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 the way to numerous decision procedures in computer algebra. For example, consider this real closed problem in existential form:

\[ \exists x y \in \mathbb{R}. x^2 - 2 = 0 \land xy > 1 \]

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,

\(^1\)Source is available from https://bitbucket.org/liwenda1990/real_algebraic_numbers

- 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 real 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 \(\alpha\) 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:

\[
\text{typedef alg = } \{x::real. \exists p::\text{int poly. } p\neq 0 \land \text{poly (of_int_poly p) x = 0}\}
\]

where \(\text{of_int_poly}\) converts coefficients of a polynomial from \(\text{int}\) to \(\text{real}\), and \(\text{poly p x}\) 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 \(\alpha\) and \(\beta\), given by polynomials \(P\) and \(Q\):

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

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

\(^2\)A description of Isabelle/HOL subtype definitions can be found in the Tutorial [13, §8.5.2]
Then we have to show that
\[ \exists R \in \mathbb{Z}[x], R \neq 0 \land R(\alpha + \beta) = 0. \] (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 \( \alpha \) 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:

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

where \( \text{scale}::\alpha \Rightarrow \beta \Rightarrow \alpha \Rightarrow \beta \) denotes scalar multiplication for \( \alpha \) a field and \( \beta \) an abelian group.

The standard library development of vector spaces has been extended by José 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 \( S = \{v_1, v_2, \ldots, v_n\} \) be a set of vectors in a vector space, then span(\( S \)) is defined as
\[ \{ w \mid w = a_1 v_1 + a_2 v_2 + \ldots + a_n v_n, \text{and } a_1, \ldots, a_n \text{ are scalars} \} \]

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. \exists S u. finite S \land S \subseteq P \land \text{setsum (\( \forall v. \text{scale (u v) v} \)) S = y}" 

where \( u \) of type \( \beta \Rightarrow \alpha \) maps each vector in \( S \) to the corresponding scalar. And \( \text{setsum (\( \forall v. \text{scale (u v) v} \)) S} \) maps each element in \( S \) using \( \text{scale (u v) v} \) and sums the results.

**Definition 3** (linearly dependent). Let \( S = \{v_1, v_2, \ldots, v_n\} \) be a set of vectors in a vector space, we say \( S \) is linearly dependent if there exist scalars \( a_1, a_2, \ldots, a_n, \) at least one of which is non-zero, such that
\[ a_1 v_1 + a_2 v_2 + \ldots + a_n v_n = 0 \]

Divasón and Aransay [4] formalize dependent as

**definition** dependent S \( \leftarrow \) (\( \exists a \in S. a \in \text{span (S - \{a\})}\)" since
\[ a_1 v_1 + a_2 v_2 + \ldots + a_n v_n = 0 \]
\[ \iff v_n = \frac{a_1}{-a_n} v_1 + \frac{a_2}{-a_n} v_2 + \ldots + \frac{a_{n-1}}{-a_n} v_{n-1} \]

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
"(\( \lambda x y. \text{of_rat x * y})::\text{rat} \Rightarrow \text{real} \Rightarrow \text{real}" 

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

\( (\lambda x y. \text{(of_rat x * y)})::\text{rat} \Rightarrow \text{real} \Rightarrow \text{real} \)

After the interpretation, we have new constants, such as
\[ \text{rat.span :: real set} \Rightarrow \text{real set} \text{ and} \]
\[ \text{rat.dependent :: real set} \Rightarrow \text{bool} \]

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 \( \{1, x, x^2, \ldots, x^n\} \) 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:
\[ \text{fixes x::real and n::nat} \]
\[ \text{assumes "rat.dependent \{x ^ k | k \leq n\}"} \]
\[ \text{shows \( \exists p::\text{rat poly}. p \neq 0 \land \text{degree p} \leq n \land \text{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:
\[ \text{fixes S :: \"\'b set\"} \]
\[ \text{assumes \"S \subseteq \text{span B}\" and \"finite B\"} \]
\[ \text{and \( \text{card S} > \text{card B}\)} \]
\[ \text{shows \"dependent S\"} \]

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

\[ (\alpha + \beta)^n \in \text{span}\{\alpha^i \beta^j | i,j \in \mathbb{N}. i \leq \text{deg}(P) \land j \leq \text{deg}(Q)\} \]

which can be derived by the following lemma in Isabelle:

**lemma** bpoly_in_rat_span:
\[ \text{fixes p q::\"\text{rat poly}\" and x y::real} \]
\[ \text{and bp::\"\text{rat poly poly}\"} \]
\[ \text{assumes \"\text{poly p x} = 0" and \"p \neq 0\"} \]
\[ \text{assumes \"p y = 0\" and \"q \neq 0\"} \]
\[ \text{shows \"\text{bpoly bp x y \in \text{rat.span (x ^ k1 * y ^ k2)} \Rightarrow k1 k2} \leq \text{degree p} \land k2 \leq \text{degree q}\"} \]

Above, \( \text{bpoly poly poly} \) means a bivariate polynomial with rational coefficients and \( \text{bpoly bp x y} \) evaluates \( \text{bp} \) at \( (x, y) \). It follows that

\[ 1, (\alpha + \beta), \ldots, (\alpha + \beta)^n ((\text{deg}(P) + 1)(\text{deg}(Q) + 1) \]
\[ \text{are linearly dependent by applying Lemma span_card_imp_dependent,} \]

since

\[ \text{(deg}(P) + 1)(\text{deg}(Q) + 1) \]
\[ > \text{card}\{\alpha^i \beta^j | i,j \in \mathbb{N}. i \leq \text{deg}(P) \land j \leq \text{deg}(Q)\} \]
\[ = (\text{deg}(P) + 1)(\text{deg}(Q) + 1) \]

hence there exists a non-zero polynomial with integer coefficients

\( \) vanishing at \( \alpha + \beta \). Similarly, there exist such polynomials for the difference \( \alpha - \beta \) and the product \( \alpha \beta \):

**lemma** root_exist:
\[ \text{fixes x y::real and p q::\"\text{rat poly}\"} \]
\[ \text{assumes \"\text{poly p x} = 0\" and \"p \neq 0\"} \]
\[ \text{assumes \"\text{poly q y} = 0\" and \"q \neq 0\"} \]
\[ \text{defines \"\text{rt}(\alpha::real. 3::int poly. r \neq 0)\"} \]
\[ \text{and \"rt (x*y)\" and \"rt (x+y)\"} \]

\[ \text{1} \]

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

\[ \text{2 We have a lemma to convert a polynomial with rational coefficients into one with integer coefficients, multiplying out the denominators.} \]
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

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

Then clearly

$$P(\alpha) = 0 \land \alpha \neq 0$$

$$\implies a_0 \left( \frac{1}{\alpha} \right)^n + a_{n-1} \left( \frac{1}{\alpha} \right)^{n-1} + \ldots + a_n = 0$$

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

**Lemma inverse_root_exist:**

fixes $x :: \text{real}$ and $p :: \text{"rat poly"}$

assumes *poly p x = 0* and *p \neq 0*

shows *\exists q :: \text{int poly}. q \neq 0 \land poly q (inverse x) = 0*.

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

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

**lift_definition plus_alg** :: "\text{alg} \Rightarrow \text{alg} \Rightarrow \text{alg}"

is *plus :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real}*

which leaves us a goal to show that the invariant condition on $\text{alg}$ is maintained (that $\text{alg}$ is closed under addition):

$$\forall \alpha \beta \in \text{real}. \exists \alpha \beta \in \text{real}. \beta \in \text{real} \Rightarrow \text{alg} = \text{alg}$$

$$\alpha \beta = \text{alg} \Rightarrow \text{alg}$$

which we obtain $0 :: \text{alg}$ and $1 :: \text{alg}$, and the ordering operations are lifted from $\text{real}$ as well:

**lift_definition zero_alg** :: $\text{alg}$ is "$0 :: \text{real}"

**lift_definition one_alg** :: $\text{alg}$ is "$1 :: \text{real}"

**lift_definition less_alg** :: "$\text{alg} \Rightarrow \text{alg} \Rightarrow \text{bool}"

is *less :: \text{real} \Rightarrow \text{real} \Rightarrow \text{bool}*

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

With zero, one, arithmetic and ordering operations defined, it follows that $\text{alg}$ forms an ordered field.

**instantiation alg :: linordered_field**

Because $\text{alg}$ is essentially a subtype of $\text{real}$, all the instance proofs of the **instantiation** above are one-liners, again thanks to the Lifting and Transfer package. For example, the associativity of $\text{alg}$ 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 $\text{alg}$ is actually a subtype of $\text{real}$, hence executability on $\text{real}$ operations can be reflected in $\text{alg}$. Therefore, our following focus is to extend executability on type $\text{real}$.

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 $\text{real}$ 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 $\text{Ratreal}$ of type $\text{rat}$ is a pseudo-constructor that constructs a real from a rat. With $\text{Ratreal}$, 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 $\text{Alg}$ of type $\_ :: \text{real}$ 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, $\text{Alg}$ is of type $\text{int poly} \Rightarrow \text{rat} \Rightarrow \text{rat} \Rightarrow \text{rat}$, where the two $\text{rat}$ arguments represent an interval.

As each $\text{real}$ number in Isabelle is presented as a Cauchy sequence of type $\text{nat} \Rightarrow \text{rat}$, we explicitly construct such a sequence using a suitable encoding.\(^5\)

**fun to_cauchy:: "\text{rat poly} \times \text{rat} \times \text{rat} \Rightarrow \text{rat} \Rightarrow \text{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 \leq 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 $\text{Alg}$:

**definition Alg:: "\text{int poly} \Rightarrow \text{rat} \Rightarrow \text{rat} \Rightarrow \text{real} where**

*Alg p lb ub = (if valid_alg p lb ub then Real (to_cauchy (p, lb, ub)) else undefined)*

where $\text{valid alg}$ checks several validity conditions,

- $\text{lb} < \text{ub}$,
- $\text{poly p lb \* poly p ub < 0}$
- $\text{p}$ contains exactly one real root\(^6\) within the interval $(\text{lb}, \text{ub})$

---

\(^5\) 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).
We have previously formalized the Sturm-Tarski theorem, and used
Tarski’s theorem [1]) provides a way to compute Tarski Queries
where

\[ a, b \]

\[ \text{div} \]

\[ \text{mod} \]

\[ \text{sgn} : \]

\[ \text{lead_coeff} \]

\[ \text{pdiv} \]

\[ \text{pmod} \]

\[ \text{smult} \]

\[ \text{spmods} \]

\[ \text{alg_bound_and_root} : \]

\[ \text{polynomial} \]

\[ \text{valid_alg} \]

\[ \text{Var} (\) \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]

\[ \text{alg_bound_and_root} : \]
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 'degenerating') 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:

\textbf{Definition} \textit{degen} :: "'a poly poly ⇒ 'a ⇒ 'a poly poly" is \(\lambda p \ y \ n. \ (\text{if} \ \text{poly}_y \ p \ y \neq 0 \ \text{then degree} \ (\text{poly}_y \ p) \ \text{then coeff} \ p \ n \ \text{else} \ 0)\)"

Note that the term \(\lambda p \ y \ n. \ \ldots\) above is of type 'a poly poly ⇒ 'a ⇒ nat ⇒ 'a poly, so degen is defined in a way where \textit{degen} \ p \ y (of type 'a poly poly) is lifted from its underlying representation, 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

```plaintext
value "degen[:[:1,1:],0,[:[-2,0,1:]]:(Alg :[:[-2,0,1:] 1 2])"
```
evaluates to \([[:1,1:],\ldots]\).

Properties of \textit{degen} include that degenerating the bivariate polynomial \(p\) with respect to \(y\) does not affect the result of evaluating it at \(y\):

\textbf{Lemma} \textit{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 \textit{degen} \(p\) \(y\) will not vanish at \(y\) unless \(p\) vanishes at \(y\):

\textbf{Lemma} \textit{degen_lc_not_vanish}::
\begin{align*}
\text{assumes} & \quad "\text{poly} (\text{lead_coeff} (\text{degen} p y)) y \neq 0" \\
\text{shows} & \quad "\text{poly (lead_coeff (degen p y))) y } \neq 0"
\end{align*}

With the help of \textit{degen}, we can define another remainder sequence \text{spmods}_y that is similar to the previous signed pseudo remainder sequence \text{spmods} except for that \text{spmods}_y \ p \ q \ y keeps degenerating each remainder with respect to \(y\):

\textbf{Function} \textit{spmods}_y :: "'a::idom poly poly ⇒ 'a poly poly ⇒ 'a ⇒ ('a poly poly) list" where \text{spmods}_y \ p \ q \ y = (\text{if} \ p \neq 0 \ \text{then} \ [] \ \text{else} \ [\text{spmods}_p (\text{mul} \ (p \ \text{spmod} \ q) \ y) \ y ] \ \text{in} \ p )

By exploiting \textit{Lemma poly_y_dist_pmod}, we have established the relationship between \textit{spmods} and \textit{spmods}_y:

\textbf{Lemma} \textit{spmods_poly_y_dist}::
\begin{align*}
\text{fixes} & \quad p :: "'a::idom poly poly" \\
\text{and} & \quad y :: "'a::idom" \\
\text{assumes} & \quad "\text{poly (lead coeff} p) y \neq 0" \\
\text{and} & \quad "\text{poly (lead coeff} q) y \neq 0" \\
\text{shows} & \quad "\text{spmods} (\text{poly}_y p y) (\text{poly}_y q y) = (\text{map} (\lambda p. \ \text{poly}_y p y) \ (\text{spmods}_y p q y))"
\end{align*}

Note, similar to what we have stated for \textit{Lemma poly_y_dist_pmod}, the importance of \textit{Lemma spmods_poly_y_dist} is that the left-hand remainder sequence (\textit{spmods}) requires arithmetic over \(\mathbb{Q}(\beta)[x]\) (provided \(p \in \mathbb{Q}[x,y]\) and \(y = \beta\)) while the right-hand sequence (\textit{spmods}_y) only requires arithmetic over \(\mathbb{Q}[x,y]\).
\[ TaQ(x - \beta, x^2 - 2, 1, 2) = \text{Var}(SPRemS'(x^2 - 2, (2x)(x - y), \beta)[y \to \beta]; 1, 2) = \text{Var}(x^2 - 2, 2x^2 - 2xy, -4xy + 8, 64y^2 - 128[y \to \beta]; 1, 2) = \text{Var}(x^2 - 2, 2x^2 - 2xy, -4xy + 8, 64y^2 - 128[x \to 1, y \to \beta]) - \text{Var}(x^2 - 2, 2x^2 - 2xy, -4xy + 8, 64y^2 - 128[x \to 2, y \to \beta]) = \text{Var}([-1, -2y + 2, -4y + 8, 64y^2 - 128[y \to \sqrt{3}]) - \text{Var}([2, -4y + 8, -8y + 64y^2 - 128[y \to \sqrt{3}]) = 1 - 2 = -1 \]

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

implements the \text{Var} operation. And also, \text{Code.abort} throws an exception when \( \text{Alg} p1 \# l1 \# u1 \) fails to be a valid real algebraic number. Essentially, Lemma \text{bsgn_at_code2} implements (6).

Thanks to \text{bsgn_at}, the example in Figure 1 can be executed as

\[
\text{value "bsgn_at [[:0,-1:],[:1:]] \ (Alg [:-2,0,1:] 1 2) \ (Alg [:-3,0,1:] 1 2)" which returns -1.}
\]

To restate: we have implemented a decision procedure (called \text{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 \text{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_code:**

\[
\text{fixes p1 \# p2 \# p3::"int poly" and l1 \# l2 \# l3 \# u1 \# u2 \# u3::"rat"; \ "rat"}
\]

\[
\text{defines "x \equiv \text{Alg} p1 \# l1 \# u1" and "y \equiv \text{Alg} p2 \# l2 \# u2" and "pxy \equiv \text{Alg} p3 \# l3 \# u3"}
\]

\[
\text{assumes valid:="valid_alg p3 \# l3 \# u3" and bsgn1:="bsgn_at \ (\text{pcompose \ (lift_x \ (\text{of_int_poly} \ p3)) \ pxy \ \# x \# y \# 0" and bsgn2:="bsgn_at \ ([:\:-of\_rat \ l3,1:],[:1:]): \ x \# y \# 0" and bsgn3:="bsgn_at \ ([:\:-of\_rat \ u3,1:],[:1:]): \ x \# y \# 0"}
\]

\[
\text{shows \ "Alg \ p3 \# l3 \# u3 \# x \# y"}
\]

Here, let \( x = \text{Alg} p1 \# l1 \# u1 \) and \( y = \text{Alg} p2 \# l2 \# u2 \).

- the assumption \text{valid} checks if \( \text{Alg} p3 \# l3 \# u3 \) is a valid real algebraic number, which guarantees that \( p3 \) has exactly one real root within interval \((l1,u1)\)
- \text{bsgn1} checks if \( p3 \) vanishes at \( x+y \), within which \text{pcompose} is polynomial composition and \( pxy \) stands for the bivariate polynomial \( x + y \)
- \text{bsgn2} and \text{bsgn3} checks if \( l3 < x+y \) and \( x+y < u3 \) respectively

With these three assumptions, all of which can be computationally checked, we can show \( \text{Alg} p3 \# l3 \# u3 = x + y \). Therefore, to calculate real algebraic addition, we can use untrusted code to compute \( p3, l3 \) and \( u3 \), and obtain the result as a sound Isabelle theorem with the help of Lemma \text{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 \text{alg_add} without attaching any definitions (see Figure 2). In Figure 2, \( \text{integer \times integer} \) encodes \( \text{rat} \) and \( \text{integer \ list} \) encodes \( \text{int \ poly} \). As Isabelle does not directly link \( \text{rat} \) 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 \( \text{rat} \), and similar reasons apply to \( \text{int \ poly} \). Also note, in Isabelle, \( \text{integer} \) is an equivalent type to \( \text{int} \) but directly maps to arbitrary precision integers (e.g. \text{IntInf.int} in \text{SML}) in the target language when doing evaluations. Essentially, we let \( \text{alg_add} \) be an unspecified constant that takes representations of two algebraic numbers and returns the representation of their addition and \( \text{(integer \times integer)} \) option, where \( \text{(integer \times integer)} \) option is a possible optimisation in case the result is a rational number.

To enable \text{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:

\[
\text{code_printing \ constant \ alg_add \ → \ (SML) \ "untrustedAdd"}
\]

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

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

The code equation for real algebraic addition is the following:

**Lemma: code:**

\[
\text{"Alg \ p1 \# l1 \# u1 + Alg \ p2 \# l2 \# u2 = (let (ns, (l3_1,l3_2), (u3_1,u3_2),..) = alg_add \ (to_alg_code \ p1 \# l1 \# u1) \ (to_alg_code \ p2 \# l2 \# u2); \ (p3,l3,ub3) = of_alg_code \ ns \# l3_1 \# l3_2 \# u3_1 \# u3_2 \ in \ (if \ (*\text{assumptions in the lemma alg_add_bsgn*} then \ Alg \ p3 \# l3 \# u3 \ else \ Code.abort (STR \ "alg_add fails to compute a valid answer") \ (\_, \ Alg \ p1 \# l1 \# u1 + Alg \ p2 \# l2 \# u2)))")}
\]
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 begin.

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]:**

\[
\begin{align*}
& \text{~Alg p lb ub =} \\
& \quad \text{if valid_alg p lb ub then} \\
& \quad \quad \text{Alg (pcompose p [0,-1:]) (-ub) (-lb)} \\
& \quad \text{else} \\
& \quad \quad \text{Code.abort (STR "invalid Alg")}
\end{align*}
\]

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

\[
p(a) = 0 \land q(x) = p(-x) \Rightarrow q(-a) = 0
\]

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]:**

\[
\begin{align*}
& \text{~inverse (Alg p lb ub) =} \\
& \quad \text{if valid_alg p lb ub then} \\
& \quad \quad \text{if lb < 0 \land 0 < ub then 0} \\
& \quad \quad \text{else Alg (rev_poly p) (inverse ub) (inverse lb)} \\
& \quad \text{else} \\
& \quad \quad \text{Code.abort (STR "invalid Alg")}
\end{align*}
\]

where rev_poly simply reverse reverses the coefficients of a polynomial. For example, rev_poly ([0,1,2]::int poly) is evaluated to [2,1,0]. 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 \((lb,ub)\).

By composing multiplicative inverse and multiplication, we obtain division:

**lemma [code]:**

\[
\begin{align*}
& \text{~Alg p1 lb1 ub1 / Alg p2 lb2 ub2 =} \\
& \quad \text{Alg p1 lb1 ub1 * (inverse (Alg p2 lb2 ub2))}
\end{align*}
\]

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]:**

\[
\begin{align*}
& \text{~sgn (Alg p lb ub) =} \\
& \quad \text{if valid_alg p lb ub then} \\
& \quad \quad \text{if lb > 0 then 1} \\
& \quad \quad \text{else if ub < 0 then -1} \\
& \quad \quad \text{else 0} \\
& \quad \text{else} \\
& \quad \quad \text{Code.abort (STR "invalid Alg")}
\end{align*}
\]

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**

\[
\begin{align*}
& \text{Alg [-2,0,1:] 1 2 / Alg [-3,0,1:] 1 2} \\
& \quad + \text{Alg [-5,0,1:] 2 3 > Alg [-7,0,2:] 1 2}
\end{align*}
\]

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**

\[
\text{RAlg :: "int poly ⇒ rat ⇒ rat ⇒ alg" is} \\
\text{\lambda p lb ub. if valid alg p lb ub then alg p lb ub else 0"}
\]

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

**value**

\[
\begin{align*}
& \text{RAlg [-2,0,1:] 1 2 * RAlg [-3,0,1:] 1 2} \\
& \quad > \text{RAlg [-5,0,1:] 2 3}
\end{align*}
\]

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 \((\sqrt{2} + \sqrt{3})^6\) to

\[
(2601^{6} - 128^{6} \cdot 8)
\]
while Mathematica\textsuperscript{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 \texttt{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 \texttt{bsgn\_at} always calculates a remainder sequence no matter whether the result is \(-1\), \(0\) or \(1\), so \texttt{bsgn\_at} should take similar amount of time if the input argument is of similar complexity. In the future, we may optimize \texttt{bsgn\_at} by letting it attempt to decide the sign using interval arithmetic before calculating a remainder sequence; in this case \texttt{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.

### 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, \texttt{Alg}, and the finite encoding, \texttt{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\textsuperscript{9} is guaranteed in the target language constant. For example, consider the ML function \texttt{serial}, which maintains a counter and returns the number of times it is called. Linking an Isabelle constant, say \texttt{time}, to the target language constant \texttt{serial} breaks referential transparency:

\begin{verbatim}
consts time :: "unit \Rightarrow integer"

code_printing constant time \as (SML) "serial"

we have

value "time () = time ()"
\end{verbatim}

which returns \texttt{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 (\texttt{bsgn\_at}) does not depend on any untrusted code (as shown in Figure 5), hence this problem does not apply to \texttt{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.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Expression & Verified evaluation & Unverified evaluation (MetiTarki) \\
\hline
\((\sqrt{2} - \sqrt{3}) + (-\sqrt{3}) - (-\sqrt{5})\) & 0.24s & 0.02s \\
\((\prod_{n=2}^{10} \sqrt{n})(\sqrt{17} - \sqrt{19})\) & 0.84s & 0.15s \\
\(\sum_{n=2}^{10} \sqrt{n}\) & 1.9s & 1.4s \\
\((\sqrt{2} + \sqrt{3})^3\) & 1.18s & 0.26s \\
\hline
\end{tabular}
\caption{Figure 3. Comparison between verified evaluation and unverified evaluation}
\end{table}

\textsuperscript{8}We use the \texttt{RootReduce} and \texttt{IsolatingInterval} command in Mathematica 9 to find the defining polynomial and root isolation interval.

\textsuperscript{9}Programs always return the same value and have the same effect if they are given the same input.
Abstract type $\alpha_1$

Pseudo constructor $\alpha_1$ and bivariate sign determination $\text{bgsn}$

Untrusted code

Algebraic arithmetic on $\alpha_1$

Executable real algebraic number $\alpha_1$

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

short description of paper 9 2016/08
• 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 remain-
der sequences if fails, as others have done [3, 15]. Moreover,
dyadic rationals (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 ra-
tional 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 in-
terval \((lb + r, ub + r)\) 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
\((x - r, r, 2r, 2r)\) 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 Is-
abelle/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 ar-
ithmetic 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 Cam-
bridge International Scholarship is generously funding Wenda Li’s
Ph.D. course.

References

Geometry (Algorithms and Computation in Mathematics). Springer-
finite and transcendental extensions of the rationals. In Automatic
sf.net/entries/Rank_Nullity_Theorem.shtml, Formal proof
development.
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
abstract and executable multivariate polynomials in isabelle.
abelle/HOL. PhD thesis, Diploma thesis, Institut für Informatik, Tech-
nische Universität München, 2009.
quotients in Isabelle/HOL. In Certified Programs and Proofs, pages
2014. ISSN 2150-914x. http://afp.sf.net/entries/Sturm_Tarski.shtml, Formal proof
development.
procedure for univariate polynomial problems in Isabelle/HOL. arXiv
In Isabelle Workshop, volume 2014, 2014.
strategies for MetiTarski proofs. In Intelligent Computer Mathematics,
numerics. Journal of Symbolic Computation, 41(9):1021 – 1038,
2006. ISSN 0747-7171. URL http://www.sciencedirect.com/
science/article/pii/S0747717106000496.

---

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