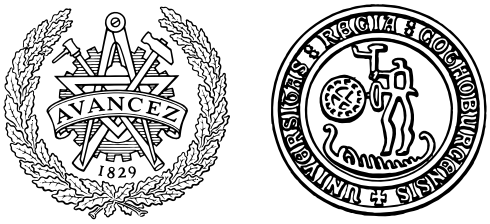
Thesis for the Degree of Licentiate of Engineering

Constructive Algebra in Type Theory

Anders Mörtberg



Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2012

Constructive Algebra in Type Theory
Anders Mörtberg
ISSN 1652-876X

Technical Report no. 96L
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
Telephone +46 (0)31-772 1000

Printed at Chalmers Reproservice
Gothenburg, Sweden 2012

Abstract

This thesis contains four papers aiming at bridging the gap between algorithms implemented in computer algebra systems and interactive proof assistants. This is done by implementing and verifying efficient algorithms using the Coq proof assistant together with the SSReflect extension.

First there is a methodology, based on refinements, for linking implementations of algorithms using rich dependent types to implementations on low-level data types. The first implementation is suitable for deriving theoretical properties while the second one is suited for computation. This methodology is illustrated on four key applications: matrix rank computation, Winograd's fast matrix product, Karatsuba's polynomial multiplication and the computation of the greatest common divisor of multivariate polynomials.

The method has also been applied for verifying an implementation of the Sasaki-Murao algorithm for computing the determinant of a square matrix over a commutative ring in polynomial time. This algorithm can be written as a short and simple functional program, but its correctness involves nontrivial mathematics. The correctness proof, which is new, has been formalized in Coq.

Next the formalization of the notion of (strongly discrete) coherent rings is described. This is a fundamental structure in constructive algebra which represents rings in which it is possible to solve (in)homogeneous systems of equations. Instances of this structure are Bézout domains (for instance $\mathbb{Z}$ and $k[x]$ where k is a field) and Prüfer domains (a generalization of Dedekind domains). We obtain formally verified algorithms for solving systems of equations that are applicable on these structures.

Finally the approach of the first paper is applied to develop algorithms for computing homology groups of simplicial complexes obtained from digital images. This give a formally verified program for counting the number of connected components and holes in digital images. We apply this to count the number of neurons in pictures obtained from synaptical structures.

Acknowledgments

First of all I would like to thank my supervisor Thierry Coquand for his help and ideas that have made it possible for me to finish this thesis. I am also grateful to Cyril Cohen and Vincent Siles for teaching me SSReflect and Coq.

Most of my time as a PhD student has been spent in an office shared with Bassel Mannaa, Guilhem Moulin and Simon Huber who I would like to thank for being such nice office mates and friends. I would also like to thank the people I have written papers with, my colleagues in the ForMath project and the Programming Logic group at Chalmers. Other important people to mention are the members of my follow up group, the administrative personnel that has made my life a lot easier and the rest of the department for making my time here so enjoyable.

On the private side I would like to thank Anjelica and Masjenka for their patience and encouragement. I would also like to thank my family and relatives for always being helpful and supportive. Even if you don't think so, it has been good for me to try to explain my work and the purpose of it. Bruno and Ikaros should also be mentioned for helping me clear my mind and enduring all of my adventures. I would also like to thank all of my friends that I have found during my years at Chalmers and in Gothenburg.

A special thanks goes to Simon Huber, Anjelica Hammersjö, Bassel Mannaa, Malin Ahlberg and Pontus Lindström for interesting and helpful comments on earlier versions of this thesis and to Ramona Enache for answering my numerous questions about practical issues.

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

Contents

Chapter 1

Preliminaries

1.1 Introduction

Computers play an increasingly important role in modern mathematics. Computer algebra systems like MATLAB and MATHEMATICA are fundamental tools in scientific computation and also to a greater extent in pure mathematics. Hence it is important that these systems are reliable. A possible approach for increasing the reliability is to use a tool for random testing like QuickCheck [6]. This kind of tool could be used to prevent bugs like the one that was found in 2009 when MATLAB gave an incorrect solution to a simple system of equations [29].

In any case, it is necessary to give mathematically precise specifications of the algorithms represented in computer algebra systems. To have completely formal specifications in an interactive proof assistant seems to be an interesting approach since the specification has to be given in full detail. In formalisms integrating computations and reasoning, like Type Theory [22], there is a clear connection between the specification of the algorithm and what is actually implemented, which can be used to further increase the reliability in the implemented algorithms.

Computers have been useful to prove theorems where standard pen and paper approaches were intractable. Two examples of this are the Four Color Theorem and the Kepler Conjecture that were both proven with the aid of computer programs. In order to increase the reliability of these proofs there have been substantial efforts to formalize the programs and mathematics involved so that all of the logical inference steps in the proofs can be checked by a computer. The first of these was finished by George Gonthier et al. in 2004 [12] and the second is, as of August 2012, still being verified by the FlySpeck Project [26] led by Thomas Hales.

There is also a reliability issue of very large and complicated mathematical results that might be understood by only a few experts. One example of such a proof is the proof of Fermat's Last Theorem where the first version was believed to be correct in 1993 but an error was corrected in 1995. Another example is the

classification of finite simple groups whose proof consists of tens of thousands of pages written by many authors over a long period of time. This problem is discussed further by Jean-Pierre Serre in an interview from when he was awarded the Abel prize in 2003 [24]. Formal proofs of these results would increase the reliability and the formalization would, hopefully, also give rise to new methods and results.

Another motivation behind formalizing mathematical results 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 might need to be improved in order to be able to represent mathematics more conveniently yielding better tools and techniques. Furthermore, formally verified efficient algorithms are useful for implementing decision procedures that can be used in proof assistants for doing proofs by reflection.

Although the formalization of big theorems like the Four Color Theorem and the Kepler Conjecture provide evidence that proof assistants are mature enough to handle modern mathematics. However, there is still a large gap between the mathematical algorithms formalized in proof assistants and the algorithms in computer algebra systems used by mathematicians and scientists. We believe in the approach of the Mathematical Components Project [20] which says that this gap can be bridged by implementing general purpose libraries of mathematical theories using ideas and design principles from software engineering to obtain scalability and reusability of both code and proofs.

The system used in the formalizations[1] presented in this thesis is the interactive proof assistant Coq [7]. This system is based on the calculus of inductive constructions which by the Curry-Howard isomorphism is not only a system for making formal proofs but also a functional programming language. This means that both the programs and their proofs of correctness can be implemented using the same language and logic. Another consequence is that the algorithms have to be written using functional programming, which differs from standard presentations of mathematical algorithms that are usually imperative. We believe that functional programming is more natural than imperative programming for implementing mathematical algorithms.

The formalizations have been performed using the *small scale reflection* (SSReflect) [13] extension to Coq. This extension was initially developed during the formalization of the Four Color Theorem by George Gonthier et al. It has since then been used in the Mathematical Components Project [20] during the ongoing formalization of the Feit-Thompson Theorem which is part of the classification of finite simple groups. The idea of small scale reflection is to use computation to automate small proof steps resulting in a concise proof style that is closer to pen and paper proofs.

The SSReflect extension contains a large and well designed library of al-

[1] The formalizations can be found at: `http://www.cse.chalmers.se/~mortberg/`

ready formalized mathematical theories containing, among other things: polynomials, matrices and an algebraic hierarchy. By using this library we avoid reimplementing these fundamental notions and may start building on what is already implemented.

However, although SSReflect has a well designed library, it imposes some limitations on the user. In order to prevent definitions from being expanded during type checking some definitions are *locked* [13] which means that computation on them are blocked. This implies that many definitions lack direct effective computation, which is strange from the point of view of Type Theory [22].

Another limitation is that the algebraic hierarchy only captures discrete structures (i.e. with decidable equality) in order to enable equational reasoning. This limitation to discrete structures is not very natural from the point of view of constructive mathematics. In constructive mathematics the law of excluded middle and proof by contradiction are not valid methods for doing proofs in general. By avoiding these principles the mathematical theories become inherently computational (see e.g. [5]) which makes them suitable for implementation on computers, but it is usually not necessary to restrict to discrete structures [21].

The work in this thesis has been carried out as part of the European project For-Math – *Formalization of Mathematics* [27]. The goal of the project is to develop formally verified libraries of mathematics concerning algebra, linear algebra, real number computations and algebraic topology. These libraries should be designed as software libraries using ideas from software engineering to increase reusability and scalability.

This thesis presents an approach for bridging the gap between computer algebra systems and proof assistants. This approach is applied to implement formally verified libraries and algorithms based on mathematical theories from linear algebra and commutative algebra using Coq and SSReflect. The main sources of constructive algebra used during the formalizations are the book by Mines, Richman and Ruitenburg [21] and the more recent book by Lombardi and Quitté [18]. The libraries and algorithms have been successfully applied to represent formally verified algorithms from computational algebraic topology with applications in biomedical engineering.

1.2 Contributions

This section contains summaries of the four papers in this thesis. The papers contains a methodology for implementing and formally verifying mathematical algorithms together with examples of this methodology being applied to obtain implementations of algorithms suitable for computation.

1.2.1 A Refinement-Based Approach to Computational Algebra in Coq

The first paper [11] presents a methodology for implementing efficient algebraic algorithms and proving them correct. This is done by implementing a simple and often inefficient version of the algorithm on rich datatypes which is refined 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 proved correct in Coq using a library that we have implemented using the SSReflect library and tactics.

The idea of program refinements used in the paper is summarized in Fig. 1.1:

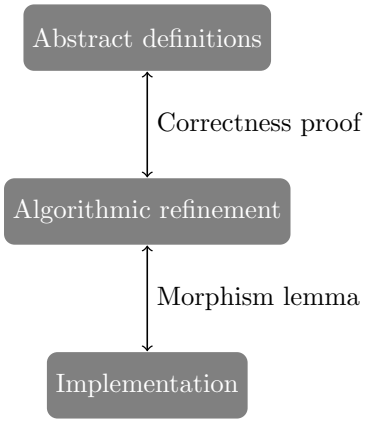


Figure 1.1: The three steps of refinements

The methodology in Fig. 1.1 can be more concretely summarized in our setting as:

1. Implement an abstract version of the algorithm using SSReflect's structures and use the libraries to prove properties about them. Here we may use the full power of dependent types when proving correctness.

2. Refine this algorithm into an efficient one using SSReflect's structures and prove that it behaves like the abstract version.

3. Translate the SSReflect structures and the efficient algorithm to the low-level data types, ensuring that they will perform the same operations as their high-level counterparts.

By separating the implementation of the algorithm used for deriving properties and the one used for computation we overcome the limitation of SSReflect

not having direct effective computation. In the rest of the thesis we will refer to the noneffective implementation as abstract and the other one as effective or computational.

Using this methodology we have implemented a library of computational structures with four main examples of algorithms from linear and commutative algebra:

- Efficient polynomial multiplication using Karatsuba's algorithm.
- Multivariate greatest common divisor (gcd) of polynomials.
- Rank computation of matrices with coefficients in a field.
- Efficient multiplication of matrices based on the Winograd algorithm.

The second of these, multivariate gcd of polynomials, is interesting from the point of view of constructive algebra as the correctness proof neither rely on the field of fractions nor unique factorization. It is instead based on Gauss' Lemma as in [17] and the notion of gcd domains [21].

My contribution to this paper is the implementation and correctness proof of the algorithms on polynomials together with the implementation of the algebraic hierarchy of computational structures used in the library. I have written the sections on polynomials and I also participated in writing the other sections of the paper.

This paper has been accepted for publication in the LNCS post-proceedings of the 2012 edition of the conference on Interactive Theorem Proving.

1.2.2 A Formal Proof of Sasaki-Murao Algorithm

The second paper [8] explains the formalization of a simple polynomial time algorithm for computing the determinant of square matrices over any commutative ring. The algorithm is based on Bareiss' algorithm [4], which can be compactly presented using functional programming notations as:

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

dvd_step :: R -> Matrix R -> Matrix R
dvd_step g M = mapM (\x -> g | x) M

bareiss_rec :: R -> Matrix R -> R
bareiss_rec g M = case M of
  Empty -> g
  Cons a l c M -> bareiss_rec a (dvd_step g (a * M - c * l))

bareiss :: Matrix R -> R
bareiss M = bareiss_rec 1 M
```

Here `R` is assumed to be a ring with a division operation `|`. The datatype `Matrix` is a convenient datastructure for this algorithm where the first element is the top-left element and the two lists are the first row and column without the top-left element. This algorithm is both simple and computes the determinant over any commutative ring in polynomial time. But, the standard proof of correctness involves complicated identities for determinants called Sylvester identities [1]. In order to formalize the correctness of this algorithm an alternative correctness proof, more suitable for formalization, was found. This proof is (arguably) simpler and some of the Sylvester identities can be proved as corollaries of it.

The Sasaki-Murao algorithm [25] uses an elegant trick to avoid zeroes on the main diagonal which is to apply the algorithm to the matrix used when computing the characteristic polynomial of a matrix, that is, we negate the matrix and add x to every element on the diagonal. This way Bareiss' algorithm can be applied without any swapping of rows, which makes it possible to obtain not only the determinant but also the characteristic polynomial of the matrix in polynomial time. Another benefit of doing the computations on the polynomial ring is that polynomial pseudo-division [17] may be used, which means that there is no need to assume that the ring has a division operation.

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 effective version is a simple and verified algorithm for computing the determinant of a matrix using operations like matrix multiplication, polynomial pseudo-division and Horner evaluation of polynomials.

My contribution to this paper is mainly in working on the formalization of the correctness proof and the implementation of the efficient version of the algorithm. I also implemented a Haskell version used for benchmarks and comparison in the section with conclusions in the paper.

This paper has been accepted for publication in the Journal of Formalized Reasoning in 2012.

1.2.3 Coherent and Strongly Discrete Rings in Type Theory

The third paper [9] presents the formalization of algebraic structures that were not present in the SSReflect libraries: coherent and strongly discrete rings. These notions abstract over the ability to solve systems of homogeneous and inhomogeneous equations. Examples are Bézout domains (for example $\mathbb{Z}$ and $k[x]$ where k is a field) and Prüfer domains (a generalization of Dedekind domains). We obtain formally verified algorithms for solving systems of equations over these structures.

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. The main difficulty seems to appear when the library has to be extended to support new operations and the reason for this is the ingenuity

of the representation of many algorithms in the SSReflect library. However we managed to implement effective versions of the algorithms which indicates that the methodology is applicable on more complicated examples as well.

In order to benefit as much as possible from the SSReflect libraries and tactics some specializations had to be made that are not natural from the point of view of constructive mathematics. For instance, we only consider coherent rings that are strongly discrete (i.e. with decidable ideal membership) in order to be able to develop ideal theory in a convenient way. However when restricting to these decidable structures the SSReflect approach yields quite compact and simple formal proofs.

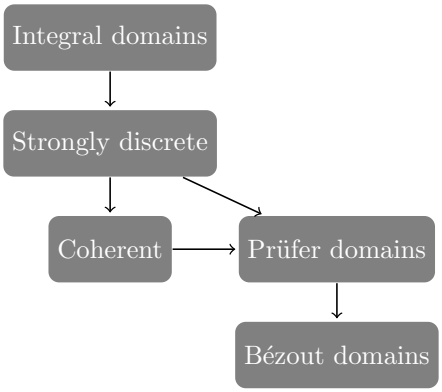


Figure 1.2: The extension to the SSReflect hierarchy

In Fig. 1.2 the extension to the SSReflect hierarchy is presented. Integral domains are already present in the hierarchy and the extension consists of the other structures. The arrows mean that the target is an instance of the source.

Our main motivation for studying these kind of structures is that strongly discrete coherent rings is a fundamental structure in constructive algebra [21] which can be used as a basis for developing formalized libraries of computational homological algebra as in [2].

I have contributed to all parts of this paper, both formalizing the results and writing the paper. I also implemented the executable versions and correctness proof of the algorithms and structures.

The paper has been submitted to the 2012 edition of the conference on Certified Programs and Proofs.

1.2.4 Towards a Certified Computation of Homology Groups for Digital Images

In the fourth paper [14] we implement formally verified algorithms for counting the number of connected components and holes in digital images by computing

homology groups. These groups are topological invariants that can be computed, in our setting, using only basic linear algebra.

The algorithm that is used for this is summarized in Fig. 1.3:

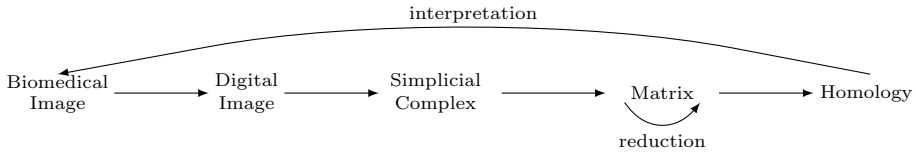


Figure 1.3: Computing homology from a digital image

The biomedical image is represented as a list of pixels that is converted to a *simplicial complex* which is a combinatorial representation of the image suitable for homology computation. From this boundary (or incidence) matrices are computed as in [15] which are then simplified using an algorithm from discrete Morse theory. Finally the homology groups of the simplicial complex is obtained by computing the rank of the matrices [16]. As we work with two dimensional digital images it suffices to perform the computations on $\mathbb{Z}/2\mathbb{Z}$ which means, as this is a field, that we can reuse the rank algorithm from the first paper.

The homology groups in dimension zero and one counts the number of connected components and holes in the image. We explain in the paper an application from biomedical engineering for counting the number of synapses in a picture of a neuron using this technique.

My contribution to this paper is mainly on the computational side of the formalization and the linking of the different effective algorithms. I also developed a Haskell prototype that was used to compare the results and for doing benchmarks.

This paper has been published in the LNCS post-proceedings of the 4th International Workshop on Computational Topology in Image Context 2012.

1.3 Future Work

This thesis presents a methodology and basis for developing formally verified efficient mathematical algorithms in Coq that seems to be applicable on many different mathematical theories and algorithms.

A possible future direction would be to use this approach to develop a formally verified library of computational homological algebra inspired by the HOMALG project [3]. Building on the results presented in the third paper, the category of finitely presented modules over strongly discrete coherent rings can be represented. The first step would be to prove that this forms an abelian category, that is, that this is suitable for building homological algebra on top of. The next step would then be to use this to implement algorithms for computing homological functors like homology, Ext and Tor.

The results could be used as a basis for extending the work presented in the fourth paper to consider other homology theories, for example, homology with coefficients in $\mathbb{Z}$. Using this the computation of more interesting homological properties of different topological spaces can be computed using formally verified algorithms.

A very important example of coherent rings are multivariate polynomial rings, $k[x_1, \ldots, x_n]$, over a field k. Proving that these are coherent would involve the formalization of the theory of Gröbner bases and Buchberger's algorithm. This has been done previously in Coq [23, 28] and would be an interesting problem to study as the simple Buchberger algorithm for computing Gröbner bases is inefficient and has many possible optimizations [10]. The methodology of the first paper should then be applicable for developing formally verified versions of more efficient algorithms for computing Gröbner bases.

Another interesting problem is to extend the work on multivariate gcd computations by considering more efficient algorithms based on subresultants [17]. This problem has also been studied previously in Coq [19] and it would be interesting to compare this work with a formalization done using the approach of the first paper.

Bibliography

[1] J. Abdeljaoued and H. Lombardi. *Méthodes matricielles - Introduction à la complexité algébrique*. Springer, 2004.

[2] M. Barakat and M. Lange-Hegermann. An Axiomatic Setup for Algorithmic Homological Algebra and an Alternative Approach to Localization. *J. Algebra Appl.*, 10(2):269–293, 2011.

[3] M. Barakat and D. Robertz. homalg – A Meta-Package for Homological Algebra. *J. Algebra Appl.*, 7(3):299–317, 2008.

[4] E. H. Bareiss. Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination. *Mathematics of Computation*, 22(103):565 – 578, 1968.

[5] D. Bridges. Constructive Mathematics. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall edition, 2012.

[6] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[7] Coq development team. The Coq Proof Assistant Reference Manual, version 8.3. Technical report, 2010.

[8] T. Coquand, A. Mörtberg, and V. Siles. A Formal Proof of Sasaki-Murao Algorithm. *Journal of Formalized Reasoning*, 2012.

[9] T. Coquand, A. Mörtberg, and V. Siles. Coherent and Strongly Discrete Rings in Type Theory. 2012.

[10] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms: An introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer, 2006.

[11] 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*, 2012.

[12] G. Gonthier. Formal Proof—The Four-Color Theorem. In *Notices of the American Mathematical Society*, volume 55, pages 1382–1393, 2008.

[13] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq System. Rapport de recherche RR-6455, INRIA, 2008.

[14] J. Heras, M. Dénès, G. Mata, A. Mörtberg, M. Poza, and V. Siles. Towards a Certified Computation of Homology Groups for Digital Images. In *Computational Topology in Image Context*, volume 7309 of *Lecture Notes In Computer Science*, pages 49–57, Bertinoro, Italie, 2012. Springer.

[15] J. Heras, M. Poza, M. Dénès, and L. Rideau. Incidence Simplicial Matrices Formalized in Coq/SSReflect. In *Proceedings 18th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (Calculemus'2011)*, volume 6824 of *Lectures Notes in Computer Science*, pages 30–44, 2011.

[16] T. Kaczynski, K. Mischaikow, and M. Mrozek. *Computational Homology*, volume 157 of *Applied Mathematical Sciences*. Springer, 2004.

[17] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1981.

[18] H. Lombardi and C. Quitté. *Algèbre commutative, Méthodes constructives: Modules projectifs de type fini*. Calvage et Mounet, 2011.

[19] A. Mahboubi. Proving Formally the Implementation of an Efficient gcd Algorithm for Polynomials. In *3rd International Joint Conference on Automated Reasoning (IJCAR)*, Lecture Notes in Artificial Intelligence, pages 438–452. Springer-Verlag, 2006.

[20] Mathematical Components Project. `http://www.msr-inria.inria.fr/Projects/math-components/`, Accessed August 2012.

[21] R. Mines, F. Richman, and W. Ruitenburg. *A Course in Constructive Algebra*. Springer-Verlag, 1988.

[22] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, USA, 1990.

[23] H. Persson. An Integrated Development of Buchberger's Algorithm in Coq. 2001.

[24] M. Raussen and C. Skau. Interview with Jean-Pierre Serre. *Notices of the American Mathematical Society*, 51(2):210–214, 2004.

[25] T. Sasaki and H. Murao. Efficient Gaussian Elimination Method for Symbolic Determinants and Linear Systems. *ACM Trans. Math. Softw.*, 8(3):277–289, Sept. 1982.

[26] The FlySpeck Project. http://code.google.com/p/flyspeck/, Accessed August 2012.

[27] The ForMath Project. http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/, Accessed August 2012.

[28] 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, London, UK, 1998. Springer-Verlag.

[29] Walking Randomly. A serious bug in MATLAB 2009b? http://www.walkingrandomly.com/?p=1964, Accessed August 2012.

Chapter 2

Paper 1: A Refinement-Based Approach to Computational Algebra in Coq

A Refinement-Based Approach to Computational Algebra in Coq*

Maxime Dénès[1], Anders Mörtberg[2], and Vincent Siles[2]

[1] INRIA Sophia Antipolis – Méditerranée, France
[2] University of Gothenburg, Sweden
Maxime.Denes@inria.fr, {mortberg,siles}@chalmers.se

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 abstract counterparts. We illustrate this methodology on key applications: matrix rank computation, Winograd'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 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 [2], the Four Color Theorem [6] and the Jordan Curve Theorem [10] 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 SSReflect library [8] for the Coq proof assistant) advocates the use of small scale reflection 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. 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

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

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 3, and polynomials in Section 4. In Section 5, we give a solution to unify both examples by describing CoqEAL[3], a library built using this methodology on top of the SSReflect libraries.

2 Refinements

Refinements are commonly used to describe successive steps when verifying a program. Typically, a specification is expressed in Hoare logic, 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. Thus the implementation is an immediate translation of the algorithm, see Fig. 1.

However, in our approach, the three 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

[3] Documentation available at `http://www-sop.inria.fr/members/Maxime.Denes/coqeal/`

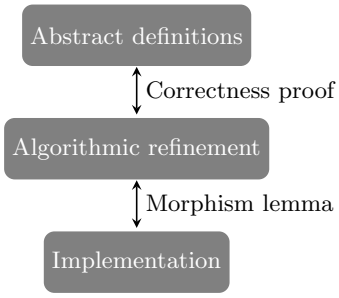


Fig. 1. The three steps of refinement

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.

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

1. Implement an abstract version of the algorithm using SSReflect's structures and use the libraries to prove properties about them. Here we can use the full power of dependent types when proving correctness.
2. Refine this algorithm into an efficient one using SSReflect's structures and prove that it behaves like the abstract version.
3. Translate the SSReflect structures and the efficient algorithm to the low-level data types, ensuring that they will perform the same operations as their high-level counterparts.

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 SSReflect library [7]. 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.

3.1 Representation

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

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

(* 'M[R]_(m,n) *)
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 $\mathtt{m} \times \mathtt{n}$ 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. A concrete matrix is built from an abstract one by mapping canonical enumerations (`enum`) of ordinals to the corresponding coefficients in the abstract 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 concrete matrices it is sufficient to prove that `seqmx_of_mx` is injective (`==` denotes boolean equality):

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

Operations like addition are straightforward to implement, and their correctness is expressed through a morphism lemma, stating that the concrete representation of the sum of two matrices is the concrete sum of their concrete representations:

```
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 / M + N >-> addseqmx M N}.
```

Here `morph` is notation meaning that `seqmx_of_mx` is an additive morphism from abstract to concrete matrices. It is worth noting that we could have stated all our morphism lemmas with the converse operator (from concrete matrices to abstract 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 richer types.

Like the `addseqmx` operation, we have developed concrete implementations of most of the matrix operations provided by the SSReflect library and proved the corresponding morphism lemmas. Among these operations we can cite: subtraction, scaling, transpose and block operations.

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 abstract 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:

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

Now pose $R = \begin{pmatrix} R_1 \\ R_2 \end{pmatrix}$, since $a_{i,1} \neq 0$, this means that $rank\ A = rank\ B = 1 + rank\ R$. 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) {struct n} : '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 SSReflect 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 concrete version of this algorithm is a direct translation of the algorithm using only concrete matrices and executable operations on them. This executable version (called `rank_elim_seqmx`) is then linked to the abstract implementation 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 concrete matrices have morphism lemmas which means that the proof can be done simply by expanding the definitions and applying the translation morphisms.

3.3 Fast matrix product

In the context we presented, the naïve matrix product (i.e. with cubic complexity) of two matrices M and N can be 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).
```

`*m` is SSReflect's notation for the matrix product. 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 [19] 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.[4] 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 [4] are slower in practice because of huge hidden constants. Thus, we implemented a variant of this algorithm suggested by Winograd in 1971 [20], decreasing the required number of additions and subtractions to 15 (instead of 18 in Strassen's original proposal). This choice reflects the implementation of matrix product in most of modern computer algebra systems. A previous formal description of this algorithm has been developed in ACL2 [17], 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 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 winograd_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
```

[4] $\log_2 7$ is approximately 2.807

```
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, `winograd_step A B` should be the product of `A` and `B` (`=2` denotes extensional equality):

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

This proof is made easy by the use of the `ring` tactic (the script 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 Winograd multiplication with dynamic peeling is called `winograd` and it is proved correct with respect to the usual matrix product:

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

The concrete version is called `winograd_seqmx` and it is also just a direct translation of `winograd` using only concrete operations on `seq` based matrices. In the next section, Fig. 2 shows some benchmarks of how well this implementation performs compared to the naïve matrix product, but we will first discuss how to implement concrete algorithms based on dependently typed polynomials.

4 Polynomials

Polynomials in the SSReflect 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 and proofs that the polynomials form a commutative ring using these operations. The implementation of these operations use big operators [3] which means that it is not possible to compute with them.

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 concrete polynomial from an abstract polynomial, simply apply the record projection (called `polyseq`) to extract the list from the record. The soundness of concrete polynomials is proved by showing that the pointwise boolean equality on the projected lists reflects the equality on abstract 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 concrete polynomials than for concrete 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 concrete 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 SSReflect polynomial library, including pseudo-division, using executable polynomials.

4.1 Fast polynomial multiplication

The naïve polynomial multiplication algorithm presented in the previous section requires $\mathcal{O}(n^2)$ operations. A more efficient algorithm is Karatsuba's algorithm [1, 11] 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 implemented efficiently by padding the list of coefficients by n zeroes). 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. Now 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 naïve algorithm.[5] 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 been implemented in Coq previously for binary natural numbers [15] and for numbers represented by a tree-like structure [9]. 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 => if (size p <= 2) || (size q <= 2) then p * q else
     let m := minn (size p)./2 (size q)./2 in
     let (p1,p2) := splitp m p in
     let (q1,q2) := splitp m q in
     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.
```

Here `splitp` is a function that splits the polynomial at the correct point using `take` and `drop`. There is also a wrapper function named `karatsuba` that calls `karatsuba_seq` with the greatest degree of `p` and `q`. The correctness of this algorithm is expressed by:

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

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

As `p` and `q` are SSReflect polynomials this lemma can be proved using all of the theory in the library. The next step is to implement the executable version (`karatsuba_seq`) of this algorithm which is done by changing all the operations in the above version to executable operations on concrete polynomials. The correctness of the concrete algorithm is then proved 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 concrete representation and the high-level ones.

In Fig. 2 the running time of the different multiplication algorithms that we have implemented is compared:

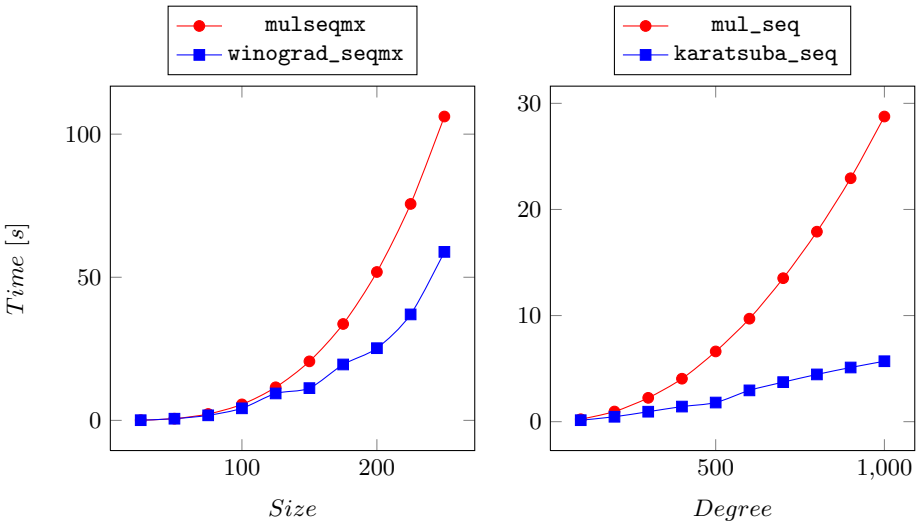


Fig. 2. Benchmarks of Winograd and Karatsuba multiplication

The benchmarks have been done by computing the square of integer matrices and polynomials using the Coq virtual machine (i.e. by running `vm_compute`). It is clear that both the implementation of Winograd matrix multiplication and Karatsuba polynomial multiplication is faster than their naïve counterparts, as expected.

4.2 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 of Knuth in [12] 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) we get an algorithm for computing the gcd of any two polynomials. Below is our implementation of this algorithm together with explanations of the operations:

```
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 (gcdsr p1) (gcdsr q1))%:P in
  d * gcdp_rec (size (pp p1)) (pp p1) (pp q1).
```

- `modp p q` computes the remainder after pseudo-dividing `p` by `q`.
- `pp p` computes the primitive part of `p` by dividing it by its content.
- `gcdsr p` computes the content of `p`.
- `gcdr (gcdsr p1)(gcdsr q1)` computes 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 : forall p q g, 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 is stated in constructive algebra [14] as: If R is a gcd domain then so is $R[X]$. Our algorithmic proof is different (and arguably simpler) than the one in [14]; for instance, we do not go via the field of fractions of the ring.

As noted in [12], this algorithm may be inefficient when applied on the polynomials over integers. The reference [12] provides a solution in this case, based on subresultants. This would be a further refinement of the algorithm, which would be interesting to explore since subresultants have been already analyzed in Coq [13].

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

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

But when running the concrete implementation there is a quite subtle problem: the `polyseq` projection links the abstract polynomials with the concrete polynomials of type `seq R` where `R` is a *ring* with a gcd operation. Let us consider multivariate polynomials, for example $R[x, y]$. In this case the concrete type will be `seq (seq R)`, but `seq R` is not a ring so our algorithm is not applicable! The next section explains how to resolve this issue so that it is possible to implement computable algorithms of the above kind that rely on the computability of the underlying ring.

5 Algebraic 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 the computation relies on the computability of the underlying ring as is the case when computing the characteristic polynomial of a square matrix for instance. For this, one needs to compute with matrices of polynomials which will require a concrete implementation of matrices with coefficients being a concrete 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 SSReflect algebraic hierarchy.

5.1 Design of the library

We have implemented computable counterparts to the basic structures in this hierarchy, e.g. $\mathbb{Z}$-modules, rings and fields. These are implemented in the same manner as presented in [5] using canonical structures. Here are a few examples of the mixins we use:

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

(* Mixin for "Computable" Z-modules *)
Record mixin_of (V : zmodType) (T: Type) : 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) : Type := 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 SSReflect structures to the computable structures and the only property that is required of it is that it is injective, so we are sure that different high-level objects are mapped to different computable objects.

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 `*m` instead of a naïve one), and the only thing we have to prove is that the implementations behave the same as SSReflect's operations *on the subset of "well-formed terms"* (e.g. for polynomials, lists that do not end with 0). This is done by providing the corresponding morphism lemmas.

The operations presented in the previous sections can then be implemented by having computable structures as the underlying structure instead of dependently typed ones. This way one can prove that polynomials represented as lists is a computable ring by assuming that the coefficients are computable and hence get ring operations that can be applied on multivariate polynomials built by iterating the 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.

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.
```

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 $\mathbb{Z}[x, y]$ one can 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 $\frac{2}{3} + \frac{2}{3}y + \frac{1}{2}xy + \frac{1}{2}xy^2$. The resulting gcd should be interpreted as $\frac{1}{3} + \frac{1}{4}xy$.

6 Conclusions and Further Work

In this paper, we showed how to use high-level libraries to prove properties of algorithms, while retaining good execution capabilities by providing efficient low-level implementations. The need of modularity of the executable structure appears naturally and the methodology explained in [5] 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 [16] are obtained by proving that these real numbers are isomorphic to an abstract, pre-existing but less efficient version. We tried to show 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, some of the structures 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. 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. Also, we could use other packaging methods, like type classes [18], to simplify the layout of the library. 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.

References

[1] J. Abdeljaoued and H. Lombardi. *Méthodes matricielles - Introduction à la complexité algébrique*. Springer, 2004.

[2] H. Barendregt, H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. The "fundamental theorem of algebra" project. `http://www.cs.ru.nl/~freek/fta/`.

[3] Y. Bertot, G. Gonthier, S. Biha, and I. Pasca. Canonical big operators. In *Theorem Proving in Higher-Order Logics (TPHOLs'08)*, volume 5170 of *LNCS*, pages 86–101, 2008.

[4] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, Mar. 1990.

[5] 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 *LNCS*, pages 327–342, 2009.

[6] G. Gonthier. Formal proof—the four-color theorem. In *Notices of the American Mathematical Society*, volume 55, pages 1382–1393, 2008.

[7] G. Gonthier. Point-Free, Set-Free concrete linear algebra. In *Interactive Theorem Proving*, volume 6898 of *LNCS*, pages 103–118, 2011.

[8] G. Gonthier and A. Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical report, Microsoft Research INRIA, 2009. `http://hal.inria.fr/inria-00258384`.

[9] B. Grégoire and L. Théry. A purely functional library for modular arithmetic and its application to certifying large prime numbers. In U. Furbach and N. Shankar, editors, *IJCAR*, volume 4130 of *LNCS*, pages 423–437. Springer, 2006.

[10] T. C. Hales. The jordan curve theorem, formally and informally. In *The American Mathematical Monthly*, volume 114, pages 882–894, 2007.

[11] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. In *USSR Academy of Sciences*, volume 145, pages 293–294, 1962.

[12] D. E. Knuth. *The art of computer programming, volume 2: seminumerical algorithms*. Addison-Wesley, 1981.

[13] A. Mahboubi. Proving formally the implementation of an efficient gcd algorithm for polynomials. In *3rd International Joint Conference on Automated Reasoning (IJCAR), LNAI*, pages 438–452. Springer-Verlag, 2006.

[14] R. Mines, F. Richman, and W. Ruitenburg. *A Course in Constructive Algebra*. Springer-Verlag, 1988.

[15] R. O'Connor. Karatsuba's multiplication. `http://coq.inria.fr/V8.2pl1/contribs/Karatsuba.html`.

[16] R. O'Connor. Certified exact transcendental real number computation in coq. In *Theorem Proving in Higher Order Logics (TPHOLs'08)*, volume 5170 of *LNCS*, pages 246–261. Springer, 2008.

[17] F. Palomo-Lozano, I. Medina-Bulo, and J. Alonso-Jiménez. Certification of matrix multiplication algorithms. strassen's algorithm in ACL2. In *Supple-*

mental Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics, volume Informatics Research Report EDI-INF-RR-0046, Edinburgh, 2001.

[18] M. Sozeau and N. Oury. First-Class type classes. In *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*, pages 278–293, 2008.

[19] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, Aug. 1969.

[20] S. Winograd. On multiplication of 2x2 matrices. *Linear Algebra and its Applications*, 4:381 – 388, 1971.

Chapter 3

Paper 2:
A Formal Proof of Sasaki-Murao Algorithm

A Formal Proof of Sasaki-Murao Algorithm*

Thierry Coquand, Anders Mörtberg, and Vincent Siles

University of Gothenburg, Sweden
{coquand,mortberg,siles}@chalmers.se

Abstract. The Sasaki-Murao algorithm computes 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.

1 Introduction

The goal of this note is to present a formal proof of the Sasaki-Murao algorithm [8]. This is an elegant algorithm 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 [1]. We believe that the proof we shall present, which was obtained by formalizing this algorithm in Type Theory (more precisely in the SSReflect [5] extension to Coq [9]) is simpler. It does not rely on Sylvester identities and indeed gives a proof of some of them as corollaries. It provides also a good example of how one can use a library of formalized mathematical results to prove formally a computer algebra program. The whole formalization can be found at [7].

2 Sasaki-Murao algorithm

2.1 Matrices

For any $n \in \mathbb{N}$, we define $I_n = \{i \in \mathbb{N} \mid i < n\}$ (with $I_0 = \emptyset$). If R is a set, a $m \times n$ matrix of elements of the set R is a function $I_m \times I_n \to R$. We can also view any such matrix 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 $p \times q$ sub-matrix[1] $M(f, g)$ by

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

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

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

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

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

If R is a ring, let 1_n be the $n \times n$ identity matrix. We can also define addition and multiplication of matrices as usual. We can decompose a non-empty $m \times n$ matrix M 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 $N_{ij} = m_{(1+i,1+j)}$

$$\left(\begin{array}{c|c} m_{00} & L \\ \hline C & N \end{array}\right)$$

With this decomposition, we define the central operation of our algorithm, which defines a $(m-1) \times (n-1)$ matrix:

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

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, for any map $f : I_p \to I_{m-1}$ and any map $g : I_q \to I_{n-1}$, we have the following identity:*

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

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

1. Selecting a sub-matrix commutes with most of the basic operations about matrices. In particular, $(M-N)(f,g) = M(f,g) - N(f,g)$, $(aM)(f,g) = aM(f,g)$. 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 $(r\ L\ C\ N)$, we have that $M(f^+, g^+)$ is the block $(r\ L(id,g)\ C(f,id)\ N(f,g))$

From this two observations, we then have:

$$\begin{aligned} M'(f,g) &= (rN - Cl)(f,g) \\ &= rN(f,g) - C(f,id)L(id,g) \\ M(f^+,g^+)' &= rN(f,g) - C(f,id)L(id,g) \end{aligned}$$

So, 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 notation):

$$Mat\ R\ ::=\ Empty \mid Mat\ R\ [R]\ [R]\ (Mat\ R)$$

So a matrix M is either the empty matrix $Empty$ or a compound matrix $Mat\ m\ L\ C\ N$. 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 we write $|M|$ 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.

2.2 The algorithm

We present 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 $Mat\ R[X] \to R[X] \to Mat\ R[X]$. We define then an auxiliary function $\phi\ a\ M$ of type $R[X] \to Mat\ R[X] \to R[X]$. The definition is:

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

From now on, we assume R to be a commutative ring.

The 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 remarkably simple (and polynomial time [1]) algorithm for computing the characteristic polynomial $\chi_M(X)$ of a matrix M. The determinant of M is then $\chi_{-M}(0)$.

3 Correctness proof

We first start to prove some auxiliary lemmas:

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

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

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

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

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 $(m_{00}\ L\ C\ N)$, and then the matrix N_2 is the block matrix $(m_{00}\ L\ 0\ M')$. 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
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 Sylvester equalities 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.

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
    | 0,_ => 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 SSReflect library to formalize the proof because it already contains many results that we need. The main scheme is to translate this program using SSReflect data types, prove its correctness and then prove that both implementations output the same results on valid inputs following the methodology presented in [3].

First, here is a description of the SSReflect data types we need:

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

Variable R : ringType.

(* 'M[R]_(m,n) a.k.a. 'M_(m,n) *)
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 sub-matrix $M(f, g)$ 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 SSReflect notations and types, we can now write the steps of the functional program (where `rdivp` is the pseudo-division operation [6] 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 than `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 CoqEAL [2], a library built on top of SSReflect libraries that we are currently developing. It allows to mirror

the main algebraic hierarchy of SSReflect with more concrete data types (e.g. here we mirror the matrix type `'M[R]_(m,n)` by the concrete type `Matrix CR`, assuming `CR` mirrors `R`) in order to prove the correctness of functional programs using the whole power of SSReflect libraries.

This process is done in the same manner as in [4] 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 SSReflect 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 SSReflect matrix into a `Matrix` always lead to a "valid" `Matrix`, and there is a bijection between SSReflect 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).
```

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 [3]. 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.

We have tested the implementation on 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 slow-down 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.

References

[1] J. Abdeljaoued and H. Lombardi. *Méthodes matricielles - Introduction à la complexité algébrique*. Springer, 2004.

[2] M. Dénès, A. Mörtberg, and V. Siles. CoqEAL, the Coq Effective Algebra Library, 2012. `http://www-sop.inria.fr/members/Maxime.Denes/coqeal`.

[3] M. Dénès, A. Mörtberg, and V. Siles. A refinement based approach to computational algebra in Coq. In *Interactive Theorem Proving*, volume 7406 of *LNCS*, pages 83–98, 2012.

[4] 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 *LNCS*, pages 327–342, 2009.

[5] G. Gonthier and A. Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical report, Microsoft Research INRIA, 2009. `http://hal.inria.fr/inria-00258384`.

[6] D. E. Knuth. *The art of computer programming, volume 2: seminumerical algorithms*. Addison-Wesley, 1981.

[7] A. Mörtberg and V. Siles. Formalization of Bareiss/Sasaki-Murao algorithm, 2012. `http://www.cse.chalmers.se/~siles/coq/formalisation.html`.

[8] T. Sasaki and H. Murao. Efficient Gaussian Elimination Method for Symbolic Determinants and Linear Systems. *ACM Trans. Math. Softw.*, 8(3):277–289, Sept. 1982.

[9] C. D. Team. The Coq Proof Assistant Reference Manual, version 8.3. Technical report, 2010.

Chapter 4

Paper 3: Coherent and Strongly Discrete Rings in Type Theory

Coherent and Strongly Discrete Rings in Type Theory*

Thierry Coquand, Anders Mörtberg, and Vincent Siles

Department of Computer Science and Engineering
University of Gothenburg, Sweden
{coquand,mortberg,siles}@chalmers.se

Abstract. We present a formalization of coherent and strongly discrete rings in type theory. This is a fundamental structure 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

1 Introduction

One of the fundamental operations in linear algebra is the ability to solve linear systems of equations. The concept of (strongly discrete) coherent rings abstracts over this ability which makes them an important notion in constructive algebra [13]. This makes these rings suitable as a basis for developing computational homological algebra, that is, linear algebra over rings instead of fields [3].

Another reason that these rings are important in constructive algebra is that they generalize the notion of Noetherian rings[1]. Classically any Noetherian 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 [15]. 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

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

[1] Rings where all ideals are finitely generated.

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 [1], namely the ability to invert ideals (which is usually hidden in standard classical treatments of Dedekind domains). This property also has applications in control theory [17].

All of these notions have been formalized[2] using the SSReflect extension [10] to the Coq proof assistant [4]. This work can be seen as a generalization of the previous formalization of linear algebra in the SSReflect library [9].

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 [3] where homological algorithms (without formalized correctness proofs) are implemented based on a notion that they call *computable* rings [2] which in fact are the same as coherent strongly discrete rings. Another source of inspiration is the work of Lombardi and Quitté [12].

This paper is organized as follows: first the formalization of coherent rings is presented followed by strongly discrete rings. Next Prüfer domains are explained together with the proofs that they are both coherent and strongly discrete. This is followed by a section on how to implement a computational version of the SSReflect development. We end by a section on conclusions and further work.

2 Coherent rings

Given a ring R (in our setting commutative but it is possible to consider non-commutative rings as well [2]) 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

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

[2] Documentation and formalization can be found at:
`http://www.cse.chalmers.se/~mortberg/coherent/`

where each M_i is a $1 \times n$ matrix first compute L_1 such that $M_1 X = 0$ iff $X = LY_1$ for some Y_1. Next compute L_2 such that $M_2 L_1 Y_1 = 0$ iff $Y_1 = L_2 Y_2$. At the end we obtain $L_1, \ldots, L_k$ such that $M_1 X = \cdots = M_k X = 0$ iff X is of the form $L_1 \cdots L_k Y$ and so does $L_1 \cdots L_k$ provide 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 development, coherent rings have been implemented as in [8] using the `Canonical Structure` mechanism of Coq. In the SSReflect 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) = 'M_(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) : Type := 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 : 'cV[R]_(size_solve V), 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 return forall M : 'M_(m,n), 'M_(n,size_solveMxN 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 : 'cV_(size_solveMxN M), 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.

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 $\forall x.\ x \in I \wedge x \in J \to x \in I \cap I$.

A convenient way to express this in Coq/SSReflect is to use strongly discrete rings that is discussed in section 3. For now we just assume that we can find 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 also 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 return forall (M : 'rV_m), 'M_(m,size_int 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 : 'cV[R]_(size_int M), 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 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 just

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

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 zeroes instead. In section 5 we will discuss how to implement a more efficient algorithm without any padding that is more suitable for computation.

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 [5, 11]) 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:

```
CoInductive 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 : ringType) : Type := Mixin {
  member : forall n, R -> 'rV_n -> option 'cV_n;
  _ : forall n (x : R) (I : 'rV_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 $x = IJ$. The use of `CoInductive` has nothing to do with coinduction but it should be seen as a datatype without any recursion schemes on which one can do case-analysis, for more information see [10].

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:

```
Fixpoint subid m n : 'rV[R]_m -> 'rV[R]_n -> bool :=
  match m return 'rV[R]_m -> 'rV[R]_n -> bool with
    | S p => fun (I : 'rV[R]_(1 + _)) J =>
         member (I 0 0) J && subid (rsubmx I) J
    | _ => fun _ _ => true
  end.

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 D, I = J *m D) (I <= J)%IS.
```

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 `D` in `subidP`.

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:

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

Now 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 otherwise get W_1 such that $R_1 W_1 = a_1$. Now 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 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_(m,n) -> 'cV_m -> option 'cV_n :=
  match m return 'M[R]_(m,n) -> 'cV[R]_m -> option 'cV[R]_n 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.

CoInductive 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: forall 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.

3.3 Bézout domains are strongly discrete and coherent

The first example of coherent strongly discrete rings that we studied were Bézout domains with explicit divisibility. These are integral domains where every finitely generated ideal is principal (generated by a single element). The two main examples of Bézout domains are $\mathbb{Z}$ 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:

```
CoInductive 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 : Type := 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[3] 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 a (this a is the greatest common divisor of all the a_i) such that $(a_1, \ldots, a_n) \subseteq (a)$ and $(a) \subseteq (a_1, \ldots, a_n)$.

To test if $x \in (a_1, \ldots, a_n)$ in a Bézout domain first compute a principal ideal (a) and then test if $a \mid x$ and if this is the case we we can construct the witness and otherwise we know that $a \notin (a_1, \ldots, a_n)$. 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).
```

[3] 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 $a = bu$.

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 $(a) \subseteq I$.

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 $I \cap J = (lcm(a, b))$, where $lcm(a, b)$ 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.

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. The one we choose here is the one in [12] that says that a Prüfer domain is an integral domains where given any x and y there exists u, v and w such that

$$ux = vy$$

and

$$(1 - u)y = wx$$

This is implemented in Coq by:

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

As we require that Prüfer domains have explicit divisibility, see beginning of section 3, it is possible for us to prove that they are strongly discrete which in turn 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.

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 give examples from algebraic geometry like $k[x, y]/(y^2 + x^4 - 1)$ and algebraic number theory like $\mathbb{Z}[\sqrt{-5}]$.

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 [6]. 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

$$\forall ijl.\ a_{lj}x_i = a_{li}x_j$$

In SSReflect 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 [6] and [12]. 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 $M = (a_{ij})$ is 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_n -> 'rV[R]_n -> 'rV[R]_n :=
  match n return 'I_n -> 'rV[R]_n -> 'rV[R]_n with
  | S p => fun (i : 'I_(1 + p)%N) (I : 'rV[R]_(1 + p)%N) =>
    (col i (plm I))^T
  | _ => fun _ _ => 0
  end.
```

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

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 decided 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 return 'rV[R]_n -> option 'cV[R]_n with
  | S p => fun I : 'rV[R]_p.+1 =>
      let: loc := plm I in
        if forallb i, I 0 i %| loc i i * x then
          Some (\col_i odflt 0 (loc 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 `forallb` is a finite forall testing that all of the elements of I divides $a_{ii}x$. Hence our implementation of Prüfer domains is strongly discrete which means that the theory about ideals developed for strongly discrete rings can be used when proving that Prüfer domains are coherent.

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 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)%IS.
```

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)%IS.

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

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 strongly discrete rings so not only can we solve homogeneous systems over them but also any linear system of equations.

4.3 Examples of Prüfer domains

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

```
Definition bezout_calc (x y: R) : (R * R * R)%type :=
  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 proof that $\mathbb{Z}[\sqrt{-5}]$ and $k[x, y]/(y^2 - 1 + x^4)$ are Prüfer domains but we have implemented these algorithms in Haskell [14].

5 Computations

In the paper algorithms are presented on structures using rich dependently typed datatypes which is convenient when proving properties but for computation this is not necessary. In fact it can be more efficient to implement the algorithms

on simply typed datatypes instead, a good example is matrices: As explained in section 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. So instead we use a library we previously developed 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. The methodology that we follow is summarized in [7] as:

1. Implement an abstract version of the algorithm using SSReflect's structures and use the libraries to prove properties about them. Here we can use the full power of dependent types when proving correctness.
2. Refine this algorithm into an efficient one using SSReflect's structures and prove that it behaves like the abstract version.
3. Translate the SSReflect 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 zeroes 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 zeroes, this would then be proved to produce a set of 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) : Type := 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 and `seqmx_of_mx` is the translation function from SSReflect matrices to list based matrices. Using this more efficient versions of the algorithms presented above can be implemented simply by changing the functions on SSReflect 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 SSReflect 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, so it is done by expanding definitions and translating using already implemented 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 $(2, 3)$.

6 Conclusions and Further 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 datatypes, this way we get certified mathematical algorithms that are suitable for computation. Hence can this work be seen as an example that the methodology presented in [7] is applicable on more complicated structures as well.

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 [16, 18]. It would be interesting to reimplement this using SSReflect and compare the complexity of the formalizations.

It would also be interesting to use this work as a basis for 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 [2]. This formalization would involve first proving that the category of finitely presented modules over coherent strongly discrete rings form an abelian category and then use this to implement further algorithms.

A consequence of the choice of using SSReflect for the formalization is that it is difficult to formalize things 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 SSReflect library and 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.

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.

References

[1] J. Avigad. Methodology and metaphysics in the development of Dedekind's theory of ideals. The architecture of modern mathematics. Essays in history and philosophy. Oxford: Oxford University Press, 2006.

[2] M. Barakat and M. Lange-Hegermann. An axiomatic setup for algorithmic homological algebra and an alternative approach to localization. *J. Algebra Appl.*, 10(2):269–293, 2011.

[3] M. Barakat and D. Robertz. homalg – a meta-package for homological algebra. *J. Algebra Appl.*, 7(3):299–317, 2008.

[4] Coq development team. The Coq Proof Assistant Reference Manual, version 8.3. Technical report, 2010.

[5] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms: An introduction to Computational Algebraic Geometry and Commutative Algebra.* Springer, 2006.

[6] 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.

[7] M. Dénès, A. Mörtberg, and V. Siles. A refinement based approach to computational algebra in Coq. In *Interactive Theorem Proving*, volume 7406 of *LNCS*, pages 83–98, 2012.

[8] 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 *LNCS*, pages 327–342, 2009.

[9] G. Gonthier. Point-Free, Set-Free concrete linear algebra. In *Interactive Theorem Proving*, volume 6898 of *LNCS*, pages 103–118, 2011.

[10] G. Gonthier and A. Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical report, Microsoft Research INRIA, 2009.

[11] H. Lombardi and H. Perdry. The Buchberger Algorithm as a Tool for Ideal Theory of Polynomial Rings in Constructive Mathematics. 1998.

[12] H. Lombardi and C. Quitté. *Algèbre commutative, Méthodes constructives: Modules projectifs de type fini.* Calvage et Mounet, 2011.

[13] R. Mines, F. Richman, and W. Ruitenburg. *A Course in Constructive Algebra.* Springer-Verlag, 1988.

[14] A. Mörtberg. Constructive Algebra in Functional Programming and Type Theory. Master's thesis, Chalmers University of Technology, 2010.

[15] H. Perdry and P. Schuster. Noetherian orders. *Mathematical. Structures in Comp. Sci.*, 21(1):111–124.

[16] H. Persson. An Integrated Development of Buchberger's Algorithm in Coq. 2001.

[17] A. Quadrat. The fractional representation approach to synthesis problems: An algebraic analysis viewpoint part ii: Internal stabilization. *SIAM J. Control Optim.*, 42(1):300–320, Jan. 2003.

[18] 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, London, UK, 1998. Springer-Verlag.

Chapter 5

Paper 4: Towards a Certified Computation of Homology Groups for Digital Images

Towards a Certified Computation of Homology Groups for Digital Images*

Jónathan Heras[1], Maxime Dénès[2], Gadea Mata[1], Anders Mörtberg[3], María Poza[1], and Vincent Siles[3]

[1] University of La Rioja, Spain
[2] INRIA Sophia Antipolis – Méditerranée, France
[3] University of Gothenburg, Sweden
{jonathan.heras,gadea.mata,maria.poza}@unirioja.es,
Maxime.Denes@inria.fr,
{mortberg,siles}@chalmers.se

Abstract. In this paper we report on a project to obtain a verified computation of homology groups of digital images. The methodology is based on programming and executing *inside* the Coq proof assistant. Though more research is needed to integrate and make efficient more processing tools, we present some examples partially computed in Coq from real biomedical images.

1 Introduction

The discipline of Algebraic Digital Topology, or more specifically, the computation of homology groups from digital images is mature enough (see, for instance, [27], one among many good references) to go one step further and investigate the possibility of a *certified computation* (i.e., formally verified by proving correctness using an *interactive* proof assistant) in digital topology, as it happens in other areas of computer mathematics (see [8]).

In a very rough manner, the process to be verified is reflected in Figure 1. Putting it into words, from the black pixels of a monochromatic image a simplicial complex is obtained (by means of a triangulation procedure); subsequently, from the simplicial complex, its *boundary (or incidence) matrices* are constructed, and finally, *homology* can be computed. If we work with coefficients over a field (and it is well-known that it is enough to take as coefficients the field $\mathbb{Z}/2\mathbb{Z}$, when we work with 2D and 3D digital images) and if only the *dimensions* of the homology groups (as vector spaces) are looked for, then having a program able to compute the rank of a matrix is sufficient to accomplish the whole task.

This architecture is particularized in this paper with a real problem that appeared in an industrial application and with the Coq proof assistant as programming and verifying tool.

* Partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01, and by the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

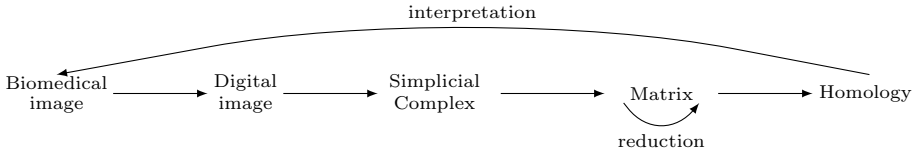


Fig. 1. Computing homology from a digital image

The rest of this paper is organized as follows. Section 2 is devoted to present an example, coming from the biomedical context, as a test-case for our formal development. The formalization process is explained in Section 3, focusing on the link between boundary matrices and homology groups. Section 4 explains how the certified programs can be used to effectively compute homology of images. A way to deal with the management of the huge matrices produced by biomedical images is presented in Section 5. The paper ends with a section of Conclusions and Further work, and the bibliography.

2 Motivation

When developing formal proofs, a major issue is ensuring that concepts are defined in a way that will be applicable to concrete use. In our case, we are developing a general theory of effective simplicial homology as part of the Formath project [1]. We decided to validate our design choices on biomedical digital images obtained from synaptical structures.

Synapses are the points of connection between neurons. The relevance of synapses comes from the fact that they are related to the computational capabilities of the brain.

The possibility of changing the number of synapses may be an important asset in the treatment of neurological diseases, such as Alzheimer, see [26]. Therefore, we can claim that an efficient, reliable and automatic method for counting synapses is instrumental in the study of the evolution of synapses in scientific experiments.

Up to now, the method to count synapses was manual, see [6]. This was impractical since it implies a considerable time investment. In order to improve this process, a plug-in called SynapCountJ [17] for the ImageJ environment [22] has been developed.

The procedure implemented in this software to handle neuron images can be split into two steps. First, taking as input three images of a neuron, namely the neuron with two different antibody markers and the structure of the neuron, SynapCountJ produces a bitmap where synapses are the connected components, see Figure 2. Then the second step consists in counting the connected components of the bitmap. A detailed explanation of the procedure was given in [13].

To test the suitability of this program, biologists consider, on the one hand, control cultures and, on the other hand, cultures under the effect of some drugs;

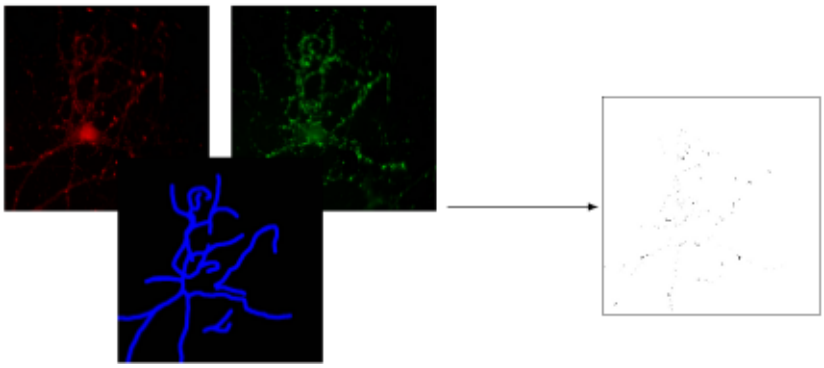


Fig. 2. Example of the results produced by SynapCountJ

in this way, the evolution of the density of the occurrence of synapses under the effect of those drugs can be determined. For instance, using the chemical inhibitor GSK3, the evolution percentage manually obtained is 36% and the one obtained with SynapCountJ is 36.6%. Thus, the experimental results obtained with SynapCountJ were considered (by the biologists) very satisfactory.

The former step of the procedure implemented in SynapCountJ, the extraction of a bitmap with the synapses from three images of the neurons, is carried out based on solid previous experience of experimental scientists; therefore, they consider it as a safe process. The latter step, the computation of connected components, can be solved with many algorithms and is an interesting test case for our framework where we can compute the homology in dimension 0 of such images. This is a well known procedure to measure the amount of connected components of an image, even if more elementary methods are also applicable.

3 Verification in Coq/SSReflect

In the introduction we have explained a method, based on simplicial homology, to study the homology of a digital image which consists of: (1) building a simplicial complex from the image, (2) generating the boundary matrices associated with the simplicial complex, and (3) computing the homology from the boundary matrices.

The correctness of the programs in charge of both the construction of a simplicial complex from an image and the generation of the boundary matrices associated with a simplicial complex have been formally proved using proof assistant tools as can be seen in [21] and [14] respectively. Then, there only remains the verification of the third point, the computation of homology groups from the boundary matrices.

In our formalization, we have used the Coq proof assistant [5]. This system provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive de-

velopment of machine-checked proofs. In addition, we take advantage of the features included in SSReflect [9], an extension for Coq whose development was started by G. Gonthier during the formal proof of the *Four Color Theorem* [8]. The SSReflect libraries include enough ingredients to undertake the task of defining and computing homology from matrices. Some details of the proofs will be omitted; the interested reader can consult the original and complete source code at `http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ProofExamples`.

First of all, we define the notion of homology in Coq. Let K be a field, $V1, V2, V3$ vector spaces on K, and $f : V1 \to V2, g : V2 \to V3$ linear applications; then, the *Homology of* f, g is the quotient between the *kernel* of g and the *image* of f. This is translated into Coq in the following way.

```
Variable (K : fieldType) (V1 V2 V3 : vectType K)
         (f : linearApp V1 V2) (g : linearApp V2 V3).
Definition Homology := ((lker g) :\: (limg f)).
```

Nevertheless, we do not usually work with linear applications when trying to compute homology but with the matrices representing those linear applications. In particular, as we are working on a field K, given two matrices with coefficients in this field, let us called them, mxf and mxg of sizes $v1 \times v2$ and $v2 \times v3$ respectively and such that their product is the null matrix, the dimension of the corresponding homology vector space is given by the formula: $v2 - rank(mxg) - rank(mxf)$. This definition is introduced in Coq as follows.

```
Definition dim_homology (mxf:'M[K]_(v1,v2)) (mxg:'M[K]_(v2,v3)) :=
  v2 - \rank mxg - \rank mxf.
```

Now, the correctness of `dim_homology` can be shown by proving that given two matrices `mxf` and `mxg` whose product is the null matrix (`mxf *m mxg = 0`), then the result obtained using `dim_homology` is the dimension of the homology group associated with the linear applications defined from `mxf` and `mxg` (`(LinearApp mxf)` and `(LinearApp mxg)`).

```
Lemma dimHomologyrankE: mxf *m mxg = 0 ->
      \dim Homology (LinearApp mxf) (LinearApp mxg) =
      dim_homology mxf mxg.
```

However the use of SSReflect libraries may trigger heavy computations during deduction steps, that would not terminate within a reasonable amount of time. To handle this issue, some definitions like matrices are locked in a way that do not allow direct computations.

To overcome this pitfall, we use the matrix representation and the rank algorithm developed in [4] to define `ex_homology` which takes as argument two such matrices (represented by means of lists of lists) `mxf` and `mxg` which dimensions are `v1`×`v2` and `v2`×`v3` respectively, and computes the homology.

```
Definition ex_homology (v1 v2 v3:nat) (mxf mxg : seqMatrix K) :=
  v2 - (rank v2 v3 mxg) - (rank v1 v2 mxf).
```

Finally, we prove the correctness of `ex_homology` by showing its equivalence to `dim_homology` up to a change of representation (this domain transformation is given by `seqmx_of_mx`).

```
Lemma ex_homology_rankE (mxf: 'M[K]_(w1,w2)) (mxg : 'M[K]_(w2,w3))
     :
  ex_homology (seqmx_of_mx mxf) (seqmx_of_mx mxg)
  = dim_homology mxf mxg.
```

Then, we have an executable program to compute homology, for any dimension, whose correctness has been verified in Coq; therefore, we can claim that its results will always be correct.

4 Computing homology with Coq

An example is presented in this section in order to clarify how we can compute homology groups in Coq. Let us consider the simplicial complex of the left side of Figure 3. If we impose a lexicographical order on the simplices of the same dimension of this simplicial complex, its boundary matrix in dimension 1 is the one presented in the right side of Figure 3; it is worth noting that the rest of boundary matrices are empty, in particular we do not consider the empty set as an element of dimension -1.

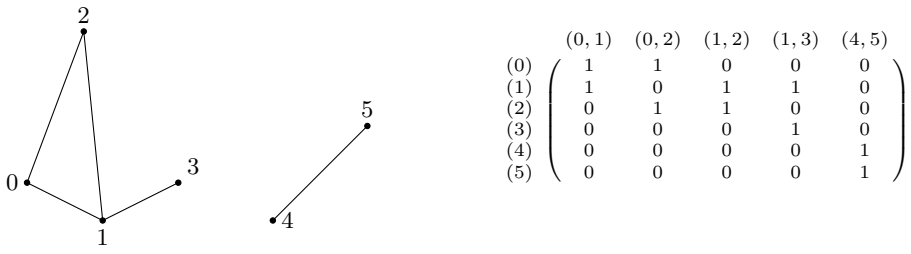


Fig. 3. Simplicial complex and its boundary matrix

The procedure to compute the homology (note that it only makes sense to compute homology in dimensions 0 and 1) of the simplicial complex of Figure 3 is as follows. Firstly, we define the boundary matrices.

```
Definition d0_ex1 := [::].
Definition d1_ex1 := [::[::1;1;0;0;0];
                        [::1;0;1;1;0];
                        [::0;1;1;0;0];
                        [::0;0;0;1;0];
                        [::0;0;0;0;1];
                        [::0;0;0;0;1]].
Definition d2_ex1 := [::].
```

Eventually, we can compute the homology using the following instructions.

```
Eval vm_compute in (ex_homology 0 6 5 d0_ex1 d1_ex1).
Eval vm_compute in (ex_homology 6 5 0 d1_ex1 d2_ex1).
```

obtaining `2` and `1` respectively. In the same way, we could compute homology from the boundary matrices associated with the simplicial complex generated from a digital image. However, if we try to compute the homology from the images produced by SynapCountJ (see Figure 2), Coq is not able to handle those images yet, due to the size of data involved.

It is worth noting that Coq is a Proof Assistant and not a Computer Algebra system. Efficient implementations of mathematical algorithms running inside Coq is an ongoing effort, as shown by recent works on efficient real numbers [16], machine integers and arrays [2] or a previous approach to compiled execution of internal computations [10].

We devise a couple of ways to achieve better efficiency:

- Improve the runtime system using the extraction mechanism which translates Coq code to a functional programming language like OCaml or Haskell. However, this would not allow us to reuse the result of our homological computations for further proofs. Indeed, output of external programs are untrusted so they cannot be imported. Instead, we are using a recent intermediate approach consisting in internally compiling Coq terms to OCaml with performance comparable to extracted code [18].
- Optimize algorithms and representations using sparse matrices, which is well suited to simplicial complexes obtained from digital images. We have developed an Haskell implementation of such an algorithm but we still need to formally verify its correctness.

In the next section we describe another method to overcome the efficiency drawback, based on reducing the size of matrices while keeping the same homological information.

5 Computing discrete vector fields

The method that we are using for the reduction process is based on *Discrete Morse Theory* [7]; namely, we work in the algebraic setting of this theory which was described in [25]. Roughly speaking, the aim of Discrete Morse Theory consists of finding *simplicial collapses* which transform a simplicial complex $\mathcal{K}$ into a smaller one but keeping its homological properties. In this context, the instrumental tool are *admissible discrete vector fields* which allows one to reduce the amount of information removing "useless" information but keeping the homological properties of the original object.

The use of these techniques from Discrete Morse Theory has been welcomed in the study of homological properties of digital images, see [3, 11, 15], for instance. This is due to the fact that the size of the cellular object associated with

an image can be huge, but the choice of an appropriate vector field can produce a much smaller object.

So, the question now is given a cellular complex how we can produce a vector field as large as possible (the larger the vector field, the smaller the reduced object). Several approaches to solve this problem have been studied as can be seen in [24, 12, 23, 19], the strategy that we have chosen was explained in [25]. It is not the aim of this paper to describe that algorithm (from now on, called RS's algorithm; RS stands for Romero–Sergeraert); but, we just introduce some ideas. This algorithm takes as input one of the boundary matrices associated with the cellular complex and provides an admissible discrete vector field (subsequently, from the matrix and the vector field a reduced matrix can be obtained).

The algorithm has been implemented in Haskell; and, some remarkable results have been obtained in the reduction process. As benchmark to test our programs, we have considered matrices coming from, on the one hand, 500 randomly generated images; and, on the other hand, biomedical images. In the former case, the size of the matrices was initially around 100×300, and after the reduction process the average size was 5×50. Using the original matrices Coq takes around 12 seconds to compute their rank; on the contrary, using the reduced matrices Coq only needs milliseconds. In the latter case, the matrices coming from biomedical images, the size of matrices is reduced from around 690×1400 to 97×500. In this case, Coq cannot deal with the original matrices; on the contrary, it is able to handle matrices as the ones obtained after applying the reduction programs and compute the results in, approximately, 25 seconds.

As a final remark, let us explain the main reason for using Haskell to implement the RS algorithm. The use of this language is due to the fact that Haskell is quite close to Coq; and, therefore, algorithms implemented in Haskell can be verified using Coq, a question which is, as we have seen, instrumental in our developments. In particular, the formalization of the correctness of the algorithm in charge of constructing an admissible discrete vector field given a matrix is ongoing work; and, up to now, we have certified that our programs build a discrete vector field. The proof of the admissibility property remains as further work.

6 Conclusions and further work

In this paper, we have presented how we can use Algebraic Topology techniques to study biomedical images in a reliable manner. The first step consists in processing the biomedical images to obtain an image where homological information is as explicit as possible. Subsequently, using programs whose correctness has been verified in the Coq/SSReflect proof assistant, homological properties from the pre-processed image are obtained, which in turn are interpreted as features of the original image.

This methodology has been applied in this paper to the problem of determining the number of synapses of a neuron. In this case, the problem is reduced to measure the number of connected components of a monochromatic image. An

issue which can be solved, even if it is not the straightforward manner, thanks to the computation of the homology group in dimension 0 of the image.

The use of certified tools able to compute homology groups will be important in the future; for instance, to recognize the structure of a neuron; a problem which seems to involve the homology group in dimension 1, see [20]. Other techniques, like the ones of persistent homology, could be applied in stacks of neurons to remove the noise of the images and help to the detection of the dendrites (the branches of the neuron).

Some formalization aspects also remain as future work. We have already mentioned the on-going work around proving the correctness of the admissible discrete vector fields programs. Moreover, certifying the correctness of integer homology computation is also further work (some results about the formalization of the Smith Normal Form are already encoded in Coq, see [4]).

As we previously mentioned, we are still working on efficiency issues but switching to better representations and more efficient algorithms will not require to redo the proofs related to homology.

References

[1] ForMath: formalisation of mathematics. `http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ForMath`.

[2] M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending coq with imperative features and its application to SAT verification. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving*, volume 6172, pages 83–98. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[3] F. Cazals, F. Chazal, and T. Lewiner. Molecular shape analysis based upon Morse-Smale complex and the Connolly function. In *Proceedings 19th ACM Symposium on Computational Geometry (SCG'03)*, pages 351–360, 2003.

[4] C. Cohen, M. Dénès, A. Mörtberg, and V. Siles. Smith Normal form and executable rank for matrices. `http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ProofExamples`.

[5] Coq development team. The Coq Proof Assistant Reference Manual, version 8.3. Technical report, 2010.

[6] G. Cuesto et al. Phosphoinositide-3-Kinase Activation Controls Synaptogenesis and Spinogenesis in Hippocampal Neurons. *The Journal of Neuroscience*, 31(8):2721–2733, 2011.

[7] R. Forman. Morse theory for cell complexes. *Advances in Mathematics*, 134:90–145, 1998.

[8] G. Gonthier. *Formal proof - The Four-Color Theorem*, volume 55. Notices of the American Mathematical Society, 2008.

[9] G. Gonthier and A. Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical report, Microsoft Research INRIA, 2009. `http://hal.inria.fr/inria-00258384`.

[10] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proceedings of the seventh ACM SIGPLAN international conference on*

Functional programming, ICFP '02, page 235–246, New York, NY, USA, 2002. ACM.

[11] A. Gyulassy, P. Bremer, B. Hamann, and V. Pascucci. A practical approach to Morse-Smale complex computation: Scalability and generality. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1619–1626, 2008.

[12] S. Harker et al. The Efficiency of a Homology Algorithm based on Discrete Morse Theory and Coreductions. In *Proceedings 3rd International Workshop on Computational Topology in Image Context (CTIC'2010)*, volume 1 of *Image A*, pages 41–47, 2010.

[13] J. Heras, G. Mata, M. Poza, and J. Rubio. Homological processing of biomedical digital images: automation and certification. Technical report, 2010. `http://wiki.portal.chalmers.se/cse/uploads/ForMath/hpbdiac`.

[14] J. Heras, M. Poza, M. Dénès, and L. Rideau. Incidence simplicial matrices formalized in Coq/SSReflect. In *Proceedings 18th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (Calculemus'2011)*, volume 6824 of *Lectures Notes in Computer Science*, pages 30–44, 2011.

[15] G. Jerse and N. M. Kosta. Tracking features in image sequences using discrete Morse functions. In *Proceedings 3rd International Workshop on Computational Topology in Image Context (CTIC'2010)*, volume 1 of *Image A*, pages 27–32, 2010.

[16] R. Krebbers and B. Spitters. Computer certified efficient exact reals in coq. In J. H. Davenport, W. M. Farmer, J. Urban, and F. Rabe, editors, *Intelligent Computer Mathematics*, volume 6824, pages 90–106. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[17] G. Mata. SynapsCountJ. University of La Rioja, 2011. `http://imagejdocu.tudor.lu/doku.php?id=plugin:utilities:synapsescountj:start`.

[18] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, volume 7086, pages 362–377. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[19] H. Molina-Abril and P. Real. A homological-based description of subdivided nD objects. In *Proceedings of the 14th international conference on Computer analysis of images and patterns (CAIP'2011)*, volume 6854 of *Lectures Notes in Computer Science*, pages 42–50, 2011.

[20] M. Mrozek et al. Homological methods for extraction and analysis of linear features in multidimensional images. *Pattern Recognition*, 45(1):285–298, 2012.

[21] F. L. R. node. From Digital Images to Simplicial Complexes: A report. Technical report, 2011. `http://wiki.portal.chalmers.se/cse/uploads/ForMath/fditscr`.

[22] W. S. Rasband. ImageJ: Image Processing and Analysis in Java, 2003. `http://rsb.info.nih.gov/ij/`.

[23] P. Real and H. Molina-Abril. Towards Optimality in Discrete Morse Theory through Chain Homotopies. In *Proceedings 3rd International Workshop on Computational Topology in Image Context (CTIC'2010)*, volume 1 of *Image A*, pages 33–40, 2010.

[24] V. Robins, P. Wood, and A. Sheppard. Theory and algorithms for constructing discrete Morse complexes from grayscale digital images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(8):1646 – 1658, 2011.

[25] A. Romero and F. Sergeraert. Discrete Vector Fields and Fundamental Algebraic Topology, 2010. `http://arxiv.org/abs/1005.5685v1`.

[26] D. J. Selkoe. Alzheimer's disease is a synaptic failure. *Science*, 298(5594):789–791, 2002.

[27] D. Ziou and M. Allili. Generating Cubical Complexes from Image Data and Computation of the Euler number. *Pattern Recognition*, 35:2833–2839, 2002.