Reasoning about Coding Theory: The Benefits We Get from Computer Algebra

Clemens Ballarin and Lawrence C. Paulson

Computer Laboratory, University of Cambridge, Cambridge CB2 3QG, UK
{Clemens.Ballarin, Larry.Paulson}@cl.cam.ac.uk
http://www.cl.cam.ac.uk/users/{cmb33, lcp}

Abstract. The use of computer algebra is usually considered beneficial for mechanised reasoning in mathematical domains. We present a case study, in the application domain of coding theory, that supports this claim: the mechanised proofs depend on non-trivial algorithms from computer algebra and increase the reasoning power of the theorem prover.
The unsoundness of computer algebra systems is a major problem in interfacing them to theorem provers. Our approach to obtaining a sound overall system is not blanket distrust but based on the distinction between algorithms we call sound and *ad hoc* respectively. This distinction is blurred in most computer algebra systems. Our experimental interface therefore uses a computer algebra library. It is based on theorem templates, which provide formal specifications for the algorithms.
Keywords. Computer algebra, mechanised reasoning, combining systems, soundness of computer algebra systems, specialisation problem, coding theory.
AISC topics. Integration of logical reasoning and computer algebra, automated theorem provers.

1 Motivation

Is the use of computer algebra technology beneficial for mechanised reasoning in and about mathematical domains? Usually it is assumed that it is. Many works in this area have, however, either only little reasoning content, or the contribution of symbolic computation is only the simplification of expressions. Exceptions are Analytica [Clarke and Zhao, 1993] and work by [Harrison, 1996]. Both these approaches do not scale up. The former trusts the computer algebra system too much, the latter, too little. Computer algebra systems are not logically sound reasoning systems, but collections of algorithms.

Apart from the verification of numerical hardware and software, linking mechanised reasoning and computer algebra gives insight into the design of logically more expressive frameworks for computer algebra, has applications in educational software and is a step towards the development of mathematical assistants. Among the applications, geometry theorem proving is a prospective candidate. For a survey on this, see [Geddes *et al.*, 1992, section 10.6].

This work presents a case study that requires hard techniques from both sides. The proofs we mechanise require algorithms from computer algebra in

order to be solved effectively. They also rely on the formalisation of natural numbers, sets and lists, which are available in Isabelle, and make heavy use of advanced proof procedures.

The outline of this article is as follows. In section 2 we briefly describe the context of interactive theorem proving and the prover Isabelle. We then present an analysis of the soundness problems in computer algebra and based on this describe the design of an interface. The rest of the paper is devoted to our case study. Section 3 introduces the mathematical background along the lines of its mechanisation in Isabelle. Section 4 is a brief introduction to coding theory and section 5 presents the mechanised proofs. Section 6 reviews important details of the implementation and in section 7 we draw conclusions.

2 Interface between Isabelle and Sumit

The interface we present is between the prover Isabelle and the computer algebra library Sumit. See [Paulson, 1994] and [Bronstein, 1996] respectively.

2.1 Isabelle

Isabelle is a natural deduction-style theorem prover. Proofs are carried out interactively by the user by applying tactics to the proof state and so replacing subgoals by simpler ones until all the subgoals are proved. Isabelle provides tactics that perform single inference steps and also highly automated proof procedures, like the simplifier and a tactic that implements a tableau prover.

Isabelle, like other LCF-style theorem provers, allows the user to program arbitrary tactics, which can implement specialised proof procedures. The design of Isabelle ensures that unsoundness cannot be introduced to the system through these procedures. This is achieved by using an abstract datatype `thm` for theorems. Theorems can only be generated by operations provided by the datatype. These operations implement the primitive inference rules of the logic.

Isabelle also provides an oracle mechanism to interface trusted external provers. An oracle can create a theorem, *i.e.* an object of type `thm`, without proving it through the inference rules. This, of course, weakens the rigour of the LCF-approach, but theorems proved later on can record on which external theorems they depend.

We use Isabelle's object logic HOL, which implements Church's theory of simple types, also known as higher order logic. This is a typed version of the λ-calculus. The logic has the usual connectives $(\wedge, \vee, \longrightarrow, \ldots)$ and quantifiers $(\forall, \exists)$. Currying is used for function application. We write $f\,a\,b$ instead of $f(a, b)$. Equality $=$ on the type `bool` is used to express if-and-only-if. For definitions we use $\equiv$, and $\Longrightarrow$ expresses entailment in a deduction rule. Some definitions require Hilbert's ϵ-operator, which is actually a quantifier: $\epsilon x.P\,x$ denotes the unique value for which the predicate P holds. The notation for formulae in this paper is close to their representation in Isabelle. We have omitted all type information from formulae to improve their legibility. If type information is necessary, we give it informally in the context.

2.2 Soundness in Computer Algebra

Computer algebra systems have been designed as tools that perform complicated algebraic computations. Their soundness or, as some authors might prefer to say, unsoundness has become a focus, see [Harrison, 1996, Homann, 1997] for examples. A systematic presentation of more examples is [Stoutemyer, 1991]. We have identified the following reasons for unsoundness in the design of computer algebra systems:

- They present a misleadingly uniform interface to collections of algorithms. An object, which is used with a particular meaning in one algorithm, can be used with a different meaning in another algorithm. Particularly problematic are symbols, which are used as formal indeterminates in polynomials and as variables in expressions. Interfacing to a computer algebra system through its user interface is therefore problematic.
- They have only limited capabilities for handling side conditions or case splits, if they exist at all. An example is $\int x^n \, dx$. Computer algebra systems return $\frac{x^{n+1}}{n+1}$. Substituting $n = -1$ yields an undefined term, while the solution of the integral is $\ln x$. This problem is known as specialisation problem, but hardly ever referred to in the literature, see [Corless and Jeffrey, 1997].
- Many of the algorithms that are implemented in computer algebra systems rest on mathematical theory and their correctness is well established: proofs for their correctness have been published. Examples for these are factorisation algorithms for polynomials, Gaussian elimination and Risch's method for integration in finite terms. The design of other algorithms is less rigorous. Simplification rules like $(x^2)^{\frac{1}{2}} = x$ are cause for some of the reported soundness problems. [Corless and Jeffrey, 1996] argue that a satisfactory treatment of these requires extending the underlying mathematical model. In this case Riemann surfaces are appropriate. We call the former sort of algorithms sound and the latter *ad hoc*. See [Calmet and Campbell, 1997, section 2] for a historic perspective on this distinction.

Of course, computer algebra systems also contain implementation errors. Depending on how rigorous one wants to be, one can reject any result of a computer algebra system without formal verification in the prover. Considering the amount of work required to re-implement these algorithms in a theorem prover, and the poor efficiency one could expect, we decide to live with possible bugs but look for ways of avoiding the systematic errors.

2.3 Design of the Interface

The interface obviously needs to translate objects between Isabelle's and the computer algebra system's representation. The translation cannot be performed uniformly, but needs to take into account which algorithm the objects are passed to or returned from. As we can only use a selection of algorithms of the system safely, we need to interface to these directly rather than to the system as a whole.

Unfortunately, it turns out to be difficult to tell sound algorithms from *ad hoc* ones in large, multipurpose computer algebra systems. Without lengthy code inspections one cannot be sure that a piece of otherwise sound code depends on a module that is *ad hoc*. We have therefore chosen the rather small computer algebra library Sumit, which is written in the strongly typed language Aldor, originally designed for the computer algebra system Axiom. References to the literature for the algorithms this library implements are available. From these, formal specifications can be extracted.

The implementation of a prototype interface between Isabelle and Sumit is straightforward. We provide stubs that translate between Isabelle's λ-terms and Sumit's algebraic objects. More than one stub is provided for Sumit types that are used for several mathematical domains. This is, for example, the case for Sumit's type `Integer`, which is used to represent both natural numbers and integers. Arguments and results of the computation are then composed to a λ-term representing a theorem. This is done using what we call a theorem template: at this experimental stage, simply a piece of code. The generated theorem is an instance of the algorithm's formal specification. The algebraic algorithms, stubs and theorem templates are wrapped to a server dealing with Isabelle's requests. The server we obtain this way is only a skeleton: stubs and theorem templates are added incrementally for algorithms that are to be used.

3 Polynomial Algebra

The algebraic approach to cyclic codes is based upon the theory of polynomial rings. We sketch this theory briefly and also show to what extent it has been formalised within Isabelle/HOL. The type system of this logic supports simple types extended by axiomatic type classes, which we use to represent abstract algebraic structures. Subtyping has to be made explicit using suitable embedding functions.

3.1 The Hierarchy of Ring Structures

One obtains various kinds of rings by imposing conditions on the ring's multiplicative monoid. *Integral domains*, or domains for short, do not contain any zero divisors other than zero: formally, $a \neq 0$ and $b \neq 0$ implies $a \cdot b \neq 0$.

An element a is said to *divide* b, if there is an element d such that $a \cdot d = b$. We write $a \mid b$. Two elements are *associated* $a \sim b$, if both $a \mid b$ and $b \mid a$. An element that divides 1 is called a *unit*. Associated elements differ by a unit factor only. An element is called *irreducible* if it is nonzero, not a unit and all its proper factors are units. Formally, $\mathrm{irred}\, a \equiv a \neq 0 \wedge a \nmid 1 \wedge (\forall d.\, d \mid a \longrightarrow d \mid 1 \vee a \mid d)$. An element is called *prime* if it is nonzero, not a unit and, whenever it divides a product, it already divides one of the factors. This is, formally, $\mathrm{prime}\, p \equiv p \neq 0 \wedge p \nmid 1 \wedge (\forall a\, b.\, p \mid a \cdot b \longrightarrow p \mid a \vee p \mid b)$. The factorisation of an element x into irreducible elements is defined by the following predicate:

$$\mathrm{Factorisation}\, x\, F\, u \equiv (x = \mathrm{foldr}\ \cdot\ F\, u) \wedge (\forall a \in F.\, \mathrm{irred}\, a) \wedge u \mid 1 \qquad (1)$$

F is the list of irreducible factors and u is a unit element. The list operator foldr combines all the elements of a list, here by means of the multiplication operation "$\cdot$". The product of the elements of F and of u is x.

An integral domain R is called *factorial* if the factorisation of the elements into irreducible factors is unique up to the order of the factors and associated elements. This is equivalent to R satisfying a divisor chain condition and every irreducible element of R being prime. The divisor chain condition is not needed in our proofs. We therefore formalise factorial domains only using the second condition, which is also called the *primeness condition*. Fields are commutative rings where every non-zero element has a multiplicative inverse.

3.2 Polynomials

Polynomials are a generic construction over rings. For every ring R there is a ring of polynomials $R[X]$. The symbol X is called the indeterminate of the polynomial ring. Further to the ring operations there is the embedding $\mathrm{const} : \left\{ \begin{smallmatrix} R \to R[X] \\ a \mapsto aX^0 \end{smallmatrix} \right\}$. We derive the representation theorem $\deg p \leq n \implies \sum_{i=0}^{n} p_i X^i = p$, where the p_i denote the coefficients of p.

Polynomials must not be confused with polynomial functions.[1] Their relation is described in terms of the evaluation homomorphism. Given another ring S and a homomorphism $\phi : R \to S$ we define $\mathrm{EVAL}\,\phi\,a\,p \equiv \sum_{i=0}^{\deg p} \phi p_i \cdot a^n$. $\mathrm{EVAL}\,\phi\,a : R[X] \to S$ is a homomorphism as well and evaluates a polynomial in S substituting $a \in S$ for the indeterminate and mapping the coefficients of p to S by ϕ.

3.3 Fields and Minimal Polynomial

The field $\mathbb{F}_2 = \{0, 1\}$ is fundamental in an algebraic treatment of binary codes. Codewords are represented as polynomials in $\mathbb{F}_2[X]$. Note that associated elements are equal in these domains.

Let h be an irreducible polynomial of degree n. The residue ring obtained from $\mathbb{F}_2[X]$ by "computing modulo h" is a field with 2^n elements. For our purpose we do not carry out this quotient construction of a field extension explicitly, as we only need it to define the notion of minimal polynomial. Let G be an extension field of F and $a \in G$. The nonzero polynomial $m \in F[X]$ of smallest degree, such that m evaluated at a is zero, is the minimal polynomial. Our definition of the minimal polynomial is as follows:

$$\mathrm{minimal}\,g\,S \equiv g \in S \wedge g \neq 0 \wedge (\forall v \in S.\, v \neq 0 \longrightarrow \deg g \leq \deg v) \tag{2}$$

$$\mathrm{min\_poly}\,h\,a \equiv \epsilon g.\,\mathrm{minimal}\,g\,\{p.\,\mathrm{EVAL}\,\mathrm{const}\;a\,p\;\mathrm{rem}\,h = 0\} \tag{3}$$

Note that here $a \in \mathbb{F}_2[X]$ and hence the embedding const is needed to lift the coefficients of p to $\mathbb{F}_2[X]$. The computation is carried out modulo h by means of the remainder function rem associated with polynomial division.

[1] Polynomial functions are a subtype of $R \to R$ and not isomorphic to $R[X]$ when R is finite: for $\mathbb{F}_2$ we have $|\mathbb{F}_2[X]| = \infty$, but $|\mathbb{F}_2 \to \mathbb{F}_2| = 4$.

4 Coding Theory

This discipline studies the transmission of information over communication channels. In practice, information gets distorted because of noise. Coding theory therefore seeks to design codes that allow for high information rates and the correction of errors introduced in the channel. At the same time, fast encoding and decoding algorithms are required to permit high transmission speeds.

The following presentation of coding theory follows [Hoffman *et al.*, 1991]. The codes we are interested in for the purpose of this case study belong to a class of binary codes with words of fixed length, so called block codes. n-error-detecting codes have the capability to detect n errors in the transmission of a word; n-error-correcting codes can even correct n errors. The distance between two codewords is the number of differing bit-positions between them. The distance of a code is the minimum distance between any two words of that code.

Definition 1 *A code is* linear *if the exclusive or of two codewords is also a codeword. It is* cyclic *if for every codeword $a_0 \cdots a_n$ its cyclic shift $a_n a_0 \cdots a_{n-1}$ is also a codeword.*

Codes that are linear and cyclic can be studied using algebraic methods. Linear codes are $\mathbb{F}_2$-vector spaces. A code with 2^k codewords has dimension k and there is a basis of codewords that span the code. It is convenient to identify codewords with polynomials:

$$a_0 \cdots a_{n-1} \quad \longleftrightarrow \quad a_0 + a_1 X + \ldots + a_{n-1} X^{n-1}$$

The cyclic shift of a codeword a is then $X \cdot a \operatorname{rem}(X^n - 1)$, where rem is the remainder function associated with polynomial division.

There is a nonzero codeword of least degree in every linear cyclic code. This is called the generator polynomial. It is unique and its cyclic shifts form a basis for the code. It is important, because a linear cyclic code is fully determined by its length and its generator polynomial. The generator polynomial has the following algebraic characterisation:

Theorem 2 (Generator polynomial) *There exists a cyclic linear code of length n such that the polynomial g is the generator polynomial of that code if and only if g divides $X^n - 1$.*

4.1 Hamming Codes

Hamming codes are linear codes of distance 3 and are 1-error-correcting. They are *perfect* codes: they attain a theoretical bound limiting the number of codewords of a code of given length and distance. For every $r \geq 2$ there are cyclic Hamming codes of length $2^r - 1$.

An irreducible polynomial of degree n that does not divide $X^m - 1$ for $m \in \{n + 1, \ldots, 2^n - 2\}$ is called *primitive*.[2] This allows us to state the following structural theorem on cyclic Hamming codes:

[2] Note that the term primitive polynomial is used with a different meaning in other areas of algebra.

Theorem 3 (Hamming code) *There exists a cyclic Hamming code of length* $2^r - 1$ *with generator polynomial* g*, if and only if* g *is primitive and* $\deg g = r$*.*

4.2 BCH Codes

Bose-Chaudhuri-Hocquengham (BCH) codes can be constructed according to a required error-correcting capability. We only consider 2-error-correcting BCH codes. These are of length $2^r - 1$ for $r \geq 4$ and have distance 5.

An element a of a field F is *primitive* if $a^i = 1$ is equivalent to $i = |F| - 1$ or $i = 0$. Let G be an extension field of $\mathbb{F}_2$ with 2^r elements and $b \in G$ a primitive element. The generator polynomial of the BCH code of length $2^r - 1$ is $m_b \cdot m_{b^3}$, where m_a denotes the minimal polynomial of a in the field extension. If we describe the field extension in terms of a primitive polynomial h, then X corresponds to a primitive element. Note that, because h is irreducible, it is minimal polynomial of X. Therefore we can define BCH codes a follows:

Definition 4 *Let* $h \in \mathbb{F}_2[X]$ *be a primitive polynomial of degree* r*. The code of length* $2^r - 1$ *generated by* $h \cdot \text{min\_poly}\, h\, X^3$ *is called a* BCH *code.*

5 Formalising Coding Theory

We formalise properties of codes with the following predicates. Codewords are polynomials over $\mathbb{F}_2$ and codes are sets of them. The statement $\text{code}\, n\, C$ means C is a code of length n. The definitions of linear and cyclic are straightforward while $\text{generator}\, n\, g\, C$ states that g is generator polynomial of the code C of length n.

$$\begin{aligned}
&\text{code}\, n\, C && \equiv \forall x \in C.\ \deg x < n \\
&\text{linear}\, C && \equiv \forall x \in C.\, \forall y \in C.\ x + y \in C \\
&\text{cyclic}\, n\, C && \equiv \forall x \in C.\ X \cdot x\, \text{rem}(X^n - 1) \in C \\
&\text{generator}\, n\, g\, C && \equiv \text{code}\, n\, C \wedge \text{linear}\, C \wedge \text{cyclic}\, n\, C \wedge \text{minimal}\, g\, C
\end{aligned}$$

5.1 The Hamming Code Proofs

We now describe our first application of the interface between Isabelle and Sumit. We use it to prove which Hamming codes of a certain length exist. Restricting the proof to a certain length allows us to make use of computational results obtained by the computer algebra system. The predicate Hamming describes which codes are Hamming codes of a certain length. Theorems 2 and 3 are required and formalised as follows:

$$0 < n \longrightarrow (\exists C.\, \text{generator}\, n\, g\, C) = g \mid X^n - 1 \tag{4}$$

$$(\exists C.\, \text{generator}(2^r - 1)\, g\, C \wedge \text{Hamming}\, r\, C) = (\deg g = r \wedge \text{primitive}\, g) \tag{5}$$

These equations are asserted as axioms and are the starting point of the proof that follows. Note that (5) axiomatises the predicate Hamming. The generators

of Hamming codes are the primitive polynomials of degree $2^r - 1$. The primitive polynomials of degree 4 are $X^4 + X^3 + 1$ and $X^4 + X + 1$. Thus for codes of length 15 we prove

$$(\exists C.\, \text{generator}\, 15\, g\, C \wedge \text{Hamming}\, r\, C) = (g \in \{X^4 + X^3 + 1, X^4 + X + 1\}).$$

We now give a sketch of this proof, which is formally carried out in Isabelle. The proof idea for the direction from left to right is that we obtain all irreducible factors of a polynomial by computing its factorisation. The generator g is irreducible by (5) and a divisor of $X^{15} - 1$ by (4). The factorisation of $X^{15} - 1$ is computed using Berlekamp's algorithm:

$$\begin{aligned}\text{Factorisation}(X^{15} - 1)\, [&X^4 + X^3 + 1, X + 1, X^2 + X + 1,\\ &X^4 + X^3 + X^2 + X + 1, X^4 + X + 1]\, 1\end{aligned}$$

Since associates are equal in $\mathbb{F}_2[X]$ every irreducible divisor of $X^{15} - 1$ is in this list. This follows from the lemma

$$\text{irred}\, c \wedge \text{Factorisation}\, x\, F\, u \wedge c \mid x \Longrightarrow \exists d.\, c \sim d \wedge d \in F, \tag{6}$$

whose proof requires an induction over the list F. It follows in particular that the generator polynomials are in the list above. But some polynomials in the list cannot be generators: $X + 1$ and $X^2 + X + 1$ do not have degree 4 and $X^4 + X^3 + X^2 + X + 1$ divides $X^5 - 1$ and is therefore not primitive. The only possible generators are thus $X^4 + X^3 + 1$ and $X^4 + X + 1$.

It remains to show that these are indeed generator polynomials of Hamming codes. This is the direction from right to left. According to (5) we need to show that $X^4 + X^3 + 1$ and $X^4 + X + 1$ are primitive and have degree 4. The proof is the same for both polynomials. Let p be one of these. The irreducibility of p is proved by computing the factorisation, which is $\text{Factorisation}\, p\, [p]\, 1$, and follows from the definition of Factorisation, equation (1).

The divisibility condition of primitiveness is shown by verifying $p \nmid X^m - 1$ for $m = 5, \ldots, 14$. □

5.2 The BCH Code Proofs

The predicate BCH is, in line with definition 4, defined as follows:

$$\begin{aligned}\text{BCH}\, r\, C \;\equiv\; (\exists h.\, &\text{primitive}\, h \wedge \deg h = r \wedge\\ &\text{generator}(2^r - 1)\, (h \cdot \text{min\_poly}\, h\, X^3)\, C)\end{aligned} \tag{7}$$

We prove that a certain polynomial is generator of a BCH code of length 15:

$$\text{generator}\, 15\, (X^8 + X^7 + X^6 + X^4 + 1)\, C \Longrightarrow \text{BCH}\, 4\, C$$

Here is the outline of the proof: $X^8 + X^7 + X^6 + X^4 + 1$ is the product of the primitive polynomial $X^4 + X + 1$ and the minimal polynomial $X^4 + X^3 + X^2 + X +$

1. According to the definition (7) we need to show that the former polynomial is primitive. This has been described in the second part of the Hamming proof. Secondly, we need to show that the latter is a minimal polynomial:

$$\text{min\_poly}(X^4 + X + 1)\, X^3 = X^4 + X^3 + X^2 + X + 1$$

In order to prove this statement, we need to show that $X^4 + X^3 + X^2 + X + 1$ is a solution of

$$\text{EVAL}\, \text{const}\ X^3\ p\ \text{rem}(X^4 + X + 1) = 0 \tag{8}$$

of minimal degree, and that it is the only minimal solution.

- Minimal solution: Simplification establishes that $X^4 + X^3 + X^2 + X + 1$ is a solution of the equation. That there are no solutions of smaller degree can be shown as follows:
  Assume $\deg p \leq 3$, so $p = p_0 + p_1 X + p_2 X^2 + p_3 X^3$ for $p_0, \ldots, p_3 \in \mathbb{F}_2$. We substitute this representation of p in (8) and obtain, after simplification,

  $$p_0 + p_1 X^3 + p_2(X^2 + X^3) + p_3(X + X^3) = 0.$$

  Comparing coefficients leads to a linear equation system, which we can solve using the Gaussian algorithm. The only solution is $p_0 = \cdots = p_3 = 0$, so $p = 0$. This is a contradiction to the definition of minimal.
- Uniqueness: We need to show that $X^4 + X^3 + X^2 + X + 1$ is the only polynomial of smallest degree satisfying (7). We study the solutions of (8) of degree of ≤ 4 by setting $p = p_0 + \ldots + p_4 X^4$ and obtain another equation system

  $$p_0 + p_1 X^3 + p_2(X^2 + X^3) + p_3(X + X^3) + p_4(1 + X + X^2 + X^3) = 0.$$

  Its set of solutions, again computed by the Gaussian algorithm, is $\{0, X^4 + X^3 + X^2 + X + 1\}$. The definition of minimality excludes $p = 0$. Therefore there are indeed no other solutions of minimal degree. □

6 Review of the Development

We have mechanised the mathematics outlined in section 3 and the proofs described in section 5 in our combination of Isabelle and Sumit. The mathematical background presented in section 3 has been formalised by asserting definitions for the entities and deriving the required theorems mechanically. This is advisable to maintain consistency. We have not done the same for coding theory. Here we have only asserted the results, namely theorems 2 and 3 and then mechanised the proofs described in section 5. This part is therefore considerably shorter than the development of the mathematical background.

The following table gives an overview on the effort. The figures are, however, misleading in such that developing proof scripts is much harder than ordinary programming.

Isabelle		Sumit	
Interface	23.7	Interface	43.3
Formalisation of algebra	61.8	Stubs and	
Coding theory proofs	14.6	theorem templates	20.4

Size of the development (code sizes in 1000 bytes)

The interface of Sumit is considerably larger, because datatypes for λ-terms and the server functionality are provided as well. The entry "Coding theory proofs" includes the implementation of proof procedures for irreducibility and primitiveness of polynomials, which automatically examine the proof state and retrieve the required theorems from Sumit.

6.1 Contributions of the Prover

We prove theorems about polynomial algebra, which do not have computational content, in Isabelle. We also establish the relation between coding theory and the specifications of the algebraic algorithms. In our informal presentation these translations may appear simple, but some of them are in fact rather difficult.

For the Hamming code proofs take lemma (6), for example, which is proved by list induction. The induction step, after unfolding definitions, is a quantifier expression, which is solved almost automatically by Isabelle's tableau prover. However, it requires search to a depth of six, which means that six "difficult" rules have to be applied, and produces a proof with 221 inferences. A depth of six is unusually deep in interactive proof. The complete proof of (6) is 372 inferences long but only requires 8 invocations of tactics, which resemble the manual proof steps.

In the proofs about BCH codes, reasoning about minimality needs the full power of first order logic. Note that the definition of minimality (2) contains a quantifier and phrases like "x is the only element, such that P" are really statements that involve quantifiers.

6.2 Contributions of Computer Algebra

Sumit computes normal forms for expressions that do not contain variables; here in the domains $\mathbb{N}, \mathbb{F}_2, \mathbb{F}_2[X]$. This includes the decision of equality, inequalities and divisibility over these expressions. Their theorem templates are of the form $a \odot b = B$, where $\odot$ is the corresponding connective and B becomes either *True* or *False*.

Polynomials are decomposed into square-free factors and then factorised over $\mathbb{F}_2[X]$ using Berlekamp's algorithm. We pass a polynomial p to this procedure and obtain a list of irreducible factors $[x_1, \ldots, x_k]$ and a unit element u. These are then assembled to the theorem

$$\text{Factorisation}\, x\, [x_1, \ldots, x_k]\, u.$$

Linear equation systems over $\mathbb{F}_2$ are solved by Gaussian elimination. The matrix $(a_0 | \cdots | a_n)$ is passed to the algorithm, where a_i is the ith column vector.

The algorithm returns a list of vectors $[v_1, \ldots, v_k]$ that span the solution space. The theorem template generates the theorem

$$(\sum_{i=0}^{n} x_i a_i = 0) \;=\; (\exists t_1 \cdots t_k.\, x = t_1 v_1 + \ldots + t_k v_k)$$

$$\text{or} \quad (\sum_{i=0}^{n} x_i a_i = 0) \;=\; (x = 0), \quad \text{if } k = 0.$$

The t_i are variables in $\mathbb{F}_2$ and the x_i are elements of the vector x. Note that we use polynomials to denote vectors in Isabelle, as indicated in the proof.

Mechanising the proofs in a system that integrates the computer algebra component without trusting it would require to additionally prove the theorems generated by these templates formally. This holds in particular for [Harrison, 1996, chapter 6] and [Kerber *et al.*, 1996], who try to reconstruct the proofs using the result of the computation and possibly further information, which resembles a certificate for the computation.

In the case of our proofs, the irreducibility of the factors, which constitute a factorisation, is hard to establish and also the direction from left to right in the theorems generated by Gaussian elimination.[3] This direction states that the solution is complete, and it is the direction needed in the proofs.

7 Conclusion

Our approach is pragmatic: we trust the computer algebra component in our system rather than reconstruct proofs for the results of computations within the prover's logic. The approach relies on implementations of algorithms that are trustworthy. This can be achieved by restricting the use of computer algebra to algorithms, for which proofs of their correctness have been published. This is sufficient to avoid systematic soundness problems of computer algebra systems. Errors in the implementation of these algorithms still jeopardise the integrity of the prover, but bugs of this sort should not be more frequent in computer algebra systems than in other software (including provers themselves).

Computational results are turned into theorems using theorem templates that can produce arbitrary theorems. This is more flexible than the approach suggested by one of us [Ballarin *et al.*, 1995], which only allowed conditional rewrite rules, because the logical meaning of the result can be exploited more easily.

Our case study shows that theorems that are rather difficult to verify occur naturally in proofs. It presents a challenge to the approach that does not trust the computer algebra component. But it also makes a contribution: it clarifies which theorems need to be certified.

[3] Over some domains theorems of this kind can be proved by decision procedures for linear arithmetic. Here, because $|\mathbb{F}_2| = 2$, this could be done by checking all the 2^{n+1} cases.

Our approach avoids Analytica's soundness problems. This means, of course, that we cannot make use of algorithms that are *ad hoc*. In an interactive environment it does not matter too much that these are not complete. They need, however, to be made sound. Expressive formalisms that are able to deal with side conditions and case splits are used in mechanised reasoning. Expertise gained here could prove useful in the redesign of these algorithms as well.

Acknowledgements. This work has been funded in part by the Studienstiftung des deutschen Volkes and by EPSRC grant GR/K57381 "Mechanizing Temporal Reasoning".

References

[Ballarin *et al.*, 1995] Clemens Ballarin, Karsten Homann, and Jacques Calmet. Theorems and algorithms: An interface between Isabelle and Maple. In A. H. M. Levelt, editor, *ISSAC '95: International symposium on symbolic and algebraic computation — July 1995, Montréal, Canada*, pages 150–157. ACM Press, 1995.

[Bronstein, 1996] Manuel Bronstein. Sumit — a strongly-typed embeddable computer algebra library. In Calmet and Limongelli [1996], pages 22–33.

[Calmet and Campbell, 1997] J. Calmet and J. A. Campbell. A perspective on symbolic mathematical computing and artificial intelligence. *Annals of Mathematics and Artificial Intelligence*, 19(3–4):261–277, 1997.

[Calmet and Limongelli, 1996] Jacques Calmet and Carla Limongelli, editors. *Design and Implementation of Symbolic Computation Systems: International Symposium, DISCO '96, Karlsruhe, Germany, September 18–20, 1996: proceedings*, number 1128 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[Clarke and Zhao, 1993] Edmund Clarke and Xudong Zhao. Analytica: A theorem prover for Mathematica. *The Mathematica Journal*, 3(1):56–71, 1993.

[Corless and Jeffrey, 1996] Robert M. Corless and David J. Jeffrey. The unwinding number. *ACM SIGSAM Bulletin*, 30(2):28–35, 1996.

[Corless and Jeffrey, 1997] R. M. Corless and D. J. Jeffrey. The Turing factorization of a rectangular matrix. *ACM SIGSAM Bulletin*, 31(3):20–28, 1997.

[Geddes *et al.*, 1992] Keith O. Geddes, Stephen R. Czapor, and George Labahan. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

[Harrison, 1996] John Robert Harrison. Theorem proving with the real numbers. Technical Report 408, University of Cambridge, Computer Laboratory, November 1996.

[Hoffman *et al.*, 1991] D. G. Hoffman, D. A. Leonard, C. C. Lindner, K. T. Phelps, C. A. Rodger, and J. R. Wall. *Coding Theory: The Essentials*. Number 150 in Monographs and textbooks in pure and applied mathematics. Marcel Dekker, Inc., New York, 1991.

[Homann, 1997] Karsten Homann. *Symbolisches Lösen mathematischer Probleme durch Kooperation algorithmischer und logischer Systeme*. Number 152 in Dissertationen zur Künstlichen Intelligenz. infix, St. Augustin, 1997.

[Kerber *et al.*, 1996] Manfred Kerber, Michael Kohlhase, and Volker Sorge. Integrating computer algebra with proof planning. In Calmet and Limongelli [1996], pages 204–215.

[Paulson, 1994] Lawrence C. Paulson. *Isabelle: a generic theorem prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[Stoutemyer, 1991] David R. Stoutemyer. Crimes and misdemeanors in the computer algebra trade. *Notices of the American Mathematical Society*, 38(7):778–785, 1991.