Journal of Automated Reasoning (2020) 64:363–389
https://doi.org/10.1007/s10817-018-09504-w

A Verified Implementation of Algebraic Numbers in Isabelle/HOL

Sebastiaan J. C. Joosten[1] · **René Thiemann**[1] · **Akihisa Yamada**[1]

Received: 10 October 2018 / Accepted: 29 November 2018 / Published online: 9 December 2018

Abstract
We formalize algebraic numbers in Isabelle/HOL. Our development serves as a verified implementation of algebraic operations on real and complex numbers. We moreover provide algorithms that can identify all the real or complex roots of rational polynomials, and two implementations to display algebraic numbers, an approximative version and an injective precise one. We obtain verified Haskell code for these operations via Isabelle's code generator. The development combines various existing formalizations such as matrices, Sturm's theorem, and polynomial factorization, and it includes new formalizations about bivariate polynomials, unique factorization domains, resultants and subresultants.

Keywords Theorem proving · Algebraic numbers · Real algebraic geometry · Resultants

1 Introduction

Algebraic numbers, i.e., the numbers that are expressed as roots of non-zero integer (or equivalently rational) polynomials, are an attractive subset of the real or complex numbers. Every satisfiable polynomial constraint has solutions in the domain of algebraic numbers; in particular, algebraic numbers are closed under arithmetic operations (addition, multiplication, integer powers, and there inverses). Moreover these arithmetic operations are precisely computable, and comparisons of algebraic numbers are decidable. As a consequence, algebraic numbers are an important utility in computer algebra systems; e.g., Collin's cylindrical algebraic decomposition algorithm for solving problems in real algebraic geometry heavily relies upon algebraic numbers [20, Sect. 8.6.5].

Our original interest in algebraic numbers stems from a certification problem about automatically generated complexity proofs, where for a given matrix $A \in \mathbb{Q}^{n \times n}$ we have to

This research was supported by the Austrian Science Fund (FWF) project Y757. The authors are listed in alphabetical order regardless of individual contributions or seniority. Sebastiaan is now working at University of Twente, the Netherlands, and Akihisa at National Institute of Informatics, Japan. We thank the reviewers of this paper for their helpful comments.

✉ René Thiemann
rene.thiemann@uibk.ac.at

[1] University of Innsbruck, Innsbruck, Austria

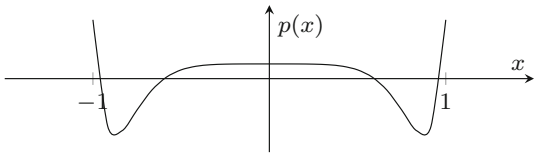

Fig. 1 The polynomial p with its four real roots

compute the growth rate of A^n for increasing n [25]. To this end, all complex roots of the characteristic polynomial of A have to be identified.

Example 1 Consider a matrix A whose characteristic polynomial is $f(x) = \frac{1}{3} \cdot (1+2x+3x^4)$ and let $\lambda_1, \ldots, \lambda_4$ be the complex roots of f. If the norm of some λ_i is larger than 1, then the growth rate of A is exponential; if all norms are below 1, then A^n tends to 0; and otherwise the growth rate is polynomial and its degree can be determined by further computations.

In order to apply this criterion, we need to compute each norm $|\lambda_i| = \sqrt{Re(\lambda_i)^2 + Im(\lambda_i)^2}$, and afterwards compare it with 1. All these computations can be performed with the help of algebraic numbers, as we will see throughout this paper. For instance, in this example we obtain the following results where roots are indexed from the smallest to the largest, and where the polynomials $g = -1 - 12x^2 + 144x^6$, $h = 7 - 216x^2 - 336x^4 - 1248x^6 + 1152x^8 + 6912x^{12}$, and $p = 1 - 3x^4 - 12x^6 - 9x^8 + 27x^{12}$ are constructed during the computation.

$$\begin{array}{lll}
Re(\lambda_1) = root\ \#1\ of\ g, & Im(\lambda_1) = root\ \#2\ of\ h, & |\lambda_1| = root\ \#3\ of\ p \\
Re(\lambda_2) = root\ \#1\ of\ g, & Im(\lambda_2) = root\ \#3\ of\ h, & |\lambda_2| = root\ \#3\ of\ p \\
Re(\lambda_3) = root\ \#2\ of\ g, & Im(\lambda_3) = root\ \#1\ of\ h, & |\lambda_3| = root\ \#4\ of\ p \\
Re(\lambda_4) = root\ \#2\ of\ g, & Im(\lambda_4) = root\ \#4\ of\ h, & |\lambda_4| = root\ \#4\ of\ p
\end{array}$$

As each norm $|\lambda_i|$ is below 1, cf. Fig. 1, we can conclude that A^n tends to 0 for increasing n.

In this paper, we describe an implementation of algebraic numbers in Isabelle/HOL [21]. Up to our knowledge it is the first implementation that is both fully verified and executable on its own, i.e., without information from external tools. The implementation already became a crucial component of the automated reasoning tool CeTA for verifying complexity proofs [1,25] that are generated during the annual termination competitions [12]. It is also used within a verified solver for linear recurrences [10].

The paper is structured as follows.

- We first introduce some basic notions of algebraic numbers and then formalize the fact that every algebraic number has a unique canonical polynomial that represents it. We argue that these canonical polynomials make a good internal representation (Sect. 2).
- For each algebraic operation, we formalize how to synthesize a polynomial that represents the output using polynomials that represent the inputs. We thus show that algebraic numbers are closed under the algebraic operations (Sect. 3).
- For the multiplication and addition of algebraic numbers, we refer to *resultants*. We implement and formalize the *subresultant remainder sequence algorithm*, which can efficiently compute resultants (Sect. 4).
- Using the above results, we implement the algebraic operations and comparisons of real and complex algebraic numbers, and a function to uniquely convert them into strings. We develop a hierarchy of four layers to represent real algebraic numbers, formalize several bisection algorithms, and integrate optimizations to obtain efficient code (Sect. 5).

– We moreover provide algorithms that identify all real and complex roots of a rational polynomial. Together with the fact that complex roots of a real polynomial come in complex conjugate pairs, we derive algorithms that completely factor rational polynomials into real or complex polynomial factors (Sect. 6).
– As we made an effort for efficiency, we experimentally compare the implementation against a version described in a preliminary version [24] of this paper, and the commercial computer algebra tool Wolfram Mathematica 11 (Sect. 7).

Most of the algorithms and proofs of our formalization are based on a textbook by Mishra [20, Chapters 7 and 8]; it contains a detailed implementation of real algebraic numbers, including proofs. When it comes to subresultants, we followed the original papers by Brown and Traub [2,3]. However, our formalization also includes algorithms and optimizations, which we did not find in the literature, though they might be known.

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 [6]. In particular, the formalization of algebraic numbers for Coq is given by Cohen [4]. He employed Bézout's theorem to derive desired properties of resultants, while we followed proofs by Mishra [20] and formalized various facts on resultants. Our work is orthogonal to the more recent work which avoids resultants [5]. A partial Coq formalization of subresultants also exists [19]. In contrast, our formalization is complete, and also integrates an optimization due to Ducos [8, Sect. 2].

For Isabelle, Li and Paulson [18] independently implemented algebraic numbers. They however did not formalize resultants; instead, they employed an 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] Due to our optimization efforts, we can execute their examples [18, Fig. 3] in 0.016 seconds on our machine, where they reported 4.16 seconds.[3]

The whole formalization is available in the archive of formal proofs (AFP), mostly in entries `Algebraic Numbers` and `Subresultants`. Additionally, on

https://doi.org/10.5281/zenodo.1411394

we link statements in the paper with the Isabelle sources and provide details on our experiments.

2 Representation of Algebraic Numbers

Our formalization is based on Isabelle/HOL, and we state theorems and definitions following Isabelle's syntax. For instance, *of-int* :: *int* $\Rightarrow \alpha$ :: *ring-1* indicates that *of-int* is a function that takes integers and returns elements of type α, which is of class *ring-1*. The type of polynomials over coefficients of type α is denoted by α *poly*. In Isabelle, a polynomial $f(x) = \sum_{i=0}^{m} f_i x^i$ is written as [: $f_0, \ldots, f_m$:] (the leading coefficient comes last in the list), *coeff f i* denotes the coefficient f_i, *degree f* the degree m, and *poly f a* the evaluation $f(a)$ at $a :: \alpha$.

A number a is *algebraic* if it is a root of a non-zero integer polynomial f. The notion is defined in Isabelle 2018 as follows.

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

[2] The suitability test cannot simply evaluate the polynomial on the algebraic point and test whether the result is 0; evaluating at an algebraic point requires the basic arithmetic operations on algebraic numbers, which are the operations we are defining in this work.

[3] However, we use a faster computer with 3.2 GHz instead of 2.66 GHz.

Definition 1 *algebraic* $a \equiv \exists f.\ (\forall i.\ coeff\ f\ i \in \mathbb{Z}) \wedge f \neq 0 \wedge poly\ f\ a = 0$

Here the condition that f is an integer polynomial is expressed by enforcing the coefficients of f to be in the set $\mathbb{Z}$, which is of type α *set*. In this definition, the polynomial $f :: \alpha$ *poly* and the algebraic number $a :: \alpha$ share the same domain type α, which will be instantiated by *real* or *complex*. Since our motivation is to implement functions that actually operate on algebraic numbers of type *real* and *complex*, manipulating polynomials in type α *poly* leads to a circular dependency. Hence we introduce the predicate *f represents a*, meaning that a non-zero *integer* polynomial $f ::$ *int poly* has $a :: \alpha$ as a root.

Definition 2 *f represents a* $\equiv$ *ipoly f a* $= 0 \wedge f \neq 0$

Here, *ipoly f a* is an abbreviation for *poly* (*of-int-poly f*) *a*, and *of-int-poly* :: *int poly* $\Rightarrow \alpha$ *poly* converts the type of integer polynomials.

We obtain the following alternative characterization of algebraic numbers.

Lemma 1 *algebraic a* $\longleftrightarrow$ ($\exists f.$ *f represents a*)

2.1 Unique Representation

An algebraic number can be represented by arbitrarily many polynomials; for instance $\sqrt{2}$ is represented by $f(x) = x^2 - 2$, $g(x) = -x^2 + 2$, $h(x) = x^4 + 2x^3 - 4x - 4$, $k(x) = 2x^2 - 4$, etc. However, every algebraic number can be uniquely represented by an integer polynomial which has no non-trivial divisors and a positive leading coefficient. The degree of this unique representative polynomial is called the degree of the algebraic number. For instance, f is this unique representative of $\sqrt{2}$, whereas g has a negative leading coefficient, and h is reducible as $h(x) = f(x) \cdot (x^2 + 2x + 2)$.

Irreducibility of a polynomial often means that it is non-constant and has no non-constant divisor of smaller degree, and in the preliminary version of this work [24] we used such a definition. Isabelle 2018, however, uses the following predicate *irreducible* for arbitrary commutative rings.

Definition 3 *irreducible* $f \equiv$
$\neg\ f\ dvd\ 1 \wedge f \neq 0 \wedge (\forall\ g\ h.\ f = g \cdot h \longrightarrow g\ dvd\ 1 \vee h\ dvd\ 1)$

Here *dvd* is Isabelle's notation for divisibility. This definition is stronger than the polynomial-specific version. In particular, *irreducible f* for non-constant integer polynomial f demands that f is *content-free*, i.e., the GCD of the coefficients of f is 1; otherwise, f is "reducible" by the GCD. For instance, the integer polynomial k above is reducible since $k(x) = (x^2 - 2) \cdot 2$. Note also that the definitions are equivalent on field polynomials.

We adopt this stronger definition in the current work, and formulate the uniqueness statement as follows.

Lemma 2 ***assumes*** *algebraic a*
shows $\exists!$*f. f represents a* $\wedge$ *irreducible f* $\wedge$ *lead-coeff f* > 0

Typical uniqueness results found in the literature (e.g., [11, pp. 700] and [20, pp. 319]) state that there is a unique representative polynomial of the *minimum degree*. Our claim is more useful for computing the unique representative: if we find any irreducible polynomial representing a number, then we do not have to search for other polynomials of lower degree that represent the same number. The typical statement is easily obtained from Lemma 2; actually the irreducible representative polynomial is of the minimum degree.

Corollary 1 ***assumes*** *irreducible f* ***and*** *f represents a* ***and*** *g represents a*
shows *degree f* ≤ *degree g*

To prove Lemma 2 we first show that polynomials over a *unique factorization domain* (UFD) forms a UFD again. Whereas the class *factorial-ring* was (independently) introduced to Isabelle 2016-1 for UFDs, it demands several extra operations, i.e., *unit-factor*, *normalize*, *gcd*, etc., to derive that polynomials over a UFD form a UFD. We instead define a more general class *ufd*:

class *ufd* = *idom* +
assumes $f \neq 0 \Longrightarrow \neg f\ dvd\ 1 \Longrightarrow \exists F.$ *mset-factors F f*
and *mset-factors F f* ⟹ *mset-factors G f* ⟹ *rel-mset* (*ddvd*) *F G*

Here the first assumption claims that for every non-zero and non-unit element f in the domain, there exists a multiset F of irreducible factors of f, denoted by the predicate *mset-factors F f*. The second assumption claims that for any element f, any two irreducible factorizations F and G of f are "associated", i.e., F and G contain the same number of factors, $f_1, \ldots, f_n$ and $g_1, \ldots, g_n$, such that $f_1\ ddvd\ g_1, \ldots, f_n\ ddvd\ g_n$. Here $a\ ddvd\ b$ is defined as $a\ dvd\ b \wedge b\ dvd\ a$.

In this general setting we show that polynomials over a UFD form a UFD.

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

This 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. This is crucial for formalizing addition and multiplication of algebraic numbers, where we extensively use bivariate polynomials.

We also establish a connection between *ufd* and the already existing locale *factorial-monoid*. Thus we can derive results from *factorial-monoid*, e.g., that irreducibility and primality are equivalent in UFDs. This yields that for an irreducible integer polynomial f with positive leading coefficient, the GCD of f and any polynomial g is either 1 or f itself:

Lemma 3 ***assumes*** *irreducible f* ***and*** *lead-coeff f* > 0
shows *gcd f g* ∈ {1, *f*}

To prove Lemma 2 we further show that the GCD of two integer polynomials stays the same up to a constant factor if we embed $\mathbb{Z}$ into $\mathbb{R}$ or $\mathbb{C}$.

Lemma 4 *gcd* (*of-int-poly f*) (*of-int-poly g*) =
inverse (*of-int* (*lead-coeff* (*gcd f g*))) · *of-int-poly* (*gcd f g*)

Our proof of Lemma 2 then works as follows: Assume that f and g are two different, positive and irreducible integer polynomials with a common real or complex root a. That is, f and g as real or complex polynomials have a common factor $x - a$ and hence, their GCD is a non-constant polynomial. On the other hand, the GCD of f and g as integer polynomials must be 1: it cannot be f or g itself, since $f \neq g$.

2.2 Unique Representation or Not?

Despite the existence of a unique (and minimal) representative polynomial of an algebraic number, it is *a priori* questionable whether it is a good choice in an implementation to stick to the unique representative polynomials. There is a trade-off between the cost of computing unique representatives from arbitrary representations via polynomial factorization, and the penalty of not using minimal representations in a sequence of operations.

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

Factorization	$n = 6$	$n = 7$	$n = 8$	$n = 9$	$n = 10$
Square-free	0.054s/64	0.807s/128	19.725s/256	3m19s/384	1h48m/768
Complete	0.019s/8	0.044s/16	0.080s/16	0.080s/16	0.117s/16

We answer this question experimentally by computing representations of the algebraic numbers $\sum_{i=1}^{n} \sqrt{i}$ for various n. In one configuration we stick to the unique representatives and perform complete polynomial factorization after each addition. In another configuration we only perform the efficient square-free factorization that eliminates duplicate factors.

The result is reported in Table 1, where the computation time t and the degree d of the representing polynomial is reported as t/d. Here the benefit of complete factorization is clear; the growth of the degrees is so rapid that manipulating the high-degree polynomials is more costly than applying complete factorization each time.

Hence we choose irreducible polynomials for representing algebraic numbers; however for those of degree 1, i.e., the rational numbers, there is already an efficient implementation. When implementing a binary arithmetic operation on algebraic numbers, we actually implement two variants: one on a rational and an algebraic number, and another one on two algebraic numbers. The former variant is faster to execute than the more generic latter one. This special treatment for rational numbers explains why there is no measurable difference in Table 1 in the computation time of $\sum_{i=1}^{8} \sqrt{i}$ and $\sum_{i=1}^{9} \sqrt{i}$: the last addition when computing

$$\left(\sum_{i=1}^{8} \sqrt{i}\right) + \sqrt{9} = \left(\sum_{i=1}^{8} \sqrt{i}\right) + 3 = \ldots$$

will be the efficient "rational + algebraic"-addition.

3 Synthesizing Representative Polynomials

In order to define arithmetic operations over algebraic numbers, the first task is the following: Given polynomials that represent the input numbers, compute a polynomial that represents the output number. In the sequel, we will illustrate the constructions for the various arithmetic operations in ascending difficulty.

3.1 Constants

Obviously, a rational number $a = \frac{n}{d}$ can be represented by $dx - n$.

Definition 4 *poly-rat a* $\equiv$ ***case*** *quotient-of a* ***of*** $(n, d) \Rightarrow [: -n, d :]$

Lemma 5 (*poly-rat a*) *represents* (*of-rat a*)

Isabelle's implementation of the rational numbers ensures that n and d are coprime and $d \geq 1$. Therefore the polynomial is already positive and irreducible.

Lemma 6 *irreducible* (*poly-rat a*) ***and*** *lead-coeff* (*poly-rat a*) > 0

3.2 Negation and Inverse

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 $-a$, the polynomial *poly-uminus*, defined as $f(-x)$, i.e., $\sum_{i=0}^{m}(-1)^i f_i x^i$, does the job.

Lemma 7 ***assumes*** *f represents a* ***shows*** (*poly-uminus f*) *represents* $(-a)$

For the inverse $\frac{1}{a}$, it is also not difficult to show that the *reciprocal polynomial* $\sum_{i=0}^{m} f_i x^{m-i}$, which is defined in Isabelle 2018 as *reflect-poly*, has $\frac{1}{a}$ as a root.

Lemma 8 ***assumes*** *f represents a* ***and*** $a \neq 0$
shows (*reflect-poly f*) *represents* (*inverse a*)

It is beneficial to also show that *poly-uminus* and *reflect-poly* preserve irreducibility, since otherwise we would have to perform polynomial factorization to maintain the invariant of always working on irreducible polynomials. We argue as follows: Suppose that f is irreducible and represents a. Clearly *poly-uminus* preserves the degree and content; thus if *poly-uminus f* is reducible, then there is a polynomial h of smaller degree that represents $-a$. Since *poly-uminus h* represents $-(-a) = a$, we obtain a polynomial representing a whose degree is smaller than f. This contradicts the uniqueness of f.

The same argument works also for *reflect-poly*, and we formalize the following lemma that generalizes the two.

Lemma 9 ***assumes*** *irreducible f* ***and*** *f represents a*
and *degree g* $\leq$ *degree f*
and *content-free g* ***and*** *g represents b*
and $\forall$ *h.* (*h represents b* $\longrightarrow$ (*I h*) *represents a* $\wedge$ *degree* (*I h*) $\leq$ *degree h*)
shows *irreducible g*

By instantiating b in the lemma by $-a$, g by *poly-uminus f*, and I by *poly-uminus*, we obtain the desired result for *poly-uminus*. Similarly we easily obtain the result for *reflect-poly*.

Lemma 10 ***assumes*** *irreducible f* ***and*** *degree f* $\neq 0$
shows *irreducible* (*poly-uminus f*)

Lemma 11 ***assumes*** *irreducible f* ***and*** *f represents a* ***and*** $a \neq 0$
shows *irreducible* (*reflect-poly f*)

3.3 Multiplication and Addition with Rational Numbers

If we had chosen rational polynomials to represent algebraic numbers, it would be easy to add or multiply a rational number to an algebraic number: when f represents a, the rational polynomials $f(x - \frac{n}{d})$ and $f(\frac{d}{n} \cdot x)$ represent $a + \frac{n}{d}$ and $a \cdot \frac{n}{d}$, respectively. In our current formalization, however, we work with integer polynomials for efficiency reasons. As neither $f(x - \frac{n}{d})$ nor $f(\frac{d}{n} \cdot x)$ is in general an integer polynomial, we define the polynomials slightly differently.

To represent $a \cdot \frac{n}{d}$, we use the following constant multiple of $f(\frac{d}{n} \cdot x)$:

$$n^m \cdot f\left(\tfrac{d}{n} \cdot x\right) = \sum_{i=0}^{m} f_i \cdot d^i \cdot n^{m-i} \cdot x^i$$

where $f(x) = \sum_{i=0}^{m} f_i \cdot x^i$. Similarly, for $a + \frac{n}{d}$ we first compute $d^m \cdot f(\frac{1}{d} \cdot x)$, and compose this polynomial and $dx - n$ to obtain $d^m \cdot f(x - \frac{d}{n})$. In the following definition, $f \circ_p g$ denotes the polynomial composition $f(g(x))$, and the monomial cx^n is denoted by *monom c n*.

Definition 5

poly-mult-rat-main n d f $\equiv \sum i \leq$ *degree f. monom* (*coeff f i* $\cdot$ $d^i \cdot n^{degree\, f - i}$) *i*
poly-mult-rat b f $\equiv$
  case *quotient-of b* ***of*** $(n, d) \Rightarrow$ *poly-mult-rat-main n d f*
poly-add-rat b f $\equiv$
  case *quotient-of b* ***of*** $(n, d) \Rightarrow$ *poly-mult-rat-main d* 1 *f* $\circ_p$ [: $-n, d$:]

We prove the desired correctness results in a straightforward way.

Lemma 12 ***assumes*** *f represents a*
  shows (*poly-add-rat b f*) *represents* (*of-rat b* + *a*)

Lemma 13 ***assumes*** *f represents a* ***and*** $b \neq 0$
  shows (*poly-mult-rat b f*) *represents* (*of-rat b* $\cdot$ *a*)

The condition $b \neq 0$ in Lemma 13 stems from the fact that we are essentially performing division. In practice this just demands a special case for $b = 0$, which trivially results in the rational number 0.

Unfortunately both *poly-add-rat* and *poly-mult-rat* do not preserve irreducibility in terms of Definition 3 in general, since they do not preserve content; e.g., for $f(x) = 2x - 3$, the unique representation of $\frac{3}{2}$, *poly-mult-rat f 2* results in the polynomial $2x - 6$, which represents 3 but is not content-free. Nevertheless, we only need to eliminate content to obtain irreducibility. We define a function *cf-pos-poly f* which divides all coefficients by the content, and additionally ensures a positive leading coefficient. Note that f represents a if and only if *cf-pos-poly f* represents a. Since each of the above functions preserves degree, and an inverse operation can be found, we apply Lemma 9 and derive the desired irreducibility results.

Lemma 14 ***assumes*** *irreducible f* ***and*** *degree f* $\neq 0$
  shows *irreducible* (*cf-pos-poly* (*poly-add-rat b f*))

Lemma 15 ***assumes*** *irreducible f* ***and*** *degree f* $\neq 0$ ***and*** $b \neq 0$
  shows *irreducible* (*cf-pos-poly* (*poly-mult-rat b f*))

3.4 *n*-th Root

For n-th root of a represented by $f(x) = \sum_{i=0}^{m} f_i x^i$, it is easy to see that $f(x^n)$, i.e., $\sum_{i=0}^{m} f_i x^{ni}$, represents $\sqrt[n]{a}$.

Definition 6 *poly-nth-root n f* $\equiv f \circ_p$ *monom* 1 *n*

Lemma 16 ***assumes*** *f represents a* ***and*** $b^n = a$ ***and*** $n \neq 0$
  shows (*poly-nth-root n f*) *represents b*

We stated the result for n-th roots without using Isabelle's operations *root* and *csqrt*, because they are defined only on types *real* and *complex*, respectively, but not on a generic field. We easily derive the results for the specific types.

Lemma 17 ***assumes*** *f represents* (*a* :: *complex*)
shows (*poly-nth-root* 2 *f*) *represents* (*csqrt a*)

Lemma 18 ***assumes*** *f represents* (*a* :: *real*) ***and*** $n \neq 0$ ***and*** $a > 0$
shows (*poly-nth-root n f*) *represents* (*root n a*)

In contrast to previous sections, *poly-nth-root* does not preserve irreducibility, even though it preserves contents. Consider, e.g., *poly-nth-root* 4 applied to $x - 64$, the unique representative of 64. The resulting polynomial is $x^4 - 64$, which can be factored into $(x^2 - 8) \cdot (x^2 + 8)$. Also for the polynomials obtained from addition and multiplication of two algebraic numbers, we cannot ensure irreducibility in general. We address this issue in Sect. 5.3.

3.5 Addition and Multiplication of Algebraic Numbers

To add or multiply two irrational algebraic numbers a and b, respectively represented as roots of polynomials f and g, 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.

For this purpose the resultant is a well-known solution. The resultant of the polynomials $f(x) = \sum_{i=0}^{m} f_i x^i$ and $g(x) = \sum_{i=0}^{n} g_i x^i$ is defined as $\mathrm{Res}(f, g) = \det(S_{f,g})$, where $S_{f,g}$ is the *Sylvester matrix*:

$$S_{f,g} = \begin{bmatrix} f_m & f_{m-1} & \cdots & f_0 & & \\ & \ddots & \ddots & & \ddots & \\ & & f_m & f_{m-1} & \cdots & f_0 \\ g_n & g_{n-1} & \cdots & g_0 & & \\ & \ddots & \ddots & & \ddots & \\ & & g_n & g_{n-1} & \cdots & g_0 \end{bmatrix} \begin{matrix} \left.\vphantom{\begin{matrix}a\\a\\a\end{matrix}}\right\} n \text{ rows} \\ \left.\vphantom{\begin{matrix}a\\a\\a\end{matrix}}\right\} m \text{ rows} \end{matrix}$$

Note that if f and g are univariate, then $\mathrm{Res}(f, g)$ is a constant. We write $\mathrm{Res}_x(f(x), g(x))$ for $\mathrm{Res}(f, g)$ when the resolved variable x should be clarified. If f and g are bivariate then $\mathrm{Res}_y(f(x, y), g(x, y))$ is a univariate polynomial over x.

We first state the desired result for addition. Here, *poly-add f g* is defined as the univariate polynomial $\mathrm{Res}_y(f(x - y), g(y))$.

Lemma 19 ***assumes*** *f represents a* ***and*** *g represents b*
shows (*poly-add f g*) *represents* $(a + b)$

We perform multiplication through division by the inverse, and division as follows: *poly-div f g* is defined as $\mathrm{Res}_y(f(x \cdot y), g(y))$ and *poly g* $0 \neq 0$ ensures that g does not represent 0, so that in particular $b \neq 0$.

Lemma 20 ***assumes*** *f represents a* ***and*** *g represents b* ***and*** *poly g* $0 \neq 0$
shows (*poly-div f g*) *represents* $(a \,/\, b)$

To prove each lemma, we need to prove two claims: the resultant has a desired root, and is a non-zero polynomial. In the next sections we prove each of the claims.

3.5.1 Resultant has Desired Roots

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

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

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:

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

Note here that only the last column depends on x. We can extract this column using the *Laplace expansion*, which we formalize as follows.

Lemma 21 ***assumes*** $A \in \textit{carrier}_m\ n\ n$ ***and*** $j < n$
shows $\textit{det}\ A = (\sum i < n.\ A_{ij} \cdot \textit{cofactor}\ A\ i\ j)$

Here $A \in \textit{carrier}_m\ n\ n$ just means that A is an $n \times n$ matrix over the considered ring, and $\textit{cofactor}\ A\ i\ j$ is defined as $(-1)^{i+j} \cdot \textit{det}\ B$, 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 $\textit{cofactor}\ A\ i\ j$ is independent from x. We obtain p and q in (1), as $\mathrm{Res}(f,g)$ is represented as follows:

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

Lemma 22 ***fixes*** $f\ g :: \alpha :: \textit{comm-ring-1 poly}$
assumes $\textit{degree}\ f > 0$ ***and*** $\textit{degree}\ g > 0$
shows $\exists p\ q.\ \textit{degree}\ p < \textit{degree}\ g \wedge \textit{degree}\ q < \textit{degree}\ f \wedge$
$[: \textit{resultant}\ f\ g :] = p \cdot f + q \cdot g$

The lemma implies that, if f and g are polynomials of positive degree with a common root, say $f(a) = g(a) = 0$, then

$$\mathrm{Res}(f,g) = p(a) \cdot f(a) + q(a) \cdot g(a) = 0$$

The result is lifted to the bivariate case: $f(a,b) = g(a,b) = 0$ implies that $\mathrm{Res}_y(f(a,y), g(a,y)) = 0$ for all a and b.

Lemma 23 ***fixes*** $f\ g :: \alpha :: \textit{comm-ring-1 poly poly}$
assumes $\textit{degree}\ f > 0 \vee \textit{degree}\ g > 0$
and $\textit{poly2}\ f\ a\ b = 0$ ***and*** $\textit{poly2}\ g\ a\ b = 0$
shows $\textit{poly}\ (\textit{resultant}\ f\ g)\ a = 0$

Here, *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, Lemma 23 indicates that the univariate polynomial $\mathrm{Res}_y(f(x-y), g(y))$ has $x = a + b$ as a root.

Lemma 24 ***fixes*** $f\ g :: \alpha ::$ *comm-ring-1 poly*
assumes $g \neq 0$ ***and*** *poly* $f\ a = 0$ ***and*** *poly* $g\ b = 0$
shows *poly* (*poly-add* $f\ g$) $(a + b) = 0$

We need a variation of Lemma 24 in which f and g are of type *int poly* while a and b are still of type α. We prove some homomorphism lemmas to obtain the following:

Lemma 25 ***assumes*** $g \neq 0$ ***and*** *ipoly* $f\ a = 0$ ***and*** *ipoly* $g\ b = 0$
shows *ipoly* (*poly-add* $f\ g$) $(a + b) = 0$

Analogously, if $b \neq 0$, then $f(x \cdot y)$ and $g(y)$ have a common root at $x = a/b$ and $y = b$.

Lemma 26 ***assumes*** $g \neq 0$ ***and*** *ipoly* $f\ a = 0$ ***and*** *ipoly* $g\ b = 0$ ***and*** $b \neq 0$
shows *ipoly* (*poly-div* $f\ g$) $(a\ /\ b) = 0$

3.5.2 Resultant is Non-Zero

Now consider the second claim: *poly-add* $f\ g$ and *poly-div* $f\ g$ are non-zero polynomials. Note that they would otherwise have any number as a root. Somewhat surprisingly, formalizing this claim is more involved than the first one.

We first strengthen Lemma 22, so that p and q are non-zero polynomials. Here, we require an integral domain *idom*, i.e., there exist no zero divisors.

Lemma 27 ***fixes*** $f\ g :: \alpha ::$ *idom poly*
assumes *degree* $f > 0$ ***and*** *degree* $g > 0$
shows $\exists\ p\ q.$ *degree* $p <$ *degree* $g \wedge$ *degree* $q <$ *degree* $f \wedge$
$[:$ *resultant* $f\ g :] = p \cdot f + q \cdot g \wedge p \neq 0 \wedge q \neq 0$

The proof is easy for the case where $\mathrm{Res}(f, g)$ is non-zero: we obtain p and q using Lemma 22, and it is easy to see that $p \cdot f + q \cdot g$ cannot be a constant if $p = 0$ or $q = 0$, using the constraints on degrees. For the case $\mathrm{Res}(f, g) = 0$, we formalize the classical result that linear equation $A\mathbf{v} = \mathbf{0}$ on an integral domain has a non-zero solution if and only if $\det(A) = 0$. Since resultants are the determinants of Sylvester matrices, from a non-zero solution to $S_{f,g}\mathbf{v} = \mathbf{0}$ one can extract non-zero polynomials p and q as a solution to $p \cdot f + q \cdot g = 0$.

If $\mathrm{Res}(f, g) = 0$, then from Lemma 27 we have $p \cdot f = -q \cdot g$. In UFDs, this implies that f and g cannot be *coprime*, i.e., that f and g have a common factor, since otherwise f must divide $-q$, contradicting $degree(f) > degree(q)$.

The definition of the predicate *coprime* in Isabelle 2018 relies on the definition of *gcd*. We generalize *coprime* as follows in order to state the above for arbitrary UFDs:

Definition 7 *coprime* $f\ g \equiv \forall h.\ h$ *dvd* $f \longrightarrow h$ *dvd* $g \longrightarrow h$ *dvd 1*

Lemma 28 ***fixes*** $f\ g :: \alpha ::$ *ufd poly*
assumes *degree* $f > 0 \vee$ *degree* $g > 0$ ***and*** *resultant* $f\ g = 0$
shows $\neg$ *coprime* $f\ g$

Finally, we reason that $\mathrm{Res}_y(f(x - y), g(y))$ and $\mathrm{Res}_y(f(x \cdot y), g(y))$ are non-zero polynomials by contradiction. As f and g are integer polynomials—a UFD— there exist irreducible factorizations: $f = f_1 \cdots f_m$ and $g = g_1 \cdots g_n$. The operation of transforming a

univariate polynomial $f(x)$ to the bivariate $f(x-y)$ is a ring homomorphism, and moreover preserves irreducibility. Thus,

$$f(x-y) = f_1(x-y)\cdots f_m(x-y)$$

is an irreducible factorization. The same property clearly holds for the transformation from $g(x)$ to $g(y)$, and we get an irreducible factorization of $g(y)$:

$$g(y) = g_1(y)\cdots g_n(y) \tag{3}$$

Now suppose that $\mathrm{Res}_y(f(x-y), g(y)) = 0$. Then Lemma 28 implies that $f(x-y)$ and $g(y)$ have a common proper factor. Without loss of generality consider an irreducible one $h(x, y)$. In UFDs, this implies that

$$f_i(x-y) \mathrel{ddvd} h(x,y) \mathrel{ddvd} g_j(y)$$

for some $i \leq m$ and $j \leq n$. By fixing y, e.g., to 0, we conclude $f_i(x)$ divides a constant $g_j(0)$, and hence f_i is a constant. This contradicts the assumption that f_i is a proper factor of f.

The reasoning is similar for division, but note that the transformation from $f(x)$ to $f(x\cdot y)$ does not preserve irreducibility: monomial x is irreducible but $x\cdot y$ is not. Nevertheless it is a ring homomorphism and we have a (possibly reducible) factorization:

$$f(x\cdot y) = f_1(x\cdot y)\cdots f_m(x\cdot y)$$

Now if $\mathrm{Res}_y(f(x\cdot y), g(y)) = 0$, then we obtain an irreducible factor $h(x, y)$ by the same reasoning as above. We still have the irreducible factorization (3) for $g(y)$, and thus $g_j(y) \mathrel{ddvd} h(x, y)$ for some j.

As $h(x, y)$ is irreducible and divides $f(x\cdot y)$, it divides some $f_i(x\cdot y)$. Here we need the fact that irreducibility and primality coincide in UFDs. Hence,

$$g_j(y) \mathrel{ddvd} h(x,y) \mathrel{dvd} f_i(x\cdot y)$$

Now we fix x to 0. Then we conclude that $g_j(y)$ divides a constant $f_i(0)$, and hence g_j is a constant, leading to a contradiction (the case $f_i(0) = 0$ will ultimately be handled by the assumption that g does not represent 0). This proves that *poly-add f g* and *poly-div f g* are non-zero polynomials, completing our proof of Lemma 19 and Lemma 20.

4 Computing the Resultant

Resultants can be computed by first building the Sylvester matrix and then computing its determinant by transformation into row echelon form. A more efficient way to compute resultants has been developed by Brown and Traub: the subresultant polynomial remainder sequence (PRS) algorithm [2,3].

The algorithm computes $\mathrm{Res}(f, g)$ 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 first consider all computations over the fraction field α *fract*, where all division operations are inherently exact. We then prove that intermediate values stay of form $\frac{a}{1}$; that is, we can use a partial division operator *div* on the integral domain α, that satisfies $(a\cdot b)$ *div* $b = a$ for $b \neq 0$, but not necessarily $(a \mathrel{div} b)\cdot b = a$. Therefore, our final implementation works solely on the integral domain, without requiring fraction field operations.

The *j-th subresultant* of the polynomials f and g with $f(x) = \sum_{i=0}^{m} f_i x^i$ and $g(x) = \sum_{i=0}^{n} g_i x^i$ is defined as follows:

$$\mathrm{Sub}_j(f(x), g(x)) = \det \begin{bmatrix} f_m & \cdots & f_{j+1} & \cdots & f_0 & & f(x) \cdot x^{n-j-1} \\ & \ddots & & \ddots & & \ddots & \vdots \\ & & f_m & \cdots & f_{j+1} & \cdots & f_0 \quad f(x) \cdot x^{j+1} \\ & & & \ddots & & \ddots & \vdots \quad \vdots \\ & & & & f_m & \cdots & f_{j+1} \quad f(x) \\ g_n & \cdots & g_{j+1} & \cdots & g_0 & & g(x) \cdot x^{m-j-1} \\ & \ddots & & \ddots & & \ddots & \vdots \\ & & g_n & \cdots & g_{j+1} & \cdots & g_0 \quad g(x) \cdot x^{j+1} \\ & & & \ddots & & \ddots & \vdots \quad \vdots \\ & & & & g_n & \cdots & g_{j+1} \quad g(x) \end{bmatrix} \tag{4}$$

Note that, in contrast to resultants, the subresultant of polynomials is a polynomial of the same type. However, due to equation (2), $\mathrm{Sub}_0(f(x), g(x))$ is actually the constant $\mathrm{Res}(f, g)$.

Lemma 29 *subresultant* $0\ f\ g =$ [: *resultant* $f\ g$:]

Using the following lemma, we can always assume $degree(f) \leq degree(g)$. In the remainder of this section, we write m for $degree(f)$ and n for $degree(g)$.

Lemma 30 *subresultant* $j\ f\ g = (-1)^{(m-j)\cdot(n-j)} \cdot$ *subresultant* $j\ g\ f$

Following Brown and Traub [3], we then formalize the following lemma, showing that a Euclidean algorithm can be used to compute subresultants. As in the Euclidean algorithm, we require a polynomial h such that $h = f + b \cdot g$ for some b, where $l = degree(h) < n$.

Lemma 31 (Subresultants via Euclidean algorithm)

1. $j < l \Longrightarrow$ *subresultant* $j\ f\ g = (-1)^{(m-j)\cdot(n-j)} \cdot (coeff\ g\ n)^{m-l} \cdot$ *subresultant* $j\ g\ h$
2. *subresultant* $l\ f\ g = (-1)^{(m-l)\cdot(n-l)} \cdot (coeff\ g\ n)^{m-l} \cdot (coeff\ h\ l)^{n-l-1} \cdot h$
3. $l < j \Longrightarrow j < n - 1 \Longrightarrow$ *subresultant* $j\ f\ g = 0$
4. *subresultant* $(n-1)\ f\ g = (-1)^{m-n+1} \cdot (coeff\ g\ n)^{m-n+1} \cdot h$

For non-field polynomials, and in particular bivariate polynomials, polynomial division is not always possible, but *pseudo-division* is: we can find h such that $h = d \cdot f + b \cdot g$ for some constant d. The following lemma allows us to use pseudo-division instead of division in subresultant computation.

Lemma 32 $d \neq 0 \Longrightarrow$ *subresultant* $j\ (d \cdot f)\ g = d^{n-j} \cdot$ *subresultant* $j\ f\ g$

Iterated application of pseudo-division results in repeated multiplication with constants d^{n-j}, and hence the coefficients of the processed polynomials increase exponentially. One approach to keep the coefficients small is to divide the polynomials by their content in every iteration, as in Collin's *primitive PRS algorithm* [3, Sect. 4]. We have implemented this approach for the preliminary version [24] of this paper.

This work additionally formalizes the more sophisticated *subresultant PRS algorithm* of Brown and Traub [2,3]. Here, a constant c—the leading coefficient of a subresultant of

the input polynomials—is carried around as an extra argument. It is used to perform exact divisions on the pseudo-remainder polynomials without the necessity to calculate the content in every iteration.

The core of this algorithm is formalized as follows, where *sdiv-poly* is an Isabelle function that divides a polynomial by a constant.

Definition 8 *subresultant-prs-main f g c* = (
let $m = degree\ f$; $n = degree\ g$; $lf = lead\text{-}coeff\ f$; $lg = lead\text{-}coeff\ g$;
$\delta = m - n$; $d = lg^{\delta}\ div\ c^{\delta - 1}$; $h = pseudo\text{-}mod\ f\ g$
in if $h = 0$ ***then*** (g, d)
else *subresultant-prs-main g* (*sdiv-poly h* $((-1)^{\delta+1} \cdot lf \cdot c^{\delta}))\ d$)

The above function works under the invariant that $n < m$ (so that $\delta - 1 \geq 0$ in Definition 8) and in particular the invariant that all divisions are exact. Thus as the initial step we establish these invariants, and obtain a suitable initial value for c.

Definition 9 *subresultant-prs f g* ≡
let $h = pseudo\text{-}mod\ f\ g$; $\delta = degree\ f - degree\ g$; $c = (lead\text{-}coeff\ g)^{\delta}$
in if $h = 0$ ***then*** (g, c)
else *subresultant-prs-main g* $((-1)^{\delta+1} \cdot h)\ c$

The invocation of *subresultant-prs f g* returns a pair (h, d), where h is a scalar multiple of the GCD of f and g, and $\mathrm{Res}(f, g) = d$ if $degree(h) = 0$, and $\mathrm{Res}(f, g) = 0$ otherwise.

In addition to the definitions of *subresultant-prs* and *subresultant-prs-main* we develop an optimized implementation in the form of code equations. These optimizations include treating common cases separately, avoiding calculating the same value twice, and replacing expressions like $(-1)^{\delta+1} \cdot h$ by a single negation. We also integrate the efficient calculation of $lg^{\delta}\ div\ c^{\delta - 1}$ described by Ducos as *dichotomous Lazard* [8, Sect. 2], but we did not integrate Ducos' second optimization about the calculation of *sdiv-poly* in Definition 8.

We define the final function as *resultant-impl*, and prove the following correctness result as a code equation:

Lemma 33 *resultant f g* = *resultant-impl f g*

We also define a function *gcd-impl* that returns the GCD of two polynomials based on *subresultant-prs*, and get a correctness result:

Lemma 34 *gcd f g* = *gcd-impl f g*

We do not state Lemma 34 as a code equation, since on simple polynomials, e.g., of type *int poly*, we experimentally see that the algorithm performs worse than the standard GCD implementation. The algorithm becomes beneficial for multivariate polynomials, e.g., *int poly poly poly*.

5 Real Algebraic Numbers

In the previous two sections, we have seen how to synthesize a polynomial f representing an algebraic number a as one of its root. To unambiguously represent a, we need to specify which root of f is actually a. Moreover, we need a concrete representation of real algebraic numbers. Both of these problems are addressed in this section, resulting in a verified implementation of real algebraic numbers.

```
(* Layer 1 *)
type-synonym real-alg-1 = int poly × rat × rat
definition real-of-1 (f, l, r) ≡ THE x. ipoly f x = 0 ∧ of-rat l ≤ x ∧ x ≤ of-rat r
definition unique-root (f, l, r) ≡ ∃! x. ipoly f x = 0 ∧ of-rat l ≤ x ∧ x ≤ of-rat r
definition invariant-1 (f, l, r) ≡
  unique-root (f, l, r) ∧ irreducible f ∧ sgn l = sgn r ∧ lead-coeff f > 0

(* Layer 2 *)
datatype real-alg-2 = Rational rat | Irrational nat real-alg-1
fun real-of-2 (Rational x) = real-of-rat x
 | real-of-2 (Irrational n y) = real-of-1 y
fun invariant-2 (Rational x) = True
 | invariant-2 (Irrational n (f, l, r)) = (
   degree f > 1 ∧
   invariant-1 (f, l, r) ∧
   n = card {x. x ≤ real-of-1 (f, l, r) ∧ ipoly f x = 0})

(* Layer 3 *)
typedef real-alg-3 = Collect invariant-2
lift-definition real-of-3 :: real-alg-3 ⇒ real is real-of-2

(* Layer 4 *)
quotient-type real-alg = real-alg-3 / λ x y. real-of-3 x = real-of-3 y
lift-definition real-of :: real-alg ⇒ real is real-of-3
```

Fig. 2 Internal type hierarchy for representing real algebraic numbers, including invariants and conversions to real numbers

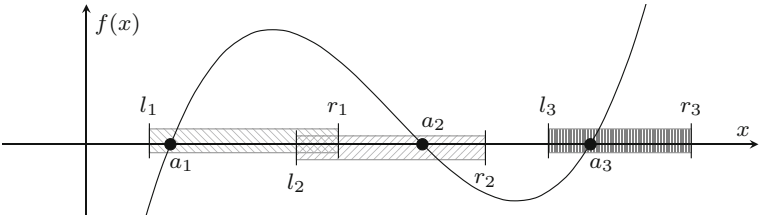

Fig. 3 Representing all roots a_1, a_2, and a_3 of some polynomial f

5.1 Datatypes for Real Algebraic Numbers

For representing real algebraic numbers, we develop a hierarchy of four layers.

On the lowest, Layer 1 (cf. Fig. 2), we define the type *real-alg-1* that represents a real algebraic number a as a triple (f, l, r), where $[l, r]$ is a rational interval in which only a is a root of f. The function *real-of-1* takes such a triple and gives back $a :: real$. This function is defined only on triples that represents exactly one algebraic number, which is ensured as a part of the invariant *invariant-1*. The invariant additionally demands that the representative polynomials are irreducible and have positive leading coefficients, and that the lower- and upper-bounds of the interval have the same sign. For instance, in Fig. 3 the three roots a_1, a_2, and a_3 of f are represented by the triples (f, l_1, r_1), (f, l_2, r_2), and (f, l_3, r_3), respectively, as each interval $[l_i, r_i]$ contains exactly one root a_i. Note also that the intervals may be overlapping.

Layer 2 introduces the datatype *real-alg-2*, which takes a special treatment for rational numbers, so that computations involving only rational numbers will not experience overheads that would arise by manipulating roots of polynomials as in Layer 1. Hence, *invariant-2*

```
(* Layer 1 *)
definition uminus-1 (f, l, r) ≡ (abs-int-poly (poly-uminus f), −r, −l)
lemma assumes invariant-1 x
  shows invariant-1 (uminus-1 x)
    and real-of-1 (uminus-1 x) = − real-of-1 x

(* Layer 2 *)
fun uminus-2 (Rational x) = Rational (−x)
 | uminus-2 (Irrational n x) = real-alg-2 (uminus-1 x)
lemma assumes invariant-2 x
  shows real-of-2 (uminus-2 x) = − real-of-2 x
    and invariant-2 (uminus-2 x)

(* Layer 3 *)
lift-definition uminus-3 is uminus-2
lemma real-of-3 (uminus-3 x) = − real-of-3 x

(* Layer 4 *)
lift-definition uminus-real-alg is uminus-3
lemma real-of (uminus-real-alg x) = − real-of x
```

Fig. 4 Definitions and properties for negation illustrated for all layers

demands that the (f, l, r)-form is used only for algebraic numbers of degree at least 2. In *real-alg-2*, we additionally store the index of the root, counted from the smallest to the largest.

Layer 3 introduces the type *real-alg-3*, which is identical to *real-alg-2* but now *invariant-2* is enforced by the type system.

Layer 4 introduces the quotient type *real-alg*, that identifies different representations of the same number. Hence, the built-in equality of Isabelle/HOL on *real-alg* corresponds to equality on the represented real numbers. We do not have this property in other layers, since they still permit a number to be represented differently; e.g., $\sqrt{2}$ is encoded by $(x^2 - 2, 1, 2)$, $(x^2 - 2, 1.4, 1.5)$, etc.

In each layer we define algebraic operations that are formalized in Sect. 3. Here we take the computation of $-a$ as a first simple example, cf. Fig. 4. Other arithmetic operations like addition require factorization and root separation which will be discussed in Sect. 5.3.

In Layer 1, *uminus-1* (f, l, r) computes the new representative polynomial *poly-uminus* f, and takes $[-r, -l]$ as the new interval. To satisfy the invariant that the leading coefficient is positive, *abs-int-poly*, which just negates the polynomial if the leading coefficient is negative, is applied. The correctness lemma states that the invariants are preserved and the desired $-a$ is represented.

In Layer 2, we perform a simple case-analysis on whether the represented number a is rational or not. If it is, then we use the rational number $-a$, and otherwise invoke *uminus-1* from Layer 1. Afterwards, *real-alg-2* :: *real-alg-1* $\Rightarrow$ *real-alg-2* is applied. This function converts the triple representation into *real-alg-2* by either extracting the rational root if f is linear, or by computing the index of the root by invoking Sturm's method.

Lifting the algorithms and the correctness lemma from Layer 2 to Layers 3 and 4 is then immediate using Isabelle's lifting and transfer package [14].

5.2 Comparison and Tightening Intervals

As we work on unique representative polynomials, we can immediately determine the equality of two algebraic numbers in Layer 2: they are equal if the representative polynomials and

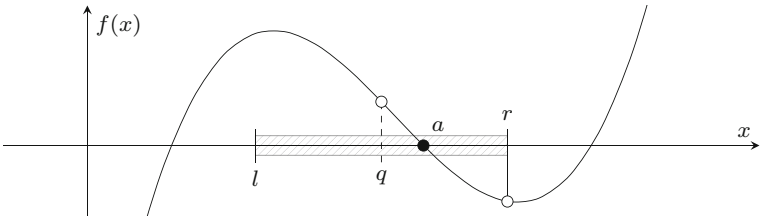

Fig. 5 Comparing an algebraic number a and a rational number q. In this case, we know $q < a$ since $f(q)$ and $f(r)$ have different signs

the root indices match. This test is implemented as follows, and satisfies the corresponding correctness statement.

fun *equal-2* (*Rational q*) (*Rational r*) = $(q = r)$
| *equal-2* (*Irrational n* (*f*, -, -)) (*Irrational m* (*g*, -, -)) = $(f = g \wedge n = m)$
| *equal-2* (*Rational* -) (*Irrational* - -) = *False*
| *equal-2* (*Irrational* - -) (*Rational* -) = *False*

Lemma 35 ***assumes*** *invariant-2 x* ***and*** *invariant-2 y*
shows *equal-2 x y* $\longleftrightarrow$ *real-of-2 x* = *real-of-2 y*

We can efficiently compare an algebraic number a and a rational number q. Suppose that a is irrational and represented by (f, l, r). Then $a = q$ is impossible as q is rational, so we want to know whether $a < q$ or $q < a$. It is trivial if $q \notin [l, r]$, and otherwise we compare the signs of $f(q)$ and $f(r)$: we know $a < q$ if the signs coincide, and $q < a$ otherwise (see Fig. 5).

Definition 10 *compare-rat-1 q* $(f, l, r) \equiv$
if q < l then Lt else if q > r then Gt else
if sgn (*ipoly f q*) = *sgn* (*ipoly f r*) *then Gt else Lt*

Using this comparison with rational numbers, we can tighten the intervals to arbitrary precision: by taking, e.g., $q = \frac{l+r}{2}$ one can halve the interval to $[l, q]$ or $[q, r]$, depending on whether $a < q$ or $q < a$.

Being able to tighten intervals, we can implement the *floor* $\lfloor a \rfloor$ and *ceiling* $\lceil a \rceil$ operations: tighten the interval of a until it contains at most one integer point, and then use the sign-based comparison to determine whether a is less or greater than the integer.

We can also compare two irrational algebraic numbers a and b by tightening intervals. The implementation of the comparison functions[4] for the first two layers is shown in Fig. 6.

In Layer 1, if a and b have disjoint intervals, then comparison is trivial. Otherwise *compare-1* tightens the intervals of a and b until they become disjoint. The procedure is terminating only if $a \neq b$, since intervals will never become disjoint if $a = b$. Hence Isabelle's **partial-function** command [16], that allows defining potentially nonterminating procedures, becomes essential. In order to conveniently prove correctness, 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 separation distance, i.e., the minimal distance of two distinct roots of some polynomial.

[4] The formalization differs slightly, since the value of *sgn (ipoly p r)* is carried around for efficiency.

```
(* Layer 1 *)
definition tighten-poly-bounds f l r ≡ let m = (l + r) / 2
  in if sgn (ipoly p m) = sgn (ipoly p r) then (l, m) else (m, r)

partial-function compare-1 f g l1 r1 l2 r2 =
  if r1 < l2 then Lt else if r2 < l1 then Gt
  else let
    (l1', r1') = tighten-poly-bounds f l1 r1;
    (l2', r2') = tighten-poly-bounds g l2 r2
    in compare-1 f g l1' r1' l2' r2'

(* Layer 2 *)
fun compare-2 (Rational q) (Rational r) = compare q r
 | compare-2 (Rational q) (Irrational a) = compare-rat-1 q a
 | compare-2 (Irrational a) (Rational q) = invert-order (compare-rat-1 q a)
 | compare-2 (Irrational n (f, l1, r1)) (Irrational m (g, l2, r2)) =
    if (f = g ∧ n = m) then Eq else compare-1 f g l1 r1 l2 r2)

lemma assumes invariant-2 x and invariant-2 y
  shows compare-2 x y = compare (real-of-2 x) (real-of-2 y)
```

Fig. 6 Comparison for the first two layers

In Layer 2, *compare-2* invokes a suitable comparison function depending on rationality: Isabelle's standard *compare* for two rational numbers, *compare-rat-1* if exactly one of the inputs is rational, and *compare-1* for two irrational numbers. Before invoking *compare-1*, *compare-2* first tests equality in order to ensure termination.

5.3 Polynomial Factorization and Root Separation

Recall the invariant of Layer 1: the representing polynomial must be irreducible and have exactly one root in the provided interval. Hence, after synthesizing a polynomial f to represent an algebraic number a, we must further ensure irreducibility of f and provide an interval in which a is the only root of f.

For unary minus and multiplicative inverse, Lemmas 10 and 11 ensure irreducibility, and moreover the obvious intervals $[-r, -l]$ and $[r^{-1}, l^{-1}]$ work, where $[l, r]$ is the interval for the input. For other arithmetic operations from Sect. 3, the synthesized polynomial is not generally irreducible, and obviously derived intervals may contain multiple roots.

We first establish irreducibility by a formalized polynomial factorization algorithm [7], and obtain irreducible polynomials $f_1, \cdots, f_n$, such that exactly one of them represents the desired a. So the remaining task is to determine which f_i has a as a root, and to provide an interval in which a is the only root of f_i.

We achieve the two goals in one go. Our algorithm maintains: the current interval $[l, r]$, which contains the desired a; a list F of candidate polynomials which have at least one root in the interval; and the total number n of roots the candidates have in the interval.

The procedure returns if $n = 1$; in this case, F contains exactly one polynomial, and this polynomial represents a. Otherwise, it tightens $[l, r]$, and then updates n and simultaneously excludes those factors from F that have no root in $[l, r]$. We will explain later in this section how to count the number of real roots a polynomial f has in a given interval.

Note that this procedure terminates only if exactly one candidate polynomial has a as a root. Consequently, we again use **partial-function** to define the procedure in Isabelle.

How $[l, r]$ is tightened depends on the actual operation we are computing. Hence we model the algorithm abstractly using the two functional parameters:

bnd-get :: $\beta \Rightarrow rat \times rat$
bnd-update :: $\beta \Rightarrow \beta$

The type variable β represents *states*, which contain sufficient information to maintain intervals. The interval is retrieved by the function *bnd-get*, and *bnd-update* updates one state to another in which a tighter interval is obtained.

For instance, if a is the addition of b and c, represented by (g, l_b, r_b) and (h, l_c, r_c), then the state is a quadruple (l_b, r_b, l_c, r_c), *bnd-get* returns $[l_b + l_c, r_b + r_c]$, and *bnd-update* tightens intervals of b and c using the bisection algorithm *tighten-poly-bounds* of Sect. 5.2. Multiplication $a = b \cdot c$ is treated in the same way, except that *bnd-get* returns the interval $[l_b \cdot l_c, r_b \cdot r_c]$; here, the main bisection algorithm for multiplication is only invoked on positive numbers, and a separate algorithm takes care of the signs.

Finally, for computing the n-th root $a = \sqrt[n]{b}$ of a positive number b, the state is an interval $[l_a, r_a]$ containing a, where initially $l_a = \lfloor \sqrt[n]{l_b} \rfloor$ and $r_a = \lceil \sqrt[n]{r_b} \rceil$. In every iteration of *bnd-update*, we first compute the rational number $m = \frac{l_a + r_a}{2}$. We then compare m with a by comparing m^n with b. This detour is necessary, since the latter comparison can be computed using *compare-rat-1* from Sect. 5.2, whereas the former comparison is problematic since a is not available. Finally, we update the interval $[l_a, r_a]$ to one of the tighter intervals $[l_a, m]$, $[m, m]$, or $[m, r_a]$, depending on whether $a < m$, $a = m$, or $m < a$.

We present the correctness statement of the generic factor selection procedure only on Layer 2 where the functions *bnd-get* and *bnd-update* are implicit arguments to *select-correct-factor-int-poly*, and where *bnd-update*^{i} *init* is iterated function application of *bnd-update* on input *init*.

Lemma 36 ***assumes*** *converges-to* (λ *i. bnd-get* (*bnd-update*^{i} *init*)) *x*
and *select-correct-factor-int-poly init f* = *r* ***and*** *ipoly f x* = 0 ***and*** $f \neq 0$
shows *invariant-2 r* ***and*** *real-of-2 r* = *x*

The actual correctness proof in Layer 1 is a rather involved inductive proof; the well-foundedness of the induction relation depends on the convergence of the bounds towards a, and the statement uses 12 invariants that are maintained throughout the proof.

It remains to count how many roots a polynomial f has in a given interval. We implement such a root-counting function rc_f using Sturm's method, with a special treatment for linear polynomials. We extend the existing formalization by Eberl [9], 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* [13]; at this point we must not yet use real number arithmetic. 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 37 *sturm* (*real-of-rat-poly f*) = *map real-of-rat-poly* (*sturm-rat f*)

For efficiency, we adapt the algorithm 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 in the candidate list 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:

Lemma 38 *count-roots-interval-rat f* = (
let *fs* = *sturm-rat* (*map-poly rat-of-int f*) `(* precompute *)`
in ...(λ *l r. sign-changes-rat fs l* − *sign-changes-rat fs r* + ...) ...)

5.4 Implementing Real and Complex Numbers via Real Algebraic Numbers

Having the arithmetic operations on real algebraic numbers, we now provide 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 39 (Code lemmas)

(*real-of x*) + (*real-of y*) = *real-of* (*x* + *y*)
`(* similar code lemmas for` $=, <, -, \cdot, /,$ `floor, etc. *)`

Note that in Lemma 39, the left-hand side of the equality is addition for type *real*, whereas the right is addition for type *real-alg*.

As a consequence, Isabelle users now can specify algorithms using algebraic operations on type *real* and will obtain executable code which uses our verified real algebraic number implementation. Similarly, one can prove a lemma over real numbers like $(\mathit{sqrt}\ 2 + \mathit{root}\ 3\ 7)^2 \notin \mathbb{Q}$ by evaluation.

To implement complex algebraic numbers, we require nearly nothing: Isabelle implements complex numbers as pairs of real numbers representing the real and imaginary part, and this is possible also in the algebraic setting. Note that a complex number is algebraic if and only if both the real part and the imaginary part are algebraic. Thus, all of the following operations become executable on the complex numbers for free: $+$, $-$, $\cdot$, $/$, $\sqrt{\cdot}$, $=$, and complex conjugate. These operations are already implemented via operations on the real numbers, and computed by real algebraic numbers via data refinement. For instance, complex square roots are computed as

$$\sqrt{x + y\mathrm{i}} = \sqrt{\frac{\sqrt{x^2 + y^2} + x}{2}} + \sqrt{\frac{\sqrt{x^2 + y^2} - x}{2}}\,\mathrm{i} \cdot \mathit{sgn}\ y$$

Another approach to implement complex algebraic numbers would be to use *one* representing polynomial in combination with a rectangle in the complex plane to uniquely identify roots, instead of using *two* real algebraic numbers, each requiring its own representing polynomial and interval. As most of our results have been formalized in a generic way, this would become possible if one had replaced the bisection algorithms by similar methods to separate complex roots, e.g., by formalizing results by Kronecker [23, Sect. 1.4.4]. Li provides such a complex root counting algorithm in the AFP [17]. However, his verified implementation currently has some limitations resulting in runtime errors, which makes it at least hard, to use it as a bisection algorithm which must always succeed.

5.5 Displaying Algebraic Numbers

We provide two approaches to display real algebraic numbers, i.e., two functions *show-real-alg* of type *real-alg* ⇒ *string*. The first one displays an approximative value of an algebraic

number a. Essentially, the rational number $\frac{\lfloor 1000a \rfloor}{1000}$ is computed and displayed as a string. For instance, $\sqrt{2}$ is displayed as "~ 1.414".

The second approach displays an algebraic number without approximation, canonically in form "root #n of f". For Layer 4 we have to prove that this approach actually defines a function, i.e., f and n are uniquely defined by the represented algebraic number. This result easily follows from the invariant that we use irreducible polynomials in combination with their uniqueness, Lemma 2.

Besides this well-definedness result, we also prove *soundness* in the following sense. The function *show-real-alg* first invokes a function *real-alg-show-info* :: *real-alg* $\Rightarrow$ *real-alg-show-info* where

datatype *real-alg-show-info* =
Rat-Info rat | *Sqrt-Info rat rat* | *Real-Alg-Info* (*int poly*) *nat*

It then converts these intermediate values into strings.

Whereas there is no soundness statement for the final *show-real-alg*, we prove the following result for *real-alg-show-info*.

Lemma 40 ***assumes*** *real-alg-show-info x* = *info* ***and*** *real-of x* = *a*
shows *info* = *Rat-Info r* $\Longrightarrow$ *a* = *of-rat r*
and *info* = *Sqrt-Info r s* $\Longrightarrow$ *a* = *of-rat r* + *sqrt* (*of-rat s*)
and *info* = *Real-Alg-Info f n* $\Longrightarrow$
f represents a $\wedge$ *n* = *card* {*y*. *y* $\leq$ *a* $\wedge$ *ipoly f y* = 0 }

Using *show-real-alg*, we define a function for displaying values of type *real* and then provide its executable implementation via a code equation.

Definition 11 *show-real a* $\equiv$
if $\exists$*x*. *a* = *real-of x* ***then*** *show-real-alg* (*THE x*. *a* = *real-of x*)
else ''*transcendental number*''

Lemma 41 *show-real* (*real-of x*) = *show-real-alg x*

Using *show-real*, it is trivial to display complex numbers. Here we only present a simplified definition. The actual definition produces nicer strings if the imaginary part or real part are zero. In the definition, Isabelle's list-append operator @ is used for string concatenation.

Definition 12 *show-complex a* $\equiv$
show-real (*Re a*) @ '' + '' @ *show-real* (*Im a*) @ ''*i*''

6 Real and Complex Roots of Rational Polynomials

In this section, we provide executable functions which identify all real or complex roots of an integer or rational polynomial, as illustrated in Example 1. Without loss of generality we only consider integer polynomials, since every rational polynomial can be converted into an integer polynomial with the same roots, namely by multiplying with the common denominator of the coefficients.

Based on the root finding algorithms, we also provide complete real and complex polynomial factorization algorithms, that work for polynomials with rational coefficients.

6.1 Real Roots of Integer Polynomials

We cannot yet represent the roots of an arbitrary polynomial f as *root #1 of* f, …, *root #n of* f as in Example 1, since we have to establish the invariant that f is irreducible, and provide an interval for each root.

Example 2 The polynomial $f = -14 + 63x + 49x^2 + -490x^3 + 469x^4 + 21x^5 + -126x^6$ has three real roots. The algorithm for computing all roots of f will result in the rational number $\frac{1}{3}$ and the first two roots of $g = 2 + 3x + -7x^2 + x^3 + 2x^4$, which are irrational and are the unique roots of g in the intervals $[-4, -2]$ and $[-2, 0]$.

So, essentially the construction works in three steps:

1. integer polynomial factorization: $f = -7 \cdot (-1 + 3x)^2 \cdot g$ in Example 2
2. construction of initial bounds for roots: all roots of g are within $[-8, 8]$
3. bisection until one finds intervals for all roots: the roots of g are the unique roots of g in the intervals $[-4, -2]$ and $[-2, 0]$

For the first step we use again the formalized polynomial factorization algorithm. Then we develop an algorithm that, given an irreducible polynomial f, produces a list of intervals such that each interval contains exactly one root of f, and every root of f is contained in one of them. Below we provide more details on the second and third step.

6.1.1 Root Bounds

Instead of searching the infinite real space for roots of a polynomial, we start with a closed interval. There are some known bounds on the maximal absolute value of roots of a polynomial. Among them we choose Cauchy's bound, as it is efficient and easy to formalize, and gives sufficient precision for our purpose.

The *Cauchy bound* $C(f)$ of a non-zero polynomial $f(x) = \sum_{i=0}^{m} f_i x^i$ is defined as follows:

$$C(f) = 1 + \frac{\max\{|f_0|, \ldots, |f_{m-1}|\}}{|f_m|}$$

Then for any root a of f, we have $|a| \leq C(f)$. In the implementation, we do not start with the exact computed bound, which most often is a fraction. Instead, we start with the smallest power of 2 such that $2^n \geq C(f)$. The advantage is that then the bisection algorithm avoids fractions as long as possible. In Example 2, $C(g) = \frac{9}{2}$ and hence, we choose 8 as the initial bound.

Definition 13 *root-bound f* ≡ ***let***
m = degree f;
n = 1 + div-ceiling (*max-list* (*map* (λ *i*. |*coeff f i*|) [0..<*m*])) |*lead-coeff f*|
in *of-int* ($2^{\textit{log-ceiling 2 n}}$)

Lemma 42 ***assumes*** *root-bound f = B* ***and*** *degree f* ≠ 0
shows *ipoly f x* = 0 ⟹ *norm x* ≤ *of-rat B* ***and*** *B* ≥ 0

6.1.2 Root Separation

Now we separate the roots using a bisection algorithm. The main idea is similar to the interval tightening algorithm in Sect. 5.3. The difference is that here we keep track of all the roots. Hence the algorithm stores two lists: a work list of intervals from which roots of input f have to be found, and a result list which stores the already found intervals each containing exactly one root of f. Initially the work list is a singleton containing the interval $[-B, B]$ where B is the initial bound explained above, and the result list is empty.

In every iteration the algorithm picks up an interval $[l, r]$ from the work list, and calls the root-counting function $rc_f\ l\ r$ to determine the number n of real roots of f within this interval. If $n = 0$ then the algorithm throws away this interval and carries on to the next of the work list. If $n = 1$ then a root is identified; the representation (f, l, r) is added to the result list. Finally, if $n > 1$ then the algorithm splits the interval into $[l, \frac{l+r}{2}]$ and $[\frac{l+r}{2}, r]$ and pushes them back to the work list. The overlap of the intervals is not problematic, since the bisection algorithm is only invoked on irreducible polynomials of degree at least 2, which cannot have a rational root like $\frac{l+r}{2}$. In particular, the algorithm will return a distinct list of roots.

The bisection algorithm is defined via **partial-function** for efficiency reasons. The root counting function rc_f, that is obtained after the first phase of Sturm's method, is passed as a parameter to the main procedure, in order to avoid recomputation. If the algorithm is invoked with an unexpected function, e.g., one that always yields $n = 2$, then it is nonterminating.

We prove that, if correct arguments are passed, then the result of the bisection algorithm is as intended. To this end, we perform well-founded induction on the work list. Here we define δ as in the bisection algorithm of Sect. 5.2, but now combine the size-measure of intervals with the multiset-extension of a well-founded order [15]. This is required, since if $n > 1$ we replace one interval by two smaller ones.

The final correctness is stated as follows.

Lemma 43 ***assumes*** $f \neq 0$
shows *set* (*real-roots-of-int-poly f*) = {*a. ipoly f a* = 0}
and *distinct* (*real-roots-of-int-poly f*)

6.2 Complex Roots of Integer Polynomial

In contrast to Sect. 5.4, where complex algebraic numbers are easily implemented via real algebraic numbers, it is not so trivial to develop a complex-number counterpart of *real-roots-of-int-poly*, i.e., a method to identify all complex roots of an integer polynomial f.

To identify complex roots, we formalize the following algorithm. It is based on a constructive proof of the fact that a complex number is algebraic if and only if its real and imaginary part are algebraic [22, Corollary 7.3].

– Consider a complex root $a + b\mathrm{i}$ of f for $a, b \in \mathbb{R}$. We have

$$2a = (a + b\mathrm{i}) + (a - b\mathrm{i})$$

and since both $a + b\mathrm{i}$ and $a - b\mathrm{i}$ are roots of f, $2a$ is a root of *poly-add f f*. Hence the following polynomial g has a as a root:

$$g = \textit{poly-mult-rat}\ \tfrac{1}{2}\ (\textit{poly-add}\ f\ f)$$

Similarly, $2\mathrm{i} \cdot b = (a + b\mathrm{i}) - (a - b\mathrm{i})$ is a root of *poly-add f* (*poly-uminus f*), and as 2i is represented by polynomial $4 + x^2$, the following polynomial h has b as a root:

$$h = \textit{poly-div}\ (\textit{poly-add}\ f\ (\textit{poly-uminus}\ f))\ [:4, 0, 1:]$$

– Let C be the set of complex numbers $a + b\mathrm{i}$ with $a \in$ *real-roots-of-int-poly g* and $b \in$ *real-roots-of-int-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-int-poly* contains several special measures to improve efficiency, e.g., factorizations are performed in between, explicit formulas are used, symmetries are exploited, etc. In particular, we explain how we efficiently compute $\{c \in C.\ f(c) = 0\}$ from C.

Since we have now executable complex algebraic operations, one can in principle evaluate $f(c)$ and test whether it is 0 or not. A drawback of this approach is the demand for manipulating polynomials of high degree. For instance, when testing $f(c) = 0$ in Example 1, complex algebraic numbers like c^4 occur. These result in factorization problems for integer polynomials of degree 144.

Instead, we formalize the following algorithm based on interval arithmetic.

– For each $c \in C$, extract the real intervals I_r and I_i from the internal representation of the real and imaginary part, respectively. Use interval arithmetic to test whether $0 \in f(I_r + I_i\mathrm{i})$. If 0 is not contained in the interval, remove c from C.
– If $|C| = \textit{degree}(f)$, return C.
– Tighten all bounds of C so that the extracted intervals will be half of the previous size and start again.

The filter algorithm is formalized on Layer 3, since it is the highest layer which still permits to access the internal interval bounds. Its termination is proven by showing some convergence properties, so that in particular all non-roots are eventually detected and removed. The correctness result of the *complex-roots-of-int-poly* looks as in the real case.

Lemma 44 ***assumes*** $f \neq 0$
shows *set* (*complex-roots-of-int-poly f*) = {*a. ipoly f a* = 0}
and *distinct* (*complex-roots-of-int-poly f*)

6.3 Factorization of Polynomials over $\mathbb{C}$ and $\mathbb{R}$

With the help of the complex roots algorithm *complex-roots-of-int-poly* and the fundamental theorem of algebra, we further develop two algorithms that factor polynomials with rational coefficients over $\mathbb{C}$ and $\mathbb{R}$, respectively. Factorization over $\mathbb{C}$ is easy: 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 factorization over $\mathbb{R}$, 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 Experiments

Now we experimentally evaluate our implementation. We compare the following three implementations of algebraic numbers:

Table 2 Total time for example computations with algebraic numbers

	Experiment	Old version	New version	Mathematica
(1)	Examples in [18, Fig. 3]	0.032*s*	0.016*s*	0.061s
(2)	$\{norm\,(x) \mid x \in \mathbb{C} \wedge 1 + 2x + 3x^4 = 0\}$	21.941*s*	0.207*s*	0.654s
(3)	$\sum_{i=1}^{10} \sqrt{i}$	0.422*s*	0.117*s*	0.070s
(4)	$\sum_{i=1}^{6} \sqrt[3]{i}$	41.779*s*	19.902*s*	0.081s
(5)	$(\sum_{i=1}^{9} \sqrt{i}) - (\sum_{i=1}^{8} \sqrt{i})$	26.459*s*	2.261*s*	0.000s

– *Old version* refers to our verified implementation as described in the preliminary version of this paper [24].
– *New version* refers to our current implementation of algebraic numbers as described in this paper.
– *Mathematica* refers to Wolfram Mathematica 11. Here, we invoke the methods `RootReduce` and `IsolatingInterval` in order to obtain the representing polynomial and an interval which uniquely identifies the root, respectively.

Differences with the old version include:

– The type of representing polynomials are now *int poly* rather than *rat poly*.
– We incorporated the verified factorization algorithm [7] while the old version uses an unverified one that does not ensure irreducibility.
– We introduced the sign-based comparison technique (Sect. 5.2) while the old version uses Sturm's method. Due to this, the old version has to keep Sturm sequences in internal representations while the new version does not.
– We introduced an algorithm that finds the correct factor and a valid interval in one go (Sect 5.3), while the old version performs these tasks sequentially: it first tightens intervals until undesired roots are excluded, and then applies factorization and selects the correct factor.
– We formalized Brown and Traub's subresultant PRS algorithm (Sect. 4), while the old version uses a variant of Collin's primitive PRS algorithm.
– We apply interval arithmetic for filtering the complex roots of a polynomial from a list of candidates. In contrast, the old version utilizes algebraic number arithmetic.

All of our implementations have been tested using extracted Haskell code which has been compiled by GHC version 8.2.1 using `ghc -O2`. The experiments with Mathematica have been conducted within the graphical user interface using Mathematica's `Timing` routine. All experiments in this paper have been executed on a 3.2 GHz 8-Core Intel Xeon W with 64 GB of RAM running macOS High Sierra.

The results of our experiments in Table 2 illustrate that our new implementation is significantly faster than the old implementation.

The big difference in experiment (2) is due to the use of interval arithmetic instead of expensive complex algebraic number computations (Sect. 6.2). For the other experiments, the improvements are mainly due to optimizations of the bisection algorithms and the resultant computation.

In Table 3 we report on detailed profiling information on experiments (4) and (5). The improvements in tightening intervals is due to the sign-based method (Sect. 5.2) and the combined algorithm which tightens intervals and selects correct factors at the same time (Sect. 5.3). In experiment (5) we also see that the subresultant PRS algorithm of Sect. 4

Table 3 Timing of individual sub-algorithms in percentage of total runtime

Experiment	Algorithm	Old version (%)	New version (%)
(4)	Tightening intervals	59.3	11.5
(4)	Resultant	0.1	0.0
(4)	Factorization	40.1	88.4
(5)	Tightening intervals	13.3	1.5
(5)	Resultant	76.1	35.9
(5)	Factorization	10.7	62.6

significantly improves the computation time of resultants. As a consequence of our optimizations, polynomial factorization is the main bottleneck of the new implementation.

Table 2 also reveals that there is further room for efficiency improvements in order to compete with the commercial product Mathematica. In particular in Experiment (5), Mathematica can first symbolically simplify the expression to $\sqrt{9}$ which is then easily computed. In contrast, our implementation computes the difference of two real algebraic numbers, each having minimal representatives of degree 16. Still, also our implementation has its benefit: it is formally verified.

8 Conclusion

We developed verified algorithms for real and complex algebraic numbers in Isabelle/HOL. These include all the algebraic operations, algorithms to identify complex roots of rational polynomials, and to uniquely present algebraic numbers as strings. The formalization is available to every Isabelle user, and the implementation is available to every programmer as verified Haskell code.

As for future work, 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. Moreover, a more efficient verified polynomial factorization algorithm would be welcome, since this algorithm is currently the most time-consuming part when computing algebraic numbers.

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 $\mathbb{C}$.

Acknowledgements Open access funding provided by Austrian Science Fund (FWF).

References

1. Avanzini, M., Sternagel, C., Thiemann, R.: Certification of complexity proofs using CeTA. In: RTA 2015. pp. 23–39. LIPIcs 36 (2015)
2. Brown, W.S.: The subresultant PRS algorithm. ACM Trans. Math. Softw. **4**(3), 237–249 (1978)

3. Brown, W.S., Traub, J.F.: On Euclid's algorithm and the theory of subresultants. J. ACM **18**(4), 505–514 (1971)
4. Cohen, C.: Construction of real algebraic numbers in Coq. In: ITP 2012. LNCS, vol. 7406, pp. 67–82 (2012)
5. Cohen, C., Djalal, B.: Formalization of a Newton series representation of polynomials. In: CPP 2016. pp. 100–109. ACM (2016)
6. Cohen, C., Mahboubi, A.: Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. Log. Methods Comput. Sci. **8**(1:02), 1–40 (2012)
7. Divasón, J., Joosten, S., Thiemann, R., Yamada, A.: A formalization of the Berlekamp-Zassenhaus factorization algorithm. In: CPP 2017, pp. 17–29 (2017)
8. Ducos, L.: Optimizations of the subresultant algorithm. J. Pure Appl. Algebra **145**, 149–163 (2000)
9. Eberl, M.: A decision procedure for univariate real polynomials in Isabelle/HOL. In: CPP 2015. pp. 75–83. ACM (2015)
10. Eberl, M.: Linear recurrences. Archive of Formal Proofs (Oct 2017), http://isa-afp.org/entries/Linear_Recurrences.html, Formal proof development
11. von zur Gathen, J., Gerhard, J.: Modern computer algebra, 2nd edn. Cambridge University Press, Cambridge (2003)
12. Giesl, J., Mesnard, F., Rubio, A., Thiemann, R., Waldmann, J.: Termination competition (termCOMP 2015). In: CADE 2015. LNCS, vol. 9195, pp. 105–108 (2015)
13. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: ITP 2013. LNCS, vol. 7998, pp. 100–115 (2013)
14. 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)
15. Jouannaud, J.P., Lescanne, P.: On multiset orderings. Inf. Process. Lett. **15**(2), 57–63 (1982)
16. Krauss, A.: Recursive definitions of monadic functions. In: PAR 2010. EPTCS, vol. 43, pp. 1–13 (2010)
17. Li, W.: Count the number of complex roots. Archive of Formal Proofs (Oct 2017), http://isa-afp.org/entries/Count_Complex_Roots.html, Formal proof development
18. Li, W., Paulson, L.C.: A modular, efficient formalisation of real algebraic numbers. In: CPP 2016. pp. 66–75. ACM (2016)
19. Mahboubi, A.: Proving formally the implementation of an efficient gcd algorithm for polynomials. In: IJCAR 2006. LNCS, vol. 4130, pp. 438–452 (2006)
20. Mishra, B.: Algorithmic Algebra. Texts and Monographs in Computer Science. Springer, New York (1993)
21. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
22. Niven, I.: Irrational Numbers. No. 11 in Carus Mathematical Monographs, Mathematical Association of America (1956)
23. Prasolov, V.V.: Polynomials. Springer (2004)
24. Thiemann, R., Yamada, A.: Algebraic numbers in Isabelle/HOL. In: ITP 2016. LNCS, vol. 9807, pp. 391–408 (2016)
25. Thiemann, R., Yamada, A.: Formalizing Jordan normal forms in Isabelle/HOL. In: CPP 2016. pp. 88–99. ACM (2016)