A Formalization of the Berlekamp–Zassenhaus Factorization Algorithm

Jose Divasón
Universidad de La Rioja, Spain

Sebastiaan Joosten René Thiemann
Akihisa Yamada
University of Innsbruck, Austria

Abstract

We formalize the Berlekamp–Zassenhaus algorithm for factoring square-free integer polynomials in Isabelle/HOL. We further adapt an existing formalization of Yun's square-free factorization algorithm to integer polynomials, and thus provide an efficient and certified factorization algorithm for arbitrary univariate polynomials.

The algorithm first performs a factorization in the prime field GF(p) and then performs computations in the ring of integers modulo p^k, where both p and k are determined at runtime. Since a natural modeling of these structures via dependent types is not possible in Isabelle/HOL, we formalize the whole algorithm using Isabelle's recent addition of local type definitions.

Through experiments we verify that our algorithm factors polynomials of degree 100 within seconds.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; I.1.2 [*Symbolic and Algebraic Manipulation*]: Algorithms—Algebraic algorithms

Keywords Polynomial Factorization, Prime Fields, Isabelle

1. Introduction

Modern algorithms to factor integer polynomials – following Berlekamp and Zassenhaus – work via polynomial factorization over prime fields GF(p) and quotient rings $\mathbb{Z}/p^k\mathbb{Z}$ [3, 4]. Algorithm 1 illustrates the basic structure of such an algorithm.[1]

[1] Our algorithm starts with step 4, so that section numbers and step-numbers coincide.

CPP'17, January 16–17, 2017, Paris, France
ACM. 978-1-4503-4705-1/17/01...$15.00
http://dx.doi.org/10.1145/3018610.3018617

Algorithm 1: A modern factorization algorithm

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

4 Choose a suitable prime p depending on f.
5 Factor f in GF(p): $f \equiv g_1 \cdot \ldots \cdot g_m \pmod{p}$.
6 Determine a suitable bound d on the degree, depending on $g_1, \ldots, g_m$. Choose an exponent k such that every coefficient of a factor of a given multiple of f in $\mathbb{Z}$ with degree at most d can be uniquely represented by a number below p^k.
7 From step 5 compute the unique factorization $f \equiv h_1 \cdot \ldots \cdot h_m \pmod{p^k}$ via the Hensel lifting.
8 Construct a factorization $f = f_1 \cdot \ldots \cdot f_n$ over the integers where each f_i corresponds to the product of one or more h_j.

In previous work on algebraic numbers [18], we implemented Algorithm 1 in Isabelle/HOL [17] as a function which takes an integer polynomial f and returns a list of polynomials fs. However, the algorithm was available only as an oracle, and thus a validity check ($f = \prod fs$) on the result factorization had to be performed. Moreover, there was no guarantee on the irreducibility of the resulting factors fs.

In this work we fully formalize the correctness of our implementation. We choose Berlekamp's algorithm in step 5; the Cantor–Zassenhaus algorithm [4] is another candidate but its formalization would be more intricate (indeed, it is a probabilistic algorithm).

THEOREM 1 (Berlekamp–Zassenhaus Algorithm).

assumes *square_free* $(f :: \textit{int poly})$
and *degree* $f \neq 0$
and *berlekamp_zassenhaus_factorization* $f = fs$
shows $f =$ *prod_list* fs
and $\forall f_i \in$ *set* fs. *irreducible* f_i

To obtain Theorem 1 we perform the following tasks.

- In Section 3 we introduce two formulations of GF(p) and $\mathbb{Z}/p^k\mathbb{Z}$. We first define a type to represent these domains, employing the idea from HOL multivariate analysis that types can encode natural numbers by means of the cardinality. This is essential for reusing many type-based algorithms from the Isabelle distribution and the AFP (Archive of Formal Proofs). At some points in our developement, the type-based setting is still too restrictive. Hence we also introduce a second formulation which is *locale-based* [1].
- The prime p in step 4 must be chosen so that f remains square-free in GF(p). For the termination of the algorithm, we prove that such a prime always exists in Section 4.
- In Section 5, we explain Berlekamp's algorithm, which factors polynomials over prime fields, and formalize its correctness using the type-based representation. Since Isabelle's code generation does not work for the type-based representation of prime fields, we define an implementation of Berlekamp's algorithm which avoids type-based polynomial algorithms and type-based prime fields. The soundness of this implementation is proved via the transfer package [7]: we transform the type-based soundness statement of Berlekamp's algorithm into a statement which speaks solely about integer polynomials. Here, we crucially rely upon local type definitions [12] to eliminate the presence of the type for the prime field GF(p).
- For step 6 we need to find a bound on the coefficients of the factors of a polynomial. For this purpose, we formalize Mignotte's factor bound in Section 6. During this formalization task we detected a bug in our previous oracle implementation, which computed improper bounds on the degrees of factors.
- In Section 7 we formalize the Hensel lifting. As for Berlekamp's algorithm, we first formalize basic operations in the type-based setting. Unfortunately, however, this result cannot be extended to the full Hensel lifting. Therefore, we model the Hensel lifting in a locale-based way so that modulo operation is explicitly applied on polynomials.
- Details on step 8 are provided in Section 8 where we closely follow the description of Knuth [9, page 452]. Here, we use the same representation of polynomials over $\mathbb{Z}/p^k\mathbb{Z}$ as for the Hensel lifting.
- In Section 9 we adapt Yun's square-free factorization algorithm [19, 21] from $\mathbb{Q}$ to $\mathbb{Z}$. In combination with the previous results this leads to a factorization algorithm for arbitrary integer and rational polynomials.
- Finally, we compare the efficiency of our factorization algorithm with the one in Mathematica 11 [20] in Section 10 and give a summary in Section 11.

To our knowledge, this is the first formalization of a modern factorization algorithm. For instance, Barthe et al. report that there is no formalization of an efficient factorization algorithm over GF(p) available in Coq [2, Section 6, note 3 on formalization]. Our work is also a non-trivial case study for the new local type definition mechanism in Isabelle.

Some key theorems leading to the algorithm have already been formalized in Isabelle or other proof assistants. In ACL2, for instance, polynomials over a field are shown to be a unique factorization domain (UFD) [5]. A more general result, namely that polynomials over a UFD are also a UFD, was already developed in Isabelle/HOL for implementing algebraic numbers [18] and an independent development by Eberl is now available in the Isabelle distribution.

An Isabelle formalization of Hensel's lemma is provided by Kobayashi et al. [10], who defined the valuations of polynomials via Cauchy sequences, and used this setup to prove the lemma. Consequently, their result requires a 'valuation ring' as a precondition in their formalization. While this extra precondition is theoretically met in our setting, we did not attempt to reuse their results, because the type of polynomials in their formalization (from HOL-Algebra) differs from the polynomials in our development (from HOL/Library). Instead, we formalize a direct proof for Hensel's lemma. Our formalizations are incomparable: On the one hand, Kobayashi et al. did not restrict to integer polynomials as we do. On the other hand, we additionally formalize the quadratic Hensel lifting [22], extend the lifting from binary to n-ary factorizations, and prove a uniqueness result, which is required for proving Theorem 1.

A Coq formalization of Hensel's lemma is also available. It is used for certifying integral roots and 'hardest-to-round computation' [14]. If one is interested in certifying a factorization, rather than a certified algorithm that performs it, it suffices to test that all the found factors are irreducible. Kirkels [8] formalized a sufficient criterion for this test in Coq: when a polynomial is irreducible modulo some prime, it is also irreducible in $\mathbb{Z}$. Both formalizations are in Coq, and we did not attempt to reuse them.

Our formalization is available in the AFP and details on the experiments are provided at

`http://cl-informatik.uibk.ac.at/software/ceta/experiments/factorization`.

The formalization as described in this paper corresponds to AFP revision c57b0e9b0d65, which compiles with Isabelle revision 03057a8fdd1f.

2. Preliminaries

Our formalization is based on Isabelle/HOL, and we state theorems, as well as certain definitions, following Isabelle's syntax. For instance, $f :: \alpha \Rightarrow \alpha$ *poly* indicates that f is a function that maps α to a polynomial over α. Isabelle's keywords are written in **bold**. Other symbols are either clear

from their notation, or defined on their appearance. We only assume the HOL axioms and local type definitions, and ensure that Isabelle can build our theories. Consequently, a sceptical reader that trusts the soundness of Isabelle/HOL only needs to check the definitions, as the proofs are checked by Isabelle.

We expect the reader to be familiar with algebra, and use some of its standard notions without further explanation. Concerning notation, we write f' for the derivative of a polynomial f, $\mathsf{lc}(f)$ for the leading coefficient of f, and $\mathsf{res}(f, g)$ for the resultant of f and another polynomial g.

A *factorization* of a polynomial f is a decomposition into irreducible *factors* $f_1, \ldots, f_n$ such that $f = f_1 \cdot \ldots \cdot f_n$. Whereas the irreducibility of a ring element x is often defined via divisibility (denoted by the binary relation *dvd* following Isabelle):

$$\neg\, x \mathrel{\mathit{dvd}} 1 \wedge \big(\forall y.\, y \mathrel{\mathit{dvd}} x \longrightarrow y \mathrel{\mathit{dvd}} 1 \vee x \mathrel{\mathit{dvd}} y\big) \qquad (1)$$

in this paper we define irreducibility of a polynomial f as

$$\text{degree } f \neq 0 \wedge \big(\forall g.\, g \mathrel{\mathit{dvd}} f \longrightarrow \text{degree } g \in \{0, \text{degree } f\}\big). \qquad (2)$$

Note that (1) and (2) are not equivalent on integer polynomials; e.g., a factorization of $f = 10x^2 - 10$ in terms of (1) will be $f = 2{\cdot}5{\cdot}(x{-}1){\cdot}(x{+}1)$, where the prime factorization of the *content*, i.e., the GCD of the coefficients, has to be performed. In contrast, (2) does not demand a prime factorization, and a factorization may be $f = (10x{-}10){\cdot}(x{+}1)$. Algorithm 1 will produce the latter factorization, where all factors except for one are *content-free*, i.e., whose content is 1. Note that definitions (1) and (2) are equivalent on content-free polynomials (and in particular for field polynomials).

In a similar way to irreducibility, we also define that a polynomial f is *square-free* if there does not exist a non-constant polynomial g such that g^2 divides f. In particular, the integer polynomial 2^2x is square-free.

3. Formalizing Prime Fields

Here we introduce two formalizations of the quotient ring $\mathbb{Z}/p^k\mathbb{Z}$ and the prime field GF(p): a *type-based* version and *locale-based* version.

3.1 Type-Based Formalization

We first define a polymorphic type to represent $\mathbb{Z}/p\mathbb{Z}$ for an arbitrary $p > 0$, which forms the prime field GF(p) when p is a prime. The advantage of having GF(p) available as a type is that we can reuse several algorithms that are available only in type-based settings, e.g., the Gauss–Jordan elimination, GCD computation for polynomials, square-free factorization, etc.

Since Isabelle does not support dependent types, we cannot directly use the term variable p in a type definition. To overcome the problem, we reuse the idea of the vector representation from HOL multivariate analysis: types can encode natural numbers. We encode p as $\mathit{CARD}(\alpha)$, i.e., the cardinality of the universe of a (finite) type represented by a *type* variable α.

typedef $(\alpha :: \mathit{finite})$ *mod_ring* $= \{0 ..< \mathit{CARD}(\alpha)\}$

Given a finite type α with p elements, α *mod_ring* is a type with elements $0, \ldots, p-1$. With the help of the lifting and transfer package, we naturally define arithmetic in α *mod_ring* modulo $\mathit{CARD}(\alpha)$; for instance, multiplication is defined as follows:

lift_definition *times_mod_ring* ::
α *mod_ring* $\Rightarrow$ α *mod_ring* $\Rightarrow$ α *mod_ring*
is $\lambda x\, y.\ (x * y)$ *mod* $\mathit{CARD}(\alpha)$

It is straightforward to show that α *mod_ring* forms a commutative ring:

instantiation *mod_ring* :: (*finite*) *comm_ring*

Note that *comm_ring* does not assume the existence of the multiplicative unit *1*. If $\mathit{CARD}(\alpha) = 1$, then α *mod_ring* is not an instance of the type class *ring_1*, for which $\mathit{0} \neq \mathit{1}$ is required. Hence we introduce the following type class:

class *nontriv* = **assumes** $\mathit{CARD}(\alpha) > \mathit{1}$

and derive the following instantiation:[2]

instantiation *mod_ring* :: (*nontriv*) *comm_ring_1*

It is well known that the ring of integers modulo some prime number forms a field. To enforce that the modulus is a prime number, we employ the same trick as above.

class *prime_card* = **assumes** *prime* $(\mathit{CARD}(\alpha))$

The key to being a field is the existence of the multiplicative inverse x^{-1}. This follows from Fermat's little theorem:

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

for any nonzero integer x and prime p; that is, $x^{-1} = x^{\mathit{CARD}(\alpha)-2}$ if $\mathit{CARD}(\alpha)$ is a prime. The theorem is already available in the Isabelle distribution for the integers, and we just have to apply the transfer tactic to lift the result to $(\alpha :: \mathit{prime\_card})$ *mod_ring*.

instantiation *mod_ring* :: (*prime_card*) *field*

In the rest of the paper, we write α *GFp* instead of $(\alpha ::$ *prime_card*$)$ *mod_ring*.[3]

[2] A formalization of the ring $\mathbb{Z}/p\mathbb{Z}$ is already present in `~~/src/HOL/Library/Numeral_Type` as a locale *mod_ring*. In principle we could reuse results from the library by proving connection between the locale and our class; however, as the resulting proofs became slightly longer than direct proofs, we did not use this library.

[3] We would like to have introduced this abbreviation also in Isabelle. However, we are not aware of how to do this, since the **type_synonym** keyword does not allow specifying type constraints such as $\alpha ::$ *prime_card*.

For efficiency, we compute x^{p-2} using the binary exponentiation algorithm. Another approach for computing x^{-1} would use the extended Euclidean algorithm; however, through experiments we observed that this approach is beneficial only when p is quite large (such as 20 digits). Since in our application p is usually small, we compute x^{-1} as x^{p-2}.

3.2 Locale-Based Version

The type-based setting is preferable whenever possible, since it allows concise theorem statements and better support for proof automation, cf. Kunčar and Popescu [12].

At some points of our development, however, the type-based approach is not expressive enough; cf. Section 7. We must reason about the ring of integers modulo m, where m cannot be given via type variables.

Hence, we also introduce a locale *poly_mod* which fixes the modulus m and defines modular arithmetic operations on type *int poly*. In particular, Mp_m :: *int poly* $\Rightarrow$ *int poly* is a function that pointwise takes modulo m of each coefficients. Other operations, such as equivalence $\equiv_m$, $coprime_m$, $unique\_factorization_m$, are defined with the help of *Mp*; e.g., $f \equiv_m g$ is defined as $Mp_m\ f = Mp_m\ g$.

4. Square-Free Polynomials in GF(p)

In Algorithm 1, step 4 mentions the selection of a *suitable* prime p. To be more precise, there are two conditions that have to be satisfied. First, p must be coprime to the leading coefficient of the input polynomial f. The other condition stems from Berlekamp's algorithm, namely f must be square-free in GF(p).

Whereas selecting a prime that satisfies the first condition is in principle easy – any prime larger than the leading coefficient will do – it is actually not so easy to formally prove that the second condition is satisfiable. We split the problem of computing a suitable prime into the following steps.

- Prove that if f is square-free, then f and its derivative f' are coprime in GF(p), and f is square-free in GF(p) for every sufficiently large prime p.
- Develop a prime number generator which returns the first prime such that f and f' are coprime in GF(p).

The prime number generator lazily generates all primes and aborts as soon as the first suitable prime is detected. This is easy to model in Isabelle by defining the generator (*suitable_prime_bz*) via **partial_function** [11].

Our formalized proof of the existence of a suitable prime proceeds along the following line. Let f be square-free over $\mathbb{Z}$. Then f is also square-free over $\mathbb{Q}$ using Gauss Lemma. For fields of characteristic 0, f is square-free if and only if f and f' are coprime. Coprimality is the same as demanding that the resultant is non-zero, so we get $\mathsf{res}(f, f') \neq 0$. The advantage of using resultants is that they admit the following property: if p is larger than $\mathsf{res}(f, f')$ and the leading coefficients of f and f', then $\mathsf{res}_p(f, f') \neq 0$, where $\mathsf{res}_p(f, g)$ denotes the resultant of f and g computed in GF(p). Now we go back from resultants to coprimality, and obtain that f and f' are coprime in GF(p). Finally we prove that the coprimality of f and f' ensures square-freeness in arbitrary fields.

Whereas the reasoning above shows that any prime larger than $\mathsf{res}(f, f')$, $\mathsf{lc}(f)$ and $\mathsf{lc}(f')$ is admitted, we still prefer to search for a small prime p since Berlekamp's algorithm has a worst case lower bound of degree$(f) \cdot p$ operations.

EXAMPLE 1. *Consider the polynomial f which will be used as a running example throughout this paper.*

$$f = 4 + 47x - 2x^2 - 23x^3 + 18x^4 + 10x^5$$

Selecting $p = 2$ or $p = 5$ is not admissible since these numbers are not coprime to 10, the leading coefficient of f. Also $p = 3$ is not admissible since the GCD of f and f' is $2 + x$ in GF(3). Finally, $p = 7$ is a valid choice since the GCD of f and f' is 1 in GF(7), and 7 and 10 are coprime.

5. Berlekamp's Algorithm

5.1 Informal Description

Algorithm 2 briefly describes Berlekamp's algorithm [3]. It focuses on the core computations that have to be performed. For a discussion on why these steps are performed we refer to Knuth [9, 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 $B_f \in \text{GF}(p)^{d \times d}$ for f, where the i-th row is the vector of the coefficients of polynomial $x^{p \cdot i}$ mod 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 construct the corresponding polynomial h_i where the entries in b_i are the coefficients of h_i.
5. Set $F := \{f\}$, $H := \{h_1, \ldots, h_r\} \setminus \{1\}$, $F_I := \emptyset$.
6. If $|F| = r \vee H = \emptyset$, return c and $F \cup F_I$.
7. Pick $h \in H$ and update $H := H \setminus \{h\}$. Update $F := \{g_{ij} \mid f_i \in F, 0 \leq j < p, g_{ij} = \gcd(f_i, h - j), g_{ij} \neq 1\}$.
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 by continuing Example 1.

EXAMPLE 2. *In Algorithm 1, step 5, we have to factor f in GF(p) for $p = 7$. To this end, we first simplify f*

$$f \equiv 4 + 5x + 5x^2 + 5x^3 + 4x^4 + 3x^5 \pmod 7$$

before passing it to Berlekamp's algorithm.

Step 1 now divides this polynomial by its leading coefficient $c = 3$ in GF(p) and obtains the new $f := 6 + 4x + 4x^2 + 4x^3 + 6x^4 + x^5$.

Step 2 computes the Berlekamp matrix as

$$B_f = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 4 & 6 & 2 & 4 & 3 \\ 2 & 3 & 6 & 1 & 4 \\ 6 & 3 & 5 & 3 & 1 \\ 1 & 5 & 5 & 6 & 6 \end{pmatrix}$$

since

$$\begin{aligned} x^0 \bmod f &\equiv 1 && \pmod 7 \\ x^7 \bmod f &\equiv 4 + 6x + 2x^2 + 4x^3 + 3x^4 && \pmod 7 \\ x^{14} \bmod f &\equiv 2 + 3x + 6x^2 + x^3 + 4x^4 && \pmod 7 \\ x^{21} \bmod f &\equiv 6 + 3x + 5x^2 + 3x^3 + x^4 && \pmod 7 \\ x^{28} \bmod f &\equiv 1 + 5x + 5x^2 + 6x^3 + 6x^4 && \pmod 7. \end{aligned}$$

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 $(B_f - I)^{\mathrm{T}}$:

$$\begin{pmatrix} 0 & 4 & 2 & 6 & 1 \\ 0 & 5 & 3 & 3 & 5 \\ 0 & 2 & 5 & 5 & 5 \\ 0 & 4 & 1 & 2 & 5 \\ 0 & 3 & 4 & 1 & 5 \end{pmatrix} \hookrightarrow \begin{pmatrix} 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

We determine $r = 2$, and extract the basis vectors $b_1 = (1\ 0\ 0\ 0\ 0)$ and $b_2 = (0\ 5\ 6\ 5\ 1)$. Step 4 converts them into the polynomials $h_1 = 1$ and $h_2 = 5x + 6x^2 + 5x^3 + x^4$, and step 5 initializes $H = \{h_2\}$, $F = \{f\}$, and $F_I = \emptyset$.

The termination condition in step 6 does not hold. So in step 7 we pick $h = h_2$ and compute the required GCDs.

$$\begin{aligned} &\gcd(f, h_2 - 1) = 6 + 5x + 6x^2 + 5x^3 + x^4 =: f_1 \\ &\gcd(f, h_2 - 4) = 1 + x =: f_2 \\ &\gcd(f, h_2 - i) = 1 \qquad \textit{for all } i \in \{0, 2, 3, 5, 6\} \end{aligned}$$

Afterwards, we update F to $\{f_1, f_2\}$ and H 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. Hence, step 8 updates $F_I := \{f_2\}$, $F := \{f_1\}$, and $r := 1$.

Now we go back to step 6, where both termination criteria fire at the same time ($|F| = 1 = r \wedge H = \emptyset$). We return $c \cdot f_1 \cdot f_2$ as final factorization, i.e.,

$$f \equiv 3 \cdot (1 + x) \cdot (6 + 5x + 6x^2 + 5x^3 + x^4) \pmod 7$$

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), the 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 GF(p) is available as a *type* of class *field*: for polynomials we use α *poly* from `~~/src/HOL/Library/Polynomial`, we take `~~/src/HOL/Number_Theory/Euclidean_Algorithm` for GCDs, and we load `$(AFP)/thys/Jordan_Normal_Form/Gauss_Jordan_Elimination` for the Gauss–Jordan elimination.

5.2 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 equations (7,8,9,10,13,14) in the textbook [9, 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* over a prime field GF(p) to derive (7) in the textbook, but we could find this theorem only for *integers* and *naturals*. Since the proofs of the theorem over such domains are quite similar, the adaptation to polynomials was not that difficult. Indeed, we formalized the theorem for *polynomials* over an arbitrary field.

None of the cited equations was straightforward to prove. More concretely, equation (13) was a little bit cumbersome, since it is proven by rewriting summations which are congruent modulo f. It also requires 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 degree$(f) - 1$ and then transform the list into a vector. In addition, proving that $(f + g)^p = f^p + g^p$, where f and g are polynomials over 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 of Algorithm 2, 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

$$W_f := \{h \mid h^p \equiv h \pmod f, \text{degree}(h) < \text{degree}(f)\}$$

and then we show that such an isomorphism transforms the basis $b_1, \ldots, b_r$ into a basis $H_b := \{h_1, \ldots, h_r\}$ of W_f. This means that H_b is a Berlekamp basis for f and then every factorization of f has at most $|H_b|$ factors. This proof also requires some extra effort since it was necessary to define another isomorphism between the vector spaces W_f and 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 [13] about vector spaces to include some necessary results which relate linear maps, isomorphisms between vector spaces, dimensions, and bases.

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

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

Finally, it follows that every non-constant reducible divisor f_i of f can be properly factored via $\gcd(f_i, h - j)$ for suitable $h \in H_b$ 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 equations. Here, H_{old} represents the set of already processed polynomials of H_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. $H_b = H \cup H_{old}$.
5. $\gcd(f_i, h - j) \in \{1, f_i\}$ for all $h \in H_{old}$, $0 \le j < p$ and $f_i \in F \cup F_I$.
6. $|F_I| + r = |H_b|$.

It is easy to see that all invariants are initially established in step 5 by picking $H_{old} = \{1\} \cap H_b$. In particular, invariant 5 is satisfied since the GCD of the monic polynomial f and a constant polynomial c is always 1 (if $c \neq 0$) or f (if $c = 0$).

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. Invariant 1 follows from (14).

The 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 $|H_b| = |F \cup F_I|$, i.e., $F \cup F_I$ is a factorization of f with the maximum number of factors, and thus every factor is irreducible. In the other case, $H = \emptyset$ and hence $H_{old} = H_b$ by invariant 4. Combining this with invariant 5 shows that every element f_i in $F \cup F_I$ cannot be factored by $\gcd(f_i, h - j)$ for any $h \in H_b$ and $0 \le j < p$. Since H_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. Here, *mset* converts a list into a multiset, and *unique_factorization* f demands that the given factorization is the unique factorization of f.

THEOREM 2 (Berlekamp's Algorithm).

assumes *square_free* $(f :: \alpha\ GFp\ poly)$
and *berlekamp_factorization* $f = (c, fs)$
shows *unique_factorization* f $(c, mset\ fs)$

In order to prove the validity of the output factorization, we basically use the invariants mentioned before. However, it still requires some tedious reasoning.

Uniqueness follows from the general theorem that the polynomials over fields form a unique factorization domain.

In the proofs, most of the time we model products of polynomials ($\prod fs$) via *prod_list* (the product of the elements of a list) instead of using *prod* (the product of the elements of a set). The reason is that *prod_list* has nicer properties. For instance *prod_list* $(f \# fs) = f \cdot$ *prod_list* fs always holds (here $\#$ is the Isabelle's syntax for the list constructor), whereas *prod* $(insert\ f\ F) = f \cdot prod\ F$ is ensured only if $f \notin F$ and F is a finite set. An alternative to *prod_list* might be products over multisets. However this will require many conversions from lists to multisets since our algorithms work on lists.

5.3 Implementing Berlekamp's Algorithm

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

$$\alpha\ GFp\ poly \Rightarrow \alpha\ GFp \times \alpha\ GFp\ poly\ list.$$

In our use case, recall that Algorithm 1 first computes a prime number p, and then invokes Berlekamp's algorithm on GF(p). This requires Algorithm 1 to construct a new type P with $CARD(P) = p$ 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

$$int \Rightarrow int\ poly \Rightarrow int \times int\ poly\ list$$

where the first argument is the dynamically chosen prime p.

As a first step to solve this problem we define conversions *to_int* $:: \alpha\ GFp \Rightarrow int$ and *of_int* $:: int \Rightarrow \alpha\ GFp$ between α *GFp* and *int* where the former is injective, and the other one applies one "modulo $CARD(\alpha)$" 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. Now only the invocation of Berlekamp's algorithm requires α *GFp*. In the lemma, $unique\_factorization_p$ and $square\_free_p$ denote a unique factorization and a square-free polynomial modulo p respectively.

LEMMA 1.

assumes $(g :: \alpha\ GFp) = of\_int\_poly\ f$
and $square\_free_p\ f$
and *berlekamp_factorization* $g = (d, gs)$

and $c = \mathit{to\_int}\ d$
and $fs = \mathit{map}\ \mathit{to\_int\_poly}\ gs$
and $p = \mathit{CARD}(\alpha)$
shows $\mathit{unique\_factorization}_p\ f\ (c, \mathit{mset}\ fs)$

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

Instead, we made copies of the algorithms, but instead of using the type-based arithmetic operations, we use a record *ops* as parameter that stores all the required arithmetic operations *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 *plus ops* x y (*plus* is the record selector), and pass *ops* as an additional argument to A', the record-based version of A.

The soundness of the record-based algorithms is then mainly proved with the help of the transfer package. We define relations which express that certain elements are representatives of each other. For instance, for integers and GF(p) we define $\mathit{GFp}_{rel} :: \mathit{int} \Rightarrow \alpha\ \mathit{GFp} \Rightarrow \mathit{bool}$ as $\mathit{GFp}_{rel}\ x\ y = (x = \mathit{to\_int}\ y)$. Similar relations are then constructed for polynomials and matrices, e.g., $\mathit{poly}_{rel}\ R\ xs\ f = (\mathit{list\_all2}\ R\ xs\ (\mathit{coeffs}\ f))$ relates lists with polynomials, where $\mathit{list\_all2}\ R\ xs\ ys$ demands that xs and ys are of equal lengths and their 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* (cf. [7]) of the following form.

$(\mathit{GFp}_{rel}\ \mathit{===>}\ \mathit{GFp}_{rel}\ \mathit{===>}\ \mathit{GFp}_{rel})\ (\mathit{op} +'_{ops})\ (\mathit{op} +)$
$(\mathit{GFp}_{rel}\ \mathit{===>}\ \mathit{GFp}_{rel})\ \mathit{inverse}'_{ops}\ \mathit{inverse}$
$(\mathit{GFp}_{rel}\ \mathit{===>}\ \mathit{GFp}_{rel}\ \mathit{===>}\ \mathit{op} =)\ (\mathit{op}\ =)\ (\mathit{op}\ =)$

For instance, the first rule states that if the arguments x and y are related to arguments $\overline{x}$ and $\overline{y}$, resp., then $x+y$ is related to $\overline{x} +'_{ops} \overline{y}$. Note that 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 the soundness of the implementation for Berlekamp's algorithm.

THEOREM 3 (Implementation of Berlekamp).

$(\mathit{poly}_{rel}\ \mathit{GFp}_{rel}\ \mathit{===>}\ \mathit{GFp}_{rel} \times_{rel} \mathit{list\_all2}\ (\mathit{poly}_{rel}\ \mathit{GFp}_{rel}))$
$(\mathit{berlekamp\_factorization}'\mathit{ops})\ \mathit{berlekamp\_factorization}$

The theorem states that if the input f represents some GF(p) polynomial g, and $\mathit{berlekamp\_factorization}'\ \mathit{ops}\ f = (c, fs)$ and $\mathit{berlekamp\_factorization}\ g = (d, gs)$, then c represents d and fs represents gs.

To obtain Theorem 3 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 helpful to see why certain transfer rules could initially not be proved automatically; these commands nicely pointed to missing transfer rules.

Most of the transfer rules for non-recursive algorithms were proved mainly 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 locally declare transfer rules (obtained from the induction hypothesis), unfold one function application iteration, and then finish the proof by *transfer_prover*.

Still, 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, $(\mathit{list\_all2}\ R\ \mathit{===>}\ R)\ \mathit{hd}\ \mathit{hd}$; when the lists are empty, we have to relate $\mathit{undefined} :: \alpha$ with $\mathit{undefined} :: \beta$. To circumvent this problem, we had to reprove invariants that *hd* is invoked only on non-empty lists.

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

In summary, the development of the separate implementation is some annoying overhead, but still a workable solution. In numbers: Theorem 2 requires around 3000 lines of difficult proofs whereas Theorem 3 demands around 600 lines of easy proofs.

Using Theorem 3 we can now reformulate the soundness of Berlekamp's algorithm (Lemma 1) as follows. Here, $\mathit{ff\_ops}\ p$ 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 2.

assumes $g = \mathit{coeffs}\ (\mathit{Mp}_p\ f)$
and $\mathit{square\_free}_p\ f$
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 $\mathit{unique\_factorization}_p\ f\ (c, \mathit{mset}\ fs)$

Note that in Lemma 2 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, a recent addition to the Isabelle distribution. It replaces the condition $p = \mathit{CARD}(\alpha :: \mathit{prime\_card})$ by $\mathit{prime}\ p$. Thus we can define a function $\mathit{berlekamp\_factorization\_int} :: \mathit{int} \Rightarrow \mathit{int\ poly} \Rightarrow \mathit{int} \times \mathit{int\ poly\ list}$ and prove its soundness without having to create a type $\alpha\ \mathit{GFp}$.

THEOREM 4 (Berlekamp Factorization on Integers).

assumes *berlekamp_factorization_int* $p\ f = (c, fs)$
and $\mathit{square\_free}_p\ f$
and *prime* p
shows $\mathit{unique\_factorization}_p\ f\ (c, \mathit{mset}\ fs)$

6. Mignotte's Factor Bound

Reconstructing the polynomials proceeds by obtaining factors modulo p^k. The value of k should be large enough, so that any coefficient of any factor of the original polynomial can be determined from the corresponding coefficients in $\mathbb{Z}/p^k\mathbb{Z}$. We can find such k by finding a bound on the coefficients of the factors of f, i.e., a function *factor_bound* such that the following statement holds:

assumes $f \neq 0$
and $g \cdot h = f$
and $\mathit{degree}(g) \leq d$
shows $|\mathit{coeff}\ g\ j| \leq \mathit{factor\_bound}\ f\ d$

Clearly, if b is a bound on the absolute value of the coefficients, and $p^k > 2 \cdot b$ then we can encode all required coefficients: In $\mathbb{Z}/p^k\mathbb{Z}$ we can represent the numbers $\{-\lfloor \frac{p^k-1}{2} \rfloor, \ldots, \lceil \frac{p^k-1}{2} \rceil\} \supseteq \{-b, \ldots, b\}$.

The *Mignotte bound* [15] provides a bound on the absolute values of the coefficients. The Mignotte bound is obtained by relating the *Mahler measure* of a polynomial to its coefficients. The Mahler measure is defined as follows:

$$\mathit{measure}\ f = |\mathsf{lc}(f)| \cdot \prod_{i=1}^{n} \max\{1, |r_i|\}$$

where $n = \mathrm{degree}(f)$ and $r_1, \ldots, r_n$ are the complex roots of f taking multiplicity into account. For nonzero f, $\mathsf{lc}(f)$ is a nonzero integer. It follows that $\mathit{measure}\ f \geq 1$. The definition of *measure* shows that $\mathit{measure}\ (g \cdot h) = \mathit{measure}\ g \cdot \mathit{measure}\ h$. We conclude that $\mathit{measure}\ g \leq \mathit{measure}\ f$ if g is a factor of f.

The Mahler measure is bounded by the coefficients from above through Landau's inequality:

$$\mathit{measure}\ f \leq \sqrt{\textstyle\sum_{i=1}^{n} (\mathit{coeff}\ f\ i)^2}$$

Mignotte showed that the coefficients also bound the measure from below: $|\mathit{coeff}\ g\ i| \leq \binom{d}{i} \cdot \mathit{measure}\ g$ whenever $\mathrm{degree}(g) \leq d$. Putting this together we get:

$$\begin{aligned} |\mathit{coeff}\ g\ j| &\leq \binom{d}{j} \cdot \mathit{measure}\ g \\ &\leq \binom{d}{\lfloor d/2 \rfloor} \cdot \mathit{measure}\ f \\ &\leq \binom{d}{\lfloor d/2 \rfloor} \cdot \sqrt{\textstyle\sum_i (\mathit{coeff}\ f\ i)^2} \\ &= \sqrt{\binom{d}{\lfloor d/2 \rfloor}^2 \cdot \textstyle\sum_i (\mathit{coeff}\ f\ i)^2} \end{aligned}$$

Consequently, we define *factor_bound* as follows:

$$\mathit{factor\_bound}\ f\ d = \lfloor \sqrt{\tbinom{d}{\lfloor d/2 \rfloor}^2 \cdot \textstyle\sum_i (\mathit{coeff}\ f\ i)^2} \rfloor$$

It remains to choose a bound on the degrees of factors of f that we require for reconstruction. A simple choice is $d = \mathrm{degree}(f)$, but we can do slightly better. After having computed the Berlekamp factorization, we know the degrees of the factors of f in GF(p). Since the degrees will not be changed by the Hensel lifting, we also know the degrees of the polynomials h_i in step 7 of Algorithm 1.

Since in step 8 of Algorithm 1 we will combine at most half of the factors, it suffices to take $d = \sum_{i=\lfloor \frac{m}{2} \rfloor}^{m} \mathrm{degree}(h_i)$, where we assume that the sequence $h_1, \ldots, h_m$ is sorted by degree, starting with the smallest. In the formalization this gives rise to the following definition:

$$\begin{aligned} \mathit{degree\_bound}\ hs = (&\textbf{let}\ ds = \mathit{sort}\ (\mathit{map}\ \mathrm{degree}\ hs) \\ &\textbf{in}\ \mathit{sum\_list}\ (\mathit{drop}\ (\mathit{length}\ ds\ \mathit{div}\ 2)\ ds)) \end{aligned}$$

Note also that in the reconstruction step we actually compute factors of $\mathsf{lc}(f) \cdot f$. Thus, we have to multiply the factor bound for f by $|\mathsf{lc}(f)|$.

EXAMPLE 3. *At the end of Example 2 we have the factorization* $f = 4 + 47x - 2x^2 - 23x^3 + 18x^4 + 10x^5 \equiv 3 \cdot (1 + x) \cdot (6 + 5x + 6x^2 + 5x^3 + x^4) \pmod 7$.

We compute $d = \mathit{degree}(6 + 5x + 6x^2 + 5x^3 + x^4) = 4$. *Hence, we have to be able to represent coefficients of at most* $10 \cdot \lfloor \sqrt{\binom{4}{2}^2 \cdot (4^2 + 47^2 + 2^2 + 23^2 + 18^2 + 10^2)} \rfloor = 3380$, *i.e., the numbers* $\{-3380, \ldots, 3380\}$. *Thus the modulus has to be larger than* $2 \cdot 3380 = 6760$. *Hence, in step 6 of Algorithm 1 we choose* $k = 5$, *since this is the least number* k *such that* $p^k = 7^k > 6760$.

Finally, we report that our previous oracle implementation had a flaw in the computation of a suitable degree bound d, since it just defined d to be the half of the degree of f. This choice might be insufficient:[4] Consider the list of degree of the h_i to be $[1, 1, 1, 1, 1, 5]$. Then the product $h_1 \cdot h_6$ of degree 6 might be a factor of f, but the degree bound in the old implementation was computed as $\frac{1+1+1+1+1+5}{2} = 5$, excluding this product. This wrong choice of d was detected only after starting to formalize the required degree bound.

7. Hensel Lifting

Given a factorization in GF(p):

$$f \equiv \mathsf{lc}(f) \cdot g_1 \cdot \ldots \cdot g_m \pmod p$$

[4] Indeed, one can reduce the degree bound to half of the degree of f if one uses a slightly more complex reconstruction algorithm which sometimes considers the complement of the selected factors. We did not investigate the trade-off between the two alternatives.

which Berlekamp's algorithm provides, the task of the Hensel lifting is to compute a factorization

$$f \equiv \mathsf{lc}(f) \cdot h_1 \cdot \ldots \cdot h_m \pmod{p^k}.$$

Hensel's lemma, following Miola and Yun [16], is stated as follows.

LEMMA 3 (Hensel). *Consider polynomials f over $\mathbb{Z}$, g_1 and h_1 over GF(p) for a prime p, such that g_1 is monic and $f \equiv g_1 \cdot h_1 \pmod{p}$. For any $k \geq 1$, there exist polynomials g_k and h_k over $\mathbb{Z}/p^k\mathbb{Z}$ such that g_k is monic, $f \equiv g_k \cdot h_k \pmod{p^k}$, $g_k \equiv g_1 \pmod{p}$, $h_k \equiv h_1 \pmod{p}$. Moreover, if f is monic, then g_k and h_k are unique (mod p^k).*

The lemma is proved inductively on k where there is a one step lifting from $\mathbb{Z}/p^k\mathbb{Z}$ to $\mathbb{Z}/p^{k+1}\mathbb{Z}$. To be more precise, the one step lifting assumes polynomials g_k and h_k over $\mathbb{Z}/p^k\mathbb{Z}$ satisfying the conditions, and computes the desired g_{k+1} and h_{k+1} over $\mathbb{Z}/p^{k+1}\mathbb{Z}$.

As explained in Section 3, it is preferable to carry on the proof in the type-based setting whenever possible, and indeed we proved the one step lifting in this way.

LEMMA 4 (Hensel lifting – one step).

assumes $\mathit{CARD}(\alpha) = \mathit{CARD}(\beta :: \mathit{prime\_card}) * \mathit{CARD}(\gamma)$
  and $\mathit{CARD}(\beta)$ *dvd* $\mathit{CARD}(\gamma)$
  and $\#f = g * h$
  and *monic* g
  and *degree* $f =$ *degree* $g +$ *degree* h
  and *hensel_1* $\mathit{TYPE}(\beta)$ f g $h = (\overline{g}, \overline{h})$
shows $f = \overline{g} * \overline{h} \wedge$ *monic* $\overline{g} \wedge g = \#\overline{g} \wedge h = \#\overline{h} \wedge$
    degree $g =$ *degree* $\overline{g} \wedge$ *degree* $h =$ *degree* $\overline{h}$
  and ... `(* uniqueness statement *)`

Here, $\mathit{CARD}(\alpha)$ represents p^{k+1}, $\mathit{CARD}(\beta)$ represents p, and $\mathit{CARD}(\gamma)$ represents p^k. The prefix "#" denotes the function that converts polynomials over integer modulo m into those over integer modulo n, where the type inference determines n.

Unfortunately, we could not see how to use Lemma 4 in the inductive proof of Lemma 3 in a type-based setting. A type-based statement of Lemma 3 would have an assumption like $\mathit{CARD}(\alpha) = p^k$. Then the induction hypothesis would look like

$$\mathit{CARD}(\alpha) = p^k \Longrightarrow \ldots \tag{3}$$

and the goal statement would be $\mathit{CARD}(\alpha) = p^{k+1} \Longrightarrow \ldots$. There is no hope to be able to apply the induction hypothesis (3) for this goal, since the assumptions are clearly incompatible. A solution to this problem seems to require extending the induction scheme to admit changing the type variables, and produce an induction hypothesis like $\mathit{CARD}(?\alpha) = p^k \Longrightarrow \ldots$ where $?\alpha$ can be instantiated. Unfortunately this is not possible in Isabelle/*HOL*.

In our development, we therefore formalized most of the reasoning for Hensel's lemma on *integer* polynomials in the locale-based setting (cf. Section 3.2), so that the modulus (the k in the p^k) can be easily altered within algorithms and inductive proofs. Working on integer polynomials also has the advantage when formalizing both the Hensel lifting, which is presented below, and the reconstruction phase, which is presented in the next section, since one has to perform operations of integer polynomials both in $\mathbb{Z}$ and in $\mathbb{Z}/p^k\mathbb{Z}$, so there is no conversion required.

In the locale *poly_mod*, the binary version of Hensel's lemma is proved as follows, and internally one step of the Hensel lifting is applied over and over again, i.e., the exponents are p, p^2, p^3, p^4, ... [16, Sect. 2.2]. In the statement, Isabelle's syntax $\exists!$ represents the unique existential quantification.

LEMMA 5 (Hensel lifting – multiple steps, binary).

assumes *prime* p **and** $\mathit{coprime}_p\ g\ h$ **and** $f \equiv_p g \cdot h$
  and $\mathit{Mp}_p\ g = g$ **and** $\mathit{Mp}_p\ h = h$
  and *monic* g **and** $k \neq 0$
shows $\exists!\ (\overline{g}, \overline{h}).\ f \equiv_{p^k} \overline{g} \cdot \overline{h} \wedge$
  monic $\overline{g} \wedge g \equiv_p \overline{g} \wedge h \equiv_p \overline{h} \wedge$
  $\mathit{Mp}_{p^k}\ \overline{g} = g \wedge \mathit{Mp}_{p^k}\ \overline{h} = h$

We also formalize the *quadratic Hensel lifting*, where the modulus during the lifting will be p, p^2, p^4, p^8, ..., p^{2^ℓ} where, ℓ is a suitable exponent such that $2^\ell \geq k$, cf. [16, Sect. 2.3]. Since 2^ℓ might be larger than k, we finally perform a Mp_{p^k}-operation in order to convert the $\mathbb{Z}/p^{2^\ell}\mathbb{Z}$ factorization into a $\mathbb{Z}/p^k\mathbb{Z}$ factorization. The quadratic version is supported in addition to the linear version, since in our experiments the quadratic algorithm is more efficient than the linear one in contrast to the result of Miola and Yun [16, Sect. 1].[5][6] Hence, the soundness lemma for the quadratic Hensel lifting does not only mention the existence of $\overline{f}$ and $\overline{g}$, but it proves that the algorithm computes these polynomials.

We further extend the binary (quadratic) lifting algorithm to an executable n-ary lifting algorithm: *hensel_lifting*.

LEMMA 6 (Hensel Lifting – general case).

assumes *hensel_lifting* $p\ k\ f\ fs = gs$
  and $k \neq 0$ **and** *prime* p **and** *coprime* $(\mathsf{lc}(f))\ p$
  and $\mathit{square\_free}_p\ f$ **and** $\mathit{factorization}_p\ f\ (c, \mathit{mset}\ fs)$
  and $c \in \{0.. < p\}$
  and $\forall f_i \in \mathit{set}\ fs.\ \mathit{set}\ (\mathit{coeffs}\ f_i) \subseteq \{0.. < p\}$
shows $\mathit{unique\_factorization}_{p^k}\ f\ (\mathsf{lc}(f), \mathit{mset}\ gs)$
  and $\forall g_i \in \mathit{set}\ gs.\ \mathit{monic}\ g_i \wedge \mathit{irreducible}_p\ g_i$

[5] Perhaps our quadratic version of Hensel lifting is faster than the linear version since we did not integrate (and prove) optimization (iv) of Miola and Yun [16, Sect. 2.4].

[6] The linear version of the Hensel lifting is still present, as it admits an easier proof of the uniqueness result.

Note that uniqueness follows from the fact that the preconditions already imply that f is *uniquely* factored in $\mathbb{Z}/p\mathbb{Z}$ – just apply Theorem 4.

We do not go into details of the proofs, but briefly mention that also here local type definitions have been essential. The reason is that the computation relies upon the extended Euclidean algorithm applied on polynomials over GF(p). Since the soundness of this algorithm is available only in a type-based version in the Isabelle distribution, we first convert it to the integer representation of GF(p) via local type definitions in a similar way as was explained in Section 5.3.

We end this section by proceeding with the running example, without providing details of the computation.

EXAMPLE 4. *Applying the Hensel lifting on the factorization of Example 2 with $k = 5$ from Example 3 yields*

$$f \equiv_{p^k} 3 \cdot (2885 + x) \cdot (14027 + 7999x + 13691x^2 + 7201x^3 + x^4)$$

8. Reconstructing True Factors

For formalizing step 8 of Algorithm 1, we basically follow Knuth, who described the reconstruction algorithm briefly and presented the soundness proof in prose [9, steps F2 and F3, pages 451 and 452]. At this point of the formalization the De Bruijn factor is quite large, i.e., the formalization is by far more detailed than the intuitive description given by Knuth.

The following definition presents (a simplified version of) the main worklist algorithm, which is formalized in Isabelle/HOL via the **partial_function** command.[7]

```
reconstruction f d rf hs res [] =
  let d̄ = d + 1
  in if rf < 2d̄ then f # res
     else reconstruction f d̄ rf hs res (sublists hs d̄)
reconstruction f d rf hs res (gs # todo) =
  let g = inv_{p^k} (Mp_{p^k} (lc(f) · prod_list gs)) in
  if ¬ g dvd lc(f) · f
  then reconstruction f d rf hs res todo
  else let
     f_i = primitive_part g;
     f̄ = f div f_i;
     r̄f̄ = rf − length gs;
     r̄ē̄s̄ = f_i # res
     in if r̄f̄ < 2d then f̄ # r̄ē̄s̄ else let
        h̄s̄ = fold remove1 gs hs;
        t̄ō̄d̄ō̄ = sublists h̄s̄ d
        in reconstruction f̄ d r̄f̄ h̄s̄ r̄ē̄s̄ t̄ō̄d̄ō̄
```

[7] Although **partial_function** does not support pattern matching, we prefer to use pattern matching in the presentation.

Here, rf is supposed to be the number of remaining factors, i.e., the length of hs; *sublists* hs d denotes the list of length-d sublists of hs; and inv_m is the inverse modulo function, which converts a polynomial with coefficients in $\{0, \ldots, m\}$ into a polynomial with coefficients in $\{-\lfloor \frac{m-1}{2} \rfloor, \ldots, \lceil \frac{m-1}{2} \rceil\}$, where the latter set is a superset of the range of coefficients of any potential factor of $\mathsf{lc}(f) \cdot f$, cf. Section 6.

Basically, for every sublist gs of hs we try to divide $\mathsf{lc}(f) \cdot f$ by the reconstructed potential factor g. If this is possible then we store f_i, the content-free version of g, in the list res of resulting integer polynomial factors and update the polynomial f and its factorization hs in $\mathbb{Z}/p^k\mathbb{Z}$ accordingly. When the worklist becomes empty or a factor is found, we update the number rf of remaining factors hs and the length d of the sublists we are interested in. Finally, when we have tested enough sublists ($rf < 2d$) we finish.

For efficiency, the actual formalization employs three improvements over the simplified version presented here.

- Values which are not frequently changed are passed as additional arguments. For instance $\mathsf{lc}(f) \cdot f$ is provided via an additional argument and not recomputed in every invocation of *reconstruction*.
- For the divisibility test we first test whether the constant term *coeff* g 0 of the candidate factor g divides that of $\mathsf{lc}(f) \cdot f$. In our experiments, in over 99 % of the cases this simple integer divisibility test can prove that g is not a factor of $\mathsf{lc}(f) \cdot f$. Moreover, this test is done before computing the polynomial g, which is a product of polynomials in gs, since the constant term of g is the product of those in gs.
- In the formalization, the enumeration of sublists is made parametric, and we developed an efficient generator of sublists which reuses results from previous iterations. Moreover, the sublist generator also shares computations to generate the constant term of g.

EXAMPLE 5. *Continuing Example 4, we have only two factors, so it suffices to consider $d = 1$. We obtain the singleton sublists $[g_1] = [2885 + x]$ and $[g_2] = [14027 + 7999x + 13691x^2 + 7201x^3 + x^4]$. The constant term of $inv_{p^k}(\mathsf{lc}(f) \cdot g_1)$ is the inverse modulo of $(10 \cdot 2885) \bmod p^k$, i.e., -4764, and similarly, for g_2 we obtain 5814. Since neither of them divides 40, the constant term of $\mathsf{lc}(f) \cdot f$, the algorithm returns $[f]$, i.e., f is irreducible.*

The formalized soundness proof of *reconstruction* is much more involved than the paper proof; it is proved inductively with several invariants that have to be maintained throughout the proof, for instance

- correct input: $rf = \mathit{length}\ hs$
- corner cases: $2d \leq rf$, $todo \neq [] \longrightarrow d < rf$, $d = 0 \longrightarrow todo = []$

- irreducible result: $\forall f_i \in$ *set* res. *irreducible* f_i
- properties of prime: *square_free*$_p$ f, *coprime* $(\mathsf{lc}(f))$ p
- factorization mod p^k: *unique_factorization*$_{p^k}$ f $(\mathsf{lc}(f), hs)$
- normalized input: $h_i \bmod p^k = h_i$ for all $h_i \in$ *set* hs
- factorization over integers: the polynomial $f \cdot \prod res$ stays constant throughout the algorithm
- all factors of $\mathsf{lc}(f) \cdot f$ with degree at most *degree_bound* hs have coefficients in the range $\{-\lfloor \frac{p^k-1}{2} \rfloor, \ldots, \lceil \frac{p^k-1}{2} \rceil\}$
- all non-empty sublists gs of hs of length at most d which are not present in $todo$ have already been tested, i.e., these gs do not give rise to a factor of f

The hardest parts in the proofs were to ensure the validity of all invariants after a factor g has been detected – since then nearly all parameters are changed – and to ensure that the final polynomial f is irreducible when the algorithm terminates.

In total, we achieve the following soundness result, which already integrates many of the results from the previous sections. Here, *berlekamp_hensel* is a simple composition of the Berlekamp factorization and the Hensel lifting, and *zassenhaus_reconstruction* invokes *reconstruction* with the right set of starting parameters.

THEOREM 5 (Zassenhaus Reconstruction of Factors).

assumes *prime* p
  and *coprime* $(\mathsf{lc}(f))$ p
  and *square_free*$_p$ f
  and *berlekamp_hensel* p k $f = hs$
  and $d =$ *degree_bound* hs
  and $2 \cdot |\mathsf{lc}(f)| \cdot$ *factor_bound* f $d < p^k$
  and *zassenhaus_reconstruction* hs p k $f = fs$
shows $f =$ *prod_list* fs
  and $\forall f_i \in$ *set* fs. *irreducible* f_i

9. Assembled Factorization Algorithm

At this point, it is straightforward to combine all the previous algorithms to get a factorization algorithm for square-free polynomials which satisfies Theorem 1.

berlekamp_zassenhaus_factorization $f =$ **let**
  $p =$ *suitable_prime_bz* f;
  $(\_, gs) =$ *berlekamp_factorization_int* p f;
  $d =$ *degree_bound* gs;
  $bnd = 2 \cdot |\mathsf{lc}(f)| \cdot$ *factor_bound* f d;
  $k =$ *find_exponent* p bnd;
  $hs =$ *hensel_lifting* p k f fs
  in *zassenhaus_reconstruction* hs p k f

Here, *find_exponent* p bnd just computes an exponent k such that $p^k > bnd$.

Since Theorem 1 has the prerequisite that the input polynomial is square-free, we need to combine this algorithm with a square-free factorization to obtain a full factorization algorithm which takes arbitrary polynomials as input. Here, we base our work on the formalization [19, Sect. 8] of Yun's square-free factorization algorithm [21] for polynomials over fields of characteristic 0. We prove that the Yun factorization also works for integer polynomials. One just has to adapt certain normalization operations in the algorithm from field polynomials to integer polynomials. For instance, instead of dividing the input field polynomial by its *leading coefficient* to obtain a *monic* field polynomial, we now divide the input integer polynomial by its *content* to obtain a *content-free* integer polynomial.

To obtain the soundness of the modified Yun factorization for $\mathbb{Z}$, we did *not* modify the existing formalization. Instead we show that all polynomials $f_{\mathbb{Z}}$ and $f_{\mathbb{Q}}$ that are constructed during the execution of Yun's algorithm on $\mathbb{Z}$ and on $\mathbb{Q}$ on the same input are related, i.e., there exists a constant c such that $c \cdot f_{\mathbb{Z}} = f_{\mathbb{Q}}$. We then connect this relationship with the existing soundness statement for rational polynomials to obtain the soundness theorem of Yun's algorithm on integer polynomials.

THEOREM 6 (Yun Factorization for Integer Polynomials).

assumes *yun_factorization_int* $f = (c, gis)$
shows *square_free_factorization* f (c, gis)
  and $\forall (g, i) \in$ *set* gis. *content* $g = 1 \wedge \mathsf{lc}(g) > 0$

Here, *square_free_factorization* f (c, gis) demands that $f = c \cdot \prod_{(g_i,i) \in \text{set } gis} g_i^{i+1}$, that gis contains no duplicates, that each g_i is square-free, and that g_i and g_j are coprime whenever $i \neq j$.

We finally assemble a factorization algorithm for integer polynomials

factorize_int_poly $f =$ **let**
  $(c, gis) =$ *yun_factorization_int* f;
  $bz =$ *berlekamp_zassenhaus_factorization*;
  in $(c, [\,(h, i).\ (g, i) \leftarrow gis,\ h \leftarrow bz\ g\,])$

and prove its soundness: a factorization into irreducible and square-free factors.[8]

THEOREM 7 (Factorization of Integer Polynomials).

assumes *factorize_int_poly* $f = (c, his)$
shows *square_free_factorization* f (c, his)
  and $\forall (h, i) \in$ *set* his. *irreducible* h

By using the Gauss lemma we also assembled a factorization algorithm for rational polynomials which just converts the input polynomial into an integer polynomial by a scalar

[8] A factorization into irreducible factors is not necessarily a square-free factorization. For instance $(x + 1) \cdot (-x - 1)$ is a factorization into irreducible factors, but it is not square-free. Our factorization algorithm produces the correct result: $-1 \cdot (x + 1)^2$.

multiplication and then invokes *factorize_int_poly*. The algorithm has exactly the same soundness statement as Theorem 7 except that the type changes from integer polynomials to rational polynomials.

10. Experimental Evaluation

We evaluate the performance of our algorithm in comparison to a modern factorization algorithm – here we choose the factorization algorithm of Mathematica 11. To evaluate the runtime of our algorithm, we use Isabelle's code extraction mechanism to extract Haskell code for *factorize_int_poly*. This code was compiled with GHC using the `O2` switch to turn on most optimizations. All experiments have been conducted under macOS Sierra 10.12 on a 6-core Intel Xeon E5 running at 3.5 Ghz.

Figure 1 shows the runtimes of our implementation compared to that of Mathematica on a logarithmic scale. The runtimes are given in seconds (including the 0.4 seconds startup time of Mathematica), and the horizontal axis shows the number of coefficients of the polynomial. The coefficients are chosen at random between -100 and 100.

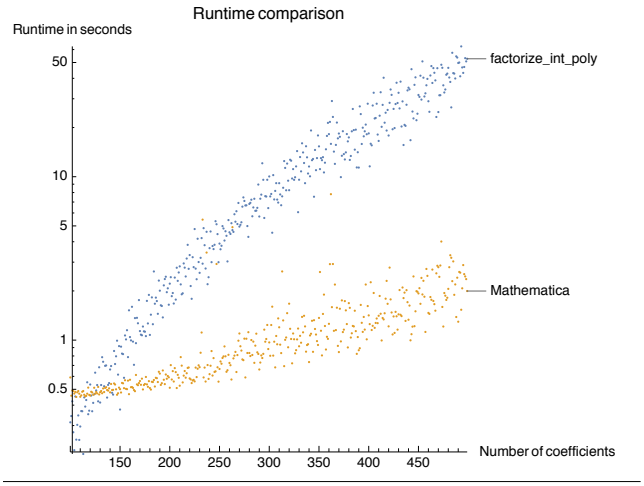

Figure 1. Runtimes compared with Mathematica

As these polynomials have been randomly generated, they are typically irreducible. In this case using a fast external factorization algorithm as a pre-processing step will currently not improve the performance, as then the preprocessing does not modify the polynomial. We conjecture that the situation could be alleviated by further incorporating an efficient irreducibility test.

Profiling revealed that for the (random) example polynomials, most of the time is spent in the Berlekamp factorization, i.e., in step 5 of Algorithm 1. Interestingly, the exponential reconstruction algorithm in step 8 does not have any significance on these random polynomials, cf. Table 1.

Nevertheless we remark that this situation can dramatically change on non-random polynomials, e.g., on polynomials from experiments with algebraic numbers. For instance, more than 98 % of the factorization time is spent in the reconstruction algorithm when computing the minimal integer polynomial that has $\sum_{i=1}^{6} \sqrt[3]{i}$ as root, and there is a timeout because of the reconstruction algorithm, if $\sum_{i=1}^{7} \sqrt[3]{i}$ is considered. As a possible optimisation, the exponential reconstruction phase can be replaced by a polynomial-time lattice-reduction algorithm [6]. Then, of course, a soundness proof would become much more involved.

Table 1. Profiling Results

step	amount of total runtime
Berlekamp factorization	67.4 %
Hensel lifting	22.8 %
Yun factorization	9.3 %
Remaining parts	0.5 %

11. Summary

We formalized the Berlekamp–Zassenhaus algorithm for factoring univariate integer polynomials. To this end we switched between different representations of finite fields and quotient rings with the help of the transfer package and local type definitions. The generated code can factor large polynomials within seconds. The whole formalization consists of about 14,000 lines of Isabelle and took about 11 person months of Isabelle experts. As far as we know, this is the first formalization of an efficient polynomial factorization algorithm in a theorem prover.

There remain numerous possibilities to extend the current formalization for optimizing the factorization algorithm even further: for instance, one can consider implementing the finite field operations on native machine integers when the prime is sufficiently small, deriving a tighter factor bound, improving Isabelle's polynomial multiplication algorithm (which is $\mathcal{O}(n^2)$), improving the GCD algorithm for integer polynomials, incorporating distinct degree factorization, the Cantor–Zassenhaus algorithm, reversed polynomials, lattice-reductions algorithms, ...

Acknowledgments

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. We thank Florian Haftmann for integrating our changes in the polynomial library into the Isabelle distribution; we thank Manuel Eberl for discussions on factorial rings in Isabelle; and we thank the anonymous reviewers for their helpful remarks and suggestions for future work.

References

[1] C. Ballarin. Locales: A module system for mathematical theories. *J. Autom. Reasoning*, 52(2):123–153, 2014.

[2] G. Barthe, B. Grégoire, S. Heraud, F. Olmedo, and S. Z. Béguelin. Verified indifferentiable hashing into elliptic curves. In *POST 2012*, volume 7215 of *LNCS*, pages 209–228, 2012.

[3] E. R. Berlekamp. Factoring polynomials over finite fields. *Bell System Technical Journal*, 46:1853–1859, 1967.

[4] D. G. Cantor and H. Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Math. Comput.*, 36(154): 587–592, 1981.

[5] J. R. Cowles and R. Gamboa. Unique factorization in ACL2: Euclidean domains. In *ACL2 2006*, pages 21–27. ACM, 2006.

[6] M. van Hoeij. Factoring polynomials and the knapsack problem. *J. Number Theory*, 95(2):167–189, 2002.

[7] B. Huffman and O. Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *CPP 2013*, volume 8307 of *LNCS*, pages 131–146, 2013.

[8] B. Kirkels. Irreducibility certificates for polynomials with integer coefficients. Master's thesis, Radboud Universiteit Nijmegen, 2004.

[9] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981. ISBN 0-201-03822-6.

[10] H. Kobayashi, H. Suzuki, and Y. Ono. Formalization of Hensel's lemma. In *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, volume 1, pages 114–118, 2005.

[11] A. Krauss. Recursive definitions of monadic functions. In *PAR 2010*, volume 43 of *EPTCS*, pages 1–13, 2010.

[12] O. Kunčar and A. Popescu. From types to sets by local type definitions in higher-order logic. In *ITP 2016*, volume 9807 of *LNCS*, pages 200–218, 2016.

[13] H. Lee. Vector spaces. *Archive of Formal Proofs*, 2014. URL `http://www.isa-afp.org/entries/VectorSpace.shtml`.

[14] É. Martin-Dorel, G. Hanrot, M. Mayero, and L. Théry. Formally verified certificate checkers for hardest-to-round computation. *J. Autom. Reasoning*, 54(1):1–29, 2015.

[15] M. Mignotte. An inequality about factors of polynomials. *Mathematics of Computation*, 28(128):1153–1157, 1974.

[16] A. Miola and D. Y. Yun. Computational aspects of Hensel-type univariate polynomial greatest common divisor algorithms. *ACM SIGSAM Bulletin*, 8(3):46–54, 1974.

[17] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[18] R. Thiemann and A. Yamada. Algebraic numbers in Isabelle/HOL. In *ITP 2016*, volume 9807 of *LNCS*, pages 391–408, 2016.

[19] R. Thiemann and A. Yamada. Formalizing Jordan normal forms in Isabelle/HOL. In *CPP 2016*, pages 88–99. ACM, 2016.

[20] Wolfram Research, Inc. *Mathematica*. Champaign, Illinois, version 11.0 edition, 2016.

[21] D. Y. Yun. On square-free decomposition algorithms. In *SYMSAC 1976*, pages 26–35, 1976.

[22] H. Zassenhaus. On Hensel factorization, I. *J. Number Theory*, 1(3):291–311, 1969.