes "poly (lead_coeff p) y ≠0"
      and "poly (lead_coeff q) y ≠0"
  shows "(poly_y p y) pmod (poly_y q y) =
         poly_y (p pmod q) y"
```

where `'a poly poly` is the type we use to represent bivariate polynomials in Isabelle/HOL. This is the so-called *recursive representation*, where for example, the bivariate polynomial

$$4xy + 3x + 2y + 1 = 1 + 2y + (3 + 4y)x \in (\mathbb{Z}[y])[x]$$

is encoded as `[:[:1,2:],[:3,4:]:]`. Moreover, the function `poly_y p a` substitutes the value `a` for variable y in `p`. For example,

```
value "poly_y [:[:1,2:],[:3,4:]:] (2::int)"
```

evaluates to `[:5, 11:]`, which can be mathematically interpreted as $(4xy + 3x + 2y + 1)[y \to 2] = 5 + 11x$.

An important point about Lemma `poly_y_dist_pmod` is that the left-hand occurrence `pmod` operates over $\mathbb{Q}(\beta)[x]$ (as `poly_y p y` can be considered to be of type $\mathbb{Q}(\beta)[x]$, provided `p` $\in \mathbb{Q}[x, y]$ and `y` is instantiated to β), which demands algebraic arithmetic, while the right-hand occurrence of `pmod` operates over $\mathbb{Q}[x, y]$, which only requires arithmetic over rational numbers. Therefore, provided the leading coefficients of `p` and `q` do not vanish when evaluating at `y`, i.e. `poly (lead_coeff p) y ≠ 0` and `poly (lead_coeff q) y ≠ 0`), we can eliminate algebraic arithmetic in `pmod`.

In order to rewrite with Lemma `poly_y_dist_pmod` inside a remainder sequence, we need to satisfy its assumptions. Therefore, we have defined a function `degen` (for 'degenerates') of type `'a poly poly ⇒'a ⇒'a poly poly`, such that `degen p y` iteratively removes the leading coefficient of `p` until it does not vanish at `y` or `p` becomes `0`:

```
lift_definition degen::"'a poly poly ⇒ 'a ⇒'a poly poly" is
  "λp y n. (if poly_y p y≠0 ∧ n ≤degree (poly_y p y)
            then coeff p n else 0)"
```

Note that the term `(λp y n. ...)` above is of type `'a poly poly ⇒ 'a ⇒ nat ⇒ 'a poly`, so `degen` is defined in a way where `degen p y` (of type `'a poly poly`) is lifted from its underlying representation,[7] which is of type `nat ⇒'a poly`.

For example, a bivariate polynomial $1 + y + (y^2 - 2)x^2$ degenerates to $1 + y$ when $y = \sqrt{2}$, hence the command

```
value "degen[:[:1,1:],0,[:-2,0,1:]:] (Alg [:-2,0,1:] 1 2)"
```

evaluates to `[:[:1,1:]:]`.

Properties of `degen` include that degenerating the bivariate polynomial `p` with respect to `y` does not affect the result of evaluating it at `y`:

```
lemma poly_y_degen: "poly_y (degen p y) y = poly_y p y"
```

This holds because only leading coefficients that vanish at `y` are removed. Moreover, the leading coefficient of `degen p y` will not vanish at `y` unless `p` vanishes at `y`:

```
lemma degen_lc_not_vanish:
  assumes "degen p y≠0"
  shows "poly (lead_coeff (degen p y)) y ≠0"
```

With the help of `degen`, we can define another remainder sequence `spmods_y` that is similar to the previous signed pseudo remainder sequence `spmods` except for that `spmods_y p q y` keeps degenerating each remainder with repect to `y`:

```
function spmods_y :: "'a::idom poly poly ⇒ 'a poly poly
    ⇒'a  ⇒ ('a poly poly) list" where
  "spmods_y p q y= (if p=0 then [] else
      let
        mul=(if even(degree p+1-degree q)
              then -1
              else -lead_coeff q);
        r= degen (smult mul (p pmod q)) y
      in
        Cons p (spmods_y q r y))"
```

By exploiting Lemma `poly_y_dist_pmod`, we have established the relationship between `spmods` and `spmods_y`:

```
lemma spmods_poly_y_dist:
  fixes p q :: "'a::idom poly poly"
    and y::"'a::idom"
  assumes "poly (lead_coeff p) y≠0"
      and "poly (lead_coeff q) y≠0"
  shows "spmods (poly_y p y) (poly_y q y)
         = map (λp. poly_y p y) (spmods_y p q y)"
```

Note, similar to what we have stated for Lemma `poly_y_dist_pmod`, the importance of Lemma `spmods_poly_y_dist` is that the left-hand remainder sequence (`spmods`) requires arithmetic over $\mathbb{Q}(\beta)[x]$ (provided `p q` $\in \mathbb{Q}[x, y]$ and `y` $= \beta$) while the right-hand sequence (`spmods_y`) only requires arithmetic over $\mathbb{Q}[x, y]$.

[7] `'a poly` is constructed as a subtype of `nat ⇒'a` (i.e. a mapping from exponents to coefficients). Haftmann et al. [7] discuss how to formalize polynomials in Isabelle/HOL.

Let `spmods_y p q y` represented as $\mathrm{SPRemS}'(p, q, y)$, we can rewrite SPRemS with Lemma `spmods_poly_y_dist`:

$$\begin{aligned}\mathrm{lc}_x(Q)(\beta) \neq 0 \Longrightarrow\\ \mathrm{SPRemS}(P_1(x), P_1(x)'Q(x,\beta)) =\\ \mathrm{SPRemS}'(P_1(x), P_1(x)'Q(x,y),\beta)[y \to \beta] \quad (5)\end{aligned}$$

where $\mathrm{lc}_x(Q) \in \mathbb{Q}[y]$ is the leading coefficient of the bivariate polynomial $Q \in \mathbb{Q}[x, y]$ with respect to x. $[y \to \beta]$ performs substitution on a list of polynomials. For example, let $[x, x + y]$ be a list of polynomials, then $[x, x + y][y \to 3] = [x, x + 3]$.

By equations (4) and (5), we have

$$\begin{aligned}\mathrm{lc}_x(Q)(\beta) \neq 0 \Longrightarrow\\ \mathrm{TaQ}(Q(x,\beta), P_1(x), a_1, b_1)\\ = \mathrm{Var}(\mathrm{SPRemS}'(P_1(x), P_1(x)'Q(x,y),\beta)[y \to \beta]; a_1, b_1) \quad (6)\end{aligned}$$

Note, SPRemS' operates over $\mathbb{Q}[x, y]$ and Var requires deciding the sign of some univariate polynomial $R \in \mathbb{Q}(\beta)[x]$ when $x = a_1 \vee x = b_1$. Fortunately, as both a_1 and b_1 are rational numbers, the sign of $R(a_1)$ and $R(b_1)$ can be decided again using our univariate sign determination procedure. Hence, evaluating the right-hand side of Equation 6 requires only arithmetic on rational numbers, and we can now decide the sign of $Q(\alpha, \beta)$ with only rational arithmetic (provided $\mathrm{lc}_x(Q)(\beta) \neq 0$).

To give an example, suppose we want to decide the sign of $\alpha - \beta$ when $\alpha = \sqrt{2} = (x^2 - 2, 1, 2)$ and $\beta = \sqrt{3} = (x^2 - 3, 1, 2)$. Figure 1 shows the calculation of

$$\mathrm{TaQ}(x - \beta, x^2 - 2, 1, 2) = -1$$

provided $\mathrm{lc}_x(x - y)(\beta) = 1 \neq 0$. Therefore, we know that $(x - y)[x \to \sqrt{2}, y \to \sqrt{3}]$ is negative.

In Isabelle, we have defined the bivariate sign determination procedure as `bsgn`:

```
definition bsgn_at::"real bpoly ⇒ real ⇒ real
    ⇒ real" where
  "bsgn_at q x y=sgn (bpoly q x y)"
```

and executability of `bsgn_at` on the algebraic reals is restored by the following code equation:

```
lemma bsgn_at_code2[code]:
  fixes q::"real poly poly"
    and p1::"int poly" and lb1 ub1::rat
    and y::real
  shows "bsgn_at q (Alg p1 lb1 ub1) y =
    (if valid_alg p1 lb1 ub1
     then
       (let
          q'=degen q y
        in (if q'=0 then 0 else
              let ps = spmods_y (lift_x p1)
                (lift_x (pderiv p1) * q') y
              in changes_bpoly_at ps lb1 y
                 - changes_bpoly_at ps ub1 y))
     else Code.abort (STR ''invalid Alg'')
            (λ_. bsgn_at q (Alg p1 lb1 ub1) y))"
```

where letting `q'=degen q y` enables `q'` to satisfy the assumption of equation (6), `pderiv` means derivation and `lift_x :: 'a::zero ⇒ 'a poly poly` lifts a univariate polynomial to bivariate. Moreover,

```
    changes_bpoly_at ps lb1 y - changes_bpoly_at ps ub1
y
```

$$
\begin{aligned}
\mathrm{TaQ}(x-\beta, x^2-2, 1, 2) &= \mathrm{Var}(\mathrm{SPRemS}'(x^2-2, (2x)(x-y), \beta)[y \to \beta]; 1, 2) \\
&= \mathrm{Var}([x^2-2, 2x^2-2xy, -4xy+8, 64y^2-128][y \to \beta]; 1, 2) \\
&= \mathrm{Var}([x^2-2, 2x^2-2xy, -4xy+8, 64y^2-128][x \to 1, y \to \beta]) \\
&\quad - \mathrm{Var}([x^2-2, 2x^2-2xy, -4xy+8, 64y^2-128][x \to 2, y \to \beta]) \\
&= \mathrm{Var}([-1, -2y+2, -4y+8, 64y^2-128][y \to \sqrt{3}]) \\
&\quad - \mathrm{Var}([2, -4y+8, -8y+8, 64y^2-128][y \to \sqrt{3}]) \\
&= 1 - 2 = -1
\end{aligned}
$$

Figure 1. Deciding the sign of $\alpha - \beta$ when $\alpha = \sqrt{2}$ and $\beta = \sqrt{3}$

implements the Var operation. And also, `Code.abort` throws an exception when `Alg p1 lb1 ub1` fails to be a valid real algebraic number. Essentially, Lemma `bsgn_at_code2` implements (6).

Thanks to `bsgn_at`, the example in Figure 1 can be executed as

```
value "bsgn_at [:[:0,-1:],[:1:]:] (Alg [:-2,0,1:] 1 2)
                                  (Alg [:-3,0,1:] 1 2)"
```

which returns `-1`.

To restate: we have implemented a decision procedure (called `bsgn_at`) to decide the sign of a *bivariate* polynomial with rational coefficients at real algebraic points. This procedure uses no real algebraic arithmetic, just arithmetic on rational numbers.

3.3 Enable Executability on Algebraic Reals

Although it is possible to do verified algebraic arithmetic as in Coq [2], with the help of `bsgn_at`, we can do better. We can actually use untrusted external code to do such arithmetic, validate the result and bring it back the framework of higher order logic. The rationale behind this methodology is that untrusted but sophisticated code usually offers by far the best performance. Using untrusted code when building decision procedures improves performance in most cases; on the other hand, to provide our own trustworthy code would require costly formal verification. Another benefit of using external untrusted code is modularity: we can easily substitute one piece of code by another without modifying any proofs.

The following lemma illustrates the idea of using untrusted code in algebraic arithmetic:

```
lemma alg_add_bsgn:
  fixes p1 p2 p3::"int poly"
      and lb1 lb2 lb3 ub1 ub2 ub3::"rat"
  defines "x≡Alg p1 lb1 ub1" and "y≡Alg p2 lb2 ub2"
      and "pxy≡[:[:0::real,1:],[:1:]:]"
  assumes valid:"valid_alg p3 lb3 ub3"
    and bsgn1:"bsgn_at (pcompose (lift_x
                (of_int_poly p3)) pxy) x y = 0"
    and bsgn2:
      "bsgn_at ([:[:-of_rat lb3,1:],[:1:]:]) x y > 0"
    and bsgn3:
      "bsgn_at ([:[:-of_rat ub3,1:],[:1:]:]) x y < 0"
  shows "Alg p3 lb3 ub3 = x + y"
```

Here, let `x = Alg p1 lb1 ub1` and `y = Alg p2 lb2 ub2`,

- the assumption `valid` checks if `Alg p3 lb3 ub3` is a valid real algebraic number, which guarantees that `p3` has exactly one real root within interval $(lb1, ub1)$
- `bsgn1` checks if `p3` vanishes at `x+y`, within which `pcompose` is polynomial composition and `pxy` stands for the bivariate polynomial $x + y$
- `bsgn2` and `bsgn3` checks if `lb3 < x+y` and `x+y < ub3` respectively

With these three assumptions, all of which can be computationally checked, we can show `Alg p3 lb3 ub3 = x + y`. Therefore, to calculate real algebraic addition, we can use untrusted code to compute `p3`, `lb3` and `ub3`, and obtain the result as a sound Isabelle theorem with the help of Lemma `alg_add_bsgn`.

In order to interact with untrusted code, we have followed the idea of foreign function interface [12]. First, we declare a constant `alg_add` without attaching any definitions (see Figure 2). In Figure 2, `integer × integer` encodes `rat` and `integer list` encodes `int poly`. As Isabelle does not directly link `rat` to the target language, we decide to use the quotient of two integers (which appears more primitive and closer to the native level) to represent `rat`, and similar reasons apply to `int poly`. Also note, in Isabelle, `integer` is an equivalent type to `int` but directly maps to arbitrary precision integers (e.g. IntInf.int in SML) in the target language when doing evaluations. Essentially, we let `alg_add` be an unspecified constant that takes representations of two algebraic numbers and returns the representation of their addition and `(integer × integer) option`, where `(integer × integer) option` is a possible optimisation in case the result is a rational number.

To enable `alg_add` to do calculations, we use the adaptation technique to link a constant in Isabelle/HOL to a target language constant, so that when the logical constant gets called in evaluation, the target language constant gets invoked instead:

```
code_printing constant alg_add ⇀ (SML) "untrustedAdd"
```

where `untrustedAdd` is currently backed up by Grant Passmore's code for algebraic operations in MetiTarski [14]. After such linking, `alg_add` becomes executable:

```
value "alg_add([-2,0,1],(1,1),(2,1))([-3,0,1],(1,1),(2,1))"
```

evaluates the sum of $\sqrt{2} = (x^2 - 2, 1, 2)$ and $\sqrt{3} = (x^3 - 3, 1, 2)$, and returns the result `([1, 0, -10, 0, 1], (2, 1), (4, 1), None)`, which encodes $\sqrt{2} + \sqrt{3}$ as $(x^4 - 10x^2 + 1, 2, 4)$.

The code equation for real algebraic addition is the following:

```
lemma [code]:
"Alg p1 lb1 ub1 + Alg p2 lb2 ub2 =
   (let
     (ns,(lb3_1,lb3_2),(ub3_1,ub3_2),_)
       = alg_add (to_alg_code p1 lb1 ub1)
                 (to_alg_code p2 lb2 ub2);
     (p3,lb3,ub3)
       = of_alg_code ns lb3_1 lb3_2 ub3_1 ub3_2
   in
     (if (*assumptions in the lemma alg_add_bsgn*) then
       Alg p3 lb3 ub3
     else
       Code.abort (STR ''alg_add fails to compute
          a valid answer'') (λ_. Alg p1 lb1 ub1
          + Alg p2 lb2 ub2)))"
```

```
consts alg_add:: "integer list × (integer × integer) × (integer × integer)
                  ⇒ integer list × (integer × integer) × (integer × integer)
                  ⇒ integer list × (integer × integer) × (integer × integer)
                     × ((integer × integer) option)"
```

Figure 2. Logical constant encoding untrusted algebraic addition

where `to_alg_code` encodes `int poly` and `rat` to `integer list` and `integer ×integer`, and `of_alg_code` does the reverse. The command `Code.abort` inserts an exception with an error message, that is, when our untrusted computation `alg_add` fails to give a correct result, an exception will be thrown. This code equation can be shown to be correct using our previous Lemma `alg_add_bsgn`.

In a very similar way of exploiting untrusted code, we have defined subtraction and multiplication. As for negation and inversion, their code equations do not require untrusted code.

The code equation for negation is the following:

```
lemma [code]:
  "- Alg p lb ub =
     if valid_alg p lb ub then
         Alg (pcompose p [:0,-1:]) (-ub) (-lb)
     else
         Code.abort (STR ''invalid Alg'')
           (λ_. - Alg p lb ub)"
```

where `pcompose p [:0,-1:]` substitutes variable x in a univariate polynomial `p` by $-x$. The rationale behind this code equation is

$$p(\alpha) = 0 \wedge q(x) = p(-x) \Longrightarrow q(-\alpha) = 0$$

Also, `p(-x)` can be shown to have exactly one real root within the interval $(-lb, -ub)$, provided that `p(x)` has exactly one within the interval (lb, ub).

The code equation to invert an algebraic real number is similar:

```
lemma [code]:
  "inverse (Alg p lb ub) =
     if valid_alg p lb ub then
         (if lb < 0 ∧ 0 < ub then 0
          else Alg (rev_poly p) (inverse ub) (inverse lb))
     else
         Code.abort (STR ''invalid Alg'')
           (λ_. inverse (Alg p lb ub))"
```

where `rev_poly` simply reverse reverses the coefficients of a polynomial. For example, `rev_poly ([:0,1,2:]::int poly)` is evaluated to `[:2,1:]`. The core idea of this code equation is the same as in the abstract level (i.e. Equation 2 in Section 2). Note, in `valid_alg` we require that $lb < ub < 0$ or $0 < lb < ub$ unless `Alg p lb ub = 0`, so `rev_poly p` can be shown to have exactly one real root within the interval `(inverse lb,inverse ub)`.

By composing multiplicative inverse and multiplication, we obtain division:

```
lemma [code]: "Alg p1 lb1 ub1 / Alg p2 lb2 ub2
    = Alg p1 lb1 ub1 * (inverse (Alg p2 lb2 ub2))"
```

As with the comparison operations, we require that the interval (lb, ub) does not contain 0 unless `Alg p lb ub = 0` in `valid_alg p lb ub`. Therefore, the sign of an algebraic real can be decided by the signs of `lb` or `ub`:

```
lemma [code]: "sgn (Alg p lb ub) ==
         if valid_alg p lb ub then
            if lb > 0 then 1
            else if ub < 0 then -1
            else 0
         else
            Code.abort (STR ''invalid Alg'')
            (λ_. sgn (Alg p lb ub))"
```

and the comparison between two algebraic reals can be obtained by subtraction and compare the result with 0.

Finally, the executability of the arithmetic of our algebraic reals can be illustrated by the following example:

```
value "Alg [:-2,0,1:] 1 2 / Alg [:-3,0,1:] 1 2
       + Alg [:-5,0,1:] 2 3 > Alg [:-7,0,2:] 1 2"
```

which stands for $\sqrt{2}/\sqrt{3} + \sqrt{5} > \sqrt{7/2}$ and returns `true`.

To repeat, we have enabled executable arithmetic and comparison operations on algebraic reals by deriving new code equations for the pseudo constructor `Alg`. Some of these code equations, such as the one for algebraic addition, depend on untrusted code, whose results are verified using the bivariate sign determination algorithm `bsgn_at`, and thus brought back into higher order logic.

3.4 Linking the Algebraic Reals to the Real Algebraic Numbers

We have just seen executable arithmetic and ordering operations on algebraic reals constructed by the constructor `Alg`, of type `int poly ⇒ rat ⇒ rat ⇒ real`. To enable the same executability on type `alg`, we only need to build a constructor for `alg` lifted from `Alg`:

```
lift_definition RAlg:: "int poly ⇒ rat ⇒ rat ⇒ alg" is
  "λp lb ub. if valid_alg p lb ub then Alg p lb ub else
0"
```

and we can then have executable arithmetic and ordering operations on `alg` as well:

```
value "RAlg [:-2,0,1:] 1 2 * RAlg [:-3,0,1:] 1 2
       > RAlg [:-5,0,1:] 2 3"
```

where `op *` and `op >` in the command above operate over `alg` instead of `real`.

4. Experiments

This section presents a few examples to demonstrate the efficiency of our implementation. All the experiments are run on a Intel Core 2 Quad Q9400 (quad core, 2.66 GHz) and 8 gigabytes RAM. When benchmarking verified operations, the expression to evaluate is first defined in Isabelle/HOL, and then extracted and evaluated in Poly/ML. The reason for this is that when invoking **value** in Isabelle/HOL to evaluate an expression, a significant and unpredictable amount of time is spent generating code, so we evaluate an extracted expression to obtain more precise results. The source of our benchmark is available from the source repository online.

Firstly, we compare evaluations of the same expression using verified arithmetic from our implementation and unverified ones from MetiTarski (see Figure 3). The data in Figure 3 indicate that our verified arithmetic is 2 to 15 times slower than unverified ones due to overhead in various validity checks and inefficient data structures. We expect to narrow this gap by further refining code equations in our implementation. The experiments have also demonstrated inefficiencies in algebraic arithmetic in the current version of MetiTarski, which evaluates $(\sqrt{2} + \sqrt{6})^3$ to

$$(x^8 - 3584x^6 + 860160x^4 - 14680064x^2 + 16777216, \frac{2601}{128}, \frac{6125}{8})$$

Expression	Verified evaluation	Unverified evaluation (MetiTarki)
$(-\sqrt{2}) + (-\sqrt{3}) - (-\sqrt{5})$	0.24s	0.02s
$(\prod_{n=2}^{10} \sqrt{n})(\sqrt{17} - \sqrt{19})$	0.84s	0.15s
$\sum_{n=2}^{5} \sqrt{n}$	1.9s	1.4s
$(\sqrt{2} + \sqrt{6})^3$	1.18s	0.26s

Figure 3. Comparison between verified evaluation and unverified evaluation

while Mathematica[8] can evaluate the same expression to

$$(x^4 - 3328x^2 + 4096, 2, 59)$$

instantly. By basing our untrusted code on more sophisticated algebraic arithmetic implementations such as Z3 and Mathematica, which effectively control coefficient and degree growth, we should obtain further improvements in our algebraic arithmetic.

We have also experimented with our bivariate sign determination procedure alone, which appears to be quite efficient. For example, given the large bivariate polynomial $P(x, y)$ shown in Figure 4, our `bsgn_at` can decide $P(\sqrt{6}, \sqrt{7}) = 0$ or $P(\sqrt{13}, \sqrt{29}) > 0$ in less than $0.05s$. Note, our current `bsgn_at` always calculates a remainder sequence no matter whether the result is -1, 0 or 1, so `bsgn_at` should take similar amount of time if the input argument is of similar complexity. In the future, we may optimize `bsgn_at` by letting it attempt to decide the sign using interval arithmetic before calculating a remainder sequence; in this case `bsgn_at` may run much faster if the polynomial does not vanish at the algebraic point.

5. Related Work

The most closely related work is Cyril Cohen's construction of the real algebraic numbers in Coq [2], from which we have gained a lot of inspiration. There are some major differences between our work and his:

- Cohen's work is part of the gigantic formalization of the odd order theorem [5] and is mainly of theoretical interest. Our work, on the contrary, is for practical purposes, as we are intending to build effective decision procedures on the top of our current formalization. This difference in intent is fundamental and leads to different design choices, such as whether to use efficient untrusted code.
- Our formalization follows Isabelle's tradition of separating abstraction and implementation, that is, formalizing theories first and restoring executability afterwards. It is possible to switch to another encoding of real algebraic numbers (such as Thom encoding) without modifying any definition or lemma on the abstract level. It is also possible to have multiple implementations of one abstraction [7], so that when doing proof by reflection the code generator can choose the most efficient one depending on the situation. On the other hand, Cohen's formalization is constructive and therefore should be executable, though it may not be very efficient.
- In Cohen's formalization, arithmetic on real algebraic numbers is defined via verified bivariate resultants, while ours is mainly based on a bivariate sign determination procedure and some untrusted code.

[8] We use the RootReduce and IsolatingInterval command in Mathematica 9 to find the defining polynomial and root isolation interval.

6. Discussion

6.1 Modularity

The dependencies between the parts of our formalization are shown in Figure 5. Modularity in our formalization is reflected in two ways:

- Separation between the abstract type, `alg`, and the finite encoding, `Alg`. Switching to another encoding does not affect anything on the abstract level or further theories based on the abstraction.
- Use of untrusted code. Untrusted code is outside the logic of Isabelle/HOL (which is why we have used a dashed arrow in Figure 5 to indicate the detached relation), hence we do not need to modify our formalization as we revise the untrusted code, or substitute new code.

This modularity should make our formalization easier to maintain.

6.2 A Potential Problem

There is one potential drawback with our formalization, and it is related to the use of untrusted code. Recall that when interfacing with untrusted code, we declare a constant in higher order logic without specifying it and link it to a constant in the target language. In this case the logic constant can be executed but no lemmas are associated with it. However, this method may undermine proofs through reflection unless referential transparency[9] is guaranteed in the target language constant. For example, consider the ML function `serial`, which maintains a counter and returns the number of times it is called. Linking an Isabelle constant, say `time`, to the target language constant `serial` breaks referential transparency:

```
consts time :: "unit ⇒ integer"

code_printing constant time ⇀ (SML) "serial"
```

we have

```
value "time () = time ()"
```

which returns `false` and breaks reflexivity. This example is due to Lochbihler and Züst [12]. So we can see that any use of external code potentially makes the system inconsistent. In the short term, this is something we have to live with.

Note, our bivariate sign determination procedure (`bsgn_at`) does not depend on any untrusted code (as shown in Figure 5), hence this problem does not apply to `bsgn_at`.

6.3 Future work

Here are some possible extensions to our current formalization:

- to improve the efficiency of our untrusted code. The efficiency of our algebraic arithmetic critically depends on the underlying untrusted code, and dramatic improvements in efficiency can be expected if the untrusted part is optimized. And thanks to our modularity, we do not need to modify existing formalizations to accommodate changes in the untrusted part.

[9] Programs always return the same value and have the same effect if they are given the same input.

$$P(x,y) = y^{14}x^{24} - 49y^{12}x^{24} + 1029y^{10}x^{24} - 12005y^{8}x^{24} + 84035y^{6}x^{24} - 352947y^{4}x^{24} + 823543y^{2}x^{24} - 823543x^{24} + 4y^{15}x^{23} - 196y^{13}x^{23} + 4116y^{11}x^{23} - 48020y^{9}x^{23} + 336140y^{7}x^{23} - 1411788y^{5}x^{23} + 3294172y^{3}x^{23} - 3294172yx^{23} + 6y^{16}x^{22} - 380y^{14}x^{22} + 10388y^{12}x^{22} - 160524y^{10}x^{22} + 1536640y^{8}x^{22} - 9344692y^{6}x^{22} + 35294700y^{4}x^{22} - 75765956y^{2}x^{22} + 70824698x^{22} + 4y^{17}x^{21} - 488y^{15}x^{21} + 18424y^{13}x^{21} - 348488y^{11}x^{21} + 3841600y^{9}x^{21} - 25950008y^{7}x^{21} + 106354696y^{5}x^{21} - 243768728y^{3}x^{21} + 240474556yx^{21} + y^{18}x^{20} - 435y^{16}x^{20} + 23124y^{14}x^{20} - 565068y^{12}x^{20} + 7991214y^{10}x^{20} - 70978362y^{8}x^{20} + 404376420y^{6}x^{20} - 1441435548y^{4}x^{20} + 2937577881y^{2}x^{20} - 2619690283x^{20} - 240y^{17}x^{19} + 21360y^{15}x^{19} - 717360y^{13}x^{19} + 12759600y^{11}x^{19} - 135416400y^{9}x^{19} + 891443280y^{7}x^{19} - 3585941520y^{5}x^{19} + 8103663120y^{3}x^{19} - 7906012800yx^{19} - 60y^{18}x^{18} + 14220y^{16}x^{18} - 682560y^{14}x^{18} + 15664320y^{12}x^{18} - 210533400y^{10}x^{18} + 1786632120y^{8}x^{18} - 9753438240y^{6}x^{18} + 33374668320y^{4}x^{18} - 65372843340y^{2}x^{18} + 56083278300x^{18} + 6480y^{17}x^{17} - 505440y^{15}x^{17} + 15876000y^{13}x^{17} - 271162080y^{11}x^{17} + 2800526400y^{9}x^{17} - 18078953760y^{7}x^{17} + 71662358880y^{5}x^{17} - 160096759200y^{3}x^{17} + 154760200560yx^{17} + 1620y^{18}x^{16} - 277020y^{16}x^{16} + 12299040y^{14}x^{16} - 269256960y^{12}x^{16} + 3483988200y^{10}x^{16} - 28557590040y^{8}x^{16} + 150730554240y^{6}x^{16} - 498587050080y^{4}x^{16} + 943236739620y^{2}x^{16} - 780471701100x^{16} - 103680y^{17}x^{15} + 7516800y^{15}x^{15} - 226074240y^{13}x^{15} + 3751816320y^{11}x^{15} - 37962691200y^{9}x^{15} + 241343141760y^{7}x^{15} - 945333244800y^{5}x^{15} + 2091930986880y^{3}x^{15} - 2006546048640yx^{15} - 25920y^{18}x^{14} + 3576960y^{16}x^{14} - 148770432y^{14}x^{14} + 3128969088y^{12}x^{14} - 39196700928y^{10}x^{14} + 311791939200y^{8}x^{14} - 1597046855040y^{6}x^{14} + 5119436939904y^{4}x^{14} - 9362458478016y^{2}x^{14} + 7462643602176x^{14} + 1088640y^{17}x^{13} - 75333888y^{15}x^{13} + 2197746432y^{13}x^{13} - 35697376512y^{11}x^{13} + 355480151040y^{9}x^{13} - 2232206242560y^{7}x^{13} + 8658032737536y^{5}x^{13} - 19006687252224y^{3}x^{13} + 18110145400704yx^{13} + 272160y^{18}x^{12} - 32169312y^{16}x^{12} + 1263911040y^{14}x^{12} - 25636818816y^{12}x^{12} + 311754598848y^{10}x^{12} - 2411253230400y^{8}x^{12} + 11999023392384y^{6}x^{12} - 37270525541760y^{4}x^{12} + 65761344808992y^{2}x^{12} - 50251170777696x^{12} - 7838208y^{17}x^{11} + 525159936y^{15}x^{11} - 14978815488y^{13}x^{11} + 239276975616y^{11}x^{11} - 2352442176000y^{9}x^{11} + 14622780566016y^{7}x^{11} - 56251597312512y^{5}x^{11} + 122646925287936y^{3}x^{11} - 116191823956992yx^{11} - 1959552y^{18}x^{10} + 205752960y^{16}x^{10} - 7683683328y^{14}x^{10} + 150666034176y^{12}x^{10} - 1780750718208y^{10}x^{10} + 13398166381824y^{8}x^{10} - 64739208683520y^{6}x^{10} + 194443460499456y^{4}x^{10} - 329440707211392y^{2}x^{10} + 239069288578176x^{10} + 39191040y^{17}x^{9} - 2564213760y^{15}x^{9} + 71876367360y^{13}x^{9} - 1133012966400y^{11}x^{9} + 11022871910400y^{9}x^{9} - 67938530042880y^{7}x^{9} + 259521420856320y^{5}x^{9} - 562515973125120y^{3}x^{9} + 530240466470400yx^{9} + 9797760y^{18}x^{8} - 936385920y^{16}x^{8} + 33399164160y^{14}x^{8} - 634130622720y^{12}x^{8} + 7287769843200y^{10}x^{8} - 53313060971520y^{8}x^{8} + 249688212560640y^{6}x^{8} - 722246796875520y^{4}x^{8} + 1165376329568640y^{2}x^{8} - 789597216374400x^{8} - 134369280y^{17}x^{7} + 8633226240y^{15}x^{7} - 238673433600y^{13}x^{7} + 3721659540480y^{11}x^{7} - 35891546342400y^{9}x^{7} + 219624001551360y^{7}x^{7} - 833893702548480y^{5}x^{7} + 1798206799334400y^{3}x^{7} - 1687547919375360yx^{7} - 33592320y^{18}x^{6} + 2972920320y^{16}x^{6} - 101683952640y^{14}x^{6} + 1871528924160y^{12}x^{6} - 20912730854400y^{10}x^{6} + 148566805309440y^{8}x^{6} - 672422071587840y^{6}x^{6} + 1861440445025280y^{4}x^{6} - 2821801438955520y^{2}x^{6} + 1729044999360000x^{6} + 302330880y^{17}x^{5} - 19147622400y^{15}x^{5} + 523435530240y^{13}x^{5} - 8088560363520y^{11}x^{5} + 77428953907200y^{9}x^{5} - 470864825948160y^{7}x^{5} + 1778446285056000y^{5}x^{5} - 3817731358586880y^{3}x^{5} + 3568748878679040yx^{5} + 75582720y^{18}x^{4} - 6273365760y^{16}x^{4} + 206451680256y^{14}x^{4} - 3686763838464y^{12}x^{4} + 40038373799424y^{10}x^{4} - 275719665553920y^{8}x^{4} + 1200874682004480y^{6}x^{4} - 3151406817119232y^{4}x^{4} + 4386241354376448y^{2}x^{4} - 2269890275159808x^{4} - 403107840y^{17}x^{3} + 25234550784y^{15}x^{3} - 683429031936y^{13}x^{3} + 10480561975296y^{11}x^{3} - 99689778155520y^{9}x^{3} + 602977978552320y^{7}x^{3} - 2266926198018048y^{5}x^{3} + 4846858942205952y^{3}x^{3} - 4514882302328832yx^{3} - 100776960y^{18}x^{2} + 7921069056y^{16}x^{2} - 251539292160y^{14}x^{2} + 4361304342528y^{12}x^{2} - 46001094580224y^{10}x^{2} + 306328298895360y^{8}x^{2} - 1276416305160192y^{6}x^{2} + 3130065461698560y^{4}x^{2} - 3834330190580736y^{2}x^{2} + 1377703055490048x^{2} + 241864704y^{17}x - 14995611648y^{15}x + 402946596864y^{13}x - 6139009916928y^{11}x + 58071715430400y^{9}x - 349591726891008y^{7}x + 1308936465801216y^{5}x - 2788603774967808y^{3}x + 2589417791041536yx + 60466176y^{18} - 4534963200y^{16} + 139314069504y^{14} - 2346571358208y^{12} + 24016802310144y^{10} - 154180404467712y^{8} + 609753012019200y^{6} - 1365846746923008y^{4} + 1344505391502336y^{2} - 49796495981568$$

Figure 4. A large bivariate polynomial

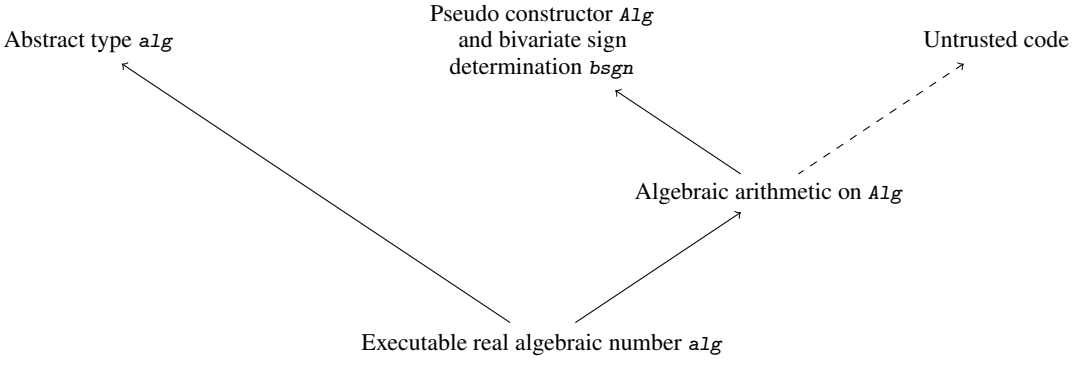

Figure 5. Dependence tree of our formalization of real algebraic numbers

- to generalize the bivariate sign determination procedure to decide the sign of a *multivariate* polynomial with rational coefficients. The idea behind a bivariate and a multivariate procedure should be the same, and the only reason we did not build a multivariate sign determination procedure directly is that Isabelle's multivariate polynomial library [7] is not finished yet.
- to integrate the sign determination algorithm with sophisticated interval arithmetic: to decide the sign using interval arithmetic first (could refine the interval for algebraic numbers a couple of times before giving up) and revert to the current signed remainder sequences if fails, as others have done [3, 15]. Moreover, dyadic rationals[10] (numbers of the shape $n2^m$ for n, $m \in \mathbb{Z}$) can be used to improve performance with intervals.
- to improve arithmetic between real algebraic numbers and rational numbers. For example, given a real algebraic number $\alpha = (p, lb, ub)$ and a rational number r, the defining polynomial for $\alpha + r$ is $p(x - r)$, which merely needs polynomial composition instead of calculating resultants. However, the interval $(lb + r, ub + r)$ may need to be refined to exclude zero, and the termination of such a refinement function may take some effort to show. For now, we treat r as an algebraic number $(x - r, r/2, 2r)$ and deploy algebraic arithmetic, which is not very efficient. And real algebraic numbers that are also rational, such as $(x^2 - 4, 1, 3)$, should be converted to rational numbers.

7. Conclusions

In this paper, we have formalized real algebraic numbers in Isabelle/HOL. The formalization is on two levels:

- on the abstract level, proofs in abstract algebra are used to show that real algebraic numbers, which are formalized as a subset of real numbers, form an ordered field;
- on the implementation level, an additional pseudo constructor for real numbers and related code equations are proved via a bivariate sign determination procedure and some untrusted code.

Experiments indicate that overhead in our verified algebraic arithmetic is reasonable (compared to unverified ones) and the bivariate sign determination procedure alone is quite efficient already.

When building practical decision procedures involving real algebraic numbers, users of our formalization should first try to build the procedure upon our sign determination procedure, as it only uses rational arithmetic and is much more efficient than exact algebraic arithmetic.

Acknowledgments

Grant Passmore has provided invaluable technical advice and even code, despite his considerable other commitments. The CSC Cambridge International Scholarship is generously funding Wenda Li's Ph.D. course.

[10] This has been implemented as the `Float` library in Isabelle, and used in interval arithmetic in the `approximation` method [8].

References

[1] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in Real Algebraic Geometry (Algorithms and Computation in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 3540330984.

[2] C. Cohen. *Formalized algebraic numbers: construction and first-order theory*. PhD thesis, École polytechnique, Nov 2012.

[3] L. De Moura and G. O. Passmore. Computation in real closed infinitesimal and transcendental extensions of the rationals. In *Automated Deduction–CADE-24*, pages 178–192. Springer, 2013.

[4] J. Divasón and J. Aransay. Rank-Nullity Theorem in Linear Algebra. *Archive of Formal Proofs*, Jan. 2013. ISSN 2150-914x. `http://afp.sf.net/entries/Rank_Nullity_Theorem.shtml`, Formal proof development.

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

[6] F. Haftmann and L. Bulwahn. Code generation from Isabelle/HOL theories. 2015. URL `https://isabelle.in.tum.de/dist/Isabelle2015/doc/codegen.pdf`.

[7] F. Haftmann, A. Lochbihler, and R. Wolfgang Schreiner. Towards abstract and executable multivariate polynomials in isabelle.

[8] J. Hölzl. *Proving real-valued inequalities by computation in Isabelle/HOL*. PhD thesis, Diploma thesis, Institut für Informatik, Technische Universität München, 2009.

[9] B. Huffman and O. Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In *Certified Programs and Proofs*, pages 131–146. Springer, 2013.

[10] W. Li. The Sturm-Tarski Theorem. *Archive of Formal Proofs*, Sept. 2014. ISSN 2150-914x. `http://afp.sf.net/entries/Sturm_Tarski.shtml`, Formal proof development.

[11] W. Li, G. O. Passmore, and L. C. Paulson. A complete decision procedure for univariate polynomial problems in Isabelle/HOL. *arXiv preprint arXiv:1506.08238*, 2015.

[12] A. Lochbihler and M. Züst. Programming TLS in Isabelle/HOL. In *Isabelle Workshop*, volume 2014, 2014.

[13] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[14] G. O. Passmore, L. C. Paulson, and L. de Moura. Real algebraic strategies for MetiTarski proofs. In *Intelligent Computer Mathematics*, pages 358–370. Springer, 2012.

[15] A. W. Strzeboński. Cylindrical algebraic decomposition using validated numerics. *Journal of Symbolic Computation*, 41(9):1021 – 1038, 2006. ISSN 0747-7171. . URL `http://www.sciencedirect.com/science/article/pii/S0747717106000496`.