Fundamenta Informaticae 34 (1999) 1–20
IOS Press

A Pragmatic Approach to Extending Provers by Computer Algebra — with Applications to Coding Theory

Clemens Ballarin
*Fakultät für Informatik**
Universität Karlsruhe
Karlsruhe, Germany
ballarin@ira.uka.de

Lawrence C. Paulson
Computer Laboratory
University of Cambridge
Cambridge, UK
lcp@cl.cam.ac.uk

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 formal specifications for the algorithms, and links the computer algebra library Sumit to the prover Isabelle.

We give details of the interface, the use of the computer algebra system on the tactic-level of Isabelle and its integration into proof procedures.

Keywords: Computer algebra, mechanised reasoning, combining systems, soundness of computer algebra systems, specialisation problem, coding theory.

*Research was carried out while at the Computer Laboratory, University of Cambridge.

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, however, either have little reasoning content, or use symbolic computation only to simplify 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.

The extension of a prover by computer algebra does not, or rather should not, change the prover's *logical* strength. In fact, trustworthy extension of a prover means that neither more nor fewer theorems should be provable in principle, but proofs for certain theorems should become shorter. The motivation for our work comes from the pragmatics of interactive proof, which often is tedious and laborious. Therefore it is not helpful, if a computation obtained by a computer algebra system needs to be verified. Many algorithms in computer algebra can be trusted. It would be possible to verify such algorithms in a prover, and this has indeed been done, for example by [Théry, 1998] for Buchberger's algorithm. However, we are not concerned with such applications of provers. Algorithms from computer algebra are generally not of much use in these proofs, because the theorems that have to be proved are rather abstract. As far as the design of a prover is concerned, not much can be learned from such experiments.

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 the prover 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 remainder 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 conclusions follow in Section 7.

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, 1996a, Bronstein, 1996b] respectively.

2.1. Isabelle

Isabelle is a natural deduction-style theorem prover. Proofs are carried out interactively 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 data-type `thm` for theorems. Theorems can only be generated by operations provided by the data-type. 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, presupposing its existence. 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.

  An example for this problem can be illustrated with the solve command:

  $$\text{solve}(\frac{(x-1)^2}{x^2-1} = 0, x)$$

  yields $x = 1$ on a number of computer algebra systems. But this is not a solution to the equation, because for $x = 1$ the denominator vanishes. If we perform the computation in

Axiom [Jenks and Sutor, 1992] and split it in two steps we obtain a hint on the reason by the system's type system. See Figure 1.

```
frame0 (1) ->a := (x-1)^2/(x^2-1)
          x - 1
   (1)    -----
          x + 1
                              Type: Fraction Polynomial Integer
frame0 (2) ->solve(a = 0, x)
   (2)  [x= 1]
          Type: List Equation Fraction Polynomial Integer
```

Figure 1. Solving the equation $\frac{(x-1)^2}{x^2-1} = 0$ in Axiom.

The fraction is treated as a fraction of polynomials and automatically cancelled. This is a valid operation on polynomials, because here x is an indeterminate, not a variable, and in particular $x - 1 \neq 0$. The solve-function, despite using the same type, treats x as a variable. This incompatibility leads to the wrong answer. For a more precise description of the problem, see Appendix A.

Therefore interfacing to a computer algebra system through its user interface is problematic. In this case, the situation is even worse, because an algorithm that is built into the system assumes the wrong semantics for the data-type it operates on.

- Computer algebra systems have only limited capabilities for handling side conditions or case splits, if such facilities 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 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. A prominent example is definite integration. Computing the area under a curve f is usually done by determining its anti-derivative F. By the fundamental theorem of integral calculus, one gets $\int_a^b f = Fb - Fa$. [Davenport, 1998] points out that this is unsound for several reasons: the integral might not exist or contain discontinuities. Also the output of Risch's method does not necessarily correspond to a continuous function.

  We call the former sort of algorithms *sound* and the latter *ad hoc*. A historic perspective on this distinction can be found in [Calmet and Campbell, 1997, Section 2].

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*. This is particularly so, because code for simplification of expressions is usually spread all over the system.

We have therefore chosen the rather small computer algebra library Sumit, which is written in the strongly typed language Aldor [Watt *et al.*, 1994], originally designed for the computer algebra system Axiom. References to the literature for the algorithms implemented by this library. From these, formal specifications can be extracted.

The outline 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. Arguments and results of the computation are 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.

The relation between types, or more generally speaking representations, in the two systems is involved. Sumit uses the type `Integer` to represent both natural numbers and integers. These have to be distinguished in Isabelle, because they obey different laws of reasoning. Conversely, polynomials in Isabelle are translated to sparse univariate polynomials in Sumit, if a factorisation is to be computed. In order to solve the equation system given by comparing coefficients, the polynomials need to be coerced to vectors of appropriate size.[1] This means that the relation between the types does not even form a mapping.

We resolve this problem by letting translations depend not only on the types, but also on the algorithm they interface. We call an algorithm together with its translation functions a *service*. Note that this also avoids the type reconstruction problem we would otherwise have, when interfacing to an untyped computer algebra system.

[1] Representation changes are common in computer algebra. One could try and perform all of them in the reasoning system. But this would not be very efficient, and given the state of art in mechanised reasoning, formalising as few algebraic domains as possible is desirable.

3. Polynomial Algebra

The algebraic approach to cyclic codes is based upon the theory of polynomial rings. We sketch this theory briefly to provide the necessary background, and also to 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. Our formalisation follows [Jacobson, 1985]. We have proved all the stated facts and theorems in Isabelle and in fact many more lemmas, most of them being too trivial to be mentioned. Nevertheless, they are important in order to reason in that domain.

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,

$$\operatorname{irred} a \;\equiv\; a \neq 0 \wedge a \nmid 1 \wedge (\forall d.\, d \mid a \longrightarrow d \mid 1 \vee a \mid d).$$

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.

$$\operatorname{prime} p \;\equiv\; p \neq 0 \wedge p \nmid 1 \wedge (\forall a\, b.\, p \mid a \cdot b \longrightarrow p \mid a \vee p \mid b)$$

The factorisation of an element x into irreducible elements is defined by the following predicate:

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

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. Therefore we 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 a given ring R, they are an abstraction of functions from $\mathbb{N} \to R$ that map all but a finite number of natural numbers to zero. Appropriate definitions of the operations make this structure a ring, which is denoted by $R[X]$. The symbol X abbreviates the function $(n \mapsto \text{if } n = 1 \text{ then } 1 \text{ else } 0)$ and is called the *indeterminate* of the polynomial ring. Further to the ring operations there is the embedding $\text{const} : \left\{ \begin{smallmatrix} R \to R[X] \\ a \mapsto aX^0 \end{smallmatrix} \right\}$. We derive the representation theorem

$$\deg p \leq n \implies \sum_{i=0}^{n} p_i X^i = p, \tag{2}$$

where the p_i denote the coefficients of p.

Polynomials must not be confused with polynomial functions.[2] Their relation is described in terms of the evaluation homomorphism. A function $f : R \to S$ over rings is a *ring homomorphism* if $f(a+b) = fa + fb$, $f(a \cdot b) = fa \cdot fb$ and $f1 = 1$. Given a homomorphism $\phi : R \to S$ we define

$$\text{EVAL}\, \phi\, a\, p \equiv \sum_{i=0}^{\deg p} \phi p_i \cdot a^n.$$

$\text{EVAL}\, \phi\, a : R[X] \to S$ is a homomorphism as well. It evaluates a polynomial in S substituting $a \in S$ for the indeterminate and mapping the coefficients of p to S by ϕ. The following facts express this characterisation formally.

$$\text{homo}\, \phi \implies \text{homo}(\text{EVAL}\, \phi\, a) \tag{3}$$

$$\text{homo}\, \phi \implies \text{EVAL}\, \phi\, a\, (X^n) = a^n \tag{4}$$

$$\text{EVAL}\, \phi\, a\, (\text{const}\, b) = \phi b \tag{5}$$

Here the predicate homo asserts that its argument is a ring homomorphism.

Polynomial rings have the following properties in terms of the hierarchy of rings: polynomials over a ring form a ring and polynomials over an integral domain again a domain. Polynomials over a field form a factorial domain.

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. We do not need to carry out this quotient construction of a field extension explicitly, as we only need to define the notion of minimal

[2] 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 $|\mathbb{F}_2 \to \mathbb{F}_2| = 4$.

polynomial. Let G be an extension field of F and $a \in G$. The non-zero 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 uses two steps:

$$\text{minimal}\, g\, S \;\equiv g \in S \wedge g \neq 0 \wedge (\forall v \in S.\, v \neq 0 \longrightarrow \deg g \leq \deg v) \tag{6}$$

$$\text{min\_poly}\, h\, a \equiv \epsilon g.\, \text{minimal}\, g\, \{p | (\text{EVAL}\, \text{const}\ a\, p) \,\text{rem}\, h = 0\} \tag{7}$$

The predicate minimal $g\, S$ abbreviates that g is a polynomial of minimal degree, but not zero, in the set S. This cannot be formalised using a function, because the minimal element need not be unique. The minimal polynomial of a in the extension constructed with h is then the unique minimal element of the set of solutions for p of the equation $(\text{EVAL}\, \text{const}\ a\, p) \,\text{rem}\, h = 0$. 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.

4. Coding Theory

This discipline studies the transmission of information over communication channels. In practice, information gets distorted because of noise. Therefore coding theory 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 4.1. 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 $(X \cdot a) \,\text{rem}\, (X^n - 1)$.

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 4.1. (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*.[3] This allows us to state the following structural theorem on cyclic Hamming codes:

Theorem 4.2. (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 the minimal polynomial of X. Therefore we can define BCH codes a follows:

Definition 4.2. 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.

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

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

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 4.1 and 4.2 are required and formalised as follows:

$$0 < n \longrightarrow (\exists C.\, \text{generator}\, n\, g\, C) = g \mid X^n - 1 \tag{8}$$

$$(\exists C.\, \text{generator}(2^r - 1)\, g\, C \wedge \text{Hamming}\, r\, C) = (\deg g = r \wedge \text{primitive}\, g) \tag{9}$$

These equations are asserted as axioms and are the starting point of the proof that follows. Note that (9) axiomatises the predicate Hamming. Both theorems are not proved formally. 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 (9) and a divisor of $X^{15} - 1$ by (8). The factorisation of $X^{15} - 1$ is computed using Berlekamp's algorithm:

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

All the irreducible divisors of $X^{15} - 1$ are in this list. This follows from

$$\text{irred}\, c \wedge \text{Factorisation}\, x\, F\, u \wedge c \mid x \Longrightarrow \exists d.\, c \sim d \wedge d \in F. \tag{10}$$

Since associates are equal in $\mathbb{F}_2[X]$ we have the stronger version

$$\text{irred}\, c \wedge \text{Factorisation}\, x\, F\, u \wedge c \mid x \Longrightarrow c \in F.$$

It follows in particular that the generator polynomials are in the list above. But some polynomials in that 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 hence is 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 (9) 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 Factorisation $p\, [p]\, 1$, and follows from the definition of Factorisation, equation (1).[4]

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

[4] One might argue that using a factorisation algorithm to do a mere irreducibility test is like cracking a walnut

5.2. The BCH Code Proofs

The predicate BCH is, in line with definition 4.2, defined as follows:

$$\begin{aligned}\mathrm{BCH}\, r\, C \;\equiv\; (\exists h.\, \mathrm{primitive}\, h \wedge \deg h = r \wedge \\ \mathrm{generator}(2^r - 1)\, (h \cdot \mathrm{min\_poly}\, h\, X^3)\, C)\end{aligned} \tag{11}$$

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

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

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 (11) 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:

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

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

$$(\mathrm{EVAL}\, \mathrm{const}\ X^3\, p) \,\mathrm{rem}\, (X^4 + X + 1) = 0 \tag{12}$$

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

- Minimal solution: We substitute $X^4 + X^3 + X^2 + X + 1$ for p in (12). The embedding const is a homomorphism, and so also EVAL const X^3. The left argument of the remainder is simplified using the properties of the evaluation homomorphism (3) to (5), and the remainder-operation is then evaluated by Sumit to 0. Hence $X^4 + X^3 + X^2 + X + 1$ is a solution of the equation.

  Assuming $\deg p \leq 3$, we get by (2) that $p = p_0 + p_1X + p_2X^2 + p_3X^3$ for $p_0, \ldots, p_3 \in \mathbb{F}_2$. We substitute this representation of p in (12) and obtain, after simplification,

  $$p_0 + p_1X^3 + p_2(X^2 + X^3) + p_3(X + X^3) = 0.$$

  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 does not meet 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 (11). We study the solutions of (12) of degree of ≤ 4 by setting $p = p_0 + \ldots + p_4X^4$ and obtain another equation system

  $$p_0 + p_1X^3 + p_2(X^2 + X^3) + p_3(X + X^3) + p_4(1 + X + X^2 + X^3) = 0.$$

with a sledgehammer. In fact, Berlekamp's algorithm first determines the number of irreducible factors and then computes them. So, in case of an irreducible polynomial, the algorithm stops after determining that there is only one factor. In the first part of the proof, the use of Berlekamp's algorithm reduces the number of possible candidates for generators dramatically. Brute force testing of polynomials of degree r is not feasible: their number increases exponentially with r.

The theorem for the solution of this equation system, again computed by the Gaussian algorithm, is

$$(p_0 + p_1X^3 + p_2(X^3 + X^2) + p_3(X^3 + X) + p_4(X^3 + X^2 + X + 1) = 0) \\ = (\exists t.\, p = t \,\dot{s}\, (X^4 + X^3 + X^2 + X + 1)).$$

The set of solutions is therefore $\{0, X^4 + X^3 + X^2 + X + 1\}$. The definition of minimality excludes $p = 0$. So 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 4.1 and 4.2 and then mechanised the proofs described in Section 5. Therefore this part is shorter than the development of the mathematical background.

Isabelle is an interactive prover. To prove a statement, one enters it as a goal. This goal is then reduced to a number of (hopefully simpler) subgoals. These are in turn reduced and the process continues until all subgoals have been resolved. The reduction of goals is done manually by the user by invoking proof-functions, which are called *tactics*. Thus the initiative to drive the proof lies with the user. Some of the tactics provided by Isabelle are very simple. They implement primitive deduction rules like *modus ponens*. Other tactics implement proof procedures, for example for rewriting or to prove subgoals in first order logic. This is where the automation comes in. Primitive tactics are usually used to guide proofs where automatic tactics fail. Isabelle's language of tactics is a full programming language and arbitrary proof procedures can be implemented by the user. A sequence of tactic invocations that leads to a proof is a *proof script*. A tactic usually produces an instantaneous answer, and only some proof procedures can take longer. In this environment the computer algebra system is presented by our interface as a function that returns theorems. These can be supplied to tactics (see Section 6.3).

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

Table 1. Size of the development (code sizes in 1000 bytes)

Table 1 gives an overview on the effort. The figures are, however, misleading in so far that developing proof scripts is much harder than ordinary programming.

The interface of Sumit is considerably larger, because data-types 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 (10), 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 (10) 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 (6) contains a quantifier and phrases like "x is the only element, such that P" are really statements that involve quantifiers.

6.2. Configuration of the Interface

Evaluation of Isabelle's λ-terms into Sumit objects can be done uniformly, using an *evaluator*. So far, λ-abstractions have not been needed, and thus the implementation does not handle this case. Abstractions will occur, for example, in the context of an integration operator. Then a choice of evaluation strategy will have to be made. Call-by-value seems appropriate, because the purpose of the evaluation is to translate the whole given term into a Sumit object. Call-by-name will not translate "unnecessary" subterms, which could save translation costs. We take the point of view that all information should be passed on to the computer algebra system. If β-reductions are desired, they can be done in the prover easily.

Aldor is a staticly typed language: all type information needs to be known at compile-time, though parametric polymorphism is possible through dependent types. For the evaluator this means that we need to provide an evaluation function for every type. This is done by instantiating the evaluator, which is a polymorphic function. Table 2 shows evaluation functions for which types are needed to evaluate expressions in $\mathbb{F}_2[X]$, together with the constants and their corresponding functions in Sumit. In Isabelle, the type bool is used also for the domain $\mathbb{F}_2$; up is the type constructor for univariate polynomials. Sumit types are abbreviated: `F2` stands for `SmallPrimeField 2` and `Up` for `SparseUnivariatePolynomial`. The constants Plus and BCons encode a binary representation for numbers in Isabelle. These are essentially stored as lists of bits, where Plus represents zero and BCons appending a least significant bit to the list

Isabelle		Sumit	
Type	Constant	Type	Operation
nat	0	`Int`	`0`
nat ⇒ nat	Suc	`Int -> Int`	$\lambda n.\, n$ `+ 1`
nat ⇒ nat ⇒ nat	$+, \cdot, \hat{}$	`Int -> Int -> Int`	`+, *, ^`
	$-$	$\lambda m\, n.$ `if` m `>=` n `then` m `-` n `else 0`	
bool	True, False	`Bool`	`true, false`
nat	Plus	`Int`	`0`
nat ⇒ bool ⇒ nat		`Int -> Bool -> Int`	
	BCons	$\lambda x\, b.$ `2 *` x `+ if` b `then 1 else 0`	
bool	0, 1	`F2`	`0, 1`
bool ⇒ bool	$-$	`F2 -> F2`	`-`
bool ⇒ bool ⇒ bool	$+, \cdot$	`F2 -> F2 -> F2`	`+, *`
bool ⇒ nat ⇒ bool	$\hat{}$	`F2 -> Int -> F2`	`^`
bool up	0, 1	`Up F2`	`0, 1`
bool up ⇒ bool up	$-$	`Up F2 -> Up F2`	`-`
bool up ⇒ bool up ⇒ bool up	$+, \cdot$, rem	`Up F2 -> Up F2 -> Up F2`	`+, *, rem`
nat ⇒ bool up	monom	`Int -> Up F2`	$\lambda n.$ `monom ^` n
bool ⇒ bool up	const	`F2 -> Up F2`	`coerce`
bool ⇒ bool up ⇒ bool up	$\dot{s}$	`F2 -> Up F2 -> Up F2`	`*`
nat ⇒ bool up ⇒ bool		`Int -> Up F2 -> F2`	
	coeff	$\lambda n\, p.$ `coefficient (`p`,` n`)`	
bool up ⇒ nat		`Up F2 -> Int`	
	deg	$\lambda p.$ `(`n `:= degree` p`;` `if` n `< 0 then 0 else` p`)`	

Table 2. Specification of the evaluator for expressions in $\mathbb{F}_2[X]$.

of bits. Bits are represented as Boolean values. λ-notation is used here to abbreviate function definitions in Aldor.

If a value of some type τ can be obtained by application of a function of type $\sigma \Rightarrow \tau$ to a value of type σ, because of Aldor's type system, we must also specify which evaluation functions have to be used for the function and for its argument. In this example, 18 such pairs of types can occur and need to be specified. Also evaluation functions for three more types, which do not have any constants associated with them, are required. We have omitted this information from Table 2. In future, we aim to generate these specifications automatically by a suitable tool. Among evaluation functions for the other types, we obtain `EvalUpF2` for $\mathbb{F}_2[X]$.

The inverse operation, translating results back into Isabelle's format, cannot be done uniformly. Sumit operations to traverse its data structures need to be used to build appropriate λ-terms. Polynomials over $\mathbb{F}_2$, for instance, are translated into sums of monomials. A suitable iterator, provided by Sumit, is used in the function `BuildUpF2` to iterate over the monomials of a polynomial. It only returns monomials with non-zero coefficients, which ensures that we obtain a sparse representation. The zero-polynomial is, of course, mapped to 0.

The instantiation of the interface is complete, if services are provided. We decribe this in more detail for factorisation in the next section.

6.3. Contributions of Computer Algebra

Sumit computes normal forms for expressions that do not contain variables, here in the domains $\mathbb{N}, \mathbb{F}_2$ and $\mathbb{F}_2[X]$. 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 over $\mathbb{F}_2[X]$ are factorised. This functionality, whose implementation we describe in a little more detail, is provided by the service `F2PolyFact`. Sumit's factorisation algorithms for fields of prime cardinality reside in the module

```
PrimeFieldUnivariateFactorizer(
    F: PrimeFieldCategory,
    P: UnivariatePolynomialCategory F )
```

where the parameter `F` is a field of prime cardinality, and `P` is an implementation of univariate polynomials over that field. The module, amongst other functions, provides

```
berlekamp : P -> List P
factor    : ( P -> List P ) -> P -> ( F, Product P )
```

`berlekamp` p decomposes p into irreducible factors and returns them as a list. The argument must be a square-free and monic polynomial. The latter means that its leading coefficient is one. The decomposion into square-free factors can be obtained relatively easily, and is a prerequisite for most factorisation algorithms. It is implemented in the function `factor`. The function `berlekamp` is Sumit's implementation of Berlekamp's algorithm.

`factor` *algorithm* p factors an arbitrary non-zero polynomial p. It decomposes p into square-free, monic polynomials and factors them into irreducible ones, using *algorithm*.[5] It returns the leading coefficient of p and all the irreducible factors. `Product P` is a type for multi-sets, but also provides an operation to obtain the product of all the factors that are in it. Leading coefficient and the product give the complete factorisation of p.

The service `F2PolyFact` takes a single polynomial x, converts it to Sumit's representation with `EvalUpF2`, factors it using `factor` with `berlekamp` as its first argument and obtains a coefficient u and a product of irreduible factors P. The coefficient is lifted to the corresponding constant polynomial, and this and the factors in P are converted back to λ-terms with the build-function `BuildUpF2`. These are then assembled to the theorem

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

[5] Sumit also implements Cantor-Zassenhaus factorisation, which could be used here instead of Berlekamp's algorithm.

where $[x_1, \ldots, x_k]$ is the list of factors in P. In $\mathbb{F}_2[X]$ the polynomial u is, of course, 1. Also in other fields, non-zero constant polynomials are always units in the corresponding polynomial domain.

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 as is the direction from left to right in the theorems generated by Gaussian elimination.[6] This direction states that the solution is complete, and it is the direction needed in the proofs.

6.4. Use of Services on the Tactic Level

Proof scripts in Isabelle are coded in the programming language ML (for an introduction see [Paulson, 1996]), which underlies the prover. The interface to the computer algebra system is available on this level. Its main functions are

```
connect_server    : string -> string list
disconnect_server : string -> unit
thm_service       : string -> string -> term list -> thm
```

`connect_server` *server* starts a process for the server with name *server* and obtains its list of identifiers for available services, which it returns.

`disconnect_server` *server* sends a terminate signal to the server and closes the connection.

`thm_service` *server service* $[t_1, \ldots, t_n]$ invokes service *service* on server *server*. The λ-terms $t_1, \ldots, t_n$ are passed to the service as arguments. Note that `term` is Isabelle's type for λ-terms. The result of the computation, a λ-term assembled by the theorem template

[6] 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.

of the service, is turned into a theorem by Isabelle's oracle mechanism. The theorem is returned.

The function `thm_service` can be called directly, say to obtain the factorisation of $X^{15} - 1$ in the proof about Hamming codes. Alternatively, it can be called in a tactic that does a particular proof task. To prove the primitiveness of a polynomial is such a task. This is needed twice in the proof in Section 5.1 and once in Section 5.2. We provide a tactic `primitive_tac` that constructs a proof for primitive p, and automatically obtains the necessary lemmas from the computer algebra system: the proof for primitiveness follows the definition and has been outlined at the end of Section 5.1. It involves checking the irreducibility of the polynomial and a number of divisibility tests.

We turn this tactic into a *simplification procedure*. This is a function that maps a λ-term to a rewrite rule. Simplification procedures have been introduced as *conversions* by [Paulson, 1983]. The simplification procedure `F2PolyPrimitive_proc`, which we obtain, maps a polynomial p to the theorem primitive $p \equiv$ True. Simplification procedures can be added to Isabelle's rewriter, together with a pattern, here primitive p. During a rewrite, when the pattern matches the current redex, the simplification procedure in invoked, and if p can be proven irreducible, the rewrite rule generated by the procedure is applied. Isabelle's rewriter is integrated with its classical reasoner, and with the above simplification procedure we can prove the last part of Section 5.1, namely the subgoal

$$\begin{aligned}&(\exists C.\ \text{generator}\,15\,g\,C\ \wedge\ \text{Hamming}\,4\,C) = (\deg g = 4\ \wedge\ \text{primitive}\,g)\ \wedge\\ &g \in \{X^4 + X^3 + 1, X^4 + X + 1\}\\ &\Longrightarrow\ (\exists C.\ \text{generator}\,15\,g\,C \wedge \text{Hamming}\,4\,C)\end{aligned}$$

automatically.

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 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. Simplification procedures can be used to emulate the behaviour of that approach.

We have shown that the generation of translation-functions for objects from one system to another can be — at least partly — automated. This is due to the choice of λ-calculus as an intermediate language. In OpenMath [Dalmas *et al.*, 1997], a project that aims at general inter-platform communication in computer algebra, it seems impossible to generate phrase books automatically [Huuskonen, 1997].

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.

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".

A. Evaluation of Rational Functions

Fractions of polynomials are usually called rational functions. This is misleading. Formally, the function $g : x \mapsto \frac{(x-1)^2}{x^2-1}$ is not a rational function. Rational functions are obtained by constructing the *fraction field* of a ring of polynomials. A fraction field $\mathrm{FF}(R)$ can be constructed over any integral domain R.

The relation $\sim$ defined by $(a, b) \sim (c, d) \iff a \cdot d = b \cdot c$ is an equivalence relation over $R \times (R \setminus \{0\})$. The induced equivalence classes are called *fractions*. One writes $\frac{a}{b}$, where (a, b) is a representative of the class. Addition and multiplication are defined in the usual way:

$$\frac{a}{b} + \frac{c}{d} = \frac{a \cdot d + b \cdot c}{b \cdot d} \qquad \text{and} \qquad \frac{a}{b} \cdot \frac{c}{d} = \frac{a \cdot c}{b \cdot d}$$

Note that these definitions are independent of the chosen representatives. The set of equivalence classes is the fraction field $\mathrm{FF}(R)$. It is indeed a field, because any element $\frac{a}{b}$ with $a \neq 0$ has the inverse $\frac{b}{a}$.

How are $g : x \mapsto \frac{(x-1)^2}{x^2-1}$ and $\frac{(X-1)^2}{X^2-1} \in \mathrm{FF}(R[X])$ related? The evaluation homomorphism for polynomials can be lifted to rational functions. Let $\Phi_a \equiv \mathrm{EVAL}\, \phi\, a$ where ϕ is the identity function on R and $a \in R$. This is the homomorphism evaluating a polynomial at a. For $p, q \in R[X]$ one would like to define $\Phi_a(\frac{p}{q}) = \frac{\Phi_a(p)}{\Phi_a(q)}$, but this depends upon the representative for the fraction $\frac{p}{q}$. If $\Phi_a(q) \neq 0$ then $\frac{\Phi_a(p)}{\Phi_a(q)}$ is defined. Although (p, q) and $(p \cdot (X - a), q \cdot (X - a))$ are in the same equivalence class, $\frac{\Phi_a(p \cdot (X-a))}{\Phi_a(q \cdot (X-a))}$ is not defined. Therefore, for a rational function $f \in \mathrm{FF}(R[X])$ one defines

$$\Phi_a(f) = \frac{\Phi_a(p)}{\Phi_a(q)}, \text{for } p, q \in R[X] \text{ such that } f = \frac{p}{q} \text{ and } \Phi_a(q) \neq 0.$$

This definition is independent of the chosen polynomials p and q. If such polynomials do not exist then $\Phi_a(f)$ is undefined. The usual notation for $\Phi_a(f)$ is $f(a)$.

Let us return to the example in Section 2.2. The expression $\frac{(x-1)^2}{x^2-1}$ is undefined if $x = 1$, but $\left(\frac{(X-1)^2}{X^2-1}\right)(1) = \left(\frac{X-1}{X+1}\right)(1) = 0$. Axiom treats the argument of the solve-function as a rational function.

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, 1996a] Manual Bronstein. *Sumit Reference Manual*. Scientific Computation Institute, ETH Zürich, 1996. Available with the Sumit distribution at http://www.inria.fr/safir/WHOSWHO/Manuel.Bronstein/sumit.html.

[Bronstein, 1996b] 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, 1997] R. M. Corless and D. J. Jeffrey. The Turing factorization of a rectangular matrix. *ACM SIGSAM Bulletin*, 31(3):20–28, 1997.

[Dalmas *et al.*, 1997] Stéphane Dalmas, Marc Gaëtano, and Stephen Watt. An OpenMath 1.0 implementation. In Wolfgang W. Küchlin, editor, *ISSAC 97: Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation — July 21–23, 1997, Maui, Hawaii, USA*, pages 241–248. ACM Press, 1997.

[Davenport, 1998] James Davenport. Is computer algebra the same as computer mathematics? Talk given at the British Colloquium for Theoretical Computer Science (BCTCS), St. Andrews, UK, 31 March– 2 April 1998.

[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.

[Huuskonen, 1997] Taneli Huuskonen. Re: Description of OpenMath. Private communication, 20 August 1997.

[Jacobson, 1985] Nathan Jacobson. *Basic Algebra*, volume I. Freeman, 2nd edition, 1985.

[Jenks and Sutor, 1992] Richard D. Jenks and Robert S. Sutor. *AXIOM. The scientific computation system*. Numerical Algorithms Group, Ltd. and Springer-Verlag, Oxford and New York, 1992.

[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, 1983] Lawrence Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

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

[Paulson, 1996] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

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

[Théry, 1998] Laurent Théry. A certified version of Buchberger's algorithm. In C. Kirchner and H. Kirchner, editors, *Automated deduction, CADE-15: 15th International Conference on Automated Deduction, Lindau, Germany: proceedings*, number 1421 in Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence, pages 349–364. Springer-Verlag, 1998.

[Watt *et al.*, 1994] Stephen M. Watt, Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach, and Robert S. Sutor. *AXIOM Library Compiler User Guide*. The Numerical Algorithms Group Limited, Oxford, 1994.