Thesis for the Degree of Doctor of Philosophy

Formalizing Refinements and Constructive Algebra in Type Theory

Anders Mörtberg

UNIVERSITY OF
GOTHENBURG

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg, Sweden

Gothenburg, 2014

Formalizing Refinements and Constructive Algebra in Type Theory
Anders Mörtberg
ISBN 978-91-982237-0-5

Technical Report no. 115D96
Department of Computer Science and Engineering
Programming Logic Research Group

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg, Sweden

Printed at Chalmers Reproservice
Gothenburg, Sweden 2014

Abstract

The extensive use of computers in mathematics and engineering has led to an increased demand for reliability in the implementation of algorithms in computer algebra systems. One way to increase the reliability is to formally verify that the implementations satisfy the mathematical theorems stating their specification. By implementing and specifying algorithms from computer algebra inside a proof assistant both the reliability of the implementation and the computational capabilities of the proof assistant can be increased.

This first part of the thesis presents a framework, developed in the interactive theorem prover Coq, for conveniently implementing and reasoning about program and data refinements. In this framework programs defined on rich dependent types suitable for proofs are linked to optimized implementations on simple types suitable for computation. The correctness of the optimized algorithms is established on the proof-oriented types and then automatically transported to the computation-oriented types. This method has been applied to develop a library containing multiple algorithms from computational algebra, including: Karatsuba's polynomial multiplication, Strassen's matrix multiplication and the Sasaki-Murao algorithm for computing the characteristic polynomial of matrices over commutative rings.

The second part of the thesis presents the formalization of notions from constructive algebra. Focus is on the theory of coherent and strongly discrete rings, which provides a general setting for developing linear algebra over rings instead of fields. Examples of such rings include Bézout domains, Prüfer domains and elementary divisor rings. Finitely presented modules over these rings are implemented using an abstraction layer on top of matrices. This enables us to constructively prove that the category of these modules form a suitable setting for developing homological algebra. We further show that any finitely presented module over an elementary divisor ring can be decomposed to a direct sum of a free module and cyclic modules in a unique way. This decomposition gives a decision procedure for testing if two finitely presented modules are isomorphic.

List of publications

This thesis is based on the work contained in the following papers:

1. A Refinement-Based Approach to Computational Algebra in Coq. Maxime Dénès, Anders Mörtberg and Vincent Siles. In *Interactive Theorem Proving*, volume 7406 of *Lectures Notes in Computer Science*, pages 83–98. Springer, 2012.

2. Refinements for free!. Cyril Cohen, Maxime Dénès and Anders Mörtberg. In *Certified Programs and Proofs*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013.

3. A Formal Proof of Sasaki-Murao Algorithm. Thierry Coquand, Anders Mörtberg and Vincent Siles. *Journal of Formalized Reasoning*, 5(1):27–36, 2012.

4. Coherent and Strongly Discrete Rings in Type Theory. Thierry Coquand, Anders Mörtberg and Vincent Siles. In *Certified Programs and Proofs*, volume 7679 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2012.

5. A Coq Formalization of Finitely Presented Modules. Cyril Cohen and Anders Mörtberg. In *Interactive Theorem Proving*, volume 8558 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2014.

6. Formalized Linear Algebra over Elementary Divisor Rings in Coq. Guillaume Cano, Cyril Cohen, Maxime Dénès, Anders Mörtberg and Vincent Siles. Preprint, 2014.

The following papers are related but not included in the thesis:

Towards a Certified Computation of Homology Groups for Digital Images. Jónathan Heras, Maxime Dénès, Gadea Mata, Anders Mörtberg, María Poza and Vincent Siles. In *Computational Topology in Image Context*, volume 7309 of *Lecture Notes In Computer Science*, pages 49–57. Springer, 2012.

Computing Persistent Homology Within Coq/SSReflect. Jónathan Heras, Thierry Coquand, Anders Mörtberg and Vincent Siles. *ACM Transactions on Computational Logic*, 14(4):1–26, 2013.

Statement of contribution

The authors' contributions to each of the papers included in the thesis are:

1. Implementation of list based polynomials and the algebraic hierarchy of computational structures. Wrote the sections on these and participated in writing the other sections.
2. Collaborated on developing the library and designing the methodology. Implemented the theory on sparse polynomials. Writing was distributed evenly among the authors.
3. Mainly contributed to the formalization of the correctness proof and the implementation of the efficient version of the algorithm.
4. Contributed to all parts, both formalization and writing. Implemented the executable versions and correctness proof of the algorithms and structures.
5. Collaborated on the formalization and wrote most parts of the paper.
6. Contributed to all parts of the formalization, in particular to the theory of rings with explicit divisibility, the proof that elementary divisor rings are coherent and the development on the Kaplansky condition. Wrote parts of most sections of the paper.

Contents

Acknowledgments

First of all I would like to thank my supervisor Thierry Coquand for the support and guidance that has made it possible for me to finish this thesis. I am also grateful to my co-supervisor Cyril Cohen, who together with Vincent Siles taught me Coq and SSReflect.

Most of my time as a PhD student has been spent in an office shared with Bassel Mannaa, Guilhem Moulin and Simon Huber who have contributed to my work not only by discussions about research but also by being great friends. I would like to thank everyone I have written papers with, the members of my follow up group, my colleagues in the ForMath project and in the Programming Logic group at Chalmers. I am also grateful to everyone else at the Department of Computer Science and Engineering for making my time here so enjoyable.

I would like to thank Anjelica Hammersjö for her patience and encouragement, and my family for always being helpful and supportive. A special thanks to my cats for not being supportive at all, sitting on my keyboard and forcing me to take breaks sometimes.

The comments from Andrea Vezzosi, Anjelica Hammersjö, Bassel Mannaa and Cyril Cohen on earlier versions of this thesis have been very helpful. The answers from Ramona Enache to my numerous questions about practical issues have made my life much easier and our walks around Guldheden have been lots of fun. I would also like to thank all of the friends that I have made in Gothenburg. Especially the three friends I met on my first day as an undergraduate student at Chalmers nine years ago: Daniel Gustafsson, Pontus Lindström and Ulf Liljengren. Thanks to all of you I have learned so many things and had lots of fun during all these years!

The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

Introduction

Computers are playing an increasingly important role in modern mathematics. Computer algebra systems like MATLAB and MATHEMATICA are fundamental tools in engineering, scientific computation and also to some extent in pure mathematics. It is important that these systems are correct – but this is not always the case [Durán et al., 2014]. For instance, MATLAB had a bug in 2009 that made it compute an incorrect solution to a very simple system of equations [Walking Randomly, 2009]. This is problematic as equation solving is one of the most fundamental operations in MATLAB on which more complex operations are based.

Many people can relate to the frustration caused by software that is not working correctly. Faulty software can also have more serious consequences: It could cost a lot of money or – even worse – human lives. Famous examples include the Pentium FDIV bug [Cipra, 1995] that costed Intel about $475 million [Nicely, 2011] and in 1991 when a U.S. Patriot missile defense system in Saudi Arabia failed to detect an attack because of a miscalculation due to rounding errors which led to the death of 28 soldiers [Marshall, 1992].

This thesis uses techniques from software verification to facilitate the implementation of formally verified programs and mathematical theories in type theory, more precisely in the COQ proof assistant [COQ Development Team, 2012]. A potential use of this is to increase the reliability in computer algebra systems and software used in safety critical applications, reducing the risk for problems like those mentioned above.

1 Software verification

The standard approach for increasing the reliability in software is **testing**. This usually means that the programmer writes tests and collects them in a test-suite, or that a user tests the system and reports any errors found. However, these two approaches are both problematic since bugs are often found in corner-cases that are hard to find and for which it is difficult to write good tests. Because of this, bugs might be found too late.

Another approach is to use a tool designed for randomized testing like QUICKCHECK [Claessen and Hughes, 2000]. This means that the programmer writes a logical specification of the program and the tool generates random input to test the specification with. This has the benefit that the programmer is relieved of the tedious task of producing test-suites and that the specification of the program has to be taken into consideration when writing it.

However, regardless of its usefulness, testing has a fundamental limitation as pointed out by Dijkstra:

> *"Program testing can be used to show the presence of bugs, but never to show their absence!"* [Dijkstra, 1970]

So the question is: how can one show the *absence* of bugs?

The answer to this is **proofs**, that is, by a convincing argument that the program satisfies its specification. There are many ways to prove the correctness of a program. The traditional method is to write the proof by hand. This has the benefit that the proof can be read and verified by another person, however there might be subtle errors that are hard to spot and corner-cases could be overlooked. An alternative approach, that aims at overcoming these issues, is to use computers for both finding and verifying proofs. However, even though one has proved that the underlying algorithm is correct one can still make mistakes when implementing it. The correctness proof should hence be about the actual implementation of the program and not only about the underlying algorithm.

A prerequisite to both testing and proving is that the program has been clearly specified. However, specifying a program and then proving that it satisfies the specification might not be enough to guarantee that the program is correct. Donald E. Knuth put it nicely in the end of one of his letters:

> *"Beware of bugs in the above code; I have only proved it correct, not tried it."* [Knuth, 1977]

A reason could be that there are errors in the specification. This kind of errors can stem from an incorrect understanding of what the program is supposed to be doing [Claessen and Hughes, 2000], or from the difficulty of specifying general purpose programs. For mathematical software the situation is slightly better. The reason for this is that a mathematical algorithm usually has a clear specification in terms of a mathematical theorem stating what is required of the input and expected by the output.

In order to make it easier to correctly specify programs it is important to write them in a suitable programming language. In the **functional programming** paradigm functions are first-class objects, that is, they can be passed as arguments to other functions or be returned by functions. This makes it possible for the programmer to find good abstractions and write short general programs. The restricted use of side-effects also makes functional programming well-suited for verification. When a function has no side-effects, it is much easier to specify what it is supposed to be doing because of referential transparency [Strachey, 2000].

One approach that is used a lot in industry to prove the correctness of complex hardware designs and for verifying concurrent and distributed software is **model checking**. This involves constructing a model of a system and then checking that a property is satisfied by the model. The properties are usually expressed in a temporal logic and might state safety properties like absence of deadlocks or race conditions. There are many tools for doing this kind of verification, used in both research and industry, like the SPIN model

checker [Holzmann, 2004] that received the prestigious ACM Software System Award in 2001 [ACM, 2001].

Another approach to automatic verification is **automated theorem proving** which involves writing programs to automatically prove mathematical theorems expressed in a logical system. Depending on the logical system, the properties of these programs varies. For instance if the underlying logic is propositional logic the problem of satisfiability of formulas (SAT) is decidable, simply by constructing truth-tables. However, as the SAT problem is NP-complete there can only be exponential time algorithms (unless the complexity classes P and NP are the same). Regardless there are many algorithms and solvers that very efficiently solve many classes of problems, which makes them feasible to use in practice.

In first-order logic, on the other hand, logical entailment is only semidecidable. This means that it is possible to write a program that finds a proof of a formula if it is provable, but if the formula is not provable the program might not terminate. There are many efficient tools that automatically try to prove first-order formulas, an example is the Vampire prover [Kovacs and Voronkov, 2013] that has won more than 30 titles in the "world cup for theorem provers".

It is also common to consider decision problems for logical formulas with respect to some background theory. This class of problems is called Satisfaction Modulo Theories (SMT). The theories are often expressed using first-order logic and can be axiomatizations of, for example, integers or lists. Attention is usually restricted to decidable theories, like Presburger arithmetic (natural numbers with addition), dense linear orders or real-closed fields. This way the correctness of programs that use these theories can be automatically verified using SMT solvers like CVC4 [Barrett et al., 2011] or Microsoft's Z3 solver [De Moura and Bjørner, 2008].

Fully automatic theorem provers have also been used to prove some interesting conjectures in mathematics. The most famous might be the proof of the Robbins conjecture, stated back in 1933, that asks whether all Robbins algebras are Boolean algebras. This problem has a simple statement, but its solution eluded many famous mathematicians and logicians, including Tarski, for many years. However, in 1996 William McCune found a proof of the conjecture using the automated theorem prover EQP [McCune, 1997].

Fully automated tools have both their pros and cons. On the one hand they are easy to use, but on the other the resulting proofs might be very big and not comprehensible by humans. In February 2014 fields medalist Timothy Gowers wrote on his blog [Gowers's Weblog, 2014] that a paper had appeared on the arXiv with an interesting result about the Erdos discrepancy problem. In the paper a certain bound is proved to be the best possible and interestingly the proof of this was found using a SAT solver. But the authors say in the paper that:

> *"The negative witness, that is, the DRUP unsatisfiability certificate, is probably one of longest proofs of a non-trivial mathematical result ever produced. Its gigantic size is comparable, for example, with the size of the whole Wikipedia, so one may have doubts about to which degree this can be accepted as a proof of a mathematical statement."* [Konev and Lisitsa, 2014]

A possible solution to increase the reliability in this kind of results is to implement a checker for verifying the certificate. However, this will rely on the correctness of the checker, so ideally this should also be verified. A possible approach for doing this is to develop it using an interactive theorem prover.

In **interactive theorem proving** a human writes a proof in some formal language that can be understood by a computer program, which then checks the correctness of the proof by verifying each deductive step. This is a more modest approach than fully automatic proving as checking the correctness of a proof is easier than actually finding it.

The systems used for this kind of theorem proving are called **proof assistants** as they assist the user in writing a formal proof that is then verified by the system. This is often done using an interface that displays what is currently known and which goals has to be proved in order to finish the proof. Many proof assistants, like for example Coq, have a so called *tactic language* for developing proofs. This is a domain specific language in which the user writes tactics that gets executed by the proof assistant and modify the current state of the proof. A tactic could for example apply an inference rule or rewrite some part of the goal using a lemma. Many proof assistants also use automatic techniques, like those mentioned above, in order to help the user discharge easy subgoals. This way formal proofs can be developed interactively by a kind of human-computer collaboration.

Proof assistants date back to the 1960s and the pioneering work of Nicolaas Govert de Bruijn on the Automath system [Nederpelt et al., 1994]. This system was very influential and has inspired many of the modern proof assistants in use today. It was for instance based on a typed lambda calculus and the Curry-Howard correspondence. Since the 1960s many more proof assistants have been developed [Wiedijk, 2006], and in 2013 the Coq proof assistant received the prestigious ACM Software System Award with the motivation:

> *"Coq has played an influential role in formal methods, programming languages, program verification and formal mathematics. As certification gains importance in academic and industrial arenas, Coq plays a critical role as a primary programming and certification tool."* [ACM, 2013]

This kind of tools have been used in some large-scale formalizations in computer science. One of the most impressive is the formal proof of the seL4 microkernel whose functional correctness was formally verified in 2009 [Klein et al., 2009]. This means that the kernel's implementation is free of bugs like deadlocks, buffer overflows, arithmetic errors and so on. This work is estimated to have taken about 25-30 person years to be completed and was performed using the Isabelle/HOL proof assistant [Nipkow et al., 2002].

Another impressive formalization effort, led by Xavier Leroy, is the Coq implementation of the optimizing C compiler CompCert [Leroy, 2006]. The compiler produces PowerPC, ARM and x86 assembly code which is guaranteed to do the same thing as the original C code. In 2011 researchers at the University of Utah wrote a program called Csmith that randomly generates C code and then compared the output of different C compilers. They managed to find bugs in state of the art compilers like GCC and LLVM, but CompCert stood out:

> *"The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task."* [Yang et al., 2011]

The completion and success of these large-scale formalizations indicates that proof assistants are becoming mature enough to develop software that can be used in safety critical systems. However, proof assistants have not only been used to verify impressive results in computer science, but also to formalize many famous theorems in mathematics.

2 Formalization of mathematics

The idea of formalizing mathematics by developing it in a formal system, starting from a basic foundation, is much older than computers. Back in 1879 Gottlob Frege wrote a book called *Begriffsschrift* which presented a logical system with quantifiers, and using this formal language he formalized basic arithmetic in the subsequent volume *Grundgesetze der Arithmetik*. However a few years later, in 1903, Bertrand Russell proved that Frege's system was inconsistent by pointing out his famous paradox about sets containing themselves. Following this, many new foundational systems for mathematics were developed. Some were based on set theory, like *Zermelo and Fraenkel set theory with the axiom of choice* (ZFC). Others on the **type theory** developed by Bertrand Russell and Alfred North Whitehead in order to write *Principia Mathematica*. In type theory each mathematical object has a type and paradoxes like the one of Russell are avoided by stratifying the types of types and not letting a type have itself as a type. Today type theories are used as a foundation for many interactive theorem provers.

One class of type theories are those based on *Church's Simple Theory of Types* [Church, 1940], these provide the basis for proof assistants in the HOL family (*e.g.* HOL4 [Gordon and Melham, 1993; Slind and Norrish, 2008], HOL Light [Harrison, 1996] and Isabelle/HOL [Nipkow et al., 2002]). These are usually "simple" in the sense that types cannot depend on arbitrary terms. Many important mathematical theorems have been formalized in these systems, for instance the prime number theorem [Avigad et al., 2007; Harrison, 2009] and the Jordan curve theorem [Hales, 2007].

Another class are the intuitionistic type theories which have the aim to be used as a foundations for **constructive mathematics**. In constructive mathematics the law of excluded middle and the axiom of choice are not accepted. By avoiding these principles the mathematical theories become inherently computational (see [Bridges and Palmgren, 2013]) which makes them suitable for implementation on computers [Martin-Löf, 1984b].

This class of systems are based on *Martin-Löf type theory* [Martin-Löf, 1984a] and the *Calculus of (Inductive) Constructions* [Coquand and Huet, 1986; Coquand and Paulin, 1990] which are dependently typed theories where types can depend on terms. This has the implication that, by the Curry-Howard correspondence, proofs and propositions correspond to terms and types. This

means that both the programs and their proofs of correctness can be implemented using the same language and logic. Also, checking the correctness of a proof corresponds to checking that a term has a certain type. This is very appealing as techniques from programming language theory can be used to develop proof assistants for this kind of type theories, for instance, type inference can be used for proof construction. Because of this, there are many implementations of systems based on intuitionistic type theory, both as proof assistants like the Coq system and as general purpose dependently typed programming languages like Agda [Norell, 2007] and Idris [Brady, 2013].

In recent years, many large-scale formalizations of mathematics have been completed. The first of these was the formalization of the **four color theorem** which states that any planar map can be colored using at most four colors in such a way that no two adjacent colors are the same. The formal proof, using the Coq system, was finished by Georges Gonthier in 2005 [Gonthier, 2005, 2008]. In 2012 a team, led by Georges Gonthier and using Coq, completed the formal proof of the **Feit-Thompson odd order theorem** stating that all finite groups of odd order are solvable [Gonthier et al., 2013]. In 2014 it was announced that the FlySpeck project, led by Thomas Hales, formalizing the **Kepler Conjecture** had been completed [The FlySpeck Project, 2014]. This conjecture (which is now a theorem) explains the optimal way to pack spheres in three dimensional space. The formal proof was carried out using a combination of both the Isabelle and HOL Light proof assistants.

The original proofs of these theorems were controversial when they were first announced. The first proof of the four color theorem, by Kenneth Appel and Wolfgang Haken, relied on extensive computations of different graph configurations that took over 1000 hours for their computers to complete [Appel, 1984]. Also Thomas Hales' original proof of the Kepler conjecture, submitted in 1998 to the Annals of Mathematics, was proved with the aid of computer programs written in approximately 40,000 lines of code. The paper and code were reviewed by 12 experts during four years, and the final verdict was that they were only "99% certain" of the correctness of the proof [Avigad and Harrison, 2014]. Because of this Thomas Hales started the FlySpeck project with the aim of formalizing his proof and removing this last percent of doubt, which now seems to have been achieved [The FlySpeck Project, 2014].

The proof of the odd order theorem of Walter Feit and John Griggs Thompson on the other hand was controversial because of the sheer length of the proof. With its 255 pages it was one of the longest proofs ever submitted to a mathematical journal back in 1963. This theorem is part of the classification theorem of finite simple groups whose final proof now consists of almost ten thousand pages written by many authors over a long period of time [Steingart, 2012]. Jean-Pierre Serre discusses this in an interview from when he was awarded the Abel prize in 2003:

> *"For years I have been arguing with group theorists who claimed that the "Classification Theorem" was a "theorem", i.e. had been proved. It had indeed been announced as such in 1980 by Gorenstein, but it was found later that there was a gap (the classification of "quasi-thin" groups). Whenever I asked the specialists, they replied something like: "Oh no, it is not a gap; it is just something which has not been written, but there is*

> *an incomplete unpublished 800-page manuscript on it." For me, it was just the same as a "gap", and I could not understand why it was not acknowledged as such."* [Raussen and Skau, 2004]

This kind of reliability issues of very large and complicated mathematical results, that might be understood by only a few experts, is problematic. Another example of such a proof is Andrew Wiles' proof of Fermat's Last Theorem where the first version was believed to be correct in 1993, but an error was found and finally corrected two years later in 1995. Because of the complexity of proofs like this, and the shortage of people with enough expertise to asses them, peer-reviewing becomes very time consuming and errors might go unnoticed for a long time. Formal proofs of these results would increase not only the reliability, but also the efficiency of the peer-reviewing process. The formalizations would, hopefully, also give rise to new mathematical methods and results.

Another motivation behind formalizing mathematics is that it involves carefully representing mathematical concepts and proofs in order to make them suited for implementation on a computer. This way simpler, clearer and more elegant proofs can be obtained. This works the other way around as well: when formalizing a mathematical result the proof assistant, or the underlying logical system, might need to be improved in order to be able to represent mathematics more conveniently, yielding better tools and techniques.

An example of this is the recent developments in type theory, building on connections to abstract homotopy theory, creating a new field called **Homotopy Type Theory** [Pelayo and Warren, 2014; Univalent Foundations Program, 2013]. This connection was discovered, among others, by Fields medalist Vladimir Voevodsky. From it a new axiom, motivated by models of type theory in simplicial sets, called the **univalence axiom** was added to type theory. This new foundations of mathematics is called **Univalent foundations** and during the 2012–2013 academic year the Institute for Advanced Study in Princeton held a special year devoted to it.

3 This thesis

Regardless of the success stories and impressive developments in the interactive theorem proving community during recent years it is hard to imagine programmers and mathematicians starting to prove their programs and theorems correct using a proof assistant on a daily basis. The main reason for this is that formalization is very time consuming and requires both a lot of training and expertise.

A goal of this thesis is to provide a step in the direction to remedy this. First by studying techniques to formally verify efficient programs and data structures, and then by formalizing constructive algebra. This involves finding good abstractions and suitable proofs to implement libraries of formalized mathematics that can be used for further developments. These together provide a basis for bridging the gap between algorithms and theories implemented in computer algebra systems and proof assistants. This way both the reliability in implementations of algorithms in computer algebra systems and the computational capabilities of proof assistants can be increased.

3.1 Method

The formalizations presented in this thesis has been performed using the Coq proof assistant together with the **Small Scale Reflection** (SSReflect) extension [Gonthier et al., 2008]. This extension was initially developed by Georges Gonthier during the formalization of the four color theorem and has since then been further developed in the **Mathematical Components** (MathComp) Project [Mathematical Components Project, 2014] during the formalization of the Feit-Thompson Theorem. The extension provides a new tactic language to Coq that is useful for doing "small scale reflection". The idea of small scale reflection is to use computation to automate small proof steps resulting in a very concise proof style.

The SSReflect/MathComp project also contains a large and well designed library of already formalized mathematical theories, which was developed during the formalization of the Feit-Thompson theorem. It contains many basic mathematical structures and theories, including an algebraic hierarchy, polynomials, matrices and linear algebra. By using this library we avoid reimplementing these fundamental notions and may start building on what has already been done. From here on we will refer to this library simply as the "MathComp library".

The main sources of constructive algebra used during the formalizations are the book by Mines, Richman and Ruitenburg [Mines et al., 1988] and the more recent book by Lombardi and Quitté [Lombardi and Quitté, 2011]. These present modern algebra from a constructive point of view, avoiding Noetheriannity whenever possible. The reason for avoiding Noetheriannity is that it is defined by quantification over all ideals, making it a logically complicated notion. By not assuming it more general results, expressed using first-order logic, can be obtained. Because of this, results are effective and have direct computational meaning, which makes them well-suited for formalization in intuitionistic type theory.

Most of the work presented in this thesis has been carried out as part of the European project ForMath – *Formalization of Mathematics* [The ForMath Project, 2014]. The goal of this project was to develop formally verified libraries of mathematics concerning abstract algebra, linear algebra, real number computation and algebraic topology. These libraries should be designed as software libraries using ideas from software engineering to increase reusability and scalability.

3.2 Formal developments

The formalizations presented in this thesis have resulted in a library of computational algebra called CoqEAL — The Coq Effective Algebra Library. The latest development version can be found at:

`https://github.com/CoqEAL/CoqEAL/`

The developments are divided into three folders. The `v0.1` folder corresponds to the first version of the CoqEAL library, while the `refinements` and `theory` folders correspond to the latest version. Some data, corresponding to the

	Definitions	Lemmas	Lines of code
`v0.1`	646	971	14850
`refinements`	264	243	9048
`theory`	355	699	9136
total	1265	1913	33034

Table 1.1: Data related to the formal developments

development version of the library at the time of writing, are collected in Table 1.1.

The numbers in the last row should be taken with a grain of salt as many results in the `v0.1` folder are reimplemented in the `refinements` and `theory` folders.

Documentation of the formalizations, corresponding to what is presented in the thesis, can be found at:

`http://www.cse.chalmers.se/~mortberg/thesis/`

3.3 Structure and organization of the thesis

This thesis is a collection of six papers, divided evenly into two parts. One part is about program and data refinements, and the other about constructive algebra. The papers have all been slightly modified to fit together in the thesis. However, all papers can still be read separately which means that there is some overlap in terms of notions and notations being defined in different papers. A person who reads the thesis from beginning to end can hence skim over these, while someone who is only interested in the results of one paper can read it without having to jump back and forth.

The rest of this introduction presents overviews of the two parts and the papers contained in them. The main results are summarized and the relationship between the papers detailed.

4 Program and data refinements

This part discusses ways to implement program and data refinements in type theory. These two kinds of refinements can be summarized as:

- Program refinements: Transform a program into a more efficient one computing the same thing using a different algorithm, while preserving the types.
- Data refinements: Change the data representation on which the program operates into a more efficient one, while preserving the involved algorithms.

The first two papers discuss a framework for conveniently expressing these kinds of refinements in Coq. The third paper presents an example of a program refinement of the Sasaki-Murao algorithm [Sasaki and Murao, 1982] for

computing the characteristic polynomial of a matrix over any commutative ring in polynomial time.

This part provides the basis of the CoqEAL library and has successfully been used in a formal proof that $\zeta(3)$ is irrational [Mahboubi et al., 2014]. More precisely, it was used to prove a lemma involving computations with rather large rational numbers by refining a unary representation to a binary one and then performing the computations with the more efficient binary representation.

4.1 A refinement-based approach to computational algebra in Coq

The first paper presents a methodology for implementing efficient algebraic algorithms and proving them correct using both program and data refinements. This is done by implementing a simple and often inefficient version of the algorithm on rich data types and then refining it to a more efficient version on simple types. The two versions of the algorithms are then linked to each other and the correctness of the translation is formally proved in Coq.

The methodology for doing program and data refinements in the paper can be summarized as:

1. Implement a *proof-oriented* version of the algorithm using rich (dependent) data types and use the MathComp library to develop theories about them.
2. Refine this algorithm into an efficient *computation-oriented* one, using the same data types as in 1., and prove that it behaves like the proof-oriented version.
3. Translate the rich data types and the computation-oriented algorithm to computation-oriented data types and prove the correctness of the translation.

By separating the implementation of the algorithm used for proving properties and the one used for computation we achieve what Dijkstra referred to as *"the separation of concerns"*:

> *"We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. [...] But nothing is gained – on the contrary! – by tackling these various aspects simultaneously. It is what I sometimes have called* ***"the separation of concerns"****."* [Dijkstra, 1974]

Using this methodology the paper presents a library of computational structures with four main examples of algorithms from linear and commutative algebra:

- Efficient polynomial multiplication using Karatsuba's algorithm.
- Greatest common divisor (gcd) of multivariate polynomials.

- Rank computation of matrices with coefficients in a field.
- Efficient matrix multiplication based on Winograd's version of Strassen's algorithm.

The second of these, gcd of multivariate polynomials, is especially interesting from the point of view of constructive algebra as the correctness proof neither rely on the field of fractions nor unique factorization as is customary in classical presentations. It is instead based on Gauss' Lemma as in [Knuth, 1981] and the notion of GCD domains [Mines et al., 1988].

This paper was published in the post-proceedings of the 2012 edition of the Interactive Theorem Proving conference. [Dénès et al., 2012]

4.2 Refinements for free!

The data refinement step of the methodology presented in the first paper was implemented by duplicating both the implementation of the efficient program and the hierarchy of structures on which it operates. It also restricted which computation-oriented types could be used to types with a subtype isomorphic to the proof-oriented one. This works fine for polynomials or matrices represented using lists, but not for non-normalized rational numbers. This paper resolves these issues and improves the methodology in the first paper by adopting the following approach:

1. *Relate* a proof-oriented data representation with a more computationally efficient one by an arbitrary heterogeneous relation.
2. *Parametrize* algorithms and the data on which they operate by an abstract type and its basic operations.
3. *Instantiate* these algorithms with proof-oriented data types and basic operations, and prove the correctness of that instance.
4. Use *parametricity* of the algorithm (with respect to the data representation on which it operates), together with points 2. and 3., to deduce that the algorithm instantiated with the more efficient data representation is also correct.

This methodology improves the one presented in the first paper with respect to the following aspects:

1. Generality: it extends to previously unsupported data types, like non-normalized rational numbers.
2. Modularity: each operation is refined in isolation instead of refining whole algebraic structures.
3. Genericity: before, every operation had to be implemented both for the proof-oriented and computation-oriented types, now only one generic implementation is sufficient.

4. Automation: there is now a clearer separation between the different steps of data refinements which makes it possible to use parametricity in order to automate proofs that previously had to be done by hand.

The last step involves proof automation using parametricity, which makes it the most subtle to implement as there is no internal parametricity in Coq. We chose to implement a proof-search algorithm using type classes [Sozeau and Oury, 2008] that finds the proof that a function is parametric when needed. In a system with internal parametricity, like Type Theory in Color [Bernardy and Moulin, 2013], one would instead get this for free. In practice the implementation works well in many cases, but when it fails the user has to provide the proofs by hand. This is still an improvement compared to the first paper though, as this kind of proofs always had to be given by hand before.

We have ported some of the algorithms of the first paper to the new framework and added new data refinements, in particular sparse polynomials and non-normalized rational numbers. The methodology presented in this paper provides the basis of the current version of the CoqEAL library.

This paper was published in the post-proceedings of the 2013 edition of the Certified Programs and Proofs conference. [Cohen et al., 2013]

4.3 A formal proof of the Sasaki-Murao algorithm

The third paper describes the formalization of a simple polynomial time algorithm for computing the determinant of matrices over any commutative ring. The algorithm is based on Bareiss' algorithm [Bareiss, 1968], which can be compactly presented using functional programming notations. The algorithm is simple and runs in polynomial time, but the standard proof of correctness involves complicated identities for determinants called Sylvester identities [Abdeljaoued and Lombardi, 2004]. In order to conveniently formalize the correctness of this algorithm an alternative proof was found and some of the Sylvester identities then follows as corollaries of it.

The correctness of Bareiss' algorithm requires that the principal minors of the matrix are regular, that is, that they are not zero divisors. The Sasaki-Murao algorithm [Sasaki and Murao, 1982] uses an elegant trick to avoid this: apply Bareiss' algorithm to the matrix used when computing the characteristic polynomial and do the computations in the polynomial ring. This way Bareiss' algorithm can be applied to any matrix to compute the characteristic polynomial, and from this the determinant easily can be obtained by setting the indeterminate to zero. A benefit of computing in the polynomial ring is that polynomial pseudo-division [Knuth, 1981] may be used. Hence there is no need to assume that the ring has a division operation and the algorithm can be applied to matrices over any commutative ring.

The effective version of the algorithm has been implemented using the approach presented in the first paper. This implementation required us to combine many of the different parts of the library as the computations are done on matrices of polynomials. The resulting version is a simple and formally verified algorithm for computing the determinant of a matrix using operations like matrix multiplication, polynomial pseudo-division and Horner evaluation of polynomials.

This paper has been published in the Journal of Formalized Reasoning in 2012. [Coquand et al., 2012a]

5 Constructive algebra in type theory

This part discusses the formalization of various structures and results in constructive algebra. Constructive algebra differs from classical algebra by avoiding the law of excluded middle and the axiom of choice. It is also common not to assume Noetheriannity (*i.e.* that all ideals are finitely generated) as it is a complicated notion from a constructive point of view [Perdry, 2004]. In fact, many results can be proved constructively in the more general setting without Noetherian assumptions [Lombardi and Quitté, 2011].

The first paper in this part is concerned with the formalization of coherent and strongly discrete rings. The next paper then develops the theory of finitely presented modules over these rings and prove that it is a good setting for doing homological algebra. Finally, in the sixth and last paper of the thesis, the formalization of elementary divisor rings is described. These are examples of coherent and strongly discrete rings, which means that we get concrete instances for the theory developed in the first two papers of this part.

5.1 Coherent and strongly discrete rings in type theory

This paper presents the formalization of algebraic structures that are important in constructive algebra: coherent and strongly discrete rings. Coherent rings can be characterized as rings where any finitely generated ideal is finitely presented, this means that it is possible to solve homogeneous systems of equations over these rings. Further, a ring is strongly discrete if membership in finitely generated ideals is decidable. If a ring is both coherent and strongly discrete it is not only possible to solve homogeneous systems of equations, but the solution to any (solvable) system can be computed.

These notions are not stressed in classical presentations as one can prove that Noetherian rings are coherent and strongly discrete, however the proof of this relies on classical logic in essential ways. This means that the proof has no computational content, which implies that it is not possible to write a program that extracts a finite presentation from a finitely generated ideal over a Noetherian ring. By dropping the Noetherian assumptions and working directly with coherent and strongly discrete rings we not only get more general results, but also proofs with computational content.

Examples of coherent and strongly discrete rings considered in the paper are Bézout domains (*e.g.* $\mathbb{Z}$ and $k[x]$ where k is a field) and Prüfer domains (*e.g.* $\mathbb{Z}[\sqrt{-5}]$ and $k[x,y]/(y^2+x^4-1)$). These are non-Noetherian analogues to the classical notions of principal ideal domains and Dedekind domains. By proving that these structures are coherent and strongly discrete we obtain formally verified algorithms for solving systems of equations over them.

The methodology of the first paper has been applied in order to develop computational versions of the structures and effective versions of the algorithms. This was complicated as some of the algorithms, especially for Prüfer

domains, are quite involved.

Our main motivation for studying these structures is that coherent and strongly discrete rings are fundamental notions in constructive algebra [Mines et al., 1988] that can be used as a basis for developing computational homological algebra as in the HOMALG system [Barakat and Lange-Hegermann, 2011; Barakat and Robertz, 2008].

This paper was published in the post-proceedings of the 2012 edition of the Certified Programs and Proofs conference. [Coquand et al., 2012b]

5.2 A Coq formalization of finitely presented modules

This paper is about formalizing the module theory of coherent and strongly discrete rings. The concept of a module over a ring is a generalization of the notion of a vector space over a field, where the scalars are elements of an arbitrary ring. We restrict to finitely presented modules as these can be concretely represented using matrices. More precisely, an R-module $\mathcal{M}$ is finitely presented if it can be represented using a finite set of generators and a finite set of relations between these. This means that they can be compactly described by an exact sequence:

$$R^{m_1} \xrightarrow{M} R^{m_0} \xrightarrow{\pi} \mathcal{M} \longrightarrow 0$$

Here π is a surjection and M a matrix representing the m_1 relations among the m_0 generators of the module $\mathcal{M}$. A morphism between finitely presented modules, $\mathcal{M}$ and $\mathcal{N}$, is given by the following commutative diagram:

$$\begin{array}{ccccccc} R^{m_1} & \xrightarrow{M} & R^{m_0} & \longrightarrow & \mathcal{M} & \longrightarrow & 0 \\ \downarrow{\scriptstyle \varphi_R} & & \downarrow{\scriptstyle \varphi_G} & & \downarrow{\scriptstyle \varphi} & & \\ R^{n_1} & \xrightarrow{N} & R^{n_0} & \longrightarrow & \mathcal{N} & \longrightarrow & 0 \end{array}$$

This means that morphisms between finitely presented modules can be represented using matrices as well. All operations can then be defined by manipulating these matrices. By assuming that the underlying ring is coherent and strongly discrete we can represent morphisms using only one matrix and a proof that the other matrix can be computed from this. We also get algorithms for computing the kernel and cokernel of morphisms. Using this we have verified that finitely presented modules over coherent and strongly discrete rings form an abelian category, which means that they are a good setting for developing homological algebra.

It is in general not possible, even if the ring is coherent and strongly discrete, to decide whether two finitely presented modules are isomorphic or not. However when working with $\mathbb{Z}$-modules, or more generally R-modules over rings where there is an algorithm computing the Smith normal form of any matrix, this is possible. The next paper discusses a class of rings with this property, called elementary divisor rings in [Kaplansky, 1949], and provides new concrete instances of coherent and strongly discrete rings. This can hence be combined with the theory developed in this paper for formalizing computational homological algebra.

This paper was published in the post-proceedings of the 2014 edition of the Interactive Theorem Proving conference. [Cohen and Mörtberg, 2014]

5.3 Formalized linear algebra over elementary divisor rings in Coq

An integral domain is called an elementary divisor ring if any matrix is equivalent to a matrix in Smith normal form:

$$\begin{bmatrix} d_1 & & 0 & \cdots & \cdots & 0 \\ & \ddots & & & & \vdots \\ 0 & & d_k & 0 & \cdots & 0 \\ \vdots & & 0 & 0 & & \vdots \\ \vdots & & \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & \cdots & 0 \end{bmatrix}$$

where $d_i \mid d_{i+1}$ for all i. This means that given M we should be able to compute invertible matrices P and Q such that $PMQ = D$ where D is in Smith normal form.

By developing the theory of linear algebra over these rings we can prove that they are coherent. It is also easy to show that any elementary divisor ring is a Bézout domain and hence strongly discrete using the theory developed in the fourth paper. We hence get that elementary divisor rings form interesting instances for our work on finitely presented modules.

Finitely presented modules over elementary divisor rings are especially well-behaved because any module is isomorphic to a direct sum of a free module and cyclic modules in a unique way. The proof of this can be seen as a constructive version of the classification theorem for finitely generated modules over principal ideal domains. We have formalized this theorem and from it we get an algorithm for deciding if two finitely presented modules over elementary divisor rings are isomorphic or not.

It is a well-known classical result that principal ideal domains (*i.e.* integral domains where every ideal is principal) are elementary divisor rings. But, as pointed out earlier, we want to avoid Noetherian hypotheses in constructive algebra. However, the problem whether any Bézout domain is an elementary divisor ring is still open [Lorenzini, 2012]. So instead we start by giving a concrete implementation of an algorithm for computing the Smith normal form of matrices with coefficients in a Euclidean domain. It is straightforward to generalize this to constructive principal ideal domains, defined as Bézout domains where strict divisibility is well-founded.

We have also formalized a reduction, due to Kaplansky [Kaplansky, 1949], that reduces the problem of proving that a Bézout domain is an elementary divisor ring to proving that it satisfies a first-order condition called the "Kaplansky condition". Using this we prove that Bézout domains extended with one of the following assumptions are elementary divisor rings:

1. Adequacy (*i.e.* the existence of a gdco operation) [Helmer, 1943].

2. Constructive Krull dimension ≤ 1 [Lombardi and Quitté, 2011].

3. Well-founded strict divisibility (constructive principal ideal domains).

The first one of these is the most general while the other two imply it. This hence gives an alternative proof that constructive principal ideal domains are elementary divisor rings. This proof also shows that while it may not be possible to just drop the Noetherian assumption altogether, it can sometimes be replaced by some first-order condition, in this case either constructive Krull dimension ≤ 1, adequacy or the Kaplansky condition.

This paper is currently under submission. [Cano et al., 2014]

Part I

Program and Data Refinements

1

A Refinement-Based Approach to Computational Algebra in Coq

Maxime Dénès, Anders Mörtberg and Vincent Siles

Abstract. We describe a step-by-step approach to the implementation and formal verification of efficient algebraic algorithms. Formal specifications are expressed on rich data types which are suitable for deriving essential theoretical properties. These specifications are then refined to concrete implementations on more efficient data structures and linked to their proof-oriented counterparts. We illustrate this methodology on key applications: matrix rank computation, Strassen's fast matrix product, Karatsuba's polynomial multiplication, and the gcd of multivariate polynomials.

Keywords. Formalization of mathematics, Computer algebra, Efficient algebraic algorithms, Coq, SSReflect.

1.1 Introduction

In the past decade, the range of application of proof assistants has extended its traditional ground in theoretical computer science to mainstream mathematics. Formalized proofs of important theorems like the fundamental theorem of algebra [Barendregt et al., 2014], the four color theorem [Gonthier, 2005, 2008] and the Jordan curve theorem [Hales, 2007] have advertised the use of proof assistants in mathematical activity, even in cases when the pen and paper approach was no longer tractable.

But since these results established proofs of concept, more effort has been put into designing an actually scalable library of formalized mathematics.

The *Mathematical Components* project (developing the MathComp library) uses the small scale reflection extension (SSReflect) for the Coq proof assistant to achieve a nearly comparable level of detail to usual mathematics on paper, even for advanced theories like the proof of the Feit-Thompson theorem [Gonthier et al., 2013]. In this approach, the user expresses significant deductive steps while low-level details are taken care of by small computational steps, at least when properties are decidable. Such an approach makes the proof style closer to usual mathematics.

One of the main features of these libraries is that they heavily rely on rich dependent types, which gives the opportunity to encode a lot of information directly into the type of objects: for instance, the type of matrices embeds their size, which makes operations like multiplication easy to implement. Also, algorithms on these objects are simple enough so that their correctness can easily be derived from the definition. However in practice, most efficient algorithms in modern computer algebra systems do not rely on dependent types and do not provide any proof of correctness. We show in this paper how to use this rich mathematical framework to develop efficient computer algebra programs *with proofs of correctness*. This is a step towards closing the gap between proof assistants and computer algebra systems.

The methodology we suggest for achieving this is the following: we are able to prove the correctness of some mathematical algorithms having all the high-level theory at our disposal and we then refine them to an implementation on simpler data structures that will be actually running on machines. In short, we aim at formally linking convenient high-level properties to efficient low-level implementations, ensuring safety of the whole approach while enjoying better performance thanks to the separation of proofs and computational content.

In the next section, we describe the methodology of refinements. Then, we give two examples of such refinements for matrices in section 1.3, and polynomials in section 1.4. In section 1.5, we give a solution to unify both examples by describing CoqEAL — The Coq Effective Algebra Library. This is a library built using the methodology presented in this paper on top of the MathComp libraries.

1.2 Refinements

Refinements are commonly used to describe successive steps when verifying a program. Typically, a specification is expressed in Hoare logic [Hoare, 1969], then the program is described in a high-level language and finally implemented in C. Each step is proved correct with respect to the previous one. By using several formalisms, one has to trust every translation step or prove them correct in yet another formalism.

Our approach is similar: we refine the definition of a concept to an efficient algorithm described on high-level data structures. Then, we implement it on data structures that are closer to machine representations, once we no longer need rich theory to prove the correctness.

However, in our approach, all of the layers can be expressed in the same

formalism (the Calculus of Inductive Constructions), though they do not use exactly the same features. On one hand, the high-level layers use rich dependent types that are very useful when describing theories because they allow abuse of notations and concise statements which quickly become necessary when working with advanced mathematics. On the other hand, the efficient implementations use simple types, which are closer to standard implementations in traditional programming languages. The main advantage of this approach is that the correctness of translations can easily be expressed in the formalism itself, and we do not rely on any additional external proofs.

The implementation is an immediate translation of the algorithm in Figure 1.1:

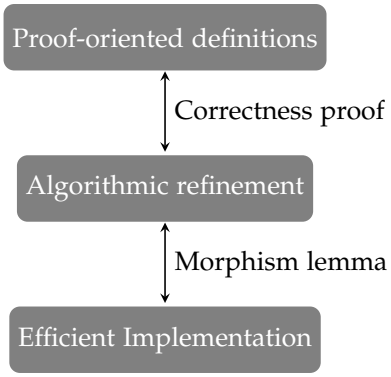


Figure 1.1: The three steps of refinement

In the next sections, we are going to use the following methodology to build efficient algorithms from high-level descriptions:

1. Implement a proof-oriented version of the algorithm using MathComp structures and use the libraries to prove properties about them.
2. Refine this algorithm into an efficient one still using MathComp structures and prove that it behaves like the proof-oriented version.
3. Translate the MathComp structures and the efficient algorithm to the low-level data types, ensuring that they will perform the same operations as their high-level counterparts.

The first two points uses the full power of dependent types to develop the theory and proving the correctness of the refinements while the third only uses simple types suitable for computation.

1.3 Matrices

Linear algebra is a natural first test-case to validate our approach, as a pervasive and inherently computational area of mathematics, which is well covered by the MathComp library [Gonthier, 2011]. In this section, we will detail the

(quite simple) data structure we use to represent matrices and then review two fundamental examples: rank computation and efficient matrix product.

1.3.1 Representation

Matrices are represented by finite functions over pairs of ordinals (the indices):

```
(* 'I_n *)
Inductive ordinal (n : nat) := Ordinal m of m < n.

(* 'M[R]_(m,n) = matrix R m n *)
(* 'rV[R]_m = 'M[R]_(1,m) *)
(* 'cV[R]_m = 'M[R]_(m,1) *)
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.
```

This encoding makes many properties easy to derive, but it is inefficient for evaluation. Indeed, a finite function over `'I_m * 'I_n` is internally represented as a flat list of $m \times n$ values which has to be traversed whenever the function is evaluated. Moreover, having the size of matrices encoded in their type allows to state concise lemmas without explicit side conditions, but it is not always flexible enough when getting closer to machine-level implementation details.

To be able to implement efficient matrix operations we introduce a low-level data type `seqmatrix` representing matrices as lists of lists. Such a matrix is built from a dependently typed one by mapping canonical enumerations (given by the `enum` function) of ordinals to the corresponding coefficients in the dependently typed matrix:

```
Definition seqmx_of_mx (M : 'M[R]_(m,n)) : seqmatrix :=
  [seq [seq M i j | j <- enum 'I_n] | i <- enum 'I_m].
```

To ensure the correct behavior of list based matrices it is sufficient to prove that `seqmx_of_mx` is injective:

```
Lemma seqmx_eqP (M N : 'M[R]_(m,n)) :
  reflect (M = N) (seqmx_of_mx M == seqmx_of_mx N).
```

The == symbol denotes boolean equality defined using the underlying equality on `R`. Operations like addition are straightforward to implement, and their correctness is expressed through a morphism lemma, stating that the list based representation of the sum of two dependently typed matrices is the sum of their representations as lists:

```
Definition addseqmx (M N : seqmatrix) : seqmatrix :=
  zipwith (zipwith (fun x y => add x y)) M N.

Lemma addseqmxE :
  {morph seqmx_of_mx : M N / M + N >-> addseqmx M N}.
```

Here `morph` is notation meaning that `seqmx_of_mx` is an additive morphism from dependently types to list based matrices. It is worth noting that we could have stated all our morphism lemmas with the converse operator (from list based matrices to dependently typed ones). But these lemmas would then have been quantified over lists of lists, with poorer types, which would have

required a well-formedness predicate as well as premises expressing size constraints. The way we have chosen takes full advantage of the information carried by the richer types.

Like the `addseqmx` operation, we have developed list based implementations of most of the matrix operations in the MathComp library and proved the corresponding morphism lemmas. Among these operations we can cite: subtraction, scaling, transpose and block operations.

1.3.2 Computing the rank

Now that the basic data structure and operations have been defined, it is possible to apply our approach to an algorithm based on Gaussian elimination which computes the rank of a matrix $A = (a_{i,j})$ over a field K. We first specify the algorithm using dependently typed matrices and then refine it to the low-level structures.

An elimination step consists of finding a nonzero pivot in the first column of A. If there is none, it is possible to drop the first column without changing the rank. Otherwise, there is an index i such that $a_{i,1} \neq 0$. By linear combinations of rows (preserving the rank) A can be transformed into the following matrix B:

$$\begin{bmatrix} 0 & a_{1,2} - \frac{a_{1,1}\times a_{i,2}}{a_{i,1}} & \cdots & a_{1,n} - \frac{a_{1,1}\times a_{i,n}}{a_{i,1}} \\ 0 & \vdots & & \vdots \\ a_{i,1} & a_{i,2} & \cdots & a_{i,n} \\ 0 & \vdots & & \vdots \\ 0 & a_{n,2} - \frac{a_{n,1}\times a_{i,2}}{a_{i,1}} & \cdots & a_{n,n} - \frac{a_{n,1}\times a_{i,n}}{a_{i,1}} \end{bmatrix} = \begin{bmatrix} 0 & & \\ \vdots & R_1 & \\ 0 & & \\ a_{i,1} & \cdots & a_{i,n} \\ 0 & & \\ \vdots & R_2 & \\ 0 & & \end{bmatrix}$$

Let $R = \begin{bmatrix} R_1 \\ R_2 \end{bmatrix}$, since $a_{i,1} \neq 0$, this means that $\mathrm{rank}(A) = \mathrm{rank}(B) = 1 + \mathrm{rank}(R)$. Hence the current rank can be incremented and the algorithm can be recursively applied on R.

In our development we defined a function `elim_step` returning the matrix R above and a boolean b indicating if a pivot has been found. A wrapper function `rank_elim` is in charge of maintaining the current rank and performing the recursive call on R:

```
Fixpoint rank_elim (m n : nat) : 'M[K]_(m,n) -> nat :=
  match n return 'M[K]_(m,n) -> nat with
   | q.+1 => fun M =>
     let (R,b) := elim_step M in (rank_elim R + b)%N
   | _ => fun _ => 0%N
  end.
```

Note that booleans are coerced to natural numbers: `b` is interpreted as `1` if true and `0` if false. The correctness of `rank_elim` is expressed by relating it to the `\rank` function of the MathComp library:

```
Lemma rank_elimP n m (M : 'M[K]_(m,n)) : rank_elim M = \rank M.
```

The proof of this specification relies on a key invariant of `elim_step`, relating the ranks of the input and output matrices:

```
Lemma elim_step_rank m n (M : 'M[K]_(m, 1 + n)) :
  let (R,b) := elim_step M in \rank M = (\rank R + b)%N.
```

Now the proof of `rank_elimP` follows by induction on `n`. The computation oriented version of this algorithm is a direct translation of the algorithm using only list based matrices and operations on them. This simply typed version (called `rank_elim_seqmx`) is then linked to `rank_elim` by the lemma:

```
Lemma rank_elim_seqmxE : forall m n (M : 'M[K]_(m, n)),
  rank_elim_seqmx m n (seqmx_of_mx M) = rank_elim M.
```

The proof of this is straightforward as all of the operations on list based matrices have morphism lemmas just like this one. This means that the proof can be done simply by expanding the definitions and applying the translation morphisms.

1.3.3 Strassen's fast matrix product

In the context we presented, the naive matrix product (*i.e.* with cubic complexity) of two matrices M and N can be compactly implemented by transposing the list of lists representing N and then for each i and j compute $\sum_k M_{i,k} N^{\mathrm{T}}_{j,k}$:

```
Definition mulseqmx (M N : seqmatrix) : seqmatrix :=
 let N' := trseqmx N in
 map (fun r => map (foldl2 (fun z x y => x * y + z) 0 r) N') M.

Lemma mulseqmxE (M : 'M[R]_(m,p)) (N : 'M[R]_(p,n)) :
  mulseqmx (seqmx_of_mx M) (seqmx_of_mx N) =
  seqmx_of_mx (M *m N).
```

Here `*m` is the notation for the matrix product from the MathComp libraries. Once again, the rich type information in the quantification of the morphism lemma ensures that it can be applied only if the two matrices have compatible sizes.

In 1969, Strassen [Strassen, 1969] showed that 2×2 matrices can be multiplied using only 7 multiplications without requiring commutativity. This yields an immediate recursive scheme for the product of two $n \times n$ matrices with $\mathcal{O}(n^{\log_2 7})$ complexity.[1] This is an important theoretical result, since matrix multiplication was commonly thought to be intrinsically of cubic complexity, it opened the way to many further improvements and gave birth to a fertile branch of algebraic complexity theory.

However, Strassen's result is also still of practical interest since the asymptotically best algorithms known today [Coppersmith and Winograd, 1990] are slower in practice because of huge hidden constants. Thus, we implemented a variant of this algorithm suggested by Winograd in 1971 [Winograd, 1971], decreasing the required number of additions and subtractions to 15 (instead of 18 in Strassen's original proposal). This choice reflects the implementation

[1] $\log_2 7$ is approximately 2.807.

of matrix product in most of modern computer algebra systems. A previous formal description of this algorithm has been developed in ACL2 [Palomo-Lozano et al., 2001], but it is restricted to matrices whose sizes are powers of 2. The extension to arbitrary matrices represents a significant part of our development, which is to the best of our knowledge the first complete formally verified description of Winograd's version of Strassen's algorithm.

We define a function expressing a recursion step in Winograd's algorithm. Given two matrices A and B and an operator f representing matrix product, it reformulates the algebraic identities involved in the description of the algorithm:

```
Definition Strassen_step (p : positive)
                         (A B : 'M[R]_(p + p)) f :=
  let A11 := ulsubmx A in let A12 := ursubmx A in
  let A21 := dlsubmx A in let A22 := drsubmx A in
  let B11 := ulsubmx B in let B12 := ursubmx B in
  let B21 := dlsubmx B in let B22 := drsubmx B in
  let X := A11 - A21 in let Y := B22 - B12 in
  let C21 := f X Y in
  let X := A21 + A22 in let Y := B12 - B11 in
  let C22 := f X Y in
  let X := X - A11 in let Y := B22 - Y in
  let C12 := f X Y in
  let X := A12 - X in
  let C11 := f X B22 in
  let X := f A11 B11 in
  let C12 := X + C12 in let C21 := C12 + C21 in
  let C12 := C12 + C22 in let C22 := C21 + C22 in
  let C12 := C12 + C11 in
  let Y := Y - B21 in
  let C11 := f A22 Y in let C21 := C21 - C11 in
  let C11 := f A12 B21 in let C11 := X + C11 in
  block_mx C11 C12 C21 C22.
```

This is an implementation of matrix multiplication that is clearly not suited for proving algebraic properties, like associativity. The correctness of this function is expressed by the fact that if `f` is instantiated by the multiplication of matrices, `Strassen_step A B` should be the product of `A` and `B` (where `=2` denotes extensional equality):

```
Lemma Strassen_stepP (p : positive) (A B : 'M[R]_(p + p)) f :
  f =2 mulmx -> Strassen_step A B f = A *m B.
```

This proof is made easy by the use of the `ring` tactic (the formal proof is two lines long). Since version 8.4 of Coq `ring` is applicable to non-commutative rings which has allowed its use in our context.

Note that the above implementation only works for even-sized matrices. This means that the general procedure has to implement a strategy for handling odd-sized matrices. Several standard techniques have been proposed, which fall into two categories. Some are static, in the sense that they preprocess the matrices to obtain sizes that are powers of 2. Others are dynamic,

meaning that parity is tested at each recursive step. Two standard treatments can be implemented either statically or dynamically: padding and peeling. The first consists of adding rows and/or columns of zeros as required to get even dimensions (or a power of 2), these lines are then simply removed from the result. Peeling on the other hand removes rows or columns when needed, and corrects the result accordingly.

We chose to implement dynamic peeling because it seemed to be the most challenging technique from the formalization point of view, since the size of matrices involved depend on dynamic information and the post processing of the result is more sophisticated than using padding. Another motivation is that dynamic peeling has shown to give good results in practice.

The function that implements multiplication by dynamic peeling is called `Strassen` and it is proved correct with respect to the usual matrix product:

```
Lemma StrassenP : forall (n : positive) (M N : 'M[R]_n),
  Strassen M N = M *m N.
```

The list based version is called `Strassen_seqmx` and it is also just a direct translation of `Strassen` using only operations defined on list based matrices. In the next section, Figure 1.2 shows some benchmarks of how well this implementation performs compared to the naive matrix product, but we will first discuss how to implement concrete algorithms based on dependently typed polynomials.

1.4 Polynomials

Polynomials in the MathComp library are represented as records with a list representing the coefficients and a proof that the last of these is nonzero. The library also contains basic operations on this representation like addition and multiplication defined using big operators [Bertot et al., 2008], which means that all operations are *locked* [Gonthier et al., 2008]. This is done in order to prevent definitions from being expanded during type checking which means that computation on them are blocked.

To remedy this we have implemented polynomials as lists without any proofs together with executable implementations of the basic operations. It is very easy to build a list based polynomial from an dependently typed one, simply apply the record projection (called `polyseq`) to extract the list from the record. The soundness of list based polynomials is proved by showing that the pointwise boolean equality on the projected lists reflects the equality on MathComp polynomials:

```
Lemma polyseqP p q : reflect (p = q) (polyseq p == polyseq q).
```

Basic operations like addition and multiplication are slightly more complicated to implement for list based polynomials than for list based matrices as it is necessary to ensure that these operations preserve the invariant that the last element is nonzero. For instance multiplication is implemented as:

```
Fixpoint mul_seq p q := match p,q with
  | [::], _ => [::]
  | _, [::] => [::]
```

```
  | x :: xs,_ => add_seq (scale_seq x q)
                         (mul_seq xs (0%R :: q))
  end.

Lemma mul_seqE : {morph polyseq : p q / p * q >-> mul_seq p q}.
```

Here `add_seq` is addition of list based polynomials and `scale_seq x q` means that every coefficient of `q` is multiplied by `x` (both of these are implemented in such a way that the invariant that the last element is nonzero is satisfied). Using this approach we have implemented a substantial part of the MathComp polynomial library, including pseudo-division.

1.4.1 Karatsuba's fast polynomial multiplication

The naive polynomial multiplication algorithm presented in the previous section requires $\mathcal{O}(n^2)$ operations. A more efficient algorithm is Karatsuba's algorithm [Abdeljaoued and Lombardi, 2004; Karatsuba and Ofman, 1962] which is a divide and conquer algorithm based on reducing the number of recursive calls in the multiplication. More precisely, in order to multiply two polynomials written as $aX^k + b$ and $cX^k + d$ the ordinary method

$$(aX^k + b)(cX^k + d) = acX^{2k} + (ad + bc)X^k + cd$$

requires four multiplications (as the multiplications by X^n can be efficiently implemented by padding the list of coefficients by n zeros). The key observation is that this can be rewritten as

$$(aX^k + b)(cX^k + d) = acX^{2k} + ((a + b)(c + d) - ac - bd)X^k + bd$$

which only requires three multiplication: ac, $(a + b)(c + d)$ and bd.

If the two polynomials have 2^n coefficients and the splitting is performed in the middle at every point then the algorithm will only require $\mathcal{O}(n^{\log_2 3})$ which is better than the naive algorithm.[2] If the polynomials do not have 2^n coefficients it is possible to split the polynomials at for example $\lfloor n/2 \rfloor$ as the formula above holds for any $k \in \mathbb{N}$ and still obtain a faster algorithm.

This algorithm has previously been implemented in Coq for binary natural numbers [O'Connor, 2014] and for numbers represented by a tree-like structure [Grégoire and Théry, 2006]. But as far as we know, it has never been implemented for polynomials before. When implementing this algorithm we first implemented it using dependently typed polynomials as:

```
Fixpoint karatsuba_rec (n : nat) p q := match n with
  | 0%N => p * q
  | n'.+1 =>
     let sp := size p in let sq := size in
     if (sp <= 2) || (sq <= 2) then p * q else
       let m := minn sp./2 sq./2 in
       let (p1,p2) := splitp m p in
       let (q1,q2) := splitp m q in
```

[2] $\log_2 3$ is approximately 1.585.

```
      let p1q1 := karatsuba_rec n' p1 q1 in
      let p2q2 := karatsuba_rec n' p2 q2 in
      let p12 := p1 + p2 in
      let q12 := q1 + q2 in
      let p12q12 := karatsuba_rec n' p12 q12 in
      p1q1 * 'X^(2 * m) +
      (p12q12 - p1q1 - p2q2) * 'X^m + p2q2
end.

Definition karatsuba p q :=
  karatsuba_rec (maxn (size p) (size q)) p q.
```

Here `splitp` is a function that splits the polynomial at the correct point using `take` and `drop`. The correctness of this algorithm is expressed by:

```
Lemma karatsubaE : forall p q, karatsuba p q = p * q.
```

As `p` and `q` are MathComp polynomials this lemma can be proved using all of the theory in the library. The next step is to implement the list version (called `karatsuba_seq`) of this algorithm which is done by changing all the operations in the above definition to the corresponding operations on list based polynomials. The correctness of `karatsuba_seq` is then expressed by:

```
Lemma karatsuba_seqE :
  {morph polyseq : p q / karatsuba p q >-> karatsuba_seq p q}.
```

The proof of this is straightforward as all of the operations have morphism lemmas for translating back and forth between the different representations.

In Figure 1.2 the running times of the different multiplication algorithms that we have implemented are compared:

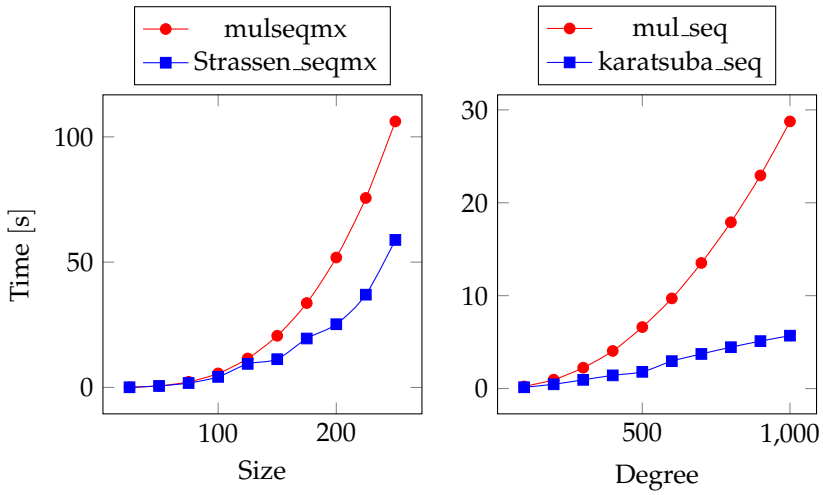


Figure 1.2: Benchmarks of Strassen and Karatsuba multiplications

The benchmarks have been performed by computing the square of integer matrices and polynomials using the Coq virtual machine [Grégoire and Leroy,

2002]. It is clear that both the implementation of Strassen matrix multiplication and Karatsuba polynomial multiplication is faster than their naive counterparts, as expected.

1.4.2 Computing the gcd of multivariate polynomials

An important feature of modern computer algebra systems is to compute the greatest common divisor (gcd) of multivariate polynomials. The main idea of our implementation is based on the observation that in order to compute the gcd of elements in $R[X_1, \ldots, X_n]$ it suffices to show how to compute the gcd in $R[X]$ given that it is possible to compute the gcd of elements in R. So, for example, to compute the gcd of elements in $\mathbb{Z}[X, Y]$ we model it as $(\mathbb{Z}[X])[Y]$, *i.e.* as univariate polynomials in Y with coefficients in $\mathbb{Z}[X]$, and then use that there is a gcd algorithm in $\mathbb{Z}$.

The algorithm that we implemented is based on the presentation in [Knuth, 1981] which uses that in order to compute the gcd of two multivariate polynomials it is possible to instead consider the task of computing the gcd of *primitive* polynomials, *i.e.* polynomials where all coefficients are coprime. Using that any polynomial can be split in a primitive part and a non-primitive part by dividing by the gcd of its coefficients (this is called the *content* of the polynomial and is denoted by $\mathrm{cont}(p)$) we get an algorithm for computing the gcd of any two polynomials. Below is our implementation of this algorithm:

```
Fixpoint gcdp_rec (n : nat) (p q : {poly R}) :=
  let r := modp p q in
  if r == 0 then q
            else if n is m.+1 then gcdp_rec m q (pp r)
                              else pp r.

Definition gcdp p q :=
  let (p1,q1) := if size p < size q then (q,p) else (p,q) in
  let d := gcdr (cont p1) (cont q1) in
  d%:P * gcdp_rec (size (pp p1)) (pp p1) (pp q1).
```

The new operations can be explained as:

- `modp p q`: the remainder after pseudo-dividing `p` by `q`.
- `pp p`: the primitive part of `p`, computed by dividing `p` by its content.
- `gcdr (cont p1) (cont q1)`: the gcd (using the operation in the underlying ring) of the content of `p1` and the content of `q1`.

The correctness of this algorithm is now expressed by:

```
Lemma gcdpP g p q : g %| gcdp p q = (g %| p) && (g %| q).
```

Here `p %| q` computes whether `p` divides `q` or not. As divisibility is reflexive this equality is a compact way of expressing that the function actually computes the gcd of `p` and `q`.

Our result can be stated in constructive algebra as: If R is a GCD domain then so is $R[X]$. Our algorithmic proof is different (and arguably simpler) than

the one found in [Mines et al., 1988]; for instance, we do not go via the field of fractions of the ring. Instead it is proved using Gauss' lemma and its corollary for the primitive part:

```
Lemma cont_mul : forall p q, cont (p * q) %= cont p * cont q.

Lemma pp_mul : forall p q, pp (p * q) %= pp p * pp q.
```

This lemma states that the content of the product of a polynomial is equal to the product of the contents of the polynomials up to multiplication by a unit. The reason that this result is important is that one can use it to derive properties of the gcd of polynomials [Knuth, 1981]. Let $\gcd_{R[x]}(p,q)$ be the gcd of p and q in $R[x]$, then Gauss' lemma implies that:

$$\begin{aligned}\operatorname{cont}(\gcd_{R[x]}(p,q)) &= \gcd_R(\operatorname{cont}(p),\operatorname{cont}(q))\\ \operatorname{pp}(\gcd_{R[x]}(p,q)) &= \gcd_{R[x]}(\operatorname{pp}(p),\operatorname{pp}(q))\end{aligned}$$

Hence

$$\gcd_{R[x]}(p,q) = \gcd_R(\operatorname{cont}(p),\operatorname{cont}(q))\gcd_{R[x]}(\operatorname{pp}(p),\operatorname{pp}(q))$$

and the computation of the gcd of polynomials can be reduced to computing the gcd of *primitive* polynomials which is exactly what is done in `gcdp_rec`.

As noted in [Knuth, 1981], this algorithm may be inefficient when applied to polynomials over integers. A possible solution is to use subresultants. This is a further refinement of the algorithm, which would be interesting to explore since subresultants already been have implemented in Coq [Mahboubi, 2006].

The list based version (`gcdp_seq`) of the algorithm has also been implemented and is linked to the version above by:

```
Lemma gcdp_seqE :
  {morph polyseq : p q / gcdp p q >-> gcdp_seq p q}.
```

However, when running the list based implementation there is a quite subtle problem: the `polyseq` projection links the MathComp polynomials with the polynomials of type `seq R` where `R` is a MathComp ring. Let us consider what happens if we want to compute with $R[x,y]$. It will be represented by `seq (seq R)`, but when we apply the `polyseq` projection we get `seq R` which is not a ring. So our programs are not applicable!

The next section explains how to resolve this issue so that it is possible to implement programs of the above kind that rely on the computability of the underlying ring.

1.5 Hierarchy of computable structures

As noted in the previous section there is a problem when implementing multivariate polynomials by iterating the polynomial construction, *i.e.* by representing $R[X,Y]$ as $(R[X])[Y]$. The same problem occurs when considering other structures where computations relies on the computability of the underlying ring. For instance when computing the characteristic polynomial of a square matrix. For this, one needs to compute with matrices of polynomials

which will require a computation oriented implementation of matrices with coefficients being a computation oriented implementation of polynomials.

However, both the list based matrices and polynomials have something in common: we can guarantee the correctness of the operations on a subset of the low-level structure. This can be used to implement another hierarchy of computable structures corresponding to the MathComp algebraic hierarchy.

1.5.1 Design of the library

We have implemented computation-oriented counterparts to the basic proof-oriented structures in the MathComp hierarchy, *e.g.* $\mathbb{Z}$-modules, rings and fields. These are implemented in the same manner as presented in [Garillot et al., 2009] using canonical structures. Here are a few examples of the mixins we use:

```
Record trans_struct (A B : Type) := Trans {
  trans : A -> B;
  _ : injective trans
}.

(* Mixin for "computable" Z-modules *)
Record mixin_of (V : zmodType) (T : Type) := Mixin {
  zero : T;
  opp : T -> T;
  add : T -> T -> T;
  tstruct : trans_struct V T;
  _ : (trans tstruct) 0 = zero;
  _ : {morph (trans tstruct) : x / - x >-> opp x};
  _ : {morph (trans tstruct) : x y / x + y >-> add x y}
}.

(* Mixin for "computable" rings *)
Record mixin_of (R : ringType) (V : czmodType R) := Mixin {
  one : V;
  mul : V -> V -> V;
  _ : (trans V) 1 = one;
  _ : {morph (trans V) : x y / x * y >-> mul x y}
}.
```

The type `czmodType` is the computable $\mathbb{Z}$-module type parametrized by a $\mathbb{Z}$-module. The `trans` function is the translation function from MathComp structures to the computation-oriented structures. The only thing required of `trans` is that it is injective so that different proof-oriented objects are mapped to different computation-oriented objects. In particular we get that the proof-oriented types are isomorphic to a subset of the computation-oriented types.

This way we can implement all the basic operations of the algebraic structures the way we want (for example using fast matrix multiplication as an implementation of matrix multiplication instead of a naive one). The only thing to prove is that the implementations behave the same as MathComp's operations *on the subset of "well-formed terms"* (*e.g.* for polynomials, lists that

do not end with 0). This is done as above by providing the corresponding morphism lemmas.

The operations presented in the previous sections of the paper can be implemented by parametrizing by computation-oriented structures instead of proof-oriented ones. This way polynomials represented as lists form a computable ring and we hence get ring operations that can be applied on multivariate polynomials built by iterating the polynomial construction.

It is interesting to note that the equational behavior of an abstract structure is carried as a parameter, but does not appear in its computable counterpart, which depends only on the operations to be implemented. For instance, the same computable ring structure can implement a commutative ring or an arbitrary one, only its parameter varies.

1.5.2 Example: computable ring of polynomials

Let us explain how the list based polynomials can be made a computable ring. First, we define:

```
Variable R : comRingType.
Variable CR : cringType R.
```

This says that `CR` is a computable ring parametrized by a commutative ring which makes sense as any commutative ring is a ring. Next we need to implement the translation function from `poly R` to `seq CR` and prove that this translation is injective:

```
Definition trans_poly (p : {poly R}) : seq CR :=
  map (@trans R CR) (polyseq p).

Lemma inj_trans_poly : injective trans_poly.
```

Assuming that computable polynomials already are an instance of the computable $\mathbb{Z}$-module structure it is possible to prove that they are computable rings by implementing multiplication (exactly like above) and then prove the corresponding morphism lemmas:

```
Lemma trans_poly1 : trans_poly 1 = [:: one CR].

Lemma mul_seqE :
  {morph trans_poly : p q / p * q >-> mul_seq p q}.
```

At this point, we could also have used the `karatsuba_seq` implementation of polynomial multiplication instead of `mul_seq` since we can prove its correctness using the `karatsubaE` and `karatsuba_seqE` lemmas. Finally this can be used to build the `CRing` mixin and make it a canonical structure.

```
Definition seq_cringMixin := CRingMixin trans_poly1 mul_seqE.

Canonical Structure seq_cringType :=
  Eval hnf in CRingType {poly R} seq_cringMixin.
```

1.5.3 Examples of computations

This computable ring structure has also been instantiated by the Coq implementation of $\mathbb{Z}$ and $\mathbb{Q}$ which means that they can be used as basis when building multivariate polynomials. To multiply $2 + xy$ and $1 + x + xy + x^2y^2$ in $\mathbb{Z}[x, y]$ we write:

```
Definition p := [:: [:: 2]; [:: 0; 1]].
Definition q := [:: [:: 1; 1]; [:: 0; 1]; [:: 0; 0; 1]].

> Eval compute in mul p q.
  = [:: [:: 2; 2]; [:: 0; 3; 1]; [:: 0; 0; 3]; [:: 0; 0; 0; 1]]
```

The result should be interpreted as $(2 + 2x) + (3x + x^2)y + 3x^2y^2 + x^3y^3$. The gcd of $1 + x + (x + x^2)y$ and $1 + (1 + x)y + xy^2$ in $\mathbb{Z}[x, y]$ can be computed by:

```
Definition p := [:: [:: 1; 1] ; [:: 0; 1; 1] ].
Definition q := [:: [:: 1]; [:: 1; 1]; [:: 0; 1]].

> Eval compute in gcdp_seq p q.
  = [:: [:: 1]; [:: 0; 1]]
```

The result is $1 + xy$ as expected. The following is an example over $\mathbb{Q}[x, y]$:

```
Definition p := [:: [:: 2 # 3; 2 # 3]; [:: 0; 1 # 2; 1 # 2]].
Definition q := [:: [:: 2 # 3]; [:: 2 # 3; 1 # 2];
                    [:: 0; 1 # 2]].

> Eval compute in gcdp_seq p q.
  = [:: [:: 1 # 3]; [:: 0; 1 # 4]]
```

The two polynomials are

$$\frac{2}{3} + \frac{2}{3}x + \frac{1}{2}xy + \frac{1}{2}x^2y$$

and

$$\frac{2}{3} + \frac{2}{3}y + \frac{1}{2}xy + \frac{1}{2}x^2$$

The resulting gcd should be interpreted as

$$\frac{1}{3} + \frac{1}{4}xy$$

1.6 Conclusions and future work

In this paper, we showed how to use high-level libraries to prove properties of algorithms, while retaining good computational properties by providing efficient low-level implementations. The need of modularity of the executable structure appears naturally and the methodology explained in [Garillot et al., 2009] works quite well. The only thing a user has to provide is a proof of an injectivity lemma stating that the translation behaves correctly.

The methodology we suggest has already been used in other contexts, like the CoRN library, where properties of real numbers described in [O'Connor, 2008] are obtained by proving that these real numbers are isomorphic to an abstract, pre-existing but less efficient version. We have here showed that this approach can be applied in a systematic and modular way.

The library we designed also helps to solve a restriction of SSReflect: due to a lot of computations during deduction steps, many of the operations are *locked* to allow type-checking to be performed in a reasonable amount of time. This locking prevents full-scale reflection on some of the most complex types like big operators, polynomials or matrices. Our implementation restores the ability to perform full-scale reflection on abstract structures, and more generally to compute with them. For instance, addition of two fully instantiated polynomials cannot be evaluated to its actual numerical result but we can refine it to a computable object that will reduce. This is a first step towards having in the same system definitions of objects on which properties can be proved and some of the usual features of a computer algebra system.

However, in its current state, the inner structure of our library is slightly more rigid than necessary: we create a type for computable $\mathbb{Z}$-modules, but in practice, all the operations it contains could be packaged independently. Indeed, on each of these operations we prove only a morphism lemma linking it to its abstract counterpart, whereas in usual algebraic structures, expressing properties like distributivity require access to several operations at once. This specificity would make it possible to reorganise the library and create independent structures for each operation, instead of creating one of them for each type.

To simplify the layout of the library we could also use other packaging methods, like Coq type classes [Sozeau and Oury, 2008]. However, modifying the library to use type classes on top of SSReflect's canonical structures is still on-going work, since we faced some incompatibilities between the different instance resolution mechanisms.

2

Refinements for free!

Cyril Cohen, Maxime Dénès and Anders Mörtberg

Abstract. Formal verification of algorithms often requires that the developer to choose between definitions that are easy to reason about and definitions that are computationally efficient. One way to reconcile both consists in adopting a high-level view when proving correctness and then refining stepwise down to an efficient low-level implementation. Some refinement steps are interesting, in the sense that they improve the algorithms involved, while others only express a switch from data representations geared towards proofs to more efficient ones designed for computation. We relieve the user of these tedious refinements by introducing a framework where correctness is first established in a proof-oriented context and then automatically transported to computation-oriented data structures. Our design is general enough to encompass a variety of mathematical objects, such as rational numbers, polynomials and matrices over refinable structures. Moreover, the rich formalism of the Coq proof assistant enables us to develop this within Coq, without having to maintain an external tool.

Keywords. Data and program refinements, formal proofs, parametricity, Coq, SSReflect.

2.1 Introduction

It is commonly conceived that computationally well-behaved programs and data structures are more difficult to study formally than naive ones. Rich formalisms like the Calculus of Inductive Constructions, on which the Coq [Coq Development Team, 2012] proof assistant relies, allow for several different representations of the same mathematical object so that users can choose the one suiting their needs.

Even simple objects like natural numbers have both a unary representation which features a very straightforward induction scheme and a binary one which is exponentially more compact, but usually entails more involved proofs. Their respective incarnations in the standard library of Coq are the two inductive types `nat` and `N` along with two isomorphisms `N.of_nat : nat -> N` and `N.to_nat : N -> nat`. Recent versions of the library make use of ML-like modules [Chrzaszcz, 2003] and functors to factor programs and proofs over these two types.

The traditional approach to abstraction is to first define an interface specifying operators and their properties, then instantiate it with concrete implementations of the operators with proofs that they satisfy the properties. However, this has at least two drawbacks in our context. First, it is not always obvious how to define the correct interface, and it is not clear if a suitable one even exists. Second, having abstract axioms goes against the type-theoretic view of objects with computational content, which means in practice that proof techniques like small scale reflection, as advocated by the SSReflect extension [Gonthier et al., 2008], are not applicable.

Instead, the approach we describe here consists in proving the correctness of programs on data structures designed for proofs — as opposed to an abstract signature — and then transporting them to more efficient implementations. We distinguish two notions: *program refinements* and *data refinements*. The first of these consists in transforming a program into a more efficient one computing the same thing using a different algorithm, but preserving the involved types. For example, standard matrix multiplication can be refined to a more efficient implementation like Strassen's fast matrix product [Strassen, 1969]. The correctness of this kind of refinements is often straightforward to state. In many cases, it suffices to prove that the two algorithms are extensionally equal. The second notion of refinement consists in changing the data representation on which programs operate while preserving the algorithm, for example a multiplication algorithm on dense polynomials may be refined to an algorithm on sparse polynomials. This kind of refinement is more subtle to express as it involves transporting both programs and their correctness proofs to the new data representation.

The two kinds of refinements can be treated independently and in the following, we focus on data refinements. A key feature of these should be compositionality, meaning that we can combine multiple data refinements. For instance, given both a refinement from dense to sparse polynomials and a refinement from unary to binary integers we should get a refinement from dense polynomials over unary integers to sparse polynomials over binary integers.

In the first paper of this thesis a framework for refining *algebraic structures*, while still allowing a step-by-step approach to prove the correctness of algorithms, is defined. The present work improves several aspects of this framework by considering the following methodology:

1. *relate* a proof-oriented data representation with a more computationally efficient one (section 2.2),

2. *parametrize* algorithms and the data on which they operate by an abstract type and its basic operations (section 2.3),

3. *instantiate* these algorithms with proof-oriented data types and their basic operations, and prove the correctness of that instance,

4. use *parametricity* of the algorithm (with respect to the data representation on which it operates), together with points 2 and 3, to deduce that the algorithm instantiated with the more efficient data representation is also correct (section 2.4).

Further, this paper also contains a detailed example of an application of this new framework to Strassen's algorithm for efficient matrix multiplication (section 2.5). In section 2.6 an overview of related work is presented and the paper is then ended with a discussion on conclusions and future work (section 2.7).

2.2 Data refinements

In this section we will study various data refinements by considering some examples. All of these fit in a general framework of data refinements based on heterogeneous relations which relate *proof-oriented* types for convenient proofs with *computation-oriented* types for efficient computation.

2.2.1 Refinement relations

In some cases we can define (possibly partial) functions from proof-oriented to computation-oriented types and *vice versa*. We call a function from proof-oriented to computation-oriented types an *implementation* function, and a function going the other way around a *specification* function.

Note that a specification function alone suffices to define a refinement relation between the two data types: a proof-oriented term p refines to a computation-oriented term c if the specification of c is p. We write the following helper functions to map respectively total and partial specification functions to the corresponding refinement relations:

```
Definition fun_hrel A B (f : B -> A) : A -> B -> Prop :=
  fun a b => f b = a.

Definition ofun_hrel A B (f : B -> option A) : A -> B -> Prop
  := fun a b => f b = Some a.
```

We will now study some examples of related types that fit in this framework.

Isomorphic types

Isomorphic types correspond to the simple case where the implementation and specification functions are inverse of each other.

The introduction mentions the two types `nat` and `N` which represent unary and binary natural numbers. These are isomorphic, which is witnessed by the implementation function `N.of_nat : nat -> N` and the specification function `N.to_nat : N -> nat`. Here, `nat` is the proof-oriented type and `N` the

computation-oriented one. Another example of isomorphic types is the efficient binary representation `Z` of integers in the Coq standard library that can be declared as a refinement of the unary, `nat`-based, representation `int` of integers in the MathComp library.

Quotients

Quotients correspond to the case where the specification and implementation functions are total and where the specification is a left inverse of the implementation. This means that the computation-oriented type may have "more elements" and that the implementation function is not necessarily surjective (unless the quotient is trivial). In this case the proof-oriented type can be seen as a quotient of the computation-oriented type by an equivalence relation defined by the specification function, *i.e.* two computation-oriented objects are related if their specifications are equal. This way of relating types by quotients is linked to the general notion of quotient types in type theory [Cohen, 2013]. The specification corresponds to the canonical surjection in the quotient, while the implementation corresponds to the choice of a canonical representative. However, here we are not interested in studying the proof-oriented type, which is the quotient type. Instead, we are interested in the computation-oriented type, which is the type being quotiented.

An important example of quotients is the type of polynomials. In the MathComp library these are represented as a record type with a list and a proof that the last element is nonzero, however this proof is only interesting when developing theory about polynomials and not for computation. Hence a computation-oriented type can be just the list of coefficients and the specification function would normalize polynomials by removing zeros in the end.

A better representation of polynomials is sparse Horner normal form [Gregoire and Mahboubi, 2005] which can be implemented as:

```
Inductive hpoly := Pc : A -> hpoly
                 | PX : A -> pos -> hpoly -> hpoly.
```

Here `A` is an arbitrary type and `pos` is the type of positive numbers, the first constructor represents a constant polynomial and `PX a n p` should be interpreted as $a + X^n p$ where a is a constant, n a positive number and p another polynomial in sparse Horner normal form. However, with this representation there are multiple ways to represent the same polynomial. For instance can X^2 be represented either by $0 + X^2 \cdot 1$ or $0 + X^1(0 + X^1 \cdot 1)$. To remedy this we implement a specification function that normalize polynomials and translate them to MathComp polynomials.

Partial quotients

Quotient based refinement relations cover a larger class of data refinements than the relations defined by isomorphisms. There are still interesting examples that are not covered though, for example when the specification function is partial. To illustrate this, let us consider rational numbers. The MathComp library contains a definition where they are defined as pairs of coprime integers with nonzero denominator:

```
Record rat := Rat {
  valq : int * int;
  _ : (0 < valq.2) && coprime `|valq.1| `|valq.2|
}.
```

Here `` `|valq.1| `` and `` `|valq.2| `` denotes the absolute values of the first and second components of the `valq` pair. This definition is well-suited for proofs, notably because elements of type `rat` can be compared using Leibniz equality since they are normalized. But maintaining this invariant during computations is often too costly since it requires multiple gcd computations. Besides, the structure also contains a proof which is not interesting for computations but only for developing the theory of rational numbers.

In order to be able to compute efficiently we would like to refine this to pairs of integers (`int * int`) that are not necessarily normalized and perform all operations on the subset of pairs with nonzero second component. The link between the two representations is depicted in Figure 2.1:

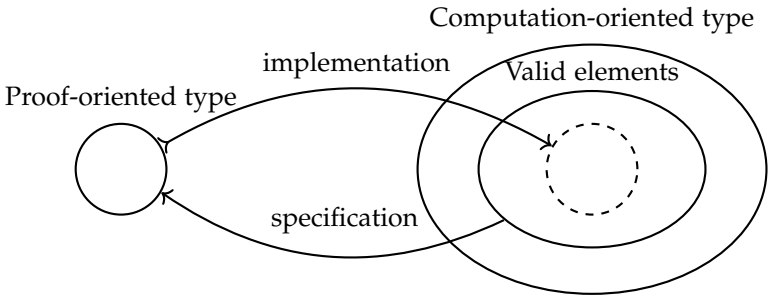


Figure 2.1: Partial quotients

In the example of rational numbers the proof-oriented type is `rat` while the computation-oriented type is `int * int`. Computations should be performed on the subset of valid elements of the computation-oriented type, *i.e.* pairs with nonzero second component. In order to conveniently implement this, the output type of the specification function has been extended to `option A` in order to make it total. The key property of the implementation and specification functions is still that the specification is a left inverse of the implementation. This means that the proof-oriented type can be seen as a quotient of the set of valid elements, *i.e.* the elements that are not sent to `None` by the specification function. For rational numbers the implementation and specification functions together with their correctness looks like:

```
Definition rat_to_Qint (r : rat) : int * int := valq r.
Definition Qint_to_rat (r : int * int) : option rat :=
  if r.2 != 0 then Some (r.1%:Q / r.2%:Q) else None.

Lemma Qrat_to_intK :
  forall (r : rat), Qint_to_rat (rat_to_Qint r) = Some r.
```

The notation `%:Q` is the cast from `int` to `rat`. Here the lemma says that the composition of the implementation with the specification is the identity. Using

this, we get a relation between `rat` and `int * int` by using `ofun_hrel` defined at the beginning of this section:

```
Definition Rrat : rat -> int * int -> Prop
  := ofun_hrel Qint_to_rat.
```

Functional relations

Partial quotients often work for the data types we define, but fails to describe refinement relations on functions. Given two relations `R : A -> B -> Prop` and `R' : A' -> B' -> Prop` we can define a relation on the function space: `R ==> R' : (A -> A') -> (B -> B') -> Prop`. It is a heterogeneous generalization of the `respectful` functions defined for generalized rewriting [Sozeau, 2009].

This definition is such that two functions are related by `R ==> R'` if they send related inputs to related outputs. We can now use this to define the correctness of addition on rational numbers:

```
Lemma Rrat_addq : (Rrat ==> Rrat ==> Rrat) +rat +int*int.
```

The lemma states that if the two arguments are related by `Rrat` then the outputs are also related by `Rrat`.

However, we have left an issue aside: we refined `rat` to `int * int`, but this is not really what we want to do as the type `int` is itself proof-oriented. Thus, taking it as the basis for our computation-oriented refinement of `rat` would be inefficient. Instead, we would like to express that `rat` refines to `C * C` for *any* type `C` that refines `int`. The next section will explain how to program, *generically*, operations in the context of such parametrized refinements. Then, in section 2.4, we will show that correctness can be proved in the specific case when `C` is `int`, and automatically transported to any other refinement by taking advantage of parametricity.

2.2.2 Comparison with the previous approach

We gain in generality with regard to the approach presented in the first paper in several ways. The previous work assumed a total injective implementation function, which intuitively corresponds to a partial isomorphism: the proof-oriented type is isomorphic to a subtype of the computation-oriented type. Since we do not rely on those translation functions anymore, we can now express refinement relations on functions. Moreover, we take advantage of (possibly partial) specification functions, rather than implementation functions.

Another important improvement is that we do not need any notion of equality on the computation-oriented type anymore. Indeed, the development used to rely on Leibniz equality, which prevented the use of setoids [Barthe et al., 2003] as computation-oriented types. In section 2.2.1, we use the setoid `int * int` of rational numbers, but the setoid equality is left implicit. This is in accordance with our principle never to do proofs on computation-oriented types. We often implement algorithms to decide equality, but these are treated as any other operation (section 2.3).

2.2.3 Indexing and using refinements

We use the Coq type class mechanism [Sozeau and Oury, 2008] to maintain a database of lemmas establishing refinement relations between proof-oriented and computation-oriented terms. The way this database is used is detailed in section 2.4.

In order to achieve this, we define a heterogeneous generalization of the `Proper` relations from generalized rewriting [Sozeau, 2009]. We call this class of relations `param` and define it by:

```
Class param (R : A -> B -> Prop) (a : A) (b : B) :=
  param_rel : R a b.
```

Here `R` is meant to be a refinement relation from `A` to `B`, and we can register an instance of this class whenever we have two elements `a` and `b` together with a proof of `R a b`. For example, we register the lemma `Rrat_addq` from section 2.2.1 using the following instance:

```
Instance Rrat_addq :
  param (Rrat ==> Rrat ==> Rrat) +rat +int*int.
```

Given a term x, type class resolution now searches for y together with a proof of `param` R x y. If R was obtained from a specification function, then $x =$ spec y and we can always substitute x by spec y and compute y. This way we can take advantage of our framework to do efficient computation steps within proofs.

2.3 Generic programming

We usually want to provide operations on the computation-oriented type corresponding to operations on the proof-oriented type. For example, we may want to define an addition operation (`addQ`) on computation oriented rationals over an abstract type `C`, corresponding to addition ($+_{rat}$) on `rat`. However this computation-oriented operation relies on both addition ($+_C$) and multiplication ($*_C$) on `C`, so we parametrize `addQ` by ($+_C$) and ($*_C$):

```
Definition addQ C (+C) (*C) : (C * C) -> (C * C) -> (C * C) :=
  fun x y => (x.1 *C y.2 +C y.1 *C x.2, x.2 *C y.2).
```

This operation is correct if ($+_{rat}$) refines to (`addQ C` ($+_C$) ($*_C$)) whenever ($+_{int}$) refines to ($+_C$) and ($*_{int}$) refines to ($*_C$). The refinement from ($+_{rat}$) to (`addQ C` ($+_C$) ($*_C$)) is explained in section 2.4.1.

Since we abstracted over operations of the underlying data type, only one implementation of each algorithm suffices, the same code can be used for doing both correctness proofs and efficient computations as it can be instantiated by both proof-oriented and computation-oriented types and programs. This means that the programs need only be written once and code is never duplicated, which is another improvement compared to the previous development.

In order to ease the writing of this kind of programs and refinement statements in the code, we use operational type classes [Spitters and van der Weegen, 2011] for standard operations like addition and multiplication together with appropriate notations. This means we define a class for each operator

and a generic notation referring to the corresponding operation. For example, in the code of addQ we can always write (+) and (*) and let the system infer the operations,

```
Instance addQ C `{add C, mul C} : add (C * C) :=
  fun x y => (x.1 * y.2 + y.1 * x.2, x.2 * y.2).
```

Here `{add C, mul C} means that C comes with type classes for addition and multiplication operators. Declaring addQ as an instance of addition on C * C enables the use of the generic (+) notation to denote it.

2.4 Parametricity

The approach presented in the above section is incomplete though: once we have proven that the instantiation of a generic algorithm to proof-oriented structures is correct, how can we guarantee that other instances will be correct as well? Doing correctness proofs directly on computation-oriented types is precisely what we are trying to avoid.

Informally, since our generic algorithms are polymorphic in their types and operators, their behavior has to be uniform across all instances. Hence, a correctness proof should be portable from one instance to another, as long as the operators instances are themselves correct.

The exact same idea is behind the interpretation of polymorphism in relational models of pure type systems [Bernardy et al., 2012]. The present section builds on this analogy to formalize the automated transport of a correctness proof from a proof-oriented instance to other instances of the same generic algorithm.

2.4.1 Splitting refinement relations

Let us illustrate the parametrization process by an example on rational numbers. For simplicity, we consider negation which is implemented by:

```
Instance oppQ C `{opp C} : opp (C * C) :=
  fun x => (-C x.1, x.2).
```

The function takes a negation operation in the underlying type C and define negation on C * C by negating the first projection of the pair (the numerator). Now let us assume that C is a refinement of int for a relation Rint : int -> C -> Prop and that we have:

```
(Rint ==> Rint) (-int) (-C)
(Rrat ==> Rrat) (-rat) (oppQ int (-int))
```

The first of these states that the (-C) parameter of oppQ is correctly instantiated, while the second expresses that the proof-oriented instance of oppQ is correct. Assuming this we want to show that (-rat) refines all the way down to oppQ, but instantiated with C and (-C) instead of their proof-oriented counterparts (int and (-int)).

In order to write this formally, we define the product and composition of relations as:

```
R * S := fun x y => R x.1 y.1 /\ S x.2 y.2
R \o S := fun x y => exists z, R x z /\ S z y
```

Using this we can define the relation `RratC : rat -> C * C -> Prop` as:

```
Definition RratC := Rrat \o (Rint * Rint).
```

We want to show that:

```
(RratC ==> RratC) (-rat) (oppQ C (-C))
```

A small automated procedure, relying on type class instance resolution, first splits this goal in two, following the composition `\o` in the definition of `RratC`:

```
(Rrat ==> Rrat) (-rat) (oppQ int (-int))
(Rint * Rint ==> Rint * Rint) (oppQ int (-int)) (oppQ C (-C))
```

The first of these is one of the assumptions while the second relates the results of the proof-oriented instance of `oppQ` to another instance. This is precisely where parametricity comes into play, as we will show in the next section.

2.4.2 Parametricity for refinements

While studying the semantics of polymorphism, Reynolds introduced a relational interpretation of types [Reynolds, 1983]. A reformulation of this is parametricity [Wadler, 1989], which is based on the fact that if a type has no free variable, its relational interpretation expresses a property shared by all terms of this type. This result extends to pure type systems [Bernardy et al., 2012] and provides a meta-level transformation $[\![\cdot]\!]$ defined inductively on terms and contexts. In the closed case, this transformation is such that if $\vdash A : B$, then $\vdash [\![A]\!] : [\![B]\!]\ A\ A$. That is, for any term A of type B, it gives a procedure to build a proof that A is related to itself for the relation interpreting the type B.

The observation we make is that the last statement of section 2.4.1 is an instance of such a *free theorem*. More precisely, we know that $[\![\texttt{oppQ}]\!]$ is a proof of

$$[\![\forall \texttt{Z}, (\texttt{Z} \to \texttt{Z}) \to \texttt{Z} * \texttt{Z} \to \texttt{Z} * \texttt{Z}]\!]\ \texttt{oppQ}\ \texttt{oppQ}$$

which expands to

$$\begin{array}{lll} \forall \texttt{Z} : \texttt{Type}, & \forall \texttt{Z}' : \texttt{Type}, & \forall \texttt{Z}_\texttt{R} : \texttt{Z} \to \texttt{Z}' \to \texttt{Prop}, \\ \forall \texttt{oppZ} : \texttt{Z} \to \texttt{Z}, & \forall \texttt{oppZ}' : \texttt{Z}' \to \texttt{Z}', & [\![\texttt{Z} \to \texttt{Z}]\!]\ \texttt{oppZ}\ \texttt{oppZ}' \to \\ \multicolumn{3}{r}{[\![\texttt{Z} * \texttt{Z} \to \texttt{Z} * \texttt{Z}]\!]\quad (\texttt{oppQ}\ \texttt{Z}\ \texttt{oppZ})\quad (\texttt{oppQ}\ \texttt{Z}'\ \texttt{oppZ}').} \end{array}$$

Then, instantiating `Z` to `int`, `Z'` to `C` and $\texttt{Z}_\texttt{R}$ to `Rint` gives us the exact statement we wanted to prove, since $[\![\texttt{Z} \to \texttt{Z}]\!]$ is what we denoted by $\texttt{Z}_\texttt{R}$ `==>` $\texttt{Z}_\texttt{R}$.

Following the term transformation $[\![\cdot]\!]$, we have designed a logic program in order to derive proofs of closed instances of the parametricity theorem. Indeed, it should be possible in practice to establish the parametric relation between two terms like `oppQ` and itself, since `oppQ` is closed.

For now, we can only express and infer parametricity on polymorphic expressions (no dependent types allowed), by putting the polymorphic types outside the relation. Hence we do not need to introduce a quantification over relations.

2.4.3 Generating the parametricity lemma

Rather than giving the details of how we programmed the proof search using type classes and hints in the Coq system, we instead show an execution of this logic program on our simple example, starting from:

```
(Rrat ==> Rrat) (-rat) (oppQ C (-C))
```

Let us first introduce the variables and their relations, and we get to prove

```
(Rint * Rint) (oppQ int (-int) a) (oppQ C (-C) b)
```

knowing `((Rint ==> Rint) (-int) (-C))` and `((Rint * Rint) a b)`.
By unfolding `oppQ`, it suffices to show that:

```
(Rint * Rint) (-int a.1, a.2) (-C b.1, b.2)
```

To do this we use parametricity theorems for the pair constructor `pair` and eliminators `_.1` and `_.2`. In our context, we have to give manual proofs for them. Indeed, we lack automation for the axioms, but the number of combinators to treat by hand is negligible compared to the number of occurrences in user-defined operations. These lemmas look like:

```
Lemma param_pair :
  forall RA RB, (RA ==> RB ==> RA * RB) pair pair.
Lemma param_fst : forall RA RB, (RA * RB ==> RA) _.1 _.1.
Lemma param_snd : forall RA RB, (RA * RB ==> RB) _.2 _.2.
```

Unfolding the type of the first of these gives:

```
forall (RA : A -> A' -> Prop) (RB : B -> B' -> Prop)
       (a : A) (a' : A') (b : B) (b' : B'),
       RA a a' -> RB b b' -> (RA * RB) (a, b) (a', b')
```

This can be applied to the initial goal, giving two subgoals:

```
Rint (-int a.1) (-C b.1)
Rint a.2 b.2
```

The second of these follow directly from `param_snd` and to show the first it suffices to prove:

```
(Rint ==> Rint) (-int) (-C)
Rint a.1 b.1
```

The first of these is one of the assumptions we started with and the second follows directly from `param_fst`.

2.5 Example: Strassen's fast matrix product

In the first paper an important application of the refinement framework was Strassen's algorithm for the product of two square matrices of size n with time complexity $\mathcal{O}(n^{2.81})$ [Strassen, 1969]. We show here how we adapted it to the new framework described in this paper.

Let us begin with one step of Strassen's algorithm: given a function f which computes the product of two matrices of size `p`, we define, generically, a function `Strassen_step f` which multiplies two matrices of size `p + p`:

```
Variable mxA : nat -> nat -> Type.

Context `{hadd mxA, hsub mxA, hmul mxA, hcast mxA, block mxA}.
Context `{ulsub mxA, ursub mxA, dlsub mxA, drsub mxA}.

Definition Strassen_step {p : positive} (A B : mxA (p+p) (p+p))
  (f : mxA p p -> mxA p p -> mxA p p) : mxA (p+p) (p+p) :=
  let A11 := ulsubmx A in let A12 := ursubmx A in
  let A21 := dlsubmx A in let A22 := drsubmx A in
  let B11 := ulsubmx B in let B12 := ursubmx B in
  let B21 := dlsubmx B in let B22 := drsubmx B in
  let X := A11 - A21 in let Y := B22 - B12 in
  let C21 := f X Y in let X := A21 + A22 in
  let Y := B12 - B11 in let C22 := f X Y in
  let X := X - A11 in let Y := B22 - Y in
  let C12 := f X Y in let X := A12 - X in
  let C11 := f X B22 in let X := f A11 B11 in
  let C12 := X + C12 in let C21 := C12 + C21 in
  let C12 := C12 + C22 in let C22 := C21 + C22 in
  let C12 := C12 + C11 in let Y := Y - B21 in
  let C11 := f A22 Y in let C21 := C21 - C11 in
  let C11 := f A12 B21 in let C11 := X + C11 in
  block_mx C11 C12 C21 C22.
```

The `mxA` variable represents the type of matrices indexed by their sizes. The various operations on this type are abstracted over by operational type classes, as shown in section 2.3. Playing with notations and scopes allows us to make this generic implementation look much like an equivalent one involving MathComp matrices.

Note that the `Strassen_step` function expresses matrix sizes using the `positive` type. These are positive binary numbers, whose recursion scheme matches the one of Strassen's algorithm through matrix block decomposition. This is made compatible with the `nat`-indexed `mxA` type thanks to a hidden coercion `nat_of_pos`.

The full algorithm is expressed by induction over `positive`. However, in order to be able to state parametricity lemmas, we do not use the primitive `Fixpoint` construction. Instead, we use the recursion scheme attached to `positive`:

```
positive_rect : forall (P : positive -> Type),
                (forall p : positive, P p -> P (p~1)) ->
                (forall p : positive, P p -> P (p~0)) ->
                P 1%positive ->
                forall (p : positive), P p
```

We thus implement three functions corresponding to the three cases given by the constructor of the `positive` inductive type: `Strassen_xI` and `Strassen_x0` for even and odd sized matrices, and `Strassen_xH` for matrices of size 1. Strassen's algorithm is then defined as:

```
Definition Strassen :=
```

```
  (positive_rect (fun p => (mxA p p -> mxA p p -> mxA p p))
                 Strassen_xI Strassen_xO Strassen_xH).
```

The `mxA` type and all the associated operational type classes are then instantiated with the MathComp proof-oriented matrix type and operators. In this context, we prove the program refinement from the naive matrix product `mulmx` to Strassen's algorithm:

```
Lemma StrassenP p :
  param (eq ==> eq ==> eq) mulmx (@Strassen p).
```

The proof of this is essentially unchanged from section 1.3.3, the present work improving only the data refinement part. The last step consists in stating and proving the parametricity lemmas. This is done in a context abstracted over both a representation type for matrices and a refinement relation:

```
Context (A : ringType) (mxC : nat -> nat -> Type).
Context (RmxA : forall m n, 'M[A]_(m, n) -> mxC m n -> Prop).
```

Operations on matrices are also abstracted, but we require them to have an associated refinement lemma with respect to the corresponding operation on proof-oriented matrices. For instance, for addition we write as follows:

```
Context `{hadd mxC, forall m n, param (RmxA ==> RmxA ==> RmxA)
         (@addmx A m n) (@hadd_op _ _ _ m n)}.
```

We also have to prove the parametricity lemma associated to our recursion scheme on `positive`:

```
Instance param_elim_positive P P'
  (R : forall p, P p -> P' p -> Prop)
  txI txI' txO txO' txH txH' :
  (forall p, param (R p ==> R (p~1)) (txI p) (txI' p)) ->
  (forall p, param (R p ==> R (p~0)) (txO p) (txO' p)) ->
  (param (R 1) txH txH') ->
  forall p, param (R p) (positive_rect P txI txO txH p)
                        (positive_rect P' txI' txO' txH' p).
```

We declare this lemma as an `Instance` of the `param` type class. This allows to automate data refinement proofs requiring induction over `positive`. Finally, we prove parametricity lemmas for `Strassen_step` and `Strassen`:

```
Instance param_Strassen_step p :
   param (RmxA ==> RmxA ==> (RmxA ==> RmxA ==> RmxA) ==> RmxA)
         (@Strassen_step (@matrix A) p) (@Strassen_step mxC p).

Instance param_Strassen p :
   param (RmxA ==> RmxA ==> RmxA)
         (@Strassen (@matrix A) p) (@Strassen mxC p).
```

Here, the improvement over the first paper is twofold: only one generic implementation of the algorithm is now required and refinement proofs are now mostly automated, including induction steps.

A possible drawback is that our generic description of the algorithms requires all the operators to take the sizes of the matrices involved as arguments.

These sizes are sometimes not necessary for the computation-oriented operators. However, some preliminary benchmarks seem to indicate that this does not entail a significant performance penalty.

2.6 Related work

Our work addresses a fundamental problem: how to change data representations in a compositional way. As such, it is no surprise that it shares aims with other work. We already mentioned ML-like modules and functors, that are available in Coq, but forbid proof methods to have a computational content.

The most general example of refinement relations we consider are partial quotients, which are often represented in type theory by setoids over partial equivalence relations [Barthe et al., 2003] and manipulated using generalized rewriting [Sozeau, 2009]. The techniques we are using are very close to a kind of heterogeneous version of the latter. Indeed, it usually involves a relation `R : A -> A -> Prop` for a given type `A`, whereas our refinement relations have the shape `R : A -> B -> Prop` where `A` and `B` can be two different types.

Some years ago, a plugin was developed for Coq for changing data representations and converting proofs from a type to another [Magaud, 2003]. However, this approach was limited to isomorphic types, and does not provide a way to achieve generic programming (only proofs are ported). Our design is thus more general, and we do not rely on an external plugin which can be costly to maintain.

In [Luo, 1994], a methodology for modular specification and development of programs in type theory is presented. The key idea is to express algebraic specifications using sigma-types which can be refined using refinement maps, and realized by concrete programs. This approach is close to the use of ML-like modules, since objects are abstracted and their behavior is represented by a set of equational properties. A key difference to our work is that these equational properties are stated using an abstract congruence relation, while we aim at proving correctness on objects that can be compared with Leibniz equality, making reasoning more convenient. This is made possible by our more relaxed relation between proof-oriented and computation-oriented representations.

Another way to reconcile data abstraction and computational content is the use of *views* [McBride and McKinna, 2004; Wadler, 1987]. In particular, it allows to derive induction schemes independently of concrete representations of data. This can be used in our setting to write generic programs utilizing these induction schemes for defining recursive programs and proving properties for generic types, in particular `param_elim_positive` (section 2.5) is an example of a view.

The closest work to ours is probably the automatic data refinement tool Autoref implemented independently for Isabelle [Lammich, 2013]. While many ideas, like the use of parametricity, are close to ours, the choice is made to rely on an external tool to synthesize executable instances of generic algorithms and refinement proofs. The richer formalism that we have at our disposal, in particular full polymorphism and dependent types makes it eas-

ier to internalize the instantiation of generic programs.

Another recent work that is related to this paper is [Haftmann et al., 2013] in which the authors explain how the Isabelle/HOL code generator uses data refinements to generate executable versions of abstract programs on abstract types like sets. In the paper they use a refinement relation that is very similar to our partial quotients (they use a domain predicate instead of an option type to denote what values are valid and which are not). The main difference is that they apply data refinements for code generation while in our case this is not necessary as all programs written in Coq can be executed as they are and data refinements are only useful to perform more efficient computations.

2.7 Conclusions and future work

In this paper an approach to data refinements where the user only needs to supply the minimum amount of necessary information and both programs and their correctness proofs gets transported to new data representation has been presented. The three main parts of the approach are:

1. A lightweight and general refinement interface to support any heterogeneous relation between two types,
2. operational type classes to increase generality of implementations and
3. parametricity to automatically transport correctness proofs.

As mentioned in the introduction of this paper, this work is an improvement of the approach presented in the first paper of the thesis. More precisely it improves the approach presented in section 1.5 in the following aspects:

1. Generality: it extends to previously unsupported data types, like the type of non-normalized rationals (section 2.2.2).
2. Modularity: each operator is refined in isolation instead of refining whole algebraic structures (section 2.2.3), as suggested in the future work section of the first paper.
3. Genericity: before, every operation had to be implemented both for the proof-oriented and computation-oriented types, now only one generic implementation is sufficient (section 2.3).
4. Automation: the current approach has a clearer separation between the different steps of data refinements which makes it possible to use parametricity (section 2.4) in order to automate proofs that previously had to be done by hand.

The implementation of points 2, 3 and 4 relies on the type class mechanism of Coq in two different ways: in order to support ad-hoc polymorphism of algebraic operations, and in order to do proof and term reconstruction automatically through logic programming. The automation of proof and term search is achieved by the same set of lemmas as in the previous paper, but now these do not impact the interesting proofs anymore.

The use of operational type classes is very convenient for generic programming. But the more complicated programs get, the more arguments they need. In particular, we may want to bundle operators in order to reduce the size of contexts that users need to write when defining generic algorithms.

The handling of parametricity is currently done by metaprogramming but requires some user input and deals only with polymorphic constructions. We should address these two issues by providing a systematic method of producing parametricity lemmas for inductive types [Bernardy et al., 2012] and extending relation constructions with dependent types. We may adopt Keller and Lasson's [Keller and Lasson, 2012] way of producing parametricity theorems and their proofs for closed terms. It would also be interesting to implement this approach to refinements in a system with internal parametricity like Type Theory in Color [Bernardy and Moulin, 2013].

Currently all formalizations have been done using standard Coq, but it would be interesting to see how Homotopy Type Theory [Univalent Foundations Program, 2013] can be used for simplifying our approach to data refinements. Indeed, in the presence of the univalence axiom, isomorphic structures are equal [Ahrens et al., 2014; Coquand and Danielsson, 2013] which should be useful when refining isomorphic types. Also in the univalent foundations there are ways to represent quotient types (see for example [Rijke and Spitters, 2014]). This could be used to refine types that are related by quotients or even partial quotients.

The work presented in this paper forms the current basis of the CoqEAL library. The development presented in this paper has enabled the addition of some new data refinements like non-normalized rational numbers and polynomials in sparse Horner normal form. The next paper in the thesis discusses the formalization of the Sasaki-Murao algorithm for efficiently computing the characteristic polynomial of a matrix [Sasaki and Murao, 1982] which is another interesting program refinement.

Acknowledgments: The authors are grateful to the anonymous reviewers for their useful comments and feedback. We also thank Bassel Mannaa and Dan Rosén for proof reading the final version of this paper.

3

A Formal Proof of the Sasaki-Murao Algorithm

Thierry Coquand, Anders Mörtberg and Vincent Siles

Abstract. The Sasaki-Murao algorithm computes the characteristic polynomial, and hence the determinant, of any square matrix over a commutative ring in polynomial time. The algorithm itself can be written as a short and simple functional program, but its correctness involves nontrivial mathematics. We here represent this algorithm in type theory with a new correctness proof, using the Coq proof assistant and the SSReflect extension.

Keywords. Formally verified algorithms, program refinements, Sasaki-Murao algorithm, Bareiss' algorithm, Coq, SSReflect.

3.1 Introduction

The goal of this paper is to present a formal proof of the Sasaki-Murao algorithm [Sasaki and Murao, 1982]. This is an elegant algorithm, based on Bareiss' algorithm [Bareiss, 1968], for computing the determinant of a square matrix over an arbitrary commutative ring in polynomial time. Usual presentations of this algorithm are quite complex, and rely on some Sylvester identities [Abdeljaoued and Lombardi, 2004]. We believe that the proof presented in this paper is simpler. The proof was obtained by formalizing the algorithm in type theory (more precisely using the SSReflect extension [Gonthier et al., 2008] to the Coq [Coq Development Team, 2012] proof assistant together with the MathComp library). It does not rely on any Sylvester identities and indeed gives a proof of some of them as corollaries.

This paper also provides an example of how one can use a library of formalized mathematical results to formally verify a computer algebra program. The methodology presented in the first paper of the thesis is used to implement a version of the program that can be efficiently executed inside Coq.

3.2 The Sasaki-Murao algorithm

3.2.1 Matrices

For any $n \in \mathbb{N}$, let $I_n = \{i \in \mathbb{N} \mid i < n\}$ (with $I_0 = \emptyset$). If R is a set, a $m \times n$ matrix of elements of the set R is a function $I_m \times I_n \to R$. Such a matrix can also be viewed as a family of elements (m_{ij}) for $i \in I_m$ and $j \in I_n$.

If M is a $m \times n$ matrix, f a function of type $I_p \to I_m$ and g a function of type $I_q \to I_n$, we define the $p \times q$ submatrix[1] $M(f,g)$ by

$$M(f,g)(i,j) = M(f\ i, g\ j)$$

We often use the following operation on finite maps: if $f : I_p \to I_m$, then $f^+ : I_{1+p} \to I_{1+m}$ is a function such that:

$$\begin{array}{rcl} f^+0 & = & 0 \\ f^+(1+x) & = & 1 + (f\ x) \end{array}$$

If R is a ring, let 1_n be the $n \times n$ identity matrix. Addition and multiplication of matrices can be defined as usual. Any non-empty $m \times n$ matrix M can be decomposed in four components:

- the top-left element m_{00}, which is an element of R
- the top-right line vector $L = m_{01}, m_{02}, \ldots, m_{0(n-1)}$
- the bottom-left column vector $C = m_{10}, m_{20}, \ldots, m_{(m-1)0}$
- the bottom-right $(m-1) \times (n-1)$ matrix $N = m_{(1+i,1+j)}$

That is:

$$M = \left[\begin{array}{c|c} m_{00} & L \\ \hline C & N \end{array}\right]$$

Using this decomposition the central operation of our algorithm can be defined:

$$M' = m_{00}N - CL$$

This operation, $M \longmapsto M'$, transforms a $m \times n$ matrix into a $(m-1) \times (n-1)$ matrix is crucial in the Sasaki-Murao algorithm. In the special case where $m = n = 2$ the matrix M' (of size 1×1) can be identified with the determinant of M.

Lemma 1. *For any $m \times n$ matrix M and any maps $f : I_p \to I_{m-1}$ and $g : I_q \to I_{n-1}$ the following identity holds:*

$$M'(f,g) = M(f^+, g^+)'$$

[1] In the usual definition of submatrix, only some lines and columns are removed, which would be enough for the following proofs. But this more general definition make the Coq formalization easier to achieve.

Proof. This lemma is easy to prove once one has realized two facts:

1. Selecting a submatrix commutes with most of the basic operations on matrices. In particular

$$(M - N)(f, g) = M(f, g) - N(f, g)$$

and

$$(aM)(f, g) = aM(f, g)$$

For multiplication, we have $(MN)(f, g) = M(f, id)N(id, g)$ where id is the identity function.

2. For any matrix M described as a block

$$\begin{bmatrix} m_{00} & L \\ C & N \end{bmatrix}$$

we have that $M(f^+, g^+)$ is the block

$$\begin{bmatrix} m_{00} & L(id, g) \\ C(f, id) & N(f, g) \end{bmatrix}$$

From these two observations, we then have:

$$\begin{array}{rcl} M'(f, g) & = & (m_{00}N - CL)(f, g) \\ & = & m_{00}N(f, g) - C(f, id)L(id, g) \\ M(f^+, g^+)' & = & m_{00}N(f, g) - C(f, id)L(id, g) \end{array}$$

Hence we can conclude that $M'(f, g) = M(f^+, g^+)'$. □

The block decomposition suggests the following possible representation of matrices in a functional language using the data type (where `[R]` is the type of lists over the type `R`, using Haskell notations):

```
data Matrix R = Empty
              | Cons R [R] [R] (Matrix R)
```

A matrix M is hence either the empty matrix `Empty` or a compound matrix `Cons m L C N`. It is direct, using this representation, to define the operations of addition, multiplication on matrices, and the operation M' on non-empty matrices. From this representation, we can also compute other standard views of a $m \times n$ matrix, such as a list of lines $l_1, \ldots, l_m$ or as a list of columns $c_1, \ldots, c_n$.

If M is a square $n \times n$ matrix over a ring R let $|M|$ denote the determinant of M. A *k-minor* of M is a determinant $|M(f, g)|$ for any strictly increasing maps $f : I_k \to I_n$ and $g : I_k \to I_n$. A leading principal minor of M is a determinant $|M(f, f)|$ where f is the inclusion of I_k into I_n.

3.2.2 The algorithm

We present the Sasaki-Murao algorithm using functional programming notations. This algorithm computes in polynomial time, not only the determinant of a matrix, but also its characteristic polynomial. We assume that we have a representation of polynomials over the ring R and that we are given an operation p/q on $R[X]$ which should be the quotient of p by q when q is a *monic* polynomial. This operation is directly extended to an operation M/q of type `Matrix R[X] -> R[X] -> Matrix R[X]`. We define then an auxiliary function ϕ of type `R[X] -> Matrix R[X] -> R[X]` by:

$$\begin{array}{lcl} \phi\ a\ Empty & = & a \\ \phi\ a\ (Mat\ m\ L\ C\ N) & = & \phi\ m\ ((mN - CL)/a) \end{array}$$

From now on, we assume R to be a commutative ring. The correctness proof relies on the notion of *regular* element of a ring: a *regular* element of R is an element a such that $ax = 0$ implies $x = 0$. An alternative (and equivalent) definition is to say that multiplication by a is injective or that a can be cancelled from $ax = ay$ giving $x = y$.

Theorem 1. *Let P be a square matrix of elements of $R[X]$. If all leading principal minors of P are monic, then $\phi\ 1\ P$ is the determinant of P. In particular, if $P = X1_n - M$ for some square matrix M of elements in R, $\phi\ 1\ P$ is the* characteristic polynomial *of M.*

This gives a simple and polynomial time [Abdeljaoued and Lombardi, 2004] algorithm that computes the characteristic polynomial $\chi_M(X)$ of a matrix M. The determinant of M is then $\chi_{-M}(0)$.

3.3 Correctness proof

We first start to prove some auxiliary lemmas:

Lemma 2. *If M is a $n \times n$ matrix and $n > 0$, then*

$$m_{00}^{n-1}|M| = m_{00}|M'|$$

In particular, if m_{00} is regular and $n > 1$, then

$$m_{00}^{n-2}|M| = |M'|$$

Proof. Let us view the matrix M as a list of lines $l_0, \ldots, l_{n-1}$ and let N_1 be the matrix $l_0,\ m_{00}l_1, \ldots,\ m_{00}l_{n-1}$. The matrix N_1 is computed from M by multiplying all of its lines (except the first one) by m_{00}. By the properties of the determinant, we can assert that $|N_1| = m_{00}^{n-1}|M|$.

Let N_2 be the matrix $l_0,\ m_{00}l_1 - m_{10}l_0,\ \ldots,\ m_{00}l_{n-1} - m_{(n-1)0}l_0$. The matrix N_2 is computed from N_1 by subtracting a multiple of l_0 from every line except l_0:

$$m_{00}l_{1+i} \leftarrow m_{00}l_{1+i} - m_{(1+i)0}l_0.$$

By the properties of the determinant, we can assert that $|N_2| = |N_1|$.

Using the definition of the previous section, we can also view the matrix M as the block matrix

$$\begin{bmatrix} m_{00} & L \\ C & N \end{bmatrix}$$

and then the matrix N_2 is the block matrix

$$\begin{bmatrix} m_{00} & L \\ 0 & M' \end{bmatrix}$$

Hence we have $|N_2| = m_{00}|M'|$. From this equality, we can now prove that

$$m_{00}^{n-1}|M| = |N_1| = |N_2| = m_{00}|M'|$$

If m_{00} is regular and $n > 2$, this equality simplifies to $m_{00}^{n-2}|M| = |M'|$. □

Corollary 1. *Let M be a $n \times n$ matrix with $n > 0$. If f and g are two strictly increasing maps from I_k to I_{n-1}, then $|M'(f,g)| = m_{00}^{k-1}|M(f^+,g^+)|$ if m_{00} is regular.*

Proof. Using Lemma 1, we know that $M'(f,g) = M(f^+,g^+)'$, so this corollary follows from Lemma 2. □

Let a be an element of R and M a $n \times n$ matrix. We say that a and M are *related* if and only if

1. a is regular,
2. a^k divides each $k+1$ minor of M, and
3. each principal minor of M is regular.

Lemma 3. *Let a be a regular element of R and M a $n \times n$ matrix, with $n > 0$. If a and M are related, then a divides every element of M'. Furthermore, if $aN = M'$ then m_{00} and N are related and if $n > 1$*

$$m_{00}^{n-2}|M| = a^{n-1}|N|$$

Proof. Let us start by stating two trivial facts: m_{00} is a 1×1 principal minor of M and for all i, j, M'_{ij} is a 2×2 minor of M. These two identities are easily verified by checking the related definitions. Therefore, since a and M are related, m_{00} is regular and a divides all the M'_{ij} (by having $k = 1$), so a divides M'.

Let us write $M' = aN$, we now need to show that m_{00} and N are related, and if $n > 1$,

$$m_{00}^{n-2}|M| = a^{n-1}|N|$$

Let us consider two strictly increasing maps $f : I_k \to I_{n-1}$, $g : I_l \to I_{n-1}$, we have $|M'(f,g)| = u^{k-1}|M(f^+,g^+)|$ by Corollary 1. From the definition of related, we also know that a^k divides $|M(f^+,g^+)|$. Since $M' = aN$ we have $|M'(f,g)| = a^k|N(f,g)|$. If we write $ba^k = |M(f^+,g^+)|$, we have that $ba^k u^{k-1} = a^k|N(f,g)|$. Since a is regular, this equality implies $bu^{k-1} = |N(f,g)|$, and we see that u^{k-1} divides each k minor of N. This

also shows that $|N(f,g)|$ is regular whenever $|M(f^+,g^+)|$ is regular. In particular, each principal minor of N is regular. Finally, since $|M'| = a^{n-1}|N|$ we have $m_{00}^{n-2}|M| = a^{n-1}|N|$ by Lemma 2. □

Since any monic polynomial is also a regular element of the ring of polynomials, Theorem 1 follows directly from Lemma 3 by performing a straightforward induction over the size n. In the case where P is $X1_n - M$ for some square matrix M over R, we can use the fact that any principal minor of $X1_n - M$ is the characteristic polynomial of a smaller matrix, and thus is always monic. In the end, the second part of the conclusion follows directly for the first: $\phi\ 1\ (X1_n - M) = \chi_M(X)$.

Now, we explain how to derive some of the Sylvester identities from Lemma 3. If we look at the computation of $\phi\ 1\ P$ we get a chain of equalities

$$\phi\ 1\ P = \phi\ u_1\ P_1 = \phi\ u_2\ P_2 = \cdots = \phi\ u_{n-1}\ P_{n-1}$$

and we have that u_k is the k:th leading principal minor of P, while P_k is the $(n-k) \times (n-k)$ matrix

$$P_k(i,j) = |P(f_{i,k}, f_{j,k})|$$

where $f_{i,k}(l) = l$ if $l < k$ and $f_{i,k}(k) = i + k$. (We have $P_0 = P$.) Lemma 3 shows that we have for $k < l$

$$|P_k| u_l^{n-l-1} = |P_l| u_k^{n-k-1}$$

This is a Sylvester equality for the matrix $P = X1_n - M$. If we evaluate this identity at $X = 0$, we get the corresponding Sylvester equality for the M matrix over an arbitrary commutative ring.

3.4 Representation in type theory

The original functional program is easily described in type theory, since it is an extension of simply typed λ-calculus:

```
Variable R : ringType.
Variable CR : cringType R.

Definition cpoly := seq CR. (* polynomials are lists *)

Inductive Matrix : Type :=
 | eM (* the empty matrix *)
 | cM of CR & seq CR & seq CR & Matrix.

Definition ex_dvd_step d (M : Matrix cpoly) :=
 mapM (fun x => divp_seq x d) M.

(* main "\phi" function of the algorithm *)
Fixpoint exBareiss_rec (n : nat) (g : cpoly) (M : Matrix cpoly)
  {struct n} : cpoly := match n,M with
```

```
    | _,eM => g
    | O,_ => g
    | S p, cM a l c M =>
      let M' := subM (multEM a M) (mults c l) in
      let M'' := ex_dvd_step g M' in
        exBareiss_rec p a M''
end.

(* This function computes det M for a matrix of polynomials *)
Definition exBareiss (n : nat) (M : Matrix cpoly) : cpoly :=
  exBareiss_rec n 1 M.

(* Applied to xI - M, this gives another definition of the
   characteristic polynomial *)
Definition ex_char_poly_alt (n : nat) (M : Matrix CR) :=
  exBareiss n (ex_char_poly_mx n M).

(* The determinant is the constant part of the char poly *)
Definition ex_bdet (n : nat) (M : Matrix CR) :=
  nth (zero CR) (ex_char_poly_alt n (oppM M)) 0.
```

The `Matrix` type allows to define "ill-shaped" matrices since there are no links between the size of the blocks. When proving correctness of the algorithm, we have to be careful and only consider *valid* inputs.

As we previously said, this is a simple functional program, but its correctness involves nontrivial mathematics. We choose to use the MathComp library to formalize the proof because it already contains many results that we need. The main scheme is to translate this program using MathComp data types, prove its correctness and then prove that both implementations output the same results on valid inputs following the methodology presented in the first paper of the thesis.

First, the MathComp data types (with their respective notations in comments) needed in the formalization are:

```
(* 'I_n *)
Inductive ordinal (n : nat) := Ordinal m of m < n.

(* 'M[R]_(m,n) = matrix R m n *)
(* 'rV[R]_m = 'M[R]_(1,m) *)
(* 'cV[R]_m = 'M[R]_(m,1) *)
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.

(* {poly R} *)
Record polynomial := Polynomial {
    polyseq :> seq R;
    _ : last 1 polyseq != 0
}.
```

Here dependent types are used to express well-formedness. For example, polynomials are encoded as lists (of their coefficients) with a proof that the

last one is not zero. With this restriction, we are sure that one list exactly represent a unique polynomial. Matrices are described as finite functions over the finite sets of indexes.

With this definition, it is easy to define the submatrix $M(f,g)$ along with minors:

```
(* M(f,g) *)
Definition submatrix m n p q (f : 'I_p -> 'I_m)
  (g : 'I_q -> 'I_n) (A : 'M[R]_(m,n)) : 'M[R]_(p,q) :=
    \matrix_(i < p, j < q) A (f i) (g j).

Definition minor m n p (f : 'I_p -> 'I_m) (g : 'I_p -> 'I_n)
  (A : 'M[R]_(m,n)) : R := \det (submatrix f g A).
```

Using MathComp notations and types, we can now write the steps of the functional program (where `rdivp` is the pseudo-division operation [Knuth, 1981] of $R[X]$):

```
Definition dvd_step (m n : nat) (d : {poly R})
   (M : 'M[{poly R}]_(m,n)) : 'M[{poly R}]_(m,n) :=
   map_mx (fun x => rdivp x d) M.

(* main "\phi" function of the algorithm *)
Fixpoint Bareiss_rec m a : 'M[{poly R}]_(1 + m) -> {poly R} :=
  match m return 'M[_]_(1 + m) -> {poly R} with
    | S p => fun (M : 'M[_]_(1 + _)) =>
      let d := M 0 0 in (* up left *)
      let l := ursubmx M in (* up right *)
      let c := dlsubmx M in (* down left *)
      let N := drsubmx M in (* down right *)
      let M' := d *: N - c *m l in
      let M'' := dvd_step a M' in
        Bareiss_rec d M''
    | _ => fun M => M 0 0
  end.

Definition Bareiss (n : nat) (M : 'M[{poly R}]_(1 + n)) :=
  Bareiss_rec 1 M.

Definition char_poly_alt n (M : 'M[R]_(1 + n)) :=
  Bareiss (char_poly_mx M).

Definition bdet n (M : 'M[R]_(1 + n)) :=
 (char_poly_alt (-M))`_0.
```

The main achievement of this paper is the formalized proof of correctness (detailed in the previous section) of this program:

```
Lemma BareissE : forall n (M : 'M[{poly R}]_(1 + n)),
 (forall p (h h' : p.+1 <= 1 + n), monic (pminor h h' M)) ->
  Bareiss M = \det M.
```

```
Lemma char_poly_altE : forall n (M : 'M[R]_(1 + n)),
  char_poly_alt M = char_poly M.

Lemma bdetE n (M : 'M[R]_(1 + n)) : bdet M = \det M.
```

Now we want to prove that the original functional program is correct. Both implementations are very close to each other, so to prove the correctness of the `ex_bdet` program, we just have to show that it computes the same result as `bdet` on similar (valid) inputs. This is one of the advantages of formalizing correctness of program in type theory: one can express the program *and* its correctness in the same language!

```
Lemma exBareiss_recE :
  forall n (g : {poly R}) (M : 'M[{poly R}]_(1 + n)),
    trans (Bareiss_rec g M) =
      exBareiss_rec (1+n) (trans g) (trans M).

Lemma exBareissE : forall n (M : 'M[{poly R}]_(1 + n)),
  trans (Bareiss M) = exBareiss (1 + n) (trans M).

Lemma ex_char_poly_mxE : forall n (M : 'M[R]_n),
  trans (char_poly_mx M) = ex_char_poly_mx n (trans M).

Lemma ex_detE : forall n (M : 'M[R]_(1 + n)),
  trans (bdet M) = ex_bdet (1 + n) (trans M).
```

To link the two implementations, we rely on the version of CoqEAL presented in the first paper. It allows to mirror the main algebraic hierarchy of MathComp with more concrete data types (*e.g.* here we mirror the matrix type `'M[R]_(m,n)` by the computation-oriented type `Matrix CR`, assuming `CR` mirrors `R`) in order to prove the correctness of functional programs using the whole power of the MathComp libraries.

This process is done in the same manner as in [Garillot et al., 2009] using the *canonical structure* mechanism of Coq to overload the `trans` function, which can then be uniformly called on elements of the ring, polynomials or matrices. This function links the MathComp structures to the one we use for the functional program description, ensuring that the correctness properties are translated the program that we actually run in practice.

We can easily prove that translating a MathComp matrix into a `Matrix` always lead to a "valid" `Matrix`, and there is a bijection between MathComp matrices and "valid" matrices, so we are sure that our program computes the correct determinant for all valid inputs.

In the end, the correctness of `ex_bdet` is proved using the lemmas `bdetE` and `ex_bdetE`, stating that for any valid input, `ex_bdet` outputs the determinant of the matrix:

```
Lemma ex_bdet_correct (n : nat) (M : 'M[R]_(1 + n)) :
  trans (\det M) = ex_bdet (1 + n) (trans M).
```

3.5 Conclusions and benchmarks

In this paper the formalization of a polynomial time algorithm for computing the determinant over any commutative ring has been presented. In order to be able to do the formalization in a convenient way a new correctness proof more suitable for formalization has been found. The formalized algorithm has also been refined to a more efficient version on simple types, following the methodology of the first paper of the thesis. This work can be seen as an indication that this methodology works well on more complicated examples involving many different computable structures, in this case matrices of polynomials.

The implementation has been tested using randomly generated matrices with $\mathbb{Z}$ coefficients:

```
(* Random 3x3 matrix *)
Definition M3 :=
  cM 10%Z [:: (-42%Z); 13%Z] [:: (-34)%Z; 77%Z]
     (cM 15%Z [:: 76%Z] [:: 98%Z]
        (cM 49%Z [::] [::] (@eM _ _))).

Time Eval vm_compute in ex_bdet 3 M3.
     = (-441217)%Z
  Finished transaction in 0. secs (0.006667u,0.s)

Definition M10 := (* Random 10x10 matrix *).

Time Eval vm_compute in ex_bdet 10 M10.
     = (-406683286186860)%Z
  Finished transaction in 1. secs (1.316581u,0.s)

Definition M20 := (* Random 20x20 matrix *).

Time Eval vm_compute in ex_bdet 20 M20.
     = 75728050107481969127694371861%Z
  Finished transaction in 63. secs (62.825904u,0.016666s)
```

This indicates that the implementation is indeed quite efficient, we believe that the slowdown of the last computation is due to the fact that the size of the determinant is so large and that the intermediate arithmetic operations has to be done on very big numbers. We have verified this by extracting the function to Haskell and the determinant of the 20×20 matrix can then be computed in `0.273` seconds. The main reasons for this is that the Haskell program has been compiled and have an efficient implementation of arithmetic operations for large numbers.

Part II

Constructive Algebra in Type Theory

4

Coherent and Strongly Discrete Rings in Type Theory

Thierry Coquand, Anders Mörtberg and Vincent Siles

Abstract. In this paper we present a formalization of coherent and strongly discrete rings in type theory. These are fundamental structures in constructive algebra that represents rings in which it is possible to solve linear systems of equations. These structures have been instantiated with Bézout domains (for instance $\mathbb{Z}$ and $k[x]$) and Prüfer domains (generalization of Dedekind domains) so that we get certified algorithms solving systems of equations that are applicable on these general structures. This work can be seen as basis for developing a formalized library of linear algebra over rings.

Keywords. Formalization of mathematics, Constructive algebra, Coq, SSReflect.

4.1 Introduction

One of the fundamental operations in linear algebra is the ability to solve linear systems of equations. The concept of coherent strongly discrete rings abstracts over this ability which makes them an important notion in constructive algebra [Mines et al., 1988]. This makes these rings suitable as a basis for developing computational homological algebra, that is, linear algebra over rings instead of fields [Barakat and Robertz, 2008].

Another reason that these rings are important in constructive algebra is that they generalize the notion of Noetherian rings[1]. Classically any Noethe-

[1] Rings where all ideals are finitely generated.

rian ring is coherent and strongly discrete but the situation in constructive mathematics is more complex and, in fact there is no standard constructive definition of Noetheriannity [Perdry, 2004; Perdry and Schuster, 2011]. Logically, Noetheriannity is expressed by a higher-order condition (it involves quantification over every ideal of the ring) while both coherent and strongly discrete are first-order notions which makes them much more suitable for formalization.

One important example (aside from fields) of coherent strongly discrete rings are Bézout domains which are a non-Noetherian generalization of principal ideal domains (rings where all ideals are generated by one element). The two standard examples of Bézout domains are $\mathbb{Z}$ and $k[x]$ where k is a field. Another example of coherent strongly discrete rings are Prüfer domains with decidable divisibility which are a non-Noetherian generalization of Dedekind domains. The condition of being a Prüfer domain captures what Dedekind thought was the most important property of Dedekind domains [Avigad, 2006], namely the ability to invert ideals (which is usually hidden in classical treatments of Dedekind domains). This property also has applications in control theory [Quadrat, 2003].

All of these notions have been formalized using the SSReflect extension [Gonthier et al., 2008] to the Coq proof assistant [Coq Development Team, 2012] together with the MathComp library. This work can be seen as a generalization of the previous formalization of linear algebra in the MathComp library [Gonthier, 2011].

The main motivation behind this work is that it can be seen as a basis for a formalization of computational homological algebra. This approach is inspired by the one of Homalg [Barakat and Robertz, 2008] where homological algorithms (without formalized correctness proofs) are implemented based on a notion that they call *computable* rings [Barakat and Lange-Hegermann, 2011] which in fact are the same as coherent strongly discrete rings. Another source of inspiration is the work presented in [Lombardi and Quitté, 2011].

This paper is organized as follows: first the formalization of coherent rings, followed by strongly discrete rings, is presented. Next Prüfer domains are discussed together with the proofs that they are both coherent and strongly discrete. This is followed by a section on how to implement a computationally efficient version of the development using the methodology of the first paper. The paper is ended by a section on conclusions and future work.

4.2 Coherent rings

Given a ring R (in our setting commutative but it is possible to consider non-commutative rings as well [Barakat and Lange-Hegermann, 2011]) one important problem to study is how to solve linear systems over R.

Given a rectangular matrix M over R we want to find a finite number of solutions $X_1, \ldots, X_n$ of the system $MX = 0$ such that any solution is of the form $a_1X_1 + \cdots + a_nX_n$ where $a_1, \ldots, a_n \in R$. If this is possible, we say that the module of solutions of the system $MX = 0$ is finitely generated. This can

be reformulated with matrices: we want to find a matrix L such that

$$MX = 0 \leftrightarrow \exists Y.\, X = LY$$

A ring is *coherent* if for any matrix M it is possible to compute a matrix L such that this holds. If this is the case it follows that $ML = 0$.

For this it is enough to consider the case where M has only one line. Indeed, assume that for any $1 \times n$ matrix M we can find a $n \times m$ matrix L such that $MX = 0$ iff $X = LY$ for some Y. To solve the system

$$M_1X = \cdots = M_kX = 0$$

where each M_i is a $1 \times n$ matrix first compute L_1 such that $M_1X = 0$ iff $X = LY_1$ for some Y_1. Next compute L_2 such that $M_2L_1Y_1 = 0$ iff $Y_1 = L_2Y_2$. At the end we obtain $L_1, \ldots, L_k$ such that $M_1X = \cdots = M_kX = 0$ iff X is of the form $L_1 \cdots L_kY$. So $L_1 \cdots L_k$ provides a system of generators for the solution of the system.

Hence it is sufficient to formulate the condition for coherent rings as: For any *row* matrix M it is possible to find a matrix L such that

$$MX = 0 \leftrightarrow \exists Y.\, X = LY$$

In the formal development, coherent rings have been implemented using mixins and canonical structures as in [Garillot et al., 2009]. In the MathComp libraries matrices are represented by finite functions over pairs of ordinals (the indices):

```
(* 'I_n *)
Inductive ordinal (n : nat) := Ordinal m of m < n.

(* 'M[R]_(m,n) = matrix R m n *)
(* 'rV[R]_m = 'M[R]_(1,m) *)
(* 'cV[R]_m = 'M[R]_(m,1) *)
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.
```

Hence the size of the matrices need to be known when implementing coherent rings. But in general the size of L cannot be predicted so we need an extra function that computes this:

```
Record mixin_of (R : ringType) := Mixin {
  size_solve : forall m, 'rV[R]_m -> nat;
  solve_row : forall m (V : 'rV[R]_m), 'M[R]_(m,size_solve V);
  _ : forall m (V : 'rV[R]_m) (X : 'cV[R]_m),
    reflect (exists Y, X = solve_row V *m Y) (V *m X == 0)
}.
```

Here `V *m X == 0` is the boolean equality of matrices and the specification says that this is reflected by the existence statement. An alternative to having a function computing the size would be to output a dependent pair but this has the undesired behavior that the pair has to be destructed when stating lemmas about it which in turn would mean that these lemmas would be cumbersome to use as it would not be possible to rewrite with them directly.

Using this we have implemented the algorithm for computing the generators of a system of equations:

```
Fixpoint solveMxN (m n : nat) :
  forall (M : 'M_(m,n)), 'M_(n,size_solveMxN M) :=
    match m with
      | S p => fun (M : 'M_(1 + _,n)) =>
        let L1 := solve_row (usubmx M)
        in L1 *m solveMxN (dsubmx M *m L1)
      | _ => fun _ => 1%:M
   end.

Lemma solveMxNP : forall m n (M : 'M[R]_(m,n)) (X : 'cV[R]_n),
  reflect (exists Y, X = solveMxN M *m Y) (M *m X == 0).
```

In order to instantiate this structure one can of course directly give an algorithm that computes the solution of a single row system. However there is another approach that will be used in the rest of the paper that is based on the intersection of finitely generated ideals.

4.2.1 Ideal intersection and coherence

In the case when R is an integral domain one way to prove that R is coherent is to show that the intersection of two finitely generated ideals is again finitely generated. This amounts to given two ideals $I = (a_1, \ldots, a_n)$ and $J = (b_1, \ldots, b_m)$ compute generators $(c_1, \ldots, c_k)$ of $I \cap J$. For $I \cap J$ to be the intersection of I and J it should satisfy $I \cap J \subseteq I$, $I \cap J \subseteq J$ and

$$\forall x.\, x \in I \wedge x \in J \rightarrow x \in I \cap I$$

A convenient way to express this in Coq is to use strongly discrete rings that is discussed in section 4.3. For now we just assume that we can find generators of the intersection of two finitely generated ideals (represented using row vectors) and column vectors V and W such that `I *m V = I ∩ J` and `J *m W = I ∩ J`. Using this there is an algorithm to compute generators of the solutions of a system:

$$m_1 x_1 + \cdots + m_n x_n = 0$$

The main idea is to compute generators, M_0, of the solution for $m_2 x_2 + \cdots + m_n x_n = 0$ by recursion and compute generators $t_1, \ldots, t_p$ of $(m_1) \cap (-m_2, \ldots, -m_n)$ together with V and W such that

$$(m_1)V = (t_1, \ldots, t_p)$$
$$(-m_2, \ldots, -m_n)W = (t_1, \ldots, t_p)$$

The generators of the module of solutions are then given by:

$$\begin{bmatrix} V & 0 \\ W & M_0 \end{bmatrix}$$

This has been implemented by:

```
Fixpoint solve_int m : forall (M : 'rV_m),'M_(m,size_int M) :=
  match m with
```

```
  | S p => fun (M' : 'rV_(1 + p)) =>
          let m1 := lsubmx M' in
          let ms := rsubmx M' in
          let M0 := solve_int ms in
          let V := cap_wl m1 (-ms) in
          let W := cap_wr m1 (-ms) in
          block_mx (if m1 == 0 then delta_mx 0 0 else V) 0
                   (if m1 == 0 then 0 else W) M0
  | 0 => fun _ => 0
end.

Lemma solve_intP : forall m (M : 'rV_m) (X : 'cV_m),
  reflect (exists Y, X = solve_int M *m Y) (M *m X == 0).
```

Here `cap_wl` computes V and `cap_wr` computes W, their implementation will be discussed in section 4.3.1. Note that some special care has to be taken if m_1 is zero, if this is the case we output a matrix:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cdots & 0 & M_0 \end{bmatrix}$$

However it would be desirable to output

$$\begin{bmatrix} 1 & 0 \\ 0 & M_0 \end{bmatrix}$$

But this would not have the correct size. This could be solved by having a more complicated function that output a sum type with matrices of two different sizes. This would give slightly more complicated proofs so we decided to pad with zeros instead. In section 4.5 we will discuss how to implement a more efficient algorithm without any padding that is more suitable for computation.

4.3 Strongly discrete rings

An important notion in constructive mathematics is the notion of *discrete* ring, that is, rings with decidable equality. Another important notion is *strongly discrete* rings, these are rings where membership in finitely generated ideals is decidable. This means that if $x \in (a_1, \ldots, a_n)$ there is an algorithm computing $w_1, \ldots, w_n$ such that $x = \sum a_i w_i$.

Examples of such rings are multivariate polynomial rings over discrete fields (via Gröbner bases [Cox et al., 2006; Lombardi and Perdry, 1998]) and Bézout domains with explicit divisibility, that is, whenever $a \mid b$ one can compute x such that $b = xa$. We have represented strongly discrete rings in Coq as:

```
Inductive member_spec (R : ringType) n (x : R) (I : 'rV[R]_n)
  : option 'cV[R]_n -> Type :=
  | Member J of x%:M = I *m J : member_spec x I (Some J)
  | NMember of (forall J, x%:M != I *m J) : member_spec x I None.
```

```
Record mixin_of R := Mixin {
  member : forall n, R -> 'rV[R]_n -> option 'cV[R]_n;
  _ : forall n x (I : 'rV[R]_n), member_spec x I (member x I)
}.
```

The structure of strongly discrete rings contains a function taking an element and a row vector (with the generators of the ideal) and return an `option` type with a column vector. This is `Some J` if `x` can be written as `IJ` and if it is `None` then there should also be a proof that there cannot be any `J` satisfying `x = IJ`.

4.3.1 Ideal theory

In the development we have chosen to represent finitely generated ideals as row vectors, so an ideal in `R` with `n` generators is represented as a row matrix of type `'rV[R]_n`. This way operations on ideals can be implemented using functions on matrices and properties can be proved using the matrix library.

A nice property of strongly discrete rings is that the inclusion relation of finitely generated ideals is decidable. This means that we can decide if $I \subseteq J$ and if this is the case express every generator of I as a linear combination of the generators of J. This is represented in Coq by:

```
Definition subid m n (I : 'rV[R]_m) (J : 'rV[R]_n) :=
  [forall i : 'I_m, member (I 0 i) J].

Notation "A <= B" := (subid A B).
Notation "A == B" := ((A <= B) && (B <= A)).

Lemma subidP : forall m n (I : 'rV[R]_m) (J : 'rV[R]_n),
  reflect (exists W, I = J *m W) (I <= J).
```

Note that this is expressed using matrix multiplication, so `subidP` says that if `I <= J` then every generator of `I` can be written as a linear combination of generators of `J`.

Ideal multiplication is an example where it is convenient to represent ideals as row vectors. As the product of two finitely generated ideals is generated by all products of generators of the ideals this can be expressed compactly using matrix operations:

```
Definition mulid m n (I : 'rV_m) (J : 'rV_n) : 'rV_(m * n) :=
  mxvec (I^T *m J).

Notation "I *i J" := (mulid I J).
```

Here `mxvec` flattens `'M[R]_(m,n)` to a row vector `'rV[R]_(m * n)` and `I^T` is the transpose of `I`. By representing ideals as row vectors we get compact definitions and quite simple proofs as the theory already developed about matrices can be used when proving properties of ideal operations.

It is also convenient to specify what the intersection of `I` and `J` is: it is an ideal `K` such that `K <= I`, `K <= J` and `forall (x : R), member x I -> member x J -> member x K`. So in order to prove that an integral domain is coherent it suffices to give an algorithm that computes `K` and prove that it satisfies these

three properties. The `cap_wr` and `cap_wl` functions used in `solve_with_int` can then be implemented easily by explicitly computing `W` in `subidP`.

4.3.2 Coherent strongly discrete rings

If a ring R is both coherent and strongly discrete it is not only possible to solve homogeneous systems $MX = 0$ but also any system $MX = A$. The algorithm for computing this is expressed by induction on the number of equations where the case of one equation follow directly from the fact that the ring is strongly discrete. In the other case the matrix looks like:

$$\begin{bmatrix} R_1 \\ M \end{bmatrix} X = \begin{bmatrix} a_1 \\ A \end{bmatrix}$$

First compute generators G_1 for the module of system of solutions of $R_1 X = 0$ and test if $a_1 \in R_1$, if this is not the case the system is not solvable and otherwise we get W_1 such that $R_1 W_1 = a_1$. Next compute by recursion the solution S of $MG_1 X = A - MW_1$ such that $MG_1 S = A - MW_1$. The solution to the initial system is then $W_1 + G_1 S$ as

$$\begin{bmatrix} R_1 \\ M \end{bmatrix} (W_1 + G_1 S) = \begin{bmatrix} R_1 W_1 + R_1 G_1 S \\ MW_1 + MG_1 S \end{bmatrix} = \begin{bmatrix} a_1 \\ A \end{bmatrix}$$

This has been implemented in Coq by:

```
Fixpoint solveGeneral m n :
  'M[R]_(m,n) -> 'cV[R]_m -> option 'cV[R]_n := match m with
   | S p => fun (M: 'M[R]_(1 + _,n)) (A : 'cV[R]_(1 + _)) =>
     let G1 := solve_row (usubmx M) in
     let W1 := member (A 0 0) (usubmx M) in
     obind (fun w1 : 'cV_n =>
       obind (fun S => Some (w1 + G1 *m S))
         (solveGeneral (dsubmx M *m G1)
                       (dsubmx A - dsubmx M *m w1))
     ) W1
   | _ => fun _ _ => Some 0
  end.

Inductive SG_spec m n (M : 'M[R]_(m,n)) (A : 'cV[R]_m)
  : option 'cV[R]_n -> Type :=
  | HasSol X0 of (forall (X : 'cV[R]_n),
                  reflect (exists Y, X = solveMxN M *m Y + X0)
                          (M *m X == A)) : SG_spec M A (Some X0)
  | NoSol of (forall X, M *m X != A) : SG_spec M A None.

Lemma solveGeneralP m n : (M : 'M[R]_(m,n)) (A : 'cV[R]_m),
  SG_spec M A (solveGeneral M A).
```

Here `obind` is the bind operation for the option type which applies the function if the output is `Some` and returns `None` otherwise.

4.3.3 Bézout domains are coherent and strongly discrete

An example of a class of rings that are coherent and strongly discrete rings are Bézout domains with explicit divisibility. These are integral domains where every finitely generated ideal is principal (generated by one element). The two main examples of Bézout domains are Z and $k[x]$ where k is a discrete field.

Bézout domains can also be characterized as rings with a gcd operation in which there is a function computing the elements of the Bézout identity:

```
Inductive bezout_spec R (a b : R) : R * R -> Type :=
  BezoutSpec x y of
    gcdr a b %= x * a + y * b : bezout_spec a b (x,y).

Record mixin_of R := Mixin {
  bezout : R -> R -> (R * R);
   _ : forall a b, bezout_spec a b (bezout a b)
}.
```

This means that given a and b one can compute x and y such that $xa + by$ is associate[2] to $\gcd(a, b)$. Based on this it is straightforward to implement a function that given a finitely generated ideal $(a_1, \ldots, a_n)$ computes g (this g is the greatest common divisor of all the a_i) such that $(a_1, \ldots, a_n) \subseteq (g)$ and $(g) \subseteq (a_1, \ldots, a_n)$.

We first prove that Bézout domains are strongly discrete. To test if $x \in (a_1, \ldots, a_n)$ in first compute a principal ideal (g) and then test if $g \mid x$ and if this is the case we we can construct the witness and otherwise we know that $g \notin (a_1, \ldots, a_n)$. This has been implemented in Coq by:

```
Definition bmember n (x : R) (I : 'rV[R]_n) :=
  match x %/? principal_gen I with
    | Some a => Some (principal_w1 I *m a%:M)
    | None => None
  end.

Lemma bmember_correct : forall n (x : R) (I : 'rV[R]_n),
  member_spec x I (bmember x I).
```

Here `%/?` is the explicit divisibility function of `R`, `principal_gen` is the generator of the principal ideal generating `I` and `principal_w1 I` is the witness that $(g) \subseteq I$.

For showing that Bézout domains are coherent let I and J be two finitely generated ideals and compute principal ideals such that $I = (a)$ and $J = (b)$. Now it easy to prove that $I \cap J = (\mathrm{lcm}(a, b))$, where $\mathrm{lcm}(a, b)$ is the lowest common multiple of a and b which is computable in our setting as any Bézout ring is a GCD domain with explicit divisibility. Hence we have now proved that both $\mathbb{Z}$ and $k[x]$ are both coherent and strongly discrete which means that we can solve arbitrary systems of equations over them.

[2] a and b are *associates* if $a \mid b$ and $b \mid a$, or equivalently that there exists a unit $u \in R$ such that $a = bu$.

4.4 Prüfer domains

Another class of rings that are coherent are *Prüfer domains*. These can be seen as non-Noetherian analogues of Dedekind domains and have many different characterizations [Fuchs and Salce, 2001]. The one we choose here is the one in [Lombardi and Quitté, 2011] that says that a Prüfer domain is an integral domains where given any x and y there exist u, v and w such that

$$\begin{aligned} ux &= vy \\ (1-u)y &= wx \end{aligned}$$

This is can be represent in Coq by:

```
Record mixin_of R := Mixin {
  prufer: R -> R -> (R * R * R);
  _ : forall x y, let: (u,v,w) := prufer x y in
      u * x = v * y /\ (1 - u) * y = w * x
}.
```

We require that Prüfer domains have explicit divisibility so that it is possible for us to prove that they are strongly discrete. This means that we can use the library of ideal theory developed for strongly discrete rings when proving that they are coherent. However, it would be possible to prove that Prüfer domains are coherent without assuming explicit divisibility [Lombardi and Quitté, 2011].

The most basic examples of Prüfer domains are Bézout domains (in particular $\mathbb{Z}$ and $k[x]$). However there are many other examples, for instance if R is a Bézout domain then the ring of elements integral over R is a Prüfer domain, this gives examples from algebraic geometry like $k[x,y]/(y^2+x^4-1)$ and algebraic number theory like $\mathbb{Z}[\sqrt{-5}]$.

4.4.1 Principal localization matrices and strong discreteness

The key algorithm in the proof that Prüfer domains with explicit divisibility are both strongly discrete and coherent is an algorithm computing a *principal localization matrix* of an ideal [Ducos et al., 2004]. This means that given a finitely generated ideal $(x_1,\ldots,x_n)$ compute a $n \times n$ matrix $M = (a_{ij})$ such that:

$$\sum_i a_{ii} = 1$$

and

$$\forall i\, j\, l.\ a_{lj}x_i = a_{li}x_j$$

In MathComp the first of these is a bit problematic as there is no constraint saying that a matrix has to be nonempty and if a matrix is empty the sum will be 0. Hence we express the property like this:

```
Definition P1 m (M : 'M[R]_m) :=
  \big[+%R/0]_(i : 'I_m) (M i i) = (0 < m)%:R.
```

```
Definition P2 m (X : 'rV[R]_m) (M : 'M[R]_m) :=
  forall (i j l : 'I_m), (M l j) * (X 0 i) = (M l i) * (X 0 j).

Definition isPLM m (X : 'rV[R]_m) (M : 'M[R]_m) :=
  P1 M /\ P2 X M.
```

The first statement uses an implicit coercion from booleans to rings where `false` is coerced to `0` and `true` to `1`. The algorithm computing a principal localization matrix, `plm`, is quite involved so we have omitted it from this presentation, the interested reader should have a look in the development and at the proofs in [Ducos et al., 2004] and [Lombardi and Quitté, 2011]. We have proved that this algorithm satisfies the above specification:

```
Lemma plmP : forall m (I : 'rV[R]_m), isPLM I (plm I).
```

The reason that principal localization matrices are interesting is that they give a way to compute the *inverse* of a finitely generated ideal I, this is a finitely generated ideal J such that IJ is principal. In fact if $I = (x_1, \ldots, x_n)$ and $M = (a_{ij})$ its principal localization matrix then the following property holds:

$$(x_1, \ldots, x_n)(a_{1i}, \ldots, a_{ni}) = (x_i)$$

That is, every column of M is an inverse to I. In Coq:

```
Lemma col_plm_mulr n (I : 'rV[R]_n.+1) i :
  I *m col i (plm I) = (I 0 i)%:M.
```

This means that we can define an algorithm for computing the inverse of ideals in Prüfer domains:

```
Definition inv_id n (i : 'I_n) (I : 'rV[R]_n) : 'rV[R]_n :=
  (col i (plm I))^T.

Lemma inv_idP n (I : 'rV[R]_n) i :
  (inv_id i I *i I == (I 0 i)%:M).
```

Here `*i` is ideal multiplication. Using this it is possible to prove that Prüfer domains with explicit divisibility are strongly discrete. To compute if $x \in I$ first compute J such that $IJ = (a)$. Now $x \in I$ iff $(x) \subseteq I$ iff $xJ \subseteq (a)$. This can be tested if we can decide when an element is divisible by a. The implementation of this is:

```
Definition pmember n (x : R) : 'rV[R]_n -> option 'cV[R]_n :=
  match n with
  | S p => fun (I : 'rV[R]_p.+1) =>
      let a := plm I in
      if [forall i, I 0 i %| a i i * x]
        then Some (\col_i odflt 0 (a i i * x %/? I 0 i))
        else None
  | _ => fun _ => if x == 0 then Some 0 else None
end.

Lemma pmember_correct : forall n (x : R) (I : 'rV[R]_n),
  member_spec x I (pmember x I).
```

Here `[forall i, I 0 i %| a i i * x]` is a test that all of the generators of I divides $a_{ii}x$. Hence our implementation of Prüfer domains is strongly discrete which means that the theory about ideals can be used when proving that they are coherent.

4.4.2 Coherence

The key property of ideals in Prüfer domains for computing the intersection is that given two finitely generated ideals I and J they satisfy:

$$(I+J)(I\cap J) = IJ$$

This means that we can devise an algorithm for computing generators for the intersection by first computing $(I+J)^{-1}$ such that $(I+J)^{-1}(I+J) = (a)$ and then we get that

$$I\cap J = \frac{(I+J)^{-1}IJ}{a}$$

Note the use of division here, in fact it is possible to compute the intersection without assuming division but then the algorithm is more complicated. Using this the function for computing the generators of the intersection is:

```
Definition pcap (n m : nat) (I : 'rV[R]_n) (J : 'rV[R]_m) :
  'rV[R]_(pcap_size I J).+1 := match find_nonzero (I +i J) with
  | Some i => let sIJ := I +i J in
              let a := sIJ 0 i in
              let acap := inv_id i sIJ *i I *i J in
              (0 : 'M_1) +i (\row_i (odflt 0 (acap 0 i %/? a)))
  | None => 0
end.
```

The reason to add 0 as a generator of the ideal is simply to have the correct size as the formalized proof that R is coherent if $I\cap J$ is computable requires that $I\cap J$ is nonempty. Now we have an algorithm for computing the intersection, but to prove that this is indeed the intersection we need to prove the property that we used:

```
Lemma pcap_id (n m : nat) (I : 'rV[R]_n) (J : 'rV[R]_m) :
  ((I +i J) *i pcap I J == I *i J).
```

Using this it is possible to prove that pcap compute the intersection:

```
Lemma pcap_subidl m n (I: 'rV_m) (J: 'rV_n): (pcap I J <= I).

Lemma pcap_subidr m n (I: 'rV_m) (J: 'rV_n): (pcap I J <= J).

Lemma pcap_member m n x (I : 'rV[R]_m) (J : 'rV[R]_n) :
  member x I -> member x J -> member x (pcap I J).
```

Hence we have now proved that Prüfer domains with explicit divisibility are coherent and strongly discrete. We can hence not only solve homogeneous systems over them but also any linear system of equations.

4.4.3 Examples of Prüfer domains

As mentioned before any Bézout domain is a Prüfer domain. The proof of this is straightforward:

```
Definition bezout_calc (x y: R) : (R * R * R) :=
  let: (g,c,d,a,b) := egcdr x y in (d * b, a * d, b * c).

Lemma bezout_calcP (x y : R) :
  let: (u,v,w) := bezout_calc x y in
  u * x = v * y /\ (1 - u) * y = w * x.
```

Here `egcdr` is the extended Bézout algorithm where g is the gcd of x and y, $x = ag$, $y = bg$ and $ca + db = 1$. We have not yet formalized the proofs that $\mathbb{Z}[\sqrt{-5}]$ and $k[x,y]/(y^2 - 1 + x^4)$ are Prüfer domains, but we have implemented them in Haskell [Mörtberg, 2010].[3]

4.5 Computations

The algorithms in the paper are all presented on structures using rich dependent types which is convenient when proving properties, but for computation this is not necessary. In fact it can be more efficient to implement the algorithms using simple types instead, an example of this is matrices: As explained in section 4.2 they are represented using finite functions from the indices (represented using ordinals). But this representation is not suitable for computation as finite functions are represented by their graph which has to be traversed linearly each time the function is evaluated. To remedy this we use the approach presented in the first paper where matrices are represented using lists of lists and implement efficient versions of the algorithms on this representations instead. These algorithms are then linked to the inefficient versions using translation lemmas. Recall the methodology of the first paper:

1. Implement a proof-oriented version of the algorithm using MathComp structures and use the libraries to prove properties about them.
2. Refine this algorithm into an efficient one still using MathComp structures and prove that it behaves like the proof-oriented version.
3. Translate the MathComp structures and the efficient algorithm to the low-level data types, ensuring that they will perform the same operations as their high-level counterparts.

So far we have only presented step 1. The second step involves giving more efficient algorithms, a good example of this is the algorithms on ideals. A simple optimization that can be made is to ensure that there are no zeros as generators in the output of the ideal operations. The goal would then be to prove that the more efficient operations generates the same ideal as the original operation. Another example is `solve_int` that can be implemented without padding with zeros, this would then be proved to produce a set of

[3] This development can be found at
http://hackage.haskell.org/package/constructive-algebra

solution of the system and then be refined to a more efficient algorithm on list based matrices.

The final step corresponds to implementing "computable" counterparts of the structures that we presented so far based on simple types. For example is computable coherent rings implemented as:

```
Record mixin_of (R : coherentRingType)
                (CR : cstronglyDiscreteType R) := Mixin {
  csize_solve : nat -> seqmatrix CR -> nat;
  csolve_row : nat -> seqmatrix CR -> seqmatrix CR;
  _ : forall n (V : 'rV[R]_n),
    seqmx_of_mx CR (solve_row V) =
    csolve_row n (seqmx_of_mx _ V);
  _ : forall n (V: 'rV[R]_n),
    size_solve V = csize_solve n (seqmx_of_mx _ V)
}.
```

Here `seqmatrix` is the list based representation of matrices with the translation function `seqmx_of_mx` from MathComp matrices to matrices defined using lists. Using this more efficient versions of the algorithms presented above can be implemented simply by changing the functions on MathComp matrices to functions on `seqmatrix`:

```
Fixpoint csolveMxN m n (M : seqmatrix CR) : seqmatrix CR :=
  match m with
    | S p =>
      let u := usubseqmx 1 M in
      let d := dsubseqmx 1 M in
      let G := cget_matrix n u in
      let k := cget_size n u in
      let R := mulseqmx n k d G in
        mulseqmx k (csize_solveMxN p k R) G (csolveMxN p k R)
    | _ => seqmx1 CR n
end.

Lemma csolveMxNE : forall m n (M : 'M[R]_(m,n)),
  seqmx_of_mx _ (solveMxN M) = csolveMxN m n (seqmx_of_mx _ M).
```

The lemma states that solving the system on MathComp matrices and then translating is the same as first translating and then compute the solution using the list based algorithm. The proof of this is straight-forward as all of the functions of the algorithm have translation lemmas.

This way we have implemented all of the above algorithms and instances and made some computations with $\mathbb{Z}$ using the algorithms for Bézout domains: First we can compute the generators of $(2) \cap (3,6)$:

```
Eval vm_compute in (cbcap 1 2 [::[::2]] [::[::3; 6]]).
     = [:: [:: 6]]
```

Next we can test if $6 \in (2)$:

```
Eval vm_compute in (cmember 1%N 6 [::[:: 2]]).
     = Some [:: [:: 3]]
```

It is also possible to solve the homogeneous system:

$$\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

```
Eval vm_compute in (csolveMxN 2 2 [::[:: 1;2];[::2;4]]).
     = [:: [:: 2; 0];
          [:: -1; 0]]
```

and the inhomogeneous system:

$$\begin{bmatrix} 2 & 3 \\ 4 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 8 \end{bmatrix}$$

```
Eval vm_compute in (csolveGeneral 2 2 [::[:: 2; 3]; [:: 4; 6]]
                                      [::[:: 4];[:: 8]]).
     = Some [:: [:: -4];
                [:: 4]]
```

We can also do some computations on the algorithms for Prüfer domains using $\mathbb{Z}$:

```
Eval vm_compute in (cplm 3 [::[:: 2; 3; 5]]).
     = [:: [:: 8; 12; 20];
          [:: 12; 18; 30];
         [:: -10; -15; -25]]

Eval vm_compute in (cinv_id 2 0 [:: [:: 2; 3]]).
     = [:: [:: -2; 2]]
```

The first computation compute the principal localization matrix of $(2,3,5)$ and the second compute the inverse of the ideal $(2,3)$.

4.6 Conclusions and future work

In this paper we have represented in type theory interesting and mathematically nontrivial results in constructive algebra. The algorithms based on coherent and strongly discrete rings have been refined to more efficient algorithms on simple data types, this way we get certified mathematical algorithms that are suitable for computation. This work can hence be seen as an example that the methodology presented in the first paper is applicable on more complicated structures as well. However, this paper also shows that the methodology of the first paper is quite verbose. By instead doing the refinements using the approach in the second paper we would have to only implement the algorithms once and not duplicate any code.

The kind of normalization of the generators of ideals discussed in the previous section would be interesting to use, not only for efficient computation, but also for doing proofs. This would involve redefining the type of ideals as records with a proof that they do not contain any zeros (and possibly other properties as well). By enforcing more properties like this on the generators some corner cases could be removed in the proofs.

In the future it would be interesting to prove that multivariate polynomial rings over discrete fields are coherent and strongly discrete. This would require a formalization of Gröbner bases and the Buchberger algorithm which has already been done in Coq [Persson, 2001; Théry, 1998]. It would be interesting to reimplement this using SSReflect and compare the complexity of the formalizations.

A consequence of the choice of using the MathComp library for the formalization is that it is difficult to formalize the notions in full generality, for instance all rings are assumed to be discrete. Also in constructive algebra ideal theory is usually developed without assuming decidable ideal membership, but in our experience are both the MathComp library and the SSReflect tactics best suited for theories with decidable functions. This is the reason that we only consider Prüfer domains with explicit divisibility as this means that they are strongly discrete which in turn means that we can use the library of ideal theory when proving that they are coherent. We actually started to formalize the coherence proof without assuming explicit divisibility but this led to too complicated proofs so we decided to assume divisibility as the examples that we are primarily interested in all have explicit divisibility anyway.

It would be more natural from the point of view of constructive mathematics to represent more general structures without these decidability conditions. A possible solution to this, using ideas from Homotopy Type Theory [Univalent Foundations Program, 2013], is discussed in the conclusions of the thesis. However, while the use of SSReflect imposes some decidability conditions, we found that in this framework of decidable structures the notations and tactics provided by SSReflect are particularly elegant and well-suited.

The results presented in this paper could be used as a basis for developing a library of formalized computational homological algebra inspired by the Homalg project. In fact `solveMxN` and `solveGeneral` are the only operations used as a basis in Homalg [Barakat and Lange-Hegermann, 2011]. The next paper takes a step in this direction by proving that the category of finitely presented modules over coherent strongly discrete rings form an abelian category.

5

A Coq Formalization of Finitely Presented Modules

Cyril Cohen and Anders Mörtberg

Abstract. This paper presents a formalization, in the intuitionistic type theory of Coq, of constructive module theory. We build an abstraction layer on top of matrix encodings, in order to represent finitely presented modules, and obtain clean definitions with short proofs justifying that it forms an abelian category. The goal is to use it as a first step to get certified programs for computing topological invariants, like homology groups and Betti numbers.

Keywords. Formalization of mathematics, constructive algebra, homological algebra, Coq, SSReflect.

5.1 Introduction

Homological algebra is the study of linear algebra over rings instead of fields, this means that one considers modules instead of vector spaces. Homological techniques are ubiquitous in many branches of mathematics like algebraic topology, algebraic geometry and number theory. Homology was originally introduced by Henri Poincaré in order to compute topological invariants of spaces [Poincaré, 1895], which provides means for testing whether two spaces cannot be continuously deformed into one another. This paper presents a formalization of constructive module theory in type theory, using the Coqproof assistant [Coq Development Team, 2012] together with the Small Scale Reflection (SSReflect) extension [Gonthier et al., 2008], which provides a potential core of a library of certified homological algebra.

Previous work, that the author of thesis was involved in, studied ways to compute homology groups of vector spaces in Coq [Heras et al., 2012, 2013]. When generalizing this to commutative rings the universal coefficient theorem

of homology [Hatcher, 2001] states that most of the homological information of an R-module over a ring R can be computed by only doing computations with elements in $\mathbb{Z}$. This means that if we were only interested in computing homology it would not really be necessary to develop the theory of R-modules in general. We could instead do it for $\mathbb{Z}$-modules which are well behaved because any matrix can be put in Smith normal form. However, by developing the theory for general rings it should be possible to implement and reason about other functors like cohomology, Ext and Tor as in the HOMALG computer algebra package [Barakat and Lange-Hegermann, 2011; Barakat and Robertz, 2008].

In [Gonthier, 2011], it is shown that the theory of finite dimensional vector spaces can be elegantly implemented in COQ by using matrices to represent subspaces and morphisms, as opposed to an axiomatic approach. The reason why abstract finite dimensional linear algebra can be concretely represented by matrices is because any vector space has a basis (a finite set of generators with no relations among the generators) and any morphism can be represented by a matrix in this canonical basis. However, for modules over rings this is no longer true: consider the ideal (X, Y) of $k[X, Y]$, it is a module generated by X and Y which is not free because $XY = YX$. This means that the matrix-based approach cannot be directly applied when formalizing module theory.

To overcome this we restrict our attention to finitely generated modules that are *finitely presented*, that is, modules with a finite set of generators and a finite set of relations among these. In constructive module theory one usually restricts attention to this class of modules and all algorithms can be described by manipulating the presentation matrices [Decker and Lossen, 2006; Greuel and Pfister, 2007; Lombardi and Quitté, 2011; Mines et al., 1988]. This paper can hence be seen as a generalization of the formalization in [Gonthier, 2011] to modules over rings instead over fields.

At the heart of the formalization in [Gonthier, 2011] is an implementation of Gaussian elimination which is used in all subspace constructions. Using it we can compute the kernel which characterizes the space of solutions of a system of linear equations. However when doing module theory over arbitrary rings, there is no general algorithm for solving systems of linear equations. Because of this we restrict our attention further to modules over rings that are *coherent* and *strongly discrete*, as is customary in constructive algebra [Lombardi and Quitté, 2011; Mines et al., 1988], which means that we can solve systems of equations.

The main contributions of this paper are the formalization of finitely presented modules over coherent strongly discrete rings (section 5.2) together with basic operations on these (section 5.3) and the formal proof that the collection of these modules and morphisms forms an abelian category (section 5.4). The fact that they form an abelian category means that they provide a suitable setting for developing homological algebra. We have also proved that, over elementary divisor rings (*i.e.* rings with an algorithm to compute the Smith normal form of matrices), it is possible to test if two finitely presented modules represent isomorphic modules or not (section 5.5). Standard examples of such rings include principal ideal domains, in particular $\mathbb{Z}$ and $k[X]$ where k is a field.

5.2 Finitely presented modules

As mentioned in the introduction, a module is finitely presented if it can be given by a finite set of generators and relations. This is traditionally expressed as:

Definition 1. *An R-module $\mathcal{M}$ is **finitely presented** if there is an exact sequence:*

$$R^{m_1} \xrightarrow{M} R^{m_0} \xrightarrow{\pi} \mathcal{M} \longrightarrow 0$$

Recall that R^m is the type of m-tuples of elements in R. More precisely, π is a surjection and M a matrix representing the m_1 relations among the m_0 generators of the module $\mathcal{M}$. This means that $\mathcal{M}$ is the cokernel of M:

$$\mathcal{M} \simeq \operatorname{coker}(M) = R^{m_0} / \operatorname{im}(M)$$

Hence a module has a finite presentation if it can be expressed as the cokernel of a matrix. As all information about a finitely presented module is contained in its presentation matrix we will omit the surjection π when giving presentations of modules.

Example 1. *The $\mathbb{Z}$-module $\mathbb{Z} \oplus \mathbb{Z}/2\mathbb{Z}$ is given by the presentation:*

$$\mathbb{Z} \xrightarrow{\begin{pmatrix} 0 & 2 \end{pmatrix}} \mathbb{Z}^2 \longrightarrow \mathbb{Z} \oplus \mathbb{Z}/2\mathbb{Z} \longrightarrow 0$$

because if $\mathbb{Z} \oplus \mathbb{Z}/2\mathbb{Z}$ is generated by (e_1, e_2) there is one relation, namely $0e_1 + 2e_2 = 2e_2 = 0$.

Operations on finitely presented modules can now be implemented by manipulating the presentation matrices, for instance if $\mathcal{M}$ and $\mathcal{N}$ are finitely presented R-modules given by presentations:

$$R^{m_1} \xrightarrow{M} R^{m_0} \longrightarrow \mathcal{M} \longrightarrow 0$$

$$R^{n_1} \xrightarrow{N} R^{n_0} \longrightarrow \mathcal{N} \longrightarrow 0$$

the presentation of $\mathcal{M} \oplus \mathcal{N}$ is:

$$R^{m_1+n_1} \xrightarrow{\begin{pmatrix} M & 0 \\ 0 & N \end{pmatrix}} R^{m_0+n_0} \longrightarrow \mathcal{M} \oplus \mathcal{N} \longrightarrow 0$$

We have represented finitely presented modules in Coq using the data structure of matrices from the MathComp library which is defined as:

```
(* 'I_n *)
Inductive ordinal (n : nat) := Ordinal m of m < n.

(* 'M[R]_(m,n) = matrix R m n *)
(* 'rV[R]_m = 'M[R]_(1,m) *)
(* 'cV[R]_m = 'M[R]_(m,1) *)
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.
```

Here `'I_n` is the type `ordinal n` which represents all natural numbers smaller than `n`. This type has exactly `n` inhabitants and can be coerced to the type of natural numbers, `nat`. Matrices are then represented as finite functions over the finite set of indices, which means that dependent types are used to express well-formedness. Finitely presented modules are now conveniently represented using a record containing a matrix and its dimension:

```
Record fpmodule := FPModule {
  nbrel : nat;
  nbgen : nat;
  pres : 'M[R]_(nbrel, nbgen)
}.
```

The direct sum of two finitely presented modules is now straightforward to implement:

```
Definition dsum (M N : fpmodule R) :=
  FPModule (block_mx (pres M) 0 0 (pres N)).
```

Here `block_mx` forms the block matrix consisting of the four submatrices. We now turn our attention to morphisms of finitely presented modules.

5.2.1 Morphisms

As for vector spaces we represent morphisms of finitely presented modules using matrices. The following lemma states how this can be done:

Lemma 4. *If $\mathcal{M}$ and $\mathcal{N}$ are finitely presented R-modules given by presentations:*

$$R^{m_1} \xrightarrow{M} R^{m_0} \longrightarrow \mathcal{M} \longrightarrow 0$$

$$R^{n_1} \xrightarrow{N} R^{n_0} \longrightarrow \mathcal{N} \longrightarrow 0$$

and $\varphi : \mathcal{M} \to \mathcal{N}$ a module morphism then there is a $m_0 \times n_0$ matrix φ_G and a $m_1 \times n_1$ matrix φ_R such that the following diagram commutes:

$$\begin{array}{ccccccc} R^{m_1} & \xrightarrow{M} & R^{m_0} & \longrightarrow & \mathcal{M} & \longrightarrow & 0 \\ \downarrow{\scriptstyle \varphi_R} & & \downarrow{\scriptstyle \varphi_G} & & \downarrow{\scriptstyle \varphi} & & \\ R^{n_1} & \xrightarrow{N} & R^{n_0} & \longrightarrow & \mathcal{N} & \longrightarrow & 0 \end{array}$$

For a proof of this see Lemma 2.1.25 in [Greuel and Pfister, 2007]. This means that morphisms between finitely presented modules can be represented by pairs of matrices. The intuition why two matrices are needed is that the morphism affects both the generators and relations of the modules, hence the names φ_G and φ_R.

In order to be able to compute for example the kernel of a morphism between finitely presented modules we need to add some constraints on the ring R. The reason is that there, in general, is no algorithm for solving systems of equations over arbitrary rings. The class of rings we consider are *coherent* and

strongly discrete which means that it is possible to solve systems of equations. In HOMALG these are called *computable rings* [Barakat and Lange-Hegermann, 2011] and form the basis of the system.

5.2.2 Coherent and strongly discrete rings

Given a ring R (in our setting commutative but it is possible to consider non-commutative rings as well [Barakat and Lange-Hegermann, 2011]) we want to study the problem of solving linear systems over R. If R is a field we have a nice description of the space of solutions by a basis of solutions. Over an arbitrary ring R there is in general no basis. For instance over the ring $k[X,Y,Z]$ where k is a field, the equation $pX + qY + rZ = 0$ has no basis of solutions. It can be shown that a generating system of solutions is given by $(-Y, X, 0)$, $(Z, 0, -X)$, $(0, -Z, Y)$. An important weaker property than having a basis is that there is a finite number of solutions which generate all solutions.

Definition 2. *A ring is **(left) coherent** if for any matrix M it is possible to compute a matrix L such that:*

$$XM = 0 \;\leftrightarrow\; \exists Y.\, X = YL$$

This means that L generates the module of solutions of $XM = 0$, hence L is the kernel of M. For this it is enough to consider the case where M has only one column [Lombardi and Quitté, 2011]. Note that the notion of coherent rings is not stressed in classical presentations of algebra since Noetherian rings are automatically coherent, but in a computationally meaningless way. It is however a fundamental notion, both conceptually [Lombardi and Quitté, 2011; Mines et al., 1988] and computationally [Barakat and Robertz, 2008].

A COQ formalization of coherent rings was presented in the previous paper of the thesis. The only difference (except for name changes) is that in the first presentation composition was read from right to left, whereas here we adopt the SSREFLECT convention that composition is read in diagrammatic order (*i.e.* from left to right).

Recall that in the development, coherent rings have been implemented using the design pattern of [Garillot et al., 2009], using packed classes and the canonical structure mechanism to help COQ automatically infer structures. As matrices are represented using dependent types denoting their size this needs to be known when defining coherent rings. In general the size of L cannot be predicted, so we include an extra function to compute this:

```
Record mixin_of (R : ringType) := Mixin {
  dim_ker : forall m n, 'M[R]_(m,n) -> nat;
  ker : forall m n (M : 'M_(m,n)), 'M_(dim_ker M,m);
  _ : forall m n (M : 'M_(m,n)) (X : 'rV_m),
      reflect (exists Y, X = Y *m ker M) (X *m M == 0)
}.
```

Here `*m` denotes matrix multiplication and `==` is the boolean equality of matrices, so the specification says that this equality is equivalent to the existence statement. An alternative to having a function computing the size would be to output a dependent pair but this has the undesirable behavior that the pair

has to be destructed when stating lemmas about it, which in turn would make these lemmas cumbersome to use as it would not be possible to rewrite with them directly.

An algorithm that can be implemented using `ker` is the kernel modulo a set of relations, that is, computing $\ker(R^m \xrightarrow{M} \operatorname{coker}(N))$. This is equivalent to computing an X such that $\exists Y.\, XM + YN = 0$, which is the same as solving $(X\ Y)(M\ N)^T = 0$ and returning the part of the solution that corresponds to XM. In the paper this is written as $\ker_N(M)$ and in the formalization as `N.-ker(M)`. Note that this is a more fundamental operation than taking the kernel of a matrix as $XM = 0$ is equivalent to $\exists Y.\, X = Y \ker_0(M)$

In order to conveniently represent morphisms we also need to be able to solve systems of the kind $XM = B$ where B is not zero. In order to do this we need to introduce another class of rings that is important in constructive algebra:

Definition 3. *A ring R is **strongly discrete** if membership in finitely generated ideals is decidable and if $x \in (a_1, \ldots, a_n)$ there is an algorithm computing $w_1, \ldots, w_n$ such that $x = \sum_i a_i w_i$.*

Examples of such rings are multivariate polynomial rings over fields with decidable equality (via Gröbner bases) [Cox et al., 2006; Lombardi and Perdry, 1998] and Bézout domains (for instance $\mathbb{Z}$ and $k[X]$ with k a field).

If a ring is both coherent and strongly discrete it is not only possible to solve homogeneous systems $XM = 0$ but also any system $XM = B$ where B is an arbitrary matrix with the same number of columns as M. This operation can be seen as division of matrices as:

```
Lemma dvdmxP m n k (M : 'M[R]_(n,k)) (B : 'M[R]_(m,k)) :
  reflect (exists X, X *m M = B) (M %| B).
```

Here `%|` is notation for the function computing the particular solution to $XM = B$, returning `None` in the case no solution exists. We have developed a library of divisibility of matrices with lemmas like

```
Lemma dvdmxD m n k (M : 'M[R]_(m,n)) (N K : 'M[R]_(k,n)) :
  M %| N -> M %| K -> M %| N + K.
```

which follow directly from `dvdmxP`. This can now be used to represent morphisms of finitely presented modules and the division theory of matrices gives short and elegant proofs about operations on morphisms.

5.2.3 Finitely presented modules over coherent strongly discrete rings

Morphisms between finitely presented R-modules $\mathcal{M}$ and $\mathcal{N}$ can be represented by a pair of matrices. However when R is coherent and strongly discrete it suffices to only consider the φ_G matrix as φ_R can be computed by solving $XN = M\varphi_G$, which is the same as testing $N \mid M\varphi_G$. In Coq this means that morphisms between two finitely presented modules can be implemented as:

```
(* 'Mor(M,N) := morphism_of M N *)
Record morphism_of (M N : fpmodule R) := Morphism {
  matrix_of_morphism : 'M[R]_(nbgen M,nbgen N);
  _ : pres N %| pres M *m matrix_of_morphism
}.
```

Using this representation we can define the identity morphism (`idm`) and composition of morphisms (`phi ** psi`) and show that these form a category. We also define the zero morphism (`0`) between two finitely presented modules, the sum (`phi + psi`) of two morphisms and the negation (`- phi`) of a morphism, respectively given by the zero matrix, the sum and the negation of the underlying matrices. It is straightforward to prove, using the divisibility theory of matrices, that this is a pre-additive category (*i.e.* that the hom-sets form abelian groups).

However, morphisms are not uniquely represented by an element of type `'Mor(M,N)`, but it is possible to test if two morphisms $\varphi\ \psi : M \to N$ are equal by checking if $\varphi - \psi$ is zero modulo the relations of N.

```
(* phi %= psi = eqmor phi psi *)
Definition eqmor (M N : fpmodule R) (phi psi : 'Mor(M,N)) :=
  pres N %| phi%:m - psi%:m.
```

As this is an equivalence relation it would be natural to either use the Coq setoid mechanism [Barthe et al., 2003; Sozeau, 2009] or quotients [Cohen, 2013] in order to avoid applying symmetry, transitivity and compatibility with operators (*e.g.* addition and multiplication) by hand where it would be more natural to use rewriting. We have begun to rewrite the library with quotients as we would get a set of morphisms (instead of a setoid), which is closer to the standard category theoretic notion.

5.3 Monos, epis and operations on morphisms

A monomorphism is a morphism $\varphi : B \to C$ such that whenever there are $\psi_1,\ \psi_2 : A \to B$ with $\psi_1\varphi = \psi_2\varphi$ then $\psi_1 = \psi_2$. When working in pre-additive categories the condition can be simplified to, whenever $\psi\varphi = 0$ then $\psi = 0$.

```
Definition is_mono (M N : fpmodule R) (phi : 'Mor(M,N)) :=
  forall (P : fpmodule R) (psi : 'Mor(P, M)),
    psi ** phi %= 0 -> psi %= 0.
```

It is convenient to think of monomorphisms $B \to C$ as defining B as a subobject of C, so a monomorphism $\varphi : M \to N$ can be thought of as a representation of a submodule M of N. However, submodules are not uniquely represented by monomorphisms even up to equality of morphisms (`%=`). Indeed, multiple monomorphisms with different sources can represent the same submodule. Although "representing the same submodule" is decidable in our theory, we chose not to introduce the notion of submodule, because it is not necessary to develop the theory.

Intuitively monomorphisms correspond to injective morphisms (*i.e.* with zero kernel). The dual notion to monomorphisms are epimorphisms, which intuitively correspond to surjective morphisms (*i.e.* with zero cokernel). For

finitely presented modules, mono- (resp. epi-) morphisms coincide with injective (resp. surjective) morphisms, but this is not clear *a priori*. The goal of this section is to clarify this by defining when a finitely presented module is zero, showing how to define kernels and cokernels, and explicit the correspondence between injective (resp. surjective) morphisms and mono- (resp. epi-) morphisms.

5.3.1 Testing if finitely presented modules are zero

As a finitely presented module is the cokernel of a presentation matrix we have that if the presentation matrix of a module is the identity matrix of dimension $n \times n$ (denoted by I_n) the module is isomorphic to n copies of the zero module. In fact it suffices that the module is presented by a matrix equivalent to a diagonal matrix with only units on the diagonal in order to be isomorphic to the zero module. Now consider the following diagram:

$$\begin{array}{ccccccc}
R^n & \xrightarrow{I_n} & R^n & \longrightarrow & 0^n & \longrightarrow & 0 \\
\downarrow{\scriptstyle X} & & \downarrow{\scriptstyle I_n} & & \downarrow & & \\
R^m & \xrightarrow{M} & R^n & \longrightarrow & \mathcal{M} & \longrightarrow & 0
\end{array}$$

which commutes if $\exists X.\, XM = I_n$, *i.e.* when $M \mid I_n$. Hence this gives a condition that can be tested in order to see if a module is zero or not.

5.3.2 Kernels

In order to compute the kernel of a morphism the key observation is that there is a commutative diagram:

$$\begin{array}{ccccccc}
 & & & & 0 & & \\
 & & & & \downarrow & & \\
R^{k_1} & \xrightarrow{\ker_M(\kappa)} & R^{k_0} & \longrightarrow & \ker(\varphi) & \longrightarrow & 0 \\
\downarrow{\scriptstyle X} & & \downarrow{\scriptstyle \ker_N(\varphi_G)=\kappa} & & \downarrow & & \\
R^{m_1} & \xrightarrow{M} & R^{m_0} & \longrightarrow & \mathcal{M} & \longrightarrow & 0 \\
\downarrow{\scriptstyle \varphi_R} & & \downarrow{\scriptstyle \varphi_G} & & \downarrow{\scriptstyle \varphi} & & \\
R^{n_1} & \xrightarrow{N} & R^{n_0} & \longrightarrow & \mathcal{N} & \longrightarrow & 0
\end{array}$$

It is easy to see that κ is a monomorphism, which means that the kernel is a submodule of $\mathcal{M}$ as expected. In Coq this is easy to define:

```
Definition kernel (M N : fpmodule R) (phi : 'Mor(M,N)) :=
  mor_of_mx ((pres N).-ker phi).
```

Where `mor_of_mx` takes a matrix K with as many columns as N and builds a morphism from $\ker_N(K)$ to M. Using this it is possible to test if a morphism is injective:

```
Definition injm (M N : fpmodule R) (phi : 'Mor(M,N)) :=
  kernel phi %= 0.
```

We have proved that a morphism is injective if and only if it is a monomorphism:

```
Lemma monoP (M N : fpmodule R) (phi : 'Mor(M,N)) :
  reflect (is_mono phi) (injm phi).
```

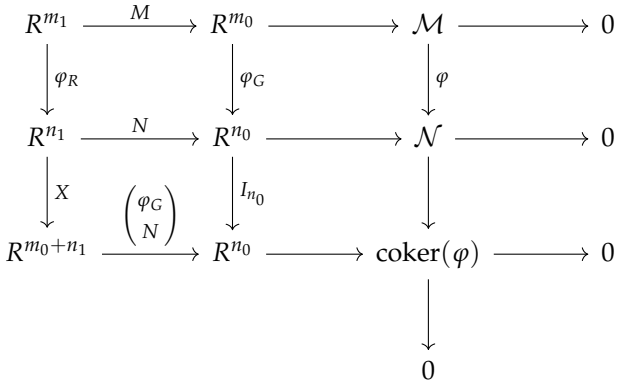
Hence we can define monomorphisms as:

```
(* 'Mono(M,N) = monomorphism_of M N *)
Record monomorphism_of (M N : fpmodule R) := Monomorphism {
  morphism_of_mono :> 'Mor(M, N);
  _ : injm morphism_of_mono
}.
```

The reason why we use `injm` instead of `is_mono` is that `injm` is a boolean predicate, which makes monomorphisms a subtype of morphisms, by Hedberg's theorem [Hedberg, 1998].

5.3.3 Cokernels

The presentation of the cokernel of a morphism can also be found using a commutative diagram:



Note that the canonical surjection onto the cokernel is given by the identity matrix. The fact that this is a morphism is clear as X may be $\begin{pmatrix}0 & I_{n_1}\end{pmatrix}$. However, before defining this we can define the more general operation of quotienting a module by the image of a morphism by stacking matrices:

```
Definition quot_by (M N : fpmodule R) (phi : 'Mor(M, N)) :=
  FPModule (col_mx (pres N) phi)
```

So the cokernel is the canonical surjection from `N` to `quot_by phi`. Since it maps each generator to itself, the underlying matrix is the identity matrix.

```
Definition coker : 'Mor(N, quot_by) :=
  Morphism1 (dvd_quot_mx (dvdmx_refl _)).
```

We can now test if a morphism is surjective by comparing the cokernel of phi with the zero morphism, which coincides with epimorphisms:

```
Definition surjm (M N : fpmodule R) (phi : 'Mor(M,N)) :=
  coker phi %= 0.

Lemma epiP (M N : fpmodule R) (phi : 'Mor(M,N)) :
  reflect (is_epi phi) (surjm phi).
```

As we have algorithms for deciding both if a morphism is injective and surjective we can easily test if it is an isomorphism:

```
Definition isom (M N : fpmodule R) (phi : 'Mor(M,N)) :=
  injm phi && surjm phi.
```

A natural question to ask is if we get an inverse from this notion of isomorphism. In order to show this we have introduced the notion of `isomorphisms` that take two morphisms and express that they are mutual inverse of each other, in the sense that given $\varphi : M \to N$ and $\psi : N \to M$ then $\varphi\psi = 1_M$ modulo the relations in M. Using this we have proved:

```
Lemma isoP (M N : fpmodule R) (phi : 'Mor(M,N)) :
  reflect (exists psi, isomorphisms phi psi) (isom phi).
```

Hence isomorphisms are precisely the morphisms that are both mono and epi. Note that this does not mean that we can decide if two modules are isomorphic, what we can do is testing if a given morphism is an isomorphism or not.

5.3.4 Homology

The homology at $\mathcal{N}$ is defined as the quotient $\ker(\psi)/\operatorname{im}(\varphi)$, in

$$\mathcal{M} \xrightarrow{\varphi} \mathcal{N} \xrightarrow{\psi} \mathcal{K}$$

where $\varphi\psi = 0$. Because of this we have that $\operatorname{im}(\varphi) \subset \ker(\psi)$ so the quotient makes sense and we have an injective map $\iota : \operatorname{im}(\varphi) \to \ker(\psi)$. The homology at $\mathcal{N}$ is the cokernel of this map. We can hence write:

```
Hypothesis mul_phi_psi (M N K : fpmodule R) (phi : 'Mor(M,N))
  (psi : 'Mor(N,K)) : phi ** psi %= 0.

Definition homology (M N K : fpmodule R) (phi : 'Mor(M,N))
  (psi : 'Mor(N,K)) := kernel psi %/ phi.
```

Where `%/` is a notation for taking the quotient of a monomorphism by a morphism with the same target.

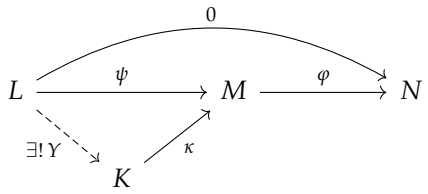
In the next section, we show that these operations satisfy the axioms of abelian categories.

5.4 Abelian categories

As mentioned in the end of section 5.2 the collection of morphisms between two finitely presented modules forms an abelian group. This means that the category of finitely presented modules and their morphisms is a **pre-additive**

category. It is easy to show that the `dsum` construction provides both a product and coproduct. This means that the category is also **additive**.

In order to show that we have a **pre-abelian** category we need to show that morphisms have both a kernel and cokernel in the sense of category theory. A morphism $\varphi : M \to N$ has a kernel $\kappa : K \to M$ if $\kappa\varphi = 0$ and for all $\psi : L \to M$ with $\psi\varphi = 0$ the following diagram commutes:



This means that any morphism ψ with $\psi\varphi = 0$ factors uniquely through the kernel κ. The dual statement for cokernels state that any morphism ψ with $\varphi\psi = 0$ factors uniquely through the cokernel of φ. The specification of the kernel can be written.

```
Definition is_kernel (M N K : fpmodule R) (phi : 'Mor(M,N))
  (k : 'Mor(K,M)) :=
  (k ** phi %= 0) *
  forall L (psi : 'Mor(L,M)),
    reflect (exists Y, Y ** k %= psi) (psi ** phi %= 0).
```

We have proved that our definition of kernel satisfies this specification:

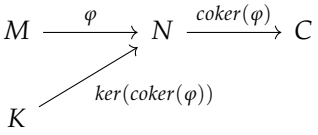
```
Lemma kernelP (M N : fpmodule R) (phi : 'Mor(M,N)) :
  is_kernel phi (kernel phi).
```

We have also proved the dual statement for cokernels. The only properties left in order to have an **abelian category** is that every mono- (resp. epi-) morphism is normal which means that it is the kernel (resp. cokernel) of some morphism. We have shown that if φ is a monomorphism then its cokernel satisfies the specification of kernels:

```
Lemma mono_ker (M N : fpmodule R) (phi : 'Mono(M,N)) :
  is_kernel (coker phi) phi.
```

This means that φ is a kernel of $coker(\varphi)$ if φ is a monomorphism, hence are all monomorphisms normal. We have also proved the dual statement for epimorphisms which means that we indeed have an abelian category.

It is interesting to note that many presentations of abelian categories say that `phi` is `kernel(coker phi)`, but this is not even well-typed as:



One cannot just subtract φ and $\ker(\mathrm{coker}(\varphi))$ as they have different sources. This abuse of language is motivated by the fact that kernels are limits which

are unique up to unique isomorphism which is why many authors speak of *the* kernel of a morphism. However, in order to express this formally we need to exhibit the isomorphism between M and K explicitly and insert it in the equation.

Note that if we introduced a notion of submodule, we could have defined the kernel as a unique submodule of N. The reason is that the type of submodules of N would be the quotient of monomorphisms into N by the equivalence relation which identifies them up to isomorphism.

5.5 Smith normal form

As mentioned before, it is in general not possible to decide if two presentations represent isomorphic modules, even when working over coherent strongly discrete rings. When the underlying ring is a field it is possible to represent a finite dimensional vector space in a canonical way as they are determined up to isomorphism by their dimension (*i.e.* the rank of the underlying matrix) which can be computed by Gaussian elimination. A generalization of this is a class of rings, called *elementary divisor rings* in [Kaplansky, 1949], where any matrix is equivalent to a matrix in Smith normal form. Recall that a matrix M is *equivalent* to a matrix D if there exist invertible matrices P and Q such that $PMQ = D$.

Definition 4. *A matrix is in* ***Smith normal form*** *if it is a diagonal matrix of the form:*

$$\begin{bmatrix} d_1 & & 0 & \cdots & \cdots & 0 \\ & \ddots & & & & \vdots \\ 0 & & d_k & 0 & \cdots & 0 \\ \vdots & & 0 & 0 & & \vdots \\ \vdots & & \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & \cdots & 0 \end{bmatrix}$$

where $d_i \mid d_{i+1}$ for all i.

The connection between elementary divisor rings and finitely presented modules is that the existence of a Smith normal form for the presentation matrix gives us:

$$\begin{array}{ccccccc} R^{m_1} & \xrightarrow{M} & R^{m_0} & \longrightarrow & \mathcal{M} & \longrightarrow & 0 \\ \downarrow{\scriptstyle P^{-1}} & & \downarrow{\scriptstyle Q} & & \downarrow{\scriptstyle \varphi} & & \\ R^{m_1} & \xrightarrow{D} & R^{m_0} & \longrightarrow & \mathcal{D} & \longrightarrow & 0 \end{array}$$

Now φ is an isomorphism as P and Q are invertible. In order to represent this in Coq we need to represent diagonal matrices. For this we use the function `diag_mx_seq`. It is a function that takes two numbers m and n, a list `s` and returns a matrix of type `'M[R]_(m,n)` where the elements of the diagonal are the elements of `s`. It is defined as follows:

```
Definition diag_mx_seq m n (s : seq R) :=
  \matrix_(i < m, j < n) (s`_i *+ (i == j :> nat)).
```

This means that the i:th diagonal element of the matrix is the i:th element of the list and the rest are zero. Now if M is a matrix, our algorithm for computing the Smith normal form should return a list `s` and two matrices P and Q such that:

1. `s` is sorted by division,
2. `P *m M *m Q = diag_mx_seq m n s` and
3. `P` and `Q` are invertible.

Any elementary divisor ring is coherent as the existence of an algorithm computing Smith normal form implies that we can compute kernels. Elementary divisor rings are also Bézout domains which means that they are strongly discrete as well (see section 4.3.3). This means that they provide a suitable instance for the theory of finitely presented modules presented in this paper.

In the next paper these ideas are developed further and we provide instances of elementary divisor rings based on Bézout domains. In particular we give a proof that Bézout domains of Krull dimension less than or equal to 1 (*e.g.* principal ideal domains like $\mathbb{Z}$ and $k[X]$ with k a field) are elementary divisor rings. The reason why we restrict our attention to Krull dimension less than or equal to 1 is that it is still an open problem whether all Bézout domains are elementary divisor rings or not [Lorenzini, 2012].

Combining this with finitely presented modules we get a constructive generalization to the classification theorem of finitely generated modules over principal ideal domains. This theorem states that any finitely generated R-module $\mathcal{M}$ over a principal ideal domain R can be decomposed into a direct sum of a free module and cyclic modules, that is, there exists $n \in \mathbb{N}$ and nonzero elements $d_1, \ldots, d_k \in R$ such that:

$$\mathcal{M} \simeq R^n \oplus R/(d_1) \oplus \cdots \oplus R/(d_k)$$

with the additional property that $d_i \mid d_{i+1}$ for $1 \leqslant i < k$.

The next paper also presents a formal proof that the Smith normal form is unique up to multiplication by units for rings with a gcd operation. This means that for any matrix M equivalent to a diagonal matrix D in Smith normal form, each of the diagonal elements of the Smith normal form of M will be associate to the corresponding diagonal element in D. This implies that the decomposition of finitely presented modules over elementary divisor rings is unique up to multiplication by units. This also gives a way for deciding if two finitely presented modules are isomorphic: compute the Smith normal form of the presentation matrices and then test if they are equivalent up to multiplication by units.

5.6 Conclusions and future work

In this paper we have presented a formalization of the category of finitely presented modules over coherent strongly discrete rings and shown that it is

an abelian category. The fact that we can represent everything using matrices makes is possible for us to reuse basic results on these when building the abstraction layer of modules on top. The division theory of matrices over coherent strongly discrete rings makes it straightforward for us to do reasoning modulo a set of relations.

It is not only interesting that we have an abelian category because it provides us with a setting to do homological algebra, but also because it is proved in [Coquand and Spiwack, 2007] that in order to show that abelian groups (and hence the category of R-modules) form an abelian category in Coq one needs the principle of unique choice. As our formalization is based on the Mathematical Components hierarchy [Garillot et al., 2009] of algebraic structures, we inherit a form of axiom of choice in the structure of discrete rings. However, we speculate that this axiom is in fact not necessary for our proof that the category of **finitely presented** modules over **coherent strongly discrete** rings is abelian.

In Homotopy Type Theory [Univalent Foundations Program, 2013] there is a distinction between pre-categories and univalent categories (just called categories in [Ahrens et al., 2014]). A *pre-category* is a category where the collection of morphisms forms a set in the sense of Homotopy Type Theory, that is, they satisfy the uniqueness of identity proofs principle. Our category of finitely presented modules satisfy the uniqueness of morphism equivalence (`phi %= psi`) proofs (by Hedberg's theorem [Hedberg, 1998]), but morphisms form a setoid instead of a set. If we quotiented morphisms by the equivalence relation on morphisms we would get a set, and thus our category of finitely presented modules would become a pre-category.

A *univalent category* on the other hand is a pre-category where the equality of objects coincides with isomorphism. As we have shown that for elementary divisor rings there is a way to decide isomorphism, we speculate that we would also get a univalent category by quotienting modules by isomorphisms. It would be interesting to develop these ideas further and define the notion of univalent abelian category and study its properties. Note that in Homotopy Type Theory, it may be no longer necessary to have the decidability of the equivalence relation to form the quotient, so we would not need to be in an elementary divisor ring to get a univalent category.

Since we have shown that we have an abelian category it would now be very interesting to formally study more complex constructions from homological algebra. It would for instance be straightforward to define resolutions of modules. We can then define define the *Hom* and tensor functors in order to get derived functors like Tor and Ext. It would also be interesting to define graded objects like chain complexes and graded finitely presented modules, and prove that they also form abelian categories.

Acknowledgments: The authors are grateful to Bassel Mannaa for his comments on early versions of the paper, and to the anonymous reviewers for their helpful comments.

6

Formalized Linear Algebra over Elementary Divisor Rings in Coq

Guillaume Cano, Cyril Cohen, Maxime Dénès, Anders Mörtberg and Vincent Siles

Abstract. This paper presents a Coq formalization of linear algebra over elementary divisor rings, that is, rings where every matrix is equivalent to a matrix in Smith normal form. The main results are the formalization that these rings support essential operations of linear algebra, the classification theorem of finitely presented modules over such rings and the uniqueness of the Smith normal form up to multiplication by units. We present formally verified algorithms computing this normal form on a variety of coefficient structures including Euclidean domains and constructive principal ideal domains. We also study different ways to extend Bézout domains in order to be able to compute the Smith normal form of matrices. The extensions we consider are: adequacy (*i.e.* the existence of a gdco operation), Krull dimension ≤ 1 and well-founded strict divisibility.

Keywords. Formalization of mathematics, constructive algebra, Smith normal form, elementary divisor rings, Coq, SSReflect.

6.1 Introduction

The goal of this paper is to develop linear algebra for *elementary divisor rings*, that is, rings where there is an algorithm for computing the Smith normal form of matrices. The algorithms we present for computing this normal form

can be seen as generalizations of Gaussian elimination that can, in particular, be defined for the ring of integers. The main source of inspiration for this work is the formalization of finite-dimensional vector spaces by Georges Gonthier [Gonthier, 2011] in which spaces are represented using matrices and all subspace constructions can be elegantly defined from Gaussian elimination. This enables a concrete and point-free presentation of linear algebra which is suitable for formalization as it takes advantage of the *small scale reflection* methodology of the SSReflect extension and the Mathematical Components library (MathComp) [Gonthier et al., 2008] for the Coq proof assistant [Coq Development Team, 2012]. When generalizing this to elementary divisor rings there are two essential problems that need to be resolved before the theory may be formalized:

1. What do we get when we generalize finite-dimensional vector spaces by considering more general classes of rings than fields as coefficients?

2. What rings are elementary divisor rings?

An answer to the first problem is finitely generated R-modules, *i.e.* finite-dimensional vector spaces with coefficients in a general ring instead of a field. However these are not as well behaved as finite-dimensional vector spaces as there might be relations among the generators. In other words, not all finitely generated modules are *free*. To overcome this, we restrict our attention further and consider *finitely presented* modules, which are modules specified by a finite set of generators and a finite set of relations between these. This class of modules may be represented concretely using matrices, which in turn means that we can apply the same approach as in [Gonthier, 2011] and implement all operations by manipulating the presentation matrices.

A standard answer to the second problem is *principal ideal domains* like the ring of integers (denoted by $\mathbb{Z}$) and the ring of univariate polynomials over a field (denoted by $k[x]$). The classical definition of principal ideal domains is integral domains where *all* ideals are principal (*i.e.* generated by one element). In particular it means that principal ideal domains are *Noetherian* as all ideals are finitely generated. Classically this is equivalent to the ascending chain condition for ideals, however in order to prove this equivalence classical reasoning is used in essential ways. In fact, if these definitions are read constructively they are so strong that no ring except the trivial ring satisfies them [Perdry, 2004]. Principal ideal domains are hence problematic from a constructive point of view as they are Noetherian.

A possible solution is to restrict the attention to Euclidean domains (which include both $\mathbb{Z}$ and $k[x]$) and show how to compute the Smith normal form of matrices over these rings. This approach is appealing as it allows for a simple definition of the Smith normal form algorithm that resembles the one of Gaussian elimination. While Euclidean domains are important, we would like to be more general. In order to achieve this we consider an alternative approach that is customary in constructive algebra: to generalize all statements and not assume Noetheriannity at all [Lombardi and Quitté, 2011]. If we do this for principal ideal domains we get *Bézout domains*, which are rings where every *finitely generated* ideal is principal. However, it is an open problem whether all Bézout domains are elementary divisor rings or not [Lorenzini, 2012]. Hence

we study different assumptions that we can add to Bézout domains in order to prove that they are elementary divisor rings. The properties we define and study independently are:

1. Adequacy (*i.e.* the existence of a *gdco* operation);
2. Krull dimension ≤ 1;
3. Strict divisibility is well-founded.

The last one can be seen as a constructive approximation to the ascending chain condition for principal ideals, so this kind of Bézout domains will be referred to as *constructive principal ideal domains*.

The main contributions of this paper are the formalization, using the Coq proof assistant with the SSReflect extension, of:

- Rings with explicit divisibility, GCD domains, Bézout domains, constructive principal ideal domains and Euclidean domains (section 6.2);
- An algorithm computing the Smith normal form of matrices with coefficients in Euclidean domains and the generalization to constructive principal ideal domains (section 6.3);
- Linear algebra over elementary divisor rings and the classification theorem for finitely presented modules over elementary divisor rings (section 6.4);
- Proofs that Bézout domains extended with one of the three properties above are elementary divisor rings and how these notions are related (section 6.5).

The paper ends with an overview of related work (section 6.6), followed by conclusions and future work (section 6.7).

6.2 Rings with explicit divisibility

In this section we recall definitions and basic properties of rings with explicit divisibility, GCD domains, Bézout domains, constructive principal ideal domains and Euclidean domains.

6.2.1 Rings with explicit divisibility

Throughout the paper all rings are discrete integral domains, *i.e.* commutative rings with a unit, decidable equality and no zero divisors. This section is loosely based on the presentation of divisibility in discrete domains of Mines, Richman and Ruitenberg in [Mines et al., 1988]. The central notion we consider is:

Definition 5. *A ring R has **explicit divisibility** if it has a divisibility test that produces witnesses.*

That is, given a and b we can test if $a \mid b$ and if this is the case get x such that $b = xa$. Two elements $a, b \in R$ are *associates* if $a \mid b$ and $b \mid a$, which is equivalent to $b = ua$ for some unit u because we have cancellation. Note that this gives rise to an equivalence relation. This notion will play an important role later as we will show that the Smith normal form of a matrix is unique up to multiplication by units, that is, up to associated elements.

A GCD domain is an example of a ring with explicit divisibility:

Definition 6. *A **GCD domain** R is a ring with explicit divisibility in which every pair of elements has a greatest common divisor, that is, for $a, b \in R$ there is* $\gcd(a, b)$ *such that* $\gcd(a, b) \mid a$, $\gcd(a, b) \mid b$ *and* $\forall g$, $(g \mid a) \wedge (g \mid b) \to g \mid \gcd(a, b)$.

Note first that we make no restriction on a and b, so they can both be zero. In this case the greatest common divisor is zero. This makes sense as zero is the maximum element for the divisibility relation. Note also that as R is assumed to be a ring with explicit divisibility we get that $\gcd(a, b) \mid a$ means that there is a' such that $a = a' \gcd(a, b)$. By Euclid's algorithm we know that both $\mathbb{Z}$ and $k[x]$ are GCD domains.

With the above definition the greatest common divisor of two elements is not necessarily unique, *e.g.* the greatest common divisor of 2 and 3 in $\mathbb{Z}$ is either 1 or -1. But if we consider equality up to multiplication by units (*i.e.* up to associatedness) the greatest common divisor is unique, so in the rest of the paper equality will denote equality up to associatedness when talking about the gcd.

Most of the rings we will study in this paper are Bézout domains:

Definition 7. *A **Bézout domain** is a GCD domain R such that for any two elements $a, b \in R$ there is $x, y \in R$ such that $ax + by = \gcd(a, b)$.*

Let a and b be two elements in a ring R. If R is a GCD domain we can compute $g = \gcd(a, b)$ together with witnesses to the ideal inclusion $(a, b) \subseteq (g)$. Further, if R is a Bézout domain we can compute witnesses for the inclusion $(g) \subseteq (a, b)$ as well. This can be generalized to multiple elements $a_1, \ldots, a_n \in R$ to obtain witnesses for the inclusions $(a_1, \ldots, a_n) \subseteq (g)$ and $(g) \subseteq (a_1, \ldots, a_n)$ where g is the greatest common divisor of the a_i. Bézout domains can hence be characterized as rings in which every finitely generated ideal is principal, which means that they are non-Noetherian generalizations of principal ideal domains.

Note that, on the one hand there exists a' and b' such that $a = a' \gcd(a, b)$ and $b = b' \gcd(a, b)$, and on the other hand we have x and y such that $ax + by = \gcd(a, b)$. Therefore, by dividing with $\gcd(a, b)$, we obtain a Bézout relation between a' and b', namely $a'x + a'y = 1$.

This definition can be extended to give a constructive version of principal ideal domains. We say that a divides b **strictly** if $a \mid b$ but $b \nmid a$, using this we can define:

Definition 8. *A **constructive principal ideal domain** is a Bézout domain in which the strict divisibility relation is well-founded.*

By well-founded we mean that any descending chain of strict divisions is finite. This can be seen as a constructive approximation to the ascending chain

condition for principal ideals and hence to Noetheriannity. Both $\mathbb{Z}$ and $k[x]$ can be proved to be Bézout domains and satisfy the condition of constructive principal ideal domains. In fact, this can be done for any ring on which the extended Euclidean algorithm can be implemented. These rings are called Euclidean domains:

Definition 9. *A **Euclidean domain** is a ring R with a Euclidean norm $\mathcal{N} : R \to \mathbb{N}$ such that for any $a \in R$ and nonzero $b \in R$ we have $\mathcal{N}(a) \leqslant \mathcal{N}(ab)$. Further, for any $a \in R$ and nonzero $b \in R$ we can find $q, r \in R$ such that $a = bq + r$ and either $r = 0$ or $\mathcal{N}(r) < \mathcal{N}(b)$.*

In the case of $\mathbb{Z}$ and $k[x]$ we can take respectively the absolute value function and the degree function as Euclidean norm. The standard division algorithms for these rings can then be used to compute q and r.

6.2.2 Formalization of algebraic structures

The algebraic structures have been formalized in the same manner as in the MathComp library [Garillot et al., 2009] using packed classes (implemented by mixins and canonical structures). We will now discuss the formalization of these new structures starting with the definition of rings with explicit divisibility:

```
Inductive div_spec (R : ringType)
                   (a b : R) : option R -> Type :=
  | DivDvd x of a = x * b : div_spec a b (Some x)
  | DivNDvd of (forall x, a != x * b) : div_spec a b None.

Record mixin_of R := Mixin {
  div : R -> R -> option R;
  _ : forall a b, div_spec a b (div a b)
}.
```

This structure is denoted by `DvdRing` and for a ring to be an instance it needs to have a function `div` that returns an `option` type, such that if `div a b = Some x` then x is the witness that $a \mid b$, and if `div a b = None` then $a \nmid b$. The notation used for `div a b` in the formalization is `a %/? b`. There is also a notation for the `div` function that returns a boolean which is written as `a %| b`. This relies on a coercion from `option` to `bool` defined in the SSReflect libraries (mapping `None` to `false` and `Some x` to `true` for any `x`). Using this we have implemented the notion of associatedness, denoted by `%=`, and the basic theory of divisibility.

Next we have the `GCDDomain` structure which is implemented as:

```
Record mixin_of R := Mixin {
  gcdr : R -> R -> R;
  _ : forall d a b, (d %| gcdr a b) = (d %| a) && (d %| b)
}.
```

For a ring to be a `GCDDomain` it needs to have a gcd function, denoted by `gcdr`, satisfying the property above. This property is sufficient as it implies gives that `gcdr a b %| a` and `gcdr a b %| b` since divisibility is reflexive.

The `BezoutDomain` structure looks like:

```
Inductive bezout_spec (R : gcdDomainType)
                      (a b : R) : R * R -> Type :=
  BezoutSpec x y of
    gcdr a b %= x * a + y * b : bezout_spec a b (x,y).

Record mixin_of R := Mixin {
  bezout : R -> R -> R * R;
   _ : forall a b, bezout_spec a b (bezout a b)
}.
```

Recall that a constructive principal ideal domain is a Bézout domain where strict divisibility is well-founded. This is denoted by `PID` and is implemented by:

```
Definition sdvdr (R : dvdRingType) (x y : R) :=
  (x %| y) && ~~(y %| x).

Record mixin_of R := Mixin {
  _ : well_founded (@sdvdr R)
}.
```

The notation `x %<| y` will be used to denote `sdvdr x y`. We will see more precisely in section 6.3.2 how `well_founded` is defined formally in Coq's standard library when we use it to prove the termination of our Smith normal form algorithm.

We also have the `EuclideanDomain` structure:

```
Inductive edivr_spec (R : ringType) (norm : R -> nat)
                     (a b : R) : R * R -> Type :=
  EdivrSpec q r of
    a = q * b + r & (b != 0) ==> (norm r < norm b)
    : edivr_spec norm a b (q,r).

Record mixin_of R := Mixin {
  enorm : R -> nat;
  ediv : R -> R -> R * R;
  _ : forall a b, a != 0 -> enorm b <= enorm (a * b);
  _ : forall a b, edivr_spec enorm a b (ediv a b)
}.
```

This structure contains the Euclidean norm and the Euclidean division function together with their proofs of correctness. We have implemented the extended version of Euclid's algorithm for Euclidean domains and proved that it satisfies `bezout_spec`. Hence we get that Euclidean domains are Bézout domains. We have also proved that any `EuclideanDomain` is a `PID` which means that strict divisibility is well-founded in both $\mathbb{Z}$ and $k[x]$.

The relationship between the algebraic structures presented in this section can be depicted by:

$$\texttt{EuclideanDomain} \subset \texttt{PID} \subset \texttt{BezoutDomain} \subset$$

$$\texttt{GCDDomain} \subset \texttt{DvdRing} \subset \texttt{IntegralDomain}$$

where `IntegralDomain` is already present in the MathComp hierarchy. In the next section we consider an algorithm for computing the Smith normal form of matrices over the first two algebraic structures in the chain of inclusions. This means that these two structures are elementary divisor rings. In section 6.5 we will generalize to Bézout domains of Krull dimension ≤ 1 and adequate domains that fit in between `PID` and `BezoutDomain` in the chain of inclusions.

6.3 A verified algorithm for the Smith normal form

In [Kaplansky, 1949] Kaplansky introduced the notion of **elementary divisor rings** as rings where every matrix is equivalent to a matrix in Smith normal form, that is, given a $m \times n$ matrix M there exist invertible matrices P and Q of size $m \times m$ and $n \times n$ respectively, such that $PMQ = D$ where D is a diagonal matrix of the form:

$$\begin{bmatrix} d_1 & & 0 & \cdots & \cdots & 0 \\ & \ddots & & & & \vdots \\ 0 & & d_k & 0 & \cdots & 0 \\ \vdots & & 0 & 0 & & \vdots \\ \vdots & & \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & \cdots & 0 \end{bmatrix}$$

with the additional property that $d_i \mid d_{i+1}$ for all i.

Let us first explain how we formalized the notion of Smith normal form in Coq, with the following representation of matrices taken from the MathComp library:

```
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.
```

Here `'I_m` is the type of ordinals (*i.e.* natural numbers bounded by `m`) which has exactly `m` inhabitants and can be coerced to `nat`. Matrices are then implemented as finite functions over finite sets of indices, with dependent types being used to ensure well-formedness. We use the notation `'M[R]_(m,n)` for the type `matrix R m n`, the notation `'rV[R]_m` for the type of row vectors of length `m` and the notation `'cV[R]_m` for column vectors of height `m`. The ring `R` is often omitted from these notations when it can be inferred from the context.

In order to express that a matrix is in Smith normal form, we define `diag_mx_seq`, which rebuilds a diagonal matrix from a list (note that the type of lists is called `seq` in the SSReflect library) of diagonal coefficients:

```
Definition diag_mx_seq m n (s : seq R) :=
  \matrix_(i < m, j < n) s`_i *+ (i == j :> nat).
```

The notation `x *+ n`, where `x` belongs to a ring and `n` is a natural number, stands for the sum `x` $+ \ldots +$ `x` iterated `n` times. In the expression of the general coefficients of the matrix above, `i` and `j` are ordinals of type `'I_m` and `'I_n` respectively. The notation `i == j :> nat` tells Coq to compare them as natural numbers and returns a boolean. A coercion then sends this boolean

to a natural number (`true` is interpreted by `1` and `false` by `0`). Thus `s`_i *+ (i == j :> nat)` denotes the element of index `i` in `s` if `i` and `j` have the same value, `0` otherwise.

Now if `M` is a matrix, an algorithm for computing the Smith normal form should return a list `s` and two matrices `P` and `Q` such that:

- The list `s` is sorted for the divisibility relation.
- The matrix `diag_mx_seq m n s` is equivalent to `M`, with transition matrices `P` and `Q`.

Which translates formally to an inductive predicate:

```
Inductive smith_spec R m n (M : 'M[R]_(m,n)) :
  'M[R]_m * seq R * 'M[R]_n -> Type :=
    SmithSpec P s Q of P *m M *m Q = diag_mx_seq m n s
                    & sorted %| s
                    & P \in unitmx
                    & Q \in unitmx : smith_spec M (P,s,Q).
```

We have packaged this in the same manner as above in order to represent elementary divisor rings:

```
Record mixin_of R := Mixin {
  smith : forall m n, 'M[R]_(m,n) -> 'M[R]_m * seq R * 'M[R]_n;
  _ : forall m n (M : 'M[R]_(m,n)), smith_spec M (smith M)
}.
```

In the rest of this section we will see direct proofs that Euclidean domains and constructive principal ideal domains provide instances of this structure.

6.3.1 Smith normal form over Euclidean domains

We mentioned in the introduction that constructive finite dimensional linear algebra over a field can be reduced to matrix encodings. Information like the rank and determinant is then reconstructed from the encoding using Gaussian elimination, which involves three kinds of operations on the matrix:

1. Swapping two rows (resp. columns)
2. Multiplying one row (resp. column) by a nonzero constant
3. Adding to a row (resp. column) the product of another one by a constant

These three operations are interesting because they are compatible with matrix equivalence. In particular, they can be expressed as left (resp. right) multiplication by invertible matrices.

The same algorithm fails to apply in general to a matrix over a ring, since it may require a division by the pivot, which could be not exact. The content of this section can thus be seen as a generalization of Gaussian elimination to Euclidean domains.

To make this extension possible, a new kind of elementary operations needs to be introduced. Let a and b be elements of a Euclidean domain R.

Bézout's identity gives u and v such that $ua + vb = \gamma$ where $\gamma = \gcd(a, b)$. Let us note $a' = \frac{a}{\gamma}$ and $b' = \frac{b}{\gamma}$, these divisions being exact by definition of the gcd. We get the identity: $ua' + vb' = 1$. Consider the following square matrix of size n:

$$E_{\text{Bezout}}(a,b,n,k) = \begin{array}{c} \\ (\text{row } k) \end{array} \begin{array}{c} \overset{\qquad\qquad\qquad\qquad\qquad\quad (\text{col. } k)}{} \\ \begin{bmatrix} u & & & & v & & & \\ & 1 & & & & & & \\ & & \ddots & & & & & \\ & & & 1 & & & & \\ -b' & & & & a' & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix} \end{array}$$

The coefficients not explicitly shown in E_{Bezout} are assumed to be zeros. Note that $\det(E_{\text{Bezout}}(a,b,n,k)) = ua' + vb' = 1$, so in particular the matrix above is invertible. We formalize these matrices as follows:

```
Definition combine_mx (a b c d : R) (m : nat) (k : 'I_m) :=
  let k' := lift 0 k in
  let d := \row_j (a *+ (j == 0) + d *+ (j == k') +
                  ((j != 0) && (j != k'))%:R) in
  diag_mx d + c *: delta_mx k' 0 + b *: delta_mx 0 k'.

Definition Bezout_mx (a b : R) (m : nat) (k : 'I_m) :=
  let: (_,u,v,a1,b1) := egcdr a b in
  combine_mx u v (-b1) a1 k.
```

For an ordinal `i` of type `'I_m`, `lift 0 i` represents the ordinal `1 + i` of type `'I_(1 + m)`. The notation `\row_(j < m) (r j)` corresponds to the row matrix `[r 0, ..., r (m-1)]`, if the dimension can be automatically inferred then we can just write `\row_j (r j)`. If `b` is a boolean, the term `b%:R` reduces to `1` if `b` is true, `0` otherwise. The matrix `diag_mx d` correspond to the diagonal matrix where diagonal coefficients are the coefficients of the row matrix `d`, and `delta_mx i j` is the matrix which has only zeros except at position (i, j), where the coefficient is `1`. Finally, `a *: A` is the matrix `A` multiplied by the scalar `a`. Note that the Bézout identity between `a` and `b` is given by the function `egcdr`, which is exported by the underlying Euclidean ring.

Like other elementary operations, multiplication by $E_{\text{Bezout}}(a,b,n,k)$ on the left corresponds to an operation on the rows:

$$E_{\text{Bezout}}(a,b,n,k) \times \begin{bmatrix} L_1 \\ L_2 \\ \vdots \\ L_{k-1} \\ L_k \\ L_{k+1} \\ \vdots \\ L_n \end{bmatrix} = \begin{bmatrix} uL_1 + vL_k \\ L_2 \\ \vdots \\ L_{k-1} \\ -b'L_1 + a'L_k \\ L_{k+1} \\ \vdots \\ L_n \end{bmatrix}$$

These row operations are described formally by:

```
Definition combine_step (a b c d : R) (m n : nat)
                        (M : 'M[R]_(1+m,1+n)) (k : 'I_m) :=
  let k' := lift 0 k in
  let r0 := a *: row 0 M + b *: row k' M in
  let rk := c *: row 0 M + d *: row k' M in
  \matrix_i (r0 *+ (i == 0) + rk *+ (i == k') +
             row i M *+ ((i != 0) && (i != k'))).

Definition Bezout_step (a b : R) (m n : nat)
                       (M : 'M[R]_(1+m,1+n)) (k : 'I_m) :=
  let: (_,u,v,a1,b1) := egcdr a b in
  combine_step u v (-b1) a1 M k.
```

Here `row i M` represents the i:th row of `M`. A lemma connects these row operations to the corresponding elementary matrices:

```
Lemma Bezout_stepE a b (m n : nat) (M : 'M[R]_(1+m,1+n)) k :
  Bezout_step a b M k = Bezout_mx a b k *m M.
```

Let $M = (a_{i,j})$ be a matrix with coefficients in R. We will now show how to reduce M to its Smith normal form using elementary operations. As for Gaussian elimination, we start by finding a nonzero pivot g in M, which is moved to the upper-left corner (if $M = 0$, M already is in Smith normal form). We search the first column for an element which is not divisible by g. Let us assume that $g \nmid a_{k,1}$, we then multiply the matrix on the left by $E_{\text{Bezout}}(g, a_{k,1}, n, k)$:

$$E_{\text{Bezout}}(g, a_{k,1}, n, k) \times \begin{bmatrix} g & L_1 \\ a_{2,1} & L_2 \\ \vdots & \vdots \\ a_{k,1} & L_k \\ \vdots & \vdots \\ a_{n,1} & L_n \end{bmatrix} = \begin{bmatrix} \gamma & uL_1 + vL_k \\ a_{2,1} & L_2 \\ \vdots & \\ -g'g + a'a_{k,1} & -g'L_1 + a'L_k \\ \vdots & \vdots \\ a_{n,1} & L_n \end{bmatrix}$$

with the Bézout identity $ug + va_{k,1} = \gamma = \gcd(g, a_{k,1})$ and posing as previously $g' = \frac{g}{\gamma}$, we have $a' = \frac{a_{k,1}}{\gamma}$.

By definition of γ, we have: $\gamma \mid -g'g + a'a_{k,1}$. Moreover, all the coefficients in the first column of M which were divisible by g are also by γ. We can therefore repeat this process until we get a matrix whose upper-left coefficient (which we still name g) divides all the coefficients in the first column. Linear combinations on rows can thence lead to a matrix B of the following shape:

$$B = \begin{bmatrix} g & b_{1,2} & \cdots & b_{1,n} \\ g & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ g & b_{m,2} & \cdots & b_{m,n} \end{bmatrix}$$

We then search the indicated submatrix of B for an element that is not divisible by g. If such a coefficient $b_{i,j}$ is found, it is moved to the top by permuting rows 1 and i. Thus g is still the upper-left coefficient[1] and multiplications on the right by E_{Bezout} matrices allow, like previously, to obtain a matrix whose upper-left coefficient divides all the others.

This first step is implemented by the function `improve_pivot_rec`:

```
Fixpoint improve_pivot_rec k {m n} :
  'M[R]_(1 + m) -> 'M[R]_(1 + m, 1 + n) -> 'M[R]_(1 + n) ->
  'M[R]_(1 + m) * 'M[R]_(1 + m, 1 + n) * 'M[R]_(1 + n) :=
  match k with
  | 0 => fun P M Q => (P,M,Q)
  | p.+1 => fun P M Q =>
     let a := M 0 0 in
     if find1 M a is Some i then
       let Mi0 := M (lift 0 i) 0 in
       let P := Bezout_step a Mi0 P i in
       let M := Bezout_step a Mi0 M i in
       improve_pivot_rec p P M Q
     else
     let u := dlsubmx M in let vM := ursubmx M in
     let vP := usubmx P in
     let u' := map_mx (fun x => 1 - odflt 0 (x %/? a)) u in
     let P := col_mx (usubmx P) (u' *m vP + dsubmx P) in
     let M := block_mx a%:M vM
                       (const_mx a) (u' *m vM + drsubmx M) in
     if find2 M a is Some (i,j) then
       let M := xrow 0 i M in let P := xrow 0 i P in
       let a := M 0 0 in
       let M0ij := M 0 (lift 0 j) in
       let Q := (Bezout_step a M0ij Q^T j)^T in
       let M := (Bezout_step a M0ij M^T j)^T in
       improve_pivot_rec p P M Q
     else (P, M, Q)
  end.
```

If A, B, C and D are four matrices (with matching dimensions) then the matrix `block_mx A B C D` is:

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

with submatrices denoted by `A = ulsubmx M`, `B = ursubmx M`, `C = dlsubmx M` and `D = drsubmx M`. Similarly `C = col_mx A B` is a column matrix with submatrices `A = usubmx C` and `B = dsubmx C` (the functions for constructing and destructing row matrices have similar names). The matrix `const_mx a` is the matrix where each coefficient is equal to `a` and `xrow i j M` is the matrix `M` with the rows `i` and `j` exchanged.

[1] This trick has been inspired to the authors by a proof-oriented formalization of a similar algorithm by Georges Gonthier.

The function `improve_pivot_rec` takes as arguments a natural number `k` that represents the number of remaining steps, the original matrix and two current transition matrices. If the number of remaining steps is zero, the matrices are returned unchanged (line 5). If not, the first column is searched for an element that is not divisible by the pivot (function `find1`, line 8). If such an element is found on a row of index `i`, a Bézout step is performed between the first row and the one of index `i`, and the function is called recursively (lines 9 to 12). If, on the contrary, the pivot divides all the elements in the first column, some linear combinations (lines 14 to 19) bring us back to a matrix of the shape of the matrix B seen above. Finally, the remaining lines search the whole matrix for an element that is not divisible by the pivot (function `find2`), perform a Bézout step on the columns if appropriate, and call the function recursively.

We have made several choices when implementing this function. First, the argument `k` bounding the number of steps makes it easy to have a structural recursion (this natural number decreases by 1 at each step). In this usual technique, `k` is often called the fuel of the recursion. The flip side is that in order to call the function, an *a priori* bound on the number of steps has to be provided. It is at this point that the hypothesis we made that R is a Euclidean domain comes in handy: we can take as a bound the Euclidean norm of the upper-left coefficient of the original matrix.

We also chose to abstract over initial transition matrices, which are updated as the process goes on. From a computational standpoint, this approach has two benefits. First, it avoids the need for products by transition matrices, asymptotically more costly than to perform the elementary operations directly. Then, it makes the function `improve_pivot_rec` tail-recursive, which can have a good impact on performance.

The flip side is that it is slightly more difficult to express and manipulate formally the link between the matrices taken as arguments and those returned by the function. Indeed, the specification of this function involves inverses of transition matrices:

```
Inductive improve_pivot_rec_spec m n P M Q :
  'M[R]_(1+m) * 'M[R]_(1+m,1+n) * 'M[R]_(1+n) -> Type :=
  ImprovePivotRecSpec P' M' Q' of
      P^-1 *m M *m Q^-1 = P'^-1 *m M' *m Q'^-1
    & (forall i j, M' 0 0 %| M' i j)
    & (forall i, M' i 0 = M' 0 0)
    & M' 0 0 %| M 0 0
    & P' \in unitmx
    & Q' \in unitmx : improve_pivot_rec_spec P M Q (P',M',Q').
```

The statement above can be read as follows: given three matrices `P`, `M` and `Q`, a triple `(P',M',Q')` satisfies the specification if applying to `M` the inverse of elementary operations represented by the initial transition matrices `P` and `Q` gives the same result as applying the inverses of the transition matrices `P'` and `Q'` to `M'`.

The correctness lemma of the function `improve_pivot_rec` states that for an initial matrix `M` whose upper-left coefficient is nonzero and has a norm smaller than a natural number `k`, and for invertible matrices `P` and `Q`, the triple

returned by `improve_pivot_rec k P M Q` satisfies the specification represented by `improve_pivot_rec_spec`:

```
Lemma improve_pivot_recP k m n
  (P : 'M[R]_(1+m)) (M : 'M[R]_(1+m,1+n)) (Q : 'M[R]_(1+n)) :
    enorm (M 0 0) <= k -> M 0 0 != 0 ->
    P \in unitmx -> Q \in unitmx ->
    improve_pivot_rec_spec P M Q (improve_pivot_rec k P M Q).
```

Initially, we call the function `improve_pivot_rec` with identity transition matrices:

```
Definition improve_pivot k m n (M : 'M[R]_(1 + m, 1 + n)) :=
  improve_pivot_rec k 1%:M M 1%:M.
```

By successive subtractions of the first row from all the others and then by linear combinations of columns, we get a matrix C:

$$C = \left[\begin{array}{c|ccc} g & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & C' & \\ 0 & & & \end{array}\right]$$

where g divides all coefficients of C'.

The global algorithm computing the Smith normal form proceeds as follows: it stores the pivot g obtained after the previous step, then divides all coeficients of C' by g and is applied recursively to the resulting matrix. Let us pose $k = \min(m, n)$. From the pivots $g_1, \ldots, g_k$ obtained, the final output of the algorithm is given by the following sequence $d_1, \ldots, d_k$:

$$d_1, d_2, \ldots, d_k = g_1, g_1 g_2, \ldots, \prod_{i=1}^{k} g_i$$

The Smith normal form of the original matrix is then the following diagonal matrix of size $m \times n$:

$$\begin{bmatrix} d_1 & & & & & & \\ & d_2 & & & & & \\ & & \ddots & & & & \\ & & & d_k & & & \\ & & & & 0 & & \\ & & & & & \ddots & \\ & & & & & & 0 \end{bmatrix}$$

This global procedure is implemented by the function `Smith` :

```
1  Fixpoint Smith m n : 'M[R]_(m,n) ->
2                       'M[R]_(m) * seq R * 'M[R]_(n) :=
3    match m, n with
4    | _.+1, _.+1 => fun M : 'M[R]_(1 + _, 1 + _) =>
5        if find_pivot M is Some (i, j) then
```

```
6       let a := M i j in let M := xrow i 0 (xcol j 0 M) in
7       let: (P,M,Q) := improve_pivot (enorm a) M in
8       let a := M 0 0 in
9       let u := dlsubmx M in let v := ursubmx M in
10      let v' := map_mx (fun x => odflt 0 (x %/? a)) v in
11      let M := drsubmx M - const_mx 1 *m v in
12      let: (P',d,Q') :=
13              Smith (map_mx (fun x => odflt 0 (x %/? a)) M) in
14      (lift0_mx P' *m block_mx 1 0 (- const_mx 1) 1 *m
15        xcol i 0 P,
16       a :: [seq x * a | x <- d],
17       xrow j 0 Q *m block_mx 1 (- v') 0 1 *m lift0_mx Q')
18    else (1%:M, [::], 1%:M)
19  | _, _ => fun M => (1%:M, [::], 1%:M)
20  end.
```

If `M` has type `'M[R]_n` then `lift0_mx M = block_mx 1 0 0 M` which has the type `'M[R]_(1 + n)`. The notation `[seq f x | x <- xs]` is like a list comprehension in Haskell and means `map f xs`.

The function `Smith` takes as argument a matrix and returns a sequence made of the nonzero diagonal coefficients of its Smith form, as well as the corresponding transition matrices. The first step (lines 5 and 6) consists in searching for a nonzero pivot in the whole matrix and moving it in the upper-left position. If no pivot is found, all the coefficients are zero and an empty sequence is therefore returned. Otherwise, the function `improve_pivot` defined previously is called (line 7), then some elementary row operations are performed (lines 9 to 11) to get a matrix of the shape of the matrix C shown above. The bottom-right submatrix is then divided by the pivot and a recursive call is performed (lines 12 and 13). The sequence of coefficients and transition matrices obtained are then updated (lines 14 to 17).

We have stated and proved the following correctness lemma:

```
Lemma SmithP m n (M : 'M[R]_(m,n)) : smith_spec M (Smith M).
```

Using this we have instantiated the structure of elementary divisor rings on Euclidean domains.

6.3.2 Extension to principal ideal domains

We mentioned in section 6.2 that constructive principal ideal domains were Bézout domains with a well-founded divisibility relation. Well-foundedness is defined in Coq's standard library using an accessibility predicate [Nordström, 1988]:

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
  Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x.
```

The idea is that all objects of the inductive type `Acc` have to be built by a finite number of applications of the constructor `Acc_intro`. Hence, for any `a` such that `Acc R a`, all chains (x_n) such that `R` x_{n+1} x_n and $x_0 = a$ have to be finite. Note however that there can be infinitely many elements `x` such that `R x a`.

Using this definition of accessibility, we can now state that a relation over a type `A` is well-founded if all elements in `A` are accessible:

```
Definition well_founded (A : Type) (R : A -> A -> Prop) :=
  forall a, Acc R a.
```

Remember that in the previous section, we used the hypothesis that the ring of coefficients was Euclidean when we computed an *a priori* bound on the number of steps the function `improve_pivot` needed to perform. To extend the algorithm to principal ideal domains, we replace the recursion on this bound with a well-founded induction on the divisibility relation.

```
Fixpoint improve_pivot_rec m n (P : 'M_(1 + m))
           (M : 'M_(1 + m, 1 + n)) (Q : 'M_(1 + n))
           (k : Acc (@sdvdr R) (M 0 0)) :
           'M_(1 + m) * 'M_(1 + m, 1 + n) * 'M_(1 + n) :=
 match k with Acc_intro IHa =>
  if find1P M (M 0 0) is Pick i Hi then
    let Ai0 := M (lift 0 i) 0 in
    let P := Bezout_step (M 0 0) Ai0 P i in
    improve_pivot_rec P Q (IHa _ (sdvd_Bezout_step Hi))
  else
    let u := dlsubmx M in let vM := ursubmx M in
    let vP := usubmx P in
    let u' := map_mx (fun x => 1 - odflt 0 (x %/? M 0 0)) u in
    let P := col_mx (usubmx P) (u' *m vP + dsubmx P) in
    let A := block_mx (M 0 0)%:M vM (const_mx (M 0 0))
                      (u' *m vM + drsubmx M) in
    if find2P A (M 0 0) is Pick (i,j) Hij then
      let A := xrow 0 i A in
      let P := xrow 0 i P in
      let a := A 0 0 in
      let A0j := A 0 (lift 0 j) in
      let Q := (Bezout_step a A0j Q^T j)^T in
      improve_pivot_rec P Q (IHa _ (sdvd_Bezout_step2 Hij))
    else (P, A, Q)
end.
```

The main difference with the `improve_pivot_rec` function defined in section 6.3.1 is that we need to prove that the upper-left element of the matrix on which we make the recursive call is strictly smaller than the one of the original matrix. To build these proofs, we use the functions `find1P` and `find2P` which have more expressive (dependent) types than their counterparts `find1` and `find2` that we used previously. They return not only an element of the matrix given as argument, but also a proof that the pivot does not divide this element.

This proof is then used to show that the upper-left coefficient of the matrix decreases, thanks to the following two lemmas:

```
Lemma sdvd_Bezout_step m n (M : 'M[R]_(1+m,1+n)) (k : 'I_m) :
  ~~ (M 0 0 %| M (lift 0 k) 0) ->
  (Bezout_step (M 0 0) (M (lift 0 k) 0) M k) 0 0 %<| M 0 0.
```

```
Lemma sdvd_Bezout_step2 m n i j u' vM (M : 'M[R]_(1+m,1+n)) :
  let B : 'M[R]_(1 + m, 1 + n) :=
       block_mx (M 0 0)%:M vM (const_mx (M 0 0))
                (u' *m vM + drsubmx M) in
  let C := xrow 0 i B in
  ~~ (M 0 0 %| B i (lift 0 j)) ->
    (Bezout_step (C 0 0) (C 0 (lift 0 j)) C^T j)^T 0 0 %<|
     M 0 0.
```

Now, to define the `improve_pivot` function, we use the hypothesis `sdvdr_wf` that the divisibility relation is well-founded:

```
Definition improve_pivot m n (M : 'M[R]_(1 + m, 1 + n)) :=
  improve_pivot_rec 1 1 (sdvdr_wf (M 0 0)).
```

The function `Smith` of section 6.3.1 is essentially unchanged, the only difference being that we removed the first argument of `improve_pivot` (which was a bound on the number of steps of `improve_pivot_rec`).

We have in this section shown how to compute the Smith normal form on Euclidean domains and more generally on constructive principal ideal domains. In the next section, we will explain how to develop a constructive theory of linear algebra based on the existence of such an algorithm.

6.4 Elementary divisor rings

The goal of this section is to develop some theory about linear algebra over elementary divisor rings and discuss the formalization of the classification theorem for finitely presented modules over these rings.

6.4.1 Linear algebra over elementary divisor rings

One of the key operations in linear algebra is to compute solutions to systems of equations. A suitable algebraic setting for doing so is rings where every finitely generated ideal is finitely presented. These rings are called coherent:

Definition 10. *A ring is **coherent** if for any matrix M it is possible to compute a matrix L such that:*

$$XM = 0 \;\leftrightarrow\; \exists Y.\, X = YL$$

This means that L generates the module of solutions of $XM = 0$, *i.e.* that L generates the kernel of M. The notion of coherent rings is usually not mentioned in classical presentations of algebra since Noetherian rings are automatically coherent, but in a computationally meaningless way. It is however a fundamental notion, both conceptually [Lombardi and Quitté, 2011; Mines et al., 1988] and computationally [Barakat and Lange-Hegermann, 2011; Barakat and Robertz, 2008]. The formalization of coherent rings have already been presented in section 4.2 , so we will not discuss the details of this here. Instead we show that elementary divisor rings are coherent.

Let M be a $m \times n$ matrix with coefficients in an elementary divisor ring. There are invertible matrices P and Q such that $PMQ = D$ where D is a diagonal matrix in Smith normal form. The rank of M, denoted $r(M)$, is the number of nonzero elements of D. The kernel of M can be computed by:

$$\ker(M) = (I_m - I_{r(M)})P$$

where I_m is a $m \times m$ identity matrix and $I_{r(m)}$ is a $m \times m$ partial identity matrix with $r(M)$ ones on the diagonal and then zeros. The idea behind this definition is that:

$$\begin{bmatrix} 0 & & & & & \\ & \ddots & & 0 & & \\ & & 0 & & & \\ & & & 1 & & \\ & 0 & & & \ddots & \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} d_1 & & & & & \\ & \ddots & & 0 & & \\ & & d_k & & & \\ & & & 0 & & \\ & 0 & & & \ddots & \\ & & & & & 0 \end{bmatrix} = 0$$

So $\ker(M)MQ = 0$ and since Q is invertible, we have $\ker(M)M = 0$. We can implement the rank operator and state its correctness by:

```
Definition mxrank m n (M : 'M[R]_(m,n)) :=
  let: (P,d,Q) := smith M in size [seq x <- d | x != 0 ].

Definition kermx m n (M : 'M[R]_(m,n)) : 'M[R]_m :=
  let: (P,d,Q) := smith M in copid_mx (mxrank M) *m P.

Lemma kermxP m n (M : 'M[R]_(m,n)) (X : 'rV[R]_m) :
  reflect (exists Y, X = Y *m kermx M) (X *m M == 0).
```

where `copid_mx` corresponds to the partial identity matrix. The `reflect` statement should be read as: the boolean equality `X *m M == 0` holds if and only if there exists `Y : 'rV[R]_m` such that `X = Y *m kermx M`.

An algorithm computing the cokernel of a matrix can be implemented in a similar fashion. This way we have implemented a small library inspired by the library of matrix algebra for fields of MathComp [Gonthier, 2011], but based on Smith normal form instead of Gaussian elimination.

Another important notion in constructive algebra is strongly discrete rings:

Definition 11. *A ring is **strongly discrete** if membership in finitely generated ideals is decidable and if whenever $x \in (x_1, \ldots, x_n)$, there exists $y_1, \ldots, y_n$ such that $x = \sum_i x_i y_i$.*

If a ring is both coherent and strongly discrete it is not only possible to solve homogeneous systems of equations but also arbitrary systems of the kind $XM = B$ where X is a $m \times n$ matrix, M a $n \times k$ matrix and B a nonzero $m \times k$ matrix.

It is easy to see that Bézout domains are strongly discrete as any finitely generated ideal is principal. To test if $x \in (a_1, \ldots, a_n)$ first compute a principal ideal (g) equivalent to $(a_1, \ldots, a_n)$ and then test if $g \mid x$. If this is the case we may construct the witness and otherwise we know that $x \notin (a_1, \ldots, a_n)$.

It is also straightforward to prove that any elementary divisor ring is a Bézout domain. Given $a, b \in R$ we can compute the Smith normal form of a row matrix containing a and b. This gives us an invertible 1×1 matrix P, an invertible 2×2 matrix Q, and $g \in R$ such that:

$$P \begin{bmatrix} a & b \end{bmatrix} Q = \begin{bmatrix} g & 0 \end{bmatrix}$$

As P and Q are invertible we get that g is the greatest common divisor of a and b. The Bézout coefficients are then found by performing the matrix multiplications on the left-hand side of the equality. Hence, as Bézout domains are strongly discrete (see section 4.3.3), we get that elementary divisor rings are not only coherent but also strongly discrete. Note that this gives an alternative proof that elementary divisor rings are coherent, however the proof presented above using the Smith normal form is more direct as we don't have to go through the intersection of finitely generated ideals.

In section 6.5 we consider extensions to Bézout domains that make them elementary divisor rings and hence form a good setting for doing linear algebra. The next subsection shows that the existence of an algorithm for computing the Smith normal form makes finitely presented modules over elementary divisor rings especially well-behaved.

6.4.2 Finitely presented modules and elementary divisor rings

Recall that a module is said to be finitely presented if it can be described using a finite set of generators and a finite set of relations among these. A convenient way to express this is:

Definition 12. *An R-module $\mathcal{M}$ is **finitely presented** if there is an exact sequence:*

$$R^{m_1} \xrightarrow{M} R^{m_0} \xrightarrow{\pi} \mathcal{M} \longrightarrow 0$$

This means that π is a surjection and M a matrix representing the m_1 relations among the m_0 generators of the module $\mathcal{M}$. Another way to think of $\mathcal{M}$ is as the cokernel of M, that is, $\mathcal{M} \simeq \operatorname{coker}(M) = R^{m_0} / \operatorname{im}(M)$. So a module has a finite presentation if it can be expressed as the cokernel of a matrix. As all information of finitely presented modules is contained in its presentation matrix we get that all algorithms on finitely presented modules can be described by manipulating the presentation matrices [Decker and Lossen, 2006; Greuel and Pfister, 2007; Lombardi and Quitté, 2011].

A morphism φ between finitely presented modules $\mathcal{M}$ and $\mathcal{N}$ given by presentations:

$$R^{m_1} \xrightarrow{M} R^{m_0} \longrightarrow \mathcal{M} \longrightarrow 0 \qquad R^{n_1} \xrightarrow{N} R^{n_0} \longrightarrow \mathcal{N} \longrightarrow 0$$

is represented by a $m_0 \times n_0$ matrix φ_G and a $m_1 \times n_1$ matrix φ_R such that the following diagram commutes:

$$\begin{array}{ccccccc} R^{m_1} & \xrightarrow{M} & R^{m_0} & \longrightarrow & \mathcal{M} & \longrightarrow & 0 \\ \downarrow{\scriptstyle \varphi_R} & & \downarrow{\scriptstyle \varphi_G} & & \downarrow{\scriptstyle \varphi} & & \\ R^{n_1} & \xrightarrow{N} & R^{n_0} & \longrightarrow & \mathcal{N} & \longrightarrow & 0 \end{array}$$

The intuition why two matrices are needed is that the morphism affects both the generators and relations of the modules, hence the names φ_G and φ_R. In this paper we adopt the SSReflect convention that composition is read in diagrammatic order (*i.e.* from left to right) when writing equations obtained from commutative diagrams. This means that the equation related to the above diagram is written $M\varphi_G = \varphi_R N$.

In order for us to be able to compute kernels of morphisms we need to assume that the underlying ring is coherent so that we can solve systems of equations involving the underlying matrices. If the underlying ring is also strongly discrete, it is possible to represent morphisms using only φ_G and a proof that $\exists X.\, XN = M\varphi_G$ as any system of equations of the kind $XM = B$ is solvable. In the previous paper of the thesis the formalization of finitely presented modules over coherent and strongly discrete rings is presented.

It is in general not possible to decide if two finitely presented modules are isomorphic or not. However, if the underlying ring is an elementary divisor ring, it becomes possible. Indeed, let R be an elementary divisor ring and M be a $m_1 \times m_0$ matrix presenting an R-module $\mathcal{M}$. As M is equivalent to a diagonal matrix D, there are invertible matrices P and Q such that $MQ = P^{-1}D$. This gives a commutative diagram:

$$\begin{array}{ccccccc} R^{m_1} & \xrightarrow{M} & R^{m_0} & \longrightarrow & \mathcal{M} & \longrightarrow & 0 \\ \downarrow{\scriptstyle P^{-1}} & & \downarrow{\scriptstyle Q} & & \downarrow{\scriptstyle \varphi} & & \\ R^{m_1} & \xrightarrow{D} & R^{m_0} & \longrightarrow & \mathcal{D} & \longrightarrow & 0 \end{array}$$

We can further prove that φ is an isomorphism as P and Q are invertible, and hence get that $\mathcal{M} \simeq \mathcal{D} \simeq \operatorname{coker}(D)$. Now, since D is a diagonal matrix with nonzero elements $d_1, \ldots, d_n \in R$ on the diagonal, we get that:

$$\mathcal{M} \simeq R^{m_0 - n} \oplus R/(d_1) \oplus \cdots \oplus R/(d_n) \tag{6.1}$$

with the additional property that $d_i \mid d_{i+1}$ for all $1 \leqslant i < n$. Note that if d_i is a unit then $R/(d_i) \simeq 0$. This means that the theory of finitely presented modules over elementary divisor rings R is particularly well-behaved as any finitely presented R-module $\mathcal{M}$ can be decomposed into a direct sum of a free module and cyclic modules. This is the first part of the classification theorem for finitely presented modules over elementary divisor rings, the second part is the fact that the d_i are unique up to multiplication by units which makes the decomposition unique.

The uniqueness part is also necessary in order to get a decision procedure for the isomorphism of finitely presented modules over elementary divisor rings. So far we only know that any module may be decomposed as above, but there is, *a priori*, no reason why two isomorphic modules should have related decompositions.

In the next section we will see that the Smith normal form is unique up to multiplication by units if the underlying ring has a gcd operation, which in turn completes the classification theorem and gives us a decision procedure for module isomorphism.

6.4.3 Uniqueness of the Smith normal form

The formal proof that the Smith normal form is unique up to multiplication by units presented here is based on [Cano and Dénès, 2013]. In order to formalize this proof we need to represent minors (determinants of submatrices) in Coq. This notion was defined in section 3.4, but we recall it here again:

```
Definition submatrix m n p q (f : 'I_p -> 'I_m)
  (g : 'I_q -> 'I_n) (M : 'M[R]_(m,n)) : 'M[R]_(p,q) :=
    \matrix_(i < p, j < q) M (f i) (g j).

Definition minor m n p (f : 'I_p -> 'I_m) (g : 'I_p -> 'I_n)
  (M : 'M[R]_(m,n)) : R := \det (submatrix f g M).
```

For example, the rows (resp. columns) of the submatrix $M(f,g)$ are the rows (resp. columns) $f(0), f(1), ...$ (resp. $g(0), g(1), ...$) of M. It would be natural to define submatrices only when `f` and `g` are strictly increasing, however this is not necessary as many theorems are true for arbitrary functions. We denote `p` in the definition of `minor` above as the order of the minor, that is, a minor of order p is the determinant of a submatrix of dimension $p \times p$.

The key result in order to prove the uniqueness theorem for the Smith normal form is that the product of the k first elements of the diagonal in the Smith normal form is associated to the gcd of the minors of order k of the original matrix. More precisely, let M be the original matrix and d_i the i:th element of the diagonal in the Smith normal form of M, also let $\overrightarrow{m}_k$ be the minors of order k of M, then the statement is:

$$\prod_{i=1}^{k} d_i = \gcd(\overrightarrow{m}_k)$$

Using the big operators library of MathComp [Bertot et al., 2008] this can be expressed compactly as:

```
Lemma Smith_gcdr_spec :
  \prod_(i < k) d`_i %=
  \big[gcdr/0]_f \big[gcdr/0]_g minor f g M.
```

The order of the minors that we consider are given by the types of `f` and `g`. For the sake of readability, we have omitted these types.

The first step in proving this is by showing that it holds for the Smith normal form of M, namely the diagonal matrix D. Since it is a diagonal matrix, the only nonzero minors of order k are the determinants of diagonal matrices of dimension $k \times k$, that are products of k elements of the diagonal of D. Also, since each element of the diagonal divides the next one, the greatest common divisor of the minors of order k is the product of the k first elements of the diagonal. For example, if the diagonal is (a, b, c) with $a \mid b$ and $b \mid c$ then $\gcd(ab, bc, ac) = ab$.

The next step is to prove that the gcd of the minors of order k of M are associated to the gcd of the minors of D (which we already know is associated to the product of the elements on the diagonal). To prove this it suffices to show that these two divide each other, as the proofs in both directions are

very similar we only show that the gcd of the minors of order k of M divides the gcd of the minors of order k of D.

By definition, x divides $\gcd(\overrightarrow{y})$ if and only if x divides every y in $\overrightarrow{y}$. So we must show that the gcd of the minors of order k of M divides each minor of order k of the diagonal matrix D. Now, there are invertible matrices P and Q such that $PMQ = D$. Hence we must show that $\gcd(\overrightarrow{m}_k)$ divides $\det((PMQ)(f,g))$ for all f and g. The right-hand side is the determinant of a product of matrices of different sizes whose product is square, which can be simplified with the Binet-Cauchy formula:

$$\det(MN) = \sum_{\substack{I \in \mathcal{P}(\{1,\ldots,l\}) \\ \#|I|=k}} \det(M_I)\det(N_I)$$

where M is a $k \times l$ matrix and N is a $l \times k$ matrix. M_I (resp. N_I) is the matrix of the k columns (resp. rows) with indices in I.

The formalization of this formula builds on the work in the third paper and follows the proof presented in [Zeng, 1993]. Note that the standard determinant identity for products of square matrices of the same size follows as a special case of the above formula. Once again the theorem can be expressed compactly using the big operators of SSReflect:

```
Lemma BinetCauchy :
  \det (M *m N) = \sum_(f : {ffun 'I_k -> 'I_l} | strictf f)
                        ((minor id f M) * (minor f id N)).
```

Here the sum is taken over all strictly increasing functions from $\{1,\ldots,k\}$ to $\{1,\ldots,l\}$. We require the functions to be strictly increasing so that the minors that we consider in the sum correspond to the mathematical concept of minor.

This theorem makes enables us to transform $\det((PMQ)(f,g))$ to a sum of minors and, once again, it suffices to show that $\gcd(\overrightarrow{m}_k)$ divides each of the summands. Hence, after some simplifications, we must show that for all `h` and `i` we have:

```
\big[gcdr/0]_f \big[gcdr/0]_g minor f g M %| minor h i M
```

which is true by definition of the gcd. Note that it is not necessary to require that `f` and `g` are strictly increasing. Indeed, if they are not, there are two cases:

- Either `f` or `g` is not injective and so `minor f g M = 0`.
- If both `f` and `g` are injective there exist permutations `r` and `s` such that `f' = f \o r` and `g' = g \o s` are strictly increasing. As the permutation of rows or columns of a matrix just leads to the determinant being multiplied by the signature of the permutation we get `minor f g M %= minor f' g' M`.

But for all a we have $\gcd(a,0) = a$ and $\gcd(a,a) = a$, so in each case the terms corresponding to the minors obtained from not strictly increasing `f` and `g` does not change the value of the gcd of the minors.

Now if the above result is applied with $k = 1$, the uniqueness of the first diagonal element is proved, and then by induction all of the diagonal elements

are showed to be unique (up to multiplication by units). This means that for any matrix M equivalent to a diagonal matrix D in Smith normal form, each of the diagonal elements of the Smith normal form of M will be associate to the corresponding diagonal element in D. The uniqueness of the Smith normal form is expressed formally as follows:

```
Lemma Smith_unicity m n (M : 'M[R]_(m,n)) (d : seq R) :
  sorted %| d -> equivalent M (diag_mx_seq m n d) ->
  forall i, i < minn m n -> (smith_seq M)`_i %= d`_i.
```

Hence we have proved that the Smith normal form is unique up to multiplication by units. This gives a test to know if two matrices are equivalent. Indeed, since the Smith normal form of a matrix is equivalent to it, two matrices are equivalent if and only if they have the same normal form. Moreover, we know that the decomposition in equation (6.1) is unique up to multiplication by units. Hence we get am algorithm for deciding if two finitely presented modules are isomorphic or not: compute the Smith normal form of the presentation matrices and then test if they are equivalent up to multiplication by units.

This concludes the classification theorem for finitely presented modules over elementary divisor rings. It can be seen as a constructive version of the classification theorem for finitely generated modules over principal ideal domains. Classical proofs of this use the fact that a principal ideal domain R is Noetherian which implies that any R-module is coherent, *i.e.* that any finitely generated module is also finitely presented. But this proof has no computational content (see exercise 3 in chapter III.2 of [Mines et al., 1988]), so instead we have to restrict to finitely presented modules. In section 6.3.2 we showed that constructive principal ideal domains are elementary divisor rings which gives us the classical result in the case of finitely presented modules. In the next section we will prove that more general classes of rings than principal ideal domains are elementary divisor rings which gives more instances of the classification theorem.

6.5 Extensions to Bézout domains that are elementary divisor rings

As mentioned in the introduction, it is an open problem whether all Bézout domains are elementary divisor rings or not. In order to overcome this, we study different properties that we can extend Bézout domains with to make them elementary divisor rings. The properties we define and discuss in this section are:

1. Adequacy (*i.e.* the existence of a gdco operation);
2. Krull dimension ≤ 1;
3. Strict divisibility is well-founded (constructive principal ideal domains).

We have already considered the last one of these in section 6.3.2, but here we formalize an alternative proof that constructive principal ideal domains

are elementary divisor rings, using a reduction due to Kaplansky [Kaplansky, 1949]. It consists in first simplifying the problem of computing Smith normal form for $m \times n$ matrices to the 2×2 case and then showing that any 2×2 matrix over a Bézout domain R has a Smith normal form if and only if R satisfies the "Kaplansky condition". This means that it suffices for us to prove that the three different extensions all imply this condition in order to show that they are elementary divisor rings.

6.5.1 The Kaplansky condition

The reduction of the computation of Smith normal form of arbitrary $m \times n$ matrices to 2×2 matrices is done by extracting an algorithm from the proof of theorem 5.1 in [Kaplansky, 1949]. The formalization is done by first implementing this algorithm, called `smithmxn`, computing the Smith normal form of arbitrary sized matrices assuming an operation computing it for 2×2 matrices and then proving that this algorithm satisfies `smith_spec`:

```
Lemma smithmxnP :
  forall (smith2x2 : 'M[R]_2 -> 'M[R]_2 * seq R * 'M[R]_2),
  (forall (M : 'M[R]_2), smith_spec M (smith2x2 M)) ->
  forall m n (M : 'M[R]_(m,n)),
    smith_spec M (smithmxn smith2x2 M).
```

This algorithm has no assumptions on the underlying ring except that it is an integral domain. It can be generalized to arbitrary commutative rings but then we also need to be able to put 1×2 and 2×1 matrices in Smith normal form.

In order to explain the reduction to the Kaplansky condition consider a 2×2 matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

with coefficients in a Bézout domain. We can compute $g = \gcd(a, c)$ and a' and c' such that $a = a'g$ and $c = c'g$. We also have u and v such that $ua' + vc' = 1$. Using this we can form:

$$\begin{aligned} \begin{bmatrix} u & v \\ -c' & a' \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} &= \begin{bmatrix} ua + vc & ub + vd \\ -c'a + a'c & -c'b + a'd \end{bmatrix} \\ &= \begin{bmatrix} ua + vc & ub + vd \\ 0 & -c'b + a'd \end{bmatrix} \end{aligned}$$

So it suffices to consider matrices of the following shape:

$$\begin{bmatrix} a & b \\ 0 & c \end{bmatrix}$$

and without loss of generality we can assume that $\gcd(a, b, c) = 1$. Now, such a matrix has a Smith normal form if and only if it satisfies the **Kaplansky condition**: for all $a, b, c \in R$ with $\gcd(a, b, c) = 1$ there exist $p, q \in R$ with $\gcd(pa, pb + qc) = 1$.

The interesting step for the reduction is the right to left direction of the "if and only if", so let us sketch how it is proved: assume that R is a Bézout domain that satisfies the Kaplansky condition and consider an upper triangular matrix with elements a, b and c with $\gcd(a,b,c) = 1$. From the Kaplansky condition we get p and q such that $\gcd(pa, pb+qc) = 1$. This means that we also have x_1 and y_1 such that $pax_1 + (pb+qc)y_1 = 1$. By reorganizing this we get $p(ax_1 + by_1) + qcy_1 = 1$, let $x = ax_1 + by_1$ and $y = cy_1$. We can form the product:

$$\begin{bmatrix} p & q \\ -y & x \end{bmatrix} \begin{bmatrix} a & b \\ 0 & c \end{bmatrix} \begin{bmatrix} x_1 & pb+qc \\ y_1 & -pa \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -ac \end{bmatrix}$$

In order to formalize this proof we assume that we have an operation taking a, b and c computing p and q satisfying the Kaplansky condition:

```
Variable kap : R -> R -> R -> R * R.

Hypothesis kapP : forall (a b c : R), gcdr a (gcdr b c) %= 1 ->
  let: (p,q) := kap a b c in coprimer (p * a) (p * b + q * c).
```

We then define a function `kapW : R -> R -> R -> R * R` to extract the two witnesses x_1 and y_1 from above, *i.e.* x_1 and y_1 such that $x_1pa + y_1(pb+qc) = 1$. To do this we first prove:

```
Lemma coprimerP (a b : R) :
  reflect (exists (xy : R * R), xy.1 * a + xy.2 * b = 1)
          (coprimer a b).
```

and we can then define a function computing (x_1, y_1) by turning the existential statement in `coprimerP` into a Σ-type (*i.e.* a dependent pair). More precisely, we have defined it by:

```
Definition kapW a b c : R * R :=
  let: (p,q) := kap a b c in
  if coprimerP (p * a) (p * b + q * c) is ReflectT P
     then projT1 (sig_eqW P) else (0,0).
```

Here `sig_eqW` is a function from the SSReflect library that transforms our existential statement into a Σ-type, the first component of the resulting Σ-type is then extracted using `projT1`. This is possible because `R` is taken to be an SSReflect "choice type", *i.e.* a type with a choice operator.

Once we have defined `kapW`, we can easily write the function computing Smith normal form of 2×2 matrices, called `kap_smith`, and prove that it satisfies `smith_spec`:

```
Definition kap_smith (M : 'M_2) : 'M[R]_2 * seq R * 'M[R]_2 :=
  let A := Bezout_step (M 0 0) (M 1 0) M 0 in
  let a00 := A 0 0 in let a01 := A 0 1 in let a11 := A 1 1 in
  let: (d,_,_,_,a,b,c) := egcdr3 a00 a01 a11 in
  if d == 0 then (Bezout_mx (M 0 0) (M 1 0) 0,[::],1%:M) else
    let: (p,q) := kap a b c in
    let: (x1,y1) := kapW a b c in
    let: (x,y) := (a * x1 + y1 * b, c * y1) in
     (mx2 p q (- y) x *m Bezout_mx (M 0 0) (M 1 0) 0,
```

```
    map (fun x => d * x) [:: 1; - a * c],
    mx2 x1 (p * b + q * c) y1 (- p * a)).

Lemma kap_smithP (M : 'M[R]_2) : smith_spec M (kap_smith M).
```

Here `mx2` is a notation to define 2×2 matrices and `egcdr3` computes the Bézout coefficients for 3 elements.

We have also formalized the other direction, so for a Bézout domain, satisfying the Kaplansky condition is equivalent to being an elementary divisor ring. Hence it suffices to prove that the various extensions to Bézout domains satisfy the Kaplansky condition in order to get that they are elementary divisor rings.

6.5.2 The three extensions to Bézout domains

In this section we discuss three extensions to Bézout domains that imply the Kaplansky condition.

Adequate domains

In [Helmer, 1943] the notion of **adequate domains** is introduced[2]. This notion extends Bézout domains by assuming that where for any $a, b \in R$, with $b \neq 0$, exists $r \in R$ such that:

1. $r \mid b$,
2. r is coprime with a, and
3. for all non unit d such that $dr \mid b$ we have that d is not coprime with a.

We have proved that this notion is equivalent to having a "gdco" function. This function has previously been introduced in [Cohen and Mahboubi, 2010] to implement quantifier elimination for algebraically closed fields. This operation has also many other applications in algebra, see [Lüneburg, 1986]. It takes two elements $a, b \in R$, with $b \neq 0$, and computes r such that:

1. $r \mid b$,
2. r is coprime with a , and
3. for all divisors d of b that is coprime to a we have $d \mid r$.

This means that r is the greatest divisor of b that is coprime to a. These notions are expressed in Coq as:

```
Inductive adequate_spec (a b : R) : R -> Type :=
  | AdequateSpec0 of b = 0 : adequate_spec a b 0
  | AdequateSpec r of
      b != 0
    & r %| b
    & coprimer r a
```

[2]Interestingly Helmer refers to what is here called Bézout domains as "Prüfer rings".

```
    & (forall d, d * r %| b -> d \isn't a unit ->
                ~~ coprimer d a)
      : adequate_spec a b r.

Inductive gdco_spec (a b : R) : R -> Type :=
  | GdcoSpec0 of b = 0 : gdco_spec a b 0
  | GdcoSpec r of
      b != 0
    & r %| b
    & coprimer r a
    & (forall d, d %| b -> coprimer d a -> d %| r)
      : gdco_spec a b r.

Lemma adequate_gdco a b r :
  adequate_spec a b r -> gdco_spec a b r.

Lemma gdco_adequate a b r :
  gdco_spec a b r -> adequate_spec a b r.
```

We have implemented an algorithm called gdco_kap that computes p and q in the Kaplansky condition using the gdco operation. Using this we have proved:

```
Lemma gdco_kapP (a b c : R) : gcdr a (gcdr b c) %= 1 ->
  let: (p,q) := gdco_kap a b c
  in coprimer (p * a) (p * b + q * c).
```

Using this we can define a function that computes the Smith normal form for any matrix over an adequate domain:

```
Definition gdco_smith := smithmxn (kap_smith gdco_kap).

Lemma gdco_smithP : forall m n (M : 'M[R]_(m,n)),
                      smith_spec M (gdco_smith M).
```

Hence we get that adequate domains are elementary divisor rings.

Krull dimension ≤ 1

The next class of rings we study are Bézout domains of Krull dimension ≤ 1. Classically Krull dimension is defined as the supremum of the length of all chains of prime ideals, this means that a ring has Krull dimension $n \in \mathbb{N}$ if there is a chain of prime ideals:

$$\mathfrak{p}_0 \subsetneq \mathfrak{p}_1 \subsetneq \cdots \subsetneq \mathfrak{p}_n$$

but no such chain of length $n+1$. For example, a field has Krull dimension 0 and any principal ideal domain (that is not a field) has Krull dimension 1. This can be defined constructively using an inductive definition as in [Lombardi and Quitté, 2011]. Concretely an integral domain R is of Krull dimension ≤ 1 if for any $a, u \in R$ there exists $n \in \mathbb{N}$ and $v \in R$ such that

$$a \mid u^n(1 - uv)$$

In order to prove that Bézout domains of Krull dimension ≤ 1 are adequate we first prove:

```
Hypothesis krull1 : forall a u,
                      exists n v, a %| u ^+ n * (1 - u * v).

Lemma krull1_factor a b : exists n b1 b2,
  [&& 0 < n, b == b1 * b2, coprimer b1 a & b2 %| a ^+ n].
```

This means that given a and b we can compute $n \in \mathbb{N}$ and $b_1, b_2 \in R$ such that $n \neq 0$, $b = b_1 b_2$, b_1 is coprime with a and $b_2 \mid a^n$. If we set r to b_1 in the definition of adequate domains we have to prove:

1. $b_1 \mid b_1 b_2$,
2. b_1 is coprime with a, and
3. for all d that are not units such that $db_1 \mid b_1 b_2$ we have that d is not coprime with a.

The first two are obvious. For the third point, we have to prove that any non-unit d that divides b_2 is not coprime with a. So it suffices to prove that any d coprime with a that divides b_2 is a unit. Now as $n \neq 0$ we get that d is coprime with a^n, but $d \mid b_2$ and $b_2 \mid a^n$ so d must be a unit. We have formalized this argument in:

```
Lemma krull1_adequate a b : { r : R & adequate_spec a b r }.
```

This means that Bézout domains of Krull dimension ≤ 1 are adequate and hence satisfy the Kaplansky condition, which in turn means that they are elementary divisor rings:

```
Definition krull1_gdco a b := projT1 (krull1_adequate a b).

Definition krull1_smith := gdco_smith krull1_gdco.

Lemma krull1_smithP : forall m n (M : 'M[R]_(m,n)),
                        smith_spec M (krull1_smith M).
```

Constructive principal ideal domains

Finally, we have showed that constructive principal ideal domains are adequate domains by proving that given a and b we can compute r satisfying `gdco_spec`:

```
Lemma pid_gdco (R : pidType) (a b : R) :
  {r : R & gdco_spec a b r}.
```

The construction of the greatest divisor of a coprime to b in a constructive principal ideal domain is done as in the particular case of polynomials in [Cohen and Mahboubi, 2010]. If $\gcd(a, b)$ is a unit, then a is trivially the result, otherwise we get a' by dividing a by $\gcd(a, b)$ and we repeat the process with a' and b. This process terminates because when $\gcd(a, b)$ is not a unit, a' strictly

divides a and by our definition of constructive principal ideal domains, there cannot be an infinite decreasing sequence for strict divisibility.

This way we get an alternative proof that constructive principal ideal domains are elementary divisor rings:

```
Definition pid_smith :=
  gdco_smith (fun a b => projT1 (pid_gdco a b)).

Lemma pid_smithP m n (M : 'M[R]_(m,n)) :
  smith_spec M (pid_smith M).
```

This proof is simpler than the one presented in section 6.3.2 in the sense that we first reduce the problem of computing the Smith normal form to computing the gdco of two elements. This way, the part of the proof based on well-founded recursion is concentrated to `pid_gdco` instead of being interleaved in the algorithm computing the Smith normal form of arbitrary $m \times n$ matrices.

6.6 Related work

Most proof assistants have one or more libraries of formalized linear algebra. However, the specificity of our work is that it is more general than the usual study of vector spaces (we do not require scalars to be in a field, but only in an elementary divisor ring) while still retaining an algorithmic basis, as opposed to a purely abstract and axiomatized development. In particular, this work constitutes to our knowledge the first formal verification of an algorithm for the Smith normal form of matrices.

A fair amount of module theory and linear algebra has been formalized in Mizar [Rudnicki et al., 2001]. But it is based on classical logic and does not account for underlying algorithmic aspects. Likewise, a HOL Light library [Harrison, 2013] proves significant results in linear algebra and on the topology of vector spaces, but it is specialized to $\mathbb{R}^n$ and also classical.

Some other developments focus more on the algebra of vectors and matrices, without providing support for point-free reasoning on subspaces. Let us cite [Obua, 2005] in Isabelle, which aims primarily to certify linear inequalities and [Gamboa et al., 2003; Hendrix, 2003] in ACL2, formalizing only matrix algebra.

In Coq too, older developments focus on the representation of matrices [Magaud, 2005], or classical linear algebra over a field [Stein, 2001], based on [Pottier, 1999]. One exception is of course the more recent work [Gonthier, 2011] that we already mentioned and on which we based this work, extending it from finite-dimensional vector spaces to finitely presented modules over elementary divisor rings.

6.7 Conclusions and future work

The relationships between the notions introduced in this paper are depicted in Figure 6.1. The numbers on the edges denote the sections in which the different implications and inclusions are proved:

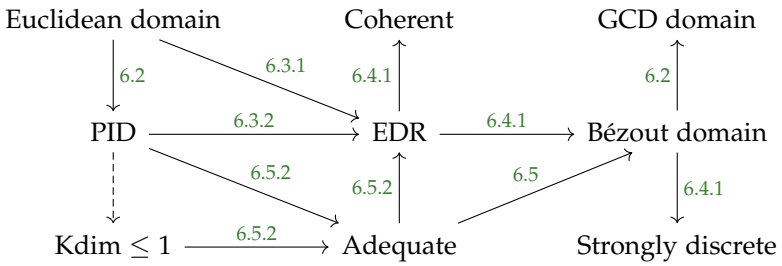


Figure 6.1: Relationship between the defined notions

The arrow between PID and Kdim ≤ 1 is dashed because it has not been formally proved yet. A constructive proof of this can be found in [Lombardi and Quitté, 2011]. We currently see two options to formalize it: either we try to develop more extensively the theory of ideals to stick close to the paper proof, or we expand statements on ideals to statements on elements. Unlike the former, the latter option would require no further infrastructure, but it is likely that the size of the proof would explode, as in some proofs where we already had to talk about elements instead of ideals (*e.g.* the lemma `krull1_factor` in the current state of the formalization).

It has been mentioned that $\mathbb{Z}$ and $k[x]$ where k is a field are the basic examples for all of these rings. Many more examples of Bézout domains are presented in the chapters on Bézout domains and elementary divisor rings in [Fuchs and Salce, 2001] (for instance, Bézout domains of arbitrary finite Krull dimension and an example of a Bézout domain that is not adequate). It would be interesting see which of these could be done in a constructive setting and formalize them in order to get more instances than $\mathbb{Z}$ and $k[x]$.

An important application of this work is to compute the homology of chain complexes which provides a means to study properties of mathematical objects like topological spaces. By computing homology one associates modules to these kinds of objects, giving a way to distinguish between them. The Smith normal form of matrices with coefficients in $\mathbb{Z}$ is at the heart of the computation of homology as the universal coefficient theorem for homology [Hatcher, 2001] states that homology with coefficients in $\mathbb{Z}$ determines homology with coefficients in any other abelian group.

Note that the Kaplansky condition in section 6.5 is expressed using first-order logic. It means that the open problem whether all Bézout domains are elementary divisor rings can be expressed using first-order logic. We have formulated the problem this way and applied various automatic theorem provers in order to try to find a proof that Bézout domains, alone, and with the two other assumptions (adequacy or Krull dimension ≤ 1) are elementary divisor rings. However, none managed so far.

We have in this paper presented the formalization of many results on elementary divisor rings. This way we get interesting examples of coherent

strongly discrete rings and concrete algorithms for studying finitely presented modules. All of the proofs have been performed in a constructive setting, and except for principal ideal domains, without chain conditions.

Acknowledgments: The authors would like to thank Thierry Coquand and Henri Lombardi for interesting discussions. The authors are also grateful to Dan Rosén and Jean-Christophe Filliâtre for their help in the study of the Kaplansky condition using various automatic theorem provers. Finally we would also like to thank Claire Tête for useful comments on a preliminary version of the paper.

Conclusions and future directions

This thesis provides a step in the direction of bridging the gap between algorithms in computer algebra systems and proof assistants. This has been achieved by considering techniques for formalizing efficient programs and theories from constructive algebra in the interactive theorem prover Coq.

1 Conclusions

Constructive algebra is especially well-suited for formalization in intuitionistic type theory because of its computational nature. The possibility to implement constructive proofs as programs that can be run inside Coq is convenient, especially when combined with the CoqEAL approach so that the programs can be made efficient as well.

The use of the SSReflect tactic language has enabled compact and robust formal proofs, while the MathComp library has made the formalization efforts feasible. By basing the developments on this library we avoid reimplementing fundamental mathematical notions and can start building on what has already been done. The developments presented in this thesis mainly rely on the basic algebraic theories provided by the MathComp library, in particular the algebraic hierarchy and the theories on polynomials and matrices. These theories constitute the basis of the MathComp library, which means that they have been very carefully implemented and using them is a pleasure.

However, although MathComp has a well designed library, it imposes some limitations on the user. One thing I found inconvenient when starting to use the library was that some definitions are locked. The reason for this is to prevent the definitions from being expanded in order to make type checking feasible, but the consequence is that computation is blocked. I would have preferred being able to first run my programs on simple examples before trying to prove them correct, but if they used some locked definitions this was not possible. In fact, this was one of the reason why I first got interested in developing CoqEAL – I wanted to run my programs.

A general insight I have reached, while developing libraries of formalized mathematics, is that one of the hardest parts is to find good abstractions and definitions of basic notions that makes it possible to conveniently develop more complicated notions. If not enough thought and effort is spent on developing the basic notions, the complexity of developing more complicated

notions can become unmanageable. This kind of careful engineering requires lots of training, but is both useful and interesting. For instance, our first formalization of the Sasaki-Murao algorithm required about 1200 lines of code, about one year later I rewrote it and managed to cut it down to less than 400 lines. The main difference was that the latter version used the proper operations from the MathComp library instead of *ad hoc* operations that we had defined ourselves.

During my PhD I had to learn lots of mathematics. The best way for me to really understand something has been to formalize it, since all notions and proofs have to be made completely clear and explicit for Coq to accept them. In fact, my understanding has often been greatly increased by just implementing the algorithmic content of a proof. This is one reason why I prefer constructive mathematics to classical mathematics: In order to really understand a proof I need to be able to implement it. Another reason has been nicely formulated by Kraftwerk:

> *"It's more fun to compute"* [Kraftwerk, 1981]

2 Future directions

There are many potential ways for extending the work and further developing the results presented in this thesis. Some ideas on how to do this are discussed below.

2.1 Refinements and constructive algebra

A possible way to extend this work is to add more data and program refinements to the CoqEAL library. It would for example be useful to have efficient representations of matrices and polynomials using arrays in Coq [Armand et al., 2010; Boespflug et al., 2011]. One could also consider further optimizing the algorithms for computing the multivariate gcd and multiplication of polynomials presented in the first paper. The first would involve developing the theory of subresultants [Knuth, 1981] and has been studied previously in Coq [Mahboubi, 2006]. The second would involve generalizing the Karatsuba method to Toom-Cook based methods [Bodrato, 2007], and then further to even more efficient methods based on the fast Fourier transform [Knuth, 1981].

The CoqEAL approach has so far only been applied for formalizing algorithms from mathematics and computer algebra. However, there is no fundamental reason why it could not be used for implementing efficient algorithms from computer science as well. Recently a similar framework was used to formalize Hopcroft's algorithm for automata minimization in Isabelle/HOL [Lammich and Tuerk, 2012]. This kind of formalization should be possible to do using the CoqEAL approach as well and it would be interesting to compare with the Isabelle/HOL development.

A very important example of coherent and strongly discrete rings are multivariate polynomial rings, $k[x_1, \ldots, x_n]$, over a discrete field k. Proving that these are coherent would involve the formalization of the theory of Gröbner

bases and Buchberger's algorithm. This theory has been represented previously in Coq [Persson, 2001; Théry, 1998], but not with the aim of proving that $k[x_1, \ldots, x_n]$ is coherent. It would also be interesting to study this as there is a rich theory on optimizing the Buchberger algorithm for computing Gröbner bases [Cox et al., 2006].

The results presented in the paper on finitely presented modules could be further developed and used to implement algorithms for computing more homological functors like cohomology, Ext and Tor. A possible future direction would be to use this to further develop a library of formally verified computational homological algebra inspired by the Homalg project [Barakat and Robertz, 2008].

2.2 Improving the refinement methodology

Another extension of this work would be to improve the methodology of CoqEAL. An obstacle when implementing the approach presented in the second paper is that there is no internal parametricity in Coq. A possible solution would be to use a tactic like the one presented in [Keller and Lasson, 2012] to derive proofs that closed Coq terms satisfy their parametricity relations. This could be used to improve the proof search algorithm and make it more robust. A more elegant solution would be to work in a system with internal parametricity, like Type Theory in Color [Bernardy and Moulin, 2013], where the parametricity theorems could be obtained for free. Parametricity can also be used for obtaining free theorems in algebra [Garillot, 2011], it would be interesting to investigate this further and see if the developments on constructive algebra could be simplified in a system with internal parametricity.

Ideas from Homotopy Type Theory could also be used to devise an alternative way for doing refinements in type theory. Recall that Homotopy Type Theory extends ordinary type theory with the univalence axiom. This axiom provides a new way to construct equalities between types from isomorphisms, hence extending the notion of equality in type theory in a very interesting way. This axiom implies the *structure identity principle* [Univalent Foundations Program, 2013, Chapter 9.8] which says that not only isomorphic types can be considered as equal, but also isomorphic structures (*e.g.* commutative rings or vector spaces). This means that by proving that two structures are isomorphic (*e.g.* the rings of unary and binary integers) we get that they are equal. To prove an equality relying on computations with unary integers, it suffices to do the computations using binary integers and then transport the proof along the equality. This approach has been discussed by Dan Licata on the Homotopy Type Theory blog [Licata, 2014]. Note that it is possible to do this in traditional type theory as well, but then the user has to transport along the isomorphism manually. In an implementation of Homotopy Type Theory with a computational interpretation of univalence this kind of proofs would instead be for free.

However, this approach to refinements based on univalence would be restricted to isomorphic types and there are many examples where this is not enough (*e.g.* list based and sparse polynomials). But with the use of higher inductive types, quotients can be implemented in Homotopy Type Theory [Rijke and Spitters, 2014]. Sparse polynomials could then be implemented as a

quotient using higher inductive types and would then be isomorphic to the list based polynomials.

2.3 Constructive algebra in Homotopy Type Theory

In many places in the formalizations we have encountered equivalence relations that we would like to quotient with, for example associate elements in rings with explicit divisibility or equality of morphisms between finitely presented modules. The traditional solution in type theory is to use setoids [Barthe et al., 2003] and generalized rewriting [Sozeau, 2009], but this is usually not efficient enough. It is also not very natural compared to standard practices in mathematics where quotienting is a very common operation. In the SSReflect approach to formalization, that heavily relies on rewriting, it would instead be desirable to be able to work with quotient types directly. The fact that quotient types can be defined in Homotopy Type Theory would hence be very useful, not only for doing refinements, but also for representing algebra in type theory.

A peculiarity in the formalizations from the point of view of constructive algebra is that the algebraic hierarchy of MathComp only captures discrete structures (*i.e.* with decidable equality). One reason for this is that types with decidable equality satisfy the *uniqueness of identity proofs* principle, that is, any two proofs of equality are equal [Hedberg, 1998]. However, this means that the algebraic hierarchy is not closed by localization, that is, the localization of a discrete ring need not have decidable equality. In Homotopy Type Theory it would be more natural to generalize the algebraic hierarchy to types that are sets in the Homotopy Type Theory sense, *i.e.* that satisfies the uniqueness of identity proofs principle. An algebraic hierarchy based on this notion of set would hence be a generalization to the one in MathComp. Localization could then be defined in such a way that the localization of a set is again a set. It would be very interesting to pursue these ideas further and develop constructive algebra in this more general setting.

Abelian categories are never explicitly defined as separate structures in the paper on finitely presented modules. The reason for this is that category theory is an especially complicated branch of mathematics to formalize in intuitionistic type theory. This is because of equality: what are good notions of equality on objects, hom-sets and between categories? The traditional way was to represent these using setoids. Homotopy Type Theory offers an alternative approach where the hom-sets are taken to be sets in the above sense (*i.e.* types that satisfy the uniqueness of identity proofs principle) and a category needs to satisfy a version of the univalence axiom [Univalent Foundations Program, 2013, Chapter 9]. We could represent abelian categories based on this and develop homological algebra in this general setting. This would be related to the formalization of elementary algebraic K-theory in Coq by Dan Grayson, in the UniMath implementation of univalent foundations [UniMath, 2014], that also uses abelian categories [Dan Grayson, 2014].

Yet another interesting consequence of working in Homotopy Type Theory is that one can define *propositional truncation* (or *squash types*) [Univalent Foundations Program, 2013, Chapter 3.7] using higher inductive types. This way any type can be turned into a proposition in the sense of Homotopy

Type Theory: a type is an h-proposition if any two elements are equal. Using propositional truncation it is possible to give an alternative definition of the classical logical connectives internally in type theory. Existential quantification can for example be encoded by truncating the Σ-type. This way many properties that traditionally were formulated using Σ-types can be expressed using this kind of existential quantifier instead. For example can the notion of divisibility be expressed by saying that there exists an x such that $b = ax$, and this x could then be extracted if a is regular. Similarly we could express that an ideal is finitely generated or that a module is finitely presented using a predicate returning an h-proposition. The generators of the ideal or the presentation of the module may then be extracted if and only if the construction do not depend on the choice of the representative.

In the MathComp library there is an alternative approach for extracting witnesses from existence statements, namely the `choiceType` infrastructure. By assuming that a type has a choice operator one can turn an `exist` into a Σ-type. We use this in various places of the formalizations, for example in section 6.5.1 to define a function extracting witnesses that two elements are coprime. The notion of existential quantification defined using propositional truncation could be used as a substitute for this. It would be interesting to develop constructive algebra using this notion of existence and see what is gained.

2.4 Computing in Homotopy Type Theory

Regardless of the potentials of formalizing refinements and constructive algebra in Homotopy Type Theory there is a fundamental problem from the point of view of intuitionistic type theory. It is not yet completely clear what the computational interpretation of the univalence axiom or higher inductive types should be. There is however a constructive model that gives a computational justification for univalence and many higher inductive types using cubical sets [Coquand et al., 2014]. The author has taken part in implementing this model using Haskell in order to develop a system called cubical [Cubical, 2014] in which notions like univalence, propositional truncation and quotients compute. The implementation is based on a connection between cubical sets and nominal sets with 01-substitutions developed in [Pitts, 2014].

The cubical system is at the time of writing not a fully fledged proof assistant. A possible solution would be to implement the necessary features for turning it into one. This would involve adding unification, implicit arguments, tactics, proof automation, etc. An alternative approach would be to use cubical as a new core for an already existing proof assistant based on type theory, like Coq or Agda.

The connection between Homotopy Type Theory and parametricity could also be further explored. It seems possible to use ideas from cubical and nominal sets to construct a type theory with internal parametricity similar to Type Theory in Color [Bernardy and Moulin, 2013]. Using this one could hope to one day having a system with good computational properties that has internal parametricity, univalence and higher inductive types. This would then be ideal for further developing the ideas presented in this thesis.

Bibliography

J. Abdeljaoued and H. Lombardi. *Méthodes matricielles - Introduction à la complexité algébrique*. Springer-Verlag, 2004.
↪ 4 citations on pages: 12, 27, 51, and 54.

ACM. Software System Award 2013. http://awards.acm.org/software_system/, Accessed November 2014a.
↪ 1 citation on page: 4.

ACM. Software System Award 2001. http://www.acm.org/announcements/ss_2001.html, Accessed November 2014b.
↪ 1 citation on page: 3.

B. Ahrens, C. Kapulkin, and M. Shulman. Univalent categories and the Rezk completion, 2014. Preprint. http://arxiv.org/abs/1303.0584.
↪ 2 citations on pages: 49 and 92.

K. I. Appel. The Use of the Computer in the Proof of the Four Color Theorem. *Proceedings of the American Philosophical Society*, 128(1):35–39, 1984. URL http://www.jstor.org/stable/986491.
↪ 1 citation on page: 6.

M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending Coq with Imperative Features and Its Application to SAT Verification. In *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 83–98. Springer-Verlag, Berlin, Heidelberg, 2010. URL http://dx.doi.org/10.1007/978-3-642-14052-5_8.
↪ 1 citation on page: 124.

J. Avigad. Methodology and metaphysics in the development of Dedekind's theory of ideals. In *The architecture of modern mathematics*, pages 159–186. Oxford University Press, 2006.
↪ 1 citation on page: 64.

J. Avigad and J. Harrison. Formally Verified Mathematics. *Communications of the ACM*, 57(4):66–75, Apr. 2014. URL http://doi.acm.org/10.1145/2591012.
↪ 1 citation on page: 6.

J. Avigad, K. Donnelly, D. Gray, and P. Raff. A Formally Verified Proof of the Prime Number Theorem. *ACM Transactions on Computational Logic*, 9(1), Dec. 2007. URL http://doi.acm.org/10.1145/1297658.1297660.

↪ 1 citation on page: 5.

M. Barakat and M. Lange-Hegermann. An Axiomatic Setup for Algorithmic Homological Algebra and an Alternative Approach to Localization. *Journal of Algebra and Its Applications*, 10(2):269–293, 2011.
↪ 6 citations on pages: 14, 64, 77, 80, 83, and 108.

M. Barakat and D. Robertz. HOMALG – A Meta-Package for Homological Algebra. *Journal of Algebra and Its Applications*, 7(3):299–317, 2008.
↪ 7 citations on pages: 14, 63, 64, 80, 83, 108, and 125.

E. H. Bareiss. Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination. *Mathematics of Computation*, 22(103):565 – 578, 1968.
↪ 2 citations on pages: 12 and 51.

H. Barendregt, H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. The "Fundamental Theorem of Algebra" Project, Accessed November 2014. http://www.cs.ru.nl/~freek/fta/.
↪ 1 citation on page: 19.

C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer Berlin Heidelberg, 2011. URL http://dx.doi.org/10.1007/978-3-642-22110-1_14.
↪ 1 citation on page: 3.

G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, 2003.
↪ 4 citations on pages: 40, 47, 85, and 126.

J. Bernardy and G. Moulin. Type-Theory in Color. In *ACM SIGPLAN International Conference on Functional Programming*, pages 61–72. ACM, 2013. URL http://doi.acm.org/10.1145/2500365.2500577.
↪ 4 citations on pages: 12, 49, 125, and 127.

J.-P. Bernardy, P. Jansson, and R. Paterson. Proofs for free. *Journal of Functional Programming*, 22:107–152, 2 2012. URL http://journals.cambridge.org/article_S0956796812000056.
↪ 3 citations on pages: 42, 43, and 49.

Y. Bertot, G. Gonthier, S. Biha, and I. Pasca. Canonical big operators. In *Theorem Proving in Higher-Order Logics (TPHOLs'08)*, volume 5170 of *Lectures Notes in Computer Science*, pages 86–101, 2008.
↪ 2 citations on pages: 26 and 112.

M. Bezem, T. Coquand, and S. Huber. A Model of Type Theory in Cubical Sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 107–128, 2014. URL http://drops.dagstuhl.de/opus/volltexte/2014/4628.
↪ 1 citation on page: 127.

M. Bodrato. Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0. In *WAIFI'07 proceedings*, volume 4547 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2007.
↪ 1 citation on page: 124.

M. Boespflug, M. Dénès, and B. Grégoire. Full Reduction at Full Throttle. In *Certified Programs and Proofs*, volume 7086 of *Lecture Notes in Computer Science*, pages 362–377. Springer-Verlag, Berlin, Heidelberg, 2011. URL http://hal.inria.fr/hal-00650940/fr/.
↪ 1 citation on page: 124.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013. URL http://dx.doi.org/10.1017/S095679681300018X.
↪ 1 citation on page: 6.

D. Bridges and E. Palmgren. Constructive Mathematics. The Stanford Encyclopedia of Philosophy. http://plato.stanford.edu/archives/win2013/entries/mathematics-constructive/, 2013.
↪ 1 citation on page: 5.

G. Cano and M. Dénès. Matrices à blocs et en forme canonique. In *JFLA - Journées francophones des langages applicatifs*, 2013. URL http://hal.inria.fr/hal-00779376.
↪ 1 citation on page: 112.

G. Cano, C. Cohen, M. Dénès, A. Mörtberg, and V. Siles. Formalized Linear Algebra over Elementary Divisor Rings in Coq, 2014. Preprint.
↪ 1 citation on page: 16.

J. Chrzaszcz. Implementing Modules in the Coq System. In *TPHOLs*, volume 2758 of *Lectures Notes in Computer Science*, pages 270–286. Springer, 2003.
↪ 1 citation on page: 36.

A. Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.
↪ 1 citation on page: 5.

F. Chyzak, A. Mahboubi, T. Sibut-Pinote, and E. Tassi. A Computer-Algebra-Based Formal Proof of the Irrationality of $\zeta(3)$. In *Interactive Theorem Proving*, volume 8558 of *Lecture Notes in Computer Science*, pages 160–176. Springer, 2014. URL http://dx.doi.org/10.1007/978-3-319-08970-6_11.
↪ 1 citation on page: 10.

B. Cipra. How number theory got the best of the pentium chip. *Science*, 267 (5195):175, January 1995.
↪ 1 citation on page: 1.

K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, pages 268–279. ACM, 2000. URL http://doi.acm.org/10.1145/351240.351266.
↪ 2 citations on pages: 1 and 2.

C. Cohen. Pragmatic Quotient Types in Coq. In *Interactive Theorem Proving*, volume 7998 of *Lectures Notes in Computer Science*, pages 213–228, 2013.
↪ 2 citations on pages: 38 and 85.

C. Cohen and A. Mahboubi. A formal quantifier elimination for algebraically closed fields. In *Proceedings of the 10th ASIC and 9th MKM international conference, and 17th Calculemus conference on Intelligent computer mathematics*, pages 189–203. Springer-Verlag, 2010. URL http://portal.acm.org/citation.cfm?id=1894483.1894502.
↪ 2 citations on pages: 117 and 119.

C. Cohen and A. Mörtberg. A Coq Formalization of Finitely Presented Modules. In *Interactive Theorem Proving*, volume 8558 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2014. URL http://dx.doi.org/10.1007/978-3-319-08970-6_13.
↪ 1 citation on page: 15.

C. Cohen, M. Dénès, and A. Mörtberg. Refinements for free! In *Certified Programs and Proofs*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer International Publishing, 2013. URL http://dx.doi.org/10.1007/978-3-319-03545-1_10.
↪ 1 citation on page: 12.

C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical. https://github.com/simhu/cubical, Accessed November 2014.
↪ 1 citation on page: 127.

D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, Mar. 1990.
↪ 1 citation on page: 24.

Coq Development Team. The Coq Proof Assistant Reference Manual, version 8.4. Technical report, INRIA, 2012.
↪ 6 citations on pages: 1, 35, 51, 64, 79, and 94.

T. Coquand and N. A. Danielsson. Isomorphism is equality. *Indagationes Mathematicae*, 24(4):1105–1120, 2013. In memory of N.G. (Dick) de Bruijn (1918–2012).
↪ 1 citation on page: 49.

T. Coquand and G. Huet. The Calculus of Constructions. Technical Report RR-0530, INRIA, May 1986. URL http://hal.inria.fr/inria-00076024.
↪ 1 citation on page: 5.

T. Coquand and C. Paulin. Inductively defined types. In *COLOG-88*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer Berlin Heidelberg, 1990. URL http://dx.doi.org/10.1007/3-540-52335-9_47.
↪ 1 citation on page: 5.

T. Coquand and A. Spiwack. Towards Constructive Homological Algebra in Type Theory. In *Proceedings of the 14th symposium on Towards Mechanized Mathematical Assistants: 6th International Conference*, Calculemus '07 / MKM '07, pages 40–54, 2007.
↪ 1 citation on page: 92.

T. Coquand, A. Mörtberg, and V. Siles. A Formal Proof of Sasaki-Murao Algorithm. *Journal of Formalized Reasoning*, 5(1):27–36, 2012a. URL http://dx.doi.org/10.6092/issn.1972-5787/2615.
↪ 1 citation on page: 13.

T. Coquand, A. Mörtberg, and V. Siles. Coherent and Strongly Discrete Rings in Type Theory. In *Certified Programs and Proofs*, volume 7679 of *Lecture Notes in Computer Science*, pages 273–288. Springer Berlin Heidelberg, 2012b. URL http://dx.doi.org/10.1007/978-3-642-35308-6_21.
↪ 1 citation on page: 14.

D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms: An introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer, 2006. ISBN 0387946802.
↪ 3 citations on pages: 67, 84, and 125.

Dan Grayson. K-theory. https://github.com/UniMath/UniMath/tree/master/UniMath/Ktheory, Accessed November 2014.
↪ 1 citation on page: 126.

L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer-Verlag, 2008. URL http://dl.acm.org/citation.cfm?id=1792734.1792766.
↪ 1 citation on page: 3.

W. Decker and C. Lossen. *Computing in Algebraic Geometry: A Quick Start using SINGULAR*. Springer, 2006.
↪ 2 citations on pages: 80 and 110.

M. Dénès, A. Mörtberg, and V. Siles. A Refinement-Based Approach to Computational Algebra in Coq. In *Interactive Theorem Proving*, volume 7406 of *Lectures Notes in Computer Science*, pages 83–98. Springer, 2012.
↪ 1 citation on page: 11.

E. W. Dijkstra. Notes on Structured Programming. http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF, Apr. 1970.
↪ 1 citation on page: 2.

E. W. Dijkstra. On the role of scientific thought. http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF, Aug. 1974.
↪ 1 citation on page: 10.

L. Ducos, H. Lombardi, C. Quitté, and M. Salou. Théorie algorithmique des anneaux arithmétiques, des anneaux de Prüfer et des anneaux de Dedekind. *Journal of Algebra*, 281(2):604–650, 2004.
↪ 2 citations on pages: 71 and 72.

A. J. Durán, M. Pérez, and J. L. Varona. The Misfortunes of a Trio of Mathematicians Using Computer Algebra Systems. Can We Trust in Them? *Notices of the American Mathematical Society*, 61(10):1249–1252, 2014.
↪ 1 citation on page: 1.

L. Fuchs and L. Salce. *Modules Over Non-Noetherian Domains*. Mathematical surveys and monographs. American Mathematical Society, 2001. ISBN 9780821819630.
↪ 2 citations on pages: 71 and 121.

R. Gamboa, J. Cowles, and J. V. Baalen. Using ACL2 arrays to formalize matrix algebra. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '03)*, 2003.
↪ 1 citation on page: 120.

F. Garillot. *Generic Proof Tools and Finite Group Theory*. PhD Thesis, Ecole Polytechnique, 2011. URL https://pastel.archives-ouvertes.fr/pastel-00649586.
↪ 1 citation on page: 125.

F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *Proceedings 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, volume 5674 of *Lectures Notes in Computer Science*, pages 327–342, 2009.
↪ 7 citations on pages: 31, 33, 59, 65, 83, 92, and 97.

G. Gonthier. A computer-checked proof of the Four Colour Theorem. http://research.microsoft.com/en-us/um/people/gonthier/4colproof.pdf, 2005.
↪ 2 citations on pages: 6 and 19.

G. Gonthier. Formal Proof—The Four-Color Theorem. In *Notices of the American Mathematical Society*, volume 55, pages 1382–1393, 2008.
↪ 2 citations on pages: 6 and 19.

G. Gonthier. Point-Free, Set-Free Concrete Linear Algebra. In *Interactive Theorem Proving*, volume 6898 of *Lectures Notes in Computer Science*, pages 103–118, 2011. URL http://www.springerlink.com/content/wx57781461004625/.
↪ 6 citations on pages: 21, 64, 80, 94, 109, and 120.

G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq System. Technical report RR-6455, INRIA, 2008. URL http://hal.inria.fr/inria-00258384.
↪ 7 citations on pages: 8, 26, 36, 51, 64, 79, and 94.

G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A Machine-Checked Proof of the Odd Order Theorem. In *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer Berlin Heidelberg, 2013. URL http://dx.doi.org/10.1007/978-3-642-39634-2_14.
↪ 2 citations on pages: 6 and 20.

M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993. URL http://www.cs.ox.ac.uk/tom.melham/pub/Gordon-1993-ITH.html.
↪ 1 citation on page: 5.

Gowers's Weblog. Recent news concerning the Erdos discrepancy problem. http://gowers.wordpress.com/2014/02/11/recent-news-concerning-the-erdos-discrepancy-problem/, Accessed November 2014.
↪ 1 citation on page: 3.

B. Grégoire and X. Leroy. A Compiled Implementation of Strong Reduction. *SIGPLAN Not.*, 37(9):235–246, 2002. URL http://doi.acm.org/10.1145/583852.581501.
↪ 1 citation on page: 28.

B. Gregoire and A. Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. In *TPHOLs*, Lectures Notes in Computer Science, pages 98–113. Springer, 2005.
↪ 1 citation on page: 38.

B. Grégoire and L. Théry. A Purely Functional Library for Modular Arithmetic and Its Application to Certifying Large Prime Numbers. In *IJCAR*, volume 4130 of *Lectures Notes in Computer Science*, pages 423–437. Springer, 2006.
↪ 1 citation on page: 27.

G.-M. Greuel and G. Pfister. *A Singular Introduction to Commutative Algebra*. Springer, 2nd edition, 2007. ISBN 3540735410, 9783540735410.
↪ 3 citations on pages: 80, 82, and 110.

F. Haftmann, A. Krauss, O. Kunčar, and T. Nipkow. Data Refinement in Isabelle/HOL. In *Interactive Theorem Proving*, Lectures Notes in Computer Science. Springer, 2013.
↪ 1 citation on page: 48.

T. C. Hales. The Jordan Curve Theorem, Formally and Informally. *The American Mathematical Monthly*, 114(10):882–894, 2007. URL http://www.jstor.org/stable/27642361.
↪ 2 citations on pages: 5 and 19.

J. Harrison. HOL light: A tutorial introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.
↪ 1 citation on page: 5.

J. Harrison. Formalizing an analytic proof of the Prime Number Theorem. *Journal of Automated Reasoning*, 43:243–261, 2009.
↪ 1 citation on page: 5.

J. Harrison. The HOL Light Theory of Euclidean Space. *Journal of Automated Reasoning*, 50(2):173–190, 2013. URL http://dx.doi.org/10.1007/s10817-012-9250-9.
↪ 1 citation on page: 120.

A. Hatcher. *Algebraic Topology*. Cambridge University Press, 1st edition, 2001. ISBN 0521795400. URL http://www.math.cornell.edu/~hatcher/AT/AT.pdf.
↪ 2 citations on pages: 80 and 121.

M. Hedberg. A Coherence Theorem for Martin-Löf's Type Theory. *Journal of Functional Programming*, 8(4):413–436, 1998.
↪ 3 citations on pages: 87, 92, and 126.

O. Helmer. The Elementary Divisor Theorem for Certain Rings Without Chain Condition. *Bulletin of the American Mathematical Society*, 49:225–236, 1943.
↪ 2 citations on pages: 15 and 117.

J. Hendrix. Matrices in ACL2. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '03)*, 2003.
↪ 1 citation on page: 120.

J. Heras, M. Dénès, G. Mata, A. Mörtberg, M. Poza, and V. Siles. Towards a certified computation of homology groups for digital images. In *CTIC'12*, volume 7309 of *Lectures Notes in Computer Science*, pages 49–57, 2012.
↪ 1 citation on page: 79.

J. Heras, T. Coquand, A. Mörtberg, and V. Siles. Computing Persistent Homology Within Coq/SSReflect. *ACM Transactions on Computational Logic*, 14 (4):1–26, November 2013. URL http://doi.acm.org/10.1145/2528929.
↪ 1 citation on page: 79.

C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, Oct 1969. URL http://doi.acm.org/10.1145/363235.363259.
↪ 1 citation on page: 20.

G. J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004. ISBN 978-0-321-22862-8.
↪ 1 citation on page: 3.

I. Kaplansky. Elementary Divisors and Modules. *Transactions of the American Mathematical Society*, 66:464–491, 1949.
↪ 5 citations on pages: 14, 15, 90, 99, and 115.

A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. In *USSR Academy of Sciences*, volume 145, pages 293–294, 1962.
↪ 1 citation on page: 27.

C. Keller and M. Lasson. Parametricity in an Impredicative Sort. In *Computer Science Logic*, volume 16, pages 381–395, 2012. URL http://hal.inria.fr/hal-00730913.
↪ 2 citations on pages: 49 and 125.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009. URL http://doi.acm.org/10.1145/1629575.1629596.
↪ 1 citation on page: 4.

D. E. Knuth. Notes on the van Emde Boas construction of priority deques: an instructive use of recursion. https://staff.fnwi.uva.nl/p.vanemdeboas/knuthnote.pdf, March 1977.
↪ 1 citation on page: 2.

D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1981. ISBN 0201038226.
↪ 6 citations on pages: 11, 12, 29, 30, 58, and 124.

B. Konev and A. Lisitsa. A SAT Attack on the Erdos Discrepancy Conjecture. *ArXiv e-prints*, February 2014.
↪ 1 citation on page: 3.

L. Kovacs and A. Voronkov. First-Order Theorem Proving and Vampire. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 1–35. Springer, 2013.
↪ 1 citation on page: 3.

Kraftwerk. Computer World, 1981.
↪ 1 citation on page: 124.

P. Lammich. Automatic Data Refinement. In *Interactive Theorem Proving*, volume 7998 of *Lectures Notes in Computer Science*, pages 84–99, 2013.
↪ 1 citation on page: 47.

P. Lammich and T. Tuerk. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In *Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 166–182. Springer Berlin Heidelberg, 2012. URL http://dx.doi.org/10.1007/978-3-642-32347-8_12.
↪ 1 citation on page: 124.

X. Leroy. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 42–54. ACM, 2006. URL http://doi.acm.org/10.1145/1111037.1111042.
↪ 1 citation on page: 4.

D. Licata. Abstract Types with Isomorphic Types, Accessed November 2014. http://homotopytypetheory.org/2012/11/12/abstract-types-with-isomorphic-types/.
↪ 1 citation on page: 125.

H. Lombardi and H. Perdry. The Buchberger Algorithm as a Tool for Ideal Theory of Polynomial Rings in Constructive Mathematics. In B. Buchberger and F. Winkler, editors, *Gröbner Bases and Applications*, pages 393–407. Cambridge University Press, 1998. ISBN 9780511565847.
↪ 2 citations on pages: 67 and 84.

H. Lombardi and C. Quitté. *Algèbre commutative, Méthodes constructives: Modules projectifs de type fini*. Calvage et Mounet, 2011.
↪ 13 citations on pages: 8, 13, 16, 64, 71, 72, 80, 83, 94, 108, 110, 118, and 121.

D. Lorenzini. Elementary Divisor domains and Bézout domains. *Journal of Algebra*, 371(0):609–619, 2012.
↪ 3 citations on pages: 15, 91, and 94.

H. Lüneburg. On a Little but Useful Algorithm. In *Proceedings of the 3rd International Conference on Algebraic Algorithms and Error-Correcting Codes*, pages 296–301. Springer-Verlag, 1986. URL http://dl.acm.org/citation.cfm?id=646022.676247.
↪ 1 citation on page: 117.

Z. Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, Inc., New York, NY, USA, 1994. ISBN 0-19-853835-9.
↪ 1 citation on page: 47.

N. Magaud. Changing Data Representation within the Coq System. In *TPHOLs*, volume 2758 of *Lectures Notes in Computer Science*, pages 87–102. Springer, 2003.
↪ 1 citation on page: 47.

N. Magaud. Programming with Dependent Types in Coq: a Study of Square Matrices, Jan 2005. Unpublished. A preliminary version appeared in Coq contributions.
↪ 1 citation on page: 120.

A. Mahboubi. Proving Formally the Implementation of an Efficient gcd Algorithm for Polynomials. In *3rd International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, pages 438–452. Springer-Verlag, 2006.
↪ 2 citations on pages: 30 and 124.

E. Marshall. Fatal Error: How Patriot Overlooked a Scud. *Science*, 255(5050): 1347, March 1992.
↪ 1 citation on page: 1.

P. Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984a. ISBN 88-7088-105-9.
↪ 1 citation on page: 5.

P. Martin-Löf. Constructive Mathematics and Computer Programming. *Royal Society of London Philosophical Transactions*, 312:501–518, 1984b.
↪ 1 citation on page: 5.

Mathematical Components Project. http://www.msr-inria.fr/projects/mathematical-components-2/, Accessed November 2014.
↪ 1 citation on page: 8.

C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
↪ 1 citation on page: 47.

W. McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 19:263–276, 1997.
↪ 1 citation on page: 3.

R. Mines, F. Richman, and W. Ruitenburg. *A Course in Constructive Algebra*. Springer-Verlag, 1988.
↪ 10 citations on pages: 8, 11, 14, 30, 63, 80, 83, 95, 108, and 114.

A. Mörtberg. Constructive Algebra in Functional Programming and Type Theory. Master's thesis, Chalmers University of Technology, 2010.
↪ 1 citation on page: 74.

R. Nederpelt, H. Geuvers, and R. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.
↪ 1 citation on page: 4.

Nicely, Thomas. Pentium FDIV flaw FAQ. http://www.trnicely.net/pentbug/pentbug.html, August 2011. Accessed November 2014.
↪ 1 citation on page: 1.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lectures Notes in Computer Science*. Springer, 2002.
↪ 2 citations on pages: 4 and 5.

B. Nordström. Terminating general recursion. *BIT*, 28(3):605–619, 1988.
↪ 1 citation on page: 106.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
↪ 1 citation on page: 6.

S. Obua. Proving Bounds for Real Linear Programs in Isabelle/HOL. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK*, volume 3603 of *Lecture Notes in Computer Science*, pages 227–244. Springer, 2005.
↪ 1 citation on page: 120.

R. O'Connor. Karatsuba's multiplication, Accessed November 2014. http://coq.inria.fr/V8.2pl1/contribs/Karatsuba.html.
↪ 1 citation on page: 27.

R. O'Connor. Certified Exact Transcendental Real Number Computation in Coq. In *Theorem Proving in Higher Order Logics (TPHOLs'08)*, volume 5170 of *Lectures Notes in Computer Science*, pages 246–261. Springer, 2008.
↪ 1 citation on page: 34.

F. Palomo-Lozano, I. Medina-Bulo, and J. Alonso-Jiménez. Certification of Matrix Multiplication Algorithms. Strassen's Algorithm in ACL2. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, 2001.
↪ 1 citation on page: 25.

Á. Pelayo and M. A. Warren. Homotopy type theory and Voevodsky's univalent foundations. *Bulletin of the American Mathematical Society*, 51:597–648, 2014.
↪ 1 citation on page: 7.

H. Perdry. Strongly Noetherian rings and constructive ideal theory. *Journal of Symbolic Computation*, 37(4):511–535, 2004.
↪ 3 citations on pages: 13, 64, and 94.

H. Perdry and P. Schuster. Noetherian orders. *Mathematical. Structures in Comp. Sci.*, 21(1):111–124, 2011.
↪ 1 citation on page: 64.

H. Persson. An Integrated Development of Buchberger's Algorithm in Coq. Technical report, INRIA, 2001. URL https://hal.inria.fr/inria-00072316.
↪ 2 citations on pages: 77 and 125.

A. M. Pitts. An Equivalent Presentation of the Bezem-Coquand-Huber Category of Cubical Sets, 2014. Preprint. http://arxiv.org/abs/1401.7807.
↪ 1 citation on page: 127.

H. Poincaré. Analysis situs. *Journal de l'École Polytechnique*, 1:1–123, 1895.
↪ 1 citation on page: 79.

L. Pottier. User contributions in Coq: Algebra, 1999.
↪ 1 citation on page: 120.

A. Quadrat. The Fractional Representation Approach to Synthesis Problems: An Algebraic Analysis Viewpoint Part II: Internal Stabilization. *SIAM Journal on Control and Optimization*, 42(1):300–320, 2003.
↪ 1 citation on page: 64.

M. Raussen and C. Skau. Interview with Jean-Pierre Serre. *Notices of the American Mathematical Society*, 51(2):210–214, 2004.
↪ 1 citation on page: 7.

J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513 – 523, 1983.
↪ 1 citation on page: 43.

E. Rijke and B. Spitters. Sets in Homotopy Type Theory, 2014. Preprint. http://arxiv.org/abs/1305.3835.
↪ 2 citations on pages: 49 and 125.

P. Rudnicki, C. Schwarzweller, and A. Trybulec. Commutative Algebra in the Mizar System. *Journal of Symbolic Computation*, 32(1/2):143–169, 2001. URL http://dx.doi.org/10.1006/jsco.2001.0456.
↪ 1 citation on page: 120.

T. Sasaki and H. Murao. Efficient Gaussian Elimination Method for Symbolic Determinants and Linear Systems. *ACM Transactions on Mathematical Software*, 8(3):277–289, Sept. 1982. URL http://doi.acm.org/10.1145/356004.356007.
↪ 4 citations on pages: 9, 12, 49, and 51.

K. Slind and M. Norrish. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer Berlin Heidelberg, 2008. URL http://dx.doi.org/10.1007/978-3-540-71067-7_6.
↪ 1 citation on page: 5.

M. Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, 2009. URL http://jfr.unibo.it/article/view/1574.
↪ 5 citations on pages: 40, 41, 47, 85, and 126.

M. Sozeau and N. Oury. First-Class type classes. In *Theorem Proving in Higher Order Logics*, volume 5170 of *Lectures Notes in Computer Science*, pages 278–293, 2008. URL http://www.springerlink.com/content/628177q55v3v7306/.
↪ 3 citations on pages: 12, 34, and 41.

B. Spitters and E. van der Weegen. Type Classes for Mathematics in Type Theory. *MSCS, special issue on 'Interactive theorem proving and the formalization of mathematics'*, 21:1–31, 2011.
↪ 1 citation on page: 41.

J. Stein. Documentation of my formalization of Linear Algebra, 2001.
↪ 1 citation on page: 120.

A. Steingart. A group theory of group theory: Collaborative mathematics and the 'uninvention' of a 1000-page proof. *Social Studies of Science*, 42(2): 185–213, April 2012.
↪ 1 citation on page: 6.

C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1-2):11–49, 2000.
↪ 1 citation on page: 2.

V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, Aug. 1969. URL http://www.springerlink.com/content/w71w4445t7m71gm5/.
↪ 3 citations on pages: 24, 36, and 44.

The FlySpeck Project. https://code.google.com/p/flyspeck/wiki/AnnouncingCompletion, Accessed November 2014.
↪ 1 citation on page: 6.

The ForMath Project. http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/, Accessed November 2014.
↪ 1 citation on page: 8.

L. Théry. A Certified Version of Buchberger's Algorithm. In *Proceedings of the 15th International Conference on Automated Deduction: Automated Deduction*, CADE-15, pages 349–364. Springer-Verlag, 1998. URL http://dl.acm.org/citation.cfm?id=648234.753471.
↪ 2 citations on pages: 77 and 125.

UniMath. Univalent Mathematics. https://github.com/UniMath/UniMath, Accessed November 2014.
↪ 1 citation on page: 126.

T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.
↪ 6 citations on pages: 7, 49, 77, 92, 125, and 126.

P. Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *POPL*, pages 307–313. ACM Press, 1987.
↪ 1 citation on page: 47.

P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.
↪ 1 citation on page: 43.

Walking Randomly. A serious bug in MATLAB 2009b? http://www.walkingrandomly.com/?p=1964, November 2009. Accessed November 2014.
↪ 1 citation on page: 1.

F. Wiedijk. *The Seventeen Provers of the World*. Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence. Springer-Verlag, 2006.
↪ 1 citation on page: 4.

S. Winograd. On multiplication of 2x2 matrices. *Linear Algebra and its Applications*, 4:381–388, 1971. URL http://www.sciencedirect.com/science/article/pii/0024379571900097.
↪ 1 citation on page: 24.

X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294. ACM, 2011. URL http://doi.acm.org/10.1145/1993498.1993532.
↪ 1 citation on page: 5.

J. Zeng. A bijective proof of Muir's identity and the Cauchy-Binet formula. *Linear Algebra and its Applications*, 184(0):79–82, 1993.
↪ 1 citation on page: 113.