Computer Algebra and Theorem Proving

Clemens Ballarin

Darwin College
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

Abstract

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. In work that has achieved more, the used methods do not scale up. They trust the computer algebra system either too much or too little.

Computer algebra systems are not as rigorous as many provers. They are not logically sound reasoning systems, but collections of algorithms. We classify soundness problems that occur in computer algebra systems. While many algorithms and their implementations are perfectly trustworthy, the semantics of symbols is often unclear and leads to errors. On the other hand, more robust approaches to interface external reasoners to provers are not always practical because the mathematical depth of proofs algorithms in computer algebra are based on can be enormous.

Our own approach takes both trustworthiness of the overall system and efficiency into account. It relies on using only reliable parts of a computer algebra system, which can be achieved by choosing a suitable library, and deriving specifications for these algorithms from their literature.

We design and implement an interface between the prover Isabelle and the computer algebra library Sumit and use it to prove non-trivial theorems from coding theory. This is based on the mechanisation of the algebraic theories of rings and polynomials. Coding theory is an area where proofs do have a substantial amount of computational content. Also, it is realistic to assume that the verification of an encoding or decoding device could be undertaken in, and indeed, be simplified by, such a system.

The reason why semantics of symbols is often unclear in current computer algebra systems is not mathematical difficulty, but the design of those systems. For Gaussian elimination we show how the soundness problem can be fixed by a small extension, and without losing efficiency. This is a prerequisite for the efficient use of the algorithm in a prover.

Preface

This technical report is intended to make my Ph.D. dissertation more easily accessible. The dissertation was submitted in June and the *viva voce* examination took place in September 1999. This revision includes some improvements that were suggested by my examiners.

My mechanisation of the theory of rings and polynomials may be useful to those who work on the mechanisation of mathematics from this area. It will become publicly available as part of the distribution of the theorem prover Isabelle. Please refer to the prover's home pages:

http://www.cl.cam.ac.uk/Research/HVG/Isabelle/
http://www4.informatik.tu-muenchen.de/~isabelle/

An "electronic version" of this document can be obtained from my home page:

http://iaks-www.ira.uka.de/iaks-calmet/ballarin/

Cambridge, October 1999

Acknowledgements

I would like to thank my supervisor Larry Paulson for steering me safely through the “adventure” Ph.D. Larry had always time to answer my questions, motivated me when I thought things were getting too difficult and had the right dose of criticism when I considered things simpler than they actually were. I was impressed by his efficiency, and I admire his ability to focus on that what is important.

Good research can only be done in a stimulating environment. The Computer Laboratory, despite its somewhat shabby buildings and furniture, proved to be an excellent place to be. I remember many interesting discussions over tea and would like to thank everyone in the Computer Laboratory — and beyond — who helped me with this project. I must not forget Jacques Calmet whose ceaseless efforts to set up international links within the scientific community were also to my benefit. I do not think I would have done my Ph.D. at such an excellent place otherwise.

The Studienstiftung des deutschen Volkes and the Cambridge European Trust have provided funding for my research. The Computer Laboratory, the Universität des Saarlandes and Darwin College helped with travel costs.

Last, but by no means least, I would like to thank my parents for their support. That I was living so far away has not always been easy for them.

To Martha and Maria

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The thesis of this work is that computer algebra can increase the reasoning power of theorem provers and verification systems. The enhanced system can be both trustworthy and efficient.

Both computer algebra and theorem-proving are forms of symbolic computation, and both have to do with mechanising mathematics on a computer. This is probably as close as the similarities get: both fields have different — almost disjoint — research- and user-communities. Before attempting a marriage between systems from both domains, it seems necessary to have a close look at how the fields evolved, the kinds of problems they try to solve, and the methods used.

1.1 Symbolic Computation

In the mid-fifties, scientists in artificial intelligence introduced a new view of computation: they saw computers as machines that manipulated symbols. Before that (see Newell, 1983), computers were seen as machines that manipulated numbers. Of course, everything could be encoded in numbers, and similarly, everything including numbers could be encoded in symbols. Before the mid-fifties, programs were classified as numerical versus non-numerical. The latter included all the things that processed data-types other than numbers — expressions, images, text and so forth. Symbolic manipulation presented a radical new view in computer science at a time when computers were seen as number manipulators.

Symbolic *mathematical* computation is concerned with the efficient manipulation of algebraic expressions. In the sixties, the focus of this discipline lay on symbolic integration (Moses, 1971). This was a stimulating problem also to artificial intelligence, because it was well-defined, and a human computer obtained heuristic solutions by means of a good deal of resources and intelligence. And only humans could compute integrals to non-trivial problems at that time. Further to that, celestial mechanics and high-energy particle physics posed computational problems that became too large to be solved by humans. So, symbolic integration was a challenge to problem-solving software of that time. But symbolic integration also triggered mathematical research, which led to Risch's decision procedure for integration (Risch, 1969). Within a decade, the view on symbolic integration had changed, and also the field of symbolic mathematical computation: *computer algebra* had become a discipline of its own, pursuing mathematical research with the aim of obtaining efficient algorithms to solve mathematical problems. Calmet and Campbell (1997) identify Risch's result as the moment when computer algebra abandoned problem-solving and became algorithmic.

Automatic theorem-proving took centre stage in artificial intelligence after the invention of the resolution principle by Robinson (1965). In the first, euphoric period it seemed that theorem-proving engines would sit at the heart of any general artificial intelligence system (Newell, 1983). Theorem-proving was applied to a range of tasks — for example, robot-planning. As a consequence of this success, theorem-proving got established as a fundamental category of activity distinct from other kinds of problem-solving. But only a few years later, the approach came up against its limitations. Provers were unable to handle any but trivial tasks. Getting to real mathematics — seen always as a major necessary hurdle — seemed as far away as ever. Theorem provers were organised as large homogeneous databases containing propositions. An inference engine would deduce new true statements from this and add them to the database. The inference engine treated expressions in its large, declarative database without regard for their semantics. Also it was not clear how provers could be given information about how to solve problems.

Automatic theorem provers are still organised in the same way today. Combinatorial explosion remains a hard problem. In the mid-seventies, theorem-proving shrank to a trickle (Newell, 1983) and planning languages emerged as a result of a shift to procedural encoding of knowledge. The Planner formalism (Hewitt, 1971) is an example. It has been applied in a robot to formulate and execute plans of action and for finding high-level descriptions of visual scenes. Recently, some progress has been made in the use of heuristics to reduce search spaces and McCune (1997) solved the Robbins conjecture. This is quite remarkable because the conjecture was an open problem for decades. McCune's attack on the Robbins problem took five CPU-weeks, the successful search eight days!

Another approach to computerised theorem-proving came from proof checkers. In the seventies de Bruijn's pioneering Automath system achieved remarkable success in the translation and validation of a standard mathematical text (see Nederpelt *et al.*, 1994). Proof checkers are merely providing the means of manipulating logical formulae and representing proofs. The proof is supplied by the human and checked only for its validity. Proof checkers therefore do not belong to the realm of artificial intelligence. In Automath, and also Stanford LCF,[1] a proof consisted of a sequence of steps, indexed by numbers, each following from previous steps by inference. In Edinburgh LCF (Gordon *et al.*, 1979), instead of indexing proofs by numbers, theorems were computed values and basic inference rules were procedures in the underlying programming language. The idea of representing inference rules as procedures was not new. Planner did that also. However, the idea to provide a programming language for the user of the proof checker was new. Inference rules now could be combined to new and more complicated ones, and arbitrary proof strategies could be implemented (Milner, 1985), turning the proof checker into a prover. The LCF system had been developed to reason about program semantics, and today LCF-style provers are successfully applied to verify both hardware and software.

1.2 Computer Algebra

Computer algebra systems have evolved into large collections of algebraic algorithms. Today, a typical computer algebra system consists of a small kernel that implements efficient arithmetic, together with a purpose-built imperative programming language, in which most of the system's library — its collection of algebraic algorithms — is written. The user interacts with the computer algebra system through a command-response-loop, as in any other interactive programming environment, and can call routines from the library to perform computations. The user can also implement new algorithms in the system's language and thus add to the library.

Computer algebra systems provide algorithms for numerous domains. Most prominent, of course, are symbolic differentiation and integration. Although Risch's method is a solution to the problem of indefinite integration, the problem of symbolic definite integration remains unsolved. Packages

[1] A proof checker for the Logic of Computable Functions, a logic due to Scott. Scott's paper (Scott, 1993) was first written in 1969 but for a long time only circulated privately.

for linear algebra provide means for matrix manipulation, various variants of the Gaussian algorithm, and can compute the Jordan normal form of a matrix, to give only a few examples. Buchberger's algorithm dominates the domain of polynomial equation systems. Computer algebra systems also provide functionality for geometry, statistics, combinatorics, number theory and many more domains. Some specialised computer algebra systems are particularly strong at group theory.

Most research in computer algebra today is concerned with the development and refinement of algorithms. A particular focus is currently on the symbolic solution of differential equations. This kind of research is constructive, applied mathematics. In spirit, it follows the example set by the success of Risch's algorithm.

A phenomenon which has to be taken into account while designing and implementing such algorithms, and which seems to be unique to computer algebra, is that the size of expressions can swell quite dramatically during a computation, even if the result is comparatively small. This *intermediate expression swell* happens, because computations are exact and no rounding occurs. An example, which is probably familiar to the reader, is solving a system of linear equations, where, after the first few transformations, the fractions that are introduced become bigger and bigger, and most time is spent on computing common denominators.

In the case of fractions of integers, it is usually of advantage to cancel fractions immediately, leading to a *canonical* representation. In other domains, canonical representations may not be efficient: obtaining a rational denominator for expressions over simple radicals leads to a canonical representation, if the radicals are independent. But

$$\begin{aligned}\frac{1}{\sqrt{2}+\sqrt{3}+\sqrt{5}+\sqrt{7}} &= \\ &\frac{1}{215}(22\sqrt{3}\sqrt{5}\sqrt{7} - 34\sqrt{2}\sqrt{5}\sqrt{7} - 50\sqrt{2}\sqrt{3}\sqrt{7} + 135\sqrt{7} \\ &\qquad + 62\sqrt{2}\sqrt{3}\sqrt{5} - 133\sqrt{5} - 145\sqrt{3} + 185\sqrt{2}),\end{aligned}$$

which is an exponential growth in size (Davenport *et al.*, 1993). For more general domains the problem of data-representation remains unsolved, and often an efficient representation does not depend only on the domain but also on the problem and the used algorithm.

How much mathematical domain structure should be made explicit is seen as an important design question by Davenport (1990). Early systems just provided a small and fixed number of domains. The drive to model structure in a more abstract way arose from users' and implementors' needs: modular arithmetic, for example, is needed in factorisation algorithms. So it is already available somewhere in the system, and it could as well be made explicit for use in other parts of the library and by the user. It then seems only a small step to providing generic implementations for routines that are shared by several domains. This has been done with some success in the Axiom system (Jenks and Sutor, 1992), using a tailor-made, object-oriented type system. The generality available in this system is enormous, and, because of the large number of domains, Axiom is more difficult to use than other computer algebra systems. Also, generality easily gets into conflict with the need for having different representations of the same data in different applications.

The desire for speed in this discipline is so strong that discussions of storage management and garbage collection can be found on conferences on computer algebra (Davenport, 1990; Norman and Fitch, 1996).

Another issue at the system level, which receives a good deal of attention at present, is interfacing with other systems. Until a few years ago, this meant the generation of Fortran code, because high-quality numerical libraries are written in Fortran, and if large formulae obtained using a computer algebra system need to be evaluated that is how it should be done. Although the generation of Fortran code is still important, interfacing technology has moved on. For example, the MathLink interface (Wolfram, 1996, Section 2.12) is an application programming interface for Mathematica. It permits use of external programs from within Mathematica — for example, visualisation tools — and also calling Mathematica from other applications — such as spreadsheets.

Kajler (1993) proposes an "integrated scientific software environment", which includes computer algebra systems, numerical libraries, graphical plotting engines and scientific text processing tools. Such an environment should ease interactions and exchanges between the different tools. It could also integrate different computer algebra systems and allow the user to reconfigure the system while it is running, if that is more efficient for a particular step in the computation.

This is not yet generally possible because existing interfaces of computer algebra systems are not compatible and neither are the data-formats they use. But standards for the exchange of mathematical data are emerging: MathML (Ion *et al.*, 1998) will provide a standard for the rendering of mathematical formulae on the World-Wide Web and could also be useful as a standard for scientific text processing. The OpenMath project (Dalmas *et al.*, 1997) goes beyond that and aims at providing a standard for the exchange of objects between mathematical computation programs.

1.3 Theorem Proving

One of the motives in artificial intelligence for the development of theorem provers has always been the dream to automate the activity of mathematicians: the discovery and understanding of mathematical structure. This is still mainly a dream, with probably the only exception being the solution of the Robbins problem (McCune, 1997). This proof, found by a computer, is readable by a human.

Theorem provers are usually based on a logic, like first-order logic, higher-order logic or a version of set theory, which provides a framework for the formalisation of (parts of) mathematics. Depending on the logic and the application, a prover may implement a single procedure, whose search behaviour can be controlled by the user by setting certain parameters. Otter (McCune, 1994) is an example for such a prover for first-order logic, based on the resolution principle. Provers that support more expressive logics tend to provide a framework in which the required proof procedures can be implemented, though not necessarily by the user, because general decision procedures do not exist. The LCF-style provers HOL (Gordon and Melham, 1993) and Isabelle (Paulson, 1994) are examples of such frameworks that can be extended safely by the user.

While formalisation is probably not the right way to go to discover new theorems, it improves the rigour of the mathematical foundations, and at least some mathematicians seem to consider that a useful activity. In the meantime, the Mizar project (Rudnicki, 1992) has accumulated a vast library of machine-checked, formalised mathematics (Association of Mizar Users, 1989), and has shown that this is feasible. Proofs in the Mizar system are driven by proof scripts that are supplied by the user and that are inspired by a natural (*i.e.,* human-readable) style of proof. The prover tries to expand proof-steps in the script to steps that are primitive in the logic. When that fails, the proof script has to be refined manually.

The areas where automatic theorem-proving really has been a success are where proofs are tedious, full of details that get easily confused by humans, but structurally homogeneous. Proofs in hardware and software verification are of this kind. Of course, here proofs are not found fully automatically either. *Interactive proof development systems* are used that are guided by the user and find routine parts of proofs automatically. They also rigorously check all the steps — manual and machine-generated ones — of a proof.

1.4 Computational Strength of Provers

Such proof assistants have been successfully applied to verify design, for example a network switching device (Curzon, 1994), and security properties of the Internet-protocol TLS (Paulson, 1997a), neither of which was designed with formal verification in mind. The verification of programs is much harder. It has been found that their verification is more feasible if software and correctness-proof are developed together. Verification can also be used to check whether the specification of software is consistent. Research in this area aims at integrating theorem-proving with the software design process in order to make the use of formal methods feasible.

Using a variant of the prover HOL, Harrison (1997) has demonstrated that it is possible to formally verify numeric algorithms for transcendental functions. The algorithm that was verified computes an approximation for the exponential function, which is suitable for single-precision binary floating-point arithmetic, and the main difficulty was to show that the approximation is sufficiently close to the actual function. The proof requires many computations with arbitrary-precision integers, which are done within the logical formalism of the prover. This is indeed very expensive. In his report, Harrison remarks that the time for building the prover and then running all the proofs was twelve hours. When the integer arithmetic of the underlying programming environment was used (and trusted), this time shrank to two hours. This enormous difference is not surprising, because the representation of numbers is very inefficient: about 100 bits are used to store one bit, and more significantly, computations are permanently accompanied by correctness proofs.

Another example, which highlights the current limitations of provers has been suggested by Harrison. The equation

$$
\begin{aligned}
&({x_1}^2 + {x_2}^2 + {x_3}^2 + {x_4}^2) \cdot ({y_1}^2 + {y_2}^2 + {y_3}^2 + {y_4}^2) \\
&\quad = (x_1 \cdot y_1 - x_2 \cdot y_2 - x_3 \cdot y_3 - x_4 \cdot y_4)^2 \\
&\quad + (x_1 \cdot y_2 + x_2 \cdot y_1 + x_3 \cdot y_4 - x_4 \cdot y_3)^2 \\
&\quad + (x_1 \cdot y_3 - x_2 \cdot y_4 + x_3 \cdot y_1 + x_4 \cdot y_2)^2 \\
&\quad + (x_1 \cdot y_4 + x_2 \cdot y_3 - x_3 \cdot y_2 + x_4 \cdot y_1)^2
\end{aligned}
$$

appears in the proof of the theorem that every natural number can be represented as the sum of the squares of four prime numbers. Isabelle takes 130 seconds to check this equation, whereas the computer algebra system Maple achieves the same in 50 milliseconds. Both times were taken on the same machine. Again, the difference is not surprising. Associative-commutative rewriting on the expanded products does basically bubble-sort with respect to the term order, while Maple obviously uses a more sophisticated algorithm. On the other hand, Isabelle's simplifier is much more flexible and can be configured easily to different equational theories.

The facility to compute with numbers and symbolic expressions in a prover seems quite attractive when reasoning in algebraic domains. And these domains are likely to be needed in future verification work. Harrison's case study about the exponential function points in that direction, and in fact he uses a computer algebra system to guide proof search already. But there is more to computer algebra than computing sums and products: algorithms that solve equation systems, compute factorisations or do symbolic integration over various domains are a main contribution of computer algebra and required a lot of research. When adding computer algebra to a prover, one would like to have all those as well.

In principle, it is of course possible to implement algebraic algorithms directly in a prover, so that it operates directly on the prover's logical calculus. Regardless whether this would be efficient or not, it is not desirable to do so, because many of the algorithms in computer algebra are very sophisticated and require a lot of expertise to obtain an efficient and correct implementation.

1.5 Suitable Domains of Application

The next question to ask is whether there are suitable domains of application, where such an integration could be useful. It is usually assumed that there are plenty of them, but it has not been easy to identify domains where proofs have large computational content, and where the mechanisation of the domain specific knowledge is in reach of provers. "Within reach of provers" means, of course, that the mechanisation can be achieved within a Ph.D. project, because the utility of the integration should be shown convincingly with a case study.

1.5.1 Analysis

When browsing through the proofs in a textbook in mathematical analysis (Apostol, 1974), one mainly finds two kinds of calculations: extensive estimations, for which only a few algorithms in computer algebra systems are implemented, and simple transformations. Typical examples of transformations are $\frac{\varepsilon}{2} + \frac{\varepsilon}{2} = \varepsilon$ in the proof of the convergence of Cauchy sequences in Euclidean spaces, and $x \cdot y - a \cdot b = (x - a) \cdot (y - b) + a \cdot (y - b) + b \cdot (x - a)$

is used in the proof that the limit of a product is equal to the product of the limits. But these kind of calculations can be dealt with easily by rewriting. The textbook does not contain proofs whose mechanisation would be substantially simplified by the use of computer algebra.

1.5.2 Algebra

Analysing the structure of finite groups involves factorisation of the group order and application of Sylow's theorems (Jacobson, 1985), *i.e.* some computation paired with reasoning. But for small groups there exist tables of their structure and those tables are also incorporated as databases into computer algebra systems that support group theory, like Magma (Butler and Cannon, 1989).

The study of groups of larger order, where the use of a computer algebra becomes more interesting, is a topic representation theory (Jacobson, 1989). Here the structure of a group is described in terms of characters, which are morphisms between groups, usually between the group being studied and the multiplicative group of the complex numbers. Studying the structure of a group involves solving equation systems which are obtained by certain relations. Character theory might well be an interesting domain of application, but it requires a highly advanced knowledge of algebra and proofs seem quite involved, so that it is not suitable for a case study.

1.5.3 Generating Function Techniques

Generating functions are used for the manipulation of formal power series. Depending on the series, the generating function can be a polynomial, or it may only be known that it lies in some space of continuous functions. According to the properties of the function, certain transformations can be applied. Manipulation of generating functions thus involves both symbolic computation — say, integration or differentiation of the function — and reasoning, in order to show that premises for the applications of those operations hold.

Generating-function techniques are applied in the analysis of the complexity of algorithms (Purdom and Brown, 1985; Hofri, 1995). The application domain utilises quite deep knowledge from algebra, complex analysis and combinatorics. Therefore only a very small part of this could be formalised within a Ph.D. project. There is a package for generating functions available for Maple (Salvy and Zimmermann, 1994).

1.5.4 Coding Theory

This discipline studies the transmission of information over noisy communication channels. 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 (Hoffman *et al.*, 1991).

A class of codes for which encoding and decoding can be easily implemented in hardware, and high bandwidths can therefore be achieved, is that of algebraic codes. The existence of such codes is related to the existence of generator polynomials. A generator polynomial basically determines a code. Such polynomials usually do not have a common shape, but algorithms are needed and have been designed to compute them, for example Berlekamp's algorithm (Geddes *et al.*, 1992, Chapter 8) for polynomial factorisation. Therefore, this theory has substantial computational content.

1.6 Outline of the Dissertation

We want to show that computer algebra can increase the reasoning power of theorem provers and verification systems, while the enhanced system can be both trustworthy and efficient. We demonstrate this by extending a theorem prover and use it to prove non-trivial theorems from coding theory. Coding theory is an area where proofs do have a substantial amount of computational content. Also, it is realistic to assume that the verification of an encoding or decoding device could be undertaken in, and indeed, be simplified by, such a system.

As prover we choose the system Isabelle (Paulson, 1994). It has been used in verification, and it is also suitable for the formalisation of mathematics, as it occurs in our domain of application. Also, it is the system which has been designed by my supervisor, and therefore knowledge about its internals is easily accessible.

In theorem-proving, and for an application in verification in particular, it is important that the overall system is trustworthy. Computer algebra systems are not as rigorous as many provers. Therefore, in Chapter 2, we classify soundness problems that occur in computer algebra systems. While many algorithms and their implementations are perfectly trustworthy, it turns out that the semantics of symbols is unclear and leads to errors. This mainly affects the simplification of expressions. Possible ways of extending a prover by an external reasoner are examined, in our case a computer algebra component. These ways to extend the prover differ in the extent to which errors of the external component jeopardise the integrity of the overall system. We discuss their suitability for coupling a computer algebra component to a prover and review earlier experiments in this field in the light of our results on both the soundness of computer algebra and the suitability of their approach to coupling. Finally, in Chapter 2, we outline our own approach, which takes both trustworthiness of the overall system and efficiency into account. It relies on using only reliable parts of a computer algebra system, which can be achieved by choosing a suitable library, and deriving specifications for these algorithms from their literature.

Many issues of interfacing systems, like the cooperation of concurrent processes or data-exchange formats, have already been addressed by research in computer algebra. In Chapter 3 we review this and find that the logical issues of an integration are neglected. Even work taking semantics into account is not rigorous enough. No formal specifications for these interfaces can be given. We design an interface to the computer algebra library Sumit (Bronstein, 1996), and describe its implementation using Isabelle's oracle mechanism, an interface that allows adding external reasoners to Isabelle. The interface does not import all of Sumit's algorithms to Isabelle at once. Rather, if more mathematics is formalised in Isabelle, specifications for more algorithms that are suitable can be added. It is important to note that algorithms cannot be specified unless the necessary mathematics has been formalised.

The necessity to formalise mathematics leads naturally to the next step in our project. Our domain of application, algebraic coding theory, is based on the theory of rings and polynomials. A mechanisation of the algebraic hierarchy of rings, and of polynomials had to be provided. This is presented in Chapter 4. Chapter 5 then shows that our approach is feasible. After introducing the necessary background of coding theory and its formalisation in Isabelle, we present two proofs of the existence of certain codes, and how they are formalised in our extension of Isabelle. The proofs depend on contributions of computer algebra, and we show that the theorems imported from Sumit could not have been verified easily in the prover alone.

In Chapter 6 we return to one of the systematic soundness problems in computer algebra, the semantics of symbols. This can only be captured correctly by realising that a symbol in an expression can represent, among others, either a variable (which is a logical entity), or an indeterminate (which is an algebraic entity). Mostly for the sake of efficiency, this difference is ignored in computer algebra. For the sake of soundness this cannot be done in a prover, and for the example of the Gaussian algorithm, applied over symbolic expressions, we suggest modifications that allow its sound use in a prover, and that are also useful to users of computer algebra systems. Finally, in Chapter 7, we summarise our contributions.

Chapter 2

Making the Integration Trustworthy

Automatic theorem provers are used in verification to assist humans to prove theorems. It is not the case that these theorems could not be proved by humans because of their mathematical depth. In fact, theorems that are too deep to prove by humans hardly stand a chance to be proved automatically. Rather, the theorems that occur in verification are large, typically with long lists of side-conditions, and the advantage of mechanised proof is that the book-keeping is done automatically. Correctness of the prover is highly important. If the prover was not correct, false statements could be proved, and in many cases this would not be noticed, simply because of their size.

One the other hand, provers like any other software are likely to contain some errors. Even if errors affect the correctness, the use of a prover can still be beneficial in verification. With the current state of the art in formal methods it is not feasible to verify software completely, but only to obtain proofs for some properties of a design. This means that errors in other aspects of the design will not be found.

Current computer algebra systems are not entirely correct. This is not surprising, because they are software products. Some of their errors have systematic reasons, though, as we shall see in the next section. The aim of this work is to enhance provers by computer algebra to make them more useful in verification. Because correctness is an important issue, the soundness problems of computer algebra systems need to be addressed, and a suitable way of interfacing needs to be found to ensure that the overall system remains trustworthy. On the other hand, this approach must not be too prohibitive, or the extension will not be useful.

As they are crucial for the design of the interface, errors in computer algebra systems are examined in this chapter. They are classified in three categories. After a review of approaches to extending provers in general and of earlier attempts to integrating provers with computer algebra systems, a new approach is described, which is based on the classification of soundness problems.

2.1 Soundness in Computer Algebra

Computer algebra systems have been designed as tools that perform complicated algebraic computations efficiently. Their soundness or, as some authors might prefer to say, unsoundness has become a focus. Stoutemyer (1991) presents a systematic discussion of examples of such errors. They are classified according to mathematical properties. This is helpful, but what is needed for the present work is an analysis of the underlying design problems which lead to these errors. Harrison (1996) and Homann (1997) present more examples of errors, which they found made it difficult to integrate provers and computer algebra systems. A number of errors in symbolic integration are reported by Adams *et al.* (1999).

2.1.1 Misleadingly Uniform Interface

Computer algebra systems present collections of algorithms through a uniform user interface. This gives the impression that objects have some sort of global semantics throughout the system. This is, however, not the case.

For a simple example note that many computer algebra systems do not distinguish between natural numbers and integers. From a computational point of view it is reasonable to use the same representation or data-type. Storing a natural number as an integer only requires an additional sign bit, but providing a separate type for the naturals would require to provide a set of operations on that type as well. In contrast to that, in a reasoner the difference is significant: integers form a ring and the induction principle is the fundamental property of natural numbers, but there is no induction principle for integers and the natural numbers do not form a ring. So, for example, both coefficient and exponent of the monomial aX^n can be represented by integers in the computer algebra system, whereas in the prover, a should be represented by an integer and n by a natural number.

Another instance of this problem occurs with the solve-command. On a number of computer algebra systems, including Maple and Axiom,

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

yields $x = 1$. But this is not a solution to the equation, because for $x = 1$ the denominator vanishes. If the computation is split in two steps, in Axiom, the type system gives a hint on the reason. See Figure 2.1.

```
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 2.1: 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. In Maple, which does not usually automatically simplify expressions, the solve-function even forces the cancellation!

In this case, the computer algebra system returns the wrong result, because the two operations *cancel* and *solve* assume different semantics for the same object. Combining them in this way is not sound. Not even Axiom's type system prevents the problem. That the semantics of symbols is unclear in most computer algebra systems has been pointed out by Stoutemyer (1991). A more precise analysis of the problem is presented in Chapter 6 towards the end of this thesis.

In general, computer algebra systems present a misleadingly uniform interface to collections of algorithms. An object that is used with a particular meaning in one algorithm may 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.

2.1.2 The Specialisation Problem

Computer algebra systems have only limited capabilities for handling side conditions or case-splits. An example where this causes problems is the integral $\int x^n\,dx$ (Homann, 1997). Computer algebra systems return $\frac{x^{n+1}}{n+1}$. Substituting $n = -1$ yields an undefined term, while the solution of the integral becomes $\ln x$. Computing the rank of a matrix with symbolic entries poses a similar problem, because the rank may depend on variable entries in the matrix.

This class of problems is known as *specialisation problem*: evaluation and substitution do not commute. It also occurs in fractions and exponentiation, again discussed by Stoutemyer (1991): $\frac{x}{x}$ is often evaluated to 1, whereas evaluating $\frac{0}{0}$ gives an error message. Similarly, x^0 is simplified to 1, 0^x to 0 and simplifying 0^0 again raises an error. The problem is well known, but hardly ever referred to in the literature, see (Corless and Jeffrey, 1997). The problem with the solve-function, discussed in the previous section, can also be seen as an instance of the specialisation problem: the system pretends to compute with polynomials because it does not handle side-conditions.

In the design of current computer algebra systems the problem is mainly ignored: these systems have only limited mechanisms to attach information to symbols. In the integration example above, information whether $n = -1$ or not could be used by the integration procedure to make the correct choice. Alternatively, the procedure could flag a side-condition $n \neq -1$ when returning the result $\frac{x^{n+1}}{n+1}$. Mechanisms that handle such assumptions are provided by most multi-purpose computer algebra systems, but are only used by a limited number of algorithms, because they have not been part of the initial design of those systems (Fateman, 1998).[1] Furthermore it is not always possible to decide for a given expression whether it is, say, zero.

The computer algebra system Macsyma is better at handling degenerate cases. During a computation, if for example a cancellation is to be performed, the system queries the user if a symbolic expression is zero or not. But if the symbol in question has been generated by the system, the user would not know how to answer that question. Also, computer algebra systems are designed to manipulate large formulae, and a user would probably, rather unnerved, abort a computation after the tenth query. Macsyma is one of the first large computer algebra systems, and it seems that querying the user has been abandoned in the design of later systems.

[1]In particular, these mechanisms are not used to prevent the kind of simplifications described by Stoutemyer. They have been introduced to fix some, but not all problems of definite integration.

2.1.3 Algorithms that are *ad hoc*

Many of the algorithms that are implemented in computer algebra systems rest on mathematical theory and proofs for their correctness have been published. Examples for these are factorisation algorithms for polynomials (Geddes *et al.*, 1992, Chapter 8), Gaussian elimination (Bareiss, 1968, for example), and Risch's method for indefinite integration (Risch, 1969).

The design of other algorithms is less rigorous. Symbolic computation over the complex numbers, for example, is intricate. Solving even an equation as simple as $y = z^w$ for w turns out to be difficult over the domain of complex numbers, and Fateman (1996b) devotes three pages to a systematic solution. The reason is that the exponential function is only bijective on an extension of the complex plane: Riemann surfaces. A treatment within the complex numbers requires case splits or recording suitable assumptions. As we have seen in the previous paragraph, current computer algebra systems have only limited capabilities for this. A discussion of possible extensions to computer algebra systems to deal with computation over complex numbers in a satisfactory way can be found in Corless and Jeffrey (1996). The correctness of simplification algorithms in current computer algebra systems is generally doubtful if they operate over these domains.

There is another problem with simplification algorithms. Computer algebra systems usually accept undefined or user-defined symbols, but do not treat them in a well-defined manner. Undefined symbols are often regarded as indeterminates of polynomials. In many cases it does not matter if, in spite of this, the symbol has some algebraic relation. In some cases it does: this is another instance of the specialisation problem. In a computer algebra system, a symbol can be defined by the user, either by supplying a function definition, if the symbol represents a function, or by asserting a set of rules. The available mechanisms vary among computer algebra systems. These rules or functions modify the simplification process for expressions. The actual effect depends not only on the simplification strategy, but also on definitions of other symbols that may be present. These mechanisms are *ad hoc* and likely to introduce unsoundness.

Definite integration — computing the area under a curve defined by a function — is another example. Given a function f, this area is usually obtained by computing the anti-derivative F and then evaluating $\int_a^b f = Fb - Fa$. This is based on the fundamental theorem of integral calculus and, depending on the definition of integral used, f must be "well behaved". It may, for example, not contain singularities. Computer algebra systems currently ignore this problem. Even worse, Risch's algorithm computes the anti-derivative in terms of differential algebra, and the integral therefore may contain additional discontinuities. See Davenport (1998). Research to improve algorithms for definite integration with the aid of theorem provers is underway. For first, promising results see Adams *et al.* (1999).

We call algorithms like Gaussian elimination or Risch's method for indefinite integration, which are based on mathematical theory, *sound*. Other algorithms we call *ad hoc*. Many *ad hoc* algorithms are unsound, because their designers did not go into the trouble of resolving issues that might be resolved by side-conditions. Calmet and Campbell (1997, Section 2) give a historic perspective on the distinction of sound and *ad hoc* algorithms. It is, of course, not straightforward to judge whether the algorithms used in a given implementation are sound or *ad hoc*. Depending on the quality of documentation it might be necessary to inspect the code and compare it to the literature.

All three kinds of problems — "uniform interface", the limited capabilities for handling side-conditions and case-splits, and *ad hoc* algorithms — make it difficult to use a computer algebra system as a sound, or *trustworthy*, black-box reasoning component. This is only possible, if at all, after analysing its true semantics carefully.

2.2 Trust and Certificates

Insufficient reasoning capabilities and unclear semantics make a bad basis for a trustworthy reasoning component. Therefore it has been suggested not to trust computer algebra systems, but to verify their results. In general, there are a number of approaches to integrating external reasoners, and they offer different degrees of rigour. They are reviewed in this section. Some of them have already been applied to integrating computer algebra into provers.

2.2.1 Proof Reconstruction

The most rigorous way to maintain soundness with the integration of an external tool is not to trust it at all, but to reconstruct a proof of its result in the logical calculus of the prover. This approach has been particularly popular with LCF-style provers, which encapsulate the correctness-critical core part of the prover in an abstract data-type. Here proof procedures can be programmed as *tactics* to operate directly on the prover's calculus. This approach was designed to make the prover programmable and sound at the same time (Milner, 1985).

Proof reconstruction is not necessarily inefficient. It is certainly faster to search for a proof with an external tool using specialised data-structures than performing it in the prover's calculus. For some mathematical properties *certificates* exist. A certificate provides information that makes it easy to check that an object has the property. Hence it is a trustworthy guarantee. For example, Pratt (1975) has shown that such certificates exist for prime numbers. A different issue is whether such certificates can be computed efficiently. In the context of proof reconstruction it is more practical to record the essential information that is required to reconstruct the proof without search. We call this a *trace*. The structure of the trace depends on the problem, as we will see later.

There is an analogy between finding a proof and checking it, and non-deterministic and deterministic programs (see Harrison, 1996, Chapter 6.2). For example, if a problem is of time-complexity NP a solution can be checked in P, if the non-deterministic choices made by the algorithm that solves the problem in NP are known. These can be determined by deterministic search in exponential time. The collection of choice-points are the trace, which is of polynomial size.

In practice, proof reconstruction has been successful as an approach, where the search space for proofs is large. For example, Paulson (1998) describes a tableau prover that outperforms Isabelle's older tactics, which do search in Isabelle's calculus. An observation by Bundy *et al.* (1991) that reconstructing a proof in the Oyster-Clam system is much slower than finding it, seems to be due to inefficiencies in Oyster or its logic.

Proof reconstruction does not offer gains where the proof is costly and does not contain search at all. This is the case, for example, for arithmetic. In the verification of a numerical algorithm for the exponential function by Harrison (1997, Section 9) 83% of the proof time were computations with natural numbers, which were done in the prover's calculus.

2.2.2 Meta Theoretical Extension of the Prover

The correctness of the prover can also be maintained over extensions by the user, by demanding that a procedure needs to be formally verified before it is accepted as a new proof procedure. A "logical prototype" of such a system has been described by Davis and Schwartz (1979). Following this idea Boyer and Moore (1981) extended their prover by a procedure that cancels summands in equations. It improved the power and performance of the prover.

In practice the external procedures will be implemented in some programming language, and the formal verification will be carried out with respect to its semantics. If one insists on being completely formal, this leads to another complication: a formalisation of the semantics of the programming language is required. If the semantics is known the most feasible approach to proving the procedure correct is probably to show that it is equivalent to a procedure encoded using only primitive rules of the prover (Slind, 1993).

How feasible is this for computer algebra? The verification of algorithms in computer algebra seems particularly difficult. The formalised correctness proof for Buchberger's algorithm by Théry (1998) requires 4700 lines of definitions and proof scripts. This does not include the development of the required mathematical domains, like polynomials *etc.* The verified version of the algorithm is simple by comparison with practical implementations of Buchberger's algorithm and other algorithms from computer algebra.

In general, the effort of verifying an algorithm does not only depend on the size of its pseudo code. More important is the difficulty of the mathematical argument for the correctness of the algorithm. For this reason it seems that algorithms in computer algebra are much harder to verify than algorithms in other software systems. In the case of the verification of Buchberger's algorithms the proof of termination needed a variant of Dixon's lemma.

A rather extreme example for a such difficulty is the exploitation of Feit and Thompson's theorem in the computer algebra system GAP (Schönert, 1997). This theorem states that any finite group of odd order is solvable. It is used in GAP as a quick test for solvability of groups before more elaborate methods are applied. The code for this test occupies two lines of code; the informal, published proof for this theorem (Feit and Thompson, 1963) is more than 250 pages long, fills an entire issue of the Pacific Journal of Mathematics and itself depends on earlier, complicated group-theoretic results. Subsequent work may have shortened the proof by 50 pages or so, but a substantially shorter proof is not known.[2]

With current technology it is not feasible to verify a large collection of algebraic algorithms. As the power of provers advances this may become feasible in future. Extracting the implementation of an algorithm from its correctness proof is an attractive way to ensure its correctness. This was done by Théry for Buchberger's algorithm. No information on the performance of the code is given, but Théry (1998) suggests that their memory efficiency is bad. This poses a particular problem in computer algebra, be-

[2]Thanks to Steve Linton for providing this information.

cause of the phenomenon of intermediate expression swell. Eventually, more and more mathematics will be formalised and more algebraic algorithms will be in the scope of formal verification. Large databases of formalised mathematics like the Mizar Mathematical Library (Association of Mizar Users, 1989), developed with the Mizar system (Rudnicki, 1992) will be important here. But the example of Feit and Thompson's theorem shows that for some of the algorithms we might have to wait for a very long time till they get verified.

2.2.3 Trust the External Reasoner

The third approach to interfacing the external reasoner is based on trusting it. Its results are translated into theorems in the prover's calculus. These theorems are specifications for particular runs of the external reasoner. It is reasonable to trust the external reasoner if it is reliable, because this is the most efficient way of using its result. One has to be aware, though, that the translation from the reasoner's to the prover's language needs to be done carefully as well, because that may introduce errors, too.

If there are doubts about the trustworthiness of the external reasoner, theorems that depend on external results can be marked, so that the results, say of a verification, can be used with the appropriate care. A simple way of achieving this in the prover HOL was suggested by Gordon (see Harrison and Théry, 1994), namely to define a logical constant *Maple* that is equivalent to false. A statement imported from the computer algebra system Maple would produce a theorem under the assumption *Maple* — for instance, $Maple \vdash A = B$. This ensures that the consistency of HOL is not corrupted: one can prove anything from falsity. Further, any theorem that is derived from this theorem inherits the assumption *Maple*. Theorems that depend on the external reasoner can thus be recognised from ones that do not. More sophisticated means for achieving this are available in provers today. *Tags* are used in HOL to annotate theorems (Slind, 1998) and Isabelle can store dependencies between theorems in *derivations* (Paulson, 1997b).

2.3 Review of Earlier Work

A number of experiments to integrate provers and computer algebra have been undertaken by various groups within the last few years. The following sections present an analysis of their suitability based on the soundness of computer algebra systems and the available approaches to integrate them.

2.3.1 Analytica

Analytica (Clarke and Zhao, 1993, 1994) is an experimental automatic theorem prover. It is written in Mathematica and can therefore directly access the latter's simplification capabilities for symbolic expressions. It is able to prove a large class of theorems in analysis. The most outstanding one is probably the proof about Weierstraß's function

$$f(x) = \sum_{n=0}^{\infty} b^n \cdot \cos(a^n \cdot \pi \cdot x).$$

The function is defined and continuous if a is an odd integer and $0 < b < 1$. When $a \cdot b > 1 + 3 \cdot \pi/2$ its derivative does not exist for any value of x. Analytica is able to prove both properties with little human intervention.

The authors of Analytica are aware that "perhaps, the most serious problem in building a theorem prover like Analytica is the soundness of the underlying symbolic computation system" (Clarke and Zhao, 1993). Despite of Mathematica's abilities to deal with side-conditions, the specialisation problem occurs also in this system. The authors of Analytica suggest to "develop the theorem prover and the symbolic computation system together so that each simplification step can be rigorously justified." It would be an enormous task to do so. For smaller domains it can be practical: Beeson (1995) wrote a special symbolic computation package for his education software Mathpert, to ensure its smooth integration with the reasoner. Mathpert is a calculus-trainer that permits students only to do sound steps when solving a problem. This is ensured by the reasoner of the system. Implementing a computer algebra package from scratch would not only be an enormous effort, it would also mean to discard a lot of good implementations of sound algorithms. Neither would the problem of *ad hoc* algorithms be solved.

What Analytica lacks is a formal description of what its mathematical entities mean. Analytica takes over the semantics implicitly given by Mathematica, and it is questionable, as pointed out by Farmer *et al.* (1993, Section 6) whether formal semantics underlies the algorithms of Mathematica. Strictly speaking, Analytica does not produce a proof, but a *proof plan.* It is the hope of the designers of Analytica that this plan can be expanded to a formal proof, but the attempt to do so is not made.

2.3.2 Interface between Isabelle and Maple

In this experiment (Ballarin, 1994; Ballarin *et al.*, 1995) Maple is made accessible from within Isabelle. The results of the computer algebra system are trusted if certain preconditions can be proved. The translation of terms between the two systems is done utilising Isabelle's powerful mechanisms for parsing and pretty-printing.

The external system is closely integrated into Isabelle's simplifier, the prover's rewrite engine. Which terms are passed to Maple is controlled by *evaluation rules.* These are of the form

$$P_1 \wedge \ldots \wedge P_n \implies t \equiv t'.$$

During rewriting, if t matches with the current redex, and $P_1, \ldots P_n$ can be proved, the term t is evaluated by Maple obtaining t'. The redex is replaced by t' and rewriting proceeds. The mechanism for checking preconditions is able to catch divisions by zero and the like in expressions, but is not powerful enough to handle more complicated situations, where the preconditions are generated during the computation, for example the divisions that occur in the Gaussian algorithm.

The integration of the computer algebra system into the rewriter is convenient if expressions containing algebraic formulae need to be simplified. In other circumstances too many manual instantiations were necessary to guide a proof (Ballarin, 1994, Section 6.3), and the use of the computer algebra system from within Isabelle seems too restrictive.

A further criticism of this work is that access to Maple is through its user interface and evaluation rules control which functionality of Maple is visible from the prover. This is not flexible enough: in order to avoid *ad hoc* algorithms in Maple, many sound algorithms cannot be used. The problem that the semantics of expressions may differ from algorithm to algorithm is not addressed, either.

2.3.3 Bridge between HOL and Maple

The design of this link by Harrison and Théry (1994) was triggered by the need to deal with real numbers in formal verification of hardware and software. It attempts to take advantage of some of Maple's algorithms within the HOL system. It is a typical example of proof reconstruction: the result returned by the computer algebra system is not trusted but used to construct a calculus-level proof in HOL with appropriate tactics.

Unfortunately, Maple does usually neither provide certificates nor traces for its results, and so the authors hope that the result of the computation does contain enough information that can serve as a certificate. Harrison (1996, Section 6.4.1) gives a number of examples where this is the case — for example, the extended Euclidean algorithm does not only return a greatest common divisor d of two elements a and b of a Euclidean ring, it also returns factors s and t such that

$$s \cdot a + t \cdot b = d \iff d \text{ is gcd of } a \text{ and } b.$$

Hence s and t make it possible to prove that d is a greatest common divisor of a and b without reiterating the computation.

Harrison's major example is the integration of expressions involving products of sines and cosines. As integration is the inverse to differentiation, and differentiation is straightforward to implement, it is hoped that this is another instance where "checking is easier than finding". But not so. The procedure that Harrison implements is as follows. For a given term, the integral is computed by Maple, and this integral is then differentiated within HOL. If the latter and the original term are equal, then surely the integral is correct, and using the (formally proved) Fundamental theorem of integral calculus a formal proof for the integral is obtained. Unfortunately there is a snag: Davenport *et al.* (1993, Section 5.1.1) point out that the real problem in differentiation is the simplification of the result, and indeed it turns out to be difficult to decide whether the derivative and the original term represent the same function. Because the computer algebra system is not trusted, this has to be decided in HOL and is rather inefficient. For the examples given in (Harrison, 1996, Section 6.4.3) deciding the equality takes 10 to 36 times longer than computing the derivative.

Harrison and Théry's interface treats the computer algebra system as a black box. In order to obtain soundness, Maple is not trusted. Utilising this link, some integrals could be proved formally in HOL that would have been harder to prove otherwise. As a general approach to interfacing computer algebra to a prover, it is limited, because many algorithms that are at the core of computer algebra need to be re-implemented in the prover, like computing normal forms in the case of integration, or tests of irreducibility.

2.3.4 Sapper

A different variant of the proof-reconstruction paradigm was suggested by Kerber *et al.* (1996). Here the context is the proof planning system Omega (Benzmüller *et al.*, 1997). This is based on a theorem prover, called the verifier, that is controlled by proof plans. These plans are generated by external, automated provers upon invocation by the user. The proofs found by these provers are translated into proof plans, which then can be expanded to calculus-level proofs by the verifier. These can then be checked by a proof-checker. A number of external provers are linked to Omega, and these provers, which all have different strengths, are based on different logics. Therefore the verifier's task to check all derived statements with respect to the target logic is important for the integrity of the whole system.

In order to obtain a system which is also computationally strong, Kerber *et al.* (1996) propose to add a computer algebra system as one of the external reasoners. In order to generate a proof plan, which is able to guide the verifier through a proof that justifies a computation, they propose to add a "verbose mode" to the computer algebra system, which emits a trace of the computation. They demonstrate this on a small, experimental computer algebra system and show how a plan for the verification of the addition of two polynomials can be generated.

The trace generated by the "verbose mode" replaces a certificate for the calculation. Good certificates for algorithms in computer algebra are often not known, and generating a trace of the computation is attractive because it seems to provide a uniform way of obtaining a certificate. In many instances these traces will not be efficient. Even worse, for a long computation it might not even be possible to store the trace. Gröbner bases are used, for example, to decide the membership-problem of polynomial ideals. Gröbner bases for ideals, which are given by a set of generating polynomials, are computed with Buchberger's algorithm. This can take tremendously long and may consume a huge amount of storage space.[3] A lot of effort was and is spent to make its implementations efficient and usable in practice. A trace is likely to be very long and repeating the proof in the calculus of the prover is not attractive.

Reconstructing a proof for the addition of polynomials is not an impressive achievement towards the sound integration of provers and computer algebra. Computer algebra systems perform this kind of operations efficiently, and there are certainly no doubts about the correctness of polynomial addition. Although computer algebra systems are not seen as black boxes by this approach, soundness problems are not seriously addressed. Emitting traces may ensure soundness, but verifying sound computations does not lead to an efficient system.

2.4 Outline of a New Approach

The aim of this work is to add computer algebra to a prover and so increase its strength in mathematical domains. This should not only result in more

[3] If d is a bound on the degree of the generating polynomials the computation of the Gröbner basis can involve polynomials of degrees proportional to 2^{2^d} (Cox *et al.*, 1992).

efficient computation inside a prover but also exploit algorithms whose development required research in the computer algebra community. What we have seen in this chapter is that this can only be reasonably achieved if existing implementations of computer algebra systems are used and, more importantly, trusted. Reconstructing proofs for computations in a prover's calculus is slow, and formally verifying algorithms of computer algebra is only gradually becoming feasible.

Because computer algebra systems are collections of algorithms that share data-structures but may use them with different meanings, translations into the calculus of a prover cannot be done uniformly, but different translations may be required for different algorithms. Therefore, in particular interfacing to the user interface of a computer algebra system is problematic. Most computer algebra systems have a built-in programming environment through which library and data-structures can be accessed directly.

Although computer algebra systems are not as sound as one might wish, many of them are trustworthy in parts. If it is possible to restrict the prover's access to only the sound parts it will not become unreliable. Unfortunately, it is hard to tell sound algorithms from *ad hoc* ones in large computer algebra systems. Code for simplification, which is often *ad hoc*, is usually spread all over the system. A perfectly sound algorithm may call library-routines that are *ad hoc* and therefore not be usable. Some more recent computer algebra systems use object-oriented type systems to structure their mathematical domains. Axiom (Jenks and Sutor, 1992) is the most prominent one of those. The advantage of these type systems is that all code for simplification in one domain is located in the module of that domain. Sound implementations are easier to identify, and, because the type system requires more discipline from the programmer, more likely to be found.

For the experimental purposes of this work, the library need not cover the latest developments in computer algebra, but the most important basic algorithms will be sufficient. These algorithms are implemented in any of the algebraic libraries. The library Sumit (Bronstein, 1996) is written in the same, statically-typed language Aldor (Watt *et al.*, 1994a,b) as Axiom and does not contain *ad hoc* code. All its modules are based on published algorithms and pointers to the literature are available.

Sumit is written in an object-oriented language, but this does not mean that only such libraries were trustworthy. Many specialised computer algebra libraries can be expected to be sound, because they deal with a smaller number of domains and they have been written by experts in the field. In fact, Sumit is such a specialised library — for the solution of differential equations. We will, however, only make use of the "algebraic infrastructure" of this library — up to solving linear equation systems and factorisation.

Side-conditions that apply to the correct use of algorithms need to be documented carefully in specifications. For sound algorithms they can be obtained from the literature. The book-keeping of those conditions is best done in the prover.

The conclusion of this chapter is that interfacing computer algebra to provers in a pragmatic way is possible. At the same time, the prover can be maintained trustworthy. The requirements to achieve this have been outlined in this section, and the design of an interface based on these requirements follows in the next chapter. It has to be kept in mind, though, that the interface that will be designed is only a framework and does not guarantee soundness itself. Choosing sound algorithms and providing appropriate translations and correct specifications for them is important to maintain soundness.

Chapter 3

Design of the Interface

The previous chapter presented a new, pragmatic approach to linking a computer algebra library to provers. In order to demonstrate that this approach is more feasible, and indeed more powerful, than not to trust the computer algebra component, an interface to the prover Isabelle is implemented. The design of this interface is described in the present chapter.

In the computer algebra community, there is interest to link computer algebra systems to each other and also to graphical and numerical systems, and scientific word processors. Experience from that research is valuable for interfacing to provers as well, but, because of limited rigour, only to a limited extent.

3.1 OpenMath

The main purpose of the interface is to translate mathematical objects between the representation used by the prover and the one of the computer algebra library. This is similar to the objective of the OpenMath project (Dalmas *et al.*, 1997), which since 1993 has worked towards interfacing different computer algebra systems with each other by providing a system independent representation for mathematical objects and appropriate translations to the systems' specific representations.

The OpenMath project has an interesting feature: it provides specifications for the mathematical objects, though only informal ones. Figure 3.1 shows that OpenMath distinguishes three layers of data-representation. The first is the private representation of a given program. The second is the representation as an *OpenMath-object*. The third layer provides encodings for the transport of data. On the second layer, expressions are represented as trees.

This layer and the third one are system-independent. The hierarchy of layers reflects how OpenMath is normally integrated into an application. *Content dictionaries* provide specifications for OpenMath-objects. Following these specifications, *phrase books* for particular computer algebra systems can be developed. Phrase books are essentially functions that translate between system-specific data and OpenMath-objects. They are implemented in the programming language of the corresponding computer algebra system and need to be coded manually (Huuskonen, 1997). For the third layer formats based on SGML and binaries are supported (Abbott, 1996). These formats are anticipated to support various sub-formats, for example encodings for floating-point numbers of different precision. OpenMath can negotiate suitable encodings between two applications and provides for encoding and decoding of OpenMath-objects.

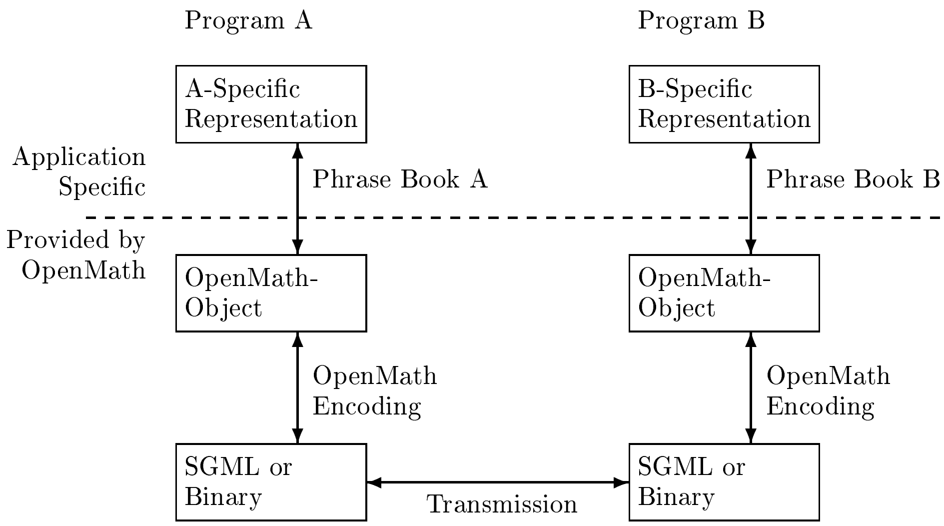

Figure 3.1: The OpenMath architecture

According to Dalmas *et al.* (1997, Section 2.4) OpenMath does not provide *exact* specifications. They write that "it would be beside the point to exactly specify the behaviour of any OpenMath [*sic*]." This was because "even when dealing with programs that compute, exact specifications could be impractical or too constraining for a given system to become OpenMath compliant." In fact, specifications of the semantics of objects only have the status of comments! For example, the contents dictionary Basic (Gonnet *et al.*, 1998), which defines objects that should be supported by all OpenMath compliant software, contains various references to mathematical literature, but no formal specifications other than type-profiles for operations. Also, the definitions of some transcendental functions are given in terms of limits, which is a notion that is generally understood, but it has not been introduced as an OpenMath-object at that stage.

This level of precision meets the standard of most computer algebra systems, seems appropriate for the purposes of OpenMath and is sufficient in order to move expressions from one system to another. This is a helpful service if the user is aware that systems may treat expressions differently. However, in order to reason mechanically, formal specifications are required.

It is not clear what Dalmas *et al.* (1997) mean by "exact specifications". As an example, take an algorithm that solves a linear equation system. Several strategies may be used in its implementation. They lead to differing sets of vectors that, of course, all span the same solution space. A specification for the algorithm should say that the returned vectors are solutions to the equation system, or it could be more specific and say that they indeed span the space of solutions. Both specifications are exact and can be formalised. The specification could be much stricter and attempt to fully determine the answer of the computation. For a linear equation system, a specification could rule out all but one of the possible bases that span the solution space. This amounts to require that the algorithm uses one of several possible strategies. In most cases, it would not be reasonable to be so specific. A user of OpenMath will probably want to use the equation solver of a different computer algebra system because it uses a different strategy, and hence the solution could be of a form more appropriate for the particular application. In this sense, formal specifications need not be exact.

However, because computer algebra systems have systematic soundness problems, one wonders whether OpenMath does not attempt to be exact in order not to get in conflict with the unsoundness of the computer algebra system. If that is the case, there is not much point in providing an OpenMath interface to a theorem prover. Various OpenMath-compliant computer algebra systems could be plugged into that interface, if they become available, but formal specifications, because they need to take their specific errors into account, would still differ.

OpenMath contains a number of interesting ideas, which we borrow. Because of the lack of rigour the interface to Isabelle is not based on OpenMath directly.

3.2 Isabelle's Theories

In the prover Isabelle, *theories* organise the hierarchical structure of a mathematical development. They contain declarations of constants and axioms of a mathematical development or *logic*. Typically, a user starts from one of the predefined theories that implement *object logics* — for example, ZF for set theory and HOL for higher-order logic — and extends them by new declarations and axioms. In a large development, the user will create a hierarchy of theories by extending and merging them. Theorems, which are derived from the axioms, are also associated to theories.

Signatures can be seen as substructures of theories. They contain all declarations that are associated to Isabelle's term-language, which is a typed version of the λ-calculus, namely constants, concrete syntax and also type information.

Theories may contain *oracles*. These allow Isabelle to take advantage of external reasoners. Invoked as an oracle, an external reasoner can create arbitrary Isabelle theorems. This provides a controlled way of bypassing the abstract data-type for theorems. Isabelle documents that the theorem has been generated by an external reasoner and records the oracle that was used. It is intended that these theorems do not change the logic of the theory, but are derivable from the axioms given in that theory or in its ancestors. Isabelle expects the external reasoner as an ML function taking two arguments: a signature and a description of the problem to be solved. The signature contains definitions and type information of λ-terms that may be passed to the oracle. The oracle may raise an exception to indicate that it cannot solve the specified problem.

3.3 Architecture

One of the design rationales was to have as much common design of the interface on the prover's and the computer algebra system's side as possible. Like in OpenMath this should at least be the case for encoding and transmission, and three layers are distinguished: from top to bottom a semantic layer, a syntactic layer and a transmission layer. Like in OpenMath, on the semantic layer objects are in the application's specific representation, on the syntactic layer a system-independent representation is used and on the transmission layer, a transport-encoding, which is also system-independent.

On the syntactic layer we use the simply typed λ-calculus. This is suitable because higher-order operations like integration can be represented in a natural way. The representation is also used for theorems, which are returned from the computer algebra component to the prover. On the transmission layer a straightforward translation of λ-terms into text-strings is used.

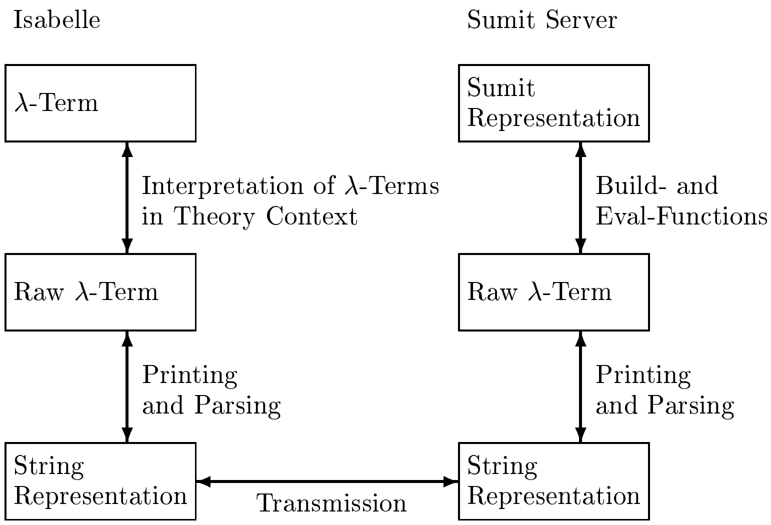

Figure 3.2: Layer structure of the interface

3.3.1 Servers and Services

The purpose of the interface is to provide algebraic algorithms as external reasoners to the prover, and to communicate the results in a form accessible to the prover. We call this presentation of an algorithm a *service*. Several services that share resources, usually the same library, are grouped to a *server*. The server also provides the necessary communication-infrastructure and translates to and from the library's representation of objects. Results of computations are packaged into theorems using *theorem templates*. Theorem templates provide specifications for the computations. They are explained in Section 3.3.2.

Every service is associated to a theory of the mathematical development in Isabelle. Specifications are interpreted with respect to the definitions and axioms of that theory. Isabelle's oracle mechanism is used in a different way than anticipated by its designer. One oracle-function is provided for the whole interface. The name of the theory associated to a service is stored in the corresponding server and can be queried. The signature of that theory, together with the specification, is supplied to the oracle-function. This function turns the specification into a theorem.

The implementation of the interface provides the necessary functionality to create child processes for server, to connect to and disconnect from them, and to make the necessary requests to services. The interface is provided as a set of ML-functions. Their signature is shown in Figure 3.3. It follows a description of these functions.

```
read_server_list       : string -> string list
list_available_servers : unit -> string list
list_connected_servers : unit -> string list
connect_server         : string -> string list
disconnect_server      : string -> unit
list_services          : string -> string list

which_theory           : string -> string -> string
term_service           : string -> string -> term list -> term
thm_service            : string -> string -> term list -> thm

exception SERVER  of {server: string, msg: string}
exception SERVICE of {server: string, service: string,
                      msg: string}
```

Figure 3.3: Signature of the interface

`read_server_list` *filename*. Names of available servers are kept in a file. This function retrieves them from the file *filename*, together with the commands to launch the servers. It returns the list of the server-names.

`list_available_servers ()` and `list_connected_servers ()` return the respective lists of servers.

`connect_server` *server* starts a process for the server with name *server* and obtains its list of available services, which it returns. An error occurs if the process could not be started for whatever reason and exception `SERVER` is raised.

`disconnect_server` *server* sends a terminate signal to the server and closes the connection. Again, exception `SERVER` may be raised.

`list_services` *server* returns a list of names of the services that are available with *server*, provided this server is connected. Otherwise, an empty list is returned.

`which_theory` *server service* returns the name of the theory associated to *service* of *server*. Results of *service* have to be interpreted in this theory. The theory name is obtained from the corresponding server.

`term_service` *server service* $[t_1, \ldots, t_n]$ invokes the given *service* on *server*. The terms $t_1, \ldots, t_n$ are passed to the service as arguments. The result is returned as a term $[\![P_1; \ldots; P_m]\!] \Longrightarrow C$ representing a meta-theorem. Its type is prop. The server may report an error, for example if the arguments do not type-check, are of the wrong type or are not understood by the service, or if the computation aborts with an error. In this case the exception `SERVICE` is raised passing on an error message from the service. Errors in the communication again raise the exception `SERVER`.

`thm_service` *server service* $[t_1, \ldots, t_n]$ calls `term_service` to carry out the requested computation. Its result must be a λ-term of type prop. The function also obtains the associated theory of the service with `which_theory`. It then uses Isabelle's oracle mechanism to generate a theorem. This is returned.

`SERVER` {`server:` *server*`,` `msg:` *message*} is generally raised if errors in the communication with the server occur.

`SERVICE` {`server:` *server*`,` `service:` *service*`,` `msg:` *message*} is raised if the server reports an error while executing a service.

3.3.2 Theorem Templates

What is an appropriate class of theorems to specify results of computations? Most of the computations by computer algebra systems have the form of simplifications. A term t is reduced to an in some sense equivalent term t'. Taking possible side-conditions into account, a suitable class of theorems could be conditional rewrites. But determining the side-conditions before the computation, as suggested by Ballarin *et al.* (1995), is not sufficient. They might arise in the process of the computation, for instance in the Gaussian algorithm, where the choice of pivots depends on all the previous computation steps. Therefore, the divisions, which are critical, can only be determined after the computation. Other results may not be naturally presentable as rewrites — for example the factorisation of a polynomial into its irreducible factors using a predicate like Factorisation $p[x_1, \ldots, x_n]$. This not because the system would become logically more expressive, but because structurally simpler theorems are usually easier to use in automated theorem proving. Therefore, it is advisable to allow the generation of arbitrary theorems.

Arguments and results of a computation, which have been translated to λ-terms, are put together, and a λ-term representing a theorem is obtained. Where appropriate, side-conditions arising from the computation are also regarded as results. This composition is done using a *theorem template*: at this experimental stage, simply a piece of code. The generated theorem is an instance of the algorithm's formal specification. If a computation fails, for instance because a division by zero occurs, or the computation runs out of resources, the generation of a theorem is refused and an exception is raised. A computation could run out of resources because of intermediate expression swell.

The theorem template for a service that computes the normal form of an expression is simple. If the service reduces a to b the theorem template creates the rewrite rule $(a = b) \equiv \text{True}$. Examples of non-trivial theorem templates are given in Sections 5.3 and 6.6.

3.3.3 Translation of Objects

The relation between types, or more generally speaking representations, in the two systems is involved. For example, 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.

This problem is resolved letting translations depend not only on the types, but also on the algorithm they interface to. An algorithm together with its translation functions and its theorem template is a service. Note that this also avoids the type reconstruction problem one would have when

[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.

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

Table 3.1: Specification of the evaluator for expressions in $\mathbb{F}_2[X]$.

interfacing to an untyped computer algebra system. Functions that convert from the server's representation to λ-terms are called *build*-functions, and functions that evaluate λ-terms into the server's representation, *eval*-functions.

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. The implemented interface passes all information 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 an evaluation function for every type needs to be provided. This is done by instantiating the evaluator, which is a polymorphic function. Table 3.1 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`. `Integer` is abbreviated to `Int` and `Boolean` to `Bool`. 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 appends a least significant bit to the list of bits. Bits are represented as Boolean values. The other constants are described in Chapter 4, where the formalisation of algebra in Isabelle is discussed. In the table λ-notation is used to abbreviate function definitions in Aldor.

In the current implementation of the evaluator, more needs to be specified. 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 in the evaluator for type τ 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 3.1. In future, these specifications should be generated 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 nonzero coefficients, which ensures that the term that is returned to Isabelle is also sparse. The zero-polynomial is, of course, mapped to 0.

Figure 3.4 shows the implementation of the function `BuildUpF2`. Code for `BuildNat`, the build-function for $\mathbb{N}$, has been omitted. The function `BuildMonom` builds a monomial aX^n. Over $\mathbb{F}_2$ X^n is the only nonzero monomial. It is translated to the term monom n. As the representation of the polynomial is sparse the case $a \neq 1$ cannot occur, provided the implementa-

```
macro F2 == SmallPrimeField 2;

BuildNat( n: Integer ): IsabelleTerm == { ... };

BuildMonom( a: F2, n: Integer ): IsabelleTerm == {
  if a = 1
  then Appl( Const( "monom", 'nat -> bool up', BuildNat( n ) ) )
  else error "BuildUpF2: Illegal coefficient."
};

BuildUpF2( p: SparseUnivariatePolynomial F2 )
  : IsabelleTerm == {

  Rec( a: F2, n: Integer,
       gen: Generator Cross( F2, Integer ) ): IsabelleTerm == {

    if empty? gen
    then BuildMonom( a, n )      -- create one (the last) monomial
    else {                       -- create sum of monomial and
      ( a2, n2 ) := value gen;   --   remaining polynomial
      step! gen;
      Appl(
        Appl( Const( "op +", 'bool up -> bool up -> bool up' ),
          BuildMonom( a, n ) ),
        Rec( a2, n2, gen ) )
    }
  };

  gen := generator p;
  step! gen;
  if empty? gen
  then Const( "<0>", 'bool up' )  -- create zero polynomial
  else {                          -- create monomials
    (a, n) := value gen;
    step! gen;
    Rec( a, n, gen )
  }
};
```

Figure 3.4: Implementation of the build-function for polynomials in $\mathbb{F}_2[X]$.

tion of the library is correct. The function `BuildUpF2` obtains the iterator `gen` for the polynomial `p`, with operations `step!` and `empty?`. The data-type of λ-terms is `IsabelleTerm`. The constructor `Const` creates a constant, and `Appl` is function application. Expressions that construct types for these λ-terms are abbreviated in single quotes, in order to improve the readability of the code.

3.4 Relation to Other Work

The main interest of this work is to demonstrate that algorithms from computer algebra can improve theorem provers. For this an experimental prototype is designed. Other issues, which are not being investigated, include the cooperation of concurrent processes (Dalmas and Gaëtano, 1996), abstracting from the transport medium (Gray *et al.*, 1996) and designing application programming interfaces (API) for services.

Only a stateless server is considered. A state could save the repeated translation and transmission of objects and could make the interface more efficient.

3.4.1 Open Mechanised Reasoning Systems

The analysis of the design of theorem provers has led to frameworks that identify components of provers and describe them formally. In their paper "Reasoning Theories" Giunchiglia *et al.* (1996) identify a *reasoning theory* component, a *control* component and an *interaction* component as the three key building blocks for a theorem prover. The reasoning theory component implements calculus and inference rules of the logic or formal system. The control part consists of a set of inference strategies. The interface implements the capability of interacting, usually with the human user. The authors call this framework *Open Mechanised Reasoning Systems* (OMRS).

In an LCF-style prover these components are clearly separated. The kernel implementing the abstract data-type for theorems is, in the view of OMRS, the reasoning theory, and tactics provide the control. The interface is provided by the programming language ML and its environment, or a graphical user interface on top of that. As far as the extensibility of the prover is concerned, this is a framework of enormous flexibility. The OMRS framework has been extended by Homann (1997) by computational services. It is not the purpose of the present dissertation to describe the presented interface formally. One observation is worth pointing out, though: a service corresponds to an elementary computational theory according to Homann

(1997). Heuristic control that might be present in a computer algebra system, and may make it unsound, is avoided by using a library, and control may be implemented in Isabelle's tactic language.

3.4.2 Prosper Plug-ins

The Prosper project, a collaboration of European universities and partners from industry, develops an extensible, open proof-tool architecture for "incorporating formal verification into industrial CAD/CASE tool flows and design methodologies". A plug-in interface (Norrish, 1999) allows to add plug-ins to the core proof engine, which is a variant of the prover HOL (Gordon and Melham, 1993). These plug-ins are anticipated to be external reasoners. Prosper's and our interface are similar. Servers correspond to plug-ins in Prosper and services to commands. Both interfaces only allow one-directional flow of control information from the prover to the external reasoner. The Prosper interface was developed after the design of our interface was complete and is more mature: plug-ins can be run on remote machines, a time-out feature exists and arbitrary data can be moved over the interface, while our interface only supports strings and λ-terms.

Chapter 4

Formalising Ring Theory

It is necessary to formalise algebra, so that results of computations can be specified. It is necessary to mechanise this knowledge, and, obviously, sufficiently much mathematics from the problem domain, to solve problems from this domain. On the other hand, it is not necessary to formalise theory, on which the informal correctness-proofs of algorithms that are used for computations depend. This is an advantage, because the mechanisation of mathematics is mainly manual and laborious.

Ring and field theory is the classical domain of computer algebra systems, although there are also a number of computer algebra systems that specialise in group theory. This chapter presents the relevant parts of classical algebra to the extent of detail that is required in the mechanisation of the proofs. Proofs are more detailed than in algebra textbooks. They are given to illustrate the difficulty of their mechanisation. The longer ones can be found in Appendix A. Then, the formalisation in Isabelle is discussed.

In this and the following chapter, following a convention in the HOL-community, juxtaposition is used for function application. Multiplication is denoted by "$\cdot$". Function application binds tighter then any other operation and is left-associative.

4.1 The Ring Hierarchy

Historically, classical algebra evolved over concrete domains like the integers, rational polynomials etc. At the beginning of this century, as the variety of domains mathematicians were interested in became more and more diverse, the investigated structures became *abstract*: they were defined by a set of axioms, not through a concrete construction. For an application in a theorem prover this abstraction is useful, because it allows one to derive results for abstract structures only once, and then instantiate them for particular domains.

It follows now the presentation the algebraic hierarchy of rings, which describes the classical domains of computation of computer algebra. It follows mainly Jacobson (1985, Chapter 2).

4.1.1 Rings

By *ring* we mean commutative ring with one. The definition is as follows.

Definition 1 (Ring) *A* ring *is a structure consisting of a nonempty set R, operations $+, \cdot : R \times R \to R$ and $- : R \to R$, and two distinguished elements $0, 1 \in R$ such that*

$$
\begin{array}{cccc}
(a+b)+c = a+(b+c) & \text{(A1)} & (a \cdot b) \cdot c = a \cdot (b \cdot c) & \text{(M1)} \\
0 + a = a & \text{(A2)} & 1 \cdot a = a & \text{(M2)} \\
(-a) + a = 0 & \text{(A3)} & (a+b) \cdot c = a \cdot c + b \cdot c & \text{(D)} \\
a + b = b + a & \text{(A4)} & a \cdot b = b \cdot a & \text{(C)}
\end{array}
$$

hold for all $a, b, c \in R$.

A number of elementary properties of rings are consequences of the fact that a ring is an abelian group relative to addition (axioms A1 to A4) and a monoid relative to multiplication (axioms M1 and M2).

For all a holds $0 \cdot a = (0+0) \cdot a = 0 \cdot a + 0 \cdot a$. By adding $-0 \cdot a$ on both sides, one obtains $0 \cdot a = 0$. Similarly $0 = 0 \cdot b = (a+(-a)) \cdot b = a \cdot b + (-a) \cdot b$, which shows that $(-a) \cdot b = -a \cdot b$.

4.1.2 Integral Domains

Various types of rings are obtained by imposing special conditions on the multiplicative monoid.

Definition 2 (Integral domain) *A ring is an* integral domain*, or* domain *for short, if $1 \neq 0$ and it does not contain any zero divisors other than zero: formally, $a \cdot b$ implies $a = 0$ or $b = 0$.*

4.1.3 Factorial Domains

Many rings have an interesting structure of divisibility. The following definitions apply for any ring.

An element a is said to *divide* b, if there is an element d such that $a \cdot d = b$. One writes $a \mid b$. 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:

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

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. This is, formally,

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

Let $a \neq 0$ be an element of a domain with factorisation $a = p_1 \cdot p_2 \cdots p_s$, where the p_i are irreducible. This is an *essentially unique* factorisation if for any other factorisation $a = p'_1 \cdot p'_2 \cdots p'_t$, p'_i irreducible, one has $t = s$ and $p'_{i'} \sim p_i$ for a suitable permutation $i \mapsto i'$ of the set $\{1, 2, \ldots, s\}$.

Definition 3 (Factorial domain) *An integral domain D is called* facto- rial *if every non-unit element of its monoid D^* of nonzero elements has an essentially unique factorisation into irreducible factors.*

This is equivalent to D^* satisfying both the following conditions (see Jacobson, 1985, Theorem 2.21):

Divisor chain condition. If $a_1, a_2, \ldots$ is a sequence of elements of D^* such that $a_{i+1} \mid a_i$ then there exists a natural number n such that $a_n \sim a_m$ for all $m \geq n$.

Primeness condition. Every irreducible element of D^* is prime.

For the purpose of the case study in Chapter 5, only the primeness condition is required, and therefore the divisor chain condition is omitted from the formalisation of factorial domains.

For computations in factorial domains it is of advantage to work with *canonical* elements. An adequate choice for a canonical representative of a class of associated elements needs to be made. A factorisation of x into irreducible canonical factors is unique up to the order of factors, but their product may differ from x by a unit factor. The definition of the predicate

$$\operatorname{Factorisation} x\; F\; u \;\equiv\; (x = \operatorname{foldr}\, \cdot\, F\; u) \wedge (\forall a \in F.\, \operatorname{irred} a) \wedge u \mid 1 \qquad (4.1)$$

accommodates this. 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. A consequence of unique factorisation is that for irreducible a, b and c

$$a \mid b \cdot c \implies a \sim b \lor a \sim c.$$

This is immediate from the primeness condition. Lifting this result to lists of factors, by induction, yields

$$\text{irred}\, c \land \text{Factorisation}\, x\ F\ u \land c \mid x \implies \exists d.\, c \sim d \land d \in F. \tag{4.2}$$

4.1.4 Fields

Definition 4 (Field) *A ring is called a* field *if* $1 \neq 0$ *and all nonzero elements have an inverse.*

Because fields do not contain irreducible elements, the factorisation of field elements is trivial, and therefore unique. Hence fields are factorial domains.

4.2 Polynomials

For a given ring R, polynomials 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 addition and multiplication make this structure a ring and an R-algebra. The necessary definitions follow:

Definition 5 (R-module) *Let R be a ring with operations $+_R, -_R, \cdot_R, 0_R$ and 1_R and M an abelian group with operations $+_M, -_M$ and 0_M. M is an* R-module *if there is* scalar multiplication $\cdot_s : R \times M \to M$ *and the following axioms*

$$\begin{aligned}(a +_R b) \cdot_s p &= a \cdot_s p +_M b \cdot_s p & \qquad (a \cdot_R b) \cdot_s p &= a \cdot_s (b \cdot_s p)\\ a \cdot_s (p +_M q) &= a \cdot_s p +_M a \cdot_s q & 1 \cdot_s p &= p\end{aligned}$$

hold for $a, b \in R$ and $p, q \in M$.

Definition 6 (R-algebra) *If the module M is itself a ring and*

$$(a \cdot_s p) \cdot_M q = a \cdot_s (p \cdot_M q)$$

holds for $a \in R$ and $p, q \in M$ then M is called an R-algebra.

Definition 7 (Univariate polynomials) *Let R be a ring. The set*

$$R[X] \equiv \{f : \mathbb{N} \to R \mid \exists n. \forall i > n. f\ i = 0\}$$

is the set of univariate polynomials *of R. For $f \in \mathbb{N} \to R$ an $n \in \mathbb{N}$ is called a* bound *if $\forall i > n. f\ i = 0$. For $p \in R[X]$, $p\ i$ is the i-th* coefficient *of p.*

One writes p_i for the ith coefficient, or coefficient of *degree* i, of the polynomial p. The set of polynomials, together with appropriate definitions of operations, forms a ring.

Theorem 8 *The univariate polynomials $R[X]$ over a ring R, together with the operations*

$$\begin{aligned}
p + q &\equiv (n \mapsto p_n + q_n) \\
p \cdot q &\equiv (n \mapsto \sum_{k=0}^{n} p_k \cdot q_{n-k}) \\
-p &\equiv (n \mapsto -p_n) \\
0 &\equiv (n \mapsto 0) \\
1 &\equiv (n \mapsto \mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ 0)
\end{aligned}$$

form a ring.

It needs to be shown that the operations are closed over $R[X]$ and that the ring axioms hold. For the proof, see Appendix A.1.

Theorem 9 *The ring of polynomials $R[X]$ with scalar multiplication*

$$a \,{}_{\S}\, p \equiv (n \mapsto a \cdot p_n)$$

is an R-algebra.

Proof. Again, closedness of the operations needs to be shown: $a \,{}_{\S}\, p$ is a polynomial if p is one. Indeed, any bound of p is also a bound of $a \,{}_{\S}\, p$. Also, the module-axioms follow directly from facts about the coefficient domain. The algebra-axiom is slightly more difficult. Comparing coefficients reduces the problem to

$$\sum_{i=0}^{n} (a \cdot p_i) \cdot q_{n-i} = a \cdot \left(\sum_{i=0}^{n} p_i \cdot q_{n-i} \right),$$

which holds because of associativity and distributivity in R. $\square$

With the definition of $X \equiv (n \mapsto \text{if } n = 1 \text{ then } 1 \text{ else } 0)$ the usual representation of polynomials is obtained. In the sequel, $a_0 \mathbin{\dot{\S}} 1 + a_1 \mathbin{\dot{\S}} X + a_2 \mathbin{\dot{\S}} X^2 + \ldots + a_n \mathbin{\dot{\S}} X^n$ is usually abbreviated to $a_0 + a_1 X + a_2 X^2 + \ldots + a_n X^n$. The constant X is called the *indeterminate* of the polynomial ring. Note that X is not an argument to the construction of $R[X]$. It is merely a convenient notation.

4.2.1 Degree

The degree of a polynomial p, denoted by $\deg p$, is the highest exponent with nonzero coefficient. The degree of the zero polynomial is often defined to be $-\infty$, appealing to rules in $\mathbb{N} \cup \{-\infty\}$ like $-\infty + n = -\infty$ or $-\infty < n$. The conventions $\deg 0 = -1$ and $\deg 0 = 0$ can also be found. For the purpose of the formalisation that is discussed here $\deg 0 = 0$ is adopted, because in a formalisation it is easier to handle a few special cases directly, rather than hiding them in a new domain: the cases will have to be considered when reasoning about the degree anyway.

Definition 10 (Degree) *The* degree *of a polynomial $p \in R[X]$,* $\deg p$*, is the least exponent such that all coefficients of higher degree are zero.*

The usual properties of the degree hold, hence it is equivalent to the common definition. It is clear that

$$\deg(p + q) \leq \max(\deg p)(\deg q).$$

For the product $p \cdot q$, $\deg p + \deg q$ is a bound, as we have already seen in the proof that multiplication is closed for polynomials. Thus

$$\deg(p \cdot q) \leq \deg p + \deg q. \tag{4.3}$$

Equality holds in (4.3) if R is an integral domain, and $p, q \neq 0$, consequently

$$p \neq 0 \wedge q \neq 0 \implies \deg(p \cdot q) = \deg p + \deg q. \tag{4.4}$$

Then the coefficient of degree $\deg p + \deg q =: k$ of $p \cdot q$ is

$$\sum_{i=0}^{k} p_i \cdot q_{k-i} \underset{(a)}{=} \sum_{i=0}^{\deg p} p_i \cdot q_{k-i} \underset{(b)}{=} p_{\deg p} \cdot q_{\deg q}$$

and nonzero because both $p_{\deg p}$ and $q_{\deg q}$ are nonzero. Equality (a) holds, because $p_i = 0$ for $i > \deg p$, and (b) holds, because for $i < \deg p$ the inequality $k - i > \deg q$ holds, and so $q_{k-i} = 0$.

For scalar multiplication, $\deg(a \,⨟\, p) \leq \deg p$, because all coefficients that are zero in p are also zero in $a \,⨟\, p$. Over an integral domain, for $a \neq 0$, the equality $\deg(a \,⨟\, p) = \deg p$ holds. Degrees for the constants 0, 1 and X are $\deg 0 = \deg 1 = 0$ and $\deg X = 1$.

4.2.2 Evaluation Homomorphism and Universal Property

Definition 11 (Ring homomorphism) *Let R and S be rings. A map $f : R \to S$ that preserves addition and multiplication, $f(a +_R b) = fa +_S fb$, $f(a \cdot_R b) = fa \cdot_S fb$, and maps one to one, $f 1_R = 1_S$, is a* ring homomorphism.

It follows that a ring homomorphism also preserves the other ring operations and, in particular, $f 0_R = 0_S$.

The embedding $\mathrm{const} : \left\{ \begin{smallmatrix} R & \to & R[X] \\ a & \mapsto & aX^0 \end{smallmatrix} \right\}$ is a ring homomorphism. The equality $(\mathrm{const}\, a) \cdot p = a \,⨟\, p$ holds. Polynomials have the following fundamental property:

Theorem 12 (Universal property) *Let R and S be rings. Then for every pair (ϕ, a), where $\phi : R \to S$ is a ring homomorphism and $a \in S$, there exists a unique homomorphism Φ such that the following diagram commutes:*

$$\begin{array}{ccc} R & \xrightarrow{\mathrm{const}} & R[X] \\ {\scriptstyle\phi}\big\downarrow & & \big\downarrow{\scriptstyle\Phi} \\ S & \xrightarrow[\mathrm{id}]{} & S \end{array}$$

In order to prove the existence one defines

$$\Phi : p \mapsto \sum_{i=0}^{\deg p} \phi p_i \cdot a^i \tag{4.5}$$

for given a and ϕ. For the proof that this is a ring homomorphism and that it is unique, see Appendix A.2.

The mapping Φ is called *evaluation homomorphism*. It describes the evaluation of a polynomial when substituting a for its indeterminate.

4.2.3 Polynomials and the Ring Hierarchy

Not only does the polynomial-construction carry the ring structure of the coefficient domain over to the domain of polynomials, other properties are preserved as well. In terms of the ring hierarchy the following holds.

Theorem 13 (Polynomials and ring hierarchy) *The polynomials over a ring form a ring, and the polynomials over an integral domain form an integral domain.*

Proof. The first part of the statement is Theorem 8. To show the second property, let p and q be polynomials with $p \cdot q = 0$. Assume both $p \neq 0$ and $q \neq 0$. By (4.4) $\deg p + \deg q = \deg(p \cdot q) = \deg 0 = 0$ and therefore $\deg p = 0$ and $\deg q = 0$. As $p \cdot q = 0$, also $p_0 \cdot q_0 = 0$. The coefficient ring is an integral domain. So either p_0 or q_0 must be zero and, because both p and q are of degree zero, also either p or q. This is a contradiction and hence the polynomials form a domain.

4.3 Notes on the Mechanisation in Isabelle/HOL

Isabelle (Paulson, 1994) is a logical framework. It provides a *meta-logic* in which application specific *object-logics* can be mechanised. The meta-logic is based on the intutionistic fragment of simply typed λ-calculus. Like in Gordon's system HOL (Gordon and Melham, 1993) the type system is extended to a first-order language, providing type variables. This type system does not have subtyping. Isabelle also provides *type classes* that allow *order-sorted* polymorphism. Type-checking for this type system is decidable and therefore done automatically in Isabelle.

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. Types must be nonempty. The logic has the usual connectives ($\wedge, \vee, \longrightarrow, \dots$) and quantifiers ($\forall, \exists$). Currying is used for function application. Equality $=$ on the type bool is used to express if-and-only-if and equivalence $\equiv$ for definitions. The symbol $\Longrightarrow$ expresses entailment in a deduction rule. Some definitions require Hilbert's ϵ-operator: $\epsilon x.P\, x$ denotes a value for which the predicate P holds, presupposing its existence. The notation for formulae in the entire document is close to their representation in Isabelle. The judgement $t :: \tau$ asserts that term t is of type τ; $\tau \Rightarrow \sigma$ denotes a function type. Usually type information from formulae has been omitted to improve their legibility.

Type classes control the overloading of constants. A type class is essentially a set of types. The fact that a type τ is in some class C is expressed by the judgement $\tau :: C$. *Axiomatic type classes* link classes to specifications. These are given as axioms. They provide a very simple module system that supports abstract reasoning. It can only deal with theories that have a single parameter, so for example with rings but not with vector spaces. Type classes and their implementation and use in Isabelle are described by Nipkow (1993), Wenzel (1997) and Wenzel (1998). Type classes without axioms are also called *syntactic type classes.*

Proof development is interactive in Isabelle. The first step in a proof is entering a statement that is to be proved — the *goal.* The goal is stored in the *proof state.* Then, proof commands are applied to the goal, creating new, but hopefully simpler, *subgoals.* This is done until all subgoals have been resolved and the statement is proved. Proof commands are called *tactics.* They range from the implementation of an inference rule to proof procedures. Tactics are programs and thus arbitrary proof procedures can be encoded easily. Among the tactics provided by Isabelle are a rewrite engine (the *simplifier*) and a generic tableau prover.

4.3.1 The Ring Hierarchy

Abstract rings are theories that only have one carrier. Therefore axiomatic type classes are well-suited to formalise the hierarchy of rings. All ring operations are overloaded, by means of the syntactic type class ringS. Rings, integral domains, factorial domains and fields correspond to the axiomatic type classes ring, domain, factorial and field. These classes are given by inheriting from the appropriate more general structure and specifying additional axioms.

Figures 4.1 and 4.2 show the theory file for the theory `Ring`, which contains the declarations for the various abstract rings in Isabelle. The theory is derived from the theories `Arith` and `FoldR`. It inherits all their definitions and theorems, including the object-logic HOL.

After the header line of the theory file, sections with declarations follow, each of them starting with a keyword. After the keyword `consts`, declarations of constants follow. Each starts with the name of the constant, followed by its type and an optional annotation for syntax. The type may contain type variables — for example `'a` in ASCII-notation. The `defs`-section contains constant definitions. These are special axioms, and they are preceded by names, which are used to refer to definitions in proof scripts. Each section with keyword `axclass` contains the declaration of a type class. A class can be the subclass of other classes. If axioms follow, the class is an axiomatic type class. The asserted class-hierarchy can be changed with `instance`-sections. If the classes are axiomatic, suitable witnesses — that is, theorems — have to supplied, so that the class-inclusion can be verified by the prover. The following ASCII-notation is used in formulae. The characters `&` and `|` stand for $\wedge$ and $\vee$, `ALL` and `EX` are quantifiers, `~` denotes negation and `~=`

inequality, and `@` is Hilbert's ϵ-operator.

```
Ring = Arith + FoldR +

axclass
  ringS < plus, times, power

consts
  "<0>"          :: 'a::ringS                  ("<0>")
  "<1>"          :: 'a::ringS                  ("<1>")
  uminus         :: 'a::ringS => 'a::ringS     ("- _" [66] 65)
  dv             :: ['a::ringS, 'a] => bool    (infixl 60)
  assoc          :: ['a::ringS, 'a] => bool    (infixl 60)
  irred          :: 'a::ringS => bool
  prime          :: 'a::ringS => bool
  inverse        :: 'a::ringS => 'a

defs
  dvd_def        "a dv b    == EX d. a * d = b"
  assoc_def      "a assoc b == a dv b & b dv a"
  irred_def      "irred a   == a ~= <0> & ~ a dv <1>
                           & (ALL d. d dv a --> d dv <1> | a dv d)"
  prime_def      "prime p   == p ~= <0> & ~ p dv <1>
                           & (ALL a b. p dv a*b --> p dv a | p dv b)"
  inverse_def    "inverse a == if a dv <1>
                           then @x. a*x = <1> else <0>"
```

Figure 4.1: Declaration of the theory Ring in Isabelle (Part 1)

The theory `Ring` declares the syntactic type class ringS. This class inherits the overloading of $+$, $\cdot$ and the exponentiation-operator from the respective classes. Then new, overloaded constants are declared. Definitions for some of the constants follow. The other constants are axiomatised in the axiomatic classes that follow. Figure 4.4(a) shows the asserted hierarchy of classes. The non-emptiness condition of rings need not be specified explicitly, because all types in HOL are nonempty. For factorial domains, only the primeness condition is asserted. After proving that fields are factorial domains and instantiation the hierarchy is as shown in Figure 4.4(b). The scripts for these proofs are in the file of proof scripts of theory `Ring`. Instantiation takes place in the theory `Field`, see Figure 4.3. The supplied theorems differ from the axioms of the parent classes only in the sort constraint of the type variable, which is α :: domain in the class domain and

```
axclass
  ring < ringS
  a_assoc       "(a + b) + c = a + (b + c)"
  l_zero        "<0> + a = a"
  l_neg         "(-a) + a = <0>"
  a_comm        "a + b = b + a"
  m_assoc       "(a * b) * c = a * (b * c)"
  l_one         "<1> * a = a"
  l_distr       "(a + b) * c = a * c + b * c"
  m_comm        "a * b = b * a"
  one_not_zero  "<1> ~= <0>"

axclass
  domain < ring
  integral      "a * b = <0> ==> a = <0> | b = <0>"

axclass
  factorial < domain
  factorial_divisor     "True"
  factorial_prime       "irred a ==> prime a"

axclass
  field < ring
  field_ax      "a ~= <0> ==> a dv <1>"

end
```

Figure 4.2: Declaration of the theory Ring in Isabelle (Part 2)

$\alpha :: \mathsf{factorial}$ in the class $\mathsf{factorial}$. The theorems are

`field_integral`	$(a :: \alpha :: \mathsf{field}) \cdot b = 0 \implies a = 0 \lor b = 0$
`field_fact_prime`	$\mathrm{irred}(a :: \alpha :: \mathsf{field}) \implies \mathrm{prime}\, a$

and `TrueI` is the theorem "True".

The axiom $1 \neq 0$ has been moved to the class ring; this is different from the presentation of the ring hierarchy in Section 4.1. The exponentiation operator is defined in the usual manner for exponents over $\mathbb{N}$. In particular $a^0 = 1$ for any a. The inverse is defined not only for fields, but arbitrary rings. The inverse exists for the unit elements of a ring. If an inverse does not exist the inverse-function returns zero: all functions in HOL are total.

The setup of Isabelle's simplifier for rings needed some experimentation. It turned out to be more practical not to compute normal forms by default.

```
Field = Ring +

instance
  field < domain    (field_integral)

instance
  field < factorial (TrueI, field_fact_prime)

end
```

Figure 4.3: Declaration of the theory Field in Isabelle

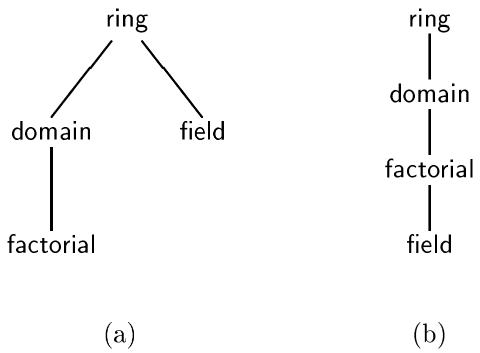

Figure 4.4: Ring hierarchy before (a) and after instantiation (b)

Ground complete ordered rewrite systems are guaranteed to terminate with Isabelle's permutative rewriter. Such systems exist for a number of algebraic theories (see Martin and Nipkow, 1990). The mechanisation provides such systems for the associative-commutative theories of the operations $+$ and $\cdot$, see Figure 4.5. They can be added if required. It is not advisable to apply these simplifications by default, firstly because this is not flexible enough: there are many situations in proofs where such normal forms are not desirable. Secondly, permutative rewriting with respect to an associative-commutative theory essentially performs bubble-sort on the term.

$$a + b = b + a \qquad (a + b) + c = a + (b + c) \qquad a + (b + c) = b + (a + c)$$

Figure 4.5: Ground complete rewrite system for the ac-theory of addition

A substantial part of the development of polynomials deals with summations. A summation operator $\mathrm{SUM}\, n\, f \equiv \sum_{i=0}^{n} f i$ is provided, which is overloaded for rings, like other ring operations. The congruence rule

$$m = n \wedge (\forall i.\, i \leq n \implies f i = g i) \implies \mathrm{SUM}\, m\, f = \mathrm{SUM}\, n\, g$$

permits Isabelle's simplifier to use the assumption $i \leq n$ when simplifying the expression x in $\mathrm{SUM}\, n\, (\lambda i.\, x)$. This is required in many of the proofs presented in Section 4.2 and Appendix A. The summation operator is used for all summations throughout the formal development. Because the standard notation is more readable, this is used in this presentation.

4.3.2 Polynomials

Polynomials are an abstraction of functions from $\mathbb{N} \to R$. We encapsulate this in a new type: the type-constructor up, for "univariate polynomial". Polynomials make use of the overloaded ring-operations: up has arity (ringS)ringS and is instantiated to (ring)ring and (domain)domain after these facts have been established formally.

In the formalisation no constant for the indeterminate X is provided but the function monom : $\mathbb{N} \to R[X]$. This function creates monomials — monom n is X^n, for example. A polynomial with given coefficients can be represented in terms of the operations monom, $\cdot_s$ and $+$. This representation is *sparse*: $X^{1000} - 1$ can be handled without iterating over 999 zero coefficients, as would be the case with the standard representation as lists. Also, this representation is natural and does not need technical overhead as other formalisations would — for example, using association lists. The function coeff returns the coefficient of a polynomial. The coefficient p_i of p is coeff $i\, p$.

Two representation theorems are provided. Two polynomials are equal if all the coefficients are equal. This is expressed by the theorem

$$(\forall n.\ \mathrm{coeff}\, n\, p = \mathrm{coeff}\, n\, q) \implies p = q.$$

It is not necessary to compare the coefficients for all $n \in \mathbb{N}$, but only the coefficients up to the degree, and therefore

$$\deg p \leq n \implies \sum_{i=0}^{n} \mathrm{coeff}\, i\, p \cdot_s X^i = p \tag{4.6}$$

holds.

4.3.3 The Degree Function

Finally, reasoning about the degree of a polynomial can be tricky, because of the definition of the degree function in terms of the ϵ-operator, which is implicit in the definition as the least exponent, such that all the coefficients of higher degree are zero. Suitable facts that simplify the reasoning are derived from the definition. Often, it is necessary to prove $\deg p = n$. This can be split into

$$\deg p \leq n \qquad \text{and} \qquad \deg p \geq n.$$

Then $\deg p \leq n$ holds if all coefficients above n are zero

$$(\forall m.\, n < m \longrightarrow \mathrm{coeff}\, m\, p = 0) \implies \deg p \leq n$$

and $\deg p \geq n$ holds if the n-th coefficient of p is distinct from zero

$$\mathrm{coeff}\, n\, p \neq 0 \implies n \leq \deg p.$$

Note that both theorems also hold for the zero-polynomial, for which also $\deg 0 = 0$ by definition.

The following facts about the coefficients of a polynomial with given degree hold. Coefficients of degree higher than the degree of the polynomial are zero

$$\deg p < m \implies \mathrm{coeff}\, m\, p = 0$$

and the coefficient corresponding to the degree of the polynomials, the *leading coefficient*, is nonzero

$$p \neq 0 \implies \mathrm{coeff}(\deg p)\, p \neq 0.$$

Again, the zero polynomial is a special case: $\mathrm{coeff}\, m\, 0 = 0$. Obviously, it is not possible to derive anything for coefficients below the degree of the polynomial from the degree alone.

4.3.4 Evaluation Homomorphism

Ring homomorphisms are formalised with the predicate homo. A function ϕ of type $\alpha :: \mathsf{ringS} \Rightarrow \beta :: \mathsf{ringS}$ is a ring homomorphism if and only if homo ϕ. Given a homomorphism $\phi : R \to S$, a polynomial $p \in R[X]$ and $a \in S$, the function EVAL is defined by

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

This function maps ϕ and a to the corresponding evaluation homomorphism. The following facts express the universal property.

$$\text{homo}\,\phi \implies \text{homo}(\text{EVAL}\,\phi\,a) \tag{4.7}$$

$$\text{homo}\,\phi \implies \text{EVAL}\,\phi\,a\,(X^n) = a^n \tag{4.8}$$

$$\text{EVAL}\,\phi\,a\,(\text{const}\,b) = \phi b \tag{4.9}$$

4.4 Related Work

The algebraic hierarchy plays a role in the implementation of domains in computer algebra systems. Also, in this section the present mechanisation is compared to work by others.

4.4.1 The Algebraic Hierarchy in Axiom

The computer algebra system Axiom (Jenks and Sutor, 1992) was the first to exhibit abstract domains. Before that, computer algebra systems implemented a fixed number of domains. A user could implement new domains, but he or she would also need to provide all functionality for them. Despite a factorisation algorithm, say, had been available in the system already, and this algorithm might have been applicable to the new domain, the user would not have been able to use it, because of the limitations of the system. In Axiom this problem was overcome by using an object-oriented type system. Abstract classes, which are called *categories*, model abstract domains. Concrete classes provide implementations of domains. They can take other domains as arguments, and they correspond to type constructors. In Axiom, they are called *functors*. A functor may provide a generic implementation of a factorisation algorithm, and a user-declared domain can take advantage of the implementation, if it is a member of the right category.

The algebraic hierarchy implemented in Axiom is vast. The system supports various kinds of factorial domains, provides several implementations of polynomials and many other constructions. It is based on work by (Davenport and Trager, 1990) and (Davenport *et al.*, 1991).

Axiom is a computational system and does not prove theorems. Categories are no axiomatisations of abstract domains, but merely provide the appropriate interfaces. However, specifying axioms in categories is encouraged by a special syntax for comments.

4.4.2 Other Mechanisations of Polynomial Algebra

Bailey (1993) has mechanised univariate polynomials in LEGO (Luo and Pollack, 1992). The LEGO system implements a constructive type theory, and Bailey's mechanisation of polynomials is constructive in most parts as well. One consequence is that a degree function for arbitrary polynomials cannot be defined: it might not be decidable if elements of the coefficient domain are zero.

Bailey's representation of polynomials is dense: they are simply lists of coefficients. Many proofs about elementary properties, in particular of polynomial multiplication, are much simpler than for a sparse representation. Factorial domains are not covered, but *Euclidean* domains are. This is the class of domains over which long division is possible.

Jackson (1995) verifies an implementation of multivariate polynomials, which is based on association lists, in the prover NuPRL (Constable, 1986), by proving that the universal property holds. This development is also constructive and a characterisation of the universal property is used that does not need a degree function. Much of the work is about properties of association lists. Factorial domains are also formalised.

Both these works are not only concerned with the mechanisation of polynomials, but also with general frameworks for representing algebra in type theory.

Chapter 5

Proofs in Coding Theory

This chapter presents a case study that demonstrates the feasibility of our approach and shows that the use of computer algebra can simplify the mechanisation of proofs. At the same time, the integrity of the formal system is maintained.

Proofs for the existence of algebraic codes have substantial computational content. For instance, the existence of Hamming codes is closely related to the existence of certain irreducible polynomials. Such polynomials can be determined efficiently by factorisation algorithms, which are implemented in computer algebra systems.

After a brief introduction to the relevant part of coding theory, we present two existence proofs of certain algebraic codes. These proofs are mechanised in Isabelle and take advantage of the link with the computer algebra library Sumit. Core parts of these proofs depend on theorems that are generated by the computer algebra library.

5.1 Coding Theory

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*. The *n-error-detecting* codes can 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.

The field $\mathbb{F}_2 = \{0, 1\}$ is fundamental in an algebraic treatment of binary codes, which we consider here. Codewords are represented as polynomials in $\mathbb{F}_2[X]$.

Definition 14 *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 in $\mathbb{F}_2[X]$:

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

The cyclic shift of a codeword a is then $(X \cdot a)\,\mathrm{rem}(X^n - 1)$. Note also that associated elements are equal in $\mathbb{F}_2$ and $\mathbb{F}_2[X]$ — that is, $a \sim b \implies a = b$.

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 (Hoffman *et al.*, 1991, Theorem 4.2.17):

Theorem 15 (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 $X^n - 1$.*

5.1.1 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*.[1] This allows us to state the following structural theorem on cyclic Hamming codes (Hoffman *et al.*, 1991, Theorem 5.3.2):[2]

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

[2]There, only the direction from right to left is stated. The other direction is not difficult. We give a brief sketch of its proof, referring to (Hoffman *et al.*, 1991). Hamming

Theorem 16 (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* $\deg g = r$.

5.1.2 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.

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. Let F denote this field. The coefficient domain $\mathbb{F}_2$ can be embedded into F by mapping elements from $\mathbb{F}_2$ to the corresponding constant polynomials "modulo h". Hence this quotient construction is called *field extension*. Let $a \in F$. The nonzero polynomial $m_a \in \mathbb{F}_2[X]$ of smallest degree, such that m_a evaluated at a is zero, is unique and called the *minimal polynomial* of a. Our definition of the minimal polynomial uses two steps:

$$\text{minimal}\, g\, S \;\equiv g \in S \wedge g \neq 0 \wedge (\forall v \in S.\, v \neq 0 \longrightarrow \deg g \leq \deg v) \quad (5.1)$$

$$\text{min\_poly}\, h\, a \equiv \epsilon g.\, \text{minimal}\, g\, \{p \mid (\text{EVAL const }\; a\, p)\, \text{rem}\, h = 0\} \quad (5.2)$$

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 in general. The function min_poly returns the minimal polynomial over $\mathbb{F}_2$ for an element a of the extension field F constructed with $h \in \mathbb{F}_2[X]$. Note that the quotient construction of F is not carried out explicitly: elements of F are represented by polynomials over $\mathbb{F}_2$. Hence $a \in \mathbb{F}_2[X]$. 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 $(\text{EVAL const }\; a\, p)\, \text{rem}\, h = 0$. The embedding const is needed to lift the coefficients of p to $\mathbb{F}_2[X]$. The equality is evaluated modulo h by means of the remainder function rem associated with polynomial division.

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

---

codes of length $2^r - 1$ have *parity-check* matrices whose $2^r - 1$ rows consist of all nonzero vectors of length r (Section 3.3). The rows of the parity-check matrix of a cyclic code of length n with generator polynomial g of degree r correspond to the polynomials $X^i \,\text{rem}\, g$ $(i = 1, \ldots, 2^r - 1)$ (remark on page 110). A cyclic Hamming code of length $2^r - 1$ has dimension $2^r - r - 1$, hence the degree of its generator polynomial g is r. As the $X^i \,\text{rem}\, g$ are all distinct, g must be a primitive polynomial (remark on page 124).

primitive element. BCH codes are defined by choosing suitable generator polynomials. The generator polynomial of the BCH code of length $2^r - 1$ is $m_b \cdot m_{b^3}$.

If we describe the field extension in terms of a primitive polynomial h, then X corresponds to a primitive element. The minimal polynomial of X is h, because h is irreducible. Therefore we can define BCH codes as follows:

Definition 17 *Let $h \in \mathbb{F}_2[X]$ be a primitive polynomial of degree r. The code of length $2^r - 1$ generated by $h \cdot \mathrm{min\_poly}\, h\, X^3$ is called a* BCH *code.*

5.2 Formalising Coding Theory

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

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

5.2.1 The Hamming Code Proofs

We now describe our first application of the interface between Isabelle and Sumit. It is used 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 15 and 16 are required and formalised as follows:

$$0 < n \longrightarrow (\exists C. \mathrm{generator}\, n\, g\, C) = g \mid X^n - 1 \tag{5.3}$$

$$(\exists C. \mathrm{generator}(2^r - 1)\, g\, C \wedge \mathrm{Hamming}\, r\, C) = (\deg g = r \wedge \mathrm{primitive}\, g) \tag{5.4}$$

These equations are asserted as axioms and are the starting point of the proof that follows. Note that (5.4) 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

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

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 (5.4) and a divisor of $X^{15} - 1$ by (5.3). The factorisation of $X^{15} - 1$ is computed using Berlekamp's algorithm:

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

All the irreducible divisors of $X^{15} - 1$ are in this list. This follows from (4.2). 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 (5.4) 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 (4.1).[3]

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

5.2.2 The BCH Code Proofs

The predicate BCH is, in line with Definition 17, defined as follows:

$$\begin{aligned}\text{BCH}\, r\ C \ \equiv\ (\exists h.\, &\text{primitive}\, h \wedge \deg h = r \wedge \\ &\text{generator}(2^r - 1)\ (h \cdot \text{min\_poly}\, h\ X^3)\ C)\end{aligned} \tag{5.5}$$

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

We prove that $X^8+X^7+X^6+X^4+1$ is generator of a BCH code of length 15. This polynomial divides $X^{15}-1$ and by Theorem 15 is generates a cyclic linear code of length 15. In order to show that this code is a BCH code it is sufficient to show:

$$\text{generator}\, 15\, (X^8+X^7+X^6+X^4+1)\, C \Longrightarrow \text{BCH}\, 4\, C \tag{5.6}$$

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 (5.5) 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

$$(\text{EVAL const}\;\, X^3\, p)\,\text{rem}\,(X^4+X+1) = 0 \tag{5.7}$$

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 (5.7). 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 (4.7) to (4.9), 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 $\deg p \le 3$, we get by the representation theorem for polynomials (4.6) 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 (5.7) 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. Hence $\deg p \ge 4$ and $X^4+X^3+X^2+X+1$ is a solution of minimal degree.

- Uniqueness: We need to show that $X^4+X^3+X^2+X+1$ is the only polynomial of smallest degree satisfying (5.5). We study the solutions of (5.7) of degree of ≤ 4 by setting $p = p_0 + \ldots + p_4X^4$ and obtain another equation system

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

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

  $$\begin{aligned}(p_0+p_1X^3+p_2(X^3+X^2)+p_3(X^3+X)+p_4(X^3+X^2+X+1) = 0)\\ = (\exists t.\, p = t \,{}^{\circ}_{\circ}(X^4+X^3+X^2+X+1)).\end{aligned}$$

  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. □

5.3 Review of the Development

In the development of algebra that was presented in Chapter 4, all facts have been derived from the definitions. In the development of coding theory, general facts about Hamming and BCH codes, in particular Theorems 15 and 16, have been asserted as axioms: our interest lies in the proofs that are discussed in Section 5.2. These proofs have been mechanised. Some facts that are not proved in the formalisation of algebra have also been asserted as axioms. These are the properties

$$\begin{aligned} c \neq 0 &\implies (a+b)\,\mathrm{rem}\,c = a\,\mathrm{rem}\,c + b\,\mathrm{rem}\,c\\ b \neq 0 &\implies (r \,{}^{\circ}_{\circ}\, a)\,\mathrm{rem}\,b = r \,{}^{\circ}_{\circ}(a\,\mathrm{rem}\,b)\end{aligned}$$

of the remainder function rem, and the fact that $\mathbb{F}_2[X]$ is a factorial domain.

Besides the mechanisation of the proofs, a server was implemented on top of Sumit that connects to the interface to Isabelle. It provides translation functions and theorem templates. The correspondence between types in Isabelle and Sumit is shown in Table 5.1.

Build- and eval-functions are provided to translate between the types, as described in Section 3.3.3. Note that the type bool is used in the binary representation of natural numbers. The eval-function that maps objects from bool to `Boolean` is only used by the eval-function for natural numbers, no attempt is made to evaluate arbitrary logical formulae.

Isabelle	Sumit
bool	`Boolean`
nat	`Integer`
bool	`SmallPrimeField 2`
bool up	`SparseUnivariatePolynomial SmallPrimeField 2`

Table 5.1: Correspondence of types between Isabelle and Sumit

The server provides services to compute normal forms for expressions that do not contain variables in the domains $\mathbb{N}, \mathbb{F}_2$ and $\mathbb{F}_2[X]$. These services take a term t of the appropriate type as argument and return a theorem of the form $t = t'$, where t' is the normal form of t.

Other services decide equality, inequality and divisibility of these domains. They accept terms of the form $a \odot b$, where $\odot$ is one of the connectives $=, <, \leq$ and $|$. Either $a \odot b =$ True or $a \odot b =$ False is returned.

The main contributions of computer algebra to the proofs are factorisation and Gaussian elimination. Services and theorem templates for these are presented in the following sections.

5.3.1 Factorisation

The service `F2PolyFact` computes factorisations of polynomials in $\mathbb{F}_2[X]$. Sumit's factorisation algorithms for fields of prime cardinality reside in the module

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

where the parameter `F` is an implementation of a field of prime cardinality,[4] and `P` is an implementation of univariate polynomials over that field. The module provides, amongst other functions,

```
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.

[4]For reasons of efficiency, Sumit distinguishes implementations where the prime p can be stored in a machine word, is an arbitrary precision integer and $p = 2^{31} - 1$.

Monic means that its leading coefficient is one. The decomposition 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 nonzero 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, factors it using `factor` with `berlekamp` as first argument and obtains a coefficient u and a product of irreducible factors P. The coefficient is lifted to the corresponding constant polynomial, and this and the factors in P are converted back to λ-terms. These are then assembled to the theorem

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

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, nonzero constant polynomials are the units of the corresponding polynomial domain.

5.3.2 Gaussian Elimination

The service `F2Gauss` provides an interface to Gaussian elimination. It solves homogeneous linear equation systems over $\mathbb{F}_2$. The service accepts a list of polynomials $[a_0, \cdots, a_n]$ over $\mathbb{F}_2$. These polynomials correspond to the column vectors of the matrix $(a_0|\cdots|a_n)$. The service determines the size of the matrix, which is $(\max_{i=0}^{n} \deg a_i + 1) \times (n+1)$, and translates the list of polynomials into a matrix. Sumit's module

```
LinearAlgebra(
    R: IntegralDomain,
    M: MatrixCategory0 R,
    E: EliminationCategory( R, M ) )
```

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

provides operations for matrices. The parameter `R` is an integral domain, `M` an implementation of matrices instantiated by `R`, and `E` provides an implementation of linear elimination. Different variants of Gaussian elimination can be used depending on whether `R` is an integral domain, a Euclidean domain or a field. We use ordinary Gaussian elimination. Sumit provides its implementation in the module `OrdinaryGaussElimination`. In order to solve the equation system $A \cdot x = 0$, the function

```
nullspace: M -> List Vector R
```

from the module `LinearAlgebra` is used. It returns a base $[v_1, \ldots, v_k]$ of the solution space of $A \cdot x = 0$. The vectors v_i are converted back to polynomials and the theorem

$$\left(\sum_{i=0}^{n} (\mathrm{coeff}\ i\ x) \mathbin{\dot{s}} a_i = 0\right) = (\exists t_1 \cdots t_k .\, x = t_1 \mathbin{\dot{s}} v_1 + \ldots + t_k \mathbin{\dot{s}} v_k) \tag{5.8}$$

$$\text{or} \quad \left(\sum_{i=0}^{n} (\mathrm{coeff}\ i\ x) \mathbin{\dot{s}} a_i = 0\right) = (x = 0), \quad \text{if } k = 0.$$

is generated from the a_i and the v_i by a theorem template. The t_i are variables in $\mathbb{F}_2$ and the coeff $i x$ are fields of the vector x, which is represented by a polynomial. This example shows also that great flexibility is needed for theorem templates.

5.3.3 Feasibility of Proof Reconstruction

Mechanising the proofs in a system that integrates the computer algebra component without trusting it would require the user to prove the theorems generated by the theorem 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 that resembles a trace for the computation.

In the case of Gaussian elimination, checking that a vector is a solution to the equation system is simpler than computing the solution space. This is the direction from right to left in (5.8). However, in the proof of the theorem on BCH codes (5.6), uniqueness and minimality of the polynomial $X^4 + X^3 + X^2 + X + 1$ depend on knowing the complete solution space. This is the direction from left to right. The completeness of the solution space is guaranteed because all transformations of the matrix performed by Gaussian elimination are equivalence transformations. Gaussian elimination reduces the matrix to triangular form in a number of elimination steps. Reconstructing a proof for the computation essentially means to re-execute it in the prover's calculus. Only code for the selection of the pivots could be omitted if they were recorded in a suitable trace. Note that the template (5.8) is not the full specification of Gaussian elimination: the information that the v_i are linearly independent has been omitted.

The main property of Berlekamp's algorithm is not that it returns factors, but that these factors are irreducible. We have exploited this in the proof about Hamming codes. Berlekamp's algorithm is based on a proof that these factors are indeed irreducible (see Geddes *et al.*, 1992, Section 8.4). Let $a \in \mathbb{F}_q[X]$ be a square-free polynomial, where $q = p^n$, p prime. The algorithm first computes the number of factors and then determines them. The algorithm constructs a subspace W of the residue ring $\mathbb{F}_q[X]/(a)$, which is a vector space. The dimension of W is the number of irreducible factors of a. It is obtained by computing a basis for W. Again, the proof, which is constructed by Berlekamp's algorithm, cannot be reconstructed in a simpler manner easily.

These arguments support the claim that proof reconstruction would essentially require recalculating the results in the prover. This would also have required the formalisation of a substantial part of linear algebra and ideal theory. The arguments are, of course, intuitive and not rigorous in a proof-theoretic sense. The sequence of elimination steps computed by Gaussian elimination is not necessarily *optimal*. It is likely that shorter sequences of elimination steps or ones with smaller intermediate terms exist that lead to the reconstruction of shorter proofs. But Gaussian elimination is *practical*. An algorithm that computes the optimal sequence of elimination steps, and that could return an optimal certificate for the reconstruction of a proof, is likely to be much less efficient than Gaussian elimination. Also we do not claim that efficient certificates for irreducibility do not exist — so far, work in computer algebra was mostly concerned with finding all the factors of a given polynomial, and to the knowledge of the author it is not known whether certificates for irreducibility do exist or not and whether they can be computed efficiently.[6] But this does not make any difference to our approach to interfacing computer algebra to provers. It is pragmatic and aims at using algorithms from computer algebra, not designing new ones. Still, this work might stimulate such design in the computer algebra community.

[6]Davenport appears to have written a paper on this topic recently, but despite some effort the author was neither able to obtain a copy nor a draft of it.

5.3.4 Automating the Use of Computer Algebra in Proofs

The function `thm_service`, which provides the link to the computer algebra library (see Section 3.3.1), can be called directly in proof scripts. This is only done on rare occasions in proof scripts, once in the Hamming code proof to obtain the factorisation of $X^{15} - 1$, and twice in the proofs about BCH codes to solve linear equation systems.

Often it is more convenient to use tactics that extract the argument for the computation directly from the proof state. Two tactics that prove irreducibility and primitiveness of polynomials are provided. These tactics can solve subgoals of the form $\mathrm{irred}\, p$ and $\mathrm{primitive}\, p$, respectively. The proof of irreducibility using the factorisation algorithm has been outlined in Section 5.2.1. The term p is extracted from the proof state. If it is of the correct type and does not contain variables, it is passed to Sumit and reduced to normal form p'. This yields the theorem $p = p'$. Then the factorisation of p' is computed. If this results in the theorem $\mathrm{Factorisation}\, p'\ [p']\ 1$, then irreducibility follows from the definition of Factorisation (4.1) and is proved. Otherwise the tactic fails.[7] Similarly, primitiveness is proved by first showing that the polynomial is irreducible and then establishing that it does not divide $X^m - 1$ for any $m \in \{n+1, \ldots, 2^n - 2\}$. This is done by calling the appropriate service of Sumit for $m = n+1, \ldots, 2^n - 2$. Again, the tactic fails if primitiveness cannot be proved.

Isabelle's simplifier can be extended by *simplification procedures*. These are functions that map terms that match a specified pattern to rewrite rules. During a rewrite, when the pattern matches the current redex, the procedure is invoked. If the simplification procedure is successful, it returns a rewrite rule that is then applied by the simplifier to rewrite the redex. Simplification procedures have been introduced as *conversions* by Paulson (1983). They are useful to automate the invocation of the computer algebra library to compute normal forms of expressions, although one has to be careful not to make this inefficient. The simplifier uses a bottom-up strategy to analyse expressions. When simplifying the expression $(X^4+X^2)\cdot(X^3+X)$ $(\in \mathbb{Z}[X])$ it is not desirable to first reduce X^4+X^2 to X^2+X^4 and X^3+X to $X+X^3$ and then $(X^2 + X^4) \cdot (X + X^3)$ to $X^3 + 2X^5 + X^7$, because the cost of communicating with the computer algebra system is significant. Instead, one would want to pass the whole term to the computer algebra system only once. It turns out that choosing only functions whose result types are different from their argument types is a good heuristic. Only expressions

[7] In $\mathbb{F}_2[X]$ the element 1 is the only unit. Over other domains a slightly more sophisticated argument is required to prove the irreducibility of $2X - 2$, say.

that match the patterns $\deg p$, $m = n$, $m < n$, $m \leq n$, $\operatorname{coeff} n\, p$, $a = b$, $\operatorname{const} a$, $p = q$ and $p \mid q$ are simplified by default in our setup of Isabelle's simplifier. Here $m, n \in \mathbb{N}$, $a \in \mathbb{F}_2$ and $p, q \in \mathbb{F}_2[X]$. Other expressions can be simplified in particular invocations of the simplifier, if required.

The tactics for irreducibility and primitiveness can also be used as simplification procedures. Because Isabelle's rewriter is integrated with its classical reasoner, these procedures can prove the last part of Section 5.2.1, 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. Unfortunately, this is not very efficient, because Sumit is invoked with too many computations that are not used in the final proof. The computation time is 92 seconds, where 40 seconds are consumed by Isabelle and 52 seconds by the server Sumit. Therefore, we use a longer proof script, using 5 steps to prove this particular subgoal. To run the proof script for the entire Hamming code proof takes only 40 (21+19) seconds.

5.3.5 Implementation Effort

Table 5.2 gives an overview on the implementation effort for the whole project: implementation of the interface, mechanisation of algebraic background and the proofs in coding theory. The figures are, however, misleading in so far that developing proof scripts is much harder than ordinary programming.

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

Table 5.2: Size of the development (code sizes in 1000 bytes)

The interface of Sumit is considerably larger, because the data-type `IsabelleTerm` for λ-terms and the server functionality are provided as well. The entry "Coding theory proofs" includes the implementation of the proof procedures for irreducibility and primitiveness of polynomials. In contrast, for example, the size of Sumit's factoriser of polynomials over finite fields, which was used in these proofs, is 38 200 bytes of source code, not including code for the underlying data-structures.

	Proof Scripts	Steps	Lines of Code
Hamming proofs	1	35	46
BCH proofs	2	59	90

Table 5.3: Size of proof scripts

The sizes of the proof scripts for the proofs described in detail in Sections 5.2.1 and 5.2.2 are given in Table 5.3. The BCH proof is split in two parts. "Steps" is the number of manual invocations of tactics.

	Isabelle	Sumit	Total
Hamming proofs	21	19	40
BCH proofs	31	21	52

Table 5.4: Execution time of proof scripts (in seconds)

Execution times for the coding theory proofs are shown in Table 5.4. Times were taken on a SPARCstation-10 with 224 MBytes of main memory. Communication between Isabelle and Sumit is slow, because λ-terms are converted to strings for data-transport and need to be parsed by the receiver. No effort has been made to optimise the communication.

Chapter 6

Semantics of Symbols

In Section 2.1 we have presented a classification of soundness-problems in computer algebra systems. While the coupling of incompatible algorithms can be seen as a problem of software-engineering, and mathematical research is necessary to replace *ad hoc* algorithms by sound ones, the specialisation problem reveals limitations in the design of computer algebra systems themselves: their treatment of symbols is not flexible enough. Not only may this lead to the misinterpretation of results by a user who is not an expert, it also imposes restrictions on the integration of algorithms in the system. Another view of *ad hoc* algorithms is that they have been designed so that they could easily be integrated into the restricted framework of a computer algebra system, rather than solving the mathematical problem in a sound way. This has certainly been the case for simplification algorithms that were erroneous because they did not distinguish whether the domain of computation were real or complex numbers (Aslaksen, 1996), and this is also the source of erroneous algorithms for definite integration (Adams *et al.*, 1999).

6.1 Symbolic Gaussian Elimination

An example of a simple algorithm where the specialisation problem occurs is Gaussian elimination over equation systems with symbolic coefficients. The use of the Gaussian algorithm over rational functions, which are fractions of polynomials, for linear equation systems with symbolic parameters was proposed by Lipson (1969) and has become standard in computer algebra. To illustrate the effect of the specialisation problem, consider the following parametric equation system, which is taken from the textbook on linear algebra by Noble and Daniel (1988, p. 136). It is given as *augmented matrix*

$$\left(\begin{array}{ccc|c} 1 & -2 & 3 & 1 \\ 2 & t & 6 & 6 \\ -1 & 3 & t-3 & 0 \end{array}\right)$$

where a row represents one equation — for example, $x - 2 \cdot y + 3 \cdot z = 1$. The parameter is t, but this could be a different symbolic expression, like $\sin u$. The Gaussian algorithm computes essentially a *reduced row echelon form*. Most computer algebra systems return

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & \frac{t+9}{t+4} \\ 0 & 1 & 0 & \frac{4}{t+4} \\ 0 & 0 & 1 & \frac{1}{t+4} \end{array}\right)$$

It is immediate that $t = -4$ is a special case. There is another special case $t = 0$, which is not obvious from the result, but can be found when inspecting the internal workings of the algorithm. For $t = 0$ and $t = -4$ the reduced row echelon forms are

$$\left(\begin{array}{ccc|c} 1 & 0 & 3 & 3 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right) \quad \text{and} \quad \left(\begin{array}{ccc|c} 1 & 0 & -5 & 0 \\ 0 & 1 & -4 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right)$$

respectively, and the sets of solutions differ from the general case. This problem is well-known: the exercise in the textbook was designed to point it out to students. Of course, the Gaussian algorithm is correct. The difficulty arises because most computer algebra systems do not report for which special cases the solution is not valid.

6.2 A Formal View of the Specialisation Problem

The specialisation problem is known in the computer algebra community, but rarely receives attention in publications, possibly because there is no *mathematical* difficulty to it. Nevertheless, the problem needs to be addressed. Parameters in general are variables and as such *logical* entities. An appropriate view of symbolic Gaussian elimination, which follows the suggestion of Lipson (1969), is to regard the parameters as indeterminates. Thus the algorithm operates over a polynomial extension $R[X]$ of the problem domain R. The Gaussian algorithm is known to be correct over this extension.

6.2.1 Fraction Fields

The evaluation homomorphism describes the relation between polynomials and variables formally. Divisions occur in the Gaussian algorithm, and therefore, evaluation needs to be lifted over fractions of polynomials. These are often 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 $\frac{b}{a}$. The domain of rational functions $\mathrm{FF}(R[X])$ is usually abbreviated to $R(X)$.

6.2.2 Evaluation of Rational Functions

How are $g : x \mapsto \frac{(x-1)^2}{x^2-1}$ and $\frac{(X-1)^2}{X^2-1} \in R(X)$ 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, $a \in R$ and EVAL as defined in Section 4.3.4. The function Φ_a 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. The following definition avoids this problem.

Definition 18 *For $a \in R$ the* evaluation homomorphism *for rational functions is*

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

where $f \in R(X)$. If polynomials p and q do not exist then $\Phi_a(f)$ is undefined.

This definition is independent of the chosen polynomials p and q. The usual notation for $\Phi_a(f)$ is $f(a)$ for both polynomials and rational functions.

Let $f = \frac{(X-1)^2}{X^2-1}$. The objects $f(a) = \Phi_a(f)$ and $g(a)$ are different because for $a = 1$ the evaluation of the rational function $f(a)$ is 0, while $g(a)$ is not defined.

6.3 The Gaussian Algorithm

Solutions obtained by the Gaussian algorithm are not necessarily valid for arbitrary substitutions of parameters. As the exceptional cases are not always visible from the solution of the system, the internal workings of the algorithm need to be examined. The following is an exposition of the algorithm and its correctness proof. We will focus on a variant of the Gaussian algorithm, but first introduce the general scheme.

6.3.1 Gaussian Elimination in General

All variants of the algorithm solve a linear equation system $A \cdot x = b$, where A is an $m \times n$-matrix over R and b a vector. R must be an integral domain. The solution of the *homogeneous* system $A \cdot x = 0$ is a d-dimensional vector space $S_{\text{hom}} = \langle v_1, \dots, v_d \rangle$ over $\text{FF}(R)$. For $d = 0$ the space of solutions is $\{0\}$. If the *inhomogeneous* system $A \cdot x = b$ is solvable, only one *particular* solution v_{p} needs to be obtained in addition to the homogeneous solution. The solution space is $S_{\text{part}} = v_{\text{p}} + \langle v_1, \dots, v_d \rangle$.

In the first step, the Gaussian algorithm transforms the augmented matrix $(A|b)$ by applying a sequence of *row transformations*. Each row transformation is one of the following:

- Exchanging two equations,
- Multiplying an equation with $c \neq 0$ and
- Adding a multiple of an equation to another equation.

Row operations do not change the set of solutions of the equation system and are thus *equivalence transformations*. Note that this requires that the computation be carried out in an integral domain. Row transformations are applied until a matrix in *row echelon form* is obtained. The row echelon form of the augmented matrix is $(A'|b')$ and has the following shape:

$$\left(\begin{array}{cccccccc|c}
0 & \dots & 0 & a'_{1j_1} & \dots & \dots & \dots & a'_{1n} & b'_1 \\
0 & \dots & \dots & 0 & a'_{2j_2} & \dots & \dots & a'_{2n} & b'_2 \\
\vdots & & & & \ddots & & & \vdots & \vdots \\
0 & \dots & \dots & \dots & 0 & a'_{rj_r} & \dots & a'_{rn} & b'_r \\
0 & \dots & \dots & \dots & \dots & \dots & \dots & 0 & b'_{r+1} \\
\vdots & & & & & & & \vdots & \vdots \\
0 & \dots & \dots & \dots & \dots & \dots & \dots & 0 & b'_m
\end{array}\right)$$

The a'_{ij_i} are nonzero and are called the *stairs*. If all the entries in the columns above the stairs are zeros and the stairs themselves are ones, the matrix is in *reduced row echelon form*.

In order to arrive at a row echelon form, row transformations have to be applied in a suitable manner. There are several possibilities to guarantee termination of this process. They are applied in different variants of the Gaussian algorithm.

In the homogeneous case, the equation system represented by the row echelon form can be solved sequentially for decreasing row index, starting with row r.

- Variables x_l that do not correspond to stairs (that is, $l \neq j_i$ for $i = 1, \dots, r$) can be chosen arbitrarily. Let

$$x_l = \tau_l \tag{6.1}$$

  for parameters $\tau_l \in \mathrm{FF}(R)$.

- For $l = r, \dots, 1$ the linear equation

$$a'_{lj_l} \cdot x_{j_l} + a'_{l,j_l+1} \cdot x_{j_l+1} + \dots + a'_{ln} \cdot x_n = 0 \tag{6.2}$$

  is equivalent to

$$x_{j_l} = -\frac{1}{a'_{lj_l}} \cdot (a'_{l,j_l+1} \cdot x_{j_l+1} + \dots + a'_{ln} \cdot x_n). \tag{6.3}$$

  The $x_{j_l+1}, \dots, x_n$ are already known at this stage. Hence equation (6.3) gives the solution for x_{j_l}.

By rearranging equations (6.1) and (6.3) one obtains the solution as a linear combination of vectors, which are denoted by v_i, with factors τ_i. Renumbering the indices from 1 to $d = n - r$, one obtains $A' \cdot x = 0$, if and only if

$x \in \langle v_1, \ldots, v_d \rangle$. The process of sequentially solving the equation system is known as *back-substitution.*

Solutions to the inhomogeneous system exist if and only if $b'_{r+1} = \ldots = b'_m = 0$. A particular solution x_p is obtained by a similar back-substitution process:

- The x_l that do not correspond to stairs can be chosen arbitrarily. As only one solution is required, let $x_l = 0$.
- For $l = r, \ldots, 1$ set

$$x_{j_l} = \frac{1}{a'_{lj_l}} \cdot (b'_l - a'_{l,j_l+1} \cdot x_{j_l+1} - \ldots - a'_{ln} \cdot x_n).$$

Then $v_\mathrm{p} = (x_1, \ldots, x_n)$ is a particular solution.

6.3.2 Fraction Free Gaussian Elimination

This variant of Gaussian elimination has been described by Bareiss (1968), see also Geddes *et al.* (1992). Its advantage is that no fractions are introduced during row transformations and back-substitution (with exception of the particular solution), while, at the same time, the entries of the matrix are kept small and so the problem of intermediate expression swell is reduced. This algorithm is important in computer algebra, because for equation systems over polynomial domains, gcd-computations for the cancellation of fractions can be avoided. The entries of the matrix must be from an integral domain R. While the computed vectors that span the solution space of the homogeneous system are over R, the particular solution and the solution space are, of course, still over $\mathrm{FF}(R)$.

The row transformations of fraction-free Gaussian elimination are usually described by the following update-formula:

$$a'_{ij} := (a'_{kj_k} \cdot a'_{ij} - a'_{kj} \cdot a'_{ij_k}) / a'_{k-1,j_{k-1}} \tag{6.4}$$

For increasing k the update is performed for $i = k+1, \ldots, m$ and introduces zeros in column j_k below the pivot a'_{kj_k}. At the beginning, one sets $a'_{ij} = a_{ij}$. Note that entries of the row echelon form that are computed by (6.4) are not changed in the sequel. Row exchanges may be necessary to select a nonzero pivot. The algorithm terminates when no more nonzero pivots are available: the remaining rows contain only zeros. One sets $a'_{0j_0} = 1$, no division is performed in the first iteration. Bareiss (1968) shows that the division never produces fractions. Therefore, row transformations can be carried out over R and the row echelon form is a matrix over R.

Vectors spanning the solution space of the homogeneous system, and that are over R, are obtained by modifying back-substitution. Equation (6.1) is replaced by

$$x_l = \prod_{i=1}^{l} a'_{ij_i} \cdot \tau_l. \tag{6.5}$$

Back-substitution for the particular solution remains unchanged. This solution is a vector over $\mathrm{FF}(R)$ in general.

6.4 Lifting the Algorithm to the Parametric Case

In the following, let $M_R(m,n)$ denote the set of $m \times n$-matrices with entries from R and R^n the set of row and column vectors of length n with entries from R. For matrices and vectors over polynomials and polynomial functions we write $A(t), b(t)$ *etc.* for the element-wise application of the evaluation homomorphism Φ_t.

We now analyse for which substitutions of the indeterminate the solution of an equation system solved over $R(X)$ and obtained by fraction-free Gaussian elimination is also valid over $\mathrm{FF}(R)$. More precisely, let $A \cdot x = b$ be an equation system where $A \in M_{R[X]}(m,n)$ and $b \in R[X]^m$. The algorithm returns $v_1, \dots, v_d \in R[X]^n$ and $v_\mathrm{p} \in R(X)^n$ such that

$$\forall x \in R(X)^n.\, A \cdot x = b \iff x \in v_\mathrm{p} + \langle v_1, \dots, v_d \rangle.$$

For which $t \in \mathrm{FF}(R)$ do these vectors span the solution space of the corresponding system over $\mathrm{FF}(R)$? Formally

$$\forall x \in \mathrm{FF}(R)^n.\, A(t) \cdot x = b(t) \iff x \in v_\mathrm{p}(t) + \langle v_1(t), \dots, v_d(t) \rangle$$

In the first phase, the algorithm computes a row echelon form that represents an equivalent equation system. Row transformations correspond to left-multiplying the augmented matrix with certain transformation matrices. Let F be the product of all these transformation matrices. The equation system corresponding to the row echelon form is $F \cdot A \cdot x = F \cdot b$ and is equivalent to the original system if and only if $\det F \neq 0$. The restrictions on row transformations given in Section 6.3.1 ensure this. Similarly, $A(t) \cdot x = b(t)$ is equivalent to $F(t) \cdot A(t) \cdot x = F(t) \cdot b(t)$ if $\det F(t) \neq 0$.

For fraction-free Gaussian elimination, the update-formula (6.4) corresponds to the transformation matrix

$$F_k = \frac{1}{a'_{k-1,j_{k-1}}} \cdot \begin{pmatrix} a'_{k-1,j_{k-1}} & & & & & \\ & \ddots & & & & \\ & & a'_{k-1,j_{k-1}} & & & \\ & & -a'_{k+1,j_k} & a'_{kj_k} & & \\ & & \vdots & & \ddots & \\ & & -a'_{mj_k} & & & a'_{kj_k} \end{pmatrix},$$

where $a'_{0j_0} = 1$. Row exchanges correspond to *permutation matrices*. These matrices commute with other matrices, and their determinants are one. Therefore, $F = F_r \cdot \ldots \cdot F_1 \cdot P$, where P is the product of the permutation matrices. The F_k are lower-triangular matrices and

$$\det F_k = \left(\frac{a'_{kj_k}}{a'_{k-1,j_{k-1}}} \right)^{m-k}.$$

It follows that

$$\det F = a'_{1j_1} \cdot \ldots \cdot a'_{r-1,j_{r-1}} \cdot {a'_{rj_r}}^{m-r}.$$

The equation system represented by the row echelon form is equivalent to the original system for substitutions t of the indeterminate with $\det F(t) \neq 0$. Note that $a'_{kj_k} \in R[X]$, because they are entries in the row echelon form.

For the analysis of back-substitution, consider first a single equation $a \cdot x = b$ for $a, b \in R(X)$. If $a \neq 0$ then $x = \frac{b}{a}$ is the only solution. The corresponding system over $\mathrm{FF}(R)$ is $a(t) \cdot x = b(t)$.

- If $a(t) \neq 0$ then $x = \frac{b(t)}{a(t)}$ is equivalent and $\frac{b(t)}{a(t)}$ is the only solution.
- Otherwise the solution space depends on $b(t)$. If $b(t) = 0$ the solution space is $\mathrm{FF}(R)$, otherwise it is empty.

If $a'_{kj_k}(t) \neq 0$ for all k the matrix $A'(t)$ is in row echelon form. Then back-substitution can be lifted to the parametric case. The variables x_l that do not correspond to stairs have the same indices as in the polynomial case. They can be chosen arbitrarily. By choosing $x_l = \prod_{i=1}^{l} a'_{ij_i}(t) \cdot \tau_l$ for parameters $\tau_l \in \mathrm{FF}(R)$ the whole solution space is covered because $\prod_{i=1}^{l} a'_{ij_i}(t) \neq 0$.

For $a'_{lj_l}(t) \neq 0$ $(l = 1, \ldots, r)$ the equation

$$a'_{lj_l}(t) \cdot x_{j_l} + a'_{l,j_l+1}(t) \cdot x_{j_l+1} + \ldots + a'_{ln}(t) \cdot x_n = 0$$

is equivalent to

$$x_{j_l} = -\frac{1}{a'_{lj_l}(t)} \cdot (a'_{l,j_l+1}(t) \cdot x_{j_l+1} + \ldots + a'_{ln}(t) \cdot x_n)$$

and so

$$A'(t) \cdot x = 0 \iff x \in \langle v_1(t), \ldots, v_d(t) \rangle.$$

Assume now that $A'(t)$ is in row echelon form. A particular solution exists if $b'_{r+1}(t) = \ldots = b'_m(t) = 0$. Analogous to the homogeneous case, lifting back-substitution poses no further problems. Variables not corresponding to stairs can be chosen arbitrarily, and the solution of the polynomial case can be lifted to the parametric case. That is, $v_\mathrm{p}(t)$ is a solution of $A'(t) \cdot x = b'(t)$.

We have proved

Theorem 19 *Let $A \in M_{R[X]}(m,n)$ and $b \in R[X]^m$. Let $(A'|b')$ be the row echelon form obtained by fraction free Gaussian elimination. Let a'_{ij_i} denote the stairs of A' ($i = 1, \ldots r$) and b'_i denote the entries of b. Let $v_1, \ldots, v_d$ be vectors obtained by this algorithm for the homogeneous solution and v_p the particular solution.*

Let $p = \prod_{i=1}^{r} a'_{ij_i}$ and $q = \prod_{i=r+1}^{m} b'_i$. If $p(t) \neq 0$ then $\langle v_1(t), \ldots v_d(t) \rangle$ is the solution space of the homogeneous equation system $A(t) \cdot x = 0$.

In this case, the solution space of the inhomogeneous system $A(t) \cdot x = b(t)$ is as follows. If $q(t) = 0$ then the solution space is

$$v_\mathrm{p}(t) + \langle v_1(t), \ldots, v_d(t) \rangle,$$

otherwise, the solution space is empty.

Note that v_p can be computed even if no particular solution exists. The theorem only covers equation systems with one parameter. Lifting it to several parameters is straightforward.

6.5 Practical Implications

Theorem 19 imposes a condition on the parameters for which the results of the Gaussian algorithm can be used safely. The condition can be obtained by a minor extension of the algorithm: some intermediate results need to be returned.

Returning a condition for the correctness of the solution is more practical than interactively querying the user (see Section 2.1.2). An algorithm using the Gaussian algorithm can be designed to analyse the condition in an appropriate way if knowledge about the parameters is available. Otherwise suitable conditions should be passed on.

Solutions for parameters t with $p(t) = 0$ can be obtained by solving $A(t) \cdot x = b(t)$. A sequence of calls to Gaussian elimination over different domains may be necessary to resolve various special cases, or if the equation system has several parameters. A complete analysis of the parameter space leads to a *system* of solutions that can be represented as a tree with conditions on the parameter space as nodes and solutions as leaves.

6.6 A Theorem Template for the Correct Result

For an integration of parametric Gaussian elimination with a prover, it does not seem sensible to solve the equation system for all special cases automatically. This might be expensive to do and furthermore not even necessary, if some cases are not relevant for the application. Instead we only report the precondition. This leads to the theorem template

$$
\begin{aligned}
&\forall t.\, p(t) \neq 0 \Longrightarrow \\
&\qquad (q(t) = 0 \Longrightarrow (\forall x.\, A(t) \cdot x = b(t) \Longleftrightarrow \\
&\qquad\qquad\qquad x \in v_{\mathrm{p}}(t) + \langle v_1(t), \dots , v_d(t)\rangle))\wedge \\
&\qquad (q(t) \neq 0 \Longrightarrow \neg(\exists x.\, A(t) \cdot x = b(t))).
\end{aligned}
$$

for the result of the computation. Note that A and b are arguments of the computation and p, q, v_{p}, d and the v_i are its results. The analysis of the conditions $p(t) \neq 0$ and $q(t) = 0$ depends on the context. Provers are generally better at handling such knowledge than computer algebra systems. Therefore it is reasonable to decide these conditions in the prover, with possible aid of the computer algebra system — for example, to evaluate the polynomials p and q for given values of t.

6.7 Summary

Our approach to ensure a sound integration of the computer algebra component relies on precise semantics of the algorithms. This prohibits the blind use of Gaussian elimination over polynomial domains for parametric equation systems and avoids mistakes like the one shown in the introduction to this chapter. Surprisingly, such an analysis of Gaussian elimination seems never to have been published, despite Lipson's original suggestion dates 30 years back. The only publication we are aware of is by Corless and Jeffrey (1997), who essentially show that the row echelon form is continuous for $p(t) \neq 0$. This analytical treatment is not appropriate for finite domains, for example. Our analysis is algebraic. It is based on the evaluation homomorphism, which links polynomials and parameters, and hence is more general.

An important design decision is not to compute a system of solutions that covers the entire parameter space at once. Parameters can represent arbitrary expressions and their analysis may require additional knowledge.

An algorithm that performs a analysis over the complete parameter space directly has been presented by Sit (1992). It determines semi-algebraic sets that cover the parameter space. On these sets solutions are uniform. Our naive analysis based on the stairs of the row echelon form may lead to solution systems where paths can be empty because they contain contradicting conditions or case-splits that do not lead to special solutions, but only a different sequence of row transformations is required. Sit's algorithm does not have this disadvantage. On the other hand, Sit's algorithm is complex and it is likely that it is of advantage only for very large equation systems.

The more general case of parametric polynomial equations has already been solved in principle, too. *Comprehensive* Gröbner bases (Becker and Weispfenning, 1993) are systems of Gröbner bases that cover the parameter space. Comprehensive Gröbner bases may contain paths with contradicting conditions, like our naive analysis.

Finally, for an approach to integrating theorem proving and computer algebra without trust, our analysis shows where proof reconstruction would fail.

Chapter 7

Conclusion

Our work shows that the use of computer algebra within a prover is beneficial to reasoning in classical algebra. Most beneficial is the use of algorithms to solve specific problems like factoring a polynomial or solving a linear equation system.

Simplifiers as implemented in computer algebra systems can be used only to a limited extent. Computation of normal forms is sound in some domains. In other domains the implemented algorithms are *ad hoc* or affected by the specialisation problem. On top of that, these simplifiers are not flexible. It is hardly possible to influence their behaviour. The latter is often required in proof and simplifiers in provers like Isabelle can easily be configured for the domain in question by supplying rewrite-rules. Often the simplifier is invoked several times with different settings in a single proof. The price for this flexibility is that simplifiers in provers are rather inefficient compared to computer algebra systems.

7.1 The Risk of Unsoundness

The design of our interface between the prover Isabelle and the computer algebra library Sumit is motivated by the pragmatics of interactive proof. The interface relies on trusting the library, because mathematical algorithms are generally hard to verify formally. It is necessary to formalise the correctness-proof and also the underlying theory. The example of Feit and Thompson's theorem, where the correctness of two lines of code depends on 250 pages of informal proof, shows that the structural complexity of an algorithm can be completely unrelated to the size of the proof. Also, the verification of every single result obtained by executing an implementation can be as hard as verifying the algorithm itself.

Two measures minimise the risk that unsoundness is introduced to the prover by relying on potentially unsound implementations of computer algebra systems. The first measure is an analysis of computer algebra systems. These are powerful computational tools but not logically sound reasoning systems. Reasons for unsoundness fall into three categories.

- Computer algebra systems present a *misleadingly uniform interface* to collections of algorithms. It is the user's responsibility to ensure that the semantics of objects are compatible when they are combined.

- The *specialisation problem* arises, because symbols do not represent logical variables, and mechanisms to handle side-conditions and case-splits are insufficient, if available at all.

- Many algorithms are *ad hoc*: they are not correct, either because correct algorithms are not known, or because the framework of computer algebra systems it too limited to implement them in a clean way.

The second measure is to make the interface modular. Only individual algorithms are made available to the prover, together with formal specifications. Modularity enforces the provision of formal specifications for every single algorithm. These have to be designed carefully. Often it will be necessary to consult their correctness proofs in the literature. The reuse of implementations is essential. Re-implementing is error-prone in itself and laborious too. We use the library Sumit instead of a monolithic computer algebra system. In ordinary computer algebra systems the code for simplification is spread all over the system. This is not the case for modern, typed libraries of computer algebra, like Sumit. Here, algorithms are implemented in modules with well-defined interfaces. Simplification is limited to the computation of normal forms in data-types. Modularity is enforced by the use of a library. Its use prevents to import a large number of algorithms at once, based on a vague notion of the system returning something equal to the argument in some sense, as could be anticipated by the user interfaces of computer algebra systems.

Of course, the degree of rigour that can be achieved when trusting an implementation is limited. This is not acceptable if the prover is used to generate calculus-level proofs of theorems, say, in order to investigate foundational issues. But the use of computer algebra is probably irrelevant in such applications anyway. On the other hand, for example in a verification application, our approach presents a way of importing knowledge from a trusted external source. It is more reliable than to do this manually, one is forced to think about the specifications of every single algorithm, and the dependencies on the external source are recorded in the theorems that are proved. Certainly any computer algebra system contains implementation errors, but bugs of that kind should not be more frequent than in other software, including provers.

7.2 Issues of Interactive Proof Development

A reason for the success of LCF-style theorem provers is their flexibility. Difficult steps in a proof can be guided manually by the user, while routine tasks can be automated. This is done either by supplying specialised tactics or, less powerful but more convenient, by configuring tactics that provide proof-tools. Our interface supports this flexibility. Arguments to computations are expected in Isabelle's term language. Terms extracted from the proof-state can be given directly to the interface. Results of computations are available as theorems and can be used in any way the logical inference mechanisms of the prover permits. In particular, they can be used to implement tactics and simplification procedures. This was useful in the case study about coding theory. Many reductions to normal forms of polynomials were performed automatically by simplification procedures that invoked the computer algebra system.

The main effort in the case study was to mechanise the mathematical background. This was necessary in order to reason about coding theory in a prover. Even for areas of mathematics that are well-established mechanisation is not straightforward. Choosing the right primitives that are suitable for the automation of proofs usually requires some experimentation. A good strategy is to make them as simple as possible. Our set of constructors for sparse univariate polynomials (namely, monom, $\cdot_s$ and $+$) is simple and turned out to be a good choice. This was not obvious from the beginning: association lists are used in computer algebra systems. Similarly, the set of lemmas suitable as primitive derived rules for the degree-function is important. In informal mathematics such lemmas are never given explicitly, but are considered immediate or "obvious" from the definition. The definition of the degree function uses Hilbert's ϵ-operator and is hard to reason about directly.

7.3 Achievements for Computer Algebra

A critical analysis of computer algebra systems from the perspective of an application in theorem proving can be beneficial for the design of future computer algebra systems. Our classification of soundness-problems may help to provide more flexible frameworks for these systems, where the soundness-problems can be avoided.

The use of theorem proving technology and also of methods from artificial intelligence has been suggested by Calmet and Campbell (1997) and Martin (1999). Modern computer algebra systems are used in a variety of applications. Each of them has its specific needs. Fateman (1996a) notes that one of the problems of computer algebra systems may be that they try to cater for too many applications at the same time. The use of theorem proving techniques will lead to an increased use of heuristics in computer algebra. In educational applications some logical underpinning could be helpful to guide users to use correct algorithms and heuristics to help find solutions. Beeson (1995) has domonstrated this for simplification. On the other hand, in computationally intensive applications, heuristics will be less welcome, because of the search involved. An expert user might not like heuristics if they make the system's behaviour hard to predict.

Computer algebra systems do not provide decision procedures — for example, for inequalities over linear arithmetic. A decision procedure would have been helpful in this work to reason about indices of coefficients in the mechanisation of polynomials, but unfortunately was not available in Isabelle. While other provers implement such procedures, they could be useful in computer algebra systems, too.

Our main suggestion for improvement of computer algebra systems is much more humble. The analysis of the Gaussian algorithm in Chapter 6 demonstrates that algorithms become more usable if side-conditions that are created during a computation are recorded and reported as a part of the result.

Appendix A

Mechanisation of Algebra: Detailed Proofs

A.1 Polynomials Form a Ring

Theorem 8 *The univariate polynomials $R[X]$ over a ring R, together with the operations*

$$\begin{aligned} p + q &\equiv (n \mapsto p_n + q_n) \\ p \cdot q &\equiv (n \mapsto \sum_{k=0}^{n} p_k \cdot q_{n-k}) \\ -p &\equiv (n \mapsto -p_n) \\ 0 &\equiv (n \mapsto 0) \\ 1 &\equiv (n \mapsto \mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ 0) \end{aligned}$$

form a ring.

Proof. It needs to be shown that the operations are closed over $R[X]$ and that the ring axioms hold. Closedness is shown by giving appropriate bounds. As p and q are polynomials, they have bounds. The maximum of these bounds is a bound for the sum $p+q$. The sum s of these bounds is a bound for the product $p \cdot q$: for every $\nu > s$, in each summand either p_i or $q_{\nu-i}$ is zero, and so the entire sum. A bound for p is also a bound for $-p$, zero is a bound for the functions 0 and 1.

The ring axioms A1 to A4 are lifted from the coefficient domain. For example, for associativity A1

$$p + q = (n \mapsto p_n + q_n) = (n \mapsto q_n + p_n) = q + p.$$

Polynomials are shown to be equal by showing that the corresponding coefficients are the same. For associativity M1, for the coefficient of degree n

$$\sum_{j=0}^{n} \left(\sum_{i=0}^{j} p_i \cdot q_{j-i} \right) \cdot r_{n-j} = \sum_{j=0}^{n} p_j \cdot \left(\sum_{i=0}^{n-j} q_i \cdot r_{n-j-i} \right)$$

needs to be shown. This follows for $k = n$ from the lemma

$$k \leq n \implies \sum_{j=0}^{k} \left(\sum_{i=0}^{j} p_i \cdot q_{j-i} \right) \cdot r_{n-j} = \sum_{j=0}^{k} p_j \cdot \left(\sum_{i=0}^{k-j} q_i \cdot r_{n-j-i} \right), \qquad \text{(A.1)}$$

which is proved by induction over k. The condition $k \leq n$ guarantees that differences like $n - j$ are well-behaved. It is hard to reason about the difference-operator "$-$" over $\mathbb{N}$ formally, because it is only linear if the first argument is greater than or equal to the second one. The base case of the induction is $(p_0 \cdot q_0) \cdot r_n = p_0 \cdot (q_0 \cdot r_n)$. For the induction step k is substituted by $k+1$ in (A.1). The summations are unfolded and

$$\begin{aligned} \sum_{j=0}^{k} \left(\sum_{i=0}^{j} p_i \cdot q_{j-i} \right) \cdot r_{n-j} + \left(\sum_{i=0}^{k} p_i \cdot q_{k+1-i} \right) \cdot r_{n-(k+1)} + (p_{k+1} \cdot q_0) \cdot r_{n-(k+1)} \\ = \sum_{j=0}^{k} p_j \cdot \left(\sum_{i=0}^{k-j} q_i \cdot r_{n-j-i} \right) + \sum_{j=0}^{k} p_j \cdot (q_{k+1-j} \cdot r_{n-j-(k+1-j)}) \\ + p_{k+1} \cdot (q_0 \cdot r_{n-(k+1)}) \end{aligned}$$

is obtained. The second and third terms on both sides are equal, because of distributivity. The first terms are also equal, by induction hypothesis.

Axiom M2, $1 \cdot p = p$, holds, because the coefficients of degree n on both sides are equal:

$$\sum_{i=0}^{n} (\text{if } i = 0 \text{ then } 1 \text{ else } 0) \cdot p_{n-i} = \sum_{i=0}^{0} p_{n-i} = p_n$$

Distributivity D is also straightforward. For coefficient of degree n

$$\sum_{i=0}^{n} (p_i + q_i) \cdot r_{n-i} = \sum_{i=0}^{n} p_i \cdot r_{n-i} + \sum_{i=0}^{n} q_i \cdot r_{n-i}$$

needs to be shown. This follows from distributivity and commutativity of the coefficient domain.

Finally, commutativity C is proved by an induction similar to the one for associativity. Namely, the lemma

$$j \leq n \implies \sum_{i=0}^{j} p_i \cdot q_{n-i} = \sum_{i=0}^{j} p_{n-i-(n-j)} \cdot q_{i+(n-j)}$$

has to be shown. The induction is over j and the induction step is

$$\sum_{i=0}^{j} p_i \cdot q_{n-i} + p_{j+1} \cdot q_{n-(j+1)}$$

$$= p_{n-(n-(j+1))} \cdot q_{n-(j+1)} + \sum_{i=0}^{j} p_{n-(i+1)-(n-(j+1))} \cdot q_{i+1+(n-(j+1))}.$$

Both single terms are equal. The summation on the right side is equal to $\sum_{i=0}^{j} p_{n-i-(n-j)} \cdot q_{i+(n-j)}$ and the equation follows by induction hypothesis. □

A.2 Universal Property of Polynomials

Theorem 12 (Universal property) *Let R and S be rings. Then for every pair (ϕ, a), where $\phi : R \to S$ is a ring homomorphism and $a \in S$, there exists a unique homomorphism Φ such that the following diagram commutes:*

$$\begin{array}{ccc} R & \xrightarrow{\text{const}} & R[X] \\ {\scriptstyle\phi}\downarrow & & \downarrow{\scriptstyle\Phi} \\ S & \xrightarrow[\text{id}]{} & S \end{array}$$

Proof. In order to prove the existence one defines

$$\Phi : p \mapsto \sum_{i=0}^{\deg p} \phi p_i \cdot a^i \tag{A.2}$$

for given a and ϕ. This is a ring homomorphism. For addition

$$\Phi(p+q) = \sum_{i=0}^{\deg(p+q)} \Phi\ (p+q)_i \cdot a^i = \sum_{i=0}^{\max(\deg p)(\deg q)} \Phi\ (p+q)_i \cdot a^i,$$

because $(p+q)_i = 0$ for $\deg(p+q) < i$. Here $(p+q)_i$ denotes the ith coefficient of $p+q$. With distributivity and commutativity in R, and because ϕ commutes over addition, one obtains

$$\Phi(p+q) = \sum_{i=0}^{\max(\deg p)(\deg q)} \Phi p_i \cdot a^i + \sum_{i=0}^{\max(\deg p)(\deg q)} \Phi q_i \cdot a^i.$$

The coefficients p_i are 0 for $i > \deg p$, and likewise for q_i. The range of summation is changed again and

$$\Phi(p+q) = \sum_{i=0}^{\deg p} \Phi p_i \cdot a^i + \sum_{i=0}^{\deg q} \Phi q_i \cdot a^i = \Phi p + \Phi q.$$

Verifying the same property for multiplication is more difficult. First

$$j \leq n+m \implies \sum_{k=0}^{n+m} \sum_{i=0}^{k} f_i \cdot g_{k-i} = \sum_{k=0}^{n+m} \sum_{i=0}^{j-k} f_k \cdot g_i \tag{A.3}$$

is shown by induction over j, for functions $f, g : \mathbb{N} \to R$. For $j = 0$, both sides of the equation become $f_0 \cdot g_0$, thus the equation holds. The induction step is

$$\begin{aligned}
\sum_{k=0}^{j+1} \sum_{i=0}^{k} f_i \cdot g_{k-i} &= \sum_{k=0}^{j} \sum_{i=0}^{k} f_i \cdot g_{k-i} + \sum_{i=0}^{j} f_i \cdot g_{j-i+1} + f_{j+1} \cdot g_0 \\
&\underset{\text{(a)}}{=} \sum_{k=0}^{j} \sum_{i=0}^{j-k} f_k \cdot g_i + \sum_{i=0}^{j} f_i \cdot g_{j-i+1} + f_{j+1} \cdot g_0 \\
&= \sum_{k=0}^{j} \left(\sum_{i=0}^{j-k} f_k \cdot g_i + f_k \cdot g_{j-k+1} \right) + f_{j+1} \cdot g_0 \\
&= \sum_{k=0}^{j+1} \sum_{i=0}^{j+1-k} f_k \cdot g_i.
\end{aligned}$$

The induction hypothesis is applied at (a). If n is a bound for f and m a bound for g the following equality for products of sums holds:

$$\sum_{k=0}^{n+m} \sum_{i=0}^{k} f_i \cdot g_{k-i} = \left(\sum_{i=0}^{n} f_i \right) \cdot \left(\sum_{i=0}^{m} g_i \right) \tag{A.4}$$

The left side of this equation is also known as *Cauchy product.* To show that the equation holds first the order of summation is changed, by means of equation (A.3), where $n+m$ is substituted for j. Then parts of the summation, which are zero, are split off:

$$\begin{aligned}
\sum_{k=0}^{n+m}\sum_{i=0}^{k} f_i \cdot g_{k-i} &= \sum_{k=0}^{n+m}\sum_{i=0}^{n+m-k} f_k \cdot g_i \\
&= \sum_{k=0}^{n}\sum_{i=0}^{n+m-k} f_k \cdot g_i + \sum_{k=n+1}^{n+m}\sum_{i=0}^{n+m-k} f_k \cdot g_i \\
&= \sum_{k=0}^{n}\left(\sum_{i=0}^{m} f_k \cdot g_i + \sum_{i=m+1}^{n+m-k} f_k \cdot g_i\right) \\
&= \sum_{k=0}^{n} f_k \cdot \left(\sum_{i=0}^{m} g_i\right) = \left(\sum_{i=0}^{n} f_i\right) \cdot \left(\sum_{i=0}^{m} g_i\right)
\end{aligned}$$

Now $\Phi(p \cdot q) = \Phi p \cdot \Phi q$ can be proved:

$$\begin{aligned}
\sum_{i=0}^{\deg(p \cdot q)} \Phi\,(p \cdot q)_i \cdot a^i &\underset{(a)}{=} \sum_{i=0}^{\deg p + \deg q} \Phi\,(p \cdot q)_i \cdot a^i \\
&\underset{(b)}{=} \sum_{i=0}^{\deg p + \deg q} \sum_{k=0}^{i} \Phi p_k \cdot \Phi q_{i-k} \cdot a^k \cdot a^{i-k} \\
&\underset{(c)}{=} \left(\sum_{i=0}^{\deg p} \Phi p_i \cdot a^i\right) \cdot \left(\sum_{i=0}^{\deg q} \Phi q_i \cdot a^i\right)
\end{aligned}$$

The change of the range of summation at (a) is valid, because $\deg(p \cdot q) \leq \deg p + \deg q$ (4.3), and $(p \cdot q)_i = 0$ for $m > \deg(p \cdot q)$. Equality (b) follows from the definition of polynomial multiplication, and (c) is equation (A.4).

The proof of $\Phi 1_i \cdot a^i = 1$ is trivial. This completes the proof of the existence of Φ. For uniqueness, observe that an arbitrary Φ must map

$$X \mapsto a \quad \text{and} \quad c \,\dot{s}\, X^0 \mapsto \phi c.$$

Therefore, by homomorphism $\Phi p = \sum_{i=0}^{\deg p} \phi p_i \cdot a^i$. So, (A.2) was the only way to define Φ and it is unique. $\square$

Bibliography

Abbott, J. (1996). OpenMath design committee report. Version of 23 December.

Adams, A. A., Gottliebsen, H., Linton, S. A., and Martin, U. (1999). VSDITLU: a verifiable symbolic definite integral table look-up. In *CADE-16*. To appear.

Apostol, T. M. (1974). *Mathematical Analysis*. Addison-Wesley.

Aslaksen, H. (1996). Multiple-valued complex functions & computer algebra. *ACM SIGSAM Bulletin*, **30**(2), 12–20.

Association of Mizar Users, editor (1989). *Journal of Formalized Mathematics*. University of Bialystok. Published on the Internet. See http://mizar.org.

Bailey, A. (1993). *Representing algebra in LEGO*. Master's thesis, University of Edinburgh, Department of Computer Science.

Ballarin, C. (1994). *Algorithmische Schritte in formalen Beweisen — Entwurf und Implementierung der Anbindung eines Computeralgebrasystems an einen Theorembeweiser*. Diplomarbeit, Universität Karlsruhe, Fakultät für Informatik.

Ballarin, C., Homann, K., and Calmet, J. (1995). 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.

Bareiss, E. H. (1968). Sylvester's identity and multistep integer-preserving Gaussian elimination. *Mathematics of Computation*, **22**, 565–578.

Becker, T. and Weispfenning, V. (1993). *Gröbner Bases: A Computational Approach to Commutative Algebra*. Springer-Verlag.

Beeson, M. (1995). Using nonstandard analysis to ensure the correctness of symbolic computations. *International Journal of Foundations of Computer Science*, **6**(3), 299–338.

Benzmüller, C., Cheikhruouhou, L., Fehrer, D., Fiedler, A., Huang, X., Kerber, M., Kohlhase, M., Konrad, K., Meier, A., Melis, E., Schaarschmidt, W., Siekmann, J., and Sorge, V. (1997). Omega: Towards a mathematical assistant. In W. McCune, editor, *Automated deduction, CADE-14: 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia, July 13–17, 1997: proceedings*, number 1249 in Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence, pages 252–255. Springer-Verlag.

Boyer, R. S. and Moore, J. S. (1981). Metafunctions: proving them correct and using them efficiently as new proof procedures. In R. S. Boyer and J. S. Moore, editors, *The correctness problem in computer science*, International Lecture Series in Computer Science, pages 103–213. Academic Press.

Bronstein, M. (1996). Sumit — a strongly-typed embeddable computer algebra library. In Calmet and Limongelli (1996), pages 22–33.

Bundy, A., van Harmelen, F., Hesketh, J., and Smaill, A. (1991). Experiments with proof plans for induction. *Journal of Automated Reasoning*, **7**(3), 303–324.

Butler, G. and Cannon, J. (1989). Cayley, version 4: the user language. In P. Gianni, editor, *Symbolic and Algebraic Computation: International Symposium ISSAC '88, Rome, Italy, July 4–8, 1988: Proceedings*, number 358 in Lecture Notes in Computer Science, pages 456–466. Springer-Verlag.

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

Calmet, J. and Limongelli, C., editors (1996). *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.

Clarke, E. and Zhao, X. (1993). Analytica: A theorem prover for Mathematica. *The Mathematica Journal*, **3**(1), 56–71.

Clarke, E. and Zhao, X. (1994). Combining symbolic computation and theorem proving: some problems of Ramanujan. In A. Bundy, editor, *Automated deduction, CADE-12: 12th International Conference on Automated Deduction, Nancy, France, June 26–July 1, 1994: proceedings*, number 814 in Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence, pages 758–763. Springer-Verlag.

Constable, R. L. (1986). *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall.

Corless, R. M. and Jeffrey, D. J. (1996). The unwinding number. *ACM SIGSAM Bulletin*, **30**(2), 28–35.

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

Cox, D., Little, J., and O'Shea, D. (1992). *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer-Verlag.

Curzon, P. (1994). Experiences formally verifying a network component. In *Proceedings of the 9th Annual IEEE Conference on Computer Assurance*, pages 183–193. IEEE Press.

Dalmas, S. and Gaëtano, M. (1996). Making systems communicate and cooperate: The central control approach. In Calmet and Limongelli (1996), pages 308–319.

Dalmas, S., Gaëtano, M., and Watt, S. (1997). An OpenMath 1.0 implementation. In W. 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.

Davenport, J. (1998). Is computer algebra the same as computer mathematics? Talk given at the British Colloquium for Theoretical Computer Science (BCTCS), St. Andrews, UK.

Davenport, J. H. (1990). Current problems in computer algebra systems design. In Miola (1990), pages 1–9.

Davenport, J. H. and Trager, B. M. (1990). Scratchpad's view of algebra I: Basic commutative algebra. In Miola (1990), pages 40–54.

Davenport, J. H., Gianni, P., and Trager, B. M. (1991). Scratchpad's view of algebra II: A categorical view of factorization. In S. M. Watt, editor, *Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation, ISSAC'91, July 15–17, 1991, Bonn, Germany*, pages 32–38. ACM Press.

Davenport, J. H., Siret, Y., and Tournier, E. (1993). *Computer Algebra: Systems and algorithms for algebraic computation*. Academic Press, second edition.

Davis, M. and Schwartz, J. T. (1979). Metamathematical extensibility for theorem verifiers and proof-checkers. *Computers & mathematics with applications*, **5**(3), 217–230.

Farmer, W. M., Guttman, J. D., and Thayer, F. J. (1993). IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, **11**(2), 213–248.

Fateman, R. J. (1996a). Symbolic mathematics system evaluators. In Lakshman (1996), pages 86–94. Extented abstract, full version available at http://http.cs.berkeley.edu/~fateman/papers/eval.ps.

Fateman, R. J. (1996b). Why computer algebra systems sometimes can't solve simple equations. *ACM SIGSAM Bulletin*, **30**(2), 8–11.

Fateman, R. J. (1998). Re: MMA's equivalent of Maple's assume. Post to the news group sci.math.symbolic.

Feit, W. and Thompson, J. G. (1963). Solvability of groups of odd order. *Pacific Journal of Mathematics*, **13**(3), 775–1029.

Geddes, K. O., Czapor, S. R., and Labahan, G. (1992). *Algorithms for Computer Algebra*. Kluwer Academic Publishers.

Giunchiglia, F., Pecchiari, P., and Talcott, C. (1996). Reasoning theories: Towards an architecture for open mechanized reasoning systems. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems — First International Workshop, FroCoS'96, Munich, Germany, March 1996*, number 3 in Applied logic series, pages 157–174. Kluwer Academic Publishers.

Gonnet, G. H., Dalmas, S., Gaëtano, M., Watt, S., and Huuskonen, T. (1998). *Basic Content Dictionary*. Available on the Internet at http://www.openmath.org/CDs/Ver1.1/Basic.

Gordon, M. J., Milner, R., and Wadsworth, C. P. (1979). *Edinburgh LCF*. Number 78 in Lecture Notes in Computer Science. Springer-Verlag.

Gordon, M. J. C. and Melham, T. F., editors (1993). *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press.

Gray, S., Kajler, N., and Wang, P. S. (1996). Pluggability issues in the Multi protocol. In Calmet and Limongelli (1996), pages 343–356.

Harrison, J. (1997). Floating point verification in HOL Light: the exponential function. Technical Report 428, University of Cambridge, Computer Laboratory.

Harrison, J. and Théry, L. (1994). Extending the HOL theorem prover with a computer algebra system to reason about the reals. In J. J. Joyce and C.-J. H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications: 6th International Workshop, HUG '93: Vancouver, B.C., Canada, August 11–13, 1993: Proceedings*, number 780 in Lecture Notes in Computer Science, pages 174–184. Springer-Verlag.

Harrison, J. R. (1996). *Theorem Proving with the Real Numbers*. Ph.D. thesis, University of Cambridge, Computer Laboratory. Available as Technical Report 408.

Hewitt, C. (1971). Procedure embedding of knowledge in Planner. In *Second International Joint Conference on Artificial Intelligence : IJCAI-71 : 1–3 September 1971 : Imperial College, London*, pages 167–182. Morgan Kaufmann for the British Computer Society.

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

Hofri, M. (1995). *Analysis of Algorithms: Computational Methods & Mathematical Tools*. Oxford University Press.

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

Huuskonen, T. (1997). Re: Description of OpenMath. Private communication.

Ion, P., Miner, R., Buswell, S., Devitt, S., Diaz, A., Poppelier, N., Smith, B., Soiffer, N., Sutor, R., and Watt, S. (1998). Mathematical markup language (MathML) 1.0 specification. Technical Report REC-MathML-19980407, The World Wide Web Consortium. Available at http://www.w3.org/TR/REC-MathML.

Jackson, P. B. (1995). Enhancing the Nuprl proof development system and applying it to computational abstract algebra. Technical Report CORNELLCS//TR95-1509, Cornell University, Department of Computer Science.

Jacobson, N. (1985). *Basic Algebra*, volume I. Freeman, 2nd edition.

Jacobson, N. (1989). *Basic Algebra*, volume II. Freeman, 2nd edition.

Jenks, R. D. and Sutor, R. S. (1992). *AXIOM. The scientific computation system*. Numerical Algorithms Group, Ltd. and Springer-Verlag, Oxford and New York.

Kajler, N. (1993). Building a computer algebra environment by composition of collaborative tools. In J. Fitch, editor, *Design and implementation of symbolic computation systems: International Symposium, DISCO '92, Bath, U.K., April 13–15 1992: proceedings*, number 721 in Lecture Notes in Computer Science, pages 85–94. Springer-Verlag.

Kerber, M., Kohlhase, M., and Sorge, V. (1996). Integrating computer algebra with proof planning. In Calmet and Limongelli (1996), pages 204–215.

Lakshman, Y. N., editor (1996). *ISSAC 96: Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation — July 24–26, 1996, Zürich, Switzerland*. ACM Press.

Lipson, J. D. (1969). Symbolic methods for the computer solution of linear equations with applications to flowgraphs. In R. G. Tobey, editor, *Proceedings of the 1968 Summer Institute on Symbolic Mathematical Computation*, pages 233–303, Maryland. IBM Federal Systems Division.

Luo, Z. and Pollack, R. (1992). LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh.

Martin, U. (1999). Computers, reasoning and mathematical practice. In U. Berger and H. Schwichtenberg, editors, *Computational Logic*. Springer-

Verlag. To appear. Currently available at http://www-theory.dcs.st-and.ac.uk/~um/publications/natoasipub.ps.

Martin, U. and Nipkow, T. (1990). Ordered rewriting and confluence. In M. E. Stickel, editor, *10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24–27, 1990: Proceedings*, number 449 in Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence, pages 366–380. Springer-Verlag.

McCune, W. (1994). Otter 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, Argonne, IL, USA.

McCune, W. (1997). Solution of the Robbins problem. *Journal of Automated Reasoning*, **19**(3), 263–276.

Milner, R. (1985). The use of machines to assist in rigorous proof. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, Series in Computer Science, pages 77–88. Prentice Hall International.

Miola, A., editor (1990). *Design and implementation of symbolic computation systems: International Symposium DISCO '90, Capri, Italy, April 10–12, 1990: proceedings*, number 429 in Lecture Notes in Computer Science. Springer-Verlag.

Moses, J. (1971). Symbolic integration: The stormy decade. *Communications of the ACM*, **14**(8), 548–560.

Nederpelt, R. P., Geuvers, J. H., and de Vrijer, R. C., editors (1994). *Selected Papers on Automath*. Number 133 in Studies in Logic. Elsevier.

Newell, A. (1983). Intellectual issues in the history of artificial intelligence. In F. Machlup and U. Mansfield, editors, *The Study of Information — Interdisciplinary Messages*, chapter 3, pages 187–227. John Wiley & Sons.

Nipkow, T. (1993). Order-sorted polymorphism in Isabelle. In G. Huet and G. Plotkin, editors, *Logical environments*, pages 164–188. Cambridge University Press.

Noble, B. and Daniel, J. W. (1988). *Applied linear algebra*. Prentice-Hall, 3rd edition.

Norman, A. and Fitch, J. (1996). Memory tracing of algebraic calculations. In Lakshman (1996), pages 113–119.

Norrish, M. (1999). *Plug-in Interface User Documentation: Deliverable D3.2b.V1-09*. PROSPER: ESPRIT LTR Project 26241. Available at http://www.cl.cam.ac.uk/users/mn200/prosper/.

Paulson, L. (1983). A higher-order implementation of rewriting. *Science of Computer Programming*, **3**, 119–149.

Paulson, L. C. (1994). *Isabelle: a generic theorem prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag.

Paulson, L. C. (1997a). Inductive analysis of the Internet protocol TLS. Technical Report 440, University of Cambridge, Computer Laboratory.

Paulson, L. C. (1997b). The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory. 4th edition.

Paulson, L. C. (1998). A generic tableau prover and its integration with Isabelle. Technical Report 441, University of Cambridge, Computer Laboratory.

Pratt, V. R. (1975). Every prime has a succinct certificate. *SIAM Journal on Computation*, **4**(3).

Purdom, Jr., P. W. and Brown, C. A. (1985). *The Analysis of Algorithms*. CBS Publishing, New York.

Risch, R. H. (1969). The problem of integration in finite terms. *Transactions of the American Mathematical Society*, **139**, 167–189.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, **12**(1), 23–41.

Rudnicki, P. (1992). An overview of the MIZAR project. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Proceedings of the 1992 workshop on types for proofs and programs: Båstad, Sweden, June 1992*, pages 311–332.

Salvy, B. and Zimmermann, P. (1994). GFUN: A Maple package for the manipulation of generating and holoniomic functions in one variable. *ACM Transactions on Mathematical Software*, **20**(2), 163–177.

Schönert, M. (1997). *GAP Manual*. Lehrstuhl D für Mathematik, RWTH Aachen, Aachen, Germany. Available on the Internet at http://www-gap.dcs.st-and.ac.uk/˜gap/.

Scott, D. S. (1993). A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, **121**(1–2), 411–440.

Sit, W. Y. (1992). An algorithm for solving parametric linear systems. *Journal of Symbolic Computation*, **13**, 353–394.

Slind, K. (1993). Adding new rules to an LCF-style logic implementation. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher order logic theorem proving and its applications: proceedings of the IFIP TC10/WG10.2 international workshop on higher order logic theorem proving and its applications: HOL '92: Leuven, Belgium, 21–24 September 1992*, number A-20 in IFIP Transactions, pages 549–559. North-Holland.

Slind, K. (1998). A tagged LCF-style proof architecture. In D. Basin and L. Vigano, editors, *Participants' proceedings of LD'98: First International Workshop on Labelled Deduction, Freiburg, Germany*. Also available at http://www.cl.cam.ac.uk/users/kxs/papers/tag.html.

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

Théry, L. (1998). 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.

Watt, S. M., Broadbery, P. A., Dooley, S. S., Iglio, P., Morrison, S. C., Steinbach, J. M., and Sutor, R. S. (1994a). *AXIOM Library Compiler User Guide*. The Numerical Algorithms Group Limited, Oxford.

Watt, S. M., Broadbery, P. A., Dooley, S. S., Iglio, P., Morrison, S. C., Steinbach, J. M., and Sutor, R. S. (1994b). A first report on the A# compiler. In *ISSAC '94: Proceedings of the 1994 International Symposium on Symbolic and Algebraic Computation: July 20–22, 1994, Oxford, England, United Kingdom*, pages 25–31. ACM Press.

Wenzel, M. (1997). Type classes and overloading in higher-order logic. In E. L. Gunter and A. Felty, editors, *Theorem proving in higher order logics: 10th International Conference, TPHOLs '97, Murray Hill, NJ, USA, August 19–22, 1997: proceedings*, number 1275 in Lecture Notes in Computer Science, pages 307–322. Springer-Verlag.

Wenzel, M. (1998). *Using Axiomatic Type Classes in Isabelle — a tutorial*. Available with the Isabelle-distribution.
See http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.

Wolfram, S. (1996). *The Mathematica book*. Wolfram Media and Cambridge University Press, third edition.