A Formalization of Berlekamp's Factorization Algorithm*

Jose Divasón[1], Sebastiaan Joosten[2], René Thiemann[2], and Akihisa Yamada[2]

[1] Universidad de La Rioja
[2] University of Innsbruck

Abstract. We formalize Berlekamp's algorithm for factoring polynomials over prime fields in Isabelle/HOL. We first formalize the algorithm by defining a dedicated type for prime fields, and state the initial soundness theorem depending on this type. We then eliminate the type from the statement using the recently added functionality of local type definitions.

In order to efficiently execute Berlekamp's algorithm, and to employ the algorithm in factorization of integer polynomials, we further generalize and optimize polynomial division algorithms.

The generated code is already used in the formalization for algebraic numbers, where it is an ingredient of a modern factorization algorithm for integer polynomials, although currently only as an oracle.

1 Introduction

Modern algorithms to factor integer polynomials – following Berlekamp and Zassenhaus – work via polynomial factorization over prime fields [2, 3]. Algorithm 1 illustrates the basic structure of such an algorithm.

Algorithm 1: Outline of a modern factorization algorithm

Input: Square-free integer polynomial f.
Output: Irreducible factors $f_1, \ldots, f_n$ such that $f = f_1 \cdot \ldots \cdot f_n$.

1. Choose a small prime p and exponent k such that any potential factor of f has coefficients less than p^k. Moreover, f must be square-free modulo p.
2. Compute an initial factorization $f \equiv g_1 \cdot \ldots \cdot g_m \pmod{p}$ using a factorization algorithm in the prime field GF(p). Possibilities are Berlekamp's algorithm or the Cantor-Zassenhaus algorithm.
3. From $f \equiv g_1 \cdot \ldots \cdot g_m \pmod{p}$ compute a factorization $f \equiv h_1 \cdot \ldots \cdot h_m \pmod{p^k}$ via the Hensel lifting.
4. Construct a factorization $f = f_1 \cdot \ldots \cdot f_n$ over the integers (where each f_i corresponds to one or more h_j).

* This research was supported by the Austrian Science Fund (FWF) project Y757.

In previous work on algebraic numbers [11], we implemented Algorithm 1 in Isabelle/HOL [10] as an oracle *factorization_oracle* :: *int poly* ⇒ *int poly list*, where we chose Berlekamp's algorithm in step 2.[3]

The correctness of the implementation is not yet formalized in Isabelle/HOL. Hence it is invoked in a certified wrapper which takes an arbitrary integer polynomial as input, performs the desired preprocessing, i.e., square-free and content-free factorization, and passes each preliminary factor f to *factorization_oracle*. It finally tests the validity of the obtained factorizations $f = f_1 \cdot \ldots \cdot f_n$, but it does not test optimality, i.e., irreducibility of the resulting factors.

The current work is a significant step forward to formally proving the soundness of *factorization_oracle*, namely by formally proving the soundness of Berlekamp's algorithm in step 2.

- We shortly explain the computations performed in Berlekamp's algorithm (Section 2).
- We define a type to represent arbitrary prime fields GF(p), reusing ideas from HOL multivariate analysis (Section 3). This is essential for working with the type-based algorithms from the Isabelle distribution and the AFP (archive of formal proofs).
- We define Berlekamp's algorithm on GF(p) and prove its correctness (Section 4).
- We implement Berlekamp's algorithm on integer lists which avoids the type-based polynomial algorithms and type-based prime fields (Section 5). The soundness of this implementation is proved via the transfer package [6]. Moreover, we transform the type-based soundness statement of Berlekamp's algorithm using prime fields into a statement which solely speaks about integer polynomials and the executable implementation. Here, we also rely upon local type definitions [8] to eliminate the presence of the type for the prime field GF(p).
  This step is essential, since the full factorization algorithm for integer polynomials will select the prime field depending on the input polynomial, cf. step 1 of Algorithm 1.
- Moreover, we formalize (efficient) division algorithms for non-field polynomials that are applied within the oracle, and also optimize the existing division algorithm for field polynomials (Section 6). The improvements are now integrated in the Isabelle distribution as code equations [4, 5].
- A comparison of the trusted code with the one from *factorization_oracle* revealed two mistakes which are now repaired (Section 7).

To our knowledge, this is the first formalization of Berlekamp's algorithm.[4] The full formalization of Berlekamp's algorithm is available at:

`http://cl-informatik.uibk.ac.at/~thiemann/berlekamp.tgz`

[3] An example illustrating (our implementation of) Algorithm 1 is provided in the appendix.

[4] For instance, Barthe et al. report that there is no formalization of an efficient factorization algorithm over GF(p) available in Coq [1, Section 6, note 3 on formalization].

2 Berlekamp's Algorithm

Algorithm 2 briefly describes Berlekamp's algorithm [2]. It focuses on the core computations that have to be performed. For a discussion on why these steps are performed we refer to Knuth [7, Section 4.6.2].

Algorithm 2: Berlekamp's factorization algorithm

Input: Square-free polynomial f over GF(p) of degree $d \neq 0$.
Output: Constant c and set F of monic and irreducible factors $f_1, \ldots, f_n$ such that $f = c \cdot f_1 \cdot \ldots \cdot f_n$

1 Let c be the leading coefficient of f. Update $f := f/c$.
2 Compute the Berlekamp matrix B_f for f. It is the $d \times d$ matrix over GF(p) where the i-th row are exactly the coefficients of the polynomial $x^{pi} \bmod f$.
3 Compute the dimension r and a basis $b_1, \ldots, b_r$ of the left null space of $B_f - I$ where I is the identity matrix of size $d \times d$.
4 For each basis vector b_i define a corresponding polynomial v_i where the entries in b_i are the coefficients of v_i.
5 Set $F := \{f\}$, $V := \{v_1, \ldots, v_r\} \setminus \{1\}$, $F_I := \emptyset$.
6 If $|F| = r \vee V = \emptyset$, return c and $F \cup F_I$.
7 Pick $v \in V$ and update $V := V \setminus \{v\}$.
Update $F := \{g_{ij} \mid f_i \in F, 0 \leq j < p, g_{ij} = \gcd(f_i, v - j), g_{ij} \neq 1\}$.
8 If one can find k irreducible polynomials in F, move them to F_I and update $r := r - k$.
9 Goto step 6.

We illustrate the algorithm via an example.

Example 1. Consider applying Algorithm 2 to prime $p = 5$ and polynomial

$$f = 3 + 2x + 3x^2 + 3x^3 + 3x^4 + 3x^5.$$

One can easily check that f is square-free in GF(p) by $\gcd(f, f') = 1$; here, f' is the derivative of f in GF(p), and also the GCD is computed in GF(p).

Step 1 divides f (modulo 5) by its leading coefficient $c = 3$ and obtains the new $f := 1 + 4x + x^2 + x^3 + x^4 + x^5$.

Step 2 computes the Berlekamp matrix as

$$B_f = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 4 & 1 & 4 & 4 & 4 \\ 4 & 4 & 1 & 2 & 3 \\ 4 & 3 & 3 & 1 & 3 \\ 3 & 2 & 1 & 2 & 1 \end{pmatrix}$$

since

$$\begin{aligned}
x^0 \bmod f &\equiv 1 && \pmod 5\\
x^5 \bmod f &\equiv 4 + x + 4x^2 + 4x^3 + 4x^4 && \pmod 5\\
x^{10} \bmod f &\equiv 4 + 4x + x^2 + 2x^3 + 3x^4 && \pmod 5\\
x^{15} \bmod f &\equiv 4 + 3x + 3x^2 + x^3 + 3x^4 && \pmod 5\\
x^{20} \bmod f &\equiv 3 + 2x + x^2 + 2x^3 + x^4 && \pmod 5.
\end{aligned}$$

Again, all operations are performed in GF(p).

Step 3 computes a basis of the left null space of $B_f - I$, e.g., by applying the Gauss-Jordan elimination to its transpose.

$$(B_f - I)^{\mathrm{T}} = \begin{pmatrix} 0\,4\,4\,4\,3\\ 0\,0\,4\,3\,2\\ 0\,4\,0\,3\,1\\ 0\,4\,2\,0\,2\\ 0\,4\,3\,3\,0 \end{pmatrix} \qquad \hookrightarrow \qquad \begin{pmatrix} 0\,1\,0\,0\,4\\ 0\,0\,1\,0\,3\\ 0\,0\,0\,1\,0\\ 0\,0\,0\,0\,0\\ 0\,0\,0\,0\,0 \end{pmatrix}$$

We determine $r = 2$, and extract the base vectors $b_1 = (1\ 0\ 0\ 0\ 0)$ and $b_2 = (0\ 1\ 2\ 0\ 1)$. Step 4 converts them into the polynomials $v_1 = 1$ and $v_2 = x + 2x^2 + x^4$, and step 5 initializes $V = \{v_2\}$, $F = \{f\}$, and $F_I = \emptyset$.

The termination condition in step 6 does not hold. So in step 7 we pick $v = v_2$ and compute the required GCDs.

$$\begin{aligned}
\gcd(f, v_2) &= 1\\
\gcd(f, v_2 - 1) &= 3 + x + 2x^2 + x^3 =: f_1\\
\gcd(f, v_2 - 2) &= 1\\
\gcd(f, v_2 - 3) &= 2 + 4x + x^2 =: f_2\\
\gcd(f, v_2 - 4) &= 1
\end{aligned}$$

Afterwards, we update F to $\{f_1, f_2\}$ and V to $\emptyset$.

Step 8 is just an optimization. For instance, in our implementation we move all linear polynomials from F into F_I, so that in consecutive iterations they do not have to be tested for further splitting in step 7.

Now we go back to step 6, where both termination criteria fire at the same time ($|F| = 2 = r$). We return $c \cdot f_1 \cdot f_2$ as final factorization, i.e.,

$$3 + 2x + 3x^2 + 3x^3 + 3x^4 + 3x^5 \equiv 3 \cdot (3 + x + 2x^2 + x^3) \cdot (2 + 4x + x^2) \pmod 5$$

All of the arithmetic operations in Algorithm 2 have to be performed in the prime field GF(p). Hence, in order to implement Berlekamp's algorithm, we basically need the following operations: arithmetic in GF(p), polynomials over GF(p), Gauss-Jordan elimination over GF(p), and GCD-computation for polynomials over GF(p).

All auxiliary algorithms are already available in the Isabelle distribution or in the AFP, provided that the prime field GF(p) is available as a *type* of sort *field*: for polynomials we use α *poly* from `~~/src/HOL/Library/Polynomial`, Gauss-Jordan elimination for matrices with dimensions determined at runtime is

available in `$(AFP)/thys/Jordan_Normal_Form/Gauss_Jordan_Elimination`, and for GCDs we refer to `~~/src/HOL/Number_Theory/Euclidean_Algorithm`.

However, we are not aware of any existing Isabelle formalization of GF(p) as a type.

3 Formalizing Prime Fields

In order to create a type for GF(p) for some arbitrary but fixed prime p, we reuse the idea of the vector representation from HOL multivariate analysis: types can encode natural numbers.

First, we define a class to encode primes as a type, via the cardinality of the universe of the type, denoted by $\mathit{CARD}(\alpha)$.

$$\textbf{class}\ \mathit{prime\_card} = \textbf{assumes}\ \mathit{prime}\ (\mathit{CARD}(\alpha))$$

Afterwards we define a polymorphic type to represent GF(p) as a subtype of the integer numbers.

$$\textbf{typedef}\ (\alpha :: \mathit{prime\_card})\ \mathit{GFp} = \{0\ ..{<}\ \mathit{CARD}(\alpha)\}$$

So whenever P is a type with p elements (for some prime p), P GFp is a type with elements $0, \ldots, p-1$, i.e., GF(p).

It is now easy to define most field operations on α GFp with the help of lifting and transfer. For instance, multiplication is just defined as

$$\begin{aligned}&\textbf{lift\_definition}\ \mathit{times\_GFp} :: \alpha\ \mathit{GFp} \Rightarrow \alpha\ \mathit{GFp} \Rightarrow \alpha\ \mathit{GFp}\ \textbf{is}\\ &\quad \lambda x\, y.\ (x * y)\ \mathit{mod}\ \mathit{CARD}(\alpha)\end{aligned}$$

and the properties of multiplications – like associativity – directly carry over from integer multiplication to multiplication in α GFp. One just needs to apply some distributivity rules between multiplication and modulo for the integers.

The only non-trivial operation is the multiplicative inverse of some $x \neq 0$ in GF(p). Here, there are in principle two approaches. One can either compute x^{p-2} or apply the extended Euclidean algorithm to determine the inverse of x. Through experiments we figured out that the former approach is more efficient if p is small, whereas the latter performs better for large p. Since in our application p is usually small, we define the inverse of x as x^{p-2}.

The soundness of this implementation follows from Fermat's little theorem, which is already available in the Isabelle distribution for the integers, and just needs to be lifted to the type for GFp via transfer.

$$x \cdot x^{p-2} \equiv x^{p-1} = 1 \pmod{p}$$

The advantage of having GF(p) available as a type is that we can reuse several algorithms that are only available in a type-based setting, e.g., Gauss-Jordan elimination, GCD-computation for polynomials, square-free factorization, etc.

4 Formalizing Soundness of Berlekamp's Algorithm

Our soundness proof for Berlekamp's algorithm is based on the description in Knuth's book.

We first formalize the numbered equations (7,8,9,10,13,14) in the textbook [7, pages 441 and 442]. To this end, we also adapt existing proofs from the Isabelle distribution and the AFP. For instance, we require the Chinese remainder theorem for *polynomials* to derive (7) in the textbook, but we only found this theorem for *integers* and *naturals*. Since the proofs of the theorem over such domains are quite similar, the adaptation was not that difficult.

None of the cited equations was straightforward to prove. More concretely, equation (13) was a little bit cumbersome, since it is a rewriting involving summations which are congruent modulo f. It also demands to write the coefficients of a polynomial as a vector of length equal to degree f. To this end, it was necessary to take the list of coefficients of the polynomial, complete such a list with zeros up to the position $\text{degree}(f) - 1$ and then transform the list into a vector. In addition, to prove that $(u + v)^p = u^p + v^p$, where u and v are polynomials over $\text{GF}(p)$, also required some properties about binomial coefficients that were missing in the library.

Having proved these equations, we eventually show that after step 3 we have a basis $b_1, \ldots, b_r$ of the left null space of $B_f - I$. Now, step 4 transforms such vectors into polynomials. A proof of this step is missing in Knuth's book. We define an isomorphism between the left null space of $B_f - I$ and the Berlekamp subspace $W_f := \{v \mid v^p \equiv v \pmod f, \text{degree}(v) < \text{degree}(f)\}$ and then we show that such an isomorphism transforms the basis $b_1, \ldots, b_r$ into a basis $V_b := \{v_1, \ldots, v_r\}$ of W_f. This means that V_b is a Berlekamp basis for f and then every factorization of f has at most $|V_b|$ factors. This proof also requires some extra effort since it was necessary to define another isomorphism between the vector spaces W_f and $\text{GF}(p)^r$ as well as the use of the Chinese remainder theorem over polynomials and the uniqueness of the solution. In order to carry out these proofs, we also extend an existing AFP entry by Lee [9] about vector spaces to include some necessary results which relate linear maps, isomorphisms between vector spaces, dimensions, and bases.

Once having proved that V_b is a Berlekamp basis for f, we can prove equality (14); for every divisor f_i of f and every $v \in V_b$, we have

$$f_i = \prod_{0 \le j < p} \gcd(f_i, v - j). \tag{14}$$

Finally, it follows that every non-irreducible, non-constant divisor f_i of f can be properly factored via $\gcd(f_i, v - j)$ for suitable $v \in V_b$ and $0 \le j < p$.

In order to prove the soundness of steps 5–9 in Algorithm 2, we use the following invariants – these are not so explicitly stated by Knuth as the numbered equation. Here, V_{old} represents the set of already processed polynomials of V_b.

1. $f = \prod F \cup F_I$.
2. All $f_i \in F \cup F_I$ are monic and non-constant.

3. All $f_i \in F_I$ are irreducible.
4. $V_b = V \cup V_{old}$.
5. $\gcd(f_i, v - j) \in \{1, f_i\}$ for all $v \in V_{old}$, $0 \leq j < p$ and $f_i \in F \cup F_I$.
6. $|F_I| + r = |V_b|$.

It is easy to see that all invariants are initially established in step 5 by picking $V_{old} = \{1\} \cap V_b$. In particular, invariant 5 is satisfied since the GCD of the monic polynomial f and a constant polynomial is always f (if the constant is zero), or 1, if the constant is nonzero.

It is also not hard to see that step 7 preserves the invariants. In particular, invariant 5 is satisfied for elements in F_I since these are irreducible. And invariant 1 follows from equality (14).

Irreducibility of the final factors that are returned in step 6 can be argued as follows. If $|F| = r$, then by invariant 6 we know that $|V_b| = |F \cup F_I|$, i.e., $F \cup F_I$ is a factorization of f with a maximal number of factors, and thus every factor is irreducible. In the other case, $V = \emptyset$ and hence $V_{old} = V_b$ by invariant 4. Combining this with invariant 5 shows that every element f_i in $F \cup F_I$ cannot be factored by $\gcd(f_i, v - j)$ for any $v \in V_b$ and $0 \leq j < p$. Since V_b is a Berlekamp basis, this means that f_i must be irreducible.

Eventually, putting everything together we arrive at the formalized main soundness statement of Berlekamp's algorithm.

Theorem 2 (Berlekamp's Algorithm on Polynomials over GF(p)).

assumes *square_free* $(f :: \alpha :: \textit{prime\_card GFp poly})$
and *berlekamp_factorization* $f = (c, fs)$
shows $f = c \cdot \prod fs$ **and** $\forall f_i \in fs.$ *irreducible* $f_i \wedge$ *monic* f_i **and** *distinct* fs

In order to prove the statement formally, we basically use the invariants mentioned before. However, it still requires some tedious reasoning to formally verify all invariants. In the proof, most of the time we model products of polynomials ($\prod fs$) via *listprod* instead of using *setprod*. The reason is that *listprod* has by far nicer simplification rules. For instance *listprod* $(f \# fs) = f \cdot$ *listprod* fs always holds, whereas *setprod* $(\textit{insert } f\ F) = f \cdot$ *setprod* F is ensured only if $f \notin F$ and F is a finite set.

5 Soundness of Berlekamp's Algorithm – Without Prime Fields

The soundness of Theorem 2 is formulated in a *type-based setting*. In particular, the function *berlekamp_factorization* has type

$$\alpha :: \textit{prime\_card GFp poly} \Rightarrow \alpha\ \textit{GFp} \times \alpha\ \textit{GFp poly list}.$$

In our use case, recall that Algorithm 1 first computes a prime number p, and then invokes Berlekamp's algorithm on GF(p). This demands Algorithm 1

to construct a new type P with $\mathit{CARD}(P) = p$ depending on the value of p, and then invoke *berlekamp_factorization* for type P *GFp*.

Unfortunately, this is not possible in Isabelle/HOL. Hence, Algorithm 1 requires Berlekamp's algorithm to have a type like

$$\mathit{int} \Rightarrow \mathit{int\ poly} \Rightarrow \mathit{int} \times \mathit{int\ poly\ list}$$

where the first argument is the dynamically chosen prime p.

As a first step to solve this problem we define conversions $\mathit{to\_int} :: \alpha\ \mathit{GFp} \Rightarrow \mathit{int}$ and $\mathit{of\_int} :: \mathit{int} \Rightarrow \alpha\ \mathit{GFp}$ between $\alpha\ \mathit{GFp}$ and *int* where the former is injective, and the other one applies one "modulo $\mathit{CARD}(\alpha)$" operation. These conversions are then lifted homomorphically to polynomials, resulting in functions *to_int_poly* and *of_int_poly*. With the help of these conversions and some homomorphism lemmas we formulate and prove the following statement of Berlekamp's algorithm which speaks about properties of integer polynomials f and integer factors fs. Only the invocation of Berlekamp's algorithm requires $\alpha\ \mathit{GFp}$. In the lemma, $\mathit{irreducible}_p$ and $=_p$ denote irreducibility and equality modulo p respectively.

Lemma 3 (Berlekamp Factorization on Integers Modulo, Version 1).

assumes $(g :: \alpha :: \mathit{prime\_card\ GFp}) = \mathit{of\_int\_poly}\ f$
and $\mathit{square\_free}\ g$
and $\mathit{berlekamp\_factorization}\ g = (d, gs)$
and $c = \mathit{to\_int}\ d$
and $fs = \mathit{map\ to\_int\_poly}\ gs$
and $p = \mathit{CARD}(\alpha)$
shows $f =_p c \cdot \prod fs$ **and** $\forall f_i \in fs.\ \mathit{irreducible}_p\ f_i \wedge \mathit{monic}\ f_i$ **and** $\mathit{distinct}\ fs$

The next step consists of implementing Berlekamp's algorithm on integers polynomials (mod p) directly. This implementation cannot make use of the polynomial and matrix algorithms directly as these are class based, cf. Section 3.

Instead, we mainly copy the algorithms, but instead of using the class-based constants for arithmetic, we use a record *ops* as parameter that stores all the required arithmetic operations as *plus*, *mult*, etc. To be more precise, whenever some auxiliary algorithm A invokes $x + y$ in the type-based version, we replace it by $x +'_{ops} y$, denoting $\mathit{plus}\ \mathit{ops}\ x\ y$, and pass *ops* as an additional argument to A', the record-based version of A.

Soundness of the record-based algorithms is then mainly proved with the help of the transfer package. To this end, we define relations which express that certain elements are representatives of each other. For instance, for integers and GF(p) we define $\mathit{GFp}_{\mathit{rel}} :: \mathit{int} \Rightarrow \alpha\ \mathit{GFp} \Rightarrow \mathit{bool}$ as $\mathit{GFp}_{\mathit{rel}}\ x\ y = (x = \mathit{to\_int}\ y)$. Similar relations are then constructed for polynomials and matrices, e.g., $\mathit{poly}_{\mathit{rel}}\ R\ x\ y = (\mathit{list\_all2}\ R\ x\ (\mathit{coeffs}\ y))$ relates lists with polynomials, where $\mathit{list\_all2}\ R$ demands lists of equal lengths where the elements are point-wise in relation R.

Then we define a locale demanding that *ops* faithfully implements the arithmetic operations in GF(p), expressed as a set of transfer rules of the following

form.

$$(GFp_{rel} \texttt{ ===> } GFp_{rel} \texttt{ ===> } GFp_{rel})\ (ops\ plus)\ (op\ +)$$
$$(GFp_{rel} \texttt{ ===> } GFp_{rel})\ inverse'_{ops}\ inverse$$
$$(GFp_{rel} \texttt{ ===> } GFp_{rel} \texttt{ ===> } op\ =)\ (op\ =)\ (op\ =)$$

Note that at the moment equality is not part of the record *ops*. This becomes visible in the last transfer rule which expresses that equality on GF(p) can be implemented as equality on the representing integers.

Within the locale, we finally prove soundness of the implementation for Berlekamp's algorithm.

Theorem 4 (*berlekamp_factorization'* implements *berlekamp_factorization*).

$$(poly_{rel}\ GFp_{rel} \texttt{ ===> } GFp_{rel} \times_{rel} list\_all2\ (poly_{rel}\ GFp_{rel}))$$
$$(berlekamp\_factorization'\ ops)\ berlekamp\_factorization$$

The theorem states that if the input integer list f represents some GF(p) polynomial g, and *berlekamp_factorization'* $ops\ f = (c, fs)$ and *berlekamp_factorization* $g = (d, gs)$, then c represents d and fs represents gs.

To obtain Theorem 4 we developed transfer rules for all auxiliary algorithms that are invoked in Berlekamp's algorithm. Here, the diagnostic commands *transfer_prover_start* and *transfer_step* were extremely helpful to see why certain transfer rules could initially not be proved automatically, i.e., these commands nicely pointed to missing transfer rules.

Most of the transfer rules for non-recursive algorithms were mainly proved by unfolding the definitions and finishing the proof by *transfer_prover*. For recursive algorithms, we often perform induction via the algorithm. To handle an inductive case, we install a local transfer rule (obtained from the induction hypothesis), unfold one function application iteration, and then finish the proof by *transfer_prover*.

Nevertheless, problems arose in case of underspecification. For instance it is impossible to prove an unconditional transfer rule for the function *hd* that returns the head of a list using the standard relator for lists, (*list_all2* R ===> R) *hd hd*; when the lists are empty, we have to relate *undefined* :: α with *undefined* :: β. To circumvent this problem, we had to reprove invariants that *hd* is only invoked on non-empty lists.

Similar problems arose when using matrix indexes where transfer rules between matrix entries A_{ij} and B_{ij} are only available if i and j are within the matrix dimensions. So, again we had to reprove the invariants on valid indexes – just unfolding the definition and invoking *transfer_prover* was not sufficient.

Using Theorem 4 we can now reformulate the soundness of Berlekamp's algorithm (Lemma 3) as follows. Here, *ff_ops* p is the record that implements arithmetic in GF(p) – as required by the locale – and *poly_of_list* converts a coefficient list into a polynomial.

Lemma 5 (Berlekamp Factorization on Integers Modulo, Version 2).

assumes $g = \mathit{coeffs}\ (\mathit{map}_{\mathit{poly}}\ (\lambda x.\ x\ \mathit{mod}\ p)\ f)$
and $\mathit{square\_free}'\ (\mathit{ff\_ops}\ p)\ g$
and $\mathit{berlekamp\_factorization}'\ (\mathit{ff\_ops}\ p)\ g = (c, gs)$
and $fs = \mathit{map}\ \mathit{poly\_of\_list}\ gs$
and $p = \mathit{CARD}(\alpha :: \mathit{prime\_card})$
shows $f =_p c \cdot \prod fs$ **and** $\forall f_i \in fs.\ \mathit{irreducible}_p\ f_i \wedge \mathit{monic}\ f_i$ **and** $\mathit{distinct}\ fs$

Note that in Lemma 5 the occurrence of the type $\alpha\ \mathit{GFp}$ vanished. All constants and types involved speak about integers, integer polynomials, and integer lists, except for the single occurrence of $\mathit{CARD}(\alpha :: \mathit{prime\_card})$.

Finally, we delete this last occurrence of α with the help of local type definitions, which replaces the condition $p = \mathit{CARD}(\alpha :: \mathit{prime\_card})$ by $\mathit{prime}\ p$. Thus we can define a function $\mathit{berlekamp\_factorization\_int} :: \mathit{int} \Rightarrow \mathit{int\ poly} \Rightarrow \mathit{int} \times \mathit{int\ poly\ list}$ where

$$\mathit{berlekamp\_factorization\_int}\ p\ f = \mathit{map}_{\mathit{prod}}\ \mathit{id}\ (\mathit{map}\ \mathit{poly\_of\_list})\ (\mathit{berlekamp\_factorization}'\ (\mathit{ff\_ops}\ p)\ (\mathit{coeffs}\ (\mathit{map}_{\mathit{poly}}\ (\lambda x.\ x\ \mathit{mod}\ p)\ f)))$$

and prove its soundness without having to create a type $\alpha\ \mathit{GFp}$.

Theorem 6 (Berlekamp Factorization on Integers Modulo).

assumes $\mathit{square\_free}'\ (\mathit{ff\_ops}\ p)\ (\mathit{coeffs}\ (\mathit{map}_{\mathit{poly}}\ (\lambda x.\ x\ \mathit{mod}\ p)\ f))$
and $\mathit{berlekamp\_factorization\_int}\ p\ f = (c, fs)$
and $\mathit{prime}\ p$
shows $f =_p c \cdot \prod fs$ **and** $\forall f_i \in fs.\ \mathit{irreducible}_p\ f_i \wedge \mathit{monic}\ f_i$ **and** $\mathit{distinct}\ fs$

6 Generalized and Efficient Polynomial Division

After Berlekamp's algorithm yields a factorization $f \equiv g_1 \cdot \ldots \cdot g_m \pmod p$, the Hensel lifting – whose formalization is left for future work – yields a factorization $f \equiv h_1 \cdot \ldots \cdot h_m \pmod{p^k}$. Now the final step towards factorization over integer polynomials is to test if some product of h_i's is a factor of f as integer polynomial. This demands divisibility test and exact division over integer polynomials, for which we find no algorithm formalized in the Isabelle distribution.

Hence we formalize a polynomial division algorithm *divide_poly* for integers, or more generally, for *idom_divide*. Moreover, since the GCD computation in Algorithm 2 (line 7) demands a series of polynomial division on finite fields, we further optimize the algorithm for fields. We experimentally verify that our algorithm is by far faster than the existing division algorithm for large polynomials.

6.1 The Polynomial Division Algorithms

While the existing division algorithm for field polynomials was akin to polynomial long-division, our new general algorithm *divide_poly* is based on synthetic division.

The core algorithm is presented as Algorithm 3. Here we represent a polynomial as the list of coefficients ordered from the highest degree to the lowest. So for $x^2 + 2$, the list we work with is $[1, 0, 2]$, whereas Isabelle's original representation is $[: 2, 0, 1 :]$. This requires list-reversal in order to convert polynomials to and from our internal polynomial representation. These conversions between lists and polynomials are performed by *divide_poly*, the wrapper that invokes the core algorithm.

Algorithm 3: Core algorithm of polynomial division

Input: Lists of coefficients f and g with no leading zeros
Output: List of coefficients representing $f \operatorname{\mathsf{div}} g$

1 If $g = [\,]$, i.e., the zero polynomial, then return $[\,]$.
2 Let $n = \max(1 + \mathsf{length}(f) - \mathsf{length}(g), 0)$, which is the number of times to execute our main loop. Assign $q = [\,]$.
3 If $n = 0$, then return q. Otherwise decrease n by one.
4 If the leading coefficient of f is zero, i.e., $\mathsf{hd}(f) = 0$, then goto step 8.
5 Let $a = \mathsf{hd}(f) \operatorname{\mathsf{div}} \mathsf{hd}(g)$. Update $f := f - (a \cdot g)$ using point-wise multiplication and subtraction.
6 If $\mathsf{hd}(f) \neq 0$, then return $[\,]$ as division is not possible.
7 Prepend a to q.
8 Remove the first element in the list from f and goto step 3.

A crucial difference with the old implementation – besides that it is not restricted to fields – is that lists are ordered such that the essential elements for division are at the heads of the lists, rather than at their tails. In addition, division stops earlier; in particular, if the degree of g is large, the new algorithm uses far fewer iterations.

Further optimization is possible for field polynomials. In that case $\mathsf{hd}(f) = 0$ always holds in step 6, so we omit that step in a specialized version for fields. Moreover, we can first normalize g to a monic polynomial ($g := g/a$, $f := f/a$ where a is the leading coefficient of g), so that no division is required within the loop – the division in step 5 is replaced by $a = \mathsf{hd}(f)$.

6.2 Incorporating to Isabelle

Correctness of Algorithm 3, meaning that *divide_poly* $(g \cdot f)$ $g = f$ if $g \neq 0$, is proved by induction on n, using that the output of the algorithm will be equal to $q + f$ for their values at step 3.

To reuse the existing results on division, however, we have to further inform Isabelle that *divide_poly* satisfies the desired property of *div*, and declare the following:

instantiation *poly* :: (*idom_divide*) *idom_divide*

Note that this general instantiation is possible only before the following more specific declaration, which is already present in the Isabelle 2016 distribution:

instantiation *poly* :: (*field*) *ring_div*

Hence, in collaboration with Florian Haftmann we integrated our formalization into the development version of Isabelle, so that the general instantiation comes before the more specific one.

6.3 Experimental Comparison

To test whether our new division algorithm is indeed faster than the previous division algorithm, we compute the following polynomials:

$$f_n(x) = (nx^{2n} + nx^{2n-1} + \cdots + nx^0) \operatorname{div} (2x^2 + x)$$
$$g_n(x) = (nx^{2n} + nx^{2n-1} + \cdots + nx^0) \operatorname{div} (2nx^n + 2nx^{n-1} + \cdots + 2nx^0)$$
$$h_n(x) = 2nx^{2n} \operatorname{div} (2x^2 + x)$$
$$i_n(x) = 2nx^{2n} \operatorname{div} nx^n$$

We use polynomials over rational numbers, obtaining specialized versions for equality, division, and the creation of the polynomials shown above. To ensure that Haskell actually computes the polynomials, we evaluate whether the polynomial is equal to itself. To make sure that the computations happens at run-time, and are not (partially) done by the Haskell compiler, we disallow inlining of the specialized functions; however, we did not observe any change in performance by disallowing this inlining. We use ghc-8.0.0.20160421 (release candidate 4 for ghc-8.0.1) with the optimizing flag -O2.

The runtimes are shown in Table 1. Note that the new implementation is particularly fast when dividing by larger polynomials, as it essentially aborts computation of $q \operatorname{div} d$ when the degree of d exceeds that of q. We also observed that the memory usage for the new implementation was slightly less than that of the old one.

Table 1. Runtimes of list-based division compared with the old implementation.

Computation	runtime before (sec)	runtime after (sec)
f_{30000}	14.05	1.65
g_{30000}	7.05	0.20
h_{30000}	5.34	1.02
i_{30000}	4.07	0.08

7 Comparison with the Oracle

After having formalized Berlekamp's algorithm, we of course also compare the resulting trusted code from the code generator with the untrusted code as part of *factorization_oracle*. By our work, we could eliminate two bugs in *factorization_oracle*, which however were not relevant in the development of algebraic numbers.

First, by running experiments with Algorithm 1 on *arbitrary* polynomials, we detected that the Mignotte factor bound was incorrectly computed in *factorization_oracle*, namely if the leading coefficient of the input polynomial is negative: the size of a leading coefficient c was determined via *nat* c, and not via *nat* (*abs* c). This bug did not cause problems in the algebraic number development since there preprocessing ensured that *factorization_oracle* is only invoked on polynomials with positive leading coefficients.

The second problem was detected during the formalization of the efficient division algorithm where we could not derive soundness of the division algorithm from the properties of the main working loop. Indeed, there was a bug in handling divisions of 0 by some constant. The number of iterations in the oracle for dividing f (of degree d_f) by g (of degree d_g) was determined as the number $\max(1 + d_f - d_g, 0)$. However, this calculation is only valid if the degree of the polynomial 0 is defined as -1, and this is not the case for *degree* in Isabelle/HOL. Hence, we replaced d_f and d_g by the length of the coefficient lists of f and g, respectively, cf. step 2 of Algorithm 3. Again, the problem did not occur in the algebraic number development as there only divisions by non-constant polynomials are performed.

8 Conclusion

We formalized a crucial part of a modern factorization algorithm for integer polynomials – Berlekamp's algorithm for finite field factorization over GF(p). Although it is initially proved via a type for GF(p), with the help of local type definitions the final soundness statement is also expressible purely over integer polynomials. This will be essential for future work, the formalization of the full factorization algorithm for integer polynomials. The latter requires formal proofs of Mignotte's factor bound and the soundness of the Hensel lifting.

Acknowledgments The authors are listed in alphabetical order regardless of individual contributions or seniority. We thank Florian Haftmann for integrating our changes in the polynomial library into the Isabelle distribution.

References

1. Barthe, G., Grégoire, B., Heraud, S., Olmedo, F., Béguelin, S.Z.: Verified indifferentiable hashing into elliptic curves. In: POST 2012. LNCS, vol. 7215, pp. 209–228 (2012)

2. Berlekamp, E.R.: Factoring polynomials over finite fields. Bell System Technical Journal 46, 1853–1859 (1967)
3. Cantor, D.G., Zassenhaus, H.: A new algorithm for factoring polynomials over finite fields. Math. Comput. 36(154), 587–592 (1981)
4. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: ITP 2013. LNCS, vol. 7998, pp. 100–115 (2013)
5. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: FLOPS 2010. LNCS, vol. 6009, pp. 103–117 (2010)
6. 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)
7. Knuth, D.E.: The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition. Addison-Wesley (1981)
8. Kunčar, O., Popescu, A.: From types to sets by local type definitions in higher-order logic. In: ITP 2016. LNCS, (to appear)
9. Lee, H.: Vector spaces. Archive of Formal Proofs (2014), `http://www.isa-afp.org/entries/VectorSpace.shtml`
10. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283 (2002)
11. Thiemann, R., Yamada, A.: Algebraic numbers in Isabelle/HOL. In: ITP 2016. LNCS, (to appear)

A Example Calculation for Algebraic Numbers

Algebraic numbers are roots of integer polynomials. Each algebraic number has a unique minimal polynomial representing that numbers. For instance, $f_4 = x^3 - 4$ is the minimal polynomial for $\sqrt[3]{4}$, and the algebraic number $\sum_{i=1}^{3} \sqrt[3]{i}$ has the minimal polynomial

$$f_{1,3} = -54 - 162x + 297x^3 - 351x^4 + 216x^5 - 99x^6 + 36x^7 - 9x^8 + x^9.$$

From these polynomials one can compute the following polynomial f that represents the algebraic number $\sum_{i=1}^{4} \sqrt[3]{i} = (\sum_{i=1}^{3} \sqrt[3]{i}) + \sqrt[3]{4}$ as a root.[5]

$$\begin{aligned} f = &- 1006885000 + 6123043800x - 15720740280x^2 + 22439312316x^3 \\ &- 20672766756x^4 + 15588244476x^5 - 13081997214x^6 + 11650042674x^7 \\ &- 8978198058x^8 + 5971437939x^9 - 3786528357x^{10} + 2406546099x^{11} \\ &- 1476369954x^{12} + 832074876x^{13} - 429477336x^{14} + 208603152x^{15} \\ &- 96826968x^{16} + 42228351x^{17} - 16716945x^{18} + 5832999x^{19} - 1765746x^{20} \\ &+ 460764x^{21} - 103086x^{22} + 19494x^{23} - 3006x^{24} + 351x^{25} - 27x^{26} + x^{27} \end{aligned}$$

At this point we would like to factor f to compute the minimal polynomial representing $\sum_{i=1}^{4} \sqrt[3]{i}$; this is where Algorithm 1 is invoked.

Here, step 1 chooses $p = 13$ and $k = 13$.

The choice of p is based on the fact that $p = 13$ is the least prime such f is square-free modulo p. Using Mignotte's factor bound we compute $B_f = 40338664525104$ such that every proper factor of f with degree at most $\lfloor \frac{27}{2} \rfloor$ has coefficients of at most B_f. And k is the least number such that p^k can represent all numbers in the range $\{-B, \ldots, B\}$, i.e., $k = \lceil log_p(2B + 1) \rceil$.

Next, step 2 computes the factorization modulo p. Here $f \ mod \ 13$ is the polynomial

$$\begin{aligned} &9 + 3x + 2x^2 + 5x^3 + 8x^4 + 12x^5 + 10x^7 + 2x^8 + 12x^9 + 12x^{10} + 9x^{11} \\ &+ 6x^{12} + 9x^{13} + 11x^{14} + 4x^{15} + 9x^{16} + 9x^{17} + 2x^{18} + 3x^{19} \\ &+ 5x^{20} + 5x^{21} + 4x^{22} + 7x^{23} + 10x^{24} + 12x^{26} + x^{27} \end{aligned}$$

[5] $f = resultant(f_{1,3}(x - y), f_4(y))$

and running Berlekamp's algorithm on this polynomial yields the factorization $f \equiv g_1 \cdot \ldots \cdot g_9 \pmod{13}$ where

$$\begin{aligned}
g_1 &= 8 + x + 10x^2 + x^3 \\
g_2 &= 2 + 6x + 10x^2 + x^3 \\
g_3 &= 4 + 4x + 10x^2 + x^3 \\
g_4 &= 11 + 11x + 10x^2 + x^3 \\
g_5 &= 1 + 10x^2 + x^3 \\
g_6 &= 12 + 2x + 10x^2 + x^3 \\
g_7 &= 12 + 10x + 10x^2 + x^3 \\
g_8 &= 9 + 12x + 10x^2 + x^3 \\
g_9 &= 7 + 7x + 10x^2 + x^3.
\end{aligned}$$

Lifting each factor via Hensel lifting in step 3 returns the factors h_i modulo p^k, i.e., $f \equiv h_1 \cdot \ldots \cdot h_9 \pmod{13^{13}}$

$$\begin{aligned}
h_1 &= 14753206565092 + 55610413374829x + 302875106592250x^2 + x^3 \\
h_2 &= 223276365664263 + 290553585300959x + 302875106592250x^2 + x^3 \\
h_3 &= 91657078432664 + 119297765940305x + 302875106592250x^2 + x^3 \\
h_4 &= 145496188306194 + 227742538225980x + 302875106592250x^2 + x^3 \\
h_5 &= 236977659214037 + 87454089657558x + 302875106592250x^2 + x^3 \\
h_6 &= 64845534362868 + 259586214508727x + 302875106592250x^2 + x^3 \\
h_7 &= 50841464948468 + 19522154991453x + 302875106592250x^2 + x^3 \\
h_8 &= 15055982429718 + 195898861943251x + 302875106592250x^2 + x^3 \\
h_9 &= 65721839853365 + 258709909018230x + 302875106592250x^2 + x^3
\end{aligned}$$

In step 4, we first try to find proper factors of f over the integers corresponding to some h_i or some product $h_i \cdot h_j$. However, there are no such factors. If we combine three h_i's, we also consider the product $(h_2 \cdot h_4 \cdot h_5)$ *mod* p^k:

$$(p^k - 550) + 504x + 234x^2 + (p^k - 33)x^3 + (p^k - 279)x^4 + 18x^5 + 15x^6 + 18x^7 + (p^k - 9)x^8 + x^9.$$

Reconstructing numbers in the range $\{-B, \ldots, B\}$ yields

$$f_1 = -550 + 504x + 234x^2 - 33x^3 - 279x^4 + 18x^5 + 15x^6 + 18x^7 - 9x^8 + x^9,$$

a proper factor of f. Next we divide f by f_1 (yielding f_2), drop h_2, h_4, and h_5 from the factor list, and try to further factor f_2 by the remaining list of h_i's. This does not yield any further proper factor, so f_1 and f_2 are the irreducible factors of f. (In this example we don't have to combine four h_i's, since 4 exceeds 3, i.e, half the number of remaining factors).

Thus, the minimal polynomial representing $\sum_{i=1}^{4} \sqrt[3]{i}$ is either f_1 or f_2, and a numerical analysis decides that indeed f_1 represents $\sum_{i=1}^{4} \sqrt[3]{i}$.