Algebraic Numbers in Isabelle/HOL

René Thiemann and Akihisa Yamada

University of Innsbruck

Abstract. We formalize algebraic numbers in Isabelle/HOL, based on existing libraries for matrices and Sturm's theorem. Our development serves as a verified implementation for real and complex numbers, and it admits to compute roots and completely factor real and complex polynomials, provided that all coefficients are rational numbers. Moreover, we provide two implementations to display algebraic numbers, an injective and expensive one, and a faster but approximative version.
To this end, we mechanize several results on resultants, which also required us to prove that polynomials over a unique factorization domain form again a unique factorization domain. We moreover formalize algorithms for factorization of integer polynomials: Newton interpolation, factorization over the integers, and Kronecker's factorization algorithm, as well as a factorization oracle via Berlekamp's algorithm with the Hensel lifting.

1 Introduction

Algebraic numbers, i.e., the numbers that are expressed as roots of non-zero rational (equivalently, integer) polynomials, are an attractive subset of the real or complex numbers. They are closed under arithmetic operations, the arithmetic operations are precisely computable, and comparisons are decidable. As a consequence, algebraic numbers are an important utility in computer algebra systems.

Our original interest in algebraic numbers stems from a certification problem about automatically generated complexity proofs, where we have to compute the Jordan normal form of a matrix in $\mathbb{Q}^{n\times n}$ [16]. To this end, all complex roots of the characteristic polynomial have to be determined.

Example 1. Consider a matrix A whose characteristic polynomial is $f(x) = 1 + 2x + 3x^4$. The complex roots of f are exactly expressed via the real roots of $g = -1 - 12x^2 + 144x^6$ and $h = 7 - 216x^2 - 336x^4 - 1248x^6 + 1152x^8 + 6912x^{12}$:

root #1 of g + (*root #2 of* h)i	*root #1 of* g + (*root #3 of* h)i
root #2 of g + (*root #1 of* h)i	*root #2 of* g + (*root #4 of* h)i

Here, real roots are indexed according to the standard order. As the norms of all of these roots are strictly less than 1 (the norms are precisely root #3 and #4 of the polynomial $i = 1 - 3x^4 - 12x^6 - 9x^8 + 27x^{12}$), we can conclude that A^n tends to 0 for increasing n.

In this paper, we provide a fully verified and efficient implementation of algebraic numbers in Isabelle/HOL [14].

- The first problem in computation with algebraic numbers is to obtain a non-zero polynomial which represents a desired algebraic number as its root. To this end, we formalize the theory of *resultants*, and thus provide a verified computation of non-zero polynomials with desired roots (Section 2).
- A direct computation of resultants as determinant is infeasible in practice. Hence, we formalize a method based on a Euclid-like algorithm in combination with *polynomial remainder sequences* [1,5] (Section 3).
- Polynomials computed via resultants are often not optimal for representing an algebraic number, and lead to exponential growth of degrees during arithmetic operations. To avoid this problem, we formalize polynomial factorization algorithms, including an efficient oracle via Berlekamp's algorithm and the Hensel lifting, and an expensive but certified version of Kronecker's algorithm. To this end, we also formalize algorithms for prime factorization and polynomial interpolation, as well as Gauss' lemma. (Section 4)
- An algebraic number a is basically represented by a triple (f, l, r) of rational polynomial f and $l, r \in \mathbb{Q}$ such that a is the unique root of f within the interval $[l, r]$. To compute such an interval, we generalize the existing formalization of Sturm's method [6] to work over the rationals, and precompute the *Sturm sequence* to avoid recomputation. We also take special care for arithmetic operations involving a rational number, and finally provide a *quotient type* for algebraic numbers, which works modulo different representations of the same algebraic numbers. (Section 5)
- We also integrate complex algebraic numbers. Our algorithms cover complex root computation, as well as a factorization for rational polynomials over $\mathbb{R}$ or $\mathbb{C}$. (Section 6)
- Finally, we develop algorithms for displaying algebraic numbers. A challenge in precisely representing algebraic numbers is to ensure the uniqueness of string representation, independent from the internal representation. Here, the certified factorization algorithm plays a crucial role. (Section 7)

For the Coq proof assistant, the Mathematical Components library[1] contains various formalized results around algebraic numbers, e.g., quantifier elimination procedures for real closed fields [4]. In particular, the executable formalization of algebraic numbers for Coq is given by Cohen [2]. He employed Bézout's theorem to derive desired properties of resultants. In contrast, we followed proofs by Mishra [13] and formalized various facts on resultants. We further mechanize an algorithm to compute resultants, as well as the polynomial factorization algorithms. Our work is orthogonal to the more recent work which completely avoids resultants [3].

For Isabelle, Li and Paulson [11] independently implemented algebraic numbers. They however did not formalize resultants; instead, they employed an

[1] See `http://math-comp.github.io/math-comp`.

external tool as an oracle to provide polynomials that represent desired algebraic numbers, and provided a method to validate that the polynomials from the oracle are suitable.[2] Although we also use untrusted oracles for polynomial factorization, the difference is crucial. First, finding polynomials is indispensable for the computation of algebraic numbers, and hence their implementation is not ensured to always succeed. On the other hand, factorization is optional, and is employed only for efficiency. Second, in addition to an external oracle interface, we also provide an internal one, so that no external tools are required. Finally, due to our optimization efforts, we can execute their examples [11, Figure 3] in 0.03 seconds on our machine, where they reported 4.16 seconds.[3]

The whole formalization has been made available in the archive of formal proofs for Isabelle 2016 (`http://afp.sourceforge.net`), cf. entries Algebraic Numbers, Polynomial Factorization, and Polynomial Interpolation.

2 Resultants

In order to define arithmetic operations over algebraic numbers, the first task is the following: Given non-zero polynomials that have the input numbers as roots, compute a non-zero polynomial that has the output number as a root.

Consider an algebraic number a represented as a root of $f(x) = \sum_{i=0}^{m} f_i x^i$. To represent the unary minus $-a$, clearly *poly-uminus* f, defined as the polynomial $f(-x)$, does the job. For the multiplicative inverse $\frac{1}{a}$, it is also not difficult to show that *poly-inverse* f, defined as $\sum_{i=0}^{m} f_i x^{m-i}$, has $\frac{1}{a}$ as a root.

For addition and multiplication, given another polynomial $g(x) = \sum_{i=0}^{n} g_i x^i$ representing an algebraic number b, we must compose non-zero polynomials *poly-add* f g and *poly-mult* f g that have $a + b$ and $a \cdot b$ as a root, resp.

For this purpose the resultant is a well-known solution. The resultant of the polynomials f and g above is defined as $\mathrm{Res}(f, g) = \det(S_{f,g})$, where $S_{f,g}$ is the *Sylvester matrix* (blank parts are filled with zeros):

$$S_{f,g} = \begin{bmatrix} f_m & f_{m-1} & \cdots & f_0 & & \\ & \ddots & \ddots & & \ddots & \\ & & f_m & f_{m-1} & \cdots & f_0 \\ g_n & g_{n-1} & \cdots & g_0 & & \\ & \ddots & \ddots & & \ddots & \\ & & g_n & g_{n-1} & \cdots & g_0 \end{bmatrix}$$

In the remainder of this section, we consider addition – multiplication is treated similarly. The desired result is informally stated as follows, where *poly-add* f g is defined as the resultant of the two bivariate polynomials $f(x - y)$ and $g(y)$, where the resultant is a univariate polynomial over x.

[2] Here one cannot just evaluate the polynomial on the algebraic point and test the result is 0; we are defining the basic arithmetic operations needed for this evaluation.

[3] However, we use a faster computer with 3.5 GHz instead of 2.66 GHz.

Lemma 2. *Let f and g be non-zero univariate polynomials with roots a and b, respectively. Then poly-add f g is a non-zero polynomial having $a + b$ as a root.*

The lemma contains two claims: *poly-add f g* has $a + b$ as a root, and *poly-add f $g \neq 0$*. In the next sections we prove each of the claims.

2.1 Resultant has Desired Roots

For non-constant polynomials f and g over a commutative ring R, we can compute polynomials p and q such that

$$\text{Res}(f, g) = p(x) \cdot f(x) + q(x) \cdot g(x) \tag{1}$$

To formally prove the result, we first define a function *mk-poly* that operates on the Sylvester matrix. For each j-th column except for the last one, *mk-poly* adds the j-th column multiplied by x^{m+n-j} to the last column. Each addition preserves determinants, and we obtain the following equation:

$$\text{Res}(f, g) = \det(\textit{mk-poly } S_{f,g}) = \det \begin{bmatrix} f_m & \cdots & f_1 & f_0 & & f(x) \cdot x^{n-1} \\ & \ddots & & \ddots & \ddots & \vdots \\ & & f_m & \cdots & f_1 & f_0 & f(x) \cdot x \\ & & & f_m & \cdots & f_1 & f(x) \\ g_n & \cdots & g_1 & g_0 & & & g(x) \cdot x^{n-1} \\ & \ddots & & \ddots & \ddots & & \vdots \\ & & g_n & \cdots & g_1 & g_0 & g(x) \cdot x \\ & & & g_n & \cdots & g_1 & g(x) \end{bmatrix} \tag{2}$$

Now we apply the *Laplace expansion*, which we formalize as follows.

lemma ***assumes*** $A \in \textit{carrier}_m\ n\ n$ `(* meaning` $A \in R^{n \times n}$ `*)` ***and*** $j < n$
shows $\textit{det}\ A = (\sum i < n.\ A_{(i,\ j)} * \textit{cofactor}\ A\ i\ j)$

Here, *cofactor A i j* is defined as $(-1)^{i+j} \cdot \det(B)$, where B is the *minor matrix* of A obtained by removing the i-th row and j-th column. Thus we can remove the last column of the matrix A in (2), by choosing $j = m + n - 1$. Note that then every *cofactor A i j* is a constant. We obtain p and q in (1) as follows:

$$\text{Res}(f, g) = \left(\sum_{i=0}^{n-1} \textit{cofactor}\ A\ i\ j \cdot x^i\right) \cdot f(x) + \left(\sum_{i=0}^{m-1} \textit{cofactor}\ A\ (n+i)\ j \cdot x^i\right) \cdot g(x)$$

Lemma 3. ***assumes*** *degree $f > 0$* ***and*** *degree $g > 0$*
shows $\exists p\ q.\ \textit{degree}\ p < \textit{degree}\ g \wedge \textit{degree}\ q < \textit{degree}\ f \wedge$
$[\!:\textit{resultant}\ f\ g\ :\!] = p * f + q * g$

Here, *[: c :]* is Isabelle's notation for the constant polynomial c. The lemma implies that, if f and g are polynomials of positive degree with a common root a, then $\text{Res}(f, g) = p(a) \cdot f(a) + q(a) \cdot g(a) = 0$. The result is lifted to the bivariate case: for any a and b, $f(a, b) = g(a, b) = 0$ implies $\text{Res}(f, g)(a) = 0$.

lemma ***assumes*** *degree f* $>$ *0* $\vee$ *degree g* $>$ *0* ***and*** *poly2 f a b* $=$ *0*
and *poly2 g a b* $=$ *0*
shows *poly* (*resultant f g*) *a* $=$ *0*

Here, *poly* is Isabelle's notation for the evaluation of univariate polynomials, and *poly2* is our notation for bivariate polynomial evaluation.

Now for univariate non-zero polynomials f and g with respective roots a and b, the bivariate polynomials $f(x-y)$ and $g(y)$ have a common root at $x = a+b$ and $y = b$. Hence, the univariate polynomial *poly-add f g* $= \mathrm{Res}(f(x-y), g(y))$ indeed has $a+b$ as a root.

lemma ***assumes*** *g* $\neq$ *0* ***and*** *poly f a* $=$ *0* ***and*** *poly g b* $=$ *0*
shows *poly* (*poly-add f g*) (*a* $+$ *b*) $=$ *0*

2.2 Resultant is Non-Zero

Now we consider the second claim: *poly-add f g* is a non-zero polynomial. Note that it would otherwise have any number as a root. Somewhat surprisingly, formalizing this claim is more involving than the first one.

We first strengthen Lemma 3, so that p and q are non-zero polynomials. Here, we require an integral domain *idom*, i.e., there exist no zero divisors.

lemma ***assumes*** *degree f* $>$ *0* ***and*** *degree g* $>$ *0*
shows $\exists$ *p q. degree p* $<$ *degree g* $\wedge$ *degree q* $<$ *degree f* $\wedge$
[: resultant f g :] $=$ *p* $*$ *f* $+$ *q* $*$ *g* $\wedge$ *p* $\neq$ *0* $\wedge$ *q* $\neq$ *0*

We further strengthen this result, so that $\mathrm{Res}(f,g) = 0$ implies f and g share a common factor. This requires polynomials over a *unique factorization domain* (UFD), which is available as a locale *factorial-monoid* in HOL/Algebra, but not as a class. We define the class *ufd* by translating the locale as follows:

class *ufd* $=$ *idom* $+$
assumes *factorial-monoid* $\langle\!|$ *carrier* $=$ *UNIV* $-$ $\{$*0*$\}$, *mult* $=$ *op* $*$, *one* $=$ *1* $|\!\rangle$

We also show that polynomials over a UFD form a UFD, a non-trivial proof.

instance *poly* :: (*ufd*) *ufd*

Note also that the result is instantly lifted to any multivariate polynomials; if α is of sort *ufd*, then so is α *poly*, and thus so is α *poly poly*, and so on.

Now we obtain the following result, where $\mathit{coprime}_I$ generalizes the predicate *coprime* (originally defined only on the class *gcd*) over *idom* as follows:

definition $\mathit{coprime}_I$ *f g* $\equiv$ $\forall$*h. h dvd f* $\longrightarrow$ *h dvd g* $\longrightarrow$ *h dvd 1*

Lemma 4. ***assumes*** *degree f* $>$ *0* $\vee$ *degree g* $>$ *0* ***and*** *resultant f g* $=$ *0*
shows $\neg$ $\mathit{coprime}_I$ *f g*

Now we reason $\text{Res}(f(x-y), g(y)) \neq 0$ by contradiction. If $\text{Res}(f(x-y), g(y)) = 0$, then Lemma 4 implies that $f(x-y)$ and $g(y)$ have a common proper factor. This cannot be the case for complex polynomials: Let $f = f_1 \cdots f_m$ and $g = g_1 \cdots g_n$ be a complete factorization of the univariate polynomials f and g. Then the bivariate polynomials $f(x-y)$ and $g(y)$ are factored as follows:

$$f(x-y) = f_1(x-y) \cdots f_m(x-y) \qquad g(y) = g_1(y) \cdots g_n(y) \tag{3}$$

Moreover, this factorization is irreducible and unique (up to permutation and scalar multiplication). Since there is a common factor among $f(x-y)$ and $g(y)$, we must have $f_i(x-y) = g_j(y)$ for some $i \leq m$ and $j \leq n$. By fixing y, e.g., to 0, we conclude $f_i(x) = g_j(0)$ is a constant. This contradicts the assumption that f_i is a proper factor of f. We conclude the following result:

lemma ***assumes*** *f* $\neq$ *0* ***and*** *g* $\neq$ *0* ***and*** *poly f x = 0* ***and*** *poly g y = 0*
shows *poly-add f g* $\neq$ *0*

In order to ensure the existence of the complete factorization (3), our original formalization employs the fundamental theorem of algebra, and thus the above lemma is initially restricted to complex polynomials. Only afterwards the lemma is translated to rational polynomials via a homomorphism lemma for *poly-add*. In the development version of the AFP (May 2016), however, we have generalized the lemma to arbitrary field polynomials.

3 Euclid-Like Computation of Resultants

Resultants can be computed by first building the Sylvester matrix and then computing its determinant by transformation into row echelon form. A better way to compute resultants has been developed by Brown via subresultants [1], and a Coq formalization of subresultants exists [12]. We leave it as future work to formalize this algorithm in Isabelle. Instead, we compute resultants using ideas from Collins' primitive PRS (polynomial remainder sequences) algorithm [5].

3.1 The Algorithm and Its Correctness

The algorithm computes resultants $\text{Res}(f, g)$ in the manner of Euclid's algorithm. It repeatedly performs the polynomial division on the two input polynomials and replaces one input of larger degree by the remainder of the division.

We formalize the correctness of this algorithm as follows. Here we assume the coefficients of polynomials are in an integral domain which additionally has a division function such that $(a \cdot b)/b = a$ for all $b \neq 0$. Below we abbreviate $m =$ *degree f*, $n =$ *degree g*, $k =$ *degree r*, and $c =$ *leading-coeff g*.

Lemma 5 (Computation of Resultants).

1. *resultant f g* $= (-1)^{n*m} *$ *resultant g f*
2. ***assumes*** *d* $\neq$ *0* ***shows*** *resultant* $(d \cdot f)$ *g* $= d^n *$ *resultant f g*

3. ***assumes*** $f = g * q + r$ ***and*** $n \leq m$ ***and*** $k < n$
shows *resultant f g* $= (-1)^{n*(m-k)} * c^{m-k} *$ *resultant r g*

Lemma 5 (1) allows swapping arguments, which is useful for a concise definition of the Euclid-like algorithm. It is proven as follows: We perform a number of row swappings on the Sylvester matrix $S_{f,g}$ to obtain $S_{g,f}$. Each swap will change the sign of the resultant. In Isabelle, we exactly describe how the transformed matrix looks like after each row-swapping operation.

Lemma 5 (2) admits computing $\text{Res}(f, g)$ via $\text{Res}(d \cdot f, g)$. As we will see, this is crucial for applying the algorithm on non-field polynomials including bivariate polynomials, which we are dealing with. To prove the result in Isabelle, we repeatedly multiply the rows in $S_{f,g}$ by d, and obtain $S_{d \cdot f,g}$.

The most important step to the algorithm is Lemma 5 (3), which admits replacing f by the remainder r of smaller degree. A paper proof again applies a sequence of elementary row transformations to convert $S_{qg+r,g}$ into $S_{r,g}$. We formalize these transformation by a single matrix multiplication, and then derive the property in a straightforward, but tedious way.

To use Lemma 5 (3), we must compute a quotient q and a remainder r such that $f = gq + r$. For field polynomials one can just perform polynomial long division to get the corresponding q and r. For non-field polynomials, we formalize the polynomial *pseudodivision*, whose key property is formalized as follows:

lemma ***assumes*** $g \neq 0$ ***and*** *pseudo-divmod f g* $= (q, r)$
shows $c^{1+m-n} \cdot f = g * q + r \wedge (r = 0 \vee k < n)$

Now we compute $\text{Res}(f, g)$ as follows: Ensure $m \geq n$ using Lemma 5 (1), and obtain r via pseudodivision. We have $\text{Res}(f, g) = \text{Res}(c^{1+m-n} f, g)/c^{(1+m-n) \cdot n}$ by Lemma 5 (2), and $\text{Res}(c^{1+m-n} f, g)$ is simplified to $\text{Res}(g, r)$ by Lemma 5 (3), where the sum of the degrees of the input polynomials are strictly decreased.

The correctness of this reduction is formalized as follows:

lemma ***assumes*** *pseudo-divmod f g* $= (q, r)$ ***and*** $m \geq n$ ***and*** $n > k$
shows *resultant f g* $= (-1)^{n*m} *$ *resultant g r* $/ c^{(1+m-n)*n+k-m}$

We repeat this reduction until the degree n of g gets to zero, and then use the following formula to finish the computation.

lemma ***assumes*** $n = 0$ ***shows*** *resultant f g* $= c^m$

3.2 Polynomial Division in Isabelle's Class Hierarchy

When formalizing the algorithms in Isabelle (version 2016), we encountered a problem in the class mechanism. There is already the division for field polynomials formalized, and based on this the instance declaration "**instantiation** *poly :: (field) ring-div*", meaning that α *poly* is in class *ring-div* if and only if α is a field. Afterwards, one cannot have a more general instantiation, such as non-field polynomials to be in class *idom-divide* (integral domains with partial divisions).

As a workaround, we made a copy of *idom-divide* with a different name, so that it does not conflict with the current class instantiation.

Table 1. Identifying the complex roots of $1 + 2x + 3x^4$ as in Example 1.

	algorithm to compute resultants	overall time
(a)	algorithm of Section 3.1	> 24h
(b)	(a) + GCD before pseudodivision	30m32s
(c)	(b) with GCD for integer polynomials	34s

class *idom-div* = *idom* + ***fixes*** *exact-div* :: $\alpha \Rightarrow \alpha \Rightarrow \alpha$
assumes $b \neq 0 \Longrightarrow$ *exact-div* $(a * b)\ b = a$

For polynomials over α :: *idom-div*, we implement the polynomial long division. This is then used as *exact-div* for α *poly* and we provide the following instantiation (which also provides division for multivariate polynomials):[4]

instantiation *poly* :: (*idom-div*) *idom-div*

We further formalize pseudodivision which actually does not even invoke a single division and is thus applicable on polynomials over integral domains.

3.3 Performance Issues

The performance of the algorithm in Section 3.1 is not yet satisfactory, due to the repeated multiplication with c^{1+m-n}, a well-known phenomenon of pseudodivision. To avoid this problem, in every iteration of the algorithm we divide g by its *content*, i.e., the GCD of its coefficients, similar to Collins' primitive PRS algorithm. At this point a formalization of the subresultant algorithm will be benefitial as it avoids the cost of content computation.

We further optimize our algorithm by switching from $\mathbb{Q}$ to $\mathbb{Z}$. When invoking *poly-add*, etc., over polynomials whose coefficients are integers (but of type *rat*), we ensure that the intermediate polynomials have integer coefficients. Thus we perform the whole computation in type *int*, and also switch the GCD algorithm from the one for rational polynomials to the one for integer polynomials.

This has a significant side-effect: In Isabelle, the GCD on rational polynomials is already defined and it has to be normalized so that the leading coefficient of the GCD is 1. Thus, the GCD of the *rational* polynomials $1000(x + 1)x$ and $2000(x + 2)(x + 1)$ is just $x + 1$. In contrast, we formalized[5] Collins' primitive PRS algorithm for GCD computation for *integer* polynomials, where the GCD of the above example is $1000(x + 1)$. Hence, dividing by the GCD will eliminate large constants when working on $\mathbb{Z}$, but not when working on $\mathbb{Q}$.

Finally, we provide experimental data in Table 1 in order to compare the various resultant computation algorithms. In each experiment the complex roots

[4] We contributed our formalization to the development version of Isabelle (May 2016). There one will find the general "**instantiation** *poly :: (idom-divide) idom-divide*".

[5] As for the division algorithm, we have not been able to work with Isabelle's existing type class for GCDs, as the GCD on polynomials is only available for fields.

Table 2. Computation time/degree of representing polynomials for $\sum_{i=1}^{n} \sqrt{i}$.

factorization	$n = 6$	$n = 7$	$n = 8$	$n = 9$	$n = 10$
none	0.16s/64	2.78s/128	2m11s/256	22m19s/512	12h19m/1024
square-free	0.17s/64	2.86s/128	2m14s/256	15m31s/384	9h31m/768
complete	0.03s/8	0.14s/16	0.35s/16	0.35s/16	0.59s/16

for f of the leading Example 1 are identified. Here, the intermediate computation invokes several times the resultant algorithm on bivariate polynomials of degree 12. Note that in experiment (c) – which applies our final resultant implementation – only 17 % of the time is spent for the resultant computation, i.e., below 6 seconds.

This and all upcoming experiments have been performed using extracted Haskell code which has been compiled with `ghc -O2`, and has been executed on a 3.5 GHz 6-Core Intel Xeon E5 with 32 GB of RAM running Mac OS X.

4 Factorization of Rational Polynomials

Iterated resultant computations will lead to exponential growth in the degree of the polynomials. Hence, after computing a resultant to get a polynomial f representing an algebraic number a, it is a good idea to factor $f = f_1^{e_1} \cdots f_k^{e_k}$ and pick the only relevant factor f_i that has a as a root.

The benefit of factorization is shown in Table 2, where $\sum_{i=1}^{n} \sqrt{i}$ is computed for various n, and the computation time t and the degree d of the representing polynomial is reported as t/d. The table reveals that factorization becomes beneficial as soon as at it can simplify the polynomial.

We provide two approaches for the factorization of rational polynomials. First, we formalize Kronecker's algorithm. The algorithm serves as a verified and complete factorization, although it is not efficient. Second, we also employ factorization oracles, an untrusted code that takes a rational polynomial and gives a list of factors (and the leading coefficient). Validating factorization is easy: the product of the factors should be the input polynomial. On the other hand, completeness is not guaranteed, i.e., the factors are not necessarily irreducible.

4.1 Verified Kronecker's Factorization

We formalize Kronecker's factorization algorithm for integer polynomials. We also formalize Gauss' lemma, which essentially states that factorization over $\mathbb{Q}$ is the same as factorization over $\mathbb{Z}$; thus the algorithm works on rational polynomials. The basic idea of Kronecker's algorithm is to construct a finite set of lists of sample points, and for each list of sample points, one performs polynomial interpolation to obtain a potential factor f and checks if f divides the input polynomial. Formally proving the soundness of this algorithm is not challenging; however, many basic ingredients were not available in Isabelle.

For instance, in order to construct the set of lists of sample points, one has to compute all divisors of an integer $n \neq 0$. If not to be done naively, this basically demands a prime factorization of $|n|$, for which we did not find any useful existing algorithm that has been formalized in Isabelle.

Therefore, we formalize algorithm A of Knuth [9, Section 4.5.4] where the list of trial divisors currently excludes all multiples of 2, 3, and 5. Here, the candidate generation works via a function *next-candidates* that takes a lower bound n as input and returns a pair (m, xs) such that xs includes all primes in the interval $[n, m)$, provided that $n = 0$ or $n \ mod \ 30 = 11$. In the following definition, *primes-1000* is a precomputed list consisting of all primes up to 1000.

definition *next-candidates* n = (***if*** n = *0* ***then*** (*1001, primes-1000*)
else (n + *30, [n, n+2, n+6, n+8, n+12, n+18, n+20, n+26]*))

Similarly, we did not find formalized results on polynomial interpolation. Here, we integrate both Lagrange and Newton interpolation where the latter is more efficient. Furthermore, we formalize a variant of the Newton interpolation specialized for integer polynomials, which will abort early and conclude that no integer interpolation polynomial exists, namely as soon as the first division of two integers in the interpolation computation yields a non-zero remainder.

Finally, we integrate a divisibility test for integer polynomials, since polynomial divisibility test is by default available only for fields. The algorithm enjoys the same early abortion property as the Newton interpolation for integers.

4.2 Factorization Oracles

We provide two different factorization oracles: a small Haskell program that communicates with Mathematica, and an implementation within Isabelle/HOL. The latter can be used within an Isabelle session (*by eval*, etc.) as well as in generated Haskell or ML code.

They both use the same wrapper which converts the factorization over $\mathbb{Q}$ to a factorization over $\mathbb{Z}$, where the latter factorization can assume a square-free and content-free integer polynomial, represented as a coefficient list. The oracle is integrated as an unspecified constant:

consts *factorization-oracle-int-poly :: int list* $\Rightarrow$ *int list list*

The internal oracle implements Berlekamp's factorization algorithm in combination with Hensel lifting [9, Section 4.6.2]. Berlekamp's algorithm involves matrices and polynomials over finite fields ($\mathbb{Z}$ modulo some prime p). Here, we reuse certified code for polynomials and matrices whenever conveniently possible; however, the finite fields cannot be represented as a *type* in Isabelle/HOL since the prime p depends on the input polynomial to be factored. As a consequence, we could not use the standard polynomial library of Isabelle *directly*. Instead, we invoke the code generator to obtain the various certified algorithms on polynomials as ML-code, then manually replace the field operations by the finite field operations, and finally define these algorithms as new functions within Isabelle.

Table 3. Comparing factorization algorithms

	Berlekamp-Hensel	Mathematica	Kronecker
factorization of h, degree 12	0.0s	0.3s	0.6s
factorization of j, degree 27	0.0s	0.3s	$>$ 24h
evaluation of $\sum_{i=1}^{5} \sqrt[3]{i}$	17.8s	9.1s	–
evaluation of $\sum_{i=1}^{6} \sqrt[3]{i}$	63.9s	57.7s	–

Eventually, we had a view on all code equations for polynomials, and detected potential optimizations in the algorithm for polynomial long division.[6]

The same problem happens for the matrix operations; however, since the matrix theory is our formalization, we just modified it. We adjusted some of the relevant algorithms so that they no longer rely upon the type class *field*, but instead take the field operations as parameters. Then in the oracle we *directly* apply these generalized matrix algorithms, passing the field operations for finite fields as parameters.

We conclude this section with experimental data where we compare the different factorizations in Table 3. Here, polynomial h is taken from Example 1 and j is the unique minimal monic polynomial representing $\sum_{i=1}^{5} \sqrt[3]{i}$, which looks like $-64437024420 + 122730984540x + \ldots + x^{27}$.

The 0.3s of Mathematica is explained by its start-up time. We can clearly see that Kronecker's algorithm is no match against the oracles, which is why we did not even try Kronecker's algorithm in computing the sums of cubic roots examples – these experiments involve factorizations of polynomials of degree 81. At least on these examples, our internal factorization oracle seems to be not too bad, in comparison with Mathematica (version 10.2.0).

5 Real Algebraic Numbers

At this point, we have fully formalized algorithms which, given algebraic numbers a and b represented as roots of rational polynomials f and g, resp., computes a rational polynomial h having c as a root, where c is any of $a+b$, $a \cdot b$, $-a$, $\frac{1}{a}$, and $\sqrt[n]{a}$. To uniquely represent an algebraic number, however, we must also provide an interval $[l, r]$ in which c is the only root of h.

For $c = -a$ and $c = \frac{1}{a}$, bounds can be immediately given from the bound $[l, r]$ for a: take $[-r, -l]$ and $[\frac{1}{r}, \frac{1}{l}]$, resp. For the other arithmetic operations, we formalized various bisection algorithms.

5.1 Separation of Roots

Our main method to separate roots via bisection is based on a root-counting function ri_f for polynomial f, such that $ri_f\ l\ r$ is the number of roots of f in

[6] These optimizations became part of the development version of Isabelle (May 2016).

the interval $[l, r]$. Internally, ri_f is defined directly for linear polynomials, and is based on Sturm's method for nonlinear polynomials.

First, we extend the existing formalization of Sturm's method by Eberl [6], which takes a *real* polynomial and *real* bounds, so that it can be applied on *rational* polynomials with *rational* bounds; nevertheless, the number of *real* roots must be determined. This extension is crucial as we later implement the real numbers by the real algebraic numbers via *data refinement* [7]; at this point we must not yet use real number arithmetics. The correctness of this extension is shown mainly by proving that all algorithms utilized in Sturm's method can be homomorphically extended. For instance, for Sturm sequences we formalize the following result:

lemma *sturm* (*real-of-rat-poly f*) = *map real-of-rat-poly* (*sturm-rat f*)

For efficiency, we adapt Sturm's method for our specific purpose. Sturm's method works in two phases: the first phase computes a Sturm sequence, and the second one computes the number of roots by counting the number of sign changes on this sequence for both the upper and the lower bounds of the interval. The first phase depends only on the input polynomial, but not on the interval bounds. Therefore, for each polynomial f we precompute the Sturm sequence once, so that when a new interval is queried, only the second phase of Sturm's method has to be evaluated. This can be seen in the following code equation:

definition *count-roots-interval-rat f* =
  (***let*** *fs* = *sturm-squarefree-rat f* `(* precompute *)`
   in ... (λ *l r. sign-changes-rat fs l* − *sign-changes-rat fs r* + ...) ...)

For this optimization, besides the essential (f, l, r) our internal representation additionally stores a function $ri :: \mathbb{Q} \to \mathbb{Q} \to \mathbb{N}$ which internally stores the precomputed Sturm sequence for f.

With the help of the root-counting functions, it is easy to compute a required interval. For instance, consider the addition of a and b, each represented by (f, l_a, r_a) and (g, l_b, r_b), and we already have a polynomial h which has $a + b$ as one of its roots. If $ri_h\ (l_a + l_b)\ (r_a + r_b) = 1$, then we are done. Otherwise, we repeat bisecting the intervals $[l_a, r_a]$ and $[l_b, r_b]$ with the help of ri_f and ri_g. Similar bisections are performed for multiplication and n-th roots.

For further efficiency, we formalize the bisection algorithms as partial functions [10]. This is motivated by the fact that many of these algorithms terminate only on valid inputs, and runtime checks to ensure termination would be an overhead. In order to conveniently prove the correctness of the algorithms, we define some well-founded relations for inductive proofs, which are reused for various bisection algorithms. For instance, we define a relation based on a decrease in the size of the intervals by at least δ, where δ is the minimal distance of two distinct roots of some polynomial.

Finally, we tighten the intervals more than what is required to identify the root. This is motivated as follows. Assume that the interval $[2, 10000]$ identifies a real root $a \approx 3134.2$ of a polynomial f. Now, consider computing the floor $\lfloor a \rfloor$,

which requires us to bisect the interval until we arrive at $[3134.003, 3134.308]$. It would be nice if we could update the bounds for a to the new tighter interval at this point. Unfortunately, we are not aware of how this can be done in a purely functional language. Hence, every time we invoke $\lfloor a \rfloor$ or other operations which depends on a, we have to redo the bisection from the initial interval. Therefore, it is beneficial to compute sufficiently tight intervals whenever constructing algebraic numbers. Currently we limit the maximal size of the intervals by $\frac{1}{8}$.

5.2 Comparisons of Algebraic Numbers

Having defined all arithmetic operations, we also provide support for comparisons of real algebraic numbers, as well as membership test in $\mathbb{Q}$, etc. For membership in $\mathbb{Q}$, we formalize the rational root test which we then integrate into a bisection algorithm. Comparison is, in theory, easy: just compute $x - y$ and determine its sign, which is trivial, since we have the invariant that the signs of the interval bounds coincide. This naive approach however requires an expensive resultant computation. Hence we pursue the following alternative approach: To compare two algebraic numbers a and b, represented by (f, l_a, r_a) and (g, l_b, r_b),

- we first decide[7] $a = b$ by testing whether $gcd\ f\ g$ has a root in $[l_a, r_a] \cap [l_b, r_b]$. The latter property can be determined using Sturm's method; and
- if $a \neq b$, then bisect the intervals $[l_a, r_a]$ and $[l_b, r_b]$ until they become disjoint. Afterwards we compare the intervals to decide $a < b$ or $a > b$.

Note that the recursive bisection in the second step is terminating only if it is invoked with $a \neq b$. At this point, Isabelle's **partial-function** command becomes essential. Note also that specifying the algorithm via **function** prohibits code generation.

If we had a proof that the internal polynomials are irreducible, then the first step could be done more efficiently, since then $f \neq g$ implies $a \neq b$. We leave it for future work to formalize more efficient factorization algorithms.

5.3 Types for Real Algebraic Numbers

As the internal representation of algebraic numbers, besides the essential (f, l, r) and already mentioned ri, we store another additional information: a flag ty of type *poly-type*, indicating whether f is known to be monic and irreducible (*Monic-Irreducible*) or whether this is unknown (*Arbitrary-Poly*). We initially choose *Arbitrary-Poly* as ty for non-linear polynomials, and *Monic-Irreducible* for linear polynomials after normalizing the leading coefficient. If we have a complete factorization, we may set the polynomial type to *Monic-Irreducible*; however, this would require the invocation of the slow certified factorization algorithm.

In the formalization we create a corresponding type abbreviation for the internal representation (an option type where *None* encodes the number 0), then define an invariant *rai-cond* which should be satisfied, and finally enforce this

[7] We thank one of the anonymous reviewers for pointing us to this equality test.

invariant in the type *real-alg-intern*. For the specification of algorithms on type *real-alg-intern*, the lifting and transfer package has been essential [8].

type-synonym *rai-intern* = (*poly-type* × *root-info* × *rat poly* × *rat* × *rat*) *option*
definition *rai-cond tuple* = (***case*** *tuple* ***of*** *Some* (*ty,ri,f,l,r*) ⇒
f ≠ *0* ∧ *unique-root f l r* ∧ *sgn l* = *sgn r* ∧ *sgn r* ≠ *0* ∧ ... | *None* ⇒ *True*)
typedef *real-alg-intern* = *Collect rai-cond*

Then, all arithmetic operations have been defined on type *real-alg-intern*.

In order to implement the real numbers via real algebraic numbers, we did one further optimization, namely integrate dedicated support for the rational numbers. The motivation is that most operations can be implemented more efficiently, if one or both arguments are rational numbers. For instance, for addition of a rational number with a real algebraic number, we provide a function *add-rat-rai :: rat* ⇒ *real-alg-intern* ⇒ *real-alg-intern* which does neither require a resultant computation, nor a factorization.

Therefore, we create a new datatype *real-alg-dt*, which has two constructors: one for the rational numbers, and one for the real algebraic numbers whose representing polynomial has degree at least two. This invariant on the degree is then ensured in a new type *real-alg-dtc*, and the final type for algebraic numbers is defined as a quotient type *real-alg* on top of *real-alg-dtc*, which works modulo different representations of the same real algebraic numbers. Here, *real-of-radtc* is the function that delivers the real number which is represented by a real algebraic number of type *real-alg-dtc*.[8]

quotient-type *real-alg* = *real-alg-dtc* / λ *x y*. *real-of-radtc x* = *real-of-radtc y*

Now we provide the following code equations to implement the real numbers via real algebraic numbers by data refinement, where *real-of :: real-alg* ⇒ *real* is converted into a constructor in the generated code.

lemma *plus-real-alg[code]:* (*real-of x*) + (*real-of y*) = *real-of* (*x* + *y*)
`(* similar code lemmas for =, <, −, *, /, floor, etc. *)`

Note that in the lemma *plus-real-alg*, the left-hand side of the equality is addition for type *real*, whereas the right is addition of type *real-alg*.

We further prove that real algebraic numbers form an Archimedean field.

instantiation *real-alg :: floor-ceiling* `(* includes Archimedean field *)`

Finally, we provide a function *real-roots-of-rat-poly :: rat poly* ⇒ *real list* which computes all real roots of a non-zero rational polynomial. It first factors the polynomial, and then for each factor it either uses a closed form to determine the roots, or computes intervals that uniquely identify each root of the factor and returns the corresponding real algebraic numbers. Below, *rpoly* denotes the evaluation of a rational polynomial at a real or complex point.

[8] Note that the quotient type can be in principle defined also directly on top of *real-alg-dt*, such that the quotient and invariant construction is done in one step, but then code generator will fail in Isabelle 2016.

lemma ***assumes*** $f \neq 0$
shows *set* (*real-roots-of-rat-poly f*) = {*a* :: *real*. *rpoly f a* = 0}

6 Complex Algebraic Numbers

All of the results on resultants have been developed in a generic way, i.e., they are available for both real and complex algebraic numbers. Hence, in principle one can pursue a similar approach as in Section 5 to integrate complex algebraic numbers, one just has to replace Sturm's method by a similar method to separate complex roots, e.g., by using results of Kronecker [15, Section 1.4.4].

Since we are not aware of any formalization of such a method, instead we just stick to Isabelle's implementation of complex numbers, i.e., pairs of real numbers representing the real and imaginary part. Note that this is also possible in the algebraic setting: a complex number is algebraic if and only if both the real and the imaginary part are algebraic.

With this representation, all of the following operations become executable on the complex numbers for free: $+$, $-$, $*$, $/$, $\sqrt{\cdot}$, $=$, and complex conjugate. These operations are already implemented via operations on the real numbers, and those are internally computed by real algebraic numbers via data refinement.

The only operation that is not immediate is a counterpart of *real-roots-of-rat-poly* – a method to determine all complex roots of a rational polynomial f. Here, the algorithm proceeds as follows, excluding optimizations.

- Consider a complex root $a+b\mathrm{i}$ of f for $a, b \in \mathbb{R}$. Since $a = \frac{1}{2}((a+b\mathrm{i})+(a-b\mathrm{i}))$, a is a root of the rational polynomial g = *poly-mult-rat* $\frac{1}{2}$ (*poly-add f f*). Here, the first f in *poly-add f f* represents $a + b\mathrm{i}$ and the second f represents $a - b\mathrm{i}$; complex conjugate numbers share the same representing polynomials. Similarly, since $b = \frac{1}{2\mathrm{i}}((a + b\mathrm{i}) - (a - b\mathrm{i}))$, b is a root of h = *poly-mult [:1,0,4:]* (*poly-add f* (*poly-uminus f*)), where *[:1,0,4:]* is the polynomial $1 + 4x^2$ with root $\frac{1}{2\mathrm{i}}$.
- Let C be the set of all numbers $a+b\mathrm{i}$ such that $a \in$ *real-roots-of-rat-poly g* and $b \in$ *real-roots-of-rat-poly h*. Then C contains at least all roots of f. Return $\{c \in C.\ f(c) = 0\}$ as the final result.

The actual formalization of *complex-roots-of-rat-poly* contains several special measures to improve the efficiency, e.g., factorizations are performed in between, explicit formulas are used, etc. The soundness result looks as in the real case.

lemma ***assumes*** $f \neq 0$
shows *set* (*complex-roots-of-rat-poly f*) = {*a* :: *complex*. *rpoly f a* = 0}

The most time-consuming task in *complex-roots-of-rat-poly* is actually the computation of $\{c \in C.\ f(c) = 0\}$ from C. For instance, when testing $f(c) = 0$ in Example 1, multiplications like $b \cdot b$ occur. These result in factorization problems for polynomials of degree 144.

With the help of the complex roots algorithm and the fundamental theorem of algebra, we further develop two algorithms that factor polynomials with rational

coefficients over $\mathbb{C}$ and $\mathbb{R}$, resp. Factorization over $\mathbb{C}$ is easy, since then every factor corresponds to a root. Hence, the algorithm and the proof mainly take care of the multiplicities of the roots and factors. Also for the real polynomials, we first determine the complex roots. Afterwards, we extract all real roots and group each pair of complex conjugate roots. Here, the main work is to prove that for each complex root c, its multiplicity is the same as the multiplicity of the complex conjugate of c.

7 Displaying Algebraic Numbers

We provide two approaches to display real algebraic numbers.

The first one displays the approximative value of an algebraic number a. Essentially, the rational number $\frac{\lfloor 1000a \rfloor}{1000}$ is computed and displayed as string. For instance, the first root of polynomial g in Example 1 is displayed as "$\sim$ -0.569".

The second approach displays a number represented by (ty, ri, f, l, r) exactly as the string "root $\#n$ of f", provided that $ty =$ *Monic-Irreducible* and that n is the number of roots of f in the interval $(-\infty, r]$. In order to determine the value of n, we just apply Sturm's method. In case $ty \neq$ *Monic-Irreducible*, at this point we invoke the expensive certified factorization.

Note that displaying a number must be a function of type *real-alg* $\Rightarrow$ *string*, i.e., the resulting string must be independent of the representative. Clearly, this is the case for the first approach. For the second approach we need a uniqueness result, namely that every algebraic number a is uniquely represented by a monic and irreducible polynomial. To this end, we first formalize the result, that the GCD of two rational polynomials stays the same if we embed $\mathbb{Q}$ into $\mathbb{R}$ or $\mathbb{C}$.

lemma *map-poly of-rat* (*gcd f g*) = *gcd* (*map-poly of-rat f*) (*map-poly of-rat g*)

Using this lemma, we provide the desired uniqueness result.

lemma ***assumes*** *algebraic a* ***shows*** $\exists!\, f.$ *alg-poly a f* $\wedge$ *monic f* $\wedge$ *irreducible f*

Our formalization of this statement works along the following line. Assume f and g are two different monic and irreducible rational polynomials with a common real or complex root a. That is, f and g have a common factor $x - a$ as a real or complex polynomial and hence, the GCD of f and g (over $\mathbb{R}$ or $\mathbb{C}$) is a non-constant polynomial. On the other hand, the GCD of f and g over $\mathbb{Q}$ must be a constant: it cannot be a proper factor of f or g since the polynomials are irreducible over $\mathbb{Q}$, and it cannot be f or g itself, since this contradicts monicity and $f \neq g$.

8 Conclusion

We integrated support for real and complex algebraic numbers in Isabelle/HOL. Although all arithmetic operations are supported, there remain some open tasks.

A formalization of an equivalent to Sturm's method for the complex numbers would admit to represent the roots in Example 1 just as root #(1,2,3,4) of f, without the need for high-degree polynomials for the real and imaginary part.

A certified efficient factorization algorithm would also be welcome: then the implementation of comparisons of algebraic numbers could be simplified and it would allow to display more algebraic numbers precisely within reasonable time.

Finally, it would be useful to algorithmically prove that the complex algebraic numbers are algebraically closed, so that one is not restricted to rational coefficients in the factorization algorithms over $\mathbb{R}$ and $\mathbb{C}$.

Acknowledgments We thank the anonymous reviewers for their helpful comments. The early abortion in our divisibility test for integer polynomials is due to Sebastiaan Joosten. This research was supported by the Austrian Science Fund (FWF) project Y757.

References

1. Brown, W.S.: The subresultant PRS algorithm. ACM Trans. Math. Softw. 4(3), 237–249 (1978)
2. Cohen, C.: Construction of real algebraic numbers in Coq. In: ITP 2012. LNCS, vol. 7406, pp. 67–82 (2012)
3. Cohen, C., Djalal, B.: Formalization of a Newton series representation of polynomials. In: CPP 2016. pp. 100–109. ACM (2016)
4. Cohen, C., Mahboubi, A.: Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. Logical Methods in Computer Science 8(1:02), 1–40 (2012)
5. Collins, G.E.: Subresultants and reduced polynomial remainder sequences. Journal of the ACM 14, 128–142 (1967)
6. Eberl, M.: A decision procedure for univariate real polynomials in Isabelle/HOL. In: CPP 2015. pp. 75–83. ACM (2015)
7. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: ITP 2013. LNCS, vol. 7998, pp. 100–115 (2013)
8. Huffman, B., Kunčar, O.: Lifting and transfer: A modular design for quotients in Isabelle/HOL. In: CPP 2013. LNCS, vol. 8307, pp. 131–146 (2013)
9. Knuth, D.E.: The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition. Addison-Wesley (1981)
10. Krauss, A.: Recursive definitions of monadic functions. In: PAR 2010. EPTCS, vol. 43, pp. 1–13 (2010)
11. Li, W., Paulson, L.C.: A modular, efficient formalisation of real algebraic numbers. In: CPP 2016. pp. 66–75. ACM (2016)
12. Mahboubi, A.: Proving formally the implementation of an efficient gcd algorithm for polynomials. In: IJCAR 2006. LNCS, vol. 4130, pp. 438–452 (2006)
13. Mishra, B.: Algorithmic Algebra. Texts and Monographs in Computer Science, Springer (1993)
14. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283 (2002)
15. Prasolov, V.V.: Polynomials. Springer (2004)
16. Thiemann, R., Yamada, A.: Formalizing Jordan normal forms in Isabelle/HOL. In: CPP 2016. pp. 88–99. ACM (2016)